26th International Conference on Database Theory

ICDT 2023, March 28–31, 2023, Ioannina, Greece

Edited by

Floris Geerts
Brecht Vandevoort

LIPICS

Editors

Floris Geerts
University of Antwerp, Belgium
floris.geerts@uantwerp.be

Brecht Vandevoort
UHasselt, Data Science Institute, ACSL, Diepenbeek, Belgium
brecht.vandevoort@uhasselt.be

ACM Classification 2012
Information systems → Data management systems; Information systems → Data streams; Information systems → Data structures; Information systems → Database views; Information systems → Graph-based database models; Information systems → Join algorithms; Information systems → Query languages for non-relational engines; Information systems → Query languages; Information systems → Query optimization; Information systems → Query planning; Information systems → Relational database query languages; Information systems → Structured Query Language; Theory of computation → Approximation algorithms analysis; Theory of computation → Data modeling; Theory of computation → Data provenance; Theory of computation → Data structures and algorithms for data management; Theory of computation → Data structures design and analysis; Theory of computation → Database query languages (principles); Theory of computation → Database query processing and optimization (theory); Theory of computation → Database theory; Theory of computation → Description logics; Theory of computation → Graph algorithms analysis; Theory of computation → Incomplete, inconsistent, and uncertain databases; Theory of computation → Logic and databases; Theory of computation → Parameterized complexity and exact algorithms; Theory of computation → Shared memory algorithms; Theory of computation → Sorting and searching; Theory of computation → Streaming, sublinear and near linear time algorithms; Mathematics of computing → Graph algorithms; Mathematics of computing → Graph theory; Computing methodologies → Network science; Networks → Network structure

ISBN 978-3-95977-270-9

Published online and open access by
Schloss Dagstuhl – Leibniz-Zentrum für Informatik GmbH, Dagstuhl Publishing, Saarbrücken/Wadern, Germany. Online available at https://www.dagstuhl.de/dagpub/978-3-95977-270-9.

Publication date
March, 2023

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 https://portal.dnb.de.

Digital Object Identifier: 10.4230/LIPIcs.ICDT.2023.0
ISBN 978-3-95977-270-9 **ISSN 1868-8969** **https://www.dagstuhl.de/lipics**

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

https://www.dagstuhl.de/lipics

Contents

26th International Conference on Database Theory (ICDT 2023).
Editors: Floris Geerts and Brecht Vandevoort
LIPICS Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Preface

The 26th International Conference on Database Theory (ICDT 2023) was held in Ioannina, Greece, from March 28 to March 31, 2021.

The Program Committee has selected 21 research papers out of 49 submissions for publication at the conference. It has further decided to give the Best Paper Award to *The I/O Complexity of Enumerating Subgraphs of Constant Sizes* by Shiyuan Deng, Francesco Silvestri and Yufei Tao, and the Best Newcomer Paper Award to *An Optimal Algorithm for Sliding Window Order Statistics* by Pavel Raykov. We congratulate the winners!

Apart from the 21 regular papers, these proceedings include invited papers associated with the (shared) EDBT/ICDT keynotes by Leonid Libkin (University of Edinburgh & ENS Paris) and Gonzalo Navarro (University of Chile), as well as the invited paper associated with the ICDT invited tutorial by Seshadhri Comandur (University of California).

A committee formed by Wang-Chiew Tan, Diego Figueira, and George Fletcher has decided to give the Test-of-Time Award for ICDT 2023 to the two ICDT 2013 papers *A Theory of Pricing Private Data* by Chao Li, Daniel Y. Li, Gerome Miklau and Dan Suciu, and *Querying Graph Databases with XPath* by Leonid Libkin, Wim Martens and Domagoj Vrgoč.

We would like to thank all people who contributed to the success of ICDT 2023, including the authors of all submitted papers, keynote and invited tutorial speakers, and, of course, all members of the Program Committee as well as the external reviewers, for the very substantial work that they have invested over the two submission cycles of ICDT 2023. Their commitment and sagacity were crucial to ensure that the final program of the conference satisfies the highest standards. We would also like to thank the ICDT Council members for their support on a wide variety of matters, and the local organizers of the EDBT/ICDT 2023 conference, led by General Chairs Nikos Mamoulis and Evaggelia Pitoura, for the great job they did in organizing the conference and co-located events. Finally, we wish to acknowledge Dagstuhl Publishing for their support with the publication of the proceedings in the LIPIcs (Leibniz International Proceedings in Informatics) series.

Floris Geerts and Brecht Vandevoort
March 2023

Organization

General Chairs

Nikos Mamoulis (University of Ioannina)

Evaggelia Pitoura (University of Ioannina)

Program Chair

Floris Geerts (University of Antwerp)

Program Committee

Sepehr Assadi (Rutgers University)

Vaishak Belle (The University of Edinburgh)

Leopoldo Bertossi (SKEMA Business School, Montreal, Canada)

Graham Cormode (The University of Warwick)

Ahmet Kara (University of Zurich)

Batya Kenig (Technion, Israel Institute of Technology)

Bas Ketsman (Vrije Universiteit Brussel)

Ester Livshits (The University of Edinburgh)

Wim Martens (University of Bayreuth)

Liat Peterfreund (CNRS, Université Gustave Eiffel)

Reinhard Pichler (Vienna University of Technology)

Andreas Pieris (University of Cyprus, The University of Edinburgh)

Marcin Przybyłko (University of Leipzig)

Juan L. Reutter (Pontificia Universidad Católica)

Sudeepa Roy (Duke University)

Jef Wijsen (University of Mons)

Ke Yi (Hong Kong University of Science and Technology)

Thomas Zeume (Ruhr-Universität Bochum)

Proceedings Chair

Brecht Vandevoort (Hasselt University)

External Reviewers

Marco Calautti

Xiao Hu

Stefan Mengel

Matthias Niewerth

Anantha Padmanabha

Marko Schmellenkamp

Nils Vortmeier

Katja Zeume

Contributing Authors

Antoine Amarilli
Aziz Amezian El Khalfioui
Sepehr Assadi
Magdalena Balazinska
Walter Cai
Seshadhri Comandur
Tamara Cucumides
Kyle Deeds
Shiyuan Deng
Thomas Feller
Diego Figueira
Nadime Francis
Amélie Gheerbrant
Amir Gilad
Paolo Guagliardo
Martin Grohe
Aviram Imber
Nirmit Joshi
Ahmet Kara
Jens Keppeler
Majd Khalil
Benny Kimelfeld
Leonid Libkin
Peter Lindner
Shangqi Lu
Timothy Lyon
Victor Marsault
Wim Martens
Jingfan Meng
Timo Merkl
Rémi Morvan
Martín Muñoz
Filip Murlak
Gonzalo Navarro
Milos Nikolic
Mitsunori Ogihara
Dan Olteanu
Piotr Ostropolski-Nalewaja
Anantha Padmanabha
Liat Peterfreund
Reinhard Pichler
Andreas Pieris
Milind Prabhu
Pavel Raykov
Juan L. Reutter
Cristian Riveros
Alexandra Rogova
Sebastian Rudolph
Jorge Salas
Thomas Schwentick
Luc Segoufin
Vihan Shah
Francesco Silvestri
Cristina Sirangelo
Oskar Skibski
Sebastian Skritek
Christopher Spinrath
Christoph Standke
Dan Suciu
Yufei Tao
Domagoj Vrgoč
Huayi Wang
Jef Wijsen
Jun Xu
Haozhe Zhang

The ICDT 2023 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 earlier ICDT conferences that have best met the "test of time". In 2023, the award recognizes two papers selected from the proceedings of the ICDT 2013 conference that have had the highest impact in terms of research, methodology, conceptual contribution, or transfer to practice over the past decade. The award was presented during the EDBT/ICDT 2023 Joint Conference, March 28 – 31, 2023.

The 2023 ToT Committee consists of Wang-Chiew Tan, Diego Figueira, and George Fletcher. After careful consideration and soliciting external assessments, the committee has chosen the following contributions for the 2023 ICDT Test-of-Time Award:

A Theory of Pricing Private Data
Chao Li, Daniel Y. Li, Gerome Miklau and Dan Suciu

This paper presents a theoretical framework for monetizing private data which empowers individuals to control their data through financial means. In this framework, data owners are financially compensated for their loss of privacy where lower prices are assigned to noisier query answers. This framework adopts and extends prior techniques on data pricing and differential privacy. It is the first time an end-to-end perspective on data pricing, combining the problems of pricing and revenue allocation, was provided. This paper has widespread influence on research on data pricing both within and beyond the database community.

Querying Graph Databases with XPath
Leonid Libkin, Wim Martens and Domagoj Vrgoč

This paper presents a graph language called GXPath (short for Graph XPath) that strikes an interesting balance between expressiveness and complexity and is influential in the Graph Query Language (GQL) standard. GXPath permits expressive queries that can be efficiently evaluated and has a strong influence on GQL as well as SQL/PGQ (for querying graph databases in SQL) which are currently being finalized in the same ISO committee that maintains the SQL Standard. This paper showcases how theoretical work can be directly influential in industry and academic community consensus building around the upcoming Graph Query Language (GQL) standard.

Wang-Chiew Tan
Facebook AI

Diego Figueira
Université de Bordeaux

George Fletcher
Eindhoven University of Technology (TU/e)

The ICDT Test-of-Time Award Committee for 2023

A Researcher's Digest of GQL

Nadime Francis
Laboratoire d'Informatique Gaspard Monge,
Université Gustave Eiffel, CNRS, France

Amélie Gheerbrant
IRIF, Université Paris Cité, CNRS,
Paris, France

Paolo Guagliardo
School of Informatics,
University of Edinburgh, UK

Leonid Libkin
University of Edinburgh, UK
RelationalAI, France
ENS, PSL University, France

Victor Marsault
Laboratoire d'Informatique Gaspard Monge,
Université Gustave Eiffel, CNRS, France

Wim Martens
Universität Bayreuth, Germany

Filip Murlak
University of Warsaw, Poland

Liat Peterfreund
Laboratoire d'Informatique Gaspard Monge,
Université Gustave Eiffel, CNRS, France

Alexandra Rogova
IRIF, Université Paris Cité, CNRS, Paris, France
Data Intelligence Institute of Paris, Inria

Domagoj Vrgoč
University of Zagreb, Coratia
Pontificia Universidad Católica de Chile,
Santiago, Chile

Abstract

GQL (Graph Query Language) is being developed as a new ISO standard for graph query languages to play the same role for graph databases as SQL plays for relational. In parallel, an extension of SQL for querying property graphs, SQL/PGQ, is added to the SQL standard; it shares the graph pattern matching functionality with GQL. Both standards (not yet published) are hard-to-understand specifications of hundreds of pages. The goal of this paper is to present a digest of the language that is easy for the research community to understand, and thus to initiate research on these future standards for querying graphs. The paper concentrates on pattern matching features shared by GQL and SQL/PGQ, as well as querying facilities of GQL.

2012 ACM Subject Classification Theory of computation → Database theory; Theory of computation → Database query languages (principles); Information systems → Graph-based database models; Information systems → Structured Query Language

Keywords and phrases GQL, Property Graph, Query Language, Graph Database, Pattern matching, Multi-Graph

Digital Object Identifier 10.4230/LIPIcs.ICDT.2023.1

Category Invited Talk

Funding This work is supported by: a Leverhulme Trust Research Fellowship; EPSRC grant S003800; French ANR projects grants ANR-18-CE40-0031 (*QUID*), ANR-19-CE48-0019 (*EQUUS*), and ANR-21-CE48-0015 (*Verigraph*); German Research Foundation (DFG) projects 431183758 and 369116833; ANID Millennium Science Initiative Program, Code ICN17_002; ANID Fondecyt Regular project 1221799; NCN grant 2018/30/E/ST6/00042.

Acknowledgements The authors are grateful to members of the ISO/IEC JTC1 SC32 WG3 committee and especially Fred Zemke for many comments on our formalization of the language.

1 Introduction

Graph databases have grown steadily in popularity this century. They handle data as it is viewed conceptually, making them easily applicable in many tasks where traditional relational databases are not easy or natural to use. While many early applications cited social networks and the Semantic Web as the key motivation (since in both cases data is naturally viewed as a graph), industry scale applications are much more diverse and include fraud detection, network management, medical data management, knowledge management, and even investigative journalism. There are several dozen graph database products on the market, including the current leader Neo4j, as well as both established and upcoming companies offering graph products (e.g., Oracle, Amazon, IBM, SAP, Redis, DataStax, TigerGraph, Memgraph, etc.).

Graph databases' widespread use happened without them having their lingua franca, which is the role that SQL is playing for relational databases. The landscape of graph languages – at least at first sight – is very varied. Neo4j has its own language called Cypher [18], which is also implemented in other products, including SAP HANA and Amazon Neptune. Oracle introduced its language PGQL [34]; TigerGraph has GSQL [13], and several products use the non-declarative graph traversal language Gremlin [33]. However, upon a closer examination, one discovers that declarative languages are more like different dialects of the same language rather than different languages altogether. This led to a proposal to define a new unifying standard for a Graph Query Language (GQL) [36]. The proposal was given a go-ahead in 2019, and since then was taken up by the same committee that produces and maintains the SQL Standard. It is known as ISO/IEC JTC1 SC32 WG3 within the International Organization for Standardization, or ISO.

In fact, this committee develops two projects in parallel:

- SQL/PGQ, a new Part 16 of the SQL Standard, that defines querying graphs specified as views over a relational schema; it is expected to be published roughly at the time of the EDBT/ICDT 2023 conference.
- GQL, a standalone language for querying property graphs, that is expected to be published in late 2023 or early 2024.

The language of the Standard, even when published (behind paywall) is hardly of the kind that the research community is accustomed to. It consists of a grammar for the constructs, supplemented with syntax and semantic rules, the latter written in natural language describing an algorithm for computing the result of a particular operation (essentially a mix of prose and pseudocode). Such descriptions are long, far from formal definitions suitable for initiating research in the area, and often prone to misinterpretation. To researchers, such a text is therefore much like a 500+ page legal document, instead of a workable definition that helps them understand the essence of the language.

This motivates the goal of the present paper: *to distill, in a form accessible to the database research community, the principal elements of the forthcoming GQL Standard, and provide their formal semantics.*

The idea of finding calculi underlying programming languages and providing their formal semantics is mainstream in the programming languages field. Recently we saw it extended to database query languages, specifically to core fragments of SQL [10, 22, 7] and Cypher [18]. The present paper follows this trend. It provides a significant simplification of the GQL Standard, which at the same time covers its key features, and yet is sufficiently simple to provide its formal semantics, thereby enabling its further study and opening up new avenues of research on graph query languages.

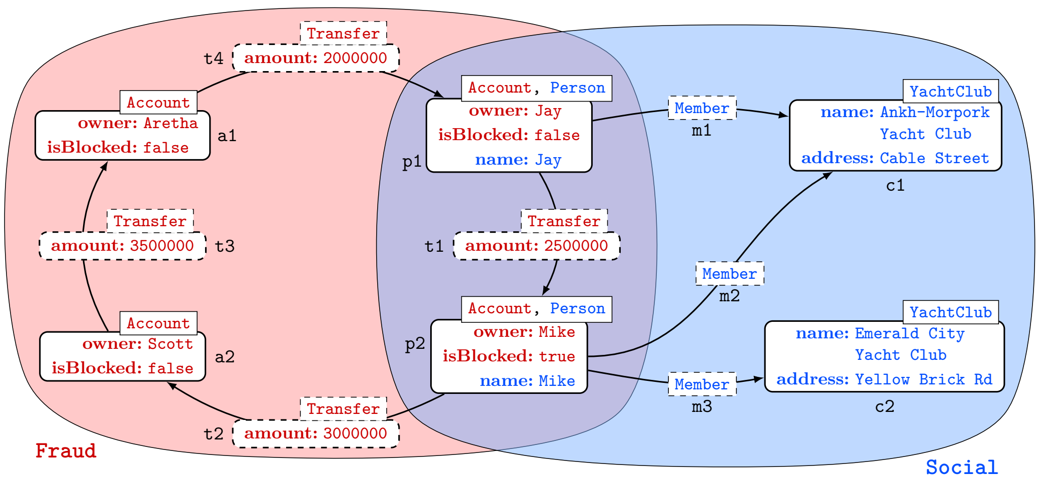


Figure 1 A database with graphs Fraud and Social.

We do not follow GQL letter to letter, for two reasons. Firstly, the Standard itself is not yet finalized, and what is written today may still change before it is published. Second, we choose to simplify some of the idiosyncrasies of a real-life language to better highlight its essential features. Queries presented here are close to the eventual features of the language – even if they change somewhat in the meantime. They come with a formal grammar that is a fragment of GQL's grammar, and a formal semantics, that is suitable as a starting point of new research in graph query languages. The paper focuses on *read-only GQL queries*, to which we will simply refer as GQL queries. That is, we do not yet consider data updates.

Previous Academic Work on GQL

The two graph languages currently standardized – GQL and SQL/PGQ – share their *pattern matching* facilities, which constitute the key part of any graph language. These were described in [12], by a group that included members of ISO's Standard group, as well as members of LDBC's Formal Semantics Working Group (FSWG), whose goal was to analyze and formalize the design of the language. FSWG then produced a theoretical reconstruction of the GQL and PGQ pattern language [16]. This paper is the next installment in the effort to distill PGQ and GQL standards for the research community.

Apart from this recent work on GQL, we note that academic foundations already influenced its design process. As seen in GQL's influence graph [19], the language draws inspiration from regular path queries [11, 30], STRUQL [14], GXPath [27], and regular queries [32].

2 GQL by Example

In this section we give a high-level description of GQL queries and their evaluation. The graph database model used by GQL is simply a collection of one or more property graphs. As an illustration, Figure 1 is a graph database consisting of two property graphs: the Fraud graph has information about bank transactions that are to be investigated for fraud, and the Social graph has information about people's social activities such as membership in a yacht club. Notice that these two graphs have a non-empty intersection: the nodes for Jay and Mike belong to both graphs, but they are seen in a different way and therefore have different labels and properties. In `Fraud`, the nodes have label `Account` and properties `owner` and `isBlocked`, indicating the status of the account. In `Social`, these nodes have label `Person` and property `name`.

We start with a simple query that looks for large (over $1M) transfers into a blocked account, and reports owners of accounts involved in such transfers:

```
1. USE Fraud
2. MATCH (x) -[z:Transfer WHERE z.amount>1000000]-> (y WHERE y.isBlocked=true)
3. RETURN x.owner AS sender, y.owner AS recipient
```

The reader familiar with Cypher will parse this query easily; it roughly follows Cypher's ascii-art syntax for expressing patterns, and also permits checking conditions on properties inside patterns. Basically, the pattern in line 2, namely:

```
(x) -[z:Transfer WHERE z.amount>1000000]-> (y WHERE y.isBlocked=true)
```

asks for nodes `x` and `y` that are connected with an edge `z` that is labeled with `Transfer`. Furthermore, the `amount` property of `z` should exceed one million and the `isBlocked` property of y should be `true`. Such patterns, called *path patterns* in GQL, are the main building block of GQL queries, and they roughly correspond to regular path queries (RPQs), which have been well studied in the research literature [30].

Note also that the query is preceded by a `USE` clause stating explicitly in which graph matches are sought. When evaluating a query, GQL keeps track of

- the *working graph*, which is the current graph in the database on which we do pattern matching and
- the *working table*, which contains intermediate results of the query, up to the current evaluation point.

Intuitively, the working table is a collection of records that gets passed from one part of the query to another in order to compute the final result. Thus, while GQL is a graph query language, it uses tables to represent intermediate and end-results of queries. In Section 4, we also discuss a third ingredient that GQL keeps track of, namely the *working record*.

Coming back to our sample query, in the first line we write `USE Fraud`, which turns the `Fraud` graph into our working graph. In line 2, we have our path pattern, preceded by the keyword `MATCH`. This clause is the main workhorse of GQL, and it tells us to do the matching of the pattern onto the working graph. When evaluating our query over the database from Figure 1, after executing line 2 of the query, we will be left with the following working table:

x	y	z
p1	p2	t1

Continuing in line 3, the working table is modified by keeping only the `owner` attribute of the nodes `x` and `y`, while renaming them, and the following is returned to the user:

sender	recipient
Jay	Mike

(1)

We next extend this query by checking for such transfers where both account owners are members of the same yacht club, reporting this time the address for the yacht club to send investigators to.

```
 1. USE Fraud {
 2.  MATCH (x) -[z:Transfer WHERE z.amount>1000000]-> (y WHERE y.isBlocked=true)
 3.  RETURN x.owner AS sender, y.owner AS recipient
 4.  THEN
 5.  USE Social
 6.  MATCH (x1) -[:Member]-> (z1:YachtClub) ,
 7.        (y1) -[:Member]-> (z1:YachtClub)
 8.  FILTER sender=x1.name AND recipient=y1.name
 9.  RETURN z1.address AS clubAddress
10. }
```

Here lines 1–3 repeat the previous query. The keyword `THEN` is used to pipe the result of this query to the following subquery. While the curly braces extend the scope of `USE Fraud` beyond `THEN`, in line 5 we switch the working graph to `Social` in order to match the pattern:

```
(x1) -[:Member]-> (z1:YachtClub) ,  (y1) -[:Member]-> (z1:YachtClub)
```

This pattern consists of two path patterns, separated by a comma. In GQL, the comma performs a join on the results of the two path patterns. From a theoretical point of view, it brings us in the realm of *conjunctive (two-way) regular path queries.* In GQL, such patterns are called *graph patterns.* When this pattern is evaluated over the `Social` graph, we obtain the following (fresh) working table:

x1	y1	z1
p1	p2	c1
p1	p1	c1
p2	p2	c1

(2)

this time with variables `x1`, `y1`, and `z1`. After evaluating the pattern, the `MATCH` statement makes the natural join of table (2) with table (1), leading to

sender	recipient	x1	y1	z1
Jay	Mike	p1	p2	c1
Jay	Mike	p1	p1	c1
Jay	Mike	p2	p2	c1

In this case, this will be the Cartesian product since the two working tables have no variables in common. The `FILTER` condition in line 8 selects only the first row of the latter table. The `RETURN` statement in line 9 tells us to keep only the `address` attribute of `z1`, renamed as `clubAddress`, resulting in:

clubAddress
Cable Street

This is also where our query ends, and the working table contains all the results to our query.

The examples we have seen thus far illustrate only a limited part of GQL since their variables only bind to single nodes or edges. Next, we show what happens to variables that can bind to *lists* and *paths.* Concerning lists, a query[1] such as

```
USE Fraud
MATCH TRAIL (x) ((y)-[:Transfer]->()){1,} (x)
RETURN x AS source, y AS moneyTrail
```

would return the following table.

[1] Notice that the query uses Cypher's ascii-art `( )` for nodes in the subexpressions `(x)`, `(y)`, and `()`, but also uses `( )` for indicating the subexpression over which `{1,}` is applied.

source	moneyTrail
p1	list(p1, p2, a2, a1)
p2	list(p2, a2, a1, p1)
a2	list(a2, a1, p1, p2)
a1	list(a1, p1, p2, a2)

Here, the variable `y` is bound to a *list of nodes.* The four outputs all describe the same trail, which is the only Transfer-cycle in the graph, but the bindings use different start nodes for `x` and therefore also order the nodes in the lists for `y` differently. Concerning paths, the query

```
USE Fraud
MATCH TRAIL p = (x) (-[:Transfer]->()){1,} (x)
RETURN x AS source, p AS path
```

would return the following table.

source	path
p1	path(p1, t1, p2, t2, a2, t3, a1, t4, p1)
p2	path(p2, t2, a2, t3, a1, t4, p1, t1, p2)
a2	path(a2, t3, a1, t4, p1, t1, p2, t2, a2)
a1	path(a1, t4, p1, t1, p2, t2, a2, t3, a1)

The output is similar to the output of the previous example, but this time we have the entire path instead of the list of nodes in each answer. We note that property graphs can have multiple edges with the same end-nodes, so the list of nodes in a path is not sufficient to determine the path.

3 Syntax of GQL

The full syntax of *GQL queries* is given in Figure 2 with $\mathbb{G}$ a set of property graphs, and the following pairwise disjoint countable sets: $\mathcal{L}$ of labels, $\mathcal{K}$ of keys, Const of value constants with a designated value null, and Vars of variables.

While somewhat intimidating at a first glance, the grammar can be roughly divided into four parts:

- *path patterns*, which mimic regular path queries [29, 30], but have additional features such as two-way navigation and conditioning;
- *graph patterns*, which generalize conjunctive two-way regular path queries [8] with the ability to return different types of paths;
- *queries*, which allow us to manipulate the results of graph patterns and combine their evaluation over different graphs in the database; and
- *expressions and conditions*, which allow filtering results obtained in previous three parts of GQL.

Of course, each of these parts has many specific features. For instance, path patterns allow using descriptors, which bind a node/edge to a variable, test its label or more complex conditions (e.g. `amount` is greater than 1000000). Simple node/edge patterns can be combined into regular expressions, by using concatenation, union or repetitions. Graph patterns, on the other hand, allow specifying the subset of matched paths that is to be returned, or joining path patterns into more complex queries. Finally, clauses/queries themselves allow us to manipulate results obtained from graph patterns, much like what is possible in the relational. Complex features such as iteration over the returned elements, passing the results to another subquery, and changing the evaluation graph, are also supported.

PATH PATTERN For $x \in \mathsf{Vars}$, $\ell \in \mathcal{L}$, $0 \leq n \leq m \in \mathbb{N}$:

$$
\begin{array}{llll}
\text{(descriptor)} & \delta & := & x \;\; \mathtt{:}\ell \;\; \mathtt{WHERE}\ \theta & x,\ \mathtt{:}\ell, \text{ and } \mathtt{WHERE}\ \theta \text{ are optional} \\
\text{(path pattern)} & \pi & := & \mathtt{(}\,\delta\,\mathtt{)} & \text{(node pattern)} \\
 & & \mid & \mathtt{-[}\,\delta\,\mathtt{]->} \mid \mathtt{<-[}\,\delta\,\mathtt{]-} \mid \mathtt{\sim[}\,\delta\,\mathtt{]\sim} & \text{(edge pattern)} \\
 & & \mid & \pi\,\pi & \text{(concatenation)} \\
 & & \mid & \pi\,\mathtt{|}\,\pi & \text{(union)} \\
 & & \mid & \pi\ \mathtt{WHERE}\ \theta & \text{(conditioning)} \\
 & & \mid & \pi\mathtt{\{}n\mathtt{,}m\mathtt{\}} & \text{(bounded repetition)} \\
 & & \mid & \pi\mathtt{\{}n\mathtt{,\}} & \text{(unbounded repetition)}
\end{array}
$$

EXPRESSION and **CONDITION** For $x \in \mathsf{Vars}$, $\ell \in \mathcal{L}$, $a \in \mathcal{K}$, $c \in \mathsf{Const}$:

$$
\begin{array}{llll}
\text{(expression)} & \chi & := & x \mid x\mathtt{.}a \mid c \\
\text{(condition)} & \theta & := & \chi \mathtt{=} \chi \mid \chi \mathtt{<} \chi \mid \chi\ \mathtt{IS\ NULL} \\
 & & & \mid x\ \mathtt{:}\ \ell \mid \mathtt{EXISTS}\ \mathtt{\{}\,\mathsf{Q}\,\mathtt{\}} \\
 & & & \mid \theta\ \mathtt{OR}\ \theta \mid \theta\ \mathtt{AND}\ \theta \mid \mathtt{NOT}\ \theta
\end{array}
$$

GRAPH PATTERN For $x \in \mathsf{Vars}$:

$$
\begin{array}{llll}
\text{(path mode)} & \mu & := & (\mathtt{ALL} \mid \mathtt{ANY})\ [\mathtt{SHORTEST}]\ [\mathtt{TRAIL} \mid \mathtt{ACYCLIC}] \\
\text{(graph pattern)} & \Pi & := & \mu\ \ [x\ \mathtt{=}]\ \ \pi \mid \Pi\mathtt{,}\,\Pi
\end{array}
$$

CLAUSE and **QUERY** For $k \geq 0$, $\ell \geq 1$, and $x, y, x_1, \ldots, x_k \in \mathsf{Vars}$, and $G \in \mathbb{G}$:

$$
\begin{array}{llll}
\text{(clause)} & \mathsf{C} & := & \mathtt{MATCH}\ \Pi \\
 & & \mid & \mathtt{LET}\ x\ \mathtt{=}\ \chi \\
 & & \mid & \mathtt{FOR}\ x\ \mathtt{IN}\ y \\
 & & \mid & \mathtt{FILTER}\ \theta \\
\text{(linear query)} & \mathsf{L} & := & \mathtt{USE}\ G\ \mathsf{L} \\
 & & \mid & \mathsf{C}\ \mathsf{L} \\
 & & \mid & \mathtt{RETURN}\ \chi_1\ \mathtt{AS}\ x_1,\ \ldots,\ \chi_k\ \mathtt{AS}\ x_k \\
\text{(query)} & \mathsf{Q} & := & \mathsf{L} \\
 & & \mid & \mathtt{USE}\ G\ \mathtt{\{}\mathsf{Q}_1\ \mathtt{THEN}\ \mathsf{Q}_2\ \cdots\ \mathtt{THEN}\ \mathsf{Q}_\ell\mathtt{\}} \\
 & & \mid & \mathsf{Q}\ \mathtt{INTERSECT}\ \mathsf{Q} \mid \mathsf{Q}\ \mathtt{UNION}\ \mathsf{Q} \mid \mathsf{Q}\ \mathtt{EXCEPT}\ \mathsf{Q}
\end{array}
$$

Figure 2 Syntax of GQL.

Well-Formed Queries

The syntax of path patterns defined in Figure 2 is permissive as it allows expressions that do not type-check. For example, `(x)-[x]->()` is syntactically permitted even though it equates a node variable with an edge variable. Other patterns would provide great expressive power, such as the graph pattern `()-[y]->{0,}(), ()-[y]->{0,}*()`, which implicitly joins on lists.

We introduced in [16] a type system operating on a subset of the patterns described in Figure 2. Its goal is to ensure that GQL path patterns and graph patterns do not exhibit the pathological behavior illustrated above. Here, we will only describe the resulting syntactic restrictions informally.

Each variable is given a *type* τ from the set $\mathbb{T}$ defined by the following grammar.

$$\tau ::= \mathsf{Node} \mid \mathsf{Edge} \mid \mathsf{Path} \mid \mathsf{Maybe}(\tau) \mid \mathsf{Group}(\tau)$$

The three atomic types are used for variables returning nodes, edges, and paths, respectively. The type constructor Maybe is used for variables occurring on one side of a disjunction only, while Group is used for variables occurring under repetition, whose bindings are grouped together. As variables in pattern matching are never bound to data values, we do not need the usual types like integers or strings here.

Types are computed in a bottom-up fashion as follows. Variables appearing in node patterns (resp. in edge patterns, resp. as names of path patterns) are of type Node (resp. Edge, resp. Path). Variables appearing on one side of a disjunction with type τ but not the other are of type Maybe(τ). Variables appearing under a repetition with type τ are of type Group(τ) higher-up in the syntax tree of the expression. Consider the pattern `(-[`x`]-> | -[`y`]->){0,}`. The type of x is Edge in `-[`x`]->`, while it is Maybe(Edge) in `-[`x`]-> | -[`y`]->`, and Group(Maybe(Edge)) in `(-[`x`]-> | -[`y`]->){0,}` .

A variable x appearing in a path/graph pattern ξ is called:

- a *singleton* variable if its type is Node or Edge with respect to ξ
- a *conditional* variable if its type is Maybe(τ) for some type τ;
- a *group* variable if its type is Group(τ) for some type τ;
- a *path* variable if its type is Path.

Here is a non-exhaustive list of the syntactic conditions a pattern must meet in order for its semantics to be defined. A pattern ξ is *well-formed* if

1. Every variable appearing in a pattern ξ has one and only one type w.r.t. ξ.
2. In concatenation and join, variables appearing in both operands are singleton variables with respect to each operand.
3. In a conditioned path pattern π **WHERE** θ, every variable appearing in θ must have a type w.r.t. π.
4. In a graph pattern of the form $\mu\ \pi$ or $\mu\ x$ `=` π such that μ is **ALL** (which is possible since all of **SHORTEST**, **TRAIL**, and **ACYCLIC** are optional), π must contain no unbounded repetition, to avoid potentially infinite outputs.
5. For every repeated pattern π`{`n`,`m`}` or π`{`n`,}`, the *minimum path length* $\|\pi\|_{\min}$ of π, defined below, is positive. This avoids applying repetitions to paths that do not match an edge.

$$\|\nu\|_{\min} = 0 \qquad \|\pi \ \textbf{WHERE}\ \theta\|_{\min} = \|\pi\|_{\min}$$
$$\|\eta\|_{\min} = 1 \qquad \|\pi_1 \mid \pi_2\|_{\min} = \min(\|\pi_1\|_{\min}, \|\pi_2\|_{\min})$$
$$\|\pi\texttt{\{}n,\texttt{\}}\|_{\min} = \|\pi\texttt{\{}n,m\texttt{\}}\|_{\min} = n \cdot \|\pi\|_{\min} \qquad \|\pi_1\,\pi_2\|_{\min} = \|\pi_1\|_{\min} + \|\pi_2\|_{\min}$$

Note that the local nature of types is important in item 2: implicit joins are allowed under repetitions, as in `( (a)-[]->(b)-[]->(a)-[]-> ){1,}`. Moreover, item 1 implies the existence of a schema, which is defined as follows:

▶ **Definition 1** (Schema). *A* schema *of a well-formed pattern* ξ *is a function* $\mathsf{sch}(\xi) : \mathsf{var}(\xi) \to \mathbb{T}$*, where* $\mathsf{var}(\xi)$ *is the set of variables appearing in* ξ*.*

We will assume these syntactic restrictions to be in place when defining the semantics of GQL queries in Section 4. Moreover, we define the semantics only when the computation goes as expected, that is, when it satisfies preconditions we state explicitly. For instance, we will assume that a variable is bound before being used, that we never run into clashes in variable names, and that if a specific type is expected for an operation, then the value will have that type at runtime. Some of the preconditions could be checked syntactically, at the cost of a tedious type system. Some of the preconditions cannot be checked before run-time because they depend on the data stored in the database. Deciding how to treat those cases (static analysis, runtime exceptions, implicit casts) is outside the scope of this paper. In some cases, the GQL standard describes how they should be treated, in others, they are implementation-dependent.

4 Semantics

In this section we present the formal semantics of GQL. At a high level, when evaluating a query, GQL keeps track of three things: (i) the *working graph*, which is the property graph we are using to match our patterns currently; (ii) the *working table*, that stores the information computed thus far; and (iii) the *working record*, which contains the tuple of the result we are currently using. In this section we provide mathematical abstractions for each of these concepts in order to define the semantics of GQL. We start by setting the preliminary definitions, and then move to defining the semantics for each portion of the language, as specified in Figure 2.

4.1 Preliminaries

Data model. We follow the formal definition adapted by the GQL Standard [20] to handle databases that contain multiple graphs. To define property graphs we need, in addition to the pairwise disjoint countable sets ($\mathcal{L}$ of labels, $\mathcal{K}$ of keys, and Const of constants) mentioned in Section 3, the following fresh pairwise disjoint countable sets: $\mathcal{N}$ of node ids, $\mathcal{E}_\mathsf{d}$ of directed edge ids, and $\mathcal{E}_\mathsf{u}$ of undirected edge ids.

▶ **Definition 2** (Property Graph). *A property graph is a tuple*

$$G = \langle N^G, E_\mathsf{d}^G, E_\mathsf{u}^G, \mathsf{lab}^G, \mathsf{endpoints}^G, \mathsf{src}^G, \mathsf{tgt}^G, \mathsf{prop}^G \rangle$$

where

- $N^G \subset \mathcal{N}$ *is a finite set of node ids used in* G*;*
- $E_\mathsf{d}^G \subset \mathcal{E}_\mathsf{d}$ *is a finite set of directed edge ids used in* G*;*
- $E_\mathsf{u}^G \subset \mathcal{E}_\mathsf{u}$ *is a finite set of undirected edge ids used in* G*;*
- $\mathsf{lab}^G : N^G \cup E_\mathsf{d}^G \cup E_\mathsf{u}^G \to 2^{\mathcal{L}}$ *is a labeling function that associates with every id a (possibly empty) finite set of labels from* $\mathcal{L}$*;*

- $\mathsf{src}^G, \mathsf{tgt}^G : E^G_{\mathsf{d}} \to N^G$ *define source and target of a directed edge;*
- $\mathsf{endpoints}^G : E^G_{\mathsf{u}} \to 2^N$ *so that* $|\mathsf{endpoints}^G(e)|$ *is 1 or 2 define endpoints of an undirected edge;*
- $\mathsf{prop}^G : (N^G \cup E^G_{\mathsf{d}} \cup E^G_{\mathsf{u}}) \times \mathcal{K} \to \mathit{Const}$ *is a partial function that associates a constant with an id and a key from* $\mathcal{K}$.

If G is clear from the context, it will be omitted in the superscript. Recall that $\mathbb{G}$ denotes the set of all property graphs.

We use *node* and *edge* to refer to node ids and edge ids, respectively, and call a node u an *ℓ-node* iff $\ell \in \mathsf{lab}(u)$; similarly for edges.

▶ **Definition 3** (Graph Database). *A (property) graph database is a tuple $D = \langle G_1, \ldots, G_k \rangle$ where each G_i is a property graph. We call the graph G_1 the* default graph.[2]

This is the most general definition of a database containing multiple graphs and it imposes no restrictions whatsoever on how labeling, properties, and topology agree across different graphs that share some node and edge ids. For example we may have the same id_1 for a person who has label *employee* and properties *salary, department* in a company graph and label *student* and properties *year, major* in a university graph. In fact it is even possible that the same edge id has different source and target in different graphs. We allow this complete flexibility because it is orthogonal to the choice of operations in the language, and thus we shall not impose restrictions that are not necessary for our purposes.

Paths and lists. GQL allows returning paths and lists as query answers. Here we define them formally. We start with paths.

▶ **Definition 4** (Path). *A* path *is an alternating sequence of nodes and edges that starts and ends with a node. We write paths as $p = \mathsf{path}(u_0, e_1, u_1, e_2, \cdots e_n, u_n)$, where $u_0, \ldots, u_n$ are nodes, $e_1, \ldots, e_n$ are (directed or undirected) edges, and $n \geq 0$. We write $\mathsf{src}(p)$ for u_0 and $\mathsf{tgt}(p)$ for u_n, and $\mathsf{len}(p)$ for its length n. We denote the set of all paths by Paths.*

For a property graph G, we say that $p \in \mathsf{Paths}$ is a path in G *if each edge in p connects the nodes before and after it in the sequence, that is, for each $i \in \{1, \ldots, n\}$, at least one of the following is true:*

(a) $\mathsf{src}(e_i) = u_{i-1}$ *and* $\mathsf{tgt}(e_i) = u_i$ *in which case we speak of e_i as a* forward *edge in the path;*
(b) $\mathsf{src}(e_i) = u_i$ *and* $\mathsf{tgt}(e_i) = u_{i-1}$ *in which case we speak of e_i as a* backward *edge in the path;*
(c) $\mathsf{endpoints}(e_i) = \{u_{i-1}, u_i\}$ *in which case we speak of e_i as an* undirected *edge in the path.*

We denote the set of paths in G by $\mathsf{Paths}(G)$.

Note that we allow $n = 0$, in which case the path consists of a single vertex and no edges. Note also that in the case of a directed self-loop, both (a) and (b) in the definition above are true, hence the cases are not mutually exclusive.

▶ **Definition 5** (Concatenation of Paths). *Two paths $p = \mathsf{path}(u_0, e_0, \ldots, u_k)$ and $p' = \mathsf{path}(u'_0, e'_0, \ldots, u'_j)$* concatenate *if $u_k = u'_0$, in which case their* concatenation *$p \cdot p'$ is defined as $\mathsf{path}(u_0, e_0, \ldots, u_k, e'_0, \ldots, u'_j)$.*

[2] The default graph is used for evaluation when a specific graph is not declared by the query.

Note that a path of length 0 is a neutral element of concatenation; that is, $p \cdot \mathsf{path}(u)$ is defined iff $u = \mathsf{tgt}(p)$, in which case $p = p \cdot \mathsf{path}(u)$; likewise for $\mathsf{path}(u) \cdot p$ and $u = \mathsf{src}(p)$.

▶ **Definition 6** (List). *We use the notation* $\mathsf{list}(v_1, \ldots, v_n)$ *to denote the list containing the objects* $v_1, \ldots, v_n$ *in this order. Lists can be empty, in which case we write* $\mathsf{list}()$*. We use* Lists *to denote the set of all lists with elements in* $\mathcal{N} \cup \mathcal{E}_\mathsf{d} \cup \mathcal{E}_\mathsf{u}$*.*

Bindings. To define the formal semantics we use bindings which specify how variables are matched to values $\mathbb{V}$ of the input graph database. Intuitively, a binding is a mathematical formalization of the concept of a working record in GQL. Formally, we set $\mathbb{V}$ as the union $\mathsf{Const} \cup \mathcal{N} \cup \mathcal{E}_\mathsf{d} \cup \mathcal{E}_\mathsf{u} \cup \mathsf{Paths} \cup \mathsf{Lists}$.

▶ **Definition 7** (Binding). *A* binding μ *is a partial function* $\mu : \mathsf{Vars} \to \mathbb{V}$ *whose domain* $\mathrm{Dom}(\mu)$ *is finite. We denote bindings* μ *explicitly by* $(x_1 \mapsto v_1, \ldots, x_n \mapsto v_n)$ *where* $x_1, \ldots, x_n$ *are variables in* $\mathrm{Dom}(\mu)$*,* $v_1, \ldots, v_n$ *are values in* $\mathbb{V}$*, and for every* i *it holds that* $\mu(x_i) = v_i$*.*

Note that the domains of bindings are not ordered, hence for instance $(a_1 \mapsto v_1, a_2 \mapsto v_2) = (a_2 \mapsto v_2, a_1 \mapsto v_1)$. The *empty binding*, that is, the binding with an empty domain, is denoted by $()$.

▶ **Definition 8** (Compatibility of Bindings). *Two bindings* μ_1, μ_2 *are said to be compatible, denoted by* $\mu_1 \sim \mu_2$*, if they agree on their shared variables, that is, for every* $x \in \mathrm{Dom}(\mu_1) \cap \mathrm{Dom}(\mu_2)$ *it holds that* $\mu_1(x) = \mu_2(x)$*.*

If $\mu_1 \sim \mu_2$, we define their *join* $\mu_1 \bowtie \mu_2$ as expected, that is $\mathrm{Dom}(\mu_1 \bowtie \mu_2) = \mathrm{Dom}(\mu_1) \cup \mathrm{Dom}(\mu_2)$ and $(\mu_1 \bowtie \mu_2)(x) = \mu_1(x)$ whenever $x \in \mathrm{Dom}(\mu_1) \setminus \mathrm{Dom}(\mu_2)$, and $(\mu_1 \bowtie \mu_2)(x) = \mu_2(x)$ whenever $x \in \mathrm{Dom}\,(\mu_2)$.

We remark here that our definition allows joins on variables that are bound to paths or lists. However, as we will see, the syntactic restrictions on queries limit this feature significantly.

4.2 Semantics of Path Patterns

We start by defining the semantics of path patterns. For the remainder of this subsection, we consider a fixed property graph

$$G = \langle N^G, E_\mathsf{d}^G, E_\mathsf{u}^G, \mathsf{lab}^G, \mathsf{endpoints}^G, \mathsf{src}^G, \mathsf{tgt}^G, \mathsf{prop}^G \rangle.$$

Moreover, we assume that all queries are well-formed and all patterns considered are restricted syntactically as described in Section 3. The semantics $[\![\pi]\!]_G$ of a pattern π is a set of pairs (p, μ) where μ a binding, and p is a path in G. In $[\![\pi]\!]_G$, G denotes the working graph in GQL parlance (specified by the keyword `USE`), and the pairs (p, μ) model what is computed over this working graph.

Semantics of Node and Edge Patterns

$$[\![\texttt{()}]\!]_G = \{ (n, ()) \mid n \in N^G \} \qquad [\![\texttt{(}x\texttt{)}]\!]_G = \{ (n, (x \mapsto n)) \mid n \in N^G \}$$

$$[\![\texttt{(:}\ell\texttt{)}]\!]_G = \left\{ (n, ()) \;\middle|\; n \in N^G,\ \ell \in \mathsf{lab}^G(n) \right\}$$

Other cases are treated by moving the label and conditions outside of the node pattern. For instance, `(`x`:`ℓ `WHERE` θ`)` is rewritten as `(`x`) WHERE (`x`:`ℓ `AND` θ`)`.

$$\begin{aligned}
[\![\texttt{-[]->}]\!]_G &= \{\, (\mathsf{path}(\mathsf{src}(e), e, \mathsf{tgt}(e)), ()) \mid e \in E_d^G \,\} \\
[\![\texttt{-[}x\texttt{]->}]\!]_G &= \{\, (\mathsf{path}(\mathsf{src}(e), e, \mathsf{tgt}(e)), (x \mapsto e)) \mid e \in E_d^G \,\} \\
[\![\texttt{-[:}\ell\texttt{]->}]\!]_G &= \left\{\, (\mathsf{path}(\mathsf{src}(e), e, \mathsf{tgt}(e)), ()) \;\middle|\; e \in E_d^G, \ell \in \mathsf{lab}^G(e) \,\right\}
\end{aligned}$$

Other cases of the forward edge patterns are treated by moving the label and conditions outside of the edge pattern, just as for node patterns. Backward edge patterns and undirected edge patterns are treated similarly, with the base cases given below.

$$\begin{aligned}
[\![\texttt{<-[]-}]\!]_G &= \{\, (\mathsf{path}(\mathsf{tgt}(e), e, \mathsf{src}(e)), ()) \mid e \in E_d^G \,\} \\
[\![\texttt{~[]~}]\!]_G &= \left\{\, (\mathsf{path}(u_1, e, u_2), ()), (\mathsf{path}(u_2, e, u_1), ()) \;\middle|\; \begin{array}{l} e \in E_u^G \\ \{u_1, u_2\} = \mathsf{endpoints}^G(e) \end{array} \right\}
\end{aligned}$$

Semantics of Concatenation, Union, and Conditioning

$$[\![\pi_1\, \pi_2]\!]_G \left\{ (p_1 \cdot p_2, \mu_1 \bowtie \mu_2) \;\middle|\; \begin{array}{l} (p_i, \mu_i) \in [\![\pi_i]\!]_G \text{ for } i = 1, 2 \\ p_1 \text{ and } p_2 \text{ concatenate} \\ \mu_1 \sim \mu_2 \end{array} \right\}$$

Note that since $\pi_1\, \pi_2$ is assumed to be well-formed, all variables shared by π_1 and π_2 are singleton variables (Condition 2 in Section 3). In other words, implicit joins over group and optional variables are disallowed; the same remark will also apply for the semantics of joins.

▶ Remark 9. Consider the pattern

```
(x) (-[:Transfer]->()-[:Transfer]->(x)]){1,}
```

This pattern is disallowed in GQL because the leftmost `x` is a singleton variable, whereas the rightmost `x` is a group variable. In GQL philosophy, the leftmost `x` will be bound to a node and the rightmost `x` will be bound to a list of nodes, which is a type mismatch.

$$[\![\pi_1 \mid \pi_2]\!]_G = \{\, (p, \mu \cup \mu') \mid (p, \mu) \in [\![\pi_1]\!]_G \cup [\![\pi_2]\!]_G \,\}$$

where μ' maps every variable in $\mathsf{var}(\pi_1 \mid \pi_2) \setminus \mathrm{Dom}(\mu)$ to null. (Recall that var maps a pattern to the set of variables appearing in it.)

$$[\![\pi \ \texttt{WHERE}\ \theta]\!]_G = \{\, (p, \mu) \in [\![\pi]\!]_G \mid [\![\theta]\!]_G^\mu = \mathsf{true} \,\}$$

Semantics of Repetition

$$\begin{aligned}
[\![\pi\texttt{\{}n, m\texttt{\}}]\!]_G &= \bigcup_{i=n}^{m} [\![\pi]\!]_G^i \\
[\![\pi\texttt{\{}n,\texttt{\}}]\!]_G &= \bigcup_{i=n}^{\infty} [\![\pi]\!]_G^i
\end{aligned}$$

Above, for a pattern π and a natural number $i \geq 0$, we use $[\![\pi]\!]_G^i$ to denote the i-th power of $[\![\pi]\!]_G$, which we define as

$$[\![\pi]\!]_G^0 = \{\, (\mathsf{path}(u), \mu) \mid u \text{ is a node in } G \,\}$$

where μ binds each variable in $\mathrm{Dom}(\mathsf{sch}(\pi))$ to $\mathsf{list}()$, that is, the empty-list value; and

$$\forall i > 0 \quad [\![\pi]\!]_G^i = \left\{ (p_1 \cdot \ldots \cdot p_i, \mu') \;\middle|\; \begin{array}{l} (p_1, \mu_1), \ldots, (p_n, \mu_i) \in [\![\pi]\!]_G \\ p_1, \ldots, p_i \text{ concatenate} \end{array} \right\}$$

where μ' binds each variable in $\mathrm{Dom}(\mathsf{sch}(\pi))$ to $\mathsf{list}\big(\mu_1(x), \ldots, \mu_i(x)\big)$. Recall that sch is defined in Section 3.

▶ Remark 10. Since π`{`$n,$`}` is assumed to be well-formed, it holds $\|\pi\|_{\min} \geq 1$. A simple induction then yields that each p_i in the definition above has positive length. A second induction then yields that, given a path p, there are finitely many assignments μ such that $(p,\mu) \in [\![\pi\texttt{\{}n,m\texttt{\}}]\!]_G$. This fact is crucial to have a finite output in the end.

For instance, consider a graph with a single node u and no edges, and the pattern `(a){0,}` which is not well-formed (the minimal path length of `()` is 0). For every i, the set $[\![\texttt{(a)}]\!]^i_G$ contains $(\mathsf{path}(u), \mu_i)$ where $\mu_i = (a \mapsto \mathsf{list}(\underbrace{u,\ldots,u}_{i\text{ times}}))$; hence the union in the definition of $[\![\pi\texttt{\{}n,\texttt{\}}]\!]_G$ above would not only yield an infinite number of elements, but all of them would be associated to the same path. As a result a graph pattern such as `ALL SHORTEST (a){0,}` would have infinitely many results.

4.3 Semantics of Graph Patterns

We now define the semantics of graph patterns. We first fully define atomic graph patterns and then define their joins.

$$[\![x\,\texttt{=}\,\pi]\!]_G = \{(p, \mu \cup \{x \mapsto p\}) \mid (p,\mu) \in [\![\pi]\!]_G\}$$

In the following we denote by $\tilde{\pi}$ a graph pattern that never uses the "`,`" operator, hence it is of the form $\mu\ x\texttt{=}\ \pi$, where μ is a path mode, x is a variable, π is a path pattern, and "x`=`" is optional.

$$[\![\texttt{TRAIL}\ \pi]\!]_G = \{\, (p,\mu) \in [\![\pi]\!]_G \mid \text{no edge occurs more than once in } p\,\}$$
$$[\![\texttt{ACYCLIC}\ \pi]\!]_G = \{\, (p,\mu) \in [\![\pi]\!]_G \mid \text{no node occurs more than once in } p\,\}$$
$$[\![\texttt{SHORTEST}\ \tilde{\pi}]\!]_G = \left\{ (p,\mu) \in [\![\tilde{\pi}]\!]_G \;\middle|\; \mathsf{len}(p) = \min \left\{ \mathsf{len}(p') \;\middle|\; \begin{array}{l} (p',\mu') \in [\![\tilde{\pi}]\!]_G \\ \mathsf{src}(p') = \mathsf{src}(p) \\ \mathsf{tgt}(p') = \mathsf{tgt}(p) \end{array} \right\} \right\}$$
$$[\![\texttt{ALL}\ \tilde{\pi}]\!]_G = [\![\tilde{\pi}]\!]_G$$
$$[\![\texttt{ANY}\ \tilde{\pi}]\!]_G = \bigcup_{(s,t)\in X} \{\mathsf{any}(\{\,(p,\mu) \mid (p,\mu) \in [\![\tilde{\pi}]\!]_G, \mathsf{endpoints}(p) = (s,t)\,\})\}$$

where $X = \{\,(\mathsf{src}(p), \mathsf{tgt}(p)) \mid (p,\mu) \in [\![\tilde{\pi}]\!]_G\,\}$ and any is a procedure that arbitrarily returns one element from a set; any need not be deterministic.

$$[\![\Pi_1\,\texttt{,}\ \Pi_2]\!]_G = \{\,(\bar{p}_1 \times \bar{p}_2, \mu_1 \bowtie \mu_2) \mid (\bar{p}_i, \mu_i) \in [\![\Pi_i]\!]_G \text{ for } i = 1,2 \text{ and } \mu_1 \sim \mu_2\,\}$$

Here, $\bar{p}_1 = (p_1^1, p_1^2, \ldots, p_1^k)$ and $\bar{p}_2 = (p_2^1, p_2^2, \ldots, p_2^l)$ are tuples of paths, and $\bar{p}_1 \times \bar{p}_2$ stands for $(p_1^1, p_1^2, \ldots, p_1^k, p_2^1, p_2^2, \ldots, p_2^l)$. Just as it is the case of concatenation, since $\Pi_1\,\texttt{,}\ \Pi_2$ is well-formed, implicit joins can occur over singleton variables only.

4.4 Semantics of Conditions and Expressions

The semantics $[\![\chi]\!]^\mu_G$ of an expression χ is an element in $\mathbb{V}$ that is computed with respect to a binding μ and a graph G. Intuitively, variables in χ are evaluated with μ and we use G to access the properties of an element. It is formally defined as follows.

$$[\![c]\!]^\mu_G = c \qquad \text{for } c \in \mathsf{Const}$$
$$[\![x]\!]^\mu_G = \mu(x) \qquad \text{for } x \in \mathrm{Dom}(\mu)$$
$$[\![x.a]\!]^\mu_G = \begin{cases} \mathsf{prop}^G(\mu(x), a) & \text{if } (\mu(x), a) \in \mathrm{Dom}(\mathsf{prop}^G) \\ \mathsf{null} & \text{else if } \mu(x) \in (\mathcal{N} \cup \mathcal{E}_\mathsf{d} \cup \mathcal{E}_\mathsf{u}) \end{cases} \qquad \text{for } x \in \mathrm{Dom}(\mu), a \in \mathcal{K}$$

▶ Remark 11. Recall that different graphs may share nodes and edges. Hence the condition $(\mu(x), a) \in \mathrm{Dom}(\mathsf{prop}^G)$, above, does imply that $\mu(x)$ is a node or an edge in G, but does **not** imply that it was matched in G.

The semantics $[\![\theta]\!]^\mu_G$ of a condition θ is an element in $\{\mathsf{true}, \mathsf{false}, \mathsf{null}\}$ that is evaluated with respect to a binding μ and a graph G, and is defined as follows:

$$[\![\chi_1 \texttt{ = } \chi_2]\!]^\mu_G = \begin{cases} \mathsf{null} & \text{if } [\![\chi_1]\!]^\mu_G = \mathsf{null} \text{ or } [\![\chi_2]\!]^\mu_G = \mathsf{null} \\ \mathsf{true} & \text{if } [\![\chi_1]\!]^\mu_G = [\![\chi_2]\!]^\mu_G \neq \mathsf{null} \\ \mathsf{false} & \text{otherwise} \end{cases}$$

$$[\![\chi_1 \texttt{ < } \chi_2]\!]^\mu_G = \begin{cases} \mathsf{null} & \text{if } [\![\chi_1]\!]^\mu_G = \mathsf{null} \text{ or } [\![\chi_2]\!]^\mu_G = \mathsf{null} \\ \mathsf{true} & \text{else if } [\![\chi_1]\!]^\mu_G < [\![\chi_2]\!]^\mu_G \\ \mathsf{false} & \text{otherwise} \end{cases}$$

$$[\![\chi \texttt{ IS NULL}]\!]^\mu_G = \begin{cases} \mathsf{true} & \text{if } [\![\chi]\!]^\mu_G = \mathsf{null} \\ \mathsf{false} & \text{otherwise} \end{cases}$$

$$[\![\chi \texttt{:} \ell]\!]^\mu_G = \begin{cases} \mathsf{true} & \text{if } [\![\chi]\!]^\mu_G \in N^G \cup E^G_u \cup E^G_d \text{ and } \ell \in \mathsf{lab}^G([\![\chi]\!]^\mu_G) \\ \mathsf{false} & \text{else if } [\![\chi]\!]^\mu_G \in \mathcal{N} \cup \mathcal{E}_\mathsf{d} \cup \mathcal{E}_\mathsf{u} \end{cases}$$

$$[\![\theta_1 \texttt{ AND } \theta_2]\!]^\mu_G = [\![\theta_1]\!]^\mu_G \wedge [\![\theta_2]\!]^\mu_G \quad {}^{(*)}$$

$$[\![\theta_1 \texttt{ OR } \theta_2]\!]^\mu_G = [\![\theta_1]\!]^\mu_G \vee [\![\theta_2]\!]^\mu_G \quad {}^{(*)}$$

$$[\![\texttt{NOT } \theta]\!]^\mu_G = \neg\, [\![\theta]\!]^\mu_G \quad {}^{(*)}$$

$^{(*)}$ Operators $\wedge$, $\vee$, and $\neg$ are defined as in SQL three-valued logic, e.g. $\mathsf{null} \vee \mathsf{true} = \mathsf{true}$ while $\mathsf{null} \wedge \mathsf{true} = \mathsf{null}$.

$$[\![\texttt{EXISTS \{} \mathsf{Q} \texttt{\}}]\!]^\mu_G = \begin{cases} \mathsf{true} & \text{if } [\![\mathsf{Q}]\!]_G(\{\mu\}) \text{ is not empty} \\ \mathsf{false} & \text{otherwise} \end{cases}$$

4.5 Semantics of Queries

Clauses and queries are interpreted as functions that operate on tables. These tables are our abstraction of GQL's working tables.

▶ **Definition 12.** *A* table T *is a set of bindings that have the same domains, referred to as* $\mathrm{Dom}(T)$.

Note that tables do not have schemas: two different bindings in a table might associate a variable to values of incompatible types.

Semantics of Clauses

The semantics $[\![\mathsf{C}]\!]_G$ of a clause C is a function that maps tables into tables, and is parametrized by a graph G. Patterns, conditions and expression in a clause are evaluated with respect to that G.

$$[\![\texttt{MATCH } \Pi]\!]_G(T) = \bigcup_{\mu \in T} \{\mu \bowtie \mu' \mid (p, \mu') \in [\![\Pi]\!]_G,\ \mu \sim \mu'\}$$

Note that if Π uses a variable that already occurs in $\mathrm{Dom}(T)$, a join is performed. Unlike in the case of path patterns and graph patterns, this join can involve variables bound to lists or paths. While this is not problematic mathematically, it might be disallowed in future iterations of GQL.

If $x \notin \mathrm{Dom}(T)$, then

$$[\![\texttt{LET } x \texttt{ = } \chi]\!]_G\,(T) = \bigcup_{\mu \in T} \{\mu \bowtie (x \mapsto [\![\chi]\!]_G^{\mu})\}$$

$$[\![\texttt{FILTER } \theta]\!]_G\,(T) = \bigcup_{\mu \in T} \{\mu \mid [\![\theta]\!]_G^{\mu} = \mathsf{true}\} \;.$$

If $x \notin \mathrm{Dom}(T)$ and, for every $\mu \in T$, $\mu(y)$ is a list or null,[3] then

$$[\![\texttt{FOR } x \texttt{ IN } y]\!]_G\,(T) = \bigcup_{\mu \in T} \{\mu \bowtie (x \mapsto v) \mid v \in \mu(y)\} \;.$$

Semantics of Linear Queries

$$[\![\texttt{USE } G' \; \mathsf{L}]\!]_G\,(T) = [\![\mathsf{L}]\!]_{G'}\,(T)$$

$$[\![C \; \mathsf{L}]\!]_G\,(T) = [\![\mathsf{L}]\!]_G\,\big([\![C]\!]_G\,(T)\big)$$

$$[\![\texttt{RETURN } \chi_1 \texttt{ AS } x_1, \ldots, \chi_\ell \texttt{ AS } x_\ell]\!]_G\,(T) = \bigcup_{\mu \in T} \{(x_1 \mapsto [\![\chi_1]\!]_G^{\mu}, \ldots, x_\ell \mapsto [\![\chi_\ell]\!]_G^{\mu})\}$$

Semantics of Queries

The *output of a query* Q is defined as

$$\mathsf{Output}(\mathsf{Q}) = [\![\mathsf{Q}]\!]_G\,(\{()\}) \;,$$

where $\{()\}$ is the unit table that consists of the empty binding, and G is the default graph in D. We define the semantics of queries recursively as follows.

$$[\![\texttt{USE } G' \;\texttt{\{}\mathsf{Q}_1 \texttt{ THEN } \mathsf{Q}_2 \;\cdots\; \texttt{THEN } \mathsf{Q}_k\texttt{\}}]\!]_G\,(T) = [\![\mathsf{Q}_k]\!]_{G'} \circ \cdots \circ [\![\mathsf{Q}_1]\!]_{G'}\,(T)$$

If $\mathrm{Dom}\left([\![\mathsf{Q}_1]\!]_G\,(T)\right) = \mathrm{Dom}\left([\![\mathsf{Q}_2]\!]_G\,(T)\right)$, then we let

$$[\![\mathsf{Q}_1 \texttt{ INTERSECT } \mathsf{Q}_2]\!]_G\,(T) = [\![\mathsf{Q}_1]\!]_G\,(T) \cap [\![\mathsf{Q}_2]\!]_G\,(T)$$

$$[\![\mathsf{Q}_1 \texttt{ UNION } \mathsf{Q}_2]\!]_G\,(T) = [\![\mathsf{Q}_1]\!]_G\,(T) \cup [\![\mathsf{Q}_2]\!]_G\,(T)$$

$$[\![\mathsf{Q}_1 \texttt{ EXCEPT } \mathsf{Q}_2]\!]_G\,(T) = [\![\mathsf{Q}_1]\!]_G\,(T) \setminus [\![\mathsf{Q}_2]\!]_G\,(T)$$

5 A Few Known Discrepancies with the GQL Standard

In pursuing the goal of introducing the key features of GQL to the research community, we inevitably had to make decisions that resulted in discrepancies between our presentation and the 500+ pages of the forthcoming Standard. In this section, we discuss a non-exhaustive list of differences between the actual GQL Standard and our digest. To start with, in all our formal development we assumed that queries are given by their syntax trees, which result from parsing them. Hence we completely omitted such parsing-related aspects as parentheses, operator precedence etc. Also we note that many GQL features, even those described here, are optional, and not every implementation is obliged to have them all.

[3] Note that null is treated just as list()

The remaining discrepancies are divided into three main categories: syntactic restrictions (Section 5.1), query evaluation (Section 5.2), and missing features (Section 5.3). The reader must bear in mind that, as the GQL Standard is roughly one year from publication in its final form, many aspects of the language may still change in a way that depends on the work of the Committee, and thus is impossible to predict.

5.1 User-Friendly Syntactic Restrictions

The GQL Standard imposes restrictions on the syntax that aim at preventing unexpected behavior, and that we generally did not describe. Two such examples are given below.

First, consider the queries $Q_1 =$ `MATCH` μ `x=-[]->*` and $Q_2 =$ `MATCH` μ `x=-[]->*()` for some path mode μ (it does not matter which one). According to our semantics, both return one binding, namely $(x \mapsto \mathsf{path}(u))$, for each node u in the graph; however, Q_1 is syntactically forbidden in the GQL Standard because no node pattern occurs. Another interesting syntactic restriction concerns *strict interior* variables under selectors, such as `c` in the following:

```
MATCH ANY (:Person) -[]->* (c:Account) -[]->* (:Person),
      ANY (:Person) -[]->* (c:Account) -[]->* (:Person)
```

The `ANY` selectors are evaluated independently, and before the implicit join on variable `c`. Then, the node bound to the variable `c` by either path pattern is arbitrary, and joining on them is very likely to fail. This situation was not deemed user-friendly by the Committee, and therefore precluded.

5.2 Query Evaluation

Bag semantics. For simplicity, we described GQL as if it was following set semantics but, in reality, GQL uses bags just like Cypher and SQL. In order to define clauses and queries under bag semantics, small changes are needed:

- tables should be defined as bags, rather than sets, of bindings;
- unions ($\cup$) over the elements of a table should be additive bag unions ($\uplus$); and
- set comprehensions should be replaced with bag comprehensions.

As an example, if we denote bags with double curly braces, then the semantics of `RETURN` is

$$[\![\texttt{RETURN}\ \chi_1\ \texttt{AS}\ x_1, \ldots, \chi_\ell\ \texttt{AS}\ x_\ell]\!]_G\,(T) = \biguplus_{\mu \in T} \{\!\{(x_1 \mapsto [\![\chi_1]\!]_G^\mu, \ldots, x_\ell \mapsto [\![\chi_\ell]\!]_G^\mu)\}\!\}$$

Note that GQL partially eliminates duplicates during pattern matching, which is reflected here by the semantics of graph patterns: $[\![\Pi]\!]_G$ is a *set* of path/binding pairs, while $[\![\texttt{MATCH}\ \Pi]\!]_G$ returns a *bag* of bindings by projecting out the paths (see the definition of $[\![\texttt{MATCH}\ \Pi]\!]_G$ in Section 4.5). Hence, different ways to compute the same path/binding pair will only contribute to one copy of the binding in the output of $[\![\texttt{MATCH}\ \Pi]\!]_G$. It is still possible to get multiple copies of some binding in the output, but these come from pairs with different paths.

Partial deduplication is an effort to unify the multiplicies of queries that express *the same* pattern in different ways. To see this, consider the queries

Q_1: `MATCH (a:Person)-[]->(b WHERE b:Person OR b:Account)`
Q_2: `MATCH (a:Person)-[]->(b:Person) | (a)-[]->(b:Account)`

and the path (v_1, e_1, v_2) matched by either of them with the binding $\mu_1 = (a \rightarrow v_1, b \mapsto v_2)$, where v_2 bears *both* labels `Person` and `Account`. As the disjunction in Q_1 is expressed using a Boolean condition, this query always returns a single copy of μ_1. In Q_2, however, the disjunction is expressed with a union (`|`) of patterns; thus, if the semantics of `|` were defined as a bag-union, the query would return two copies of μ_1.

Finally, as in SQL, the operations `INTERSECT`, `UNION`, and `EXCEPT` remove duplicates in GQL, while the variants `INTERSECT ALL`, `UNION ALL`, and `EXCEPT ALL` do not.

Path bindings. In a nutshell, a *path binding* is a path where each element may be annotated with variables, and it is inconsistent as soon as two different elements have the same annotation (see [12] for details). Thus, a path binding defines a single path/binding pair, whereas a path/binding pair can define several path bindings. In GQL Standard, pattern matching computes a set of consistent path bindings, while our semantics computes a set of path/binding pairs, and the results are bags formed by projecting away paths. Consequently, our semantics might sometimes return fewer results than GQL's, but the difference only affects multiplicity. For example, consider `MATCH ()-[]->(a) | (a)-[]->()` on a graph with a single node u and a single (looping) edge. According to our semantics, only one copy of $(a \mapsto u)$ is returned, while two occurrences of it are returned according to GQL Standard.

Postponed evaluation of conditions. In our treatment of the language, the semantics of the following query is undefined:

```
MATCH ALL SHORTEST -[x]-> ( ()-[y]->() WHERE x.amount < y.amount ){10,10}
```

Indeed, when the condition `WHERE x.amount < y.amount` is evaluated, the variable `x` is not yet bound, as `-[x]->` occurs in a different branch of the query's syntax tree. In GQL Standard, however, the above query is legal, because the evaluation of `WHERE` conditions is postponed for as long as possible.[4] While the meaning of the query is clear, its evaluation is non-trivial. The context of each condition (here, `y` is bound to ten successive edge ids) must be recorded, because it will be different when the evaluation occurs. Note that the evaluation of conditions must occur before the evaluation of `SHORTEST`, hence queries like

```
MATCH -[x]->, ALL SHORTEST ( -[y]-> WHERE x.amount < y.amount ){10,10}
```

are not allowed in GQL.

Referencing the input table in conditions during pattern patching. In our semantics, the input table is not passed on to pattern matching, so one cannot refer to variables from it in `WHERE` conditions. As an example, the semantics of `LET x=42 MATCH (a WHERE a.amount=x)` is undefined. It is not yet clear whether such a query is allowed in the GQL Standard or not.

5.3 Missing Features

Syntactic sugar. The GQL Standard includes a lot of syntactic sugar that we disregarded. For instance, several other types of edge patterns exist, such as `-[`δ`]-`, which matches edges regardless of their direction. Another example is the possibility of using `*` and `+` as shorthands for `{0,}` and `{1,}`, respectively.

Complex label expressions. We only allow a single label in descriptors, but the GQL Standard allows complex label expressions, as in `MATCH (a:YachtClub|(Person&!Account))`. Using `WHERE`, this could be rewritten as

```
MATCH (a WHERE a:YachtClub OR (a:Person AND NOT a:Account))
```

Label expressions can also use the special atom "`%`" to check the nonemptyness of the label set. For example, `MATCH (a:%)` matches nodes with at least one label and `MATCH (a:!%)` matches node with no labels. Note that "`%`" cannot be used to define a regular expression of labels, unlike its usage in the `LIKE` expressions of SQL.

[4] This is orthogonal to left-to-right evaluation: `-[x]->` could be placed on the right instead.

Complex path modes. GQL allows more complex path modes than described here. Recall that `SHORTEST` partitions matched paths by endpoints and returns the shortest paths for each pair of endpoints. `SHORTEST` k `GROUPS` generalizes this: for each pair of endpoints, it returns all paths of length at most i_k, where $i_1 < i_2 < \cdots < i_k$ are the k smallest lengths of paths between these endpoints. `SHORTEST k PATHS` returns k shortest paths for each pair of endpoints. Another mode present in GQL is `SIMPLE`: it is similar to `ACYCLIC` but allows the first and the last node on a path to be the same, i.e., a simple cycle. There is also the keyword `WALK` to explicitly indicate the absence of a path mode.

GQL's `TRAIL` differs from Cypher's *trail semantics* [18, 17]. The latter corresponds to GQL's *match mode* `DIFFERENT EDGES`, which is omitted in this digest. Indeed, Cypher's requirement that all matched edges must be different operates at the level of graph patterns, whereas GQL's `TRAIL` operates at the level of path patterns. Hence, while the GQL query `MATCH TRAIL ()-[e1]->(), TRAIL ()-[e2]->()` will return bindings in which `e1` and `e2` are equal, the Cypher query `MATCH ()-[e1]>(), ()-[e2]->()` would not; the latter behaviour is captured by the GQL query `MATCH DIFFERENT EDGES ()-[e1]>(), ()-[e2]->()`.

Finally, we only use path modes at the beginning of path patterns. GQL's rules are more involved, in that they allow `TRAIL` and `ACYCLIC` to be used inside patterns.

Projection clauses. The GQL Standard includes several clauses similar to `RETURN`, such as `YIELD`, `PROJECT`, and `SELECT`. We ignored these because, although they are not allowed at the same positions in queries, they can be simulated by simple rewritings in terms of `RETURN`.

Combination of queries. In addition to set operations (`UNION`, etc.) and bag operations (`UNION ALL`, etc.), queries could be of the form Q_1 `OTHERWISE` Q_2. Its semantics is as follows: $[\![\mathsf{Q}_1 \texttt{ OTHERWISE } \mathsf{Q}_2]\!]\,(T)$ equals $[\![\mathsf{Q}_1]\!]\,(T)$ if table $[\![\mathsf{Q}_1]\!]\,(T)$ is non-empty, otherwise it equals $[\![\mathsf{Q}_2]\!]\,(T)$.

Aggregation. The GQL Standard will feature two kinds of aggregation. The first one, much like `GROUP BY` in SQL, groups together bindings under which the evaluation of an expression produces the same value, then an aggregate value is computed for each group. The exact details are still under development, but it appears likely that such aggregation will be limited to `RETURN` statements, thus having a very relational character.

The second kind will aggregate along matched paths to compute a value, both during and after pattern matching. Computing the length of a path is a typical example; one can have more complex aggregates, such as the sum of the values `n.amount` for each node `n` in the path. This is similar to `reduce` in Cypher. The use of this feature in pattern matching requires strong syntactic restrictions for query evaluation to be decidable [16].

Subqueries. GQL has a facility to run subqueries through the `CALL` Q clause, the semantics of which is roughly as follows: for each binding μ in the input table, $[\![Q]\!]_G\,(\{\mu\})$ is evaluated in a sub-process, and the resulting table is left-joined with the current working table. An important detail is that `CALL` can only expand bindings. It cannot remove columns from the input table nor change the values in them. The existence of read-only columns matters in clauses like `RETURN`, which cannot therefore be treated with our semantics as is. In GQL, this is handled with a notion of *working record.*

Note also that `CALL` Q will make nondeterminism much harder to detect if updates happen in Q. Tables are unordered sets (or bags) but in an update clause each binding causes changes in the graph (see next item) and so it can modify the evaluation of the clause for the next binding. In such cases, inconsistent changes may be detected [21].

Updates. Graph database updates in GQL are outside the scope of this paper. They will work similarly to Cypher updates [21], by using clauses that can add and remove elements (`INSERT` and `DELETE`), or modify elements' attributes (`SET` and `REMOVE`). Therefore, pattern matching and updates can be mixed together and result in bulk updates to the graph based on its contents, as in the example below:

```
MATCH  (a:Account)          -- match every Account a
INSERT (p:Person)           -- create a new Person node for each a
SET     p.name = a.owner    -- set the name of the new node
INSERT (p)-[:Owns]->(a)     -- create a new "Owns" edge from p to a
REMOVE  a.owner             -- remove the owner property from a
```

6 What the Future Holds

In this paper we have summarized the key elements of GQL, which is currently being developed as a new standard graph query language (the timeline of ISO calls for the publication of the Standard in either late 2023 or early 2024). At the time when the first version of the SQL Standard was produced, many key elements of relational theory were already in place. For GQL, the standardization work is well ahead of the academic developments it should ideally be based upon. In what follows, we bring to the attention of the community several directions of academic work that will facilitate the development of graph query languages and their standardization.

Expressiveness and complexity. For relational query languages, the database research community has uncovered a rich landscape of fragments (conjunctive queries, positive queries, and queries with inequalities are some very well studied examples) and extensions (for example, adding counting and aggregation, or adding recursion as in many instantiations of datalog), see [1, 4]. For these, we understand the trade-off between their expressiveness and the complexity of query evaluation. Here we have described a basic language for graphs, essentially the core of GQL, akin to relational algebra and calculus. Now we need to develop its theory, starting with understanding expressiveness and complexity and their trade-offs, in a way similar to what we know about relational databases. For the pattern matching facilities of GQL, shared with SQL/PGQ, some early results are available [16].

Query processing and optimization. Query processing and optimization is a central area in relational database research that needs yet to be developed for GQL. In a more theoretical level, the basis for understanding optimization is query equivalence and containment. We know a thing or two about containment for (conjunctive) regular path queries [9, 15] and extensions with data [26] but not for queries that resemble the real-life language. Moving to more practical aspects, one needs efficient and practical algorithms and data structures for processing graph queries in GQL, whether in a native system, or a relational implementation. Of course there is significant work in this direction [37, 23, 5, 28, 35, 31, 25] but it needs to be adjusted to languages that will dominate the practical landscape for decades.

Design decisions and alternatives. We explained in Remark 9 how GQL currently forbids concatenating patterns that contain different kinds of variables. Notice, however, that this current state reflects a design decision and it may be interesting to explore other avenues for graph query languages. For instance, one could consider a semantics in which both occurrences of `x` in Remark 9 should be bound to single nodes. Under such a semantics, the pattern would essentially perform a join on the even nodes of the path and would match "flower" shaped paths centered around node `x`, consisting of Transfer-loops of length two. Alternatively, one could consider a semantics in which, as soon as `x` occurs as a group variable, all occurrences of `x` are considered to be group variable occurrences. In this case, the query would match Transfer-paths of even length and bind `x` to the list of "even" nodes on such paths. In line with this work would be the study of an automaton model with group variables that would allow classical evaluation and automata-theoretic constructions such as the product, determinization, etc. Since GQL is a complex language, there are many such places in which fundamental research can either help to validate the current design decisions or propose alternatives.

Updates. We have concentrated on the read-only part of the languages and have not touched updates. Designing a proper update language is not a simple task: in Cypher, for example, the initial design exhibited a multitude of problems [21]. GQL largely follows Cypher, which means its updates and transaction processing facilities need to be designed with care and subjected to the same research scrutiny as their relational counterpart.

Graph-to-graph queries. GQL, as its precursors including Cypher, is a very good tool for turning graphs into relations. The ever reappearing issue in the field of graph languages is how to design a graph-to-graph language whose queries output graphs. Queries are then composable: a query can be applied to the output of a previous one. We also regain such basic concepts as views and subqueries, taken for granted in relational databases, but very limited in the current graph database landscape.

Metadata. Looking into the future, we need to have a good schema language for graphs, and see how it interacts with graph query languages. Some efforts in this direction have already been made: for example, the PG-Keys proposal introduces keys for property graphs [3] and more recently proposed PG-Schema [2] specifies a schema language for property graphs that should lead to future schema standards. As these are formulated, much theory needs to be developed, for example semantic query optimization, as well as incremental validation of schemas and constraints following work for relational and semistructured data [24, 6].

References

1 Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases.* Addison-Wesley, 1995.

2 Renzo Angles, Angela Bonifati, Stefania Dumbrava, George Fletcher, Alastair Green, Jan Hidders, Bei Li, Leonid Libkin, Victor Marsault, Wim Martens, Filip Murlak, Stefan Plantikow, Ognjen Savkovic, Michael Schmidt, Juan Sequeda, Slawek Staworko, Dominik Tomaszuk, Hannes Voigt, Domagoj Vrgoc, Mingxi Wu, and Dusan Zivkovic. PG-Schema: Schemas for property graphs, 2022. `arXiv:2211.10962`.

3 Renzo Angles, Angela Bonifati, Stefania Dumbrava, George Fletcher, Keith W. Hare, Jan Hidders, Victor E. Lee, Bei Li, Leonid Libkin, Wim Martens, Filip Murlak, Josh Perryman, Ognjen Savkovic, Michael Schmidt, Juan F. Sequeda, Slawek Staworko, and Dominik Tomaszuk. PG-Keys: Keys for property graphs. In *SIGMOD '21: International Conference on Management of Data*, pages 2423–2436. ACM, 2021. `doi:10.1145/3448016.3457561`.

4 Marcelo Arenas, Pablo Barceló, Leonid Libkin, Wim Martens, and Andreas Pieris. *Database Theory.* Open source at `https://github.com/pdm-book/community`, 2022.

5 Jorge A. Baier, Dietrich Daroch, Juan L. Reutter, and Domagoj Vrgoč. Evaluating navigational RDF queries over the web. In *HT*, pages 165–174. ACM, 2017. `doi:10.1145/3078714.3078731`.

6 Denilson Barbosa, Alberto O. Mendelzon, Leonid Libkin, Laurent Mignet, and Marcelo Arenas. Efficient incremental validation of XML documents. In *ICDE*, pages 671–682. IEEE Computer Society, 2004. doi:10.1109/ICDE.2004.1320036.

7 Véronique Benzaken and Evelyne Contejean. A coq mechanised formal semantics for realistic SQL queries: formally reconciling SQL and bag relational algebra. In *CPP*, pages 249–261. ACM, 2019. doi:10.1145/3293880.3294107.

8 Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Moshe Y. Vardi. Containment of conjunctive regular path queries with inverse. In *KR*, pages 176–185. Morgan Kaufmann, 2000.

9 Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Moshe Y. Vardi. Reasoning on regular path queries. *SIGMOD Rec.*, 32(4):83–92, 2003. doi:10.1145/959060.959076.

10 Shumo Chu, Konstantin Weitz, Alvin Cheung, and Dan Suciu. HoTTSQL: proving query rewrites with univalent SQL semantics. In *PLDI*, pages 510–524. ACM, 2017. doi:10.1145/3062341.3062348.

11 Isabel F. Cruz, Alberto O. Mendelzon, and Peter T. Wood. A graphical query language supporting recursion. In *SIGMOD Conference*, pages 323–330. ACM Press, 1987. doi:10.1145/38713.38749.

12 Alin Deutsch, Nadime Francis, Alastair Green, Keith Hare, Bei Li, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Wim Martens, Jan Michels, Filip Murlak, Stefan Plantikow, Petra Selmer, Oskar van Rest, Hannes Voigt, Domagoj Vrgoc, Mingxi Wu, and Fred Zemke. Graph pattern matching in GQL and SQL/PGQ. In *SIGMOD Conference*, pages 2246–2258. ACM, 2022. doi:10.1145/3514221.3526057.

13 Alin Deutsch, Yu Xu, Mingxi Wu, and Victor E. Lee. Aggregation support for modern graph analytics in TigerGraph. In *SIGMOD Conference*, pages 377–392. ACM, 2020. doi:10.1145/3318464.3386144.

14 Mary F. Fernandez, Daniela Florescu, Alon Y. Levy, and Dan Suciu. A query language for a web-site management system. *SIGMOD Rec.*, 26(3):4–11, 1997. doi:10.1145/262762.262763.

15 Diego Figueira, Adwait Godbole, Shankara Narayanan Krishna, Wim Martens, Matthias Niewerth, and Tina Trautner. Containment of simple conjunctive regular path queries. In *KR*, pages 371–380, 2020. doi:10.24963/kr.2020/38.

16 Nadime Francis, Amélie Gheerbrant, Paolo Guagliardo, Leonid Libkin, Victor Marsault, Wim Martens, Filip Murlak, Liat Peterfreund, Alexandra Rogova, and Domagoj Vrgoč. GPC: A pattern calculus for property graphs. In *PODS'23*, 2023. To appear.

17 Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Martin Schuster, Petra Selmer, and Andrés Taylor. Formal semantics of the language Cypher, 2018. arXiv:1802.09984.

18 Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. Cypher: An evolving query language for property graphs. In *SIGMOD Conference*, pages 1433–1445. ACM, 2018. doi:10.1145/3183713.3190657.

19 GQL influence graph. https://www.gqlstandards.org/existing-languages, 2023. Accessed: 2023-01-17.

20 Alastair Green, Paolo Guagliardo, and Leonid Libkin. Property graphs and paths in GQL: Mathematical definitions. Technical Reports TR-2021-01, Linked Data Benchmark Council (LDBC), October 2021. doi:10.54285/ldbc.TZJP7279.

21 Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Martin Schuster, Petra Selmer, and Hannes Voigt. Updating graph databases with Cypher. *Proc. VLDB Endow.*, 12(12):2242–2253, 2019. doi:10.14778/3352063.3352139.

22 Paolo Guagliardo and Leonid Libkin. A formal semantics of SQL queries, its validation, and applications. *Proc. VLDB Endow.*, 11(1):27–39, 2017. doi:10.14778/3151113.3151116.

23 Andrey Gubichev, Srikanta J. Bedathur, and Stephan Seufert. Sparqling Kleene: Fast property paths in RDF-3X. In *GRADES*. CWI/ACM, 2013. doi:10.1145/2484425.2484443.

24 A. Gupta and I.S. Mumick. *Materialized Views: Techniques, Implementations, and Applications.* MIT Press, 1999.

25 Aidan Hogan, Cristian Riveros, Carlos Rojas, and Adrián Soto. A worst-case optimal join algorithm for SPARQL. In *ISWC (1)*, volume 11778 of *Lecture Notes in Computer Science*, pages 258–275. Springer, 2019. `doi:10.1007/978-3-030-30793-6_15`.

26 Egor V. Kostylev, Juan L. Reutter, and Domagoj Vrgoc. Containment of queries for graphs with data. *J. Comput. Syst. Sci.*, 92:65–91, 2018. `doi:10.1016/j.jcss.2017.09.005`.

27 Leonid Libkin, Wim Martens, and Domagoj Vrgoč. Querying graphs with data. *Journal of the ACM*, 63(2):14:1–14:53, 2016. `doi:10.1145/2850413`.

28 Wim Martens, Matthias Niewerth, Tina Popp, Stijn Vansummeren, and Domagoj Vrgoč. Representing paths in graph database pattern matching, 2022. `arXiv:2207.13541`.

29 Alberto O. Mendelzon, George A. Mihaila, and Tova Milo. Querying the world wide web. In *Proceedings of the Fourth International Conference on Parallel and Distributed Information Systems, December 18-20, 1996, Miami Beach, Florida, USA*, pages 80–91. IEEE Computer Society, 1996. `doi:10.1109/PDIS.1996.568671`.

30 Alberto O. Mendelzon and Peter T. Wood. Finding regular simple paths in graph databases. *SIAM J. Comput.*, 24(6):1235–1258, 1995. `doi:10.1137/S009753979122370X`.

31 Dung T. Nguyen, Molham Aref, Martin Bravenboer, George Kollias, Hung Q. Ngo, Christopher Ré, and Atri Rudra. Join processing for graph patterns: An old dog with new tricks. In *GRADES*, pages 2:1–2:8. ACM, 2015. `doi:10.1145/2764947.2764948`.

32 Juan L. Reutter, Miguel Romero, and Moshe Y. Vardi. Regular queries on graph databases. *Theory Comput. Syst.*, 61(1):31–83, 2017. `doi:10.1007/s00224-016-9676-2`.

33 Marko A. Rodriguez. The Gremlin graph traversal machine and language. In *DBPL*, pages 1–10. ACM, 2015. `doi:10.1145/2815072.2815073`.

34 Oskar van Rest, Sungpack Hong, Jinha Kim, Xuming Meng, and Hassan Chafi. PGQL: a property graph query language. In *GRADES*, page 7. ACM, 2016. `doi:10.1145/2960414.2960421`.

35 Domagoj Vrgoč. Evaluating regular path queries under the all-shortest paths semantics, 2022. `arXiv:2204.11137`.

36 Wikipedia contributors. GQL graph query language, 2020. URL: `https://en.wikipedia.org/wiki/GQL_Graph_Query_Language`.

37 Nikolay Yakovets, Parke Godfrey, and Jarek Gryz. Query planning for evaluating SPARQL property paths. In *SIGMOD Conference*, pages 1875–1889. ACM, 2016. `doi:10.1145/2882903.2882944`.

Compact Data Structures Meet Databases

Gonzalo Navarro ✉
Millennium Institute for Foundational Research on Data (IMFD), Santiago, Chile
Department of Computer Science, University of Chile, Santiago, Chile

Abstract

We describe two success stories on the application of compact data structures (cds) to solve the problem of the excessively redundant space requirements posed by worst-case-optimal (wco) algorithms for multijoins in databases, and particularly basic graph patterns on graph databases. The aim of cds is to represent the data *and* additional data structures on it, using total space close to that of the plain (and, sometimes, compressed) data, while efficiently simulating the data structure operations. Cds turn out to be a perfect approach for the described problem: We designed and implemented cds that effectively use space close to that of the plain or compressed data, which is orders of magnitude less than existing systems, while retaining worst-case optimality and performing competitively with those systems in query time, sometimes being even considerably faster.

2012 ACM Subject Classification Information systems → Data structures; Theory of computation → Data structures design and analysis

Keywords and phrases succinct data structures, tries, multidimensional grids, text searching

Digital Object Identifier 10.4230/LIPIcs.ICDT.2023.2

Category Invited Talk

Funding Supported by ANID – Millennium Science Initiative Program – Code ICN17_002, Chile.

1 Motivation

1.1 Graph databases

Graph databases [48, 26] have gained momentum with the rise of large unstructured repositories of information that emphasize relations between entities. They have become an attractive alternative to the relational model in cases where the information has no fixed structure. Dozens of graph database management systems [41, 51, 12, 36], prototypes [1, 35, 29, 2], models and languages [25, 17, 27, 3], and large repositories like Wikidata [54], illustrate how active is the interest on this relatively new technology.

A graph database represents information in the form of a labeled graph or network. There are many possible models to represent information in this way, such as knowledge graphs [27], property graphs [17], and RDF [34], to name a few. In general, the graph nodes represent objects and the edges between them represent relations. The models differ on what kind of information can be associated with the nodes or the edges, whether the edge labels can also be objects, and so on. For concreteness, we will focus on the RDF model, where the graph is seen as a set of *triples* (s, p, o), where s is the *subject* (or source node), p is the *predicate* (or label of the edge), and o is the *object* (or target node). Consider the graph of Figure 1 (cf. [4]) as our running example. The nodes are scientists and the Nobel prize. The arrows indicate that a scientist advised another and that a scientist won the Nobel prize.

The language to query a graph database also varies. In the widely used SPARQL standard [25], queries are built on relatively small *graph patterns*, which have to be matched in the database graph. In its simplest form, this can be just a *triple pattern*, which searches for the existence of a certain edge (or triple). The triple pattern specifies constants or variables for the subject, predicate, and object of the desired triples; every matching triple in

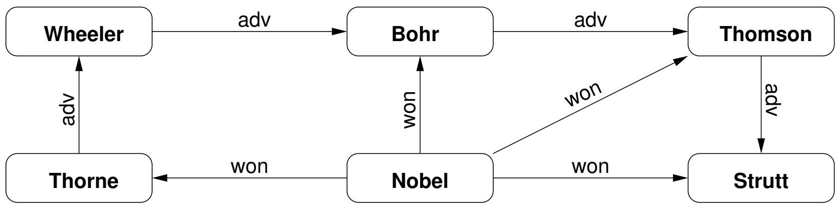


Figure 1 An example labeled graph.

the graph corresponds to *binding* the variables of the triple pattern. In our example, the triple $(\mathsf{Nobel}, \mathsf{won}, ?x)$ returns all the bindings of x to Nobel prize winners (i.e., $x = \mathsf{Thorne}$, $x = \mathsf{Bohr}$, etc.).

Basic graph patterns (BGPs) are sets of triple patterns sharing variables. They correspond to matching a subgraph in the database, returning all the variable bindings that make the subgraph occur. BGPs are akin to *multijoins* in relational databases, or *full conjunctive queries* in logic databases. In our example, the BGP

$$(?y, \mathsf{adv}, ?x),\ (\mathsf{Nobel}, \mathsf{won}, ?x),\ (\mathsf{Nobel}, \mathsf{won}, ?y) \tag{1}$$

returns pairs (y, x) where y advised x and both won the Nobel prize (those are $(y, x) = (\mathsf{Bohr}, \mathsf{Thompson})$ and $(y, x) = (\mathsf{Thompson}, \mathsf{Strutt})$).

The third type of graph pattern are the *regular path queries (RPQs)*. An RPQ is basically a regular expression that matches variable-length paths in the database graph, so that the sequence of traversed labels belong to the language denoted by the regular expression. RPQs are distinctive of graph databases and cannot be emulated in the relational algebra. In our example, the RPQ “$\mathsf{Wheeler}\ \mathsf{adv}^+\ ?x$” retrieves the academic descent of Wheeler (i.e., $x = \mathsf{Bohr}$, $x = \mathsf{Thomson}$, and $x = \mathsf{Strutt}$).

1.2 Worst-case optimality and the space problem

While triple patterns are easily solved with plain data retrieval structures, BGPs and RPQs are much more challenging and pose serious performance issues on graph database engines (e.g., it is typical to set timeouts in the minutes). Join evaluation is the most costly part in relational queries, and this carries over graph databases, where in addition it is not strange to see BGPs joining tens of triple patterns (e.g., up to 22 were found in a Wikidata query log [33]). Languages like SPARQL also enable projections, unions, and other operations, though the efficiency focus of database engines is generally on BGPs and RPQs.

An important breakthrough in the resolution of multijoin queries was the development of *worst-case optimal (wco)* join algorithms. A join algorithm is wco if its time complexity is of the order of the so-called *AGM bound* [7], that is, the maximum possible output of the query over some database with the same table attributes and sizes of the one at hand. It was shown that the techniques used by relational engines since the sixties, where multijoins were performed pairwise, were doomed to be non-wco. At the same time, several wco algorithms were developed [43, 44, 53, 31, 45, 42]. This technique was translated to graph databases [29], where it is particularly relevant because multijoins tend to be large and complex [45, 1, 30, 29]. It was shown that wco algorithms considerably outperformed traditional join algorithms on complex queries, especially when the BGPs contained cycles [1].

This improvement came at the cost of space, however. For example, the most popular wco algorithm, *Leapfrog Triejoin (LTJ)* [53], requires to index the rows of every database table as sequences of values in trie data structures, *in every possible ordering of the attributes.*

That is, a table of d columns needs to be stored in $d!$ tries. In particular, supporting wco joins on triples (s, p, o) poses a space overhead factor of $3! = 6$. Other wco algorithms pose similar or worse space problems. This is particularly unfortunate with the emergence of enormous repositories of unstructured data in graph form, and hinders the adoption of the faster wco strategies in the resolution of complex multijoin queries. To illustrate, Wikidata is approaching 14 billion triples,[1] so 6 copies of it, using just 32 bits per element and without the additional trie structures, surpasses the terabyte.

1.3 Compact data structures to the rescue

Our recent research has shown that the use of *compact data structures (cds)* can play a significant role in the reduction of the space required by wco multijoin algorithms. Compact data structures [38] aim to represent the data *and* its needed data structures within space close to the *entropy*, or amount of information, present in the data. There exist to date a number of compact representations for bit vectors, sequences, trees, graphs, matrices, point grids, texts, and many others. Cds have been very successful in reducing the size of relevant data structures by orders of magnitude, as well as greatly increasing the functionality of data representations within space close to the actual information of the data.

It is then more than natural to apply cds to the problem of supporting wco algorithms on graph databases, with the aim of retaining time optimality while removing the redundancy. We have recently proved the viability of this concept in two forms.

Qdags [40, 6]: We represented relations of d attributes as d-dimensional *point grids* called *qdags*, where every tuple becomes a point (qdags are a kind of compressed quadtrees [49, 50]). To solve a multijoin between several tables, qdags traverse and intersect all their grids in synchronization. The resulting algorithm not only was proved to be wco and to require only *one* copy of the data independently of d, but it was also shown to be competitive in time with the state of the art (at least for low d; the query time is exponential on d). Qdags use orders of magnitude less space than other indices on graph databases, actually *compressing* the graph to less than its plain size. As they are compositional (i.e., the result of a query is also a qdag) we also showed how to extend their functionality to the full relational algebra.

Ring [4, 5]: We represented the database triples as *texts*, and built on text indexing cds [14, 16] to support the LTJ algorithm. The resulting structure, the *ring*, is once again wco and uses just one copy of the data. In many cases (but not when d is very small), the ring is faster than qdags, but it requires $O(2^d)$ (not $d!$ as classical approaches) copies on d-dimensional tables, though one suffices no graph databases, where $d = 3$. Both qdags and the ring could index the Wikidata in under 70 GB. In a further development [5], we solved RPQs on the ring in time competitive with the state of the art, while using an order of magnitude less space than other indices. This solution uses techniques from text searching, like converting the RPQ to its Glushkov automaton [23, 39] and exploiting the flexibility of the wavelet trees [24, 37] used by the ring to represents its text.

Our results demonstrate that cds can be used to compactly represent graph databases while efficiently solving BGPs and RPQs. In this survey we describe those results in some detail and discuss the features and challenges of this new and promising technology. We strive for simplicity and informality in this introductory survey, further details and precisions can be found in the references.

[1] `https://grafana.wikimedia.org/d/000000489/wikidata-query-service`

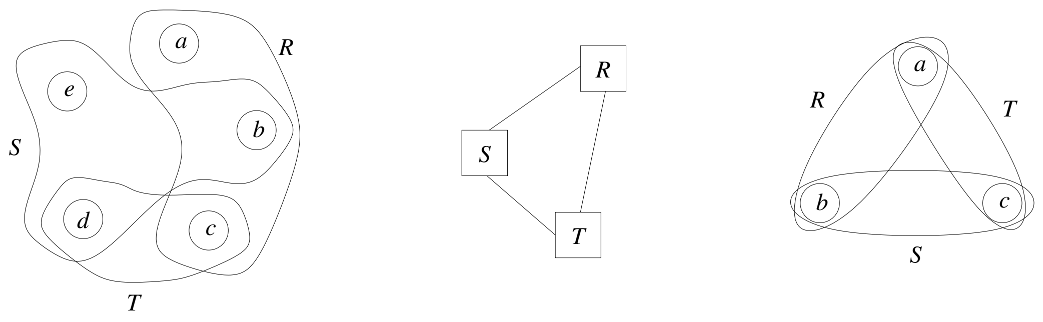


Figure 2 On the left, a join query seen as a hypergraph where nodes are attributes. In the middle, as a graph where nodes are relations. On the right, the triangle query on the graph where nodes are attributes.

2 State of the Art in (Graph) Databases

2.1 Multijoin queries

We focus on so-called *multijoin* queries, which compute the natural join between a set of tables (we discuss the case of graph databases soon). It is customary to regard multijoin queries as hypergraphs, where the nodes are attributes and the involved relations are hyperedges covering their attribute nodes. For example, the left hypergraph in Figure 2 corresponds to the join $R(a, b, c) \bowtie S(b, d, e) \bowtie T(c, d)$.

An alternative view, shown in the middle of the figure, is to regard the relations as nodes and put edges between relations that share attributes. We speak of *cyclic* and *acyclic* queries referring to this second kind of graph.

2.2 The AGM bound

Asterias, Grohe, and Marx [7] showed how to compute the maximum possible size of the output of a multijoin query. The maximization is done over every possible content of the tables participating in the join, while retaining their attributes and sizes. This bound, also shown to be tight, is since then known as the *AGM bound.* The bound immediately yields the concept of worst-case optimality: a join algorithm is *worst-case optimal (wco)* if the time it takes to compute a join is of the order of the AGM bound (possibly multiplied by a factor that depends at most polylogarithmically on the data size), because that is the possible output size for this query on *some* tables, and we need at least time to write the output. This differs from the stricter *instance optimal* algoritms, which take time proportional to the output of the query on *the given* tables.

The precise form of the bound is not important for our discussion; what is relevant for now is that the bound implies that *no* pair-wise join strategy can be wco. The paradigmatic example is the so-called "triangle query" $R(a, b) \bowtie S(b, c) \bowtie T(c, a)$ (see the right of Figure 2). If the tables have n rows, the triangle query can produce only $O(n^{3/2})$ results; however there exist tables R, S and T where every pairwise join strategy takes time $\Omega(n^2)$. It is worth noting that all the classical work on query plan optimization since the birth of the relational model built on pair-wise joins.

A number of wco algorithms appeared since then [43, 44, 53, 31, 45, 42]; we describe the most popular one in some detail next. For the particular case of acyclic queries, it is indeed possible to obtain the famous instance-optimal Yannakakis' algorithm [57]. It is also possible to obtain intermediate measures, like the *fractional hypertreewidth (fhw)*, which is related to the strategy of separating a cyclic query into a tree of cyclic components, solving each of those with a wco algorithm, and then solving the resulting acyclic query instance-optimally [1] (we omit some details for simplicity). The fhw bound is then the sum of the AGM bounds for the nodes in the best possible decomposition of the query into a tree of cycles.

2.3 Leapfrog Triejoin

Leapfrog Triejoin (LTJ) [53] is the most popular wco multijoin algorithm. Instead of performing the joins pair-wise (or, we could say, table-wise), LTJ proceeds *attribute-wise* over all the tables at the same time. We say that LTJ *binds* one attribute at a time, meaning that it finds all the possible values it may get in the output. Say that we decide to start by binding attribute A. We then find the values of A that appear in *all* the joined tables. For each such value $A = a$, we run a branch where the tables keep only their rows where A has value a, and continue binding the next variables. This branching continues until either there are no binding values for an attribute (and thus the current branch is abandoned), or we have bound all the attributes (and then we output all the possible combinations of the non-joined attributes, as a Cartesian product). We show an example soon.

For LTJ to run efficiently, it is convenient to arrange the tables as *tries* [19], or digital trees. Each row of the table becomes a root-to-leaf path in its trie. The order in which the attributes are read root to leaf must correspond to the order in which they are bound along the query process, and the attributes not participating in the join must come at the end. Each branch of LTJ then starts with a pointer to a node of the trie of each joined relation (all start at the root). When it comes to bind A, all the relations having attribute A intersect the children of their current node. For each value a that appears in the children of all the relevant trie nodes, LTJ descends to that child in all those tries and continues by that branch, where now we have bound $A = a$.

Because one cannot predict which attributes will be joined in queries, and furthermore it is convenient to choose different binding orders to improve performance, LTJ requires that each relation with d attributes is indexed in $d!$ tries, one per possible attribute ordering. This is the main problem for using LTJ in practice. An interesting alternative is to build query plans that combine wco algorithms with (non-wco) pairwise joins [35, 20].

2.4 The case of graph databases

A graph database can be seen as a single relation over three attributes; every edge $s \xrightarrow{p} o$ is interpreted as a tuple (s, p, o) in the relation (for subject, predicate, and object, following the RDF terminology [34]). Alternatively, it can be seen as a set of relations over two attributes, one per predicate p containing the pairs (s, o) such that the edge $s \xrightarrow{p} o$ is in the graph.

Standard query languages like SPARQL and many others feature two core query types, *Basic Graph Patterns (BGPs)* and *Regular Path Queries (RPQs)*.

Basic graph patterns. BGPs can be seen as a composition of a selection and a join in the corresponding relational database. A BGP is a set of *triple patterns*, each describing a graph edge where each of the components s, p, and o can be a fixed constant (hence the selection) or a variable. Shared variables among the triple patterns of the BGP correspond to equijoins by the corresponding attributes. When the predicates are constant, we can see the hypergraph of the query as a classic labeled digraph; we can support variable predicates by allowing labels be variables too. With this modelling, solving the BGP query corresponds

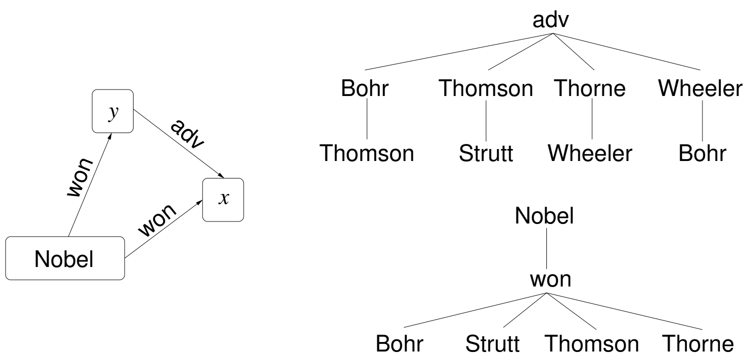


Figure 3 On the left, the subgraph of Eq. (1). On the right, the tries to traverse when solving this query using LTJ.

exactly to finding all the bindings of the variables that make that graph be a subgraph of the database. On the left of Figure 3 we see the graph of the BGP of Eq. (1); note it is analogous to the triangle query.

By regarding every triple pattern *as a relation*, where some attributes may by bound from the beginning (if they are constants) or be named after a variable otherwise, we can adapt LTJ to solve BGPs, resulting in a wco algorithm as well [29]. The relevant parts of the tries for our example query, in the correct order to bind first y and then x, are shown on the right of Figure 3 (we use the first trie for $(?y, \mathsf{adv}, ?x)$, starting at the node "adv", and two copies of the second trie for $(\mathsf{Nobel}, \mathsf{won}, ?x)$ and $(\mathsf{Nobel}, \mathsf{won}, ?y)$, starting at the node "won"). When we bind y, we intersect the lists of children of both nodes, obtaining bindings $y = \mathsf{Bohr}$, $y = \mathsf{Thomson}$, and $y = \mathsf{Thorne}$. Branching with each such value of y we intersect the only child of each node in the first trie with the children of "won" in the second, obtaining $x = \mathsf{Thomson}$ when $y = \mathsf{Bohr}$, and $x = \mathsf{Strutt}$ when $y = \mathsf{Thomson}$.

The space issues of LTJ carry over the graph database formulation, so we require to store $3! = 6$ tries, each representing the whole database graph in a different order, (s, p, o), (s, o, p), (o, s, p), (o, p, s), (p, o, s), and (p, s, o). Alternatives are giving up with wco algorithms, as mentioned, or building some orders at query time, which is generally too expensive.

Regular path queries. RPQs are akin to regular expressions that must be matched to paths in the database graph. They may fix the starting node x and/or the ending node y, and otherwise they specify the language of the sequences of labels that can connect x with y. Apart from the regular expression operations, one can use $\hat{}p$ to denote an edge labeled p in reverse direction. This can be handled by duplicating the graph database edges so as to include those reversed labels.

There are no wco algorithms for RPQs. The standard solution is to build the *product graph* between the graph database and the nondeterministic finite automaton (NFA) of the RPQ. The product graph has nodes (u, v) for each node u of the graph and v of the NFA. There is an edge $(u, v) \xrightarrow{p} (u', v')$ iff there is an edge $u \xrightarrow{p} u'$ in the graph and we can go from v to v' by symbol p in the NFA. We then traverse the product automaton from every possible initial node (x, i) (where x may be fixed or not in the RPQ and i is the initial NFA state) towards every possible final node (y, f) (where y may be fixed or not in the RPQ and f is a final NFA state) and report all the pairs (x, y).

Several heuristics have been proposed over this basic solution [32, 56, 46], aiming at filtering the traversal of the product graph. For example, if the RPQ forces the existence of a certain label in the path that is infrequent in the graph, then it is more convenient to focus on those edges and trying to match the RPQ path in both directions from the arrow.

An elegant way to mix BGPs and RPQs is to permit triple patterns of the form (x, R, y), where x and y are the endpoints of the RPQ R. This can then be treated as just another relation to join. For example, one can run the RPQ and materialize the output, and then run the query as a normal BGP. Or one can run the rest of the BGP so that these triples, which are likely to be more expensive, are processed at the end, only for the bound variables that have survived all the intersections.

3 Compact Data Structures

We describe in this section the compact data structures we used in our developments. Again, we aim at an intuitive description; more details and references can be found elsewhere [38].

3.1 Bitvectors

A bitvector $B[1..n]$ is a sequence of n bits that provides the following two operations:
$\boldsymbol{rank_b(B, i)}$ is the number of bits equal to $b \in \{0, 1\}$ in $B[1..i]$.
$\boldsymbol{select_b(B, j)}$ is the position of the jth occurrence of $b \in \{0, 1\}$ in B.

It is possible to represent B within $n + o(n)$ bits so that both operations are supported in constant time [11]. It is also possible to represent B in compressed space when it has many more or fewer 0s than 1s, while retaining constant-time operations. Let m be the number of 0s, then the compressed representation uses $\log_2 \binom{n}{m} + o(n)$ bits [47].

3.2 Cardinal trees

A cardinal tree is a rooted tree where each node has children with labels in $[1..\sigma]$; each node has at most one child with a given label. The basic operations supported by this data structure are:
$\boldsymbol{root(T)}$ is the root node of T.
$\boldsymbol{child(v, a)}$ is the child of node v labeled a, or *null* if there is no such child.
$\boldsymbol{parent(v)}$ is the parent of node v, or *null* if v is the root.

It is possible to represent a cardinal tree with n nodes within $(\log_2 \sigma + 2 + o(1))n$ bits, while supporting the given operations in constant time [9]. We are going to use tries of alphabet size $\sigma = 4$, in which case a more practical representation using the same space is the k^2-tree [10] (with $k = 2$). It represents each node with 4 bits, which marks which children exist. The 4-bit signatures of all the nodes are concatenated in levelwise form, into a large bitvector $T[1..]$. The node identifiers correspond to the index of their signatures in this array. The root identifier is 0, corresponding to the first signature, and the identifier of the ith child of a node with identifier v is $child(v, i) = rank_1(T, 4v + i)$. The identifier of the parent of v, instead, is $parent(v) = \lceil select(T, v)/4 \rceil - 1$.

3.3 Quadtrees

A quadtree is a geometric data structure that represents points in a discrete two-dimensional grid. The quadtree is a tree of arity four. The root represents the whole grid, which is divided as evenly as possible into four quadrants. Each quadrant is recursively represented by a child of the root: top-left, top-right, bottom-left, and bottom-right. If the grid has no points, the corresponding quadtree node is a leaf and the grid is not further subdivided. When the nodes represent single cells, they also become leaves that store a point or not.

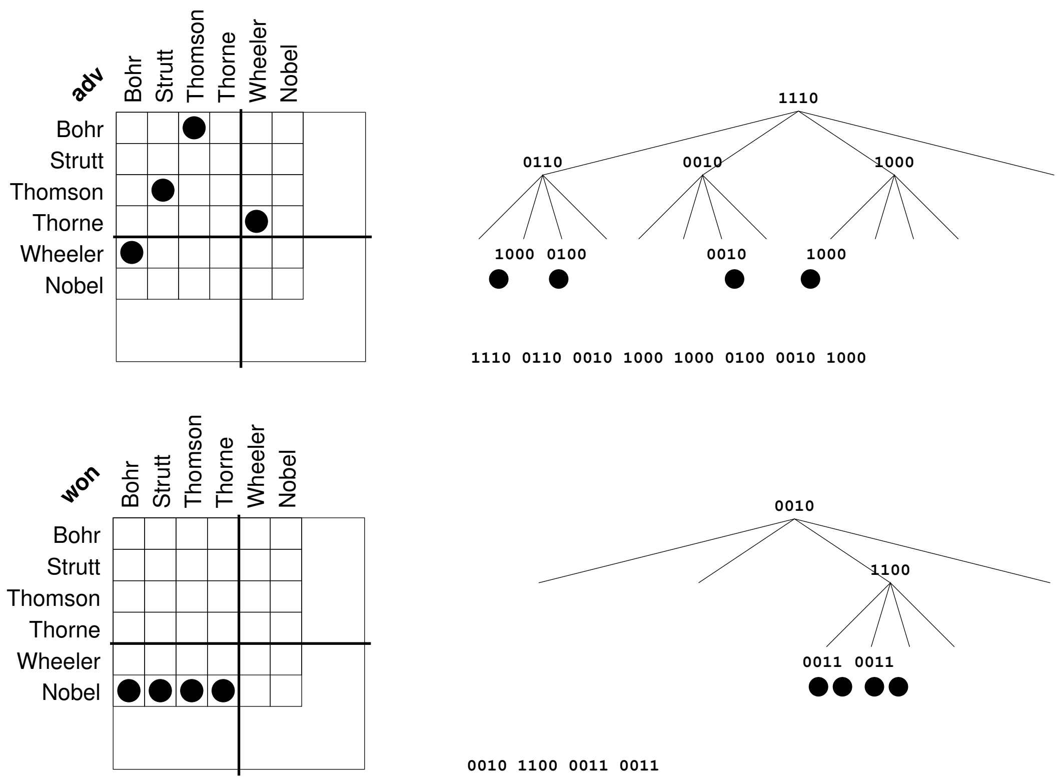


Figure 4 On the left, the relations adv and won of Figure 1 seen as two-dimensional grids. On the right, their representations as quadtrees, which are just cardinal trees of arity four (we show the signatures of the tree nodes), and their final representation as k^2-tree bitvectors at their bottom.

A compact representation of a quadtree can be obtained by seeing it as a cardinal tree of arity $\sigma = 4$. Each grid point then corresponds to a root-to-leaf path of length $\ell = \lceil \log_4(u^2) \rceil = \lceil \log_2 u \rceil$, on a $u \times u$ grid. Since all the paths in this trie are of the same length, we do not need to store information on the children of the nodes of depth ℓ.

Figure 4 shows how our two predicates adv and won can be regarded as two-dimensional grids (as done with qdags). We also show how those grids are represented as quadtrees, which in turn are seen as cardinal trees of $\sigma = 4$ children. Their final concrete representation, as k^2-trees, is just a sequence of bits.

The space of this representation is, in the worst case, $4 \log_2 u$ bits per point, which is twice the $2 \log_2 u$ bits needed by a representation as pairs of coordinates. When the points have some regularity, like clustering, the space decreases, as shown in the figure for relation won. Within this space, the quadtree can efficiently search for points.

3.4 Wavelet trees

A *wavelet tree* [24, 37] represents a sequence $S[1..n]$ over an alphabet $[1..\sigma]$ using $n \log_2 \sigma + o(n \log \sigma)$ bits, so that several interesting queries can be answered, including the following ones in $O(\log \sigma)$ time:

$\boldsymbol{access(S,i)}$ returns $S[i]$.

$\boldsymbol{rank_a(S,i)}$ is the number of symbols equal to $a \in [1..\sigma]$ in $S[1..i]$.

$\boldsymbol{select_a(S,j)}$ is the position of the jth occurrence of $a \in [1..\sigma]$ in S.

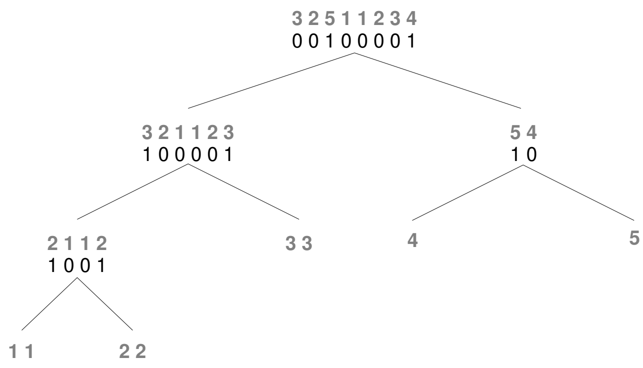


Figure 5 The wavelet tree of $S = 32511234$. Each node v shows in gray the string S_v it represents (but does not store) and below it the bitvector B_v it stores.

The wavelet tree is a balanced binary tree with σ leaves, where each node handles a range of the alphabet; the root represents $[1..\sigma]$ and each leaf represents one symbol. If an internal node v represents range $[a..b]$, then its left child represents $[a..m]$ and its right child represents $[m+1..b]$, with $m = \lfloor (a+b)/2 \rfloor$. The node v represents, virtually, the subsequence S_v of S with symbols in $[a..b]$, but it only stores a bitvector B_v of length $|S_v|$, where $B_v[i] = 0$ if $S_v[i]$ belongs to the left child of v, and $B_v[i] = 1$ otherwise. Figure 5 shows an example.

The wavelet tree has height $\lceil \log_2 \sigma \rceil$. At each level, it stores exactly n bits because each position of S is in exactly one node at that level. By representing those bitvectors so that they answer *rank* and *select* in constant time (Section 3.1), the space is $n + o(n)$ bits per level and $n \log_2 \sigma + o(n \log \sigma)$ overall. Note that a plain representation of S requires $n \log_2 \sigma$ bits, and it can be less if we use compressed bitvectors or give the tree a Huffman shape.

It is not hard to see how to support the basic operations in $O(\log \sigma)$ time, with a top-down or bottom-up traversal on the wavelet tree. For example, to compute $S[i]$, we start with v being the root. If $B_v[i] = 0$, we move to its left child with $i := rank_0(B_v, i)$, otherwise we move to its right child with $i := rank_1(B_v, i)$. When we arrive at a leaf, its symbol is $S[i]$. Wavelet trees can perform more complex operations on S; we will mention them as needed.

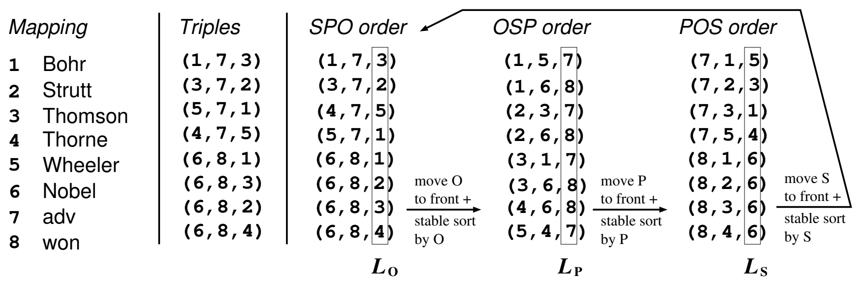
3.5 The FM-Index

Rather than describing the general FM-Index [14, 16], which belongs to the realm of text compression and searching, we show the ideas that adapt to our particular case of interest. Consider a set of n distinct strings of length ℓ, $S_i[1..\ell]$ for $1 \le i \le n$. Sort them lexicographically and write them one per row. The last column of symbols, read downwards, is called L_ℓ.

Now take the last symbol of each S_i and put it in front of the first symbol, that is, S_i becomes $S_i[\ell] \cdot S_i[1..\ell-1]$. Stably re-sort the strings and call $L_{\ell-1}$ the last column. Continue with this process until obtaining all the strings L_j, for $1 \le j \le \ell$.

If we consider the strings S_i as the rows of a relational table, then the strings L_j are akin to a column store, where the table is represented column-wise and the columns have some suitable order. We do not require pointers to connect the same row across different columns, because the row i' in L_{j-1} corresponding to row i in L_j turns out to be

$$i' = C_j[c] + rank_c(L_j, i),$$



Mapping		Triples	SPO order	OSP order	POS order
1	Bohr	(1,7,3)	(1,7,3)	(1,5,7)	(7,1,5)
2	Strutt	(3,7,2)	(3,7,2)	(1,6,8)	(7,2,3)
3	Thomson	(5,7,1)	(4,7,5)	(2,3,7)	(7,3,1)
4	Thorne	(4,7,5)	(5,7,1)	(2,6,8)	(7,5,4)
5	Wheeler	(6,8,1)	(6,8,1)	(3,1,7)	(8,1,6)
6	Nobel	(6,8,3)	(6,8,2)	(3,6,8)	(8,2,6)
7	adv	(6,8,2)	(6,8,3)	(4,6,8)	(8,3,6)
8	won	(6,8,4)	(6,8,4)	(5,4,7)	(8,4,6)

move O to front + stable sort by O
move P to front + stable sort by P
move S to front + stable sort by S
L_O
L_P
L_S

Figure 6 On the left, a mapping the nodes and labels of Figure 1 to integers. Right to it, the resulting table of triples. On the right, the three reorderings from which the columns L_O, L_P, and L_S are obtained.

where $c = L_j[i]$ and $C_j[c]$ is the number of symbols smaller than c in L_j. The same formula navigates from L_1 to L_ℓ. We can therefore extract any row S_i in time $O(\ell \log \sigma)$ by representing the strings L_j with wavelet trees (Section 3.4), from its position in any column. We can also navigate forwards, from $L_{j-1}[i']$ to $L_j[i]$, with the inverse formula

$$i \;=\; select_c(L_j, i' - C_{j-1}[c]),$$

where c is such that $C_{j-1}[c] < i' \leq C_{j-1}[c]$.

This representation, which uses basically the same $n\ell \log_2 \sigma$ bits of a plain representation of the rows S_i, allows for other interesting queries. In particular, given some substring $X[1..t]$ and a column a, we can obtain the set of all rows S_i such that $S_i[a+1..a+t] = X$, by starting from $s_{t+1} = 1$, $e_{t+1} = n$, and then, for $j = t$ down to 1, computing $c = X[j]$ and

$$\begin{aligned} s_j &= C_{a+j}[c] + rank_c(L_{a+j}, s_{j+1} - 1) + 1, \\ e_j &= C_{a+j}[c] + rank_c(L_{a+j}, e_{j+1}). \end{aligned}$$

At the end, the desired strings are those represented in the range $L_{a+1}[s_1..e_1]$. This process is called *backward search.*

Figure 6 illustrates this structure on the three-column table that results from representing the labeled graph of Figure 1. The table is represented by the resulting columns L_O, L_P, and L_S. These three strings, represented as wavelet trees, plus the corresponding arrays C_*, form the ring data structure for graph databases. Note that Figure 5 shows the wavelet tree of L_O.

4 Qdags

Qdags [40, 6] represent each d-attribute table as a d-dimensional version of the quadtrees described in Section 3.3. A multijoin query between several tables represented by such quadtrees is solved by:

1. Converting each quadtree into one that includes the missing attributes that appear in any other joined table, all in the same order.
2. Traversing the quadtrees in synchronization to collect the points in common.
3. Writing the output of the query as a new quadtree on the increased dimension.

Qdags solve the problem of lifting the dimension of the quadtrees (step 1) at almost no extra cost. A qdag is a quadtree plus a *mapping function* that can be used to permute attributes and, most importantly, lift its dimension: for each d-dimensional point $(x_1, \ldots, x_d)$,

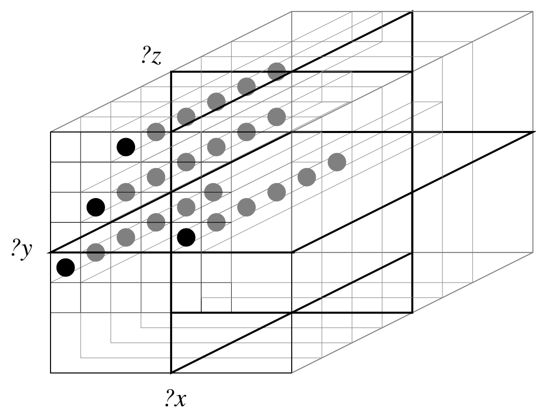


Figure 7 Extending the quadtree adv of Figure 4 to a third dimension to account for variable $?z$.

if we raise the dimension to d', we assume that the points $(x_1, \ldots, x_d, y_{d+1}, \ldots, y_{d'})$ exist for all the possible values of $y_{d+1}, \ldots, y_{d'}$. The qdag then simulates the operations on the virtual d'-dimensional quadtree without materializing it.

To illustrate, consider the following variant of the BGP of Eq. (1)

$$(?y, \mathsf{adv}, ?x), \ (?z, \mathsf{won}, ?x), \ (?z, \mathsf{won}, ?y)$$

which has the same output in our database with the binding $?z = \mathsf{Nobel}$. Since the output will be a table with attributes $(?z, ?y, ?x)$, we need to raise the dimension of the intervening quadtrees (shown in Figure 4) to three. For the first triple pattern, $(?y, \mathsf{adv}, ?x)$, we must create the third dimension, $?z$. The corresponding qdag must represent an octree (i.e., a 3-dimensional version of a quadtree) where every point $(?y, ?x)$ in the quadtree adv is extended to every possible value of $?z$; see Figure 7. Instead of materializing that octree, the qdag combines the quadtree adv with the mapping function $(1, 2, 3, 4, 1, 2, 3, 4)$. This indicates how to traverse the 8 children of every node in the octree, reading the 4 front cubes and then the 4 back cubes; note the 4 back cubes are identical to the 4 front cubes. The quadtree is then used to support the octree navigation with just this $O(2^{d'})$ additive space and time penalty. Analogously, the qdag for the triple pattern $(?z, \mathsf{won}, ?x)$ is built from the quadtree won and the mapping function $(1, 2, 1, 2, 3, 4, 3, 4)$, and the triple pattern $(?z, \mathsf{won}, ?y)$ is built from the same quadtree won and the mapping function $(1, 1, 2, 2, 3, 3, 4, 4)$.

Optimality. Note that the intersection process may work on subgrids where no output points are found, so the intersection process is not necessarily instance-optimal. It was shown, however, that there is always a database which, essentially, has points wherever the algorithm traverses in the grids, which makes this multijoin algorithm wco.

Full algebra. The algorithm is compositional, since the output is also a quadtree (and hence a qdag, with the identity mapping function). This compositionality leads to including the other operations of the relational algebra. For this sake, qdags are extended to the so-called *lazy qdags (lqdags)*, which are akin to the syntax tree of the algebraic expression, through which the results flow on demand. The scheme stays wco for Boolean operations (under some constraints), but not for other operations like general selections and projections.

In practice. The practical implementation uses k^d-trees to represent the quadtrees, as described in Section 3.3. The resulting quads are evaluated on a subset of Wikidata, where one two-dimensional qdag is built for each distinct predicate. The qdags use less than 5 bytes per triple, about half of the plain representation and 10–300 times less than state-of-the-art engines. Their times to solve BGPs from a query log are competitive, from much faster to much slower depending on the query types. Qdags perform better in general on lower dimensions of the output and are unbeaten on small cyclic queries.

5 The Ring

The ring [4] is a text-based compressed representation for the database triples, which can simulate the 6 tries needed by the LTJ algorithm with just a single copy of the data. The high-level idea is that each (s, p, o) triple is regarded as a *circular* string that can be navigated forwards or backwards. Any of the 6 orders can be then obtained by starting somewhere on the circle and moving in some direction.

As described in Section 3.5, the ring represents the table of triples (s, p, o) by means of the sequences L_{O}, L_{P}, and L_{S}. The key idea to simulate the LTJ algorithm of Section 2.3 is that every node of each of the 6 tries corresponds to a range in some of the L_* sequences: both represent sets of triples with some attributes already bound. We then start by associating each triple pattern in the BGP to a range in some L_* corresponding to its bound values. To find that range, we use backward search (Section 3.5). For example, for the BGP of Eq. (1) the triple $(?y, \mathsf{adv}, ?x)$ corresponds to the range $L_{\mathrm{S}}[1..4]$, whereas $(\mathsf{Nobel}, \mathsf{won}, ?x)$ and $(\mathsf{Nobel}, \mathsf{won}, ?y)$ correspond to $L_{\mathrm{O}}[5..8]$ (see Figure 6).

The LTJ algorithm is then started, binding the variables one by one. The main primitive needed to implement the intersections carried out by LTJ is: given a value k, find the leftmost child of the current trie node with value $k' \geq k$. In the ring, this corresponds to finding the smallest value $k' \geq k$ appearing in a given range $L_*[i..j]$. This can be done in logarithmic time on the wavelet tree of L_* [8, 37]. Wavelet trees also implement the needed primitives to simulate trie navigation on the sequences L_*, forwards or backwards as needed [21, 37].

For example, if we first bind $?y$, we must find the common values between $L_{\mathrm{S}}[1..4]$ and $L_{\mathrm{O}}[5..8]$, yielding 1 (Bohr), 3 (Thomson), and 4 (Thorne). Those are the instances of $?y$ that advised someone and won a Nobel prize (recall Section 2.4). Consider the branch $y = 1$. We use the backward search formula to move from $L_{\mathrm{S}}[1..4]$ to $L_{\mathrm{O}}[1..1]$ (which represents the further bounded triple pattern $(\mathsf{Bohr}, \mathsf{adv}, ?x)$), and from $L_{\mathrm{O}}[5..8]$ to $L_{\mathrm{P}}[2..2]$ (which represents $(\mathsf{Nobel}, \mathsf{won}, \mathsf{Bohr})$; this triple pattern is now totally bound). Now we bind $?x$, looking for the common values in $L_{\mathrm{O}}[1..1]$ and $L_{\mathrm{O}}[5..8]$. We here find the common value 3 (Thomson), and take that binding by moving from $L_{\mathrm{O}}[1..1]$ to $L_{\mathrm{P}}[5..5]$ (representing the triple $(\mathsf{Bohr}, \mathsf{adv}, \mathsf{Thomson})$) and from $L_{\mathrm{O}}[5..8]$ to $L_{\mathrm{P}}[6..6]$ (representing $(\mathsf{Nobel}, \mathsf{won}, \mathsf{Thomson})$). Now we have bound all the variables and the three triples represent one solution of the BGP: Bohr advised Thomson and both won the Nobel prize.

Optimality and practical performance. The ring handles BGPs in wco time, since it directly simulates the LTJ algorithm. Depending on how much it compresses its wavelet trees, the ring can use about the same space of qdags, and it is also competitive in time with the state of the art (it is faster than qdags in most cases, but not on small cyclic queries). Without wavelet tree compression, it uses about 13 bytes per triple (close to the space needed by the raw graph data, and still 5–140 times smaller than the other indices) and it is on average twice as fast as the next-best competitor.

Higher dimensions. The ring can be extended to higher dimensions, needing much fewer than the $d!$ copies required by classical schemes (e.g., one needs 7 rings, instead of 720 tries, for $d = 6$). This makes it usable to implement LTJ on relational tables where the classical wco indices are completely impractical. Still, the number of required rings grows as $O(2^d)$, so it soon ceases to be practical as well.

Regular path queries. The ring was also used to solve RPQs [5] by just performing the classical traversal of the product automaton, with a couple of twists. First, the NFA is produced by Glushkov's algorithm [23, 39], which obtains the worst-case minimum number of states and has some regularities that are exploited (e.g., all the transition leading to a given state have the same label). Second, the wavelet trees of the sequences L_* are enhanced so as to avoid spending any time on edges of the product automaton that lead to no active NFA states, or that cycle on the automaton. The resulting algorithm, even if not using any filtration technique, is competitive with the state of the art (3 times faster than the next-best, on average), while using 3–5 times less space than all of them. More recent developments using filtration techniques become about 5 times faster than the others.

6 Now What?

Our research has demonstrated that cds can successfully implement the core of graph database engines, providing wco multijoin algorithms that are also efficient in practice, and removing all of the redundancy associated with those algorithms. As such, they can make a reality the efficient querying of the huge graph databases that are emerging.

But we have just scratched the surface of the problem. There are many issues to consider in the way, on many of which we are working. We list only some of the most prominent ones.

How to handle higher dimensions? While three dimensions (or four, in some models) suffice to describe graph databases, relational ones may have many more columns. Both qdags and the ring have time or space troubles with higher dimensions, and even if they can handle them better than current schemes, they soon become impractical. In order to provide competitive solutions for relational data, we must probably combine wco and non-wco schemes [35, 20]. Cds have demonstrated that they can provide more than the basic functionality on the data, so an interesting question is what can they support in the direction of combining both kinds of query plans.

Can we provide more functionality? A formidable challenge is to combine BGPs and RPQs in an efficient manner, as this is supported in SPARQL and other query languages. Interactive querying requires retrieving (possibly only some) results in decreasing order of relevance [52]. Providing more semantics to the nodes leads to problems like supporting similarity joins [13], spatio-temporal predicates, and so on. The concept of wco with those extended semantics is yet to be studied. Again, cds can provide novel and more efficient solutions to those problems.

How to scale to a real system? Real graph database systems are very complex, and thus it is not direct to put our performance improvements, which focus on specific subproblems, into use. Consider for example going from our BGPs and RPQs to the full SPARQL support. The fastest path is to integrate our research prototypes into an existing system. A good candidate for this is MillenniumDB [55], a full-fledged graph database system with strong algorithmic foundations and designed to plug-and-play different solutions to subproblems.

Can we support graph analytics? BGPs serve not only for *querying* graph databases, but also as building blocks to support *graph analytics* [28]. In our example graph, we could ask how likely is that the advisee of a Nobel winner also wins the prize, by *counting* the number of answers to BGP queries (rather than listing them all). Graph analytics require various sorts of summarization operations on the query results, where in addition it is acceptable to return approximate answers. It is interesting to see if cds can support counting (perhaps approximately) the number of results of queries without listing them all; some of their extended functionality can be of use. More in general, we can explore the use of cds to represent other objects that are key in analytics, like matrices. There is some preliminary work in this direction [15, 18, 22].

References

1 C. R. Aberger, A. Lamb, S. Tu, A. Nötzli, K. Olukotun, and C. Ré. Emptyheaded: A relational engine for graph processing. *ACM Transactions on Database Systems*, 42, 2017.

2 W. Ali, M. Saleem, B. Yao, A. Hogan, and A.-C. Ngonga Ngomo. A survey of RDF stores & SPARQL engines for querying knowledge graphs. *The VLDB Journal*, 31(3):1–26, 2022.

3 R. Angles, M. Arenas, P. Barceló, A. Hogan, J. L. Reutter, and D. Vrgoc. Foundations of modern query languages for graph databases. *ACM Computing Surveys*, 50(5):68:1–68:40, 2017.

4 D. Arroyuelo, A. Hogan, G. Navarro, J. Reutter, J. Rojas-Ledesma, and A. Soto. Worst-case optimal graph joins in almost no space. In *Proc. 47th ACM International Conference on Management of Data (SIGMOD)*, pages 102–114, 2021.

5 D. Arroyuelo, A. Hogan, G. Navarro, and J. Rojas-Ledesma. Time- and space-efficient regular path queries. In *Proc. 38th IEEE International Conference on Data Engineering (ICDE)*, pages 3091–3105, 2022.

6 D. Arroyuelo, G. Navarro, J. L. Reutter, and J. Rojas-Ledesma. Optimal joins using compressed quadtrees. *ACM Transactions on Database Systems*, 47(2):article 8, 2022.

7 A. Atserias, M. Grohe, and D. Marx. Size bounds and query plans for relational joins. *SIAM Journal on Computing*, 42(4):1737–1767, 2013.

8 J. Barbay, F. Claude, and G. Navarro. Compact binary relation representations with rich functionality. *Information and Computation*, 232:19–37, 2013.

9 D. Benoit, E. D. Demaine, J. I. Munro, R. Raman, V. Raman, and S. S. Rao. Representing trees of higher degree. *Algorithmica*, 43(4):275–292, 2005.

10 N. Brisaboa, S. Ladra, and G. Navarro. Compact representation of web graphs with extended functionality. *Information Systems*, 39(1):152–174, 2014.

11 D. Clark. *Compact Pat Trees.* PhD thesis, University of Waterloo, 1996.

12 O. Erling. Virtuoso, a hybrid RDBMS/graph column store. *Data Engineering Bulletin*, 35(1):3–8, 2012.

13 S. Ferrada, B. Bustos, and A. Hogan. Extending SPARQL with similarity joins. In *Proc. 19th International Semantic Web Conference (ISWC)*, pages 201–217, 2020.

14 P. Ferragina and G. Manzini. Indexing compressed texts. *Journal of the ACM*, 52(4):552–581, 2005.

15 P. Ferragina, G. Manzini, T. Gagie, D. Köppl, G. Navarro, M. Striani, and F. Tosoni. Improving matrix-vector multiplication via lossless grammar-compressed matrices. *Proceedings of the VLDB Endowment*, 2022. To appear. See `https://www.dcc.uchile.cl/gnavarro/ps/pvldb22.pdf`.

16 P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. Compressed representations of sequences and full-text indexes. *ACM Transactions on Algorithms*, 3(2):article 20, 2007.

17 N. Francis, A. Green, P. Guagliardo, L. Libkin, T. Lindaaker, V. Marsault, S. Plantikow, M. Rydberg, P. Selmer, and A. Taylor. Cypher: An evolving query language for property graphs. In *Proc. SIGMOD International Conference on Management of Data*, pages 1433–1445, 2018.

18 A. P. Francisco, T. Gagie, D. Köppl, S. Ladra, and G. Navarro. Graph compression for adjacency-matrix multiplication. *SN Computer Science*, 3:article 193, 2022.

19 E. Fredkin. Trie memory. *Communications of the ACM*, 3:490–500, 1960.

20 M. J. Freitag, M. Bandle, T. Schmidt, A. Kemper, and T. Neumann. Adopting worst-case optimal joins in relational database systems. *Proceedings of the VLDB Endowment*, 13(11):1891–1904, 2020.

21 T. Gagie, G. Navarro, and S. J. Puglisi. New algorithms on wavelet trees and applications to information retrieval. *Theoretical Computer Science*, 426-427:25–41, 2012.

22 F. Geerts, T. Muñoz, C. Riveros, J. van den Bussche, and D. Vrgoc. Matrix query languages. *SIGMOD Record*, 50(3):6–19, 2021.

23 V-M. Glushkov. The abstract theory of automata. *Russian Mathematical Surveys*, 16:1–53, 1961.

24 R. Grossi, A. Gupta, and J. Vitter. High-order entropy-compressed text indexes. In *Proc. 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 841–850, 2003.

25 S. Harris, A. Seaborne, and E. Prud'hommeaux. SPARQL 1.1 Query Language. W3C Recommendation. URL: `https://www.w3.org/TR/sparql11-query/`.

26 A. Hogan. *The Web of Data*. Springer, 2020.

27 A. Hogan, E. Blomqvist, M. Cochez, C. d'Amato, G. de Melo, C. Gutiérrez, S. Kirrane, J.E. Labra Gayo, R. Navigli, S. Neumaier, A.-C. Ngonga Ngomo, A. Polleres, S. M. Rashid, A. Rula, L. Schmelzeisen, J. Sequeda, S. Staab, and A. Zimmermann. *Knowledge Graphs*. Synthesis Lectures on Data, Semantics, and Knowledge. Morgan & Claypool Publishers, 2021.

28 A. Hogan, J. L. Reutter, and A. Soto. In-database graph analytics with recursive SPARQL. In *Proc. 19th International Semantic Web Conference (ISWC)*, pages 511–528, 2020.

29 A. Hogan, C. Riveros, C. Rojas, and A. Soto. A worst-case optimal join algorithm for SPARQL. In *Proc. 18th International Semantic Web Conference (ISWC)*, pages 258–275, 2019.

30 O. Kalinsky, Y. Etsion, and B. Kimelfeld. Flexible caching in trie joins. In *Proc. 20th International Conference on Extending Database Technology (EDBT)*, pages 282–293, 2017.

31 M. A. Khamis, H. Q. Ngo, C. Ré, and A. Rudra. Joins via geometric resolutions: Worst case and beyond. *ACM Transactions on Database Systems*, 41(4), 2016.

32 A. Koschmieder and U. Leser. Regular path queries on large graphs. In *Proc. International Conference on Scientific and Statistical Database Management (SSDBM)*, volume 7338 of *LNCS*, pages 177–194. Springer, 2012.

33 S. Malyshev, M. Krötzsch, L. González, J. Gonsior, and A. Bielefeldt. Getting the most out of Wikidata: Semantic technology usage in Wikipedia's knowledge graph. In *Proc. 17th International Semantic Web Conference (ISWC)*, pages 376–394, 2018.

34 F. Manola and E. Miller. RDF primer. W3C Recommendation, 2004. URL: `http://www.w3.org/TR/rdf-primer/`.

35 A. Mhedhbi and S. Salihoglu. Optimizing subgraph queries by combining binary and worst-case optimal joins. *Proc. VLDB Endowment*, 12(11):1692–1704, 2019.

36 J. J. Miller. Graph database applications and concepts with Neo4j. In *Proc. Southern Association for Information Systems Conference*, pages 141–147, 2013.

37 G. Navarro. Wavelet trees for all. *Journal of Discrete Algorithms*, 25:2–20, 2014.

38 G. Navarro. *Compact Data Structures – A practical approach*. Cambridge Univ. Press, 2016.

39 G. Navarro and M. Raffinot. *Flexible Pattern Matching in Strings – Practical on-line search algorithms for texts and biological sequences*. Cambridge Univ. Press, 2002.

40 G. Navarro, J. Reutter, and J. Rojas-Ledesma. Optimal joins using compact data structures. In *Proc. 23rd International Conference on Database Theory (ICDT)*, pages 21:1–21:21, 2020.

41 T. Neumann and G. Weikum. The RDF-3X engine for scalable management of RDF data. *VLDB Journal*, 19:91–113, 2010.

42 H. Q. Ngo. Worst-case optimal join algorithms: Techniques, results, and open problems. In *Proc. 37th Symposium on Principles of Database Systems (PODS)*, pages 111–124, 2018.

43 H. Q. Ngo, E. Porat, C. Ré, and A. Rudra. Worst-case optimal join algorithms. In *Proc. 31st Symposium on Principles of Database Systems (PODS)*, pages 37–48, 2012.

44 H. Q. Ngo, C. Ré, and A. Rudra. Skew strikes back: new developments in the theory of join algorithms. *SIGMOD Record*, 42(4):5–16, 2013.

45 D. Nguyen, M. Aref, M. Bravenboer, G. Kollias, H. Q. Ngo, C. Ré, and A. Rudra. Join processing for graph patterns: An old dog with new tricks. In *Proc. 3rd International Workshop on Graph Data Management Experiences and Systems (GRADES)*, pages 2:1–2:8, 2015.

46 V.-Q. Nguyen and K. Kim. Efficient regular path query evaluation by splitting with unit-subquery cost matrix. *IEICE Transactions on Information and Systems*, 100-D(10):2648–2652, 2017.

47 R. Raman, V. Raman, and S. S. Rao. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Transactions on Algorithms*, 3(4):article 43, 2007.

48 I. Robinson, J. Webber, and E. Eifrem. *Graph Databases.* O'Reilly, 2nd edition, 2015.

49 H. Samet. The quadtree and related hierarchical data structures. *ACM Computing Surveys*, 16(2):187–260, 1984.

50 H. Samet. *Foundations of Multidimensional and Metric Data Structures.* Morgan Kaufmann, 2006.

51 B. B. Thompson, M. Personick, and M. Cutcher. The Bigdata®RDF Graph Database. In *Linked Data Management*, pages 193–237. Chapman and Hall/CRC, 2014.

52 N. Tziavelis, D. Ajwani, W. Gatterbauer, M. Riedewald, and X. Yang. Optimal algorithms for ranked enumeration of answers to full conjunctive queries. *Proceedings of the VLDB Endowment*, 13(9):1582–1597, 2020.

53 T. L. Veldhuizen. Triejoin: A simple, worst-case optimal join algorithm. In *Proc. 17th International Conference on Database Theory (ICDT)*, pages 96–106, 2014.

54 D. Vrandecic and M. Krötzsch. Wikidata: A free collaborative knowledgebase. *Communications of the ACM*, 57(10):78–85, 2014.

55 D. Vrgoc, C. Rojas, R. Angles, M. Arenas, D. Arroyuelo, C. Buil Aranda, A. Hogan, G. Navarro, C. Riveros, and J. Romero. MillenniumDB: A persistent, open-source, graph database. *CoRR*, abs/2111.01540, 2021. `arXiv:2111.01540`.

56 X. Wang, J. Wang, and X. Zhang. Efficient distributed regular path queries on RDF graphs using partial evaluation. In *Proc. International Conference on Information and Knowledge Management (CIKM)*, pages 1933–1936, 2016.

57 M. Yannakakis. Algorithms for acyclic database schemes. In *Proc. 7th International Conference on Very Large Databases (VLDB)*, pages 82–94, 1981.

Some Vignettes on Subgraph Counting Using Graph Orientations

C. Seshadhri
Department of Computer Science & Engineering, University of California, Santa Cruz, CA, USA

Abstract

Subgraph counting is a fundamental problem that spans many areas in computer science: database theory, logic, network science, data mining, and complexity theory. Given a large input graph G and a small pattern graph H, we wish to count the number of occurrences of H in G. In recent times, there has been a resurgence on using an old (maybe overlooked?) technique of orienting the edges of G and H, and then using a combination of brute-force enumeration and indexing. These orientation techniques appear to give the best of both worlds. There is a rigorous theoretical explanation behind these techniques, and they also have excellent empirical behavior (on large real-world graphs). Time and again, graph orientations help solve subgraph counting problems in various computational models, be it sampling, streaming, distributed, etc. In this paper, we give some short vignettes on how the orientation technique solves a variety of algorithmic problems.

2012 ACM Subject Classification Mathematics of computing → Graph algorithms

Keywords and phrases subgraph counting, graph degeneracy, homomorphism counting, graph algorithms

Digital Object Identifier 10.4230/LIPIcs.ICDT.2023.3

Category Invited Talk

Funding NSF DMS-2023495, CCF-1740850, 1908384, 1909790

1 Introduction

A central problem in computer science is to count or enumerate the occurrences of a small pattern graph H in a large input graph G. The applications of graph pattern counts occur across numerous scientific areas, including logic, biology, statistical physics, database theory, social sciences, machine learning, and network science [36, 13, 17, 12, 24, 9, 30, 55, 48, 21, 47]. The tutorial [51] has more details on applications. A rich and deep theory has emerged from the study of graph pattern counting [41, 14, 32, 19, 42, 2, 18, 48, 49].

Let us formalize this problem through the language of graph homomorphisms (or graph mappings). The pattern simple graph is denoted $H = (V(H), E(H))$, and is thought of constant-sized. The input simple graph is denoted $G = (V(G), E(G))$. An H-homomorphism is a map $f : V(H) \to V(G)$ that preserves edges. Formally, $\forall (u, v) \in E(H)$, $(f(u), f(v)) \in E(G)$. If the map is 1-1, then it is called a *subgraph*. If the map also preserves non-edges, then it is an *induced* subgraph/homomorphism. For this high-level survey, we will not commit to any specific problem variant. We use "subgraph counting" an umbrella terms that refers to all of these problems.

The study of efficient algorithms for subgraph counting is almost a subfield in of itself [37, 3, 12, 24, 22, 19, 9, 18, 10, 49]. It would take us too far out to survey the state of this area. Even the simplest version, when H is a triangle, receives much attention. Let $n = |V(G)|$ and $k = |V(H)|$. The trivial algorithm that simply tries all k-tuples of vertices runs in $O(n^k)$ time. By $\#W[1]$-hardness even for H being a k-clique, we do not expect $n^{o(k)}$ algorithms for general H [19]. The algorithmic study of subgraph counting focuses on understanding conditions on H and G when the trivial n^k running time bound can be beaten.

The algorithmic technique of *graph orientations* has repeatedly helped in solving such counting problems. It is also practically viable and is a key tool in subgraph counting applications for real-world graphs. The study of this technique has led to a rich hoard of mathematical results, which further inspire empirical work. In this paper, we will describe a few short vignettes, describing the application of the technique and specific results.

2 Triangle counting through graph orientations

Let us begin with the basic problem of triangle counting, where H is a 3-clique. A fairly direct algorithm is the *wedge enumeration* procedure. For each vertex u, list all pairs of neighbors v, w. If (v, w) is an edge, then (u, v, w) form a triangle. Observe that we enumerate two-paths (or wedges) v, u, w; hence the name wedge enumeration.

Let the graph G have n vertices and m edges. For a vertex v, let d_v denote its degree. The running time of the above procedure is $O(\sum_v d_v^2)$. Not surprisingly, high-degree vertices greatly affect the running time.

Graph orientations can be thought of as a technique to cut down the running time of wedge enumeration. This method has been rediscovered many times, but the earliest reference is by Chiba-Nishizeki [14]. Chrobak-Eppstein use this idea to deal with planar graphs [15]. It has been rediscovered by Schank-Wagner [50] and Cohen [16].

▶ **Definition 1.** *Given any undirected, simple graph G, an* acyclic orientation *of G is a DAG D such that (u, v) is an edge in D iff (u, v) is an edge in G.*

Let the partial order on vertices induced by D be denoted $\prec_D$.

We can also construct a DAG by defining a total order π on the vertices, and then orienting the edges from lower to higher vertex.

We consider an acycle orientiation D, and instead enumerate all (directed) triangles in D. Observe that a triangle has a unique acyclic orientation. Moreover, from every u, we will only find triangles (u, v, w) such that $u \prec_D v, w$. This is what allows for the major savings in computation.

Formally, the meta-algorithm is:

```
1. Compute an acyclic orientation $D$ of the input graph $G$.
2. For every vertex $u$:
   a. For every pair of *outneighbors* $v, w$, check if edge $(v, w)$ is present.
```

The key difference in the enumeration method is to only look at the outneighbors of u, which is at most the degree of u. Suppose the outdegree of a vertex v is denoted d_v^+. Then the running time of the meta-algorithm is $O(\sum_v (d_v^+)^2)$.

So how do we choose the orientation to make this sum small? We will consider two schemes: *degree orientations* and *degeneracy orientations.*

Degree orientations. We order vertices by degree, breaking ties by vertex id. Formally, $u \prec_D v$ iff $d(u) < d(v)$ or $d(u) = d(v)$ and u has smaller id. This was the ordering proposed in Chiba-Nishizeki's original paper [14]. It is implicitly used in the `forward` algorithm of Schank-Wagner [50].

Degeneracy orientations. This is a more sophisticated approach. Think of the "peeling process", where we repeatedly remove a vertex of minimum degree. (Note that the degree keeps decreasing as more vertices are removed.) The order of removal is the degeneracy ordering, and one simply creates a DAG from this ordering.

The degeneracy orientation is a byproduct of the classic core decomposition of Matula and Beck [43]. It was first used for subgraph counting by Chrobak-Eppstein [15]. Schank-Wagner independently give the equivalent `node-iterator-core` algorithm [50].

2.1 Graph orientations and degeneracy

There is a remarkable connection between the graph orientations given above, the concept of *graph degeneracy*, and measures of graph density. Let us begin with a classic definition from graph theory that directly ties into our problem. For a directed graph D, we use $d^+_v(D)$ to denote the outdegree of v in D.

▶ **Definition 2.** *The graph degeneracy, denoted $\kappa(G)$, is defined as follows.*

$$\kappa(G) = \min_{D \text{ orientation of } G} \max_v d^+_v(D)$$

In plain English, the graph degeneracy is the smallest possible maximum outdegree of an acyclic orientation of G. This quantity is also called the *coloring number*, due to connections with graph coloring (Sec. 5.2 of [23]). For convenience, we will simply denote the degeneracy of G as κ.

Matula-Beck gave a simple linear time algorithm to compute the graph degeneracy, which is exactly the peeling process [43]. Quite surprisingly, the peeling process (or core decomposition) discovers the orientation that minimizes the maximum outdegree. Observe that the running time of the triangle enumeration process can be bounded as $O(\sum_v (d^+_v)^2) = O(\max_v d^+_v \sum_v d^+_v) = O(m \max_v d^+_v)$. If we choose the degeneracy orientation, then the running time is $O(m\kappa)$. This algorithm is somewhat folklore, and explicitly stated by Schank-Wagner [50].

The $O(m\kappa)$ bound was first achieved by Chiba-Nishizeki, using degree orientations [14]. They (implicitly) proved the following theorem, which is stronger than what is required.

▶ **Theorem 3** ([14]). *For the degree orientation, $\sum_v d_v d^+_v = O(m\kappa)$.*

Asymptotically, both degree and degeneracy orientations provide the same running time benefit for triangle counting. This result of Chiba-Nishizeki was expressed in terms of the graph *arboricity*, a closely related parameter. But this result sparked off an entire subarea of algorithms, where the running time is parameterized by the graph degeneracy.

To get more context, let us dig deeper into the meaning of degeneracy and its connection to other graph parameters.

2.2 Degeneracy and graph density

The (half) average degree of a graph, $m(G)/n(G)$, is a natural graph parameter. Yet it appears to be a weak measure of the density of a graph. One may have a graph with a linear number of edges, but containing a clique of size $\sqrt{n}$. A stronger notion of sparsity would be the minimum average degree over all subgraphs of G.

The following theorem builds on classic results of Nash-Williams [44]. It relates the degeneracy to strong notions of graph sparsity.

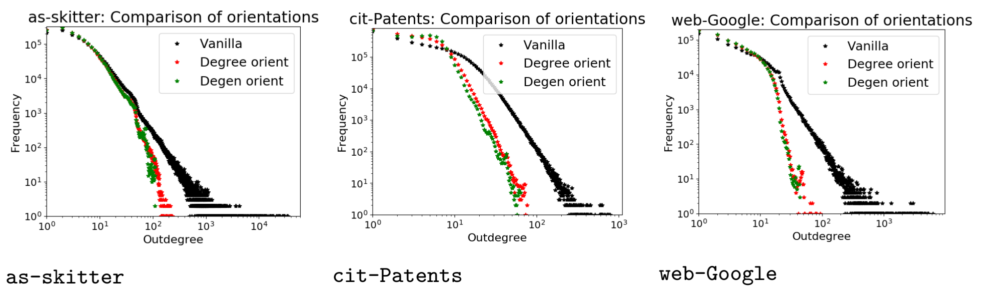


Figure 1 We plot the outdegree distributions of the degree and degeneracy orientation for different real-world graphs. For context, the plots also give the original (vanilla) degree distribution, to see how the orientations cut down the heavy tail. Observe that both orientations do quite well, though the degeneracy orientation leads to a smaller maximum degree.

▶ **Theorem 4.** *Let* $\alpha(G) = \max_{G' \textit{subgraph of } G} \frac{m(G)}{n(G)-1}$. *Then* $\alpha(G) \leq \kappa(G) \leq 2\alpha(G)$.

Ignoring constant factors, a low degeneracy graph is one where *all* subgraphs have low average degree. One can show that $\kappa(G) \leq \sqrt{2m}$, which shows that triangle counting for any graph can be done in $O(m^{3/2})$ time.

This concept motivates *bounded degeneracy graph classes*. These are graph classes with constant degeneracy, or alternately, graphs where all subgraphs are sparse. Bounded degeneracy graph classes are immensely rich; they contain all minor-closed families. Preferential attachment graphs have constant degeneracy. Real-world graphs typically have a small degeneracy, comparable to their average degree ([33, 39, 53, 4, 8], also Table 2 in [4]). The repeated occurrence of bounded degeneracy graphs across many scenarios underscores the importance of graph orientations as an algorithmic technique.

2.3 Taming real-world heavy tails

The heavy-tailed degree distribution is one of the hallmarks of real-world graphs. While these graphs are sparse, their degrees show high variance. These heavy tails pose particular challenges for subgraph counting and other algorithmic tasks. Orientations give a simple and effective method to cut down these tails.

In Figure 1, we plot the (out)degree distributions for three different real-world networks with millions of edges [56]. The degree distribution is the number of vertices of a given degree, plotted in log-log scale. The "vanilla" points, marked in black, give the original degree distribution. One can see the characteristic heavy tail in all cases.

We then plot the outdegree distributions of the degree and degeneracy orientations, in red and green respectively. Observe how both these orientations dramatically reduce the tail. The degeneracy orientation is only slight lower than the degree orientation. As expected the maximum degree of the degeneracy orientation is smaller than that of the degree orientation. In general, the quantity $\sum_v (d_v^+)^2$ is similar for both orientations.

These observations explain why the orientation technique has so much practical utility. The original algorithm for triangle counting is immensely effective in practice. A well-engineered implementation can count triangles in real-world graphs with hundreds of millions of edges within minutes on a commodity machine [48, 1]. As the plots in Figure 1 show, for triangle counting, the degree orientation is as effective as the degeneracy orientation. Degree orientations have the additional benefit of being locally computable and easily parallelizable. Cohen [16] and Suri-Vassilvitskii [54] independently proposes this orientation for Map-Reduce listing of triangles.

2.4 Practical clique counting

The power of degeneracy orientations is central to most practical clique counting algorithms. Following the template for triangle counting, k-clique counting can be done by searching all $(k-1)$-tuples of outneighbors. So, for each vertex v, we consider $\binom{d_v^+}{k-1}$ tuples. This leads to a total running time of $O(\sum_v (d_v^+)^{k-1})$. For the degeneracy orientation, $\max_v d_v^+ = \kappa$. Hence, $\sum_v (d_v^+)^{k-1} \leq \kappa^{k-2} \sum_v d_v^+ = m\kappa^{k-2}$. Thus, we can get a $O(m\kappa^{k-2})$ time algorithm for counting all k-cliques.

In practice, this is a remarkably powerful tool for clique counting. Instead of enumerating within outneighborhoods, observe that k-clique counting on the input graph G is reduced to $(k-1)$-clique counting on the n outneighborhoods. Each outneighborhood is potentially small (at most size κ). Each "outneighborhood problems" can be parallelized or distributed; being small problems, one can fit each of them into the memory of a small machine. This idea is central to almost all state-of-the-art practical clique counting algorithms [29, 31, 38, 20, 52].

3 Beyond clique counting

It is natural to ask whether the power of orientations goes beyond counting cliques. A nice twist on the triangle counting algorithm can be used to count 4-cycles. As before, we will orient our input graph G using the degree or degeneracy orientation. Each 4-cycle of G will become an oriented version, and there are three possible non-isomorphic orientations of the cycle. These are shown in Figure 2.

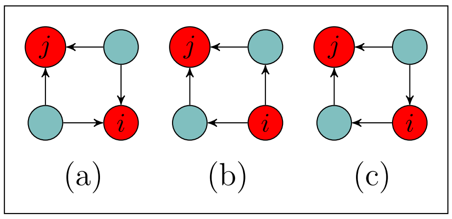


Figure 2 All acyclic orientations of the 4-cycle.

Notice that for all the three cases, the directed wedge between i and j (marked in red) is either an out-wedge or an inout-wedge. These wedges are given in Figure 3. Hence, one can enumerate only these wedges, index them appropriately, and get the total 4-cycle count.

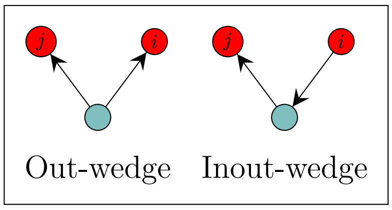


Figure 3 Directed wedges.

For two vertices i, j, let W_{ij}^{++} and W_{ij}^{+-} be the number of out-wedges and inout wedges respectively between i and j. The algorithm is:

1. Compute an acyclic (degree or degeneracy) orientation D of the input graph G.
2. Enumerate all out-wedges and inout-wedges (shown in Figure 3). Through this enumeration, compute, for each pair (i, j) of vertices, compute the numbers W^{++}_{ij} and W^{+-}_{ij}.
3. Output the sum $\sum_{i,j}\left(\binom{W^{++}_{ij}}{2} + \binom{W^{+-}_{ij}}{2} + W^{+-}_{ij} \cdot W^{++}_{ij}\right)$.

A few comments. By appropriate indexing and use of data structures, the entire running time can be made linear in the total number of out-wedges and inout wedges. The sum given above separately computes the various directed 4-cycles. There are three terms, each corresponding to one pattern in Figure 2. Observe that the algorithm gets an exact count *without* enumeration of 4-cycles. This leads to a large savings in running time.

The total number of wedges enumerated is at most $\sum_v d_v d^+_v$. This is somewhat larger than triangle counting, where only out-wedges are enumerated. Nonetheless, for the degeneracy ordering, $\max_v d^+_v = \kappa$. So the running time is $O(m\kappa)$. For the degree orientation, by Theorem 3, we also get the $O(m\kappa)$ running time.

This bound was first achieved by Chiba-Nishizeki, but through a more complicated algorithm and analysis. The presentation given here is from Pinar et al. [48]. An equivalent formulation was given earlier by Cohen [16].

The grand generalization. How far can this technique go? The overall template for counting H-subgraphs is to first construct all acyclic orientations of H, and count each of them in the degeneracy (or degree) oriented G. For each acyclic orientation of H, we break it up into a collection of directed rooted trees. By the outdegree bounds of the degeneracy orientation, we can enumerate all these directed rooted trees in G. These directed trees needed to indexed appropriately so the overall H-subgraph count can be efficiently computed (as in the case of 4-cycles, by the three terms).

A series of papers performed these generalizations [48, 6, 7, 5], and most significant is probably Bressan's notion of DAG treewidth [11]. By combining various results, one arrives at the following dichotomy theorem (technically for homomorphisms).

▶ **Theorem 5** ([7, 5]). *Suppose the longest induced cycle of H has length at most 5. Then, there is an algorithm exactly computing the H-homomorphism count that runs in $O(m\, poly(\kappa) \log n)$ time.*

Suppose the longest induced cycle of H has length strictly greater than 5. Assume the strong Triangle Detection Conjecture from fine-grained complexity. Then, for all (computable) functions $g : \mathbb{N} \to \mathbb{N}$ and all $\delta > 0$, there does not exist an algorithm computing H-homomorphism counts is $O(m^{4/3-\delta} g(\kappa))$ time.

This is surprisingly precise dichotomy theorem for when bounded degeneracy helps in subgraph/homomorphism counting. The limits of the orientation technique remarkably match up with the hardness result. The strong form of the Triangle Detection Conjecture states that there is no algorithm that can find a triangle in a graph in $O(m^{4/3-\delta})$ time. (The best upper bound is much larger, and would become $m^{4/3}$ if the matrix multiplication exponent is 2.)

Many practical algorithms for large-scale graph pattern counting use versions of these algorithms for bounded degeneracy graphs [2, 40, 48, 46, 39, 47]. While they may not be explicitly stated in the language above, the algorithmic techniques combine orientations and indexing. The concept of DAG treewidth captures the essence of the algorithms, and the upper bound of Theorem 5 subsumes all the applications.

4 A sublinear application

Let us consider a seemingly unrelated problem. We are given access to the adjacency list of a massive graph G. We can sample a uniform at random (uar) vertex, query the degree of a vertex, and can sample uar neighbors of a given vertex.

Our aim is to estimate the average degree $\sum_v d_v/n = 2m/n$, with the fewest queries to the graph. An obvious approach is to sample a set of uar vertices and compute the average degree of the sample. While this is an unbiased estimator, the variance can be extremely high. As an extreme example, suppose the graph is a star. So all vertices except the center have degree one, while the center vertex has degree $n-1$. The average degree is $2(n-1)/n = 2 - o(1)$. But the sampled average will be 1, with extremely high probability.

We have observed that orientations provide a way of "cutting down the tails". So consider the following algorithm.

```
1. Pick a uar vertex $u$.
2. Pick a uar neighbor $v$ of $u$.
3. If $d_u < d_v$, output $2d_u$. If $d_u = d_v$ and the ID of $u$ is less than the ID of $v$, output
   $2d_u$. Otherwise, output 0.
```

To analyze this algorithm, it is convenient to think of the degree orientation. When the procedure picks a directed edge leaving u, then it outputs $2d_u$. The expected output of this procedure is

$$\frac{1}{n}\sum_u \frac{d_u^+}{d_u}\cdot 2d_u = \frac{2\sum_u d_u^+}{n} = 2m/n$$

Thus, this procedure is also an unbiased estimator for the average degree. Observe what this procedure does for the star. The central vertex has no neighbors of high degree and thus, does not contribute to the estimator. Hence, the variance of the estimator is much smaller.

Remarkably, the variance can be bounded by the degeneracy.

▶ **Theorem 6** ([26]). *With high probability, the average of $O(\epsilon^3\kappa)$ samples is a $(1+\epsilon)$-approximation to the average degree.*

Since κ is at most $\sqrt{2m}$, this shows that the average degree of a graph can be approximated with (the optimal) $O(\sqrt{m})$ samples. The algorithm given above is substantially simpler than existing procedures that achieved this bound [35]. (Refer to Chapter 10.3 of [34] for more details.)

This idea of exploiting orientations by a sampling process has succeeded in solving a number of sublinear graph estimation problems [25, 26, 27, 28]. For such algorithms, we can only afford to use the degree orientation since it is locally computable. One of the challenges in these results is related properties of the degree orientation to desired properties of the degeneracy orientation.

5 Conclusion

These vignettes show the varied algorithmic uses of orientations for subgraph counting problems. Given the relative simplicity of the orientation technique, it is surprisingly effective in designing efficient algorithms. And as Theorem 5 shows, these algorithms are often optimal. Each of the above sections merely scratches the surface of what orientations can achieve. We discussed four related applications: triangles, cliques, four-cycles, and degree estimation.

The orientation technique has led to optimal and practical algorithms in each application. Moreover, there is a rich theory emerging on the basis of orientations. The connections to density in Section 2.2 form a starting point to much deeper inquiry into graph sparsity, developed by Nešetřil and Ossana de Mendez [45]. The sublinear subgraph counting results referenced in Section 4 have all emerged from understanding the power of degree orientations in reducing the variance of specific random variables.

References

1 Escape. `https://bitbucket.org/seshadhri/escape`.

2 Nesreen K. Ahmed, Jennifer Neville, Ryan A. Rossi, and Nick Duffield. Efficient graphlet counting for large networks. In *International Conference on Data Mining*, 2015.

3 Noga Alon, Raphael Yuster, and Uri Zwick. Finding and counting given length cycles. *Algorithmica*, 17(3):209–223, 1997.

4 Suman K Bera, Amit Chakrabarti, and Prantar Ghosh. Graph coloring via degeneracy in streaming and other space-conscious models. In *International Colloquium on Automata, Languages, and Programming (ICALP)*, 2020.

5 Suman K. Bera, Lior Gishboliner, Yevgeny Levanzov, C. Seshadhri, and Asaf Shapira. Counting subgraphs in degenerate graphs. *Journal of the ACM*, 69(3), 2022.

6 Suman K Bera, Noujan Pashanasangi, and C Seshadhri. Linear time subgraph counting, graph degeneracy, and the chasm at size six. In *Innovations in Theoretical Computer Science*, 2020.

7 Suman K. Bera, Noujan Pashanasangi, and C. Seshadhri. Near-linear time homomorphism counting in bounded degeneracy graphs: The barrier of long induced cycles. In *Proceedings of the Symposium on Discrete Algorithms (SODA)*, pages 2315–2332, USA, 2021.

8 Suman K Bera and C Seshadhri. How the degeneracy helps for triangle counting in graph streams. In *Principles of Database Systems*, pages 457–467, 2020.

9 Christian Borgs, Jennifer Chayes, László Lovász, Vera T Sós, and Katalin Vesztergombi. Counting graph homomorphisms. In *Topics in discrete mathematics*, pages 315–371. Springer, 2006.

10 Marco Bressan. Faster subgraph counting in sparse graphs. In *International Symposium on Parameterized and Exact Computation (IPEC)*, 2019.

11 Marco Bressan. Faster algorithms for counting subgraphs in sparse graphs. *Algorithmica*, 83:2578–2605, 2021.

12 Graham R Brightwell and Peter Winkler. Graph homomorphisms and phase transitions. *Journal of combinatorial theory, series B*, 77(2):221–262, 1999.

13 Ashok K Chandra and Philip M Merlin. Optimal implementation of conjunctive queries in relational data bases. In *Symposium on Theory of Computing (STOC)*, pages 77–90, 1977.

14 Norishige Chiba and Takao Nishizeki. Arboricity and subgraph listing algorithms. *SIAM Journal on Computing*, 14:210–223, 1985.

15 Marek Chrobak and David Eppstein. Planar orientations with low out-degree and compaction of adjacency matrices. *Theor. Comput. Sci.*, 86(2):243–266, 1991.

16 Jonathan Cohen. Graph twiddling in a MapReduce world. *Computing in Science & Engineering*, 11:29–41, 2009.

17 J. Coleman. Social capital in the creation of human capital. *American Journal of Sociology*, 94:S95–S120, 1988.

18 Radu Curticapean, Holger Dell, and Dániel Marx. Homomorphisms are a good basis for counting small subgraphs. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing*, pages 210–223, 2017.

19 Víctor Dalmau and Peter Jonsson. The complexity of counting homomorphisms seen from the other side. *Theoretical Computer Science*, 329(1-3):315–323, 2004.

20 Maximilien Danisch, Oana Denisa Balalau, and Mauro Sozio. Listing k-cliques in sparse real-world graphs. In *Conference on the World Wide Web (WWW)*, pages 589–598, 2018.

21 Holger Dell, Marc Roth, and Philip Wellnitz. Counting answers to existential questions. In *International Colloquium on Automata, Languages, and Programming (ICALP)*, 2019.

22 Josep Díaz, Maria Serna, and Dimitrios M Thilikos. Counting h-colorings of partial k-trees. *Theoretical Computer Science*, 281(1-2):291–309, 2002.

23 Reinhard Diestel. *Graph Theory, Fourth Edition.* Springer, 2010.

24 Martin Dyer and Catherine Greenhill. The complexity of counting graph homomorphisms. *Random Structures & Algorithms*, 17(3-4):260–289, 2000.

25 T. Eden, A. Levi, D. Ron, and C Seshadhri. Approximately counting triangles in sublinear time. In *Proceedings of the Symposium on Foundations of Computer Science (FOCS)*, pages 614–633, 2015.

26 T. Eden, D. Ron, and C. Seshadhri. Sublinear time estimation of degree distribution moments: The degeneracy connection. In *International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 7:1–7:13, 2017. `doi:10.4230/LIPIcs.ICALP.2017.7`.

27 T. Eden, D. Ron, and C. Seshadhri. On approximating the number of k-cliques in sublinear time. In *Symposium on Theory of Computing (STOC)*, pages 722–734, 2018.

28 T. Eden and W. Rosenbaum. On sampling edges almost uniformly. In *Symposium on Simplicity in Algorithms (SOSA)*, pages 1–9, 2018. `doi:10.4230/OASIcs.SOSA.2018.7`.

29 David Eppstein, Maarten Löffler, and Darren Strash. Listing all maximal cliques in large sparse real-world graphs. *ACM Journal of Experimental Algorithmics*, 18, 2013.

30 G. Fagiolo. Clustering in complex directed networks. *Phys. Rev. E*, 76:026107, August 2007.

31 Irene Finocchi, Marco Finocchi, and Emanuele G. Fusco. Clique counting in mapreduce: Algorithms and experiments. *ACM Journal of Experimental Algorithmics*, 20, 2015.

32 Jörg Flum and Martin Grohe. The parameterized complexity of counting problems. *SIAM Journal on Computing*, 33(4):892–922, 2004.

33 G. Goel and J. Gustedt. Bounded arboricity to determine the local structure of sparse graphs. In *International Workshop on Graph-Theoretic Concepts in Computer Science*, pages 159–167. Springer, 2006.

34 O. Goldreich. *Introduction to Property Testing.* Cambridge University Press, 2017.

35 O. Goldreich and D. Ron. Approximating average parameters of graphs. *Random Structures and Algorithms*, 32(4):473–493, 2008.

36 P. Holland and S. Leinhardt. A method for detecting structure in sociometric data. *American Journal of Sociology*, 76:492–513, 1970.

37 Alon Itai and Michael Rodeh. Finding a minimum circuit in a graph. *SIAM Journal on Computing*, 7(4):413–423, 1978.

38 Shweta Jain and C. Seshadhri. A Fast and Provable Method for Estimating Clique Counts Using Turán's Theorem. In *Conference on the World Wide Web (WWW)*, pages 441–449, 2017.

39 Shweta Jain and C Seshadhri. A fast and provable method for estimating clique counts using turán's theorem. In *Conference on the World Wide Web (WWW)*, pages 441–449, 2017.

40 Madhav Jha, C Seshadhri, and Ali Pinar. Path sampling: A fast and provable method for estimating 4-vertex subgraph counts. In *Conference on the World Wide Web (WWW)*, pages 495–505, 2015.

41 László Lovász. Operations with structures. *Acta Mathematica Academiae Scientiarum Hungarica*, 18(3-4):321–328, 1967.

42 László Lovász. *Large networks and graph limits*, volume 60. American Mathematical Soc., 2012.

43 David W Matula and Leland L Beck. Smallest-last ordering and clustering and graph coloring algorithms. *Journal of the ACM (JACM)*, 30(3):417–427, 1983.

44 C. St. J. A. Nash-Williams. Decomposition of finite graphs into forests. *Journal of the London Mathematical Society*, 39(1):12, 1964.

45 J. Nešetřil and P. Ossana de Mendez. *Sparsity: Graphs, Structures, and Algorithms.* Springer, 2012.

46 Mark Ortmann and Ulrik Brandes. Efficient orbit-aware triad and quad census in directed and undirected graphs. *Applied network science*, 2(1), 2017.

47 Noujan Pashanasangi and C Seshadhri. Efficiently counting vertex orbits of all 5-vertex subgraphs, by evoke. In *International Conference on Web Search and Data Mining (WSDM)*, pages 447–455, 2020.

48 Ali Pinar, C Seshadhri, and Vaidyanathan Vishal. Escape: Efficiently counting all 5-vertex subgraphs. In *Conference on the World Wide Web (WWW)*, pages 1431–1440, 2017.

49 Marc Roth and Philip Wellnitz. Counting and finding homomorphisms is universal for parameterized complexity theory. In *Proceedings of the Symposium on Discrete Algorithms (SODA)*, pages 2161–2180, 2020.

50 Thomas Schank and Dorothea Wagner. Finding, counting and listing all triangles in large graphs, an experimental study. In *Experimental and Efficient Algorithms*, pages 606–609. Springer Berlin / Heidelberg, 2005.

51 C. Seshadhri and Srikanta Tirthapura. Scalable subgraph counting: The methods behind the madness: WWW 2019 tutorial. In *Conference on the World Wide Web (WWW)*, 2019.

52 Jessica Shi, Laxman Dhulipala, and Julian Shun. Parallel clique counting and peeling algorithms. In *Proceedings of the Conference on Applied and Computational Discrete Algorithms (ACDA)*, pages 135–146, 2021.

53 K. Shin, T. Eliassi-Rad, and C. Faloutsos. Patterns and anomalies in k-cores of real-world graphs with applications. *Knowledge and Information Systems*, 54(3):677–710, 2018.

54 Siddharth Suri and Sergei Vassilvitskii. Counting triangles and the curse of the last reducer. In *Conference on the World Wide Web (WWW)*, pages 607–614, 2011.

55 Johan Ugander, Lars Backstrom, and Jon M. Kleinberg. Subgraph frequencies: mapping the empirical and extremal geography of large graph collections. In *Conference on the World Wide Web (WWW)*, pages 1307–1318, 2013.

56 Stanford Network Analysis Project (SNAP). Available at `http://snap.stanford.edu/`.

Enumerating Subgraphs of Constant Sizes in External Memory

Shiyuan Deng ✉
The Chinese University of Hong Kong, China

Francesco Silvestri ✉
University of Padova, Italy

Yufei Tao ✉
The Chinese University of Hong Kong, China

Abstract

We present an indivisible I/O-efficient algorithm for *subgraph enumeration*, where the objective is to list all the subgraphs of a massive graph $G := (V, E)$ that are isomorphic to a pattern graph Q having $k = O(1)$ vertices. Our algorithm performs $O(\frac{|E|^{k/2}}{M^{k/2-1}B} \log_{M/B} \frac{|E|}{B} + \frac{|E|^{\rho}}{M^{\rho-1}B})$ I/Os with high probability, where ρ is the fractional edge covering number of Q (it always holds $\rho \geq k/2$, regardless of Q), M is the number of words in (internal) memory, and B is the number of words in a disk block. Our solution is optimal in the class of indivisible algorithms for all pattern graphs with $\rho > k/2$. When $\rho = k/2$, our algorithm is still optimal as long as $M/B \geq (|E|/B)^{\epsilon}$ for any constant $\epsilon > 0$.

2012 ACM Subject Classification Theory of computation → Graph algorithms analysis; Information systems → Join algorithms

Keywords and phrases Subgraph Enumeration, Conjunctive Queries, External Memory, Algorithms

Digital Object Identifier 10.4230/LIPIcs.ICDT.2023.4

Funding Shiyuan Deng and Yufei Tao were supported in part by GRF Projects 14207820, 14203421, and 14222822 from HKRGC. Francesco Silvestri was supported in part by PRIN 20174LF3T8 AHeAD and Uni-Impresa Big-Mobility projects, and by the Italian National Center for HPC, Big Data, and Quantum Computing.

1 Introduction

Subgraph enumeration is the problem of listing all the subgraphs of a data graph $G := (V, E)$ that are isomorphic to a pattern graph Q. It is fundamental to a wide range of applications and has been extensively studied in computer science; see [2,3,5–8,10–12,14–18,24,26,29,38] for entry points into the literature. The problem is NP-hard [9] when Q is allowed to have arbitrarily many vertices. In practice, however, a pattern Q of interest is often *considerably* smaller than the data graph G and usually remains the same even as G increases in size over time. For these reasons, research in recent years has focused on pattern graphs Q having $O(1)$ vertices. In the random access machine (RAM) model, numerous (subgraph enumeration) algorithms [4,25,28,30–33,37] have been discovered to achieve worst-case optimal running time (sometimes up to an $O(\text{polylog}\,|E|)$ factor) on such pattern graphs.

RAM algorithms, designed to minimize CPU time, are ill-fitted for *massive* graphs G that cannot be stored in a machine's (main) memory and thus must reside at least partially in the disk. In those environments, the efficiency bottleneck is no longer CPU time, but instead, the number of *I/O accesses* transferring data between the disk and memory. As data graphs' volume continues to outgrow commodity machines' memory capacity, designing I/O-efficient solutions to subgraph enumeration has been a critical challenge. This work will present new progress in tackling the challenge that brings us close to unraveling the problem's I/O complexity.

1.1 Problem Definitions and Complexity Parameters

This subsection will formally define the subgraph enumeration problem, the computation model assumed, and the parameters used to characterize algorithm efficiency.

Subgraph Enumeration. We are given a simple undirected graph $G := (V, E)$ called the *data graph* and a simple undirected graph $Q := (V_Q, E_Q)$ called the *pattern graph*. We require that Q should have a constant number $k := |V_Q|$ of vertices (and hence, $|E_Q| = O(k^2) = O(1)$). A *subgraph* of G is defined as a simple undirected graph $G_{sub} := (V_{sub}, E_{sub})$ where $V_{sub} \subseteq V$ and $E_{sub} \subseteq E$. We call G_{sub} an *occurrence* of Q if the former is isomorphic to the latter. The goal of the subgraph enumeration problem is to enumerate all the occurrences of Q.

We assume that Q is connected (i.e., it has only one connected component), and G has no isolated vertices (i.e., vertices with no incident edges). Isolated vertices cannot participate in any occurrence and, thus, can be safely deleted. As such, $|V| \leq 2|E|$ always holds.

Computation Model. We will investigate the problem in the *external memory* (EM) model [1], the de-facto model for studying I/O-efficient algorithms. Under this model, a machine is equipped with M words of *memory* and a disk of an unbounded size that has been formatted into *blocks* of B words. The values of M and B satisfy $M \geq 2B$. A *disk I/O* – henceforth, simply *I/O* – either reads a block of data from the disk into memory or conversely writes B words from memory into a disk block. The *cost* of an algorithm is defined as the number of I/Os performed (CPU computation is for free).

For subgraph enumeration, the data graph is provided under the adjacency format in $O(|E|/B)$ disk blocks, and the pattern graph is stored in memory using $O(1)$ words. In early research (see [21, 34] and the references therein), an algorithm was required to write all the occurrences of Q to the disk. However, as the number of occurrences can be gigantic, the cost of result outputting alone may dominate an algorithm's total cost, thus hampering an effective investigation into the problem's intrinsic I/O complexity. Moreover, in some applications, disk materialization may not even be the intended approach for result reporting, e.g., an algorithm could be instructed to send out all the occurrences by network. For these reasons, the mainstream research nowadays strips off the outputting cost by introducing an *emit*(.) function. Once an occurrence of Q – say, G_{sub} – has been found, the algorithm can invoke *emit*(G_{sub}) to report G_{sub} *for free*; the algorithm is said to have *emitted* G_{sub} in that case. The algorithm must ensure that every occurrence should be emitted exactly once. It is worth mentioning that an algorithm designed to work with an *emit*(.) function can be easily adapted to produce the result in the disk with $O(\mathrm{OUT}/B)$ extra I/Os, where OUT is the total number of occurrences.

We will concentrate on the class of *indivisible* algorithms (sometimes referred to as *tuple-based* algorithms). Such an algorithm adheres to the constraint that each I/O can bring $O(B)$ edges into memory (this rule prevents, for example, encoding tricks that can compress $\omega(B)$ edges into memory). Furthermore, to emit an occurrence G_{sub}, an indivisible algorithm is required to have loaded all the edges of G_{sub} in memory simultaneously. Although the indivisible class does not capture all possible algorithms, it encapsulates all existing subgraph enumeration algorithms we are aware of, with a single exception (to be discussed in Section 1.2). Hence, understanding the optimal I/O complexity achievable by this class offers meaningful insight into the problem's characteristics.

Fractional Edge Covering Numbers. Next, we will introduce the *fractional edge covering number*, a notion from graph theory that plays an imperative role in characterizing the I/O complexity of subgraph enumeration. As before, let $Q := (V_Q, E_Q)$ be the input pattern graph. Define W as a function that maps each edge $e \in E_Q$ to a real-valued weight $W(e) \geq 0$. We call W a *fractional edge covering* of Q if it holds for every vertex $v \in V_Q$ that:

$$\sum_{e \in E_Q : e \text{ incident to } v} W(e) \geq 1.$$

We refer to $\sum_{e \in E_Q} W(e)$ as the *total weight* of W. The *fractional edge covering number* of Q, denoted as ρ, is the smallest total weight of all the fractional edge coverings of Q.

Complexity Parameters and Math Conventions. We will measure the I/O cost of a subgraph enumeration algorithm using five parameters: $|E|$, ρ, k, M, and B. Whenever a random event is said to hold "with high probability" – w.h.p. for short – we require the event to hold with probability at least $1 - 1/|E|^{\xi}$, where ξ can be set to an arbitrarily large constant. For an integer $x \geq 1$, $[x]$ represents the set $\{1, 2, ..., x\}$. We define sort(n) to be the I/O complexity of sorting n elements; it is known [1] that sort$(n) = O(\lceil \frac{n}{B} \rceil \log_{M/B} \lceil \frac{n}{B} \rceil)$.

1.2 Previous Work

In the EM model, research on subgraph enumeration started with *triangle enumeration*, where the pattern graph is $Q :=$ 3-clique, a.k.a., triangle. Pagh and Silvestri [35] gave a randomized algorithm that can emit the triangles of a data graph $G := (V, E)$ in $O(|E|^{1.5}/(\sqrt{M}B))$ I/Os in expectation. They also de-randomized their algorithm to obtain a deterministic I/O bound of $O(\frac{|E|^{1.5}}{\sqrt{M}B} \log_{M/B} \frac{|E|}{B})$ [35]. Their result was later improved by Hu, Qiao, and Tao [20], who managed to emit all triangles deterministically in $O(|E|^{1.5}/(\sqrt{M}B))$ I/Os. The result of [20] matches an I/O lower bound of $\Omega(|E|^{1.5}/(\sqrt{M}B))$ [21, 35] on indivisible algorithms.

The lower bound argument of [21, 35] can be extended [19, 20] to show that, for any pattern graph Q (of a constant size), every indivisible algorithm – no matter randomized or deterministic – must perform $\Omega(|E|^{\rho}/(M^{\rho-1}B))$ I/Os in the worst case to emit all the occurrences of Q, where ρ is the fractional edge covering number of Q (for triangle, $\rho = 1.5$). Matching this lower bound for arbitrary Q has been an intriguing open problem. In [19], Hu and Yi developed a deterministic algorithm that achieves an I/O complexity of $O(\frac{|E|^{\rho}}{M^{\rho-1}B} \cdot \log_{M/B} \frac{|E|}{B})$ for any acyclic pattern graph Q; their algorithm, however, does not work for cyclic Q. In [27], Koutris, Beame, and Suciu presented a technique that can convert an algorithm from the so-called *massively parallel computation* (MPC) model to an algorithm in EM. By combining their technique with a recent MPC algorithm of Ketsman, Suciu, and Tao [23], one can obtain a randomized EM algorithm that can solve, w.h.p., the subgraph enumeration problem for any pattern graph Q in $O(\frac{|E|^{\rho}}{M^{\rho-1}B} \cdot \text{polylog}\,|E|)$ I/Os, as long as $M \geq |E|^c$ where $c < 1$ is a positive constant dependent on Q.

All the above algorithms are indivisible. Outside the indivisible class, we are aware of only one algorithm due to Eppstein et al. [13], which is designed for triangle enumeration. Their (randomized) solution guarantees an expected I/O cost of $O(\text{sort}(\alpha|V|) + \text{sort}(|E| \lceil \frac{\alpha \log wlen}{wlen} \rceil) + \text{sort}(\text{OUT}))$ where OUT is the number of occurrences, *wlen* is the number of bits in a word, and α is the arboricity value of G (the algorithm was designed for writing all occurrences to the disk; it can also be deployed for result *emission*, but the I/O complexity does not decrease). The value of α falls between 1 and $O(\sqrt{E})$. Compared to the aforementioned indivisible solutions [20, 35] to triangle enumeration, the algorithm of [13] may have a lower I/O complexity when α and OUT are sufficiently small.

Table 1 Comparison of our and previous results on subgraph enumeration.

pattern Q	I/O cost in big-O	source	remark
triangle	$\|E\|^{1.5}/(\sqrt{M}B)$ expected	[35]	rand.
triangle	$\frac{\|E\|^{1.5}}{\sqrt{M}B}\log_{M/B}\frac{\|E\|}{B}$	[35]	det.
triangle	$\|E\|^{1.5}/(\sqrt{M}B)$	[20]	det.
triangle	$\text{sort}(\alpha\|V\|)+\text{sort}(\|E\|\lceil\frac{\alpha\log wlen}{wlen}\rceil)$ $+\text{sort}(\text{OUT})$ expected	[13]	$\alpha :=$ arboricity of G $wlen :=$ word length rand., outside indivisible class
acyclic	$\frac{\|E\|^{\rho}}{M^{\rho-1}B}\log_{M/B}\frac{\|E\|}{B}$	[19]	det.
arbitrary	$\frac{\|E\|^{\rho}}{M^{\rho-1}B}\cdot\text{polylog}\,\|E\|$ w.h.p.	[23]	rand., needs $M \geq \|E\|^{c}$ for some Q-dependent constant $c \in (0,1)$
arbitrary	$\frac{\|E\|^{k/2}}{M^{k/2-1}B}\log_{\frac{M}{B}}\frac{\|E\|}{B}+\frac{\|E\|^{\rho}}{M^{\rho-1}B}$ w.h.p. and expected	ours	rand., optimal when $\rho > \frac{k}{2}$ or $\frac{M}{B} \geq (\frac{\|E\|}{B})^{\epsilon}$ for any constant $\epsilon > 0$

1.3 Our Contributions

The main result of this paper is:

▶ **Theorem 1.** *Let $G := (V, E)$ be a simple undirected graph with no isolated vertices. Let $Q := (V_Q, E_Q)$ be a simple undirected connected pattern graph with $k := O(1)$ vertices. When $|E| \geq M$, there is an algorithm in EM that, with high probability, emits every occurrence of Q in G exactly once with $O(\frac{|E|^{k/2}}{M^{k/2-1}B}\log_{M/B}\frac{|E|}{B}+\frac{|E|^{\rho}}{M^{\rho-1}B})$ I/Os, where ρ is the fractional edge covering number of Q, M is the number of words in memory, and B is the number of words in a disk block. The same I/O complexity holds also in expectation.*

The theorem applies to all M and B satisfying $M \geq 2B$. The value of ρ is at least $k/2$ for all pattern graphs Q, but can reach $k-1$ for some Q [36]. Our algorithm is indivisible; when $\rho > k/2$, its I/O complexity becomes $O(\frac{|E|^{\rho}}{M^{\rho-1}B})$, matching the indivisible lower bound $\Omega(\frac{|E|^{\rho}}{M^{\rho-1}B})$ (see Section 1.2). When $\rho = k/2$, the algorithm is still optimal as long as $M/B \geq (|E|/B)^{\epsilon}$ for an arbitrarily small constant $\epsilon > 0$ (a condition likely to hold in reality). Table 1 presents a comparison between our and previous results.

2 Preliminaries

We will cast subgraph enumeration as a join problem for two reasons. First, it permits us to simplify presentation by leveraging relational algebra's expressive power. Second, our algorithm has a crucial connection to the isolated cartesian product theorem recently developed by Ketsman, Suciu, and Tao [23], which is stated on joins and still lacks an intuitive interpretation on graphs currently. In Section 2.1, we will define the relevant concepts of joins, formulate the join enumeration problem in EM, and review a textbook join algorithm. In Section 2.2, we will explain how to reduce subgraph enumeration to joins. Finally, in Section 2.3, we will introduce a concentration bound that will be useful in our analysis.

2.1 Joins on Binary Relations

Joins. Define **att** as an arbitrary finite set of *attributes*. A *tuple* over a set $U \subseteq \textbf{att}$ of attributes is a function $\boldsymbol{t} : U \to \textbf{dom}$, where **dom** is an arbitrary infinite set. For any $U_{sub} \subseteq U$, we define $\boldsymbol{t}[U_{sub}]$ as the tuple $\boldsymbol{t_{sub}}$ over U_{sub} such that $\boldsymbol{t_{sub}}(X) = \boldsymbol{t}(X)$ for every

$X \in U_{sub}$. A *relation* is a set R of tuples over the same set U of attributes; the *schema* of R – denoted as $schema(R)$ – is U. R is *unary* if $schema(R)$ has one attribute, or *binary* if $schema(R)$ has two attributes.

We define a *join* as a set $\mathcal{Q}$ of relations. Let $schema(\mathcal{Q}) := \bigcup_{R \in \mathcal{Q}} schema(R)$. The join result, denoted as $join(\mathcal{Q})$, is a relation over $schema(\mathcal{Q})$ that can be formalized as

$$join(\mathcal{Q}) := \{\text{tuple } \boldsymbol{t} \text{ over } schema(\mathcal{Q}) \mid \forall R \in \mathcal{Q} : \boldsymbol{t}[schema(R)] \in R\}.$$

The *input size* of $\mathcal{Q}$ is defined as $\sum_{R \in \mathcal{Q}} |R|$, namely, the total number of tuples in all relations. We will call $\mathcal{Q}$ a *binary join* if all its relations are binary.

Schema Graphs. We define the *schema graph* of a join $\mathcal{Q}$ as the hypergraph $\mathcal{G} := (\mathcal{X}, \mathcal{E})$ where $\mathcal{X} := schema(\mathcal{Q})$ and $\mathcal{E} := \{schema(R) \mid R \in \mathcal{Q}\}$. We will consistently refer to the vertices in $\mathcal{X}$ as "attributes" and to the elements in $\mathcal{E}$ as "hyperedges". Note that $\mathcal{G}$ is a "hyper"-graph, rather than just a "graph", because each of its hyperedges may not have exactly two attributes (e.g., if a relation $R \in \mathcal{Q}$ is unary, then the hyperedge $schema(R) \in \mathcal{E}$ has only one attribute). Moreover, $\mathcal{G}$ may have identical hyperedges (this happens when two relations in $\mathcal{Q}$ have the same schema). A hyperedge $e \in \mathcal{E}$ is *unary* if $|e| = 1$, or *binary* if $|e| = 2$. Two vertices $X_1 \in X$ and $X_2 \in X$ are *adjacent* in $\mathcal{G}$ if there exists a hyperedge $e \in \mathcal{E}$ containing both X_1 and X_2.

A function W mapping each hyperedge $e \in \mathcal{E}$ to a non-negative real value $W(e)$ is called a *fractional edge covering* of $\mathcal{G}$ if it satisfies the following condition: for every attribute $X \in \mathcal{X}$, $\sum_{e \in \mathcal{E}: X \in e} W(e) \geq 1$, namely, the weights of all the hyperedges containing X add up to at least 1. The *total weight* of W is defined as $\sum_{e \in \mathcal{E}} W(e)$. The *fractional edge covering number* of $\mathcal{G}$ is the smallest total weight of all the fractional edge coverings of $\mathcal{G}$.

Active Domains and Degrees. Let $\mathcal{Q}$ be a binary join with schema graph $\mathcal{G} := (\mathcal{X}, \mathcal{E})$. For each attribute $X \in \mathcal{X}$, we define the *active domain* of X as $\mathbf{adom}(X) := \bigcup_{R \in \mathcal{Q}: X \in schema(R)} \{\boldsymbol{t}(X) \mid \boldsymbol{t} \in R\}$. Henceforth, we will take the view that the attributes in $schema(\mathcal{Q})$ have mutually disjoint active domains (this loses no generality because one can conceptually prefix each value with an attribute name, if necessary). Define the *combined active domain* of $\mathcal{Q}$ as

$$\mathbf{adom} := \bigcup_{X \in schema(\mathcal{Q})} \mathbf{adom}(X). \tag{1}$$

Fix any attribute $X \in \mathcal{X}$ and any value $v \in \mathbf{adom}(X)$. We define the *degree* of v as

$$\max_{R \in \mathcal{Q}: X \in schema(R)} |\{\boldsymbol{u} \in R \mid \boldsymbol{u}(X) = v\}|.$$

Intuitively, the degree tells us at most how many tuples can carry value v under attribute X in a relation of $\mathcal{Q}$. Moreover, define

$$\text{degree of } \mathcal{Q} := \max_{v \in \mathbf{adom}} \text{degree of } v. \tag{2}$$

Join Result Enumeration in EM. Let $\mathcal{Q}$ be a binary join with input size $N := \sum_{R \in \mathcal{Q}} |R|$ and schema graph $\mathcal{G}$. We will study the evaluation of $\mathcal{Q}$ under the EM model, assuming that $\mathcal{G}$ has $O(1)$ attributes. At the beginning of an algorithm, each relation $R \in \mathcal{Q}$ is stored in $O(|R|/B)$ consecutive blocks in the disk, and $\mathcal{G}$ is stored in memory using $O(1)$ words. Result reporting is done through a special function *emit*(.): every time the algorithm finds a tuple $\boldsymbol{t} \in join(\mathcal{Q})$, it can *emit* $\boldsymbol{t}$ for free by calling *emit*($\boldsymbol{t}$). Every tuple in $join(\mathcal{Q})$ should be emitted exactly once. If the algorithm is randomized, we will use the statement "an event holds with high probability (w.h.p.)" to state that the event holds with probability at least $1 - 1/N^\xi$, where ξ can be an arbitrarily large constant.

Blocked Nested Loop (BNL). This textbook algorithm works for arbitrary joins:

▶ **Lemma 2.** *Let $\mathcal{Q}$ be a join with $r = O(1)$ input relations. The BNL algorithm emits every tuple of join($\mathcal{Q}$) exactly once in $O(\lceil \frac{N^r}{M^{r-1}B} \rceil)$ I/Os, where $N := \sum_{R \in \mathcal{Q}} |R|$, M is the number of words in memory, and B is the number of words in a disk block.*

The proof is trivial and omitted. BNL will serve as a building block in our algorithms.

2.2 Reduction from Subgraph Enumeration to Binary Joins

We can convert subgraph enumeration to binary-join evaluation with no degradation in terms of worst-case I/O complexity. Consider an instance of subgraph enumeration with data graph $G := (V, E)$ and pattern graph $Q := (V_Q, E_Q)$. We create a binary join $\mathcal{Q}$ on $|E_Q|$ relations by executing the following steps for each edge $\{X_1, X_2\} \in E_Q$ (where X_1 and X_2 are distinct vertices in V_Q):

- Add a relation R to $\mathcal{Q}$ with schema *schema*$(R) := \{X_1, X_2\}$.
- For each edge $\{u, v\} \in E$ (where u and v are distinct vertices in V), define a tuple $\boldsymbol{t_1}$ with $\boldsymbol{t_1}(X_1) := u$ and $\boldsymbol{t_1}(X_2) := v$, and another tuple $\boldsymbol{t_2}$ with $\boldsymbol{t_2}(X_1) := v$ and $\boldsymbol{t_2}(X_2) := u$. Add both $\boldsymbol{t_1}$ and $\boldsymbol{t_2}$ to R.

The above conversion has several properties. First, the schema graph $\mathcal{G} := (\mathcal{X}, \mathcal{E})$ of $\mathcal{Q}$ is isomorphic to the pattern graph Q. Second, each relation $R \in \mathcal{Q}$ has $2|E|$ tuples such that the input size of $\mathcal{Q}$ is $2|E| \cdot |E_Q| = \Theta(|E|)$. Third, the relations in $\mathcal{Q}$ have distinct schemas.

The lemma below, which is proved in Appendix A, shows that an efficient algorithm for evaluating $\mathcal{Q}$ implies an efficient algorithm for performing subgraph enumeration on G.

▶ **Lemma 3.** *Consider any input to the subgraph enumeration problem with data graph G and pattern graph Q. Let $\mathcal{Q}$ be the join constructed in the way explained above. If we have an algorithm to emit all the tuples of join($\mathcal{Q}$) in T I/Os w.h.p., then we can emit every occurrence of Q in G exactly once using $T + O(\lceil |E|/B \rceil)$ I/Os w.h.p..*

By virtue of the above lemma, we will turn our attention to joins on binary relations.

2.3 A Concentration Bound under Partial Dependence

Next, we will review a Chernoff-like result due to Janson [22]. Let $X_1, X_2, ..., X_n$ be random variables satisfying $X_i - \mathbf{E}[X_i] \leq 1$ for all $i \in [n]$; these variables may *not* follow the same distribution. Suppose that we are also given a dependency graph G_{dep} with $\{X_1, X_2, ..., X_n\}$ as the vertex set. G_{dep} must fulfill the following *independence requirement*: for any $S \subseteq \{X_1, X_2, ..., X_n\}$ and any vertex $X_i \notin S$ (for some $i \in [n]$), if X_i is not adjacent to any vertex in S, then X_i is independent of the joint distribution of the variables in S. In Theorem 2.3 of [22], Janson proved:

▶ **Lemma 4** ([22]). *Set $X := \sum_{i=1}^n X_i$, $\mu := \mathbf{E}[X]$, and σ to any value at least $\sum_{i=1}^n \mathbf{Var}(X_i)$. Define Δ to be the maximum vertex degree in G_{dep}. Then, for any $\epsilon > 0$, it holds that*

$$\Pr[X \geq (1+\epsilon)\mu] \quad \leq \quad \exp\left(-\frac{8\epsilon^2 \cdot \mu^2}{25\Delta(\sigma + \epsilon \cdot \mu/3)}\right). \tag{3}$$

3 An EM Algorithm for Binary Joins of Bounded Degrees

This section serves as a proof of:

▶ **Lemma 5.** *Consider a binary join $\mathcal{Q}$ whose relations have distinct schemas. Let $\mathcal{G} := (\mathcal{X}, \mathcal{E})$ be the schema graph of $\mathcal{Q}$, and set $N := \sum_{R \in \mathcal{Q}} |R|$ and $k := |\mathcal{X}|$. Fix any value $\lambda \geq \sqrt{NM}$, where M is the number of words in memory. If $N \geq M$ and $\mathcal{Q}$ has a degree at most λ, there is an EM algorithm that, with probability at least $1 - 1/\lambda^{\xi}$, emits every tuple of join($\mathcal{Q}$) exactly once in $O(\lambda^k/(M^{k-1}B))$ I/Os, where ξ can be an arbitrarily large constant, and B is the number of words in a disk block.*

Note that the success probability is expressed using λ rather than N. This will be an essential feature in Section 4 where we utilize the lemma as a subroutine to tackle general binary joins. To prove Lemma 5, we consider only $k \geq 3$; if $k = 2$, $\mathcal{G}$ has only two attributes – namely, $\mathcal{Q}$ has only one single relation – in which case we can trivially emit the tuples of *join*($\mathcal{Q}$) exactly once in $O(N/B)$ I/Os. When $k \geq 3$, it holds that $\text{sort}(N) = O(N^{1.5}/(\sqrt{M}B)) = O(\lambda^k/(M^{k-1}B))$.

3.1 An Algorithmic Framework

We will describe a high-level framework for evaluating the join $\mathcal{Q}$ in Lemma 5. Depending on M, we will instantiate the framework differently in Sections 3.2 and 3.3, which together will make a complete algorithm with the guarantees in Lemma 5.

Coloring. Set $r := |\mathcal{Q}|$, i.e., the number of relations in $\mathcal{Q}$ (also the number of hyperedges in $\mathcal{E}$). Furthermore, define

$$s := \lceil \lambda/M \rceil \tag{4}$$

and assume that we are given a function

$$\Gamma : \mathbf{adom} \to [s]. \tag{5}$$

Recall that **adom** is the combined active domain of $\mathcal{Q}$; see (1). We will refer to each possible output of Γ as a *color*; in other words, Γ maps each value of **adom** to a color in $[s]$. We will also assume that a *coloring step* has been performed to color all the tuples by Γ; namely, for every relation $R \in \mathcal{Q}$, any tuple $\boldsymbol{t} \in R$, and each attribute $X \in$ *schema*(R), the color $\Gamma(\boldsymbol{t}(X))$ is stored together with $\boldsymbol{t}$ (this means two extra words for each tuple). The choice of Γ (henceforth named the *coloring function*), as well as the coloring step, is the key to instantiating our algorithmic framework.

Color Schemes. We can divide the join result *join*($\mathcal{Q}$) by how the tuples therein are colored by Γ. We say that two tuples $\boldsymbol{t_1}$ and $\boldsymbol{t_2}$ in *join*($\mathcal{Q}$) have the same *color scheme* if $\Gamma(\boldsymbol{t_1}(X)) = \Gamma(\boldsymbol{t_2}(X))$ for every attribute $X \in \mathcal{X}$. Formally, a *color scheme* is a function

$$\gamma : \mathcal{X} \to [s]. \tag{6}$$

As each attribute can be colored any value in $[s]$, there are in total $s^{|\mathcal{X}|} = s^k$ color schemes. Every color scheme γ spawns a join of its own. For each relation $R \in \mathcal{Q}$, define

$$R_\gamma := \{\boldsymbol{t} \in R \mid \Gamma(\boldsymbol{t}(X)) = \gamma(X) \text{ for all } X \in \textit{schema}(R)\}.$$

Intuitively, R_γ is the subset of tuples in R that are colored by Γ in a way consistent with γ. We can now define a join induced by γ:

$$\mathcal{Q}_\gamma := \{R_\gamma \mid R \in \mathcal{Q}\}.$$

The sets *join*($\mathcal{Q}_\gamma$) of all color schemes γ are mutually disjoint and their union is *join*($\mathcal{Q}$).

Algorithm. In Appendix B, we show that, after a preprocessing step with I/O cost $O(\text{sort}(N))$, we can store the input relations of $\mathcal{Q}_\gamma$ – for every color scheme γ – in consecutive disk blocks. Then, for each γ, we deploy the BNL algorithm of Lemma 2 to emit the tuples of *join*($\mathcal{Q}_\gamma$). This completes the algorithm for evaluating $\mathcal{Q}$.

Analysis. By Lemma 2, the BNL execution of all s^k color schemes incurs a total I/O cost of

$$O\left(\sum_\gamma \left\lceil \frac{N_\gamma}{M} \right\rceil^{r-1} \left\lceil \frac{N_\gamma}{B} \right\rceil\right) = O\left(s^k + \sum_\gamma \frac{{N_\gamma}^r}{M^{r-1}B}\right) = O\left(s^k + \frac{\sum_\gamma \sum_{R \in \mathcal{Q}} |R_\gamma|^r}{M^{r-1}B}\right) \tag{7}$$

where N_γ is the input size of $\mathcal{Q}_\gamma$, and the second equality used the fact that $|\mathcal{Q}|$ has only a constant number of relations.

To facilitate the analysis of (7), let us impose an arbitrary ordering on the attributes in $\mathcal{X}$; we use the notation $X_1 < X_2$ to denote the fact that attribute $X_1 \in \mathcal{X}$ precedes another attribute $X_2 \in \mathcal{X}$ in the ordering. Fix any two colors $c_1 \in [s]$ and $c_2 \in [s]$. For each relation $R \in \mathcal{Q}$ whose *schema*(R) has attributes X_1 and X_2 with $X_1 < X_2$, define

$$R_{c_1,c_2} := \{\boldsymbol{t} \in R \mid \Gamma(\boldsymbol{t}(X_1)) = c_1 \text{ and } \Gamma(\boldsymbol{t}(X_2)) = c_2\};$$

namely, R_{c_1,c_2} includes every tuple of R that receives colors c_1 and c_2 on attributes X_1 and X_2, respectively. We can now derive:

$$(7) = O\Big(s^k + \frac{\sum_{R \in \mathcal{Q}} \sum_\gamma |R_\gamma|^r}{M^{r-1}B}\Big) = O\Big(s^k + \frac{s^{k-2}}{M^{r-1}B} \sum_{R \in \mathcal{Q}} \sum_{c_1,c_2 \in [s]} |R_{c_1,c_2}|^r\Big) \tag{8}$$

where the second equality holds because each pair (c_1, c_2) is relevant to s^{k-2} color schemes.

Given the value of s in (4), the term s^k is $O((\lambda/M)^k) = O(\lambda^k/(M^{k-1}B))$. What is non-trivial is to argue that the term $\frac{s^{k-2}}{M^{r-1}B} \sum_{R \in \mathcal{Q}} \sum_{c_1,c_2 \in [s]} |R_{c_1,c_2}|^r$ can also be bounded by $O(\lambda^k/(M^{k-1}B))$. We will do so by choosing the coloring function Γ carefully according to the memory size M.

3.2 When $M = O(\lambda/\log^2 \lambda)$

We will first explain how to choose the coloring function Γ to ensure that the algorithm described in Section 3.1 performs $O(\lambda^k/(M^{k-1}B))$ I/Os in expectation. Then, we will slightly modify the algorithm to achieve the same I/O complexity with probability at least $1 - 1/\lambda^\xi$.

Choice of Γ. We decide Γ by independently mapping each value of **adom** to a color chosen uniformly at random from $[s]$. This Γ can be stored as a list of (value, color) pairs in $O(N/B)$ blocks. The coloring step (as defined in Section 3.1) can then be performed in sort(N) I/Os.

Identically-Colored Subsets. Let us first study a probability question that arises from our analysis. Take an arbitrary relation $R \in \mathcal{Q}$. Let X_1 and X_2 be the two attributes in $schema(R)$; w.l.o.g., assume $X_1 < X_2$ (recall from Section 3.1 that we have imposed an arbitrary ordering on the attributes). Given an integer $i \in [r]$, define an *i-subset* of R to be a subset $S \subseteq R$ with $|S| = i$. We say that S is *identically colored* if all the tuples in S belong to the same color scheme; in other words, for any $\boldsymbol{t_1}, \boldsymbol{t_2} \in S$, it holds that $\Gamma(\boldsymbol{t_1}(X_1)) = \Gamma(\boldsymbol{t_2}(X_1))$ and $\Gamma(\boldsymbol{t_1}(X_2)) = \Gamma(\boldsymbol{t_2}(X_2))$. Define:

$$Y_i \quad := \quad \text{the number of identically colored } i\text{-subsets of } R. \tag{9}$$

Note that Y_i is a random variable because its value varies with Γ. We want to understand how large Y_i is in expectation. The lemma below provides an answer.

▶ **Lemma 6.** $\mathbf{E}[Y_i] = O(\lambda^2 \cdot M^{i-2})$ *for each* $i \in [r]$.

Proof. Recall from the statement of Lemma 5 that $\lambda \geq \sqrt{NM}$. Next, we will use induction to prove the claim "$\mathbf{E}[Y_i] \leq (4r^2 \cdot M)^{i-1}N$ for all $i \in [r]$", which will establish the lemma because $M^{i-1}N \leq \lambda^2 \cdot M^{i-2}$. For $i = 1$, $\mathbf{E}[Y_1]$ is trivially bounded by N; hence, the claim holds at $i = 1$.

Assuming the claim's correctness for $i = j - 1$ where $j \geq 2$, next we give the proof for $i = j$. Consider, w.l.o.g., $|R| \geq j$ (otherwise, $Y_j = 0$ and the claim is vacuously true). Given a $(j-1)$- or j-subset S of R, we define $Z(S)$ to be 1 if S is identically colored; otherwise, $Z(S) := 0$. Impose an arbitrary ordering on the tuples of R. Given a j-subset S_j of R, we can list the tuples of S_j in ascending order as $\boldsymbol{t_1}, \boldsymbol{t_2}, ..., \boldsymbol{t}_j$. We call S_j an *extension* of the $(j-1)$-subset $S_{j-1} := \{\boldsymbol{t_1}, \boldsymbol{t_2}, ..., \boldsymbol{t}_{j-1}\}$. Next, we discuss the relationship between $\Pr[Z(S_{j-1}) = 1]$ and $\Pr[Z(S_j) = 1]$ by distinguishing three cases.

- Case 1: $\boldsymbol{t}_j(X_1) \in \Pi_{X_1}(S_{j-1})$ and $\boldsymbol{t}_j(X_2) \in \Pi_{X_2}(S_{j-1})$. Clearly, $Z(S_j)$ always equals $Z(S_{j-1})$ and, hence, $\Pr[Z(S_{j-1}) = 1] = \Pr[Z(S_j) = 1]$.
- Case 2: $\boldsymbol{t}_j(X_1) \in \Pi_{X_1}(S_{j-1})$ but $\boldsymbol{t}_j(X_2) \notin \Pi_{X_2}(S_{j-1})$. First note that if $Z(S_{j-1})$ is 0, so must be $Z(S_j)$. Consider now $Z(S_{j-1}) = 1$; hence, Γ maps all the values in $\Pi_{X_2}(S_{j-1})$ to the same color, say, $c \in [s]$. $Z(S_j) = 1$ if and only if $\Gamma(\boldsymbol{t}_j(X_2)) = c$. Thus, $\Pr[Z(S_j) = 1] = \Pr[Z(S_{j-1}) = 1] \cdot \Pr[\Gamma(\boldsymbol{t}_j(X_2)) = c \mid Z(S_{j-1}) = 1] = \Pr[Z(S_{j-1}) = 1]/s$.
- Case 3: $\boldsymbol{t}_j(X_1) \notin \Pi_{X_1}(S_{j-1})$ but $\boldsymbol{t}_j(X_2) \in \Pi_{X_2}(S_{j-1})$. This is symmetric to Case 2, and we also have $\Pr[Z(S_j) = 1] = \Pr[Z(S_{j-1}) = 1]/s$.
- Case 4: $\boldsymbol{t}_j(X_1) \notin \Pi_{X_1}(S_{j-1})$ and $\boldsymbol{t}_j(X_2) \notin \Pi_{X_2}(S_{j-1})$. Again, if $Z(S_{j-1})$ is 0, so must be $Z(S_j)$. When $Z(S_{j-1}) = 1$, Γ maps (i) all the values in $\Pi_{X_1}(S_{j-1})$ to the same color, say, $c_1 \in [s]$, and (ii) all the values in $\Pi_{X_2}(S_{j-1})$ to the same color, say, $c_2 \in [s]$. $Z(S_j) = 1$ if and only if $\Gamma(\boldsymbol{t}_j(X_1)) = c_1$ and $\Gamma(\boldsymbol{t}_j(X_2)) = c_2$. Hence, $\Pr[Z(S_j) = 1] = \Pr[Z(S_{j-1}) = 1] \cdot \Pr[\Gamma(\boldsymbol{t}_j(X_1)) = c_1, \Gamma(\boldsymbol{t}_j(X_2)) = c_2 \mid Z(S_{j-1}) = 1] = \Pr[Z(S_{j-1}) = 1]/s^2$.

Denote by $\mathcal{S}_j$ and $\mathcal{S}_{j-1}$ the set of all j- and $(j-1)$-subsets of R, respectively. We bound $\mathbf{E}[Y_j] = \sum_{S_j \in \mathcal{S}_j} \Pr[Z(S_j) = 1]$ using a charging argument. Each $S_j \in \mathcal{S}_j$ is the extension of a unique $(j-1)$-subset S_{j-1}. If S_j is a Case-1 extension, we charge a weight of 1 on S_{j-1}; if S_j is a Case -2 or -3 extension, we charge a weight of $1/s$ on S_{j-1}; if S_j is a Case -4 extension, we charge a weight of $1/s^2$ on S_{j-1}. Note that each $S_{j-1} \in \mathcal{S}_{j-1}$ can be charged more than once because S_{j-1} can have multiple extensions. The above discussion implies

$$\mathbf{E}[Y_j] = \sum_{S_j \in \mathcal{S}_j} \Pr[Z(S_j) = 1] = \sum_{S_{j-1} \in \mathcal{S}_{j-1}} \Pr[Z(S_{j-1}) = 1] \cdot \text{total weight charged on } S_{j-1}. \tag{10}$$

To analyze how much total weight can be charged on a $(j-1)$-subset S_{j-1} of R, observe:

- S_{j-1} has at most $(j-1)^2$ extensions of Case 1. Such an extension must add to S_{j-1} a tuple $\boldsymbol{t}_j$ satisfying $\boldsymbol{t}_j(X_1) \in \Pi_{X_1}(S_{j-1})$ and $\boldsymbol{t}_j(X_2) \in \Pi_{X_2}(S_{j-1})$. Since each of $\Pi_{X_1}(S_{j-1})$ and $\Pi_{X_2}(S_{j-1})$ has size at most $j-1$, at most $(j-1)^2$ tuples can be selected as $\boldsymbol{t}_j$.
- S_{j-1} has at most $(j-1)\lambda$ extensions of Case 2. Such an extension must add to S_{j-1} a tuple $\boldsymbol{t}_j \in R$ satisfying $\boldsymbol{t}_j(X_1) = v$, for some $v \in \Pi_{X_1}(S_{j-1})$. As the degree of v is at most λ (by definition of λ), at most λ tuples in R can be selected as $\boldsymbol{t}_j$. The bound $(j-1)\lambda$ thus follows from the fact $|\Pi_{X_1}(S_{j-1})| \leq j-1$.
- Symmetrically, S_{j-1} has at most $(j-1)\lambda$ extensions of Case 3.
- Trivially, S_{j-1} has at most N extensions of Case 4.

It thus follows that the total weight charged on S_{j-1} is at most $(j-1)^2 + \frac{2(j-1)\lambda}{s} + \frac{N}{s^2}$, which is at most $4r^2M$ given the value of s in (4). We can then obtain from (10):

$$\mathbf{E}[Y_j] \leq \sum_{S_{j-1} \in \mathcal{S}_{j-1}} \Pr[Z(S_{j-1}) = 1] \cdot (4r^2M) = \mathbf{E}[Y_{j-1}] \cdot (4r^2M) \leq (4r^2M)^{j-1}N$$

where the last inequality used our inductive assumption $\mathbf{E}[Y_{j-1}] \leq (4r^2M)^{j-2}N$. ◀

I/O Cost in Expectation. We now proceed to analyze the expected I/O cost of the algorithm in Section 3.1. The lemma below is essentially a corollary of Lemma 6.

▶ **Lemma 7.** *For any $R \in \mathcal{Q}$, $\mathbf{E}[\sum_{c_1,c_2 \in [s]} |R_{c_1,c_2}|^r] = O(\lambda^2 \cdot M^{r-2})$.*

Proof. Because r is a constant, $\sum_{c_1,c_2 \in [s]} |R_{c_1,c_2}|^r = O(s^2 + \sum_{c_1,c_2 \in [s]:|R_{c_1,c_2}| \geq r} \binom{|R_{c_1,c_2}|}{r})$, where the term $O(s^2)$ accounts for the at most s^2 pairs of (c_1, c_2) satisfying $|R_{c_1,c_2}| < r$.[1] Observe that $\sum_{c_1,c_2 \in [s]:|R_{c_1,c_2}| \geq r} \binom{|R_{c_1,c_2}|}{r}$ is exactly Y_r as defined in (9). Hence, $\mathbf{E}[\sum_{c_1,c_2 \in [s]} |R_{c_1,c_2}|^r] = \mathbf{E}[O(s^2 + Y_r)] = O(\lambda^2 \cdot M^{r-2})$ (here, we applied Lemma 6 and the value of s in (4)). ◀

Hence, the term $\frac{s^{k-2}}{M^{r-1}B} \sum_{R \in \mathcal{Q}} \sum_{c_1,c_2 \in [s]} |R_{c_1,c_2}|^r$ has an expectation of $O(\frac{s^{k-2}}{M^{r-1}B} \cdot \lambda^2 \cdot M^{r-2}) = O(\lambda^k/(M^{k-1}B))$. We can now conclude that the algorithm in Section 3.1 has an expected I/O cost of $O(\lambda^k/(M^{k-1}B))$ overall.

Achieving High Probability. Our analysis indicates that the I/O cost is $O(\lambda^k/(M^{k-1}B))$ as long as $\sum_{c_1,c_2 \in [s]} |R_{c_1,c_2}|^r = O(\lambda^2 \cdot M^{r-2})$ for every $R \in \mathcal{Q}$. By Markov inequality, the probability for $\sum_{c_1,c_2 \in [s]} |R_{c_1,c_2}|^r$ to exceed $2r \cdot \mathbf{E}[\sum_{c_1,c_2 \in [s]} |R_{c_1,c_2}|^r] = O(\lambda^2 \cdot M^{r-2})$ is at most $1/(2r)$. The union bound then assures us that, with probability at least $1/2$, $\sum_{c_1,c_2 \in [s]} |R_{c_1,c_2}|^r = O(\lambda^2 \cdot M^{r-2})$ holds for all the r relations $R \in \mathcal{Q}$. Once the coloring function Γ has been chosen, by sorting, we can obtain the precise value $\sum_{c_1,c_2 \in [s]} |R_{c_1,c_2}|^r$ for every $R \in \mathcal{Q}$ in sort(N) I/Os. As long as any $\sum_{c_1,c_2 \in [s]} |R_{c_1,c_2}|^r$ falls out of $O(\lambda^2 \cdot M^{r-2})$, we repeat from scratch by choosing another Γ. It takes $O(\log \lambda)$ repeats to ensure $\sum_{c_1,c_2 \in [s]} |R_{c_1,c_2}|^r = O(\lambda^2 \cdot M^{r-2})$ for all $R \in \mathcal{Q}$ with probability at least $1 - 1/\lambda^\xi$ for an arbitrarily large constant ξ. With the above modification, our algorithm has an I/O cost $O(\text{sort}(N) \cdot \log \lambda + \lambda^k/(M^{k-1}B))$ with probability at least $1 - 1/\lambda^\xi$. The complexity is $O(\lambda^k/(M^{k-1}B))$ as long as $M = O(\lambda/\log^2 \lambda)$.

[1] Every such pair can contribute at most $(r-1)^r = O(1)$ to $\sum_{c_1,c_2 \in [s]} |R_{c_1,c_2}|^r$.

3.3 When $M = \Omega((\lambda \log \lambda)^{2/3})$

This subsection will present another instantiation of the framework in Section 3.1 that runs in $O(\lambda^k/(M^{k-1}B))$ I/Os with probability at least $1 - 1/\lambda^{\xi}$ when $M = \Omega((\lambda \log \lambda)^{2/3})$. Combining this instantiation with the one in Section 3.2 proves Lemma 5.

Choice of Γ. We classify a value $v \in \mathbf{adom}$ as a *low-degree* value if its degree is less than $\lambda/\sqrt{M}$, or a *high-degree* value otherwise (review Section 2.1 for the notion "degree"). Let $\mathbf{adom}_{lo}$ (resp. $\mathbf{adom}_{hi}$) be the set of low- (resp. high-) degree values. We can obtain the degrees of all values in $\mathbf{adom}$ – hence, $\mathbf{adom}_{lo}$ and $\mathbf{adom}_{hi}$ – in $O(\text{sort}(N))$ I/Os.

Different strategies are deployed to map $\mathbf{adom}_{lo}$ and $\mathbf{adom}_{hi}$ to $[s]$. For $\mathbf{adom}_{lo}$, we independently map each value therein to a color chosen from $[s]$ uniformly at random. The strategy for $\mathbf{adom}_{hi}$ is, however, deterministic. By scanning $\mathbf{adom}_{hi}$ once in $O(N/B)$ I/Os, we can break $\mathbf{adom}_{hi}$ into at most N/λ disjoint *groups* such that, for each group, the total degree of all the values therein is at most 5λ.[2] Note that, as a value in $\mathbf{adom}_{hi}$ has degree at least $\lambda/\sqrt{M}$, each group contains at most $\frac{5\lambda}{\lambda/\sqrt{M}} = 5\sqrt{M}$ values. Moreover, since $\lambda \geq \sqrt{NM}$, there can be no more than $N/\lambda \leq \lambda/M \leq s$ groups. We treat each group as a distinct color, and define function $\Gamma_2 : \mathbf{adom}_{hi} \to [s]$ that maps a value $v \in \mathbf{adom}_{hi}$ to color $c \in [s]$ if v appears in the c-th group. Functions Γ_1 and Γ_2 together define the coloring function Γ in (5). The coloring step (defined in Section 3.1) can then be performed with sorting in $O(\text{sort}(N))$ I/Os.

Analysis. Next, we analyze the I/O cost of our algorithm in Section 3.1, given the above choice of Γ. Our objective is to prove that (8) is bounded by $O(\lambda^k/(M^{k-1}B))$. The lemma below establishes a crucial fact towards that purpose.

▶ **Lemma 8.** *When $M = \Omega((\lambda \log \lambda)^{2/3})$, $|R_{c_1,c_2}| = O(M)$ holds with probability at least $1 - 1/\lambda^{\xi'}$ for any relation $R \in \mathcal{Q}$ and any colors $c_1, c_2 \in [s]$, where ξ' can be an arbitrarily large constant.*

Proof. Let X_1, X_2 be the attributes in $schema(R)$ such that $X_1 < X_2$ (recall from Section 3.1 that we have imposed an arbitrary total order on attributes). Divide R_{c_1,c_2} into four subsets:

- $R^{lo,lo}_{c_1,c_2}$, the set of tuples $\boldsymbol{t} \in R_{c_1,c_2}$ such that $\boldsymbol{t}(X_1)$ and $\boldsymbol{t}(X_2)$ are both in $\mathbf{adom}_{lo}$;
- $R^{lo,hi}_{c_1,c_2}$, the set of tuples $\boldsymbol{t} \in R_{c_1,c_2}$ such that $\boldsymbol{t}(X_1) \in \mathbf{adom}_{lo}$ but $\boldsymbol{t}(X_2) \in \mathbf{adom}_{hi}$;
- $R^{hi,lo}_{c_1,c_2}$, the set of tuples $\boldsymbol{t} \in R_{c_1,c_2}$ such that $\boldsymbol{t}(X_1) \in \mathbf{adom}_{hi}$ but $\boldsymbol{t}(X_2) \in \mathbf{adom}_{lo}$;
- $R^{hi,hi}_{c_1,c_2}$, the set of tuples $\boldsymbol{t} \in R_{c_1,c_2}$ such that $\boldsymbol{t}(X_1)$ and $\boldsymbol{t}(X_2)$ are both in $\mathbf{adom}_{hi}$.

We will show that each subset has size $O(M)$ with probability at least $1 - 1/(4\lambda^{\xi'})$, which is sufficient for proving the lemma.

The case of $R^{hi,hi}_{c_1,c_2}$ is the easiest. Every tuple $\boldsymbol{t} \in R^{hi,hi}_{c_1,c_2}$ must set $\boldsymbol{t}(X_1)$ to a high-degree value from color (a.k.a. group) c_1 and $\boldsymbol{t}(X_2)$ to a high-degree value from color (a.k.a. group) c_2. As mentioned, every group has at most $5\sqrt{M}$ values. Hence, $|R^{hi,hi}_{c_1,c_2}| \leq 25M$.

[2] Add the next high-degree value v to the current group as long as doing so will not push the group's total degree over 3λ. Otherwise, start a new group with only v; the preceding group must have a total weight as least 2λ because the degree of v is bounded by λ. If the last group has a total degree less than 2λ, combine it with the previous group (if it exists), which will yield a group with total weight at most 5λ. This way, we guarantee that either only a single group exists, or every group has a total weight at least 2λ. As each tuple can contribute one to the degrees of at most two values in $\mathbf{adom}_{hi}$, the total weights of all the groups add up to at most $2N$. The number of groups is therefore at most $2N/(2\lambda) = N/\lambda$.

To analyze $R^{lo,lo}_{c_1,c_2}$, define $R^{lo,lo}$ to be the set of tuples $\boldsymbol{t} \in R$ such that $\boldsymbol{t}(X_1)$ and $\boldsymbol{t}(X_2)$ are both low-degree values. For each tuple $\boldsymbol{t} \in R^{lo,lo}$, introduce a random variable $Z_{\boldsymbol{t}}$ that equals 1 if $\boldsymbol{t} \in R^{lo,lo}_{c_1,c_2}$, or 0 otherwise. Our function Γ_1 ensures $\Pr[Z_{\boldsymbol{t}} = 1] = 1/s^2$ with variance $\mathbf{Var}(Z_{\boldsymbol{t}}) = \frac{1}{s^2} - \frac{1}{s^4}$. Define $Z := |R^{lo,lo}_{c_1,c_2}| = \sum_{\boldsymbol{t} \in R^{lo,lo}} Z_{\boldsymbol{t}}$. We will deploy Lemma 4 to analyze how likely Z can deviate significantly from $\mathbf{E}[Z]$. For this purpose, create a dependency graph $G^{lo,lo}$ as follows. Each vertex of $G^{lo,lo}$ is the variable $Z_{\boldsymbol{t}}$ of a distinct tuple $\boldsymbol{t} \in R^{lo,lo}$. Two vertices $Z_{\boldsymbol{t_1}}$ and $Z_{\boldsymbol{t_2}}$ are adjacent in $G^{lo,lo}$ if and only if tuples $\boldsymbol{t_1}$ and $\boldsymbol{t_2}$ share the same value on attribute X_1 or X_2. It is easy to verify that $G^{lo,lo}$ fulfills the independence requirement described in Section 2.3 and has a maximum vertex degree at most $2\lambda/\sqrt{M}$ by definition of low-degree value. Now, apply Lemma 4 with $\mu := \mathbf{E}[Z] = |R^{lo,lo}|/s^2$, $\sigma := |R^{lo,lo}|/s^2 > \sum_{\boldsymbol{t} \in R^{lo,lo}} \mathbf{Var}(Z_{\boldsymbol{t}})$, $\Delta := 2\lambda/\sqrt{M}$, and $\epsilon := Ms^2/|R^{lo,lo}|$. The application yields $\Pr[Z \geq 2M] \leq \exp(-\Theta(1) \cdot \frac{M^{1.5}}{\lambda})$, which is at most $1/(4\lambda^{\xi'})$ as long as $M = \Omega((\lambda \log \lambda)^{2/3})$.

The analysis of $R^{hi,lo}_{c_1,c_2}$ and $R^{lo,hi}_{c_1,c_2}$ is similar and deferred to Appendix C. ◀

We now return to our algorithm's I/O cost in (8). As mentioned before, s^k is bounded by $O(\lambda^k/(M^{k-1}B))$. By the above lemma, with probability at least $1 - 1/\lambda^{\xi}$ for an arbitrarily large constant ξ, $\frac{s^{k-2}}{M^{r-1}B} \sum_{R \in \mathcal{Q}} \sum_{c_1,c_2 \in [s]} |R_{c_1,c_2}|^r$ is bounded by $O(\frac{s^{k-2}}{M^{r-1}B} \cdot s^2 M^r) = O(\lambda^k/(M^{k-1}B))$, applying the value of s in (4). We thus complete the proof of Lemma 5.

4 An EM Algorithm for Arbitrary Binary Joins

This section serves as a proof of:

▶ **Theorem 9.** *Consider a binary join $\mathcal{Q}$ whose relations have distinct schemas. Let $\mathcal{G} := (\mathcal{X}, \mathcal{E})$ be the schema graph of $\mathcal{Q}$, $k := |\mathcal{X}|$, and $N := \sum_{R \in \mathcal{Q}} |R|$. There is an algorithm in EM that, with high probability, emits every tuple of join($\mathcal{Q}$) exactly once in $O(\frac{N^{k/2}}{M^{k/2-1}B} \log_{M/B} \frac{N}{B} + \frac{N^{\rho}}{M^{\rho-1}B})$ I/Os, where ρ is the fractional edge covering number of $\mathcal{G}$, M is the number of words in memory, and B is the number of words in a disk block.*

Theorem 1 follows from the above result and Lemma 3. Our solution can be regarded as an efficient EM translation of an MPC algorithm in [23]. The non-trivial part is to show that the I/O cost is as claimed. We will achieve the purpose by utilizing a mathematical property of binary joins recently revealed by the *isolated cartesian product theorem* [23].

We consider $|\mathcal{X}| \geq 3$; otherwise, $\mathcal{Q}$ has only one relation and the tuples of *join*($\mathcal{Q}$) can be emitted in $O(N/B)$ I/Os. For each hyperedge $e \in \mathcal{E}$, we will use R_e to denote the (only) relation in $\mathcal{Q}$ with schema e. As before, let **adom** be the combined active domain of $\mathcal{Q}$. Henceforth, we will fix

$$\lambda := \sqrt{NM}. \tag{11}$$

4.1 Residual Joins

We say that a value $v \in$ **adom** is *heavy* if its degree is at least λ, or *light* otherwise. The number of heavy values is $O(N/\lambda)$. Let $\mathcal{H}$ be any subset of $\mathcal{X} :=$ *schema*($\mathcal{Q}$). A *configuration* of $\mathcal{H}$ is defined as a tuple $\boldsymbol{\eta}$ over $\mathcal{H}$ whose $\boldsymbol{\eta}(X)$ is heavy for every attribute $X \in \mathcal{H}$. Let *config*($\mathcal{H}$) be the set of all configurations $\boldsymbol{\eta}$ of $\mathcal{H}$ satisfying

$$\boldsymbol{\eta}[e] \in R_e \text{ for every } e \in \mathcal{E} \text{ such that } e \subseteq \mathcal{H}. \tag{12}$$

It is clear that

$$|config(\mathcal{H})| = O((N/\lambda)^{|\mathcal{H}|}) = O((N/M)^{|\mathcal{H}|/2}) \tag{13}$$

Fix any configuration $\boldsymbol{\eta} \in config(\mathcal{H})$. For each hyperedge $e \in \mathcal{E}$ satisfying $e \setminus \mathcal{H} \neq \emptyset$, we define relation $R_e(\boldsymbol{\eta})$ to be a subset of R_e that includes every tuple $\boldsymbol{t} \in R_e$ satisfying (i) $\boldsymbol{t}(X) = \boldsymbol{\eta}(X)$ for all $X \in e \cap \mathcal{H}$; (ii) $\boldsymbol{t}(X)$ is light for every $X \in e \setminus \mathcal{H}$. Note that if $e \cap \mathcal{H} = \emptyset$, then $R_e(\boldsymbol{\eta}) = R_e$. Every such hyperedge e has a *residual relation* $R'_e(\boldsymbol{\eta})$ defined as

$$R'_e(\boldsymbol{\eta}) := \Pi_{e \setminus \mathcal{H}}(R_e(\boldsymbol{\eta})). \tag{14}$$

The configuration $\boldsymbol{\eta}$ induces a *residual join* $\mathcal{Q}'(\boldsymbol{\eta})$ formalized as

$$\mathcal{Q}'(\boldsymbol{\eta}) := \{R'_e(\boldsymbol{\eta}) \mid e \in \mathcal{E}, e \setminus \mathcal{H} \neq \emptyset\} \tag{15}$$

whose input size is

$$N_{\boldsymbol{\eta}} := \sum_{R \in \mathcal{Q}'(\boldsymbol{\eta})} |R|. \tag{16}$$

▶ **Example 10.** Figure 1(a) shows the schema graph $\mathcal{G} := (\mathcal{X}, \mathcal{E})$ of a join $\mathcal{Q}$, where $\mathcal{X} := \{\texttt{A}, \texttt{B}, ..., \texttt{L}\}$ and the hyperedges in $\mathcal{E}$ are represented as ellipses. Set $\mathcal{H} := \{\texttt{E}, \texttt{F}, \texttt{I}\}$ and consider the configuration $\boldsymbol{\eta}$ with heavy values $\boldsymbol{\eta}(\texttt{E}) := \texttt{e}$, $\boldsymbol{\eta}(\texttt{F}) := \texttt{f}$, and $\boldsymbol{\eta}(\texttt{I}) := \texttt{i}$. The figure illustrates $\boldsymbol{\eta}$ by darkening vertices E, F, and I. Suppose $\boldsymbol{\eta} \in config(\mathcal{H})$, which means that $\boldsymbol{\eta}[\texttt{EF}]$ is a tuple in $R_{\texttt{EF}}$, and $\boldsymbol{\eta}[\texttt{EI}]$ is a tuple in $R_{\texttt{EI}}$. Relation $R_{\texttt{DE}}(\boldsymbol{\eta})$ includes all such tuples $\boldsymbol{t} \in R_{\texttt{DE}}$ that use value e for $\boldsymbol{t}(\texttt{E})$ and a light value for $\boldsymbol{t}(\texttt{D})$. The residual relation $R'_{\texttt{DE}}(\boldsymbol{\eta})$ is a unary relation that is the projection of $R_{\texttt{DE}}(\boldsymbol{\eta})$ on D. The reader can verify that the residual relations $R'_{\texttt{AD}}(\boldsymbol{\eta})$, $R'_{\texttt{DG}}(\boldsymbol{\eta})$, and $R'_{\texttt{DH}}(\boldsymbol{\eta})$ are identical to $R_{\texttt{AD}}$, $R_{\texttt{DG}}$, and $R_{\texttt{DH}}$, respectively. Edges EI and EF define no residual relations. ⌟

The lemma below, proved in Appendix D, will be useful in our analysis later.

▶ **Lemma 11.** *The statements below are true for every $\mathcal{H} \subseteq \mathcal{X}$:*

- $\sum_{\boldsymbol{\eta} \in config(\mathcal{H})} N_{\boldsymbol{\eta}} = O(N^{k/2}/M^{k/2-1})$;
- *in $O(\frac{N^{k/2}}{M^{k/2-1}B} \log_{M/B} \frac{N}{B})$ I/Os, we can ensure the following for all $\boldsymbol{\eta} \in config(\mathcal{H})$: each relation of $\mathcal{Q}'(\boldsymbol{\eta})$ is stored in consecutive disk blocks.*

It is easy to verify that

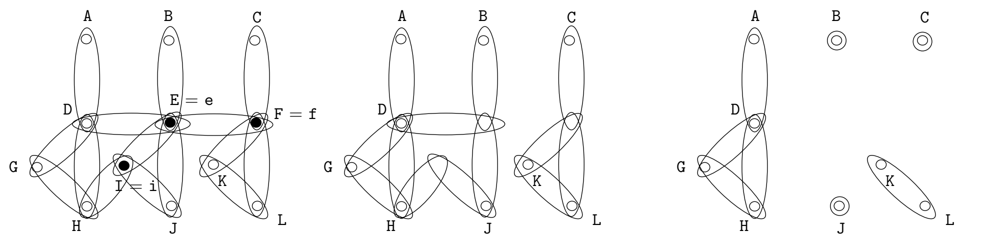
$$join(\mathcal{Q}) = \bigcup_{\mathcal{H}} \Big(\bigcup_{\boldsymbol{\eta} \in config(\mathcal{H})} join(\mathcal{Q}'(\boldsymbol{\eta})) \times \{\boldsymbol{\eta}\} \Big). \tag{17}$$

Next, we will present an algorithm that, given any $\mathcal{H} \subseteq \mathcal{X}$, emits each tuple of $\bigcup_{\boldsymbol{\eta} \in config(\mathcal{H})} join(\mathcal{Q}'(\boldsymbol{\eta})) \times \{\boldsymbol{\eta}\}$ exactly once. Executing the algorithm for all the $2^{|\mathcal{X}|} = O(1)$ subsets $\mathcal{H} \subseteq \mathcal{X}$ emits the entire $join(\mathcal{Q})$, with no tuple emitted twice.

4.2 Simplifying Residual Joins

Fix an arbitrary $\mathcal{H} \subseteq \mathcal{X}$ and define $\mathcal{L} := \mathcal{X} \setminus \mathcal{H}$. We will call the attributes in $\mathcal{H}$ and $\mathcal{L}$ as *heavy* and *light* attributes, respectively. An attribute $X \in \mathcal{X}$ is a *border* attribute if it is adjacent to at least one heavy attribute.

By removing all the heavy attributes from $\mathcal{G} := (\mathcal{X}, \mathcal{E})$, we obtain a *residual graph* $\mathcal{G}' := (\mathcal{X}', \mathcal{E}')$ where $\mathcal{X}' := \mathcal{X} \setminus \mathcal{H}$ and $\mathcal{E}' := \{e \setminus \mathcal{H} \mid e \in \mathcal{E} \text{ and } e \setminus \mathcal{H} \neq \emptyset\}$. An attribute $X \in \mathcal{X}'$ is *isolated* if it is adjacent to no other attributes in $\mathcal{G}'$. Denote by $\mathcal{I}$ the set of isolated attributes. An isolated attribute is always a border attribute, but the reverse is not true.



(a) Schema graph $\mathcal{G}$ of $\mathcal{Q}$. (b) Residual graph $\mathcal{G}'$. (c) Simplified residual graph. $\mathcal{G}''$

Figure 1 A running example.

▶ **Example 12.** Figure 1(b) shows the residual graph $\mathcal{G}' := (\mathcal{X}', \mathcal{E}')$ of the schema graph in Figure 1(a), after removing E, F, and I, as well as the hyperedges that have become empty. The border attributes are B, C, D, H, J, K, and L, while the set of isolated attributes is $\mathcal{I} = \{\mathtt{B}, \mathtt{C}, \mathtt{J}\}$. ⌟

Fix any configuration $\boldsymbol{\eta} \in \mathit{config}(\mathcal{H})$. $\mathcal{G}'$ is the schema graph of $\mathcal{Q}'(\boldsymbol{\eta})$ (true for all $\boldsymbol{\eta}$). For each border attribute $X \in \mathcal{X}'$, $\mathcal{G}'$ has at least one relation with schema $\{X\}$. Define:

$$R''_X(\boldsymbol{\eta}) := \bigcap_{e \in \mathcal{E}' : X \in e} \Pi_X(R'_e(\boldsymbol{\eta})) \tag{18}$$

where $R'_e(\boldsymbol{\eta})$ is defined in (14). Only the values in $R''_X(\boldsymbol{\eta})$ can contribute to $\mathit{join}(\mathcal{Q}'(\boldsymbol{\eta}))$.

▶ **Example 13.** Continuing on Example 12, consider again the residual graph $\mathcal{G}' := (\mathcal{X}', \mathcal{E}')$ in Figure 1(b). Set X to the border attribute D. The residual join $\mathcal{Q}'(\boldsymbol{\eta})$ has four relations whose schemas contain D: unary relation $R'_{\mathtt{DE}}(\boldsymbol{\eta})$ and binary relations $R'_{\mathtt{AD}}(\boldsymbol{\eta})$, $R'_{\mathtt{DG}}(\boldsymbol{\eta})$, and $R'_{\mathtt{DH}}(\boldsymbol{\eta})$, all of which were explained in Example 10. $R''_{\mathtt{D}}(\boldsymbol{\eta})$ equals the intersection of $\Pi_{\mathtt{D}}(R'_{\mathtt{DE}}(\boldsymbol{\eta})) = R'_{\mathtt{DE}}(\boldsymbol{\eta})$, $\Pi_{\mathtt{D}}(R'_{\mathtt{AD}}(\boldsymbol{\eta}))$, $\Pi_{\mathtt{D}}(R'_{\mathtt{DG}}(\boldsymbol{\eta}))$, and $\Pi_{\mathtt{D}}(R'_{\mathtt{DH}}(\boldsymbol{\eta}))$. As another example, set X to the isolated attribute J, which appears in two hyperedges in $\mathcal{G}'$, both unary. $R''_{\mathtt{J}}(\boldsymbol{\eta})$ is the intersection of $\Pi_{\mathtt{J}}(R'_{\mathtt{IJ}}(\boldsymbol{\eta})) = R'_{\mathtt{IJ}}(\boldsymbol{\eta})$ and $\Pi_{\mathtt{J}}(R'_{\mathtt{EJ}}(\boldsymbol{\eta})) = R'_{\mathtt{EJ}}(\boldsymbol{\eta})$. ⌟

For every binary $e := \{X_1, X_2\} \in \mathcal{E}'$, we define a relation $R''_e(\boldsymbol{\eta}) \subseteq R'_e(\boldsymbol{\eta})$ as follows:

- If X_1 and X_2 are both border attributes, then $R''_e(\boldsymbol{\eta}) := R'_e(\boldsymbol{\eta}) \ltimes R''_{X_1}(\boldsymbol{\eta}) \ltimes R''_{X_2}(\boldsymbol{\eta})$;
- If only X_1 is a border attribute, then $R''_e(\boldsymbol{\eta}) := R'_e(\boldsymbol{\eta}) \ltimes R''_{X_1}(\boldsymbol{\eta})$;
- If only X_2 is a border attribute, then $R''_e(\boldsymbol{\eta}) := R'_e(\boldsymbol{\eta}) \ltimes R''_{X_2}(\boldsymbol{\eta})$;
- If neither is a border attribute, then $R''_e(\boldsymbol{\eta}) := R'_e(\boldsymbol{\eta})$.

Only the tuples in $R''_e(\boldsymbol{\eta})$ can contribute to $\mathit{join}(\mathcal{Q}'(\boldsymbol{\eta}))$.

▶ **Example 14.** Continuing on Example 13, if we set the hyperedge e to DH in $\mathcal{E}'$, then $R''_e(\boldsymbol{\eta})$ contains only the tuples $\boldsymbol{t} \in R'_{\mathtt{DH}}(\boldsymbol{\eta})$ with $\boldsymbol{t}(\mathtt{D}) \in R''_{\mathtt{D}}(\boldsymbol{\eta})$ and $\boldsymbol{t}(\mathtt{H}) \in R''_{\mathtt{H}}(\boldsymbol{\eta})$. For another example, if $e :=$ GH, then $R''_e(\boldsymbol{\eta})$ contains only the tuples $\boldsymbol{t} \in R'_{\mathtt{GH}}(\boldsymbol{\eta})$ with $\boldsymbol{t}(\mathtt{H}) \in R''_{\mathtt{H}}(\boldsymbol{\eta})$. ⌟

We can now define a *simplified residual join* induced by $\boldsymbol{\eta}$:

$$\mathcal{Q}''(\boldsymbol{\eta}) \quad := \quad \{R''_e(\boldsymbol{\eta}) \mid \text{binary } e \in \mathcal{E}'\} \cup \{R''_X(\boldsymbol{\eta}) \mid X \in \mathcal{I}\}. \tag{19}$$

Define $\mathcal{G}'' := (\mathcal{X}'', \mathcal{E}'')$ – the *simplified residual graph* – as the hypergraph where $\mathcal{X}'' := \mathcal{X}'$, and $\mathcal{E}''$ includes (i) all the binary edges in $\mathcal{E}'$ and (ii) a unary edge $\{X\}$ for every isolated attribute $X \in \mathcal{I}$. $\mathcal{G}''$ is the schema graph of $\mathcal{Q}''(\boldsymbol{\eta})$ for all $\boldsymbol{\eta} \in \mathit{config}(\mathcal{H})$.

▶ **Example 15.** Continuing on Ex.14, Figure 1(c) shows the simplified residual graph $\mathcal{G}''$. ⌟

It is rudimentary to verify several facts about the join $\mathcal{Q}''(\boldsymbol{\eta})$ in (19). First, its input size is at most that of $\mathcal{Q}'(\boldsymbol{\eta})$, which is $N_{\boldsymbol{\eta}}$; see (16). Second, its relations have distinct schemas. Third, as each relation of $\mathcal{Q}'(\boldsymbol{\eta})$ has been stored in consecutive disk blocks (Lemma 11), we can achieve the same for $\mathcal{Q}''(\boldsymbol{\eta})$ in $O(\text{sort}(N_{\boldsymbol{\eta}}))$ I/Os; doing so for all $\boldsymbol{\eta} \in \mathit{config}(\mathcal{H})$ requires

$$\sum_{\boldsymbol{\eta}\in \mathit{config}(\mathcal{H})} O(\text{sort}(N_{\boldsymbol{\eta}})) = O\Big(|\mathit{config}(\mathcal{H})| + \sum_{\boldsymbol{\eta}\in \mathit{config}(\mathcal{H})} \frac{N_{\boldsymbol{\eta}}}{B}\log_{\frac{M}{B}}\frac{N}{B}\Big) = O\Big(\frac{N^{k/2}}{M^{k/2-1}B}\log_{\frac{M}{B}}\frac{N}{B}\Big)$$

I/Os, where the last equality used (13) and the first bullet of Lemma 11. Fourth, $\mathit{join}(\mathcal{Q}''(\boldsymbol{\eta})) = \mathit{join}(\mathcal{Q}'(\boldsymbol{\eta}))$; hence, to process the original join $\mathcal{Q}$ in Theorem 9, it suffices to emit every result tuple of $\mathit{join}(\mathcal{Q}''(\boldsymbol{\eta}))$ exactly once, for all $\boldsymbol{\eta} \in \mathit{config}(\mathcal{H})$ and $\mathcal{H} \subseteq \mathcal{X}$.

4.3 Processing Simplified Residual Joins

Fix an arbitrary $\mathcal{H} \subseteq \mathcal{X}$, and define $\mathcal{L}$, $\mathcal{I}$, and $\mathcal{G}'' := (\mathcal{X}'', \mathcal{E}'')$ as in the previous subsection. Given an arbitrary $\boldsymbol{\eta} \in \mathit{config}(\mathcal{H})$, we will present an algorithm for processing the simplified residual join $\mathcal{Q}''(\boldsymbol{\eta})$. First, let us divide $\mathcal{Q}''(\boldsymbol{\eta})$ into $\mathcal{Q}''_{bin}(\boldsymbol{\eta}) := \{R''_e(\boldsymbol{\eta}) \mid \text{binary } e \in \mathcal{E}''\}$ and $\mathcal{Q}''_{iso}(\boldsymbol{\eta}) := \{R''_X(\boldsymbol{\eta}) \mid X \in \mathcal{I}\}$. It is clear that $\mathit{join}(\mathcal{Q}''(\boldsymbol{\eta})) = \mathit{join}(\mathcal{Q}''_{bin}(\boldsymbol{\eta})) \times \mathit{join}(\mathcal{Q}''_{iso}(\boldsymbol{\eta}))$.

▶ **Example 16.** Continuing on Example 15, $\mathcal{Q}''_{bin}(\boldsymbol{\eta})$ includes relations $R''_{\mathtt{AD}}(\boldsymbol{\eta})$, $R''_{\mathtt{DG}}(\boldsymbol{\eta})$, $R''_{\mathtt{DH}}(\boldsymbol{\eta})$, $R''_{\mathtt{GH}}(\boldsymbol{\eta})$, and $R''_{\mathtt{KL}}(\boldsymbol{\eta})$, while $\mathcal{Q}''_{iso}(\boldsymbol{\eta})$ includes $R''_{\mathtt{B}}(\boldsymbol{\eta})$, $R''_{\mathtt{C}}(\boldsymbol{\eta})$, and $R''_{\mathtt{J}}(\boldsymbol{\eta})$. ⌟

Observe that $\mathcal{Q}''_{bin}(\boldsymbol{\eta})$ is a binary join whose scheme graph has $|\mathcal{L} \setminus \mathcal{I}|$ attributes; furthermore, the relations of $\mathcal{Q}''_{bin}(\boldsymbol{\eta})$ contain only light values, implying that $\mathcal{Q}''_{bin}(\boldsymbol{\eta})$ has a degree at most λ. On the other hand, $\mathcal{Q}''_{iso}(\boldsymbol{\eta})$ is merely the cartesian product of all the (unary) relations therein. We process $\mathcal{Q}''(\boldsymbol{\eta})$ by integrating BNL with the algorithm in Lemma 5. Specifically, we chop each relation $R''_X(\boldsymbol{\eta}) \in \mathcal{Q}''_{iso}(\boldsymbol{\eta})$ into $O(\lceil |R''_X(\boldsymbol{\eta})|/M \rceil)$ disjoint subsets – called *chunks* – each of which fits in $M/(k+1)$ words. Define a *chunk combination* as a collection of $|\mathcal{Q}''_{iso}|$ chunks, each from a distinct relation in $\mathcal{Q}''_{iso}$. For every chunk combination, load the $|\mathcal{Q}''_{iso}|$ corresponding chunks in memory and then use the remaining at least $M/(k+1) = \Omega(M)$ words of memory to run the algorithm in Lemma 5. Every time the algorithm emits a tuple $\boldsymbol{t} \in \mathit{join}(\mathcal{Q}''_{bin}(\boldsymbol{\eta}))$, we emit all the tuples of $\mathit{join}(\mathcal{Q}''(\boldsymbol{\eta}))$ that can be produced by $\boldsymbol{t}$ and the memory-resident chunk data (this requires only CPU computation). As there are at most $O(\prod_{X\in\mathcal{I}} \lceil \frac{|R''_X(\boldsymbol{\eta})|}{M} \rceil)$ chunk combinations, the total I/O cost spent on $\mathcal{Q}''(\boldsymbol{\eta})$ is $O(\prod_{X\in\mathcal{I}} \lceil \frac{|R''_X(\boldsymbol{\eta})|}{M} \rceil \cdot \frac{\lambda^{|\mathcal{L}\setminus\mathcal{I}|}}{M^{|\mathcal{L}\setminus\mathcal{I}|-1}B})$ with probability at least $1 - 1/\lambda^{\xi'}$ for an arbitrarily large constant ξ'.

Processing all $\boldsymbol{\eta} \in \mathit{config}(\mathcal{H})$ in the above manner incurs a total I/O cost of

$$O\Big(\frac{\lambda^{|\mathcal{L}\setminus\mathcal{I}|}}{M^{|\mathcal{L}\setminus\mathcal{I}|-1}B} \sum_{\boldsymbol{\eta}\in \mathit{config}(\mathcal{H})} \prod_{X\in\mathcal{I}} \Big\lceil \frac{|R''_X(\boldsymbol{\eta})|}{M} \Big\rceil\Big) = O\Big(\Big(\frac{N}{M}\Big)^{\frac{|\mathcal{L}\setminus\mathcal{I}|}{2}} \frac{M}{B} \sum_{\boldsymbol{\eta}\in \mathit{config}(\mathcal{H})} \prod_{X\in\mathcal{I}} \Big(\frac{|R''_X(\boldsymbol{\eta})|}{M} + 1\Big)\Big) \tag{20}$$

where the derivation applied the definition of λ in (11).

▶ **Lemma 17.** $\sum_{\boldsymbol{\eta}\in \mathit{config}(\mathcal{H})} \prod_{X\in\mathcal{I}} \Big(\frac{|R''_X(\boldsymbol{\eta})|}{M} + 1\Big) = O((\frac{N}{M})^{\rho - \frac{|\mathcal{L}\setminus\mathcal{I}|}{2}})$, *where ρ is the fractional edge covering number of the join $\mathcal{Q}$ stated in Theorem 9.*

Proof. For each $\boldsymbol{\eta} \in \mathit{config}(\mathcal{H})$ and any non-empty $\mathcal{J} \subseteq \mathcal{I}$, define $\mathit{CPsize}_{\mathcal{J}}(\boldsymbol{\eta}) := \prod_{X\in\mathcal{J}} |R''_X(\boldsymbol{\eta})|$. Crucially, observe that

$$\prod_{X\in\mathcal{I}} \Big(\frac{|R''_X(\boldsymbol{\eta})|}{M} + 1\Big) = 1 + \sum_{\mathcal{J}\subseteq\mathcal{I}:\mathcal{J}\neq\emptyset} \prod_{X\in\mathcal{J}} \frac{|R''_X(\boldsymbol{\eta})|}{M} = 1 + \sum_{\mathcal{J}\subseteq\mathcal{I}:\mathcal{J}\neq\emptyset} \frac{\mathit{CPsize}_{\mathcal{J}}(\boldsymbol{\eta})}{M^{|\mathcal{J}|}}.$$

Next, we will show that $\sum_{\boldsymbol{\eta} \in \mathit{config}(\mathcal{H})} 1$ and the term $\sum_{\boldsymbol{\eta} \in \mathit{config}(\mathcal{H})} \frac{\mathit{CPsize}_{\mathcal{J}}(\boldsymbol{\eta})}{M^{|\mathcal{J}|}}$ of each non-empty $\mathcal{J} \subseteq \mathcal{I}$ are all bounded by $O((N/M)^{\rho - \frac{|\mathcal{L} \setminus \mathcal{I}|}{2}})$, which will then establish Lemma 17 because there are $2^{|\mathcal{I}|} - 1 = O(1)$ choices for $\mathcal{J}$.

From (11) and (13), we know $\sum_{\boldsymbol{\eta} \in \mathit{config}(\mathcal{H})} 1 = |\mathit{config}(\mathcal{H})| = O((N/M)^{|\mathcal{H}|/2})$. As a well-known fact [36], the value of ρ is at least $|\mathcal{X}|/2$, where $|\mathcal{X}|$ is the number of attributes in the schema graph of $\mathcal{Q}$. Because $|\mathcal{X}| = |\mathcal{H}| + |\mathcal{L}| \geq |\mathcal{H}| + |\mathcal{L} \setminus \mathcal{I}|$, we know $|\mathcal{H}|/2 \leq (|\mathcal{X}| - |\mathcal{L} \setminus \mathcal{I}|)/2 \leq \rho - \frac{|\mathcal{L} \setminus \mathcal{I}|}{2}$.

It remains to analyze the term $\sum_{\boldsymbol{\eta} \in \mathit{config}(\mathcal{H})} \frac{\mathit{CPsize}_{\mathcal{J}}(\boldsymbol{\eta})}{M^{|\mathcal{J}|}}$ for each non-empty $\mathcal{J} \subseteq \mathcal{I}$. We apply Lemma 11 of [23] – a weaker version of the isolated cartesian product theorem in [23] – which states $\sum_{\boldsymbol{\eta} \in \mathit{config}(\mathcal{H})} \mathit{CPsize}_{\mathcal{J}}(\boldsymbol{\eta}) = O((N/M)^{\rho - (|\mathcal{J}| + |\mathcal{L}|)/2} \cdot N^{|\mathcal{J}|})$. This yields

$$\sum_{\boldsymbol{\eta} \in \mathit{config}(\mathcal{H})} \frac{\mathit{CPsize}_{\mathcal{J}}(\boldsymbol{\eta})}{M^{|\mathcal{J}|}} = O\Big(\frac{(\frac{N}{M})^{\rho - \frac{|\mathcal{J}| + |\mathcal{L}|}{2}} \cdot N^{|\mathcal{J}|}}{M^{|\mathcal{J}|}}\Big) = O\Big(\Big(\frac{N}{M}\Big)^{\rho - \frac{|\mathcal{L} \setminus \mathcal{J}|}{2}}\Big) = O\Big(\Big(\frac{N}{M}\Big)^{\rho - \frac{|\mathcal{L} \setminus \mathcal{I}|}{2}}\Big)$$

where the derivation used the fact $\mathcal{J} \subseteq \mathcal{I} \subseteq \mathcal{L}$. ◀

Plugging the result of Lemma 17 into (20), we know that the simplified residual joins induced by all the $\boldsymbol{\eta} \in \mathit{config}(\mathcal{H})$ can be processed using $O(N^\rho/(M^{\rho-1}B))$ I/Os in total with probability at least $1 - |\mathit{config}(\mathcal{H})|/\lambda^{\xi'}$, namely, w.h.p. if we set ξ' sufficiently large. Repeating the algorithm for all the $O(1)$ subsets $\mathcal{H} \subseteq \mathcal{X}$ settles the the original join $\mathcal{Q}$ in $O(N^\rho/(M^{\rho-1}B))$ w.h.p.. We thus complete the proof of Theorem 9.

5 Conclusions

This paper has presented new progress in designing I/O-efficient algorithms for subgraph enumeration, where the objective is to find all the occurrences of a pattern graph Q having $k = O(1)$ vertices in a data graph $G := (V, E)$. Our algorithm guarantees an I/O complexity $O(\frac{|E|^{k/2}}{M^{k/2-1}B} \log_{M/B} \frac{|E|}{B} + \frac{|E|^\rho}{M^{\rho-1}B})$ with high probability, where $\rho \geq k/2$ is the fractional edge covering number of Q, M is the number of words in memory, and B is the number of words in a disk block. The algorithm matches an existing I/O lower bound of $\Omega(\frac{|E|^\rho}{M^{\rho-1}B})$ on the class of indivisible algorithms whenever $\rho > k/2$ or $M/B \geq (|E|/B)^\epsilon$ for any constant $\epsilon > 0$. The main open problem left behind by our work is to eliminate the $\log_{M/B}(|E|/B)$ factor altogether, thus obtaining an algorithm that matches the lower bound in all cases.

References

1 Alok Aggarwal and Jeffrey Scott Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM (CACM)*, 31(9):1116–1127, 1988.

2 Noga Alon, Raphael Yuster, and Uri Zwick. Color-coding. *Journal of the ACM (JACM)*, 42(4):844–856, 1995.

3 Noga Alon, Raphael Yuster, and Uri Zwick. Finding and counting given length cycles. *Algorithmica*, 17(3):209–223, 1997.

4 Kaleb Alway, Eric Blais, and Semih Salihoglu. Box covers and domain orderings for beyond worst-case join processing. In *Proceedings of International Conference on Database Theory (ICDT)*, pages 3:1–3:23, 2021.

5 Suman K. Bera, Noujan Pashanasangi, and C. Seshadhri. Near-linear time homomorphism counting in bounded degeneracy graphs: The barrier of long induced cycles. In *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2315–2332, 2021.

6 Andreas Bjorklund, Petteri Kaski, and Lukasz Kowalik. Counting thin subgraphs via packings faster than meet-in-the-middle time. *ACM Transactions on Algorithms*, 13(4):48:1–48:26, 2017.

7 Andreas Bjorklund, Rasmus Pagh, Virginia Vassilevska Williams, and Uri Zwick. Listing triangles. In *Proceedings of International Colloquium on Automata, Languages and Programming (ICALP)*, pages 223–234, 2014.

8 N. Chiba and T. Nishizeki. Arboricity and subgraph listing algorithms. *SIAM Journal of Computing*, 14(1):210–223, 1985.

9 Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of ACM Symposium on Theory of Computing (STOC)*, pages 151–158, 1971.

10 Radu Curticapean, Holger Dell, and Dániel Marx. Homomorphisms are a good basis for counting small subgraphs. In *Proceedings of ACM Symposium on Theory of Computing (STOC)*, pages 210–223, 2017.

11 David Eppstein. Arboricity and bipartite subgraph listing algorithms. *Information Processing Letters (IPL)*, 51(4):207–211, 1994.

12 David Eppstein. Subgraph isomorphism in planar graphs and related problems. *J. Graph Algorithms Appl.*, 3(3):1–27, 1999.

13 David Eppstein, Michael T. Goodrich, Michael Mitzenmacher, and Manuel R. Torres. 2-3 cuckoo filters for faster triangle listing and set intersection. In *Proceedings of ACM Symposium on Principles of Database Systems (PODS)*, pages 247–260, 2017.

14 David Eppstein, Maarten Loffler, and Darren Strash. Listing all maximal cliques in sparse graphs in near-optimal time. In *International Symposium on Algorithms and Computation (ISAAC)*, volume 6506, pages 403–414, 2010.

15 Peter Floderus, Miroslaw Kowaluk, Andrzej Lingas, and Eva-Marta Lundell. Detecting and counting small pattern graphs. *SIAM J. Discret. Math.*, 29(3):1322–1339, 2015.

16 Fedor V. Fomin, Daniel Lokshtanov, Venkatesh Raman, Saket Saurabh, and B. V. Raghavendra Rao. Faster algorithms for finding and counting subgraphs. *Journal of Computer and System Sciences (JCSS)*, 78(3):698–706, 2012.

17 Pierre-Louis Giscard, Nils M. Kriege, and Richard C. Wilson. A general purpose algorithm for counting simple cycles and simple paths of any length. *Algorithmica*, 81(7):2716–2737, 2019.

18 Chinh T. Hoang, Marcin Kaminski, Joe Sawada, and R. Sritharan. Finding and listing induced paths and cycles. *Discrete Applied Mathematics*, 161(4-5):633–641, 2013.

19 Xiao Hu and Ke Yi. Towards a worst-case i/o-optimal algorithm for acyclic joins. In *Proceedings of ACM Symposium on Principles of Database Systems (PODS)*, pages 135–150, 2016.

20 Xiaocheng Hu, Miao Qiao, and Yufei Tao. I/O-efficient join dependency testing, loomis-whitney join, and triangle enumeration. *Journal of Computer and System Sciences (JCSS)*, 82(8):1300–1315, 2016.

21 Xiaocheng Hu, Yufei Tao, and Chin-Wan Chung. I/O-Efficient Algorithms on Triangle Listing and Counting. *ACM Transactions on Database Systems (TODS)*, 39(4):27:1–27:30, 2014.

22 Svante Janson. Large deviations for sums of partly dependent random variables. *Random Structures and Algorithms*, 24(3):234–248, 2004.

23 Bas Ketsman, Dan Suciu, and Yufei Tao. A near-optimal parallel algorithm for joining binary relations. *Log. Methods Comput. Sci.*, 18(2), 2022.

24 Mahmoud Abo Khamis, Hung Q. Ngo, Christopher Ré, and Atri Rudra. Joins via geometric resolutions: Worst-case and beyond. In *Proceedings of ACM Symposium on Principles of Database Systems (PODS)*, pages 213–228, 2015.

25 Mahmoud Abo Khamis, Hung Q. Ngo, Christopher Re, and Atri Rudra. Joins via geometric resolutions: Worst case and beyond. *ACM Transactions on Database Systems (TODS)*, 41(4):22:1–22:45, 2016.

26 Ton Kloks, Dieter Kratsch, and Haiko Müller. Finding and counting small induced subgraphs efficiently. *Information Processing Letters (IPL)*, 74(3-4):115–121, 2000.

27 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.

28 Gonzalo Navarro, Juan L. Reutter, and Javiel Rojas-Ledesma. Optimal joins using compact data structures. In *Proceedings of International Conference on Database Theory (ICDT)*, volume 155, pages 21:1–21:21, 2020.

29 Jaroslav Nesetril and Svatopluk Poljak. On the complexity of the subgraph problem. *Commentationes Mathematicae Universitatis Carolinae*, 26(2):415–419, 1985.

30 Hung Q. Ngo, Dung T. Nguyen, Christopher Re, and Atri Rudra. Beyond worst-case analysis for joins with minesweeper. In *Proceedings of ACM Symposium on Principles of Database Systems (PODS)*, pages 234–245, 2014.

31 Hung Q. Ngo, Ely Porat, Christopher Ré, and Atri Rudra. Worst-Case Optimal Join Algorithms: [Extended Abstract]. In *Proceedings of ACM Symposium on Principles of Database Systems (PODS)*, pages 37–48, 2012.

32 Hung Q. Ngo, Ely Porat, Christopher Re, and Atri Rudra. Worst-case optimal join algorithms. *Journal of the ACM (JACM)*, 65(3):16:1–16:40, 2018.

33 Hung Q. Ngo, Christopher Re, and Atri Rudra. Skew strikes back: new developments in the theory of join algorithms. *SIGMOD Rec.*, 42(4):5–16, 2013.

34 Anna Pagh and Rasmus Pagh. Scalable computation of acyclic joins. In *Proceedings of ACM Symposium on Principles of Database Systems (PODS)*, pages 225–232, 2006.

35 Rasmus Pagh and Francesco Silvestri. The input/output complexity of triangle enumeration. In *Proceedings of ACM Symposium on Principles of Database Systems (PODS)*, pages 224–233, 2014.

36 Edward R. Scheinerman and Daniel H. Ullman. *Fractional Graph Theory: A Rational Approach to the Theory of Graphs.* Wiley, New York, 1997.

37 Todd L. Veldhuizen. Triejoin: A simple, worst-case optimal join algorithm. In *Proceedings of International Conference on Database Theory (ICDT)*, pages 96–106, 2014.

38 Virginia Vassilevska Williams and Ryan Williams. Finding, minimizing, and counting weighted subgraphs. *SIAM Journal of Computing*, 42(3):831–854, 2013.

A Proof of Lemma 3

Every tuple $\boldsymbol{t} \in join(\mathcal{Q})$ defines $|E_Q|$ edges in G as follows: for every relation $R \in \mathcal{Q}$ with $schema(R) := \{X_1, X_2\}$, $\boldsymbol{t}$ defines an edge $\{\boldsymbol{t}(X_1), \boldsymbol{t}(X_2)\}$ in G. In general, for every occurrence G_{sub} of Q in G, there must be at least one tuple $\boldsymbol{t} \in join(\mathcal{Q})$ defining exactly the $|E_Q|$ edges in G_{sub}. The reverse, however, is not true: the $|E_Q|$ edges of a tuple $\boldsymbol{t} \in join(\mathcal{Q})$ may not always induce a subgraph of G isomorphic to Q.

Every time a tuple $\boldsymbol{t} \in join(\mathcal{Q})$ is emitted, all the $|E_Q|$ edges defined by $\boldsymbol{t}$ are memory-resident. Hence, we can check for free if those edges induce a subgraph of G isomorphic to Q. If the answer is negative, we ignore $\boldsymbol{t}$. Next, let us focus on the scenario where the $|E_Q|$ edges *do* induce a subgraph G_{sub} isomorphic to Q. If we always emit G_{sub} in such a case, we may risk emitting G_{sub} multiple times because $join(\mathcal{Q})$ can contain multiple tuples all of which define the edges of G_{sub}. Let S be the set of those tuples. To avoid duplicate emissions, a simple strategy is to impose an (arbitrary) ordering on S, and emit G_{sub} only if $\boldsymbol{t}$ is the smallest tuple in S according to the ordering. Whether $\boldsymbol{t}$ is indeed the smallest can be checked in memory with no extra I/Os. This is because S is determined by the $|E_Q|$ edges defined by $\boldsymbol{t}$ and, hence, can be enumerated in memory for free.

It is clear from the above discussion that, apart from the initial construction of $\mathcal{Q}$ which incurs $O(\lceil |E|/B \rceil)$ I/Os, we can emit all the occurrences of Q in G with no more I/Os compared to evaluating $\mathcal{Q}$. This completes the proof of the lemma.

B Obtaining the Input Relations of Each $\mathcal{Q}_\gamma$ in Section 3.1

Decide an arbitrary ordering on the attributes of $\mathcal{X}$; w.l.o.g., denote the attributes as $X_1, X_2, ..., X_k$ in ascending order. Every color scheme γ can now be represented as a vector $(\gamma(X_1), \gamma(X_2), ..., \gamma(X_k))$. Let us impose a lexicographic order on the s^k color schemes, viewing each $(\gamma(X_1), \gamma(X_2), ..., \gamma(X_k))$ as a k-character string. Let R be a relation in $\mathcal{Q}$; w.l.o.g., assume that $schema(R) = \{X_i, X_j\}$ for some i, j satisfying $1 \le i < j \le k$. In preprocessing, we sort the tuples $\boldsymbol{t} \in R$ by lexicographic order on $(\Gamma(\boldsymbol{t}(X_i)), \Gamma(\boldsymbol{t}(X_j)))$ – viewing the pair as a 2-character string – and group those tuples by $(\Gamma(\boldsymbol{t}(X_i)), \Gamma(\boldsymbol{t}(X_j)))$ in the disk. We do so for all the relations $R \in \mathcal{Q}$; the total preprocessing cost is $O(\text{sort}(N))$.

As mentioned in Section 4.1, we deploy BNL to evaluate the joins $\mathcal{Q}_\gamma$ induced by all the color schemes γ. We do so according to the lexicographic order on $(\gamma(X_1), \gamma(X_2), ..., \gamma(X_k))$. For each γ, every relation $R_\gamma \in \mathcal{Q}_\gamma$ is a group inside the relation $R \in \mathcal{Q}$ and, hence, has been stored in consecutive blocks. The groups of each relation $R \in \mathcal{Q}$ are accessed in the same lexicographic order determined in preprocessing. For each γ, the I/O cost of reading the input relations of $\mathcal{Q}_\gamma$ is dominated by that of BNL.

C Completing the Proof of Lemma 8

Because the analysis of $R^{hi,lo}_{c_1,c_2}$ is symmetric to that of $R^{lo,hi}_{c_1,c_2}$, we will discuss only the former. Define $R^{hi,lo}_{c_1}$ as the set of tuples $\boldsymbol{t} \in R$ such that $\boldsymbol{t}(X_1)$ is a high-degree value mapped to color c_1, and $\boldsymbol{t}(X_2)$ is a low-degree value. Hence, $|R^{hi,lo}_{c_1}| \le 5\lambda$ (because "group c_1" – see the creation of function Γ_2 in Section 3.3 – has a total degree at most 5λ). For each tuple $\boldsymbol{t} \in R^{hi,lo}_{c_1}$, introduce a random variable $Z_{\boldsymbol{t}}$ that equals 1 if $\boldsymbol{t} \in R^{hi,lo}_{c_1,c_2}$, or 0 otherwise. Our function Γ_1 ensures $\Pr[Z_{\boldsymbol{t}} = 1] = 1/s$ and $\mathbf{Var}(Z_{\boldsymbol{t}}) = \frac{1}{s} - \frac{1}{s^2}$. Define $Z := |R^{hi,lo}_{c_1,c_2}| = \sum_{\boldsymbol{t} \in R^{hi,lo}_{c_1}} Z_{\boldsymbol{t}}$. Create a dependency graph $G^{hi,lo}_{c_1}$ as follows. Each vertex of $G^{hi,lo}_{c_1}$ is the variable $Z_{\boldsymbol{t}}$ of a distinct tuple $\boldsymbol{t} \in R^{hi,lo}_{c_1}$. Two vertices $Z_{\boldsymbol{t_1}}$ and $Z_{\boldsymbol{t_2}}$ are adjacent in $G^{hi,lo}_{c_1}$ if and only if tuples $\boldsymbol{t_1}$ and $\boldsymbol{t_2}$ share the same value on attribute X_2. It is easy to verify that $G^{hi,lo}_{c_1}$ fulfils the independence requirement described in Section 2.3 and has a maximum vertex degree at most $\lambda/\sqrt{M}$. Applying Lemma 4 with $\mu := \mathbf{E}[Z] = |R^{hi,lo}_{c_1}|/s$, $\sigma := |R^{hi,lo}_{c_1}|/s > \sum_{\boldsymbol{t} \in R^{hi,lo}_{c_1}} \mathbf{Var}(Z_{\boldsymbol{t}})$, $\Delta := \lambda/\sqrt{M}$, and $\epsilon := Ms/|R^{hi,lo}_{c_1}|$ yields $\Pr[Z \ge 6M] \le \exp(-\Theta(1) \cdot \frac{M^{1.5}}{\lambda})$, which is at most $1/(4\lambda^{\xi'})$ as long as $M = \Omega((\lambda \log \lambda)^{2/3})$.

D Proof of Lemma 11

The first statement follows directly from Lemma 6 of [23]. We will prove only the second statement. We first perform $O(\text{sort}(N))$ I/Os to obtain, for each attribute $X \in \mathcal{H}$, the list of all the $O(\sqrt{N/M})$ heavy values in the active domain of $\mathcal{X}$. Then, we compute the cartesian product – denoted as S – of the $|\mathcal{H}|$ lists in $O(|S|/B)$ I/Os; note that S is the set of all configurations. In another $O(\text{sort}(|S|))$ I/Os, we can remove from S those configurations violating (12) and, thus, produce $config(\mathcal{H})$ in the disk. The cost so far is $O(\text{sort}(|S|)) = O(\frac{N^{k/2}}{M^{k/2-1}B} \log_{M/B} \frac{N}{B})$ because $|S| = O((\sqrt{N/M})^{|\mathcal{H}|})$.

Next, we explain how to generate the input relations of the residual joins $\mathcal{Q}'(\boldsymbol{\eta})$ for all $\boldsymbol{\eta} \in config(\mathcal{H})$. For this purpose, consider each hyperedge $e \in \mathcal{E}$ in turn (recall that $\mathcal{G} := (\mathcal{X}, \mathcal{E})$ is the schema graph of $\mathcal{Q}$). If $e \cap \mathcal{H} = \emptyset$, R_e (i.e., the relation in $\mathcal{Q}$ with schema e) appears in all the residual joins $\mathcal{Q}'(\boldsymbol{\eta})$ where $\boldsymbol{\eta} \in config(\mathcal{H})$. As R_e is already stored in consecutive blocks, this relation requires no more processing. If $e \subseteq \mathcal{H}$, then R_e contributes nothing to residual joins. It remains to discuss the case where $e \cap \mathcal{H}$ has a single attribute.

W.o.l.g., assume that $e = \{X_1, X_2\}$ with $X_1 \notin \mathcal{H}$ but $X_2 \in \mathcal{H}$. Each tuple $\boldsymbol{t} \in R_e$ appears in the residual join of every configuration $\boldsymbol{\eta} \in \mathit{config}(\mathcal{H})$ satisfying $\boldsymbol{\eta}(X_2) = \boldsymbol{t}(X_2)$; the number of such $\boldsymbol{\eta}$ is $O((\sqrt{N/M})^{|\mathcal{H}|-1})$. To compute residual relations, we first sort both R_e and $\mathit{config}(\mathcal{H})$ on attribute X_2 and then produce $R_e \bowtie \mathit{config}(\mathcal{H})$ in the disk by merging the two sorted lists; the cost is $O(\text{sort}(|\mathit{config}(\mathcal{H})|)+\text{sort}(N)+|R_e \bowtie \mathit{config}(\mathcal{H})|/B)$. Then, group the tuples $\boldsymbol{t} \in R_e \bowtie \mathit{config}(\mathcal{H})$ by $\boldsymbol{t}[\mathcal{H}]$, which can be done in $O(\text{sort}(|R_e \bowtie \mathit{config}(\mathcal{H})|))$ I/Os. Each group corresponds to a configuration $\boldsymbol{\eta} \in \mathit{config}(\mathcal{H})$ where $\boldsymbol{\eta} = \boldsymbol{t}[\mathcal{H}]$ for an arbitrary tuple $\boldsymbol{t}$ in the group (all tuples in the group share the same $\boldsymbol{t}[\mathcal{H}]$). The X_1-values of the group's tuples constitute the residual relation $R'_e(\boldsymbol{\eta})$. The total I/O cost is $O(\frac{N^{k/2}}{M^{k/2-1}B} \log_{M/B} \frac{N}{B})$ because $|R_e \bowtie \mathit{config}(\mathcal{H})| = O(N \cdot (\sqrt{N/M})^{|\mathcal{H}|-1})$ and $|\mathcal{H}| \leq k-1$ (recall that $X_1 \notin \mathcal{H}$).

An Optimal Algorithm for Sliding Window Order Statistics

Pavel Raykov
Google Zürich, Switzerland

Abstract

Assume there is a data stream of elements and a window of size m. Sliding window algorithms compute various statistic functions over the last m elements of the data stream seen so far. The time complexity of a sliding window algorithm is measured as the time required to output an updated statistic function value every time a new element is read. For example, it is well known that computing the sliding window maximum/minimum has time complexity $O(1)$ while computing the sliding window median has time complexity $O(\log m)$. In this paper we close the gap between these two cases by (1) presenting an algorithm for computing the sliding window k-th smallest element in $O(\log k)$ time and (2) prove that this time complexity is optimal.

2012 ACM Subject Classification Theory of computation → Sorting and searching

Keywords and phrases sliding window, order statistics, median, selection algorithms

Digital Object Identifier 10.4230/LIPIcs.ICDT.2023.5

Acknowledgements We thank anonymous reviewers for their useful comments which led to an improved manuscript. We would also like to thank Mikhail Churakov, Christian Matt, and Dimitris Paparas for proofreading the paper.

1 Introduction

Selection and sliding window algorithms are considered to be among the classical computer science algorithms with numerous applications [7]. In this paper we consider the overlap between these two areas: what is the optimal sliding window algorithm for selecting the k-th smallest element? The sliding window average, median, minimum, and maximum algorithms have been well studied and have become a part of folklore. It is well known that for a data stream of elements and a window of size m, one can compute the sliding window average, minimum, and maximum in $O(1)$ time each time the sliding window moves [9, 25]. The situation is different for the median – the best possible algorithm can only compute the sliding window median in $O(\log m)$ time [11, 14, 24]. Motivated by the gap between the median and other order statistics algorithms, we study the sliding window algorithms for selecting the k-th smallest element for arbitrary k. In this paper we present the first algorithm computing the sliding window k-th smallest element in time $O(\log k)$ while using only $O(m)$ of memory storage. We also present a lower bound showing that this algorithm has the optimal running time complexity.

1.1 Prior Work

Algorithms for computing various statistics over a data stream play an important role in computer science and database processing in particular [4]. One can roughly divide these algorithms into two groups: exact and approximate ones. The research in the area of the approximate algorithms focuses on the problem where the whole data stream (or the sliding window) cannot fit into the memory and hence one seeks for a tradeoff between how much of the data need to be stored additionally while reading the input stream a limited number of times vs how accurate the output computed statistics can be [2, 3, 5, 8, 12, 18].

The exact algorithms can be further subdivided into those who operate on the whole input set and on the sliding window only. Once it was shown that the selection algorithms can be linear [6], the research in the area of the exact algorithms that work with the whole input set focused on algorithms minimizing the overall number of basic operations [22] and the algorithms performing well in practice [1]. There is also a corpus of the exact algorithms that do not only compute a statistical function over a fixed data set but also support range queries over it [17, 27].

The exact algorithms working with the sliding window include algorithms for standard aggregation functions like maximum, minimum, average, sum, count [9, 19, 26] and their monoid-compatible extensions [25]. The exact algorithms for computing quantiles over the sliding window are limited to the median filtering algorithms [14, 24]. These algorithms produce their output over the sliding window by maintaining a data structure holding the elements of the window and updating it accordingly whenever the window moves. Depending on the aggregation function all the known exact sliding window algorithms can update the sliding window in constant, logarithmic, or linear time in terms of the sliding window size.

This paper considers the latter model where the aggregation function outputs the k-th smallest element. For a window of size m and the order statistic k, we present an algorithm that requires $O(m)$ memory to store the elements of the sliding window and can make updates to it in $O(\log k)$ time whenever the sliding window moves.

2 Notation and Tools

We define a sliding window algorithm as an algorithm that exposes a single interface update-window reading a new data stream element v and outputting a statistic function over the last m elements read. If fewer than m elements have been read so far, the output is not defined. *The time complexity* of the algorithm is defined as the time complexity of a single invocation of update-window. *The space complexity* of the algorithm is defined as the amount of storage the algorithm utilises.

In the context of this paper we consider the sliding window algorithms that compute the k-th smallest element of the sliding window:

▶ **Definition 1.** *We say that a sliding window algorithm parameterised with integers k and m computes* the sliding window k-smallest element *if update-window returns the k-th smallest element among the last m elements read by the algorithm.*

The smallest element is indexed starting with 1, i.e., the 1-st smallest element corresponds to the window minimum, while the m-th smallest element corresponds to the window maximum.

We will also assume that $k \leq m/2$ everywhere. To derive the same results for $k > m/2$, one needs to update all the algorithms to use a reverse order on the elements and output the $(m-k+1)$-th smallest element.

Without loss of generality, we assume that the input data stream has unique elements only.[1] The easiest approach to achieve this is to enumerate each new data stream value v with an increasing index i and then operate on the tuples (i, v) instead of the values only. Then, we can compare the tuples in the natural way by first looking at their values, and if they are the same, break the ties using the index entry. In the end, when producing the output one needs to drop the index entry of the tuples and output the value only.

[1] We use this assumption to simplify the machinery around the binary search trees which do not have a canonical multiset support. We believe that this requirement can be dropped at an expense of a more sophisticated analysis of handling the duplicate elements.

We employ a standard notion of arrays. Given an array a, we refer to an individual element in the array at index i by writing $a[i]$; we start array indexing with 0 by default. We write $a[i,j]$ to denote the range of array elements $a[i], a[i+1], \ldots, a[j]$. If $j < i$, then the expression $a[i,j]$ returns the empty set. Let k-smallest-set($a[i,j]$) denote the set of the k smallest elements in $a[i,j]$.

We assume there exists an implementation of AWBBS (augmented weight-balanced binary search) trees [20](e.g., based on red-black trees where each node is additionally *augmented* with the size of its subtree) that supports execution of the following operations on a tree with n elements in $O(\log n)$ time:

- add: adds an element to the tree;
- remove: removes an element from the tree;
- max: outputs the maximum element of the tree;
- size: outputs the size of the tree;
- find-rank-one-tree: outputs an element of the tree with the given rank.

We also assume that the tree can be implemented using $O(n)$ space to store n elements.

Given several separate trees we define the k-th smallest element between them to be the k-th smallest element among all the elements of the trees. Given three AWBBS trees of size at most n we assume that there is a function computing the k-th smallest element between them in time $O(\log n)$. An example implementation of such a function find-rank-three-trees is given in Section A.

3 The Algorithm

3.1 Overview

The first trivial approach for the k-th smallest element sliding window algorithm is to maintain an AWBBS tree with m elements of the sliding window. Then, at each invocation of update-window we just execute add for the newly added element, call remove on the element that falls outside of the sliding window, and then search for the element with rank k in the tree by invoking find-rank-one-tree. The issue with this algorithm is that each operation takes time proportional to the size of the sliding window $O(\log m)$ and we would like to design a faster algorithm.

The first refinement of the trivial approach is to limit the size of the AWBBS tree by storing only the k smallest elements of the sliding window in it. This will bring down the complexity of the elementary operations from $O(\log m)$ to $O(\log k)$ but it is not clear how to implement this approach. In particular, while adding a new element to such a tree is straightforward (we add a new element and then remove the maximum if the tree size exceeds k), removal of the elements that fall outside of the sliding window is not clear. If the removed element is one of the k smallest elements we need to find the smallest element outside of the maintained tree that now needs to be added to it. This seems to require sorting the elements of the sliding window which again incurs $O(\log m)$ costs.

The second refinement is based on the following idea: while it does not seem to be possible to maintain a single AWBBS tree with the k smallest elements of the sliding window, we can split the sliding window into separate blocks and maintain a separate AWBBS tree with the k smallest elements of *each block* and not the whole sliding window. Then, finding the k-th smallest element of the sliding window now requires searching for an element with rank k in multiple AWBBS trees which still can be done in $O(\log k)$ time (see Section A for an example of such an algorithm). Note that the "maintenance burden" of keeping the AWBBS tree with the k smallest window elements is now distributed among multiple blocks. In particular, one of the blocks will receive a new data stream element, while another one has the element falling outside of the sliding window and has to deal with its deletion. The trick is that while the window slides we will have enough time to prepare each block for deletions so that they also will incur only $O(\log k)$ costs once the deletion happens.

3.2 Description

We assume that the input data stream is split into consecutive blocks of size $m/2$.[2] At any moment of time the sliding window W spans over three consecutive blocks B_{left}, B_{middle} and B_{right} as shown in the picture:

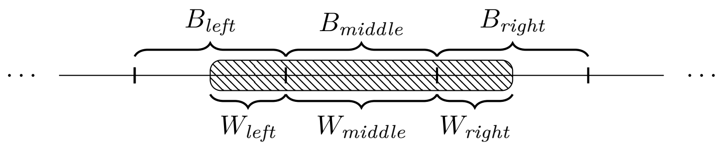


Figure 1 Sliding window ▨ spanning over $B_{left}, B_{middle}, B_{right}$.

The sliding window then consists of the following three parts: W_{left} intersecting with B_{left}, W_{middle} equals to B_{middle}, and W_{right} intersecting with B_{right}.[3] The proposed algorithm maintains AWBBS trees T_{left}, T_{middle}, and T_{right} with the k smallest elements of the W_{left}, W_{middle}, and W_{right} parts, respectively. Computing the k-th smallest element of the sliding window is done by calling function find-rank-three-trees (see Section A for its description) over T_{left}, T_{middle}, and T_{right}.

The maintenance of the trees is done separately for each of the trees. T_{right} is maintained in a straightforward way as we have already noticed before: we add a new element to it and then remove the maximum if the size of T_{right} exceeds k. Nothing is done for maintaing T_{middle}: it is just assigned to T_{right} every $m/2$ steps. The core of the algorithm lies in preparing a data for maintaining T_{left}. This data is computed based on B_{middle} and then used for maintaining T_{left} once the window slides to a point of time when the elements of B_{middle} become B_{left}. We prepare a data structure which allows us to move from a tree containing k-smallest-set($B_{middle}[0, m/2-1]$) to a tree containing k-smallest-set($B_{middle}[1, m/2-1]$) (then to a tree containing k-smallest-set($B_{middle}[2, m/2-1]$) and so on). The difficulty is that while we know that we need to remove $B_{middle}[0]$ from the k-smallest-set($B_{middle}[0, m/2-1]$) , we do not know which element should be added back to it if $B_{middle}[0]$ was one of the k smallest elements. We solve this by first learning how to construct k-smallest-set($B_{middle}[0, m/2-1]$) from k-smallest-set($B_{middle}[1, m/2-1]$). This can be done by the same procedure we described for T_{right}: first add $B_{middle}[0]$ to k-smallest-set($B_{middle}[1, m/2-1]$) and then remove the maximum from this set if its size exceeds k. The key observation is that while doing this we can record the maximum max that was removed. This allows us to "revert" the process in order to obtain k-smallest-set($B_{middle}[1, m/2-1]$) from k-smallest-set($B_{middle}[0, m/2-1]$): we add this max and then remove $B_{middle}[0]$. Now we record these max's during the preparation phase. The preparation costs are also distributed across the iterations so that the total complexity stays at $O(\log k)$ per invocation of update-window. The maintenance of T_{left} is now done based on the max array computed when B_{left} was B_{middle}: we add the stored max element and remove the element falling outside the window from T_{left}.

[2] Here and everywhere through paper we assume that m is even to simplify the notation. If m is odd one needs to update all the places where $m/2$ is used with $\lfloor m/2 \rfloor$ or $\lceil m/2 \rceil$ accordingly.
[3] We use a convention that if W is shifted by a multiple of $m/2$ then W_{left} is empty while $W_{right} = B_{right}$.

Algorithm 1 The sliding window k-th smallest element algorithm.

```
Input: integers $k, m$
1: Initialize $T_{left}$, $T_{middle}$, $T_{right}$, and $T_{temp}$ as empty AWBBS trees.
2: Initialize incoming elements counter $j$ as 0.
3: Initialize $B_{left}, B_{middle}, B_{right}$ as arrays of size $m/2$.
4: Initialize $max$ and $max_{temp}$ as arrays of size $m/2 - k$.
5: procedure UPDATE-WINDOW($v$)
6:     ▷ General updates:
7:     $shift := j \bmod m/2$
8:     $j := j + 1$
9:     if $shift = 0$ then
10:        $max := max_{temp}$
11:        $T_{left} := T_{middle}$
12:        $T_{middle} := T_{right}$
13:        $B_{left} := B_{middle}$
14:        $B_{middle} := B_{right}$
15:        Set $T_{right}$ and $T_{temp}$ to empty tree and $B_{right}$ to a new array of size $m/2$
16:    end if
17:    $B_{right}[shift] := v$
18:
19:    ▷ Maintaining $T_{right}$:
20:    add($T_{right}, v$)
21:    if size($T_{right}$) $> k$ then
22:        remove($T_{right}$, max($T_{right}$))
23:    end if
24:
25:    ▷ Preparing $B_{middle}$:
26:    if $j > m/2$ then
27:        add($T_{temp}, B_{middle}[m/2 - 1 - shift]$)
28:        if $shift \geq k$ then
29:            $max_{temp}[m/2 - 1 - shift]$ := max($T_{temp}$)
30:            remove($T_{temp}, max_{temp}[m/2 - 1 - shift]$)
31:        end if
32:    end if
33:
34:    ▷ Maintaining $T_{left}$:
35:    if $j > m$ then
36:        if $shift \leq m/2 - k - 1$ then
37:            add($T_{left}, max[shift]$)
38:        end if
39:        remove($T_{left}, B_{left}[shift]$)
40:    end if
41:
42:    ▷ Producing the output (if $j \geq m$)
43:    Output find-rank-three-trees($T_{left}, T_{middle}, T_{right}, k$)
44: end procedure
```

▶ **Theorem 2.** *Algorithm 1 computes the sliding window k-th smallest element with time complexity $O(\log k)$ and space complexity $O(m)$.*

Proof. The algorithm splits the update-window method into four logical parts:

- (lines 6–17) The general part responsible for keeping the counters and proper wiring of the *left*,*middle*,*right* parts whenever we reach a block's boundary (when the *shift* variable is 0).
- (lines 19–23) The T_{right} maintenance part.
- (lines 25–32) The B_{middle} preparation part.
- (lines 34–40) The T_{left} maintenance part.

The proof will show that at each iteration T_{left}, T_{middle}, and T_{right} indeed contain the k smallest elements of the corresponding window parts.

Correctness. We start by proving three lemmata about the state of T_{left}, T_{middle}, and T_{right} in the update-window method.

▶ **Lemma 3.** *In the end of the method update-window at line 43 T_{right} contains only k-smallest-set$(B_{right}[0, shift])$.*

Proof. Consider an iteration where the right part of the sliding window is $B_{right}[0, shift]$. By construction we consecutively added elements $B_{right}[0], B_{right}[1], \ldots, B_{right}[shift]$ to T_{right}. Whenever the size of T_{right} grew bigger than k (line 21), we removed the largest element of T_{right} (line 22) to bring the size of T_{right} back to k. Hence T_{right} contains only k-smallest-set$(B_{right}[0, shift])$. ◀

▶ **Lemma 4.** *If $j > m/2$, in the end of the method update-window at line 43 T_{middle} contains only k-smallest-set$(B_{middle}[0, m/2 - 1])$.*

Proof. Note that T_{middle} only changes at line 12 whenever the next $m/2$ elements have been consumed and *shift* equals to 0. Because of Lemma 3, we know that at such an iteration T_{right} contains only k-smallest-set$(B_{right}[0, m/2 - 1])$. Since T_{middle} is assigned to T_{right} at line 12 and B_{middle} is assigned to B_{right} at line 14, we conclude that T_{middle} always contains only k-smallest-set$(B_{middle}[0, m/2 - 1])$. ◀

▶ **Lemma 5.** *If $j > m$, in the end of the method update-window at line 43 T_{left} contains only k-smallest-set$(B_{left}[shift + 1, m/2 - 1])$ if $shift < m/2 - 1$ and is empty if shift is $m/2 - 1$.*

Proof. We start by proving a slightly different statement that at line 36 T_{left} contains only k-smallest-set$(B_{left}[shift, m/2 - 1])$. We prove this by induction on the *shift* variable. The base case of $shift = 0$ holds by construction since in this case T_{left} has just been assigned to T_{middle}, while B_{left} has been assigned to B_{middle}. Because of Lemma 4 it holds that T_{middle} contains only k-smallest-set$(B_{middle}[0, m/2 - 1])$, and hence T_{left} contains only k-smallest-set$(B_{left}[0, m/2 - 1])$ at line 36 when *shift* is 0. Assume now this holds for some $shift = t$ and we need to prove it for $shift = t + 1$. By the preparation phase at lines 25–32 $max[t]$ is the maximum element of k-smallest-set$(B_{left}[t + 1, m/2 - 1])$ with the added $B_{left}[t]$. Hence, adding $max[t]$ to k-smallest-set$(B_{left}[t, m/2 - 1])$ and removing $B_{left}[t]$ results in k-smallest-set$(B_{left}[t + 1, m/2 - 1])$ which becomes the new value of T_{left} after line 40 is completed. Finally, we observe that whenever *shift* becomes strictly bigger than $m/2 - k - 1$ then the size of the left part of the window becomes $\leq k$ and it is sufficient to always remove $B_{left}[shift]$ from T_{left} in this part of the maintenance phase so that T_{left} always contains k-smallest-set$(B_{left}[shift, m/2 - 1])$.

The main lemma statement follows from observing that if at line 36 T_{left} contains only k-smallest-set$(B_{left}[shift, m/2-1])$, then at line 43 T_{left} must contain only k-smallest-set$(B_{left}[shift+1, m/2-1])$ for *shift* values $< m/2-1$ while for $shift = m/2-1$ the tree T_{left} is empty. ◀

Now we have that T_{left} consists of the k smallest elements of $B_{left}[shift+1, m/2-1]$, T_{middle} consists of the k smallest elements of $B_{middle}[0, m/2-1]$, and T_{right} consists of the k smallest elements of $B_{right}[0, shift]$. Since find-rank-three-trees is correct we have that find-rank-three-trees will output a correct k-th smallest element of T_{left}, T_{middle} and T_{right} every time line 43 is executed.

Complexity analysis. In update-window we invoke max at most two times (lines 22, 29), add at most three times (lines 20, 27, 37), and remove at most three times (lines 22, 30, 39). While these operations take $O(\log k)$ time, all other operations are elementary (like assigning variables and objects) and take $O(1)$ time. Also, the specialized operation find-rank-three-trees takes $O(\log k)$ time (see Theorem 11). Hence, the time complexity of update-window is $O(\log k)$.

The space complexity is $O(m)$ since we have used $O(k)$ memory for the trees T_{left}, T_{middle}, T_{right}, T_{temp}; and $O(m)$ memory for storing the window parts $B_{left}, B_{middle}, B_{right}$ and auxiliary arrays *max* and max_{temp}.

We also note that array assigning operations among B_{left}, B_{middle} and B_{right} can take $\Omega(m)$ time if implemented naively. Instead, whenever we assign arrays to each other we assign their pointers in $O(1)$ time and *do not* copy the array contents. Furthermore, we do not need to allocate a new memory for B_{right} at line 15 – instead, it is sufficient to assign its pointer to the contents of B_{left} which become unused otherwise. With this approach all array allocations happen once at lines 3 and 4 only. ◀

3.3 Discussing the Preparation Phase

Essentially, the preparation phase allows us to create a directly accessible view of every k-smallest-set$(B_{middle}[i, m/2-1])$ for $i = 0, 1, \ldots, m-1$. First of all, we could have simplified the algorithm by running all the preparation steps at once whenever B_{middle} becomes available and extract this functionality as a separate method. While the amortised time complexity of the algorithm would not change in this case, the worst case complexity would become $O(m \log k)$ instead of $O(\log k)$ which is undesirable. Second, instead of building an ad-hoc algorithm computing *max* array used for "reverting" operations on AWBBS trees, we could have tried using the standard approach for persistent data structures [10] to build the sequence of AWBBS trees [21, 13, 23]. However, as explained in [17] such a generic approach would require copying the tree paths each time add/remove operation is invoked. This would lead to the suboptimal $O(m \log k)$ space complexity of the algorithm.

4 Lower Bounds

In order to prove lower bounds on the time complexity for the sliding window k-smallest element algorithms we give a reduction from them to sorting algorithms. Then, based on the well-known lower bounds for sorting algorithms we establish the lower bounds for the sliding window k-smallest element algorithms.

We give a reduction from computing the sliding window k-th smallest element to piecewise sorting. As opposed to the classical sorting piecewise sorting only sorts contiguous blocks of the input array without ensuring any order between the elements from different blocks. In order to sort each contiguous block of size k of an input array we will run a sliding window k-smallest element algorithm on this array with the caveat that when comparing elements from blocks i and j we use a custom comparison operator ensuring that all elements from block i are smaller than the elements from block j for $i < j$.

We note that our reduction is similar to the reductions that are used to prove the lower bounds on the sliding window median algorithms [11, 14, 17, 24]. While all these papers also present a reduction to sorting, the main difference to the reduction presented below is that all the previous reductions could only be adapted to work in a setting where k is $\Theta(m)$ ($k = m/2$ is a concrete case of the median considered there), whereas the reduction presented in this paper works for arbitrary $k \leq m/2$.

▶ **Definition 6.** *An array a is* k-piecewise sorted *if each contiguous k-size block $a[k \cdot i, k \cdot (i+1) - 1]$ is sorted.*

An algorithm that takes an input array a and outputs array x which is k-piecewise sorted and each contiguous k-size block $x[k \cdot i, k \cdot (i+1) - 1]$ is a permutation of $a[k \cdot i, k \cdot (i+1) - 1]$ is called a k-piecewise sorting algorithm*.*

Algorithm 2 A reduction from sliding window k-smallest element to k-piecewise sorting.

```
Input: array of $n$ elements $a[0], a[1], \dots, a[n-1]$
Output: array of $n$ elements $x[0], x[1], \dots, x[n-1]$ where each consecutive $k$ size block
$x[k \cdot i, k \cdot (i+1) - 1]$ is sorted and is a permutation of $a[k \cdot i, k \cdot (i+1) - 1]$.
1: Consider array $s$ of $n + m - 1$ elements where
     - ($k - 1$ prefix elements) $s[i] := (-\infty, -\infty)$ for $i = -1, -2, \dots, -k+1$;
     - ($n$ main elements) $s[i] := (\lfloor i/k \rfloor, a[i])$ for $i = 0, 1, \dots, n-1$;
     - ($m - k$ suffix elements) $s[i] := (+\infty, +\infty)$ for $i = n, n+1, \dots, n+m-k-1$.
2: Define comparison operator on tuples in $s$ as: $(i, v) < (j, u)$ if $(i < j)$ or $(i = j$ and
   $v < u)$.
3: Invoke a sliding window $k$-th smallest algorithm on array $s$ (by feeding $s[-k+1], s[-k+2], \dots, s[n+m-k-1]$ one by one to its update-window interface). Denote the produced
   output as $y[0], y[1], \dots, y[n-1]$.
4: Let $x[i]$ be the second tuple entry of $y[i]$ for $i = 0, 1, \dots, n-1$. Output array $x$.
```

▶ **Lemma 7.** *In the regime where $m \geq 2k$, Algorithm 2 sorts each consecutive block of size k of the input array in time $(n + m - 1) \cdot f(m, k) + O(n + m)$ where $f(m, k)$ is the time complexity of the underlying sliding window k-smallest element algorithm instantiated with the window size m and the order statistics k.*

Proof. The core of the reduction lies in the construction of the intermediate array s such that running a sliding window k-smallest element algorithm on it basically sorts the input array. The array s elements are constructed at step 1 based on the input array elements a with the feature that the elements from each contiguous block of size k in s are greater than the elements from the previous blocks. This is achieved by augmenting each element within the block with the block index and using the block index to compare the elements. Because the sliding window k-smallest element algorithm operates on windows of size m and outputs the k-the smallest element, we also need to add $k - 1$ prefix elements in the beginning of s and $m - k - 1$ suffix elements to its end, so that the sliding window algorithm operates correctly around the ends of the array.

Correctness. Consider any k-size contiguous block in the input array $a[k \cdot i, k \cdot (i+1) - 1]$. We claim that $y[k \cdot i + j]$ is the $(j+1)$-th smallest element in this array for $j = 0, 1, \ldots k-1$. By construction, $y[k \cdot i + j]$ corresponds to the k-th smallest element in the window $W = s[k \cdot (i-1) + j + 1, k \cdot (i-1) + j + m]$. We now define a partition of W around the indices divisible by k in blocks:

- we let the block B_1 contain all the window W elements up to the $(k \cdot i)$-th index, i.e., $B_1 = s[k \cdot (i-1) + j + 1, k \cdot i - 1]$;
- all consecutive blocks have fixed size k, i.e., $B_2 = s[k \cdot i, k \cdot (i+1) - 1]$, $B_3 = s[k \cdot (i+1), k \cdot (i+2) - 1]$ and so on as long as we can take a full block of size k within W.
- the last block contains the remaining elements of W.

Because $m \geq 2k$ we know that the second block $B_2 = s[k \cdot i, k \cdot (i+1) - 1]$ completely lies inside the sliding window $W = s[k \cdot (i-1) + j + 1, k \cdot (i-1) + j + m]$ and hence the window W consists of at least B_1 and B_2.

By construction, elements in the earlier blocks are smaller than the elements in the successive blocks, e.g., all the elements in B_1 are smaller than the elements in B_2. This means that the k-th smallest element in W is greater than all $k - j - 1$ elements in B_1, and is actually the $(j+1)$-th smallest element in B_2. The $(j+1)$-th smallest element in B_2 corresponds to the $(j+1)$-the smallest element in $a[k \cdot i, k \cdot (i+1) - 1]$.

Complexity analysis. The reduction takes $O(n+m)$ time to process elements at steps 1 and 4. Then, at step 3 the invocation of the underlying sliding window k-smallest element algorithm takes $f(m,k)$ time. Hence, the resulting time complexity is $(n+m-1) \cdot f(m,k) + O(n+m)$. ◀

We will now apply existing sorting lower bounds in the comparison model to obtain lower bounds for the sliding window k-th smallest element algorithms using the reduction we have just described in Lemma 7.

▶ **Theorem 8.** *Any algorithm computing the sliding window k-th smallest element has time complexity $\Omega(\log k)$ in the comparison model.*

Proof. Assume we have an algorithm that can solve sliding window k-th smallest element with time complexity $f(m,k)$. Take an arbitrary array of $n - m + 1$ elements where $m \leq n/2$. Then, based on the reduction in Lemma 7 we can k-piecewise sort this array in time $n \cdot f(m,k) + O(n)$. We know that k-piecewise sorting of $(n - m + 1)$-size array can only be done in $\Omega((n - m + 1) \cdot \log k)$ [15]. Given that $m \leq n/2$, we have that $n \cdot f(m,k) + O(n)$ must be in $\Omega(n \log k)$. This means that $f(m,k)$ is $\Omega(\log k)$. ◀

5 Conclusions and Future Work

In this paper we have presented the first optimal algorithm for finding the sliding window k-th smallest element with time complexity $O(\log k)$ and proved that this complexity is optimal. We note that the presented algorithm has fixed time complexity of $O(\log k)$ per newly read element which is independent of the window size m; it also requires only $O(m)$ memory storage.

The presented algorithm subsumes the existing results on the sliding window minimum/maximum algorithms [9, 25] (case $k = 1$) and median algorithms [14] (case $k = m/2$). One distinct feature of the specialized algorithms [9, 25, 14] is the possibility of their extreme concise representation which effectively fits into several lines of pseudocode. While the algorithm presented in this paper does share some ideas with these specialized algorithms (like maintaining an ordered tree of the window elements [14] and keeping only those window elements that can potentially become the corresponding order statistics [9]), its pseudocode representation is still significantly larger. It would be interesting to investigate if all these sliding windows algorithms can be cast to a common framework with a simplified pseudocode representation.

References

1 Andrei Alexandrescu. Fast deterministic selection. In Costas S. Iliopoulos, Solon P. Pissis, Simon J. Puglisi, and Rajeev Raman, editors, *16th International Symposium on Experimental Algorithms, SEA 2017, June 21-23, 2017, London, UK*, volume 75 of *LIPIcs*, pages 24:1–24:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPIcs.SEA.2017.24.

2 Noga Alon, Yossi Matias, and Mario Szegedy. The space complexity of approximating the frequency moments. *J. Comput. Syst. Sci.*, 58(1):137–147, 1999. doi:10.1006/jcss.1997.1545.

3 Arvind Arasu and Gurmeet Singh Manku. Approximate counts and quantiles over sliding windows. In Catriel Beeri and Alin Deutsch, editors, *Proceedings of the Twenty-third ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 14-16, 2004, Paris, France*, pages 286–296. ACM, 2004. doi:10.1145/1055558.1055598.

4 Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and issues in data stream systems. In Lucian Popa, Serge Abiteboul, and Phokion G. Kolaitis, editors, *Proceedings of the Twenty-first ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 3-5, Madison, Wisconsin, USA*, pages 1–16. ACM, 2002. doi:10.1145/543613.543615.

5 Brian Babcock, Mayur Datar, Rajeev Motwani, and Liadan O'Callaghan. Maintaining variance and k-medians over data stream windows. In Frank Neven, Catriel Beeri, and Tova Milo, editors, *Proceedings of the Twenty-Second ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 9-12, 2003, San Diego, CA, USA*, pages 234–243. ACM, 2003. doi:10.1145/773153.773176.

6 Manuel Blum, Robert W. Floyd, Vaughan R. Pratt, Ronald L. Rivest, and Robert Endre Tarjan. Time bounds for selection. *J. Comput. Syst. Sci.*, 7(4):448–461, 1973. doi:10.1016/S0022-0000(73)80033-9.

7 Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 3rd Edition.* MIT Press, 2009. URL: http://mitpress.mit.edu/books/introduction-algorithms.

8 Mayur Datar, Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Maintaining stream statistics over sliding windows. *SIAM J. Comput.*, 31(6):1794–1813, 2002. doi:10.1137/S0097539701398363.

9 Scott C. Douglas. Running max/min calculation using a pruned ordered list. *IEEE Trans. Signal Process.*, 44(11):2872–2877, 1996. doi:10.1109/78.542446.

10 James R. Driscoll, Neil Sarnak, Daniel Dominic Sleator, and Robert Endre Tarjan. Making data structures persistent. In Juris Hartmanis, editor, *Proceedings of the 18th Annual ACM Symposium on Theory of Computing, May 28-30, 1986, Berkeley, California, USA*, pages 109–121. ACM, 1986. doi:10.1145/12130.12142.

11 David Eppstein. Nontrivial algorithm for computing a sliding window median [online]. 2014. URL: https://cstheory.stackexchange.com/questions/21730/nontrivial-algorithm-for-computing-a-sliding-window-median.

12 Michael Greenwald and Sanjeev Khanna. Space-efficient online computation of quantile summaries. In Sharad Mehrotra and Timos K. Sellis, editors, *Proceedings of the 2001 ACM SIGMOD international conference on Management of data, Santa Barbara, CA, USA, May 21-24, 2001*, pages 58–66. ACM, 2001. doi:10.1145/375663.375670.

13 Dana Jansens. Persistent binary search trees [online]. 2009. URL: https://cglab.ca/~dana/pbst/.

14 Martti Juhola, Jyrki Katajainen, and Timo Raita. Comparison of algorithms for standard median filtering. *IEEE Trans. Signal Process.*, 39(1):204–208, 1991. `doi:10.1109/78.80784`.

15 Donald E. Knuth. *The Art of Computer Programming, Volume I: Fundamental Algorithms, 2nd Edition.* Addison-Wesley, 1973. URL: `https://www.worldcat.org/oclc/310903895`.

16 Georgiy Korneev. Designing an algorithm that searches for the kth largest element between two AVL trees [online]. 2016. URL: `https://stackoverflow.com/questions/40473890/designing-an-algorithm-that-searches-for-the-kth-largest-element-between-two-avl`.

17 Danny Krizanc, Pat Morin, and Michiel H. M. Smid. Range mode and range median queries on lists and trees. *Nord. J. Comput.*, 12(1):1–17, 2005.

18 Xuemin Lin, Hongjun Lu, Jian Xu, and Jeffrey Xu Yu. Continuously maintaining quantile summaries of the most recent N elements over a data stream. In Z. Meral Özsoyoglu and Stanley B. Zdonik, editors, *Proceedings of the 20th International Conference on Data Engineering, ICDE 2004, 30 March - 2 April 2004, Boston, MA, USA*, pages 362–373. IEEE Computer Society, 2004. `doi:10.1109/ICDE.2004.1320011`.

19 Bongki Moon, Inés Fernando Vega López, and Vijaykumar Immanuel. Scalable algorithms for large temporal aggregation. In David B. Lomet and Gerhard Weikum, editors, *Proceedings of the 16th International Conference on Data Engineering, San Diego, California, USA, February 28 - March 3, 2000*, pages 145–154. IEEE Computer Society, 2000. `doi:10.1109/ICDE.2000.839401`.

20 Jürg Nievergelt and Edward M. Reingold. Binary search trees of bounded balance. In Patrick C. Fischer, H. Paul Zeiger, Jeffrey D. Ullman, and Arnold L. Rosenberg, editors, *Proceedings of the 4th Annual ACM Symposium on Theory of Computing, May 1-3, 1972, Denver, Colorado, USA*, pages 137–142. ACM, 1972. `doi:10.1145/800152.804906`.

21 Chris Okasaki. Red-black trees in a functional setting. *J. Funct. Program.*, 9(4):471–477, 1999. `doi:10.1017/s0956796899003494`.

22 Mike Paterson. Progress in selection. In Rolf G. Karlsson and Andrzej Lingas, editors, *Algorithm Theory - SWAT '96, 5th Scandinavian Workshop on Algorithm Theory, Reykjavík, Iceland, July 3-5, 1996, Proceedings*, volume 1097 of *Lecture Notes in Computer Science*, pages 368–379. Springer, 1996. `doi:10.1007/3-540-61422-2_146`.

23 Abhiroop Sarkar. Persistent red black trees in Haskell [online]. 2017. URL: `https://abhiroop.github.io/Haskell-Red-Black-Tree/`.

24 Jukka Suomela. Median filtering is equivalent to sorting. *CoRR*, abs/1406.1717, 2014. `arXiv:1406.1717`.

25 Kanat Tangwongsan, Martin Hirzel, and Scott Schneider. Low-latency sliding-window aggregation in worst-case constant time. In *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems, DEBS 2017, Barcelona, Spain, June 19-23, 2017*, pages 66–77. ACM, 2017. `doi:10.1145/3093742.3093925`.

26 Jun Yang and Jennifer Widom. Incremental computation and maintenance of temporal aggregates. *VLDB J.*, 12(3):262–283, 2003. `doi:10.1007/s00778-003-0107-z`.

27 Andrew Chi-Chih Yao. Space-time tradeoff for answering range queries (extended abstract). In Harry R. Lewis, Barbara B. Simons, Walter A. Burkhard, and Lawrence H. Landweber, editors, *Proceedings of the 14th Annual ACM Symposium on Theory of Computing, May 5-7, 1982, San Francisco, California, USA*, pages 128–136. ACM, 1982. `doi:10.1145/800070.802185`.

A Finding the k-smallest Element in Three AWBBS Trees

We assume that each node of the input trees is *augmented* with its weight, i.e., contains the size of its subtree. Then, searching for the k-th smallest element in such a tree is straightforward: one traverses the tree from the root down to the leaves and at each step chooses the left or the right subtree depending on whether the k-th smallest element can be there. The task of finding the k-th smallest element in multiple trees is similar: we simultaneously traverse all trees from their roots down to the leaves while choosing in which subtrees the k-th smallest element still can be. One needs to carefully analyze the traversal condition to make sure that the algorithm works in linear time in terms of the tree heights, since otherwise the incurred costs can lead to higher complexity (e.g., for two trees it can become quadratic [16]).

The algorithm that we present below can be seen as the fixed version of the algorithm from Lemma 2 of [17], where the condition 2 is corrected to continue the search instead of stopping. (Without this correction the algorithm of [17] won't work.)

A.1 The Algorithm

We assume that there exists a function find-rank-one-tree which outputs the k-th smallest element in one tree in time $O(h)$ where h is the height of the tree. At each step of our algorithm we maintain the remaining trees T_1, T_2 and T_3 where we continue to search for the k-th smallest element. Wlog, we assume that the root elements of the trees are ordered, i.e., the root r_1 of T_1 is smaller than the root r_2 of T_2 which is smaller than the root r_3 of T_3. Let S denote the elements of the left subtrees L_1, L_2 and L_3. By construction, we know that the rank of r_1 in the union of T_1, T_2 and T_3 is at most $|S| + 1$, whereas the rank of r_3 in the union of T_1, T_2 and T_3 is at least $|S| + 3$. These inequalities allow us to identify the subtree where the element of rank k *cannot* be: if $|S| + 3 > k$, then the k-th smallest element cannot be r_3 and cannot be in R_3; if $|S| + 3 \leq k$, then r_1 and elements in L_1 are all smaller than the k-th smallest element and hence can be discarded. We continue this traversal until one of the trees becomes empty, then we apply the same reasoning for the two remaining trees only. Finally, once we are left with a single tree we use find-rank-one-tree to output the k-th smallest element in the remaining tree.

Algorithm 3 find-rank-three-trees finds the k-smallest element in three trees.

Input: trees T_1, T_2, T_3 and the target rank k.
Output: k-th smallest element in T_1, T_2 and T_3.

```
1:  target-rank := k
2:  while True do
3:      if all but one tree are empty then
4:          return target-rank-th smallest element in the non-empty tree by calling find-rank-one-tree.
5:      end if
6:      Let $T_1, T_2, \ldots, T_\ell$ denote the non-empty trees left
7:      Let $L_j, R_j, r_j$ be the left subtree, the right subtree, and the root of $T_j$, respectively
8:      Wlog, assume $r_1 < r_2 < \cdots < r_\ell$ (otherwise, rename the trees)
9:      if $\sum_{j=1}^{\ell} \mathsf{size}(L_j) + \ell >$ target-rank then
10:         $T_\ell := L_\ell$
11:     else
12:         $T_1 := R_1$
13:         target-rank := target-rank $- \mathsf{size}(L_1) - 1$
14:     end if
15: end while
```

▶ **Lemma 9.** *Algorithm 3 finds the k-th smallest element in T_1, T_2 and T_3 in $O(h_1 + h_2 + h_3)$ time where h_1, h_2 and h_3 are the tree heights of T_1, T_2 and T_3, respectively.*

Proof.

Correctness. Let T_j^i denote the state of the trees at the beginning of the while loop at step 2 at iteration i. That is, T_1^0, T_2^0 and T_3^0 represent the initial input trees, T_1^1, T_2^1 and T_3^1 represent the trees after one iteration, and so on. Similarly, let *target-rank*^{i} denote the state of the target rank variable at the beginning of the i-th iteration. We prove correctness by validating the following invariant:

▷ Claim 10. The *target-rank*^{i}-th smallest element among T_1^i, T_2^i and T_3^i is the *target-rank*$^{i+1}$-th smallest element among T_1^{i+1}, T_2^{i+1} and T_3^{i+1}.

Proof. Let L_j^i, R_j^i, r_j^i denote the corresponding loop variables during the iteration i. Consider two cases:

- $\sum_{j=1}^{\ell} \mathsf{size}(L_j^i) + \ell > \textit{target-rank}^i$: Because the rank of r_ℓ^i is at least $\sum_{j=1}^{\ell} \mathsf{size}(L_j^i) + \ell$, and $\ell \geq 2$ the element with rank *target-rank*^{i} must be smaller than r_ℓ^i and all the elements in R_ℓ^i. Hence, we can continue searching for the element with rank *target-rank*^{i} in $T_1, \ldots, T_{\ell-1}$ and the left subtree of T_ℓ which is L_ℓ. This is implemented in line 10.
- $\sum_{j=1}^{\ell} \mathsf{size}(L_j^i) + \ell \leq \textit{target-rank}^i$: Because the rank of r_1^i is at most $\sum_{j=1}^{\ell} \mathsf{size}(L_j^i) + 1$, and $\ell \geq 2$ the element with rank *target-rank*^{i} must be greater than r_1^i and all the elements in L_1^i. Hence, we can continue searching for the element with rank $\textit{target-rank}^{i+1} = \textit{target-rank}^i - \mathsf{size}(L_1^i) - 1$ in $T_2, T_3, \ldots, T_\ell$ and the right subtree of T_1 which is R_1. This is implemented in lines 12 and 13. ◁

As long as more than two trees $T_1^i, T_2^i, \ldots T_\ell^i$ are non-empty we search for the right element in them. Once all but one tree are empty we will output the element with the correct rank because find-rank-one-tree is correct.

Complexity analysis. Every time the while loop (lines 2 to 15) is executed either T_1's, T_2's, or T_3's height is decreased by 1. Hence, the maximum number of loop executions is $h_1 + h_2 + h_3$. Furthermore, whenever only one of the trees is non empty the loop returns the output of the find-rank-one-tree call. One invocation of find-rank-one-tree takes time $O(h)$ where h is the height of the non-empty tree during the invocation. By construction h is smaller than h_1, h_2 and h_3. This means that the overall time complexity is $O(h_1 + h_2 + h_3)$. ◀

Instantiating Algorithm 5 in the setting where the binary search trees are weight-balanced we obtain the desired result.

▶ **Theorem 11.** *Algorithm 5 finds the k-th smallest element in $O(\log n)$ time where T_1, T_2 and T_3 are AWBBS trees of size at most n.*

Proof. The theorem follows by combining the result of Lemma 9 and the observation that the height of T_1, T_2 and T_3 is $O(\log n)$. ◀

Space-Query Tradeoffs in Range Subgraph Counting and Listing

Shiyuan Deng ✉
The Chinese University of Hong Kong, China

Shangqi Lu ✉
The Chinese University of Hong Kong, China

Yufei Tao ✉
The Chinese University of Hong Kong, China

Abstract

This paper initializes the study of *range subgraph counting* and *range subgraph listing*, both of which are motivated by the significant demands in practice to perform graph analytics on subgraphs pertinent to only selected, as opposed to all, vertices. In the first problem, there is an undirected graph G where each vertex carries a real-valued attribute. Given an interval q and a pattern Q, a query counts the number of occurrences of Q in the subgraph of G induced by the vertices whose attributes fall in q. The second problem has the same setup except that a query needs to enumerate (rather than count) those occurrences with a small delay. In both problems, our goal is to understand the tradeoff between *space usage* and *query cost*, or more specifically: (i) given a target on query efficiency, how much pre-computed information about G must we store? (ii) Or conversely, given a budget on space usage, what is the best query time we can hope for? We establish a suite of upper- and lower-bound results on such tradeoffs for various query patterns.

2012 ACM Subject Classification Theory of computation → Graph algorithms analysis

Keywords and phrases Subgraph Pattern Counting, Subgraph Pattern Listing, Conjunctive Queries

Digital Object Identifier 10.4230/LIPIcs.ICDT.2023.6

Related Version *Full Version*: `https://arxiv.org/abs/2301.03390`

Funding Supported in part by GRF Projects 14207820, 14203421, and 14222822 from HKRGC.

1 Introduction

Consider $G = (V, E)$ as a *data graph* and Q as a *pattern graph*. A subgraph of G, if isomorphic to Q, is said to be an *occurrence* of Q. The goal of *pattern searching* is to either list the occurrences of Q or to count the number of them. Both are fundamental problems in computer science and have attracted considerable attention in the past few decades.

This paper studies pattern searching in vertex-induced subgraphs. Here, a query selects a subset $U \subseteq V$ of vertices and needs to count/list the occurrences of Q in G', where G' is the subgraph of G induced by U. Note that if an occurrence uses any vertex outside U, the occurrence should not be counted/listed. Trivially, one can answer the query by first generating G' and then counting/listing Q in G' "from scratch", but this does not leverage the power of *preprocessing*. Instead, our goal is to store G in a data structure that can answer all queries with non-trivial guarantees. It is intriguing to investigate how much we can minimize the query time subject to a space budget, and conversely, how much space we must consume to achieve a target query time.

Vertex selection in database systems is done with a predicate q, which determines U as $\{v \in V \mid v \text{ satisfies } q\}$. Concentrating on *range predicates*, the problems we consider are:

Problem 1 (Range Subgraph Counting). $G = (V, E)$ is an undirected graph where each vertex $v \in V$ carries a real-valued *attribute* A_v. For an interval $q = [x_1, x_2]$, define $V_q = \{v \in V \mid x_1 \leq A_v \leq x_2\}$ and G_q as the subgraph of G induced by V_q. Let Q be a connected (only one connected component) pattern graph with $O(1)$ vertices. Given an interval q, a query returns the number of occurrences of Q in G_q. The pattern Q is fixed for all queries.

Problem 2 (Range Subgraph Listing). Same setup except that a query reports the occurrences of Q in G_q.

Universal Notations. Several notations will apply throughout the paper. Set $n = |V|$ and $m = |E|$. Symbol $\omega < 2.37286$ [1] represents the matrix multiplication exponent. The notations $\tilde{O}(.)$ and $\tilde{\Omega}(.)$ hide a factor polylogarithmic to the underlying problem's parameters.

1.1 Motivation

Practical Applications. Subgraph patterns are important for understanding the characteristics of a data graph G, as has been documented in a long string of papers, e.g., [2, 3, 8, 10, 11, 17, 18, 24–28, 30, 33, 36–38, 50]. In practice, analysts are interested in not only patterns from the whole G but also those pertinent only to selected vertices. Consider a social network G where each vertex represents an individual. A graph's *clustering coefficient* [49], a popular measurement in network science, is the ratio between the number of triangles (3-cliques[1]) and the number of wedges (2-paths[2]). The coefficient of G, however, is just a single value revealing little about the features of specific demographic groups. It is more informative to, for example, compare the coefficients of (i) the subgraph of G induced by people with ages $\in [20, 30]$, and (ii) that induced by age $\in [60, 70]$. A step further, by putting together the coefficients induced by "age $\in [i \cdot 10, (i + 1) \cdot 10]$" for each $i \in [1, 10]$, one obtains an interesting comparison across different age groups. Refined analysis can then concentrate on the pattern occurrences of a target group. The power of the above analysis owes to queries of Problem 1 and 2 with *arbitrary* selection ranges. Designing effective data structures is essential to avoid lengthy response time.

Importance of Space-Query Tradeoffs. One should not confuse the space-query tradeoff with the tradeoff between *preprocessing time* and *query cost*, as has been studied in the literature on join processing [5, 12, 20–23, 35, 41–43, 45]. Both tradeoffs are important, but they matter in different ways. Unlike preprocessing time, which is "one-time cost" (because a structure, once built, can be used forever), the space consumption is permanent. In other words, the space-query tradeoff has a (much) more durable effect on the underlying database system. However, in spite of their importance, the space-query tradeoffs on joins have received surprisingly little attention: we are aware of only a single paper [20], which, as will be discussed in Section 1.3, does not consider query predicates (or equivalently, only one query, which always outputs the entire join, exists) and concerns only reporting (but not counting). Our work can be thought of as a step in the same direction as [20] because, as explained in Section 5, subgraph searching can be cast as a join problem (in fact, some of our results are explicitly about joins), and actually the first step on predicate-driven queries and counting.

[1] An *ℓ-clique* is a clique with ℓ vertices.
[2] An *ℓ-path* is a path with ℓ edges.

Table 1 A summary of our results.

Problem	Pattern Q	Space	Query	Remark
1 (cnt)	any fixed Q	$O(n^2)$	$\tilde{O}(1)$	near optimal†
1	wedge	$\tilde{O}(m^2/\lambda^2)$	$\tilde{O}(\lambda)$	for any $\lambda \in [1, \sqrt{m}]$, near optimal†
1 (lower bound)	wedge	$\tilde{O}(m^{2-\delta}/\lambda^2)$ impossible	$\tilde{O}(\lambda)$	for $\lambda \in [1, \sqrt{m}]$ and any $\delta > 0$, subj. to strong set disjointness conj.
1	ℓ-clique	$O(m)$	$\tilde{O}(1)$	
2 (rep)	any fixed Q	$\tilde{O}(m + m^{\rho^*}/\Delta)$	delay $\tilde{O}(\Delta)$	for any $\Delta \geq 1$, ρ^* = frac. edge covering num. of Q
2	triangle	$O(m)$	delay $\tilde{O}(1 + (m^*)^{\frac{\omega-1}{\omega+1}})$	m^* = num. of edges in at least one triangle in G_q
2	ℓ-star	$O(m)$	delay $\tilde{O}(1)$	near optimal
2	2ℓ-cycle	$\tilde{O}(\#P_\ell)$	delay $\tilde{O}(1)$	$\#P_\ell$ = num. of ℓ-paths in G

Remark: "near optimal" means no polynomial improvement (i.e., n^δ for arbitrary small constant $\delta > 0$) possbile. The near optimality marked with † is subject to the strong set disjointness conjecture.

Finally, it is worth mentioning that a useful structure, no matter how little space it occupies, must be constructible in polynomial time. This is true for all the structures developed in our paper. In fact, each of our structures can be built with at most the time needed to find all the occurrences of the query pattern Q, ignoring polylog factors.

1.2 Our Contributions

Table 1 summarizes the main results of this paper. Next, we will explain the results in detail.

1.2.1 Problem 1

Wedges. We will show:

▶ **Theorem 1.** *Consider Problem 1 with Q = wedge. For any real value $\lambda \in [1, \sqrt{m}]$, there is a structure of $\tilde{O}(m^2/\lambda^2)$ space that answers a query in $\tilde{O}(\lambda)$ time.*

The space-query tradeoff may look disappointing. After all, wedge counting is easy in *one-off* computation: we can count the number of wedges in G using $O(n + m)$ time. It is natural to wonder whether the space in Theorem 1 is necessary. We answer the question by showing that any substantial improvement to Theorem 1 will yield a major breakthrough on *set disjointness*:

Set Disjointness. The data is a collection of $s \geq 2$ sets S_1, S_2, ..., S_s. Given distinct set ids $a, b \in [1, s]$, a query returns whether $S_a \cap S_b$ is empty.

Let $N = \sum_{i=1}^{s} |S_i|$ be the input size of set disjointness. Given any $\lambda \in [1, \sqrt{N}]$, there is a simple structure of $O(N^2/\lambda^2)$ space with $O(\lambda)$ query time (see Appendix B). Improving the tradeoff by a polynomial factor even for one arbitrary λ has been a long-standing open problem. The *strong set disjointness conjecture* [31, 32] states that a structure with query time λ must use $\tilde{\Omega}(N^2/\lambda^2)$ space for any $\lambda \geq 1$. We will prove:

▶ **Theorem 2.** *Consider Problem 1 with Q = wedge. Fix any $\lambda \in [1, \sqrt{m}]$ and any constant $\delta > 0$. Suppose that we can obtain a structure of $\tilde{O}(m^{2-\delta}/\lambda^2)$ space answering a query in $\tilde{O}(\lambda)$ time. Then, for any set disjointness input of size N, we can obtain a structure of $\tilde{O}(N^{2-\delta}/\lambda^2)$ space answering a query in $\tilde{O}(\lambda)$ time (thus breaking the strong set disjointness conjecture).*

Cliques. We will show:

▶ **Theorem 3.** *For Problem 1 with $Q = \ell$-clique, there is a structure of $O(m)$ space answering a query in $\tilde{O}(1)$ time.*

Counting triangles ($\ell = 3$) appears harder than counting wedges: in one-off computation, the fastest known algorithm for the former takes $O(m^{\frac{2\omega}{\omega+1}})$ time. It is thus surprising to see Q = triangle easier than Q = wedge in Problem 1. From Theorem 1 and 3, one sees that the problem of calculating the clustering coefficient (see Section 1.1) of G_q for any q boils down to counting the wedges in G_q. Effectively, this implies optimal settlement of that problem (subject to the strong set disjointness conjecture), which bears practical significance due to the popularity of clustering coefficients.

Arbitrary Subgraphs. We will show:

▶ **Theorem 4.** *For any Q, there is a structure for Problem 1 that uses $O(n^2)$ space and answers a query in $\tilde{O}(1)$ time.*

The above result is difficult to improve: reducing the space by an n^δ factor for any constant $\delta > 0$ breaks the strong set disjointness conjecture. To explain, assume $n = O(m)$.[3] If there was a structure of $O(n^{2-\delta}) = O(m^{2-\delta})$ space and $\tilde{O}(1)$ query time, applying the structure to Q = wedge would yield a breakthrough on set disjointness by way of Theorem 2. The reader should note that the hardness comes from producing a guarantee on all Q; it is possible to do better for special patterns (Theorem 3). The hardness thus endows Q = wedge with unique significance in Problem 1. Theorem 4 further implies that Problem 1 under Q = wedge is the hardest when G is the sparsest: $m = o(n^{1+\epsilon})$ for any constant $\epsilon > 0$. To see why, set $m = n^{1+\epsilon}$, which gives $n^2 = m^{\frac{2}{1+\epsilon}}$. Since $\frac{2}{1+\epsilon} = 2 - \frac{2\epsilon}{1+\epsilon}$, Theorem 4 yields a structure of $O(m^{2-\delta})$ space and $\tilde{O}(1)$ query time with $\delta = \frac{2\epsilon}{1+\epsilon}$, improving Theorem 1 by a polynomial factor at $\lambda = \tilde{O}(1)$.

1.2.2 Problem 2

A listing query ensures a *delay* Δ if it reports a new occurrence of Q or declares "no more occurrences" within Δ time after the previous occurrence[4].

Arbitrary Subgraphs. We will show:

▶ **Theorem 5.** *For any Q and $\Delta \geq 1$, there is a structure for Problem 2 that uses $\tilde{O}(m + m^{\rho^*}/\Delta)$ space and has a query delay of $\tilde{O}(\Delta)$, where ρ^* is the fractional edge covering number of Q.*

[3] Discard "isolated" vertices with no incident edges.
[4] The reader may assume that a dummy occurrence is always output at the beginning of a query algorithm.

Imagine assigning each edge of Q a non-negative weight such that (i) for each vertex of Q, all its incident edges receive a combined weight at least 1 and (ii) the total weight of all edges is minimized. The fractional edge covering number ρ^* of Q is the total weight of an optimal assignment. The maximum number of occurrences of Q in G is $O(m^{\rho^*})$ [4] and the bound is tight in the worst case.

Our structure actually settles a problem on natural joins:

> **Range Join.** Let $\mathcal{R}$ be a set of $O(1)$ relations each with $O(1)$ real-valued attributes. Denote by $\mathit{join}(\mathcal{R})$ the natural join result on the relations in $\mathcal{R}$. Given an interval $q = [x_1, x_2]$, a query reports all the tuples $t \in \mathit{join}(\mathcal{R})$ such that every attribute of t falls in q.

Let N be the total number of tuples in the relations of $\mathcal{R}$. For any $\Delta \geq 1$, we give a structure of $\tilde{O}(N + N^{\rho^*}/\Delta)$ space answering a query with an $\tilde{O}(\Delta)$ delay. Here, the fractional edge covering number ρ^* is with respect to the join's hypergraph (details deferred to Section 5).

The challenge behind Theorem 5 is to design a structure that works for all Q. It is possible to do better for specific Q. Next, we present three examples that are not only important subproblems themselves but also illustrate different techniques.

Triangles. We will show:

▶ **Theorem 6.** *For Problem 2 with Q = triangle, there is a structure of $O(m)$ space answering a query with an $\tilde{O}(1 + (m^*)^{\frac{\omega-1}{\omega+1}})$ delay, where m^* is the number of edges appearing in at least one reported triangle.*

The fractional edge covering number ρ^* is 1.5 for Q = triangle. To ensure $\tilde{O}(m)$ space, Theorem 5 needs to set $\Delta = \sqrt{m}$. As $\frac{\omega-1}{\omega+1} < 0.408$, Theorem 6 achieves a polynomial improvement in delay. The reader should note that the value m^* in Theorem 6 never exceeds m but can be much less (this happens when there are few triangles to list). When q is fixed to $(-\infty, \infty)$, Problem 2 with Q = triangle had been used as a motivating problem in the previous work of [20], which described a structure of $O(m)$ space with a delay $\tilde{O}(\sqrt{m})$ and is thus strictly improved by Theorem 6.

ℓ-Stars. An *ℓ-star* is a tree with ℓ leaves and one non-leaf vertex (a wedge is a 2-star). We will show:

▶ **Theorem 7.** *For Problem 2 where Q = ℓ-star, there is a structure of $O(m)$ space answering a query with an $\tilde{O}(1)$ delay.*

As a corollary, for any interval q, $O(m)$ space suffices to *detect* the presence of an ℓ-star in G_q using $\tilde{O}(1)$ time. For Q = wedge, this means that the hardness manifested by Theorem 2 is indeed due to *counting*.

2ℓ-Cycles[5]**.** We will show:

▶ **Theorem 8.** *For Problem 2 with Q = 2ℓ-cycle where $\ell \geq 2$, there is a structure of $\tilde{O}(\#P_\ell)$ space answering a query with an $\tilde{O}(1)$ delay, where $\#P_\ell$ is the number of ℓ-paths in G.*

[5] A cycle with 2ℓ vertices.

The fractional edge covering number ρ^* is ℓ for a 2ℓ-cycle. Theorem 5 needs $\tilde{O}(m^\ell)$ space to achieve an $\tilde{O}(1)$ delay. The space in Theorem 8 is significantly better. For $\ell = 2$ ($Q =$ 4-cycle), the space is $\tilde{O}(nm)$ which is the maximum number of wedges in G. For $\ell > 2$, the space is $\tilde{O}(m^{\lceil(\ell+1)/2\rceil})$ which is the maximum number of ℓ-paths in G.

1.3 Related Work

The preceding sections have covered the most relevant existing results. We will now proceed to discuss other related work.

Pattern searching has been extensively studied in one-off computation. We refer the reader to [3, 8, 10, 17, 18, 27, 28, 30, 37, 50] and [2, 11, 17, 24–26, 33, 36, 38, 39], as well as the references therein, for algorithms on counting and listing, respectively. Those algorithms can be applied in Problem 1 and 2 after G_q has been generated. Our focus in this work is to avoid a full generation of G_q because doing so can take $\Omega(m)$ time.

In the other extreme, one can precompute the set S of occurrences of Q in G. The size of S is $O(m^{\rho^*})$ (AGM bound), assuming that Q has a constant size. By resorting to standard computational geometry techniques [19], we can store S in structures of $\tilde{O}(m^{\rho^*})$ space to answer a query of Problem 1 in $\tilde{O}(1)$ time and a query of Problem 2 with an $\tilde{O}(1)$ delay. For Problem 1, Theorem 4 achieves a better space bound on every Q with $\rho^* \geq 2$. When $\rho^* < 2$, Q has at most three vertices: a 1-path (single edge), a wedge, or a triangle. We have resolved the wedge and triangle cases (Theorem 1 and 3), while Problem 1 is trivial for $Q =$ 1-path. For Problem 2, Theorem 5 captures the above extreme idea as a special case with $\Delta = \tilde{O}(1)$ and offers a tunable space-query tradeoff.

A *relational event graph*, introduced by Bannister et al. [6], is a graph $G = (V, E)$ where every *edge* $e \in E$ carries a real-valued timestamp. For an interval $q = [x_1, x_2]$, let G_q^{edge} be the subgraph of G induced by all the edges whose timestamps are covered by q. A pattern searching query counts/lists the occurrences of a pattern Q in G_q^{edge}. See [6, 14, 15] for several data structures designed for such queries. Similar as it sounds, pattern searching on a relational event graph is drastically different from Problem 1 and 2 such that there is little overlap – in neither results nor techniques – between our solutions and those in [6, 14, 15].

Delay minimization is an important topic in the literature of joins and conjunctive queries; see [5, 9, 12, 13, 20–23, 34, 35, 41, 43–45] and their references. Regarding our problems, we are not aware of previous work giving a result better than what has already been mentioned. Our formulation of range join listing (Section 1.2.2) suggests that the presence of query predicates can pose new challenges on joins (also conjunctive queries) from the indexing's perspective. Deep and Koutris [20] proved a result equivalent to Theorem 5 (up to an $\tilde{O}(1)$ factor) on Problem 2, but only in the special scenario where a query concerns the whole G, i.e., fixing the query range q to $(-\infty, \infty)$.

2 Preliminaries

In this section, we will describe several technical tools to be deployed in our solutions.

Structures for Multidimensional Points. We will utilize some well-known geometry data structures as introduced below. The reader does not need to be bothered with the details of these structures because we will apply them as "black boxes". Let P be a set of n points in d-dimensional space $\mathbb{R}^d$ where d is a constant. Given a rectangle q of the form $[x_1, y_1] \times [x_2, y_2] \times ... \times [x_d, y_d]$, a *range reporting* query enumerates the points in $P \cap q$. We can create a *range tree* [7, 19] on P, which uses $\tilde{O}(n)$ space and permits us to answer such a query with an $\tilde{O}(1)$ delay. When $d = 2$, we can replace the range tree with a *Chazelle's structure* [16] which retains the aforementioned query performance but reduces the space consumption to $O(n)$.

We will also need *range sum* queries on P in the scenario where each point in P is 2D (i.e., $d = 2$) and carries a real-valued *weight*. Given a rectangle $q = [x_1, y_1] \times [x_2, y_2]$, such a query reports the total weight of the points in $P \cap q$. We can again build a Chazelle's structure of [16] on P which occupies $O(n)$ space and answers a query in $\tilde{O}(1)$ time.

From "Delays with Duplicates" to "Delays under Distinctness". Let us consider a duplicate-removal scenario often encountered in designing algorithms with small delays. Suppose that we have an algorithm $\mathcal{A}$ for enumerating a set S of elements. With a delay of Δ, $\mathcal{A}$ can report an element $e \in S$, but cannot guarantee that e has never been reported before. The good news, on the other hand, is that $\mathcal{A}$ can output the same element at most α times for some $\alpha \geq 1$.

By modifying a *buffering technique* in [47], we can convert $\mathcal{A}$ into an algorithm that enumerates only the *distinct* elements of S with a delay of $O(\alpha \cdot \Delta \log |S|)$. Conceptually, divide the execution of $\mathcal{A}$ into *epochs*, each of which runs for $\alpha \cdot \Delta$ time[6]. As $\mathcal{A}$ runs, we use a *buffer* B to stash the set of distinct elements that have been found by $\mathcal{A}$ but not yet reported. Every time $\mathcal{A}$ finds an element $e \in S$, we check whether e has ever existed in B (this takes $O(\log |S|)$ time, using a binary search tree on all the elements that have ever been found so far). If so, e is ignored; otherwise, it is added to B. At the end of each epoch, we output an arbitrary element from B and remove it from B. Finally, after $\mathcal{A}$ has terminated, we simply output the remaining elements in B.

B always contains at least one element at the end of each epoch. To see why, consider the end of the t-th epoch for some $t \geq 1$. At this moment, $\mathcal{A}$ has been running for $t \cdot \alpha \cdot \Delta$ time and therefore must have reported $t \cdot \alpha$ elements, which may not be distinct. However, as each element can be reported at most α times, there must be at least t (distinct) ones among those $t \cdot \alpha$ elements. Since we have reported only $t - 1$ elements in the preceding epochs, B must still have at least one element at the end of epoch t. It is now straightforward to verify that the modified algorithm has a delay of $O(\alpha \cdot \Delta \log |S|)$ in enumerating the distinct elements of S.

3 Problem 1: Matching Upper and Lower Bounds

This section will establish the conditional lower bound in Theorem 2 and its matching upper bound in Theorem 4. Our discussion on the upper bound will also establish Theorem 3. Throughout the paper, we will assume that the vertices of G have distinct attribute values. The assumption loses no generality because one can break ties by vertex id.

3.1 Lower Bound

Suppose that Problem 1 under Q = wedge admits a structure that uses $\tilde{O}(m^{2-\delta}/\lambda^2)$ space and answers a query in $\tilde{O}(\lambda)$ time for some $\lambda \geq 1$. We will design a structure for set disjointness that uses $\tilde{O}(N^{2-\delta}/\lambda^2)$ space and answers a query in $\tilde{O}(\lambda)$ time. Recall that the data input to set disjointness consists of $s \geq 2$ sets $S_1, ..., S_s$ with a total size of N. Define $\mathcal{U} = \bigcup_{i=1}^{s} S_i$.

[6] Recall that "time" in the RAM model is defined as the number of atomic operations (e.g., addition, multiplication, comparison, accessing a memory word, etc.) executed. Each epoch is essentially a sequence of $\alpha \cdot \Delta$ such operations.

Create a graph $G = (V, E)$ as follows. V has $2s + |\mathcal{U}|$ vertices, including $2s$ *set vertices* and $|\mathcal{U}|$ *element vertices*. Each set S_i ($i \in [1, s]$) defines two set vertices, whose attribute values are set to i and $s + i$, respectively. Each element in $\mathcal{U}$ defines an element vertex with the same attribute value $s + 1/2$. Set E contains $2N$ edges: for each element $e \in S_i$, add to E two edges each between the element vertex of e and a set vertex of S_i. Now, create a Problem-1 structure under Q = wedge on G. The structure occupies $\tilde{O}(N^{2-\delta}/\lambda^2)$ space.

Consider a set disjointness query with set ids a and b. Assuming w.l.o.g. $a < b$, we issue four Problem-1 wedge-counting queries on G with intervals $q_1 = [a, s + b]$, $q_2 = [a + 1, s + b]$, $q_3 = [a, s + b - 1]$, and $q_4 = [a + 1, s + b - 1]$, respectively. Let $c_1, c_2, ..., c_4$ be the counts returned. We declare $S_a \cap S_b$ non-empty if and only if $c_1 - c_2 - c_3 + c_4 > 0$. The query time is $\tilde{O}(\lambda)$. Appendix A proves the algorithm's correctness. This completes the proof of Theorem 2.

3.2 Upper Bound

Next, we will attack Problem 1 by allowing Q to be an arbitrary pattern graph. Consider any occurrence of Q in G. Let u (resp. v) be the vertex in this occurrence with the smallest (resp. largest) attribute. We *register* the occurrence at the pair (u, v). Denote by $c_{u,v}$ the number of occurrences registered at (u, v).

For a query with $q = [x_1, x_2]$, an occurrence registered at (u, v) appears in G_q (i.e., the subgraph of G induced by V_q) if and only if $A_u \geq x_1$ and $A_v \leq x_2$. We can therefore convert the problem to *range sum* on 2D points. For each pair $(u, v) \in V \times V$, create a point (A_u, A_v) with weight $c_{u,v}$. Let P be the set of points created; clearly, $|P| = O(n^2)$. The query result is simply the total weight of all the points in P covered by the rectangle $[x_1, \infty) \times (-\infty, x_2]$ (a range sum operation). We can store P in a Chazelle's structure (see Section 2) that occupies $O(|P|) = O(n^2)$ space and performs a range sum operation in $\tilde{O}(1)$ time. This establishes Theorem 4.

Improvement for Cliques. The space of our structure can be lowered to $O(m)$ when Q is a clique. The crucial observation is that registering an occurrence at (u, v) implies $\{u, v\} \in E$. We add to P only the points (A_u, A_v) with a non-zero $c_{u,v}$ (points with zero weights do not affect a range sum operation). This reduces the size of P to at most m and, hence, the space of the Chazelle's structure to $O(m)$. We thus complete the proof of Theorem 3.

4 Problem 1: Wedges

The section will explain how to achieve the guarantees in Theorem 1 for Problem 1 under Q = wedge. We will represent a wedge occurrence in $G = (V, E)$ as *wedge*(u, v, w) where u, v, and w are vertices in V, and $\{u, v\}$ and $\{v, w\}$ are edges in E. Let us introduce a slightly different problem:

> **Colored Range Wedge Counting.** Define $G = (V, E)$ and A_v for each $v \in V$ as in Problem 1. Each vertex in V is colored black or white. Given an interval q, a query returns the number of occurrences *wedge*(u, v, w) such that $A_u \in q$, $A_w \in q$, and v is black.

Note that no requirements exist on A_v and the colors of u and w.

Let $\mathcal{C}$ be a set of subsets of V. We call $\mathcal{C}$ a *canonical collection* if

- (P4-1) each vertex of V appears in $\tilde{O}(1)$ subsets in $\mathcal{C}$;
- (P4-2) for any interval q, we can partition V_q (i.e., the set of vertices in V with attribute values in q) into $\tilde{O}(1)$ disjoint subsets, each being a member of $\mathcal{C}$. The ids of these subsets can be obtained in $\tilde{O}(1)$ time.

It is rudimentary to find a canonical collection $\mathcal{C}$ satisfying $\sum_{U\in\mathcal{C}} |U| = \tilde{O}(n)$.[7] We will work with such a $\mathcal{C}$ henceforth. In Appendix B, we prove:

▶ **Lemma 9.** *Consider the colored range wedge counting problem. For any real value $\lambda \in [1, \sqrt{m}]$, there is a structure of $\tilde{O}(m^2/\lambda^2)$ space that answers a query in $\tilde{O}(\lambda)$ time.*

Equipped with the above, we now return to Problem 1 with Q = wedge.

Structure. For each $U \in \mathcal{C}$ (where U is a subset of V), we create a graph G_U by adding edges in three steps:

1. Initialize G_U as an empty graph with no vertices and edges.
2. For every vertex $u \in U$, we add all its edges in G (i.e., the original data graph) to G_U. The addition of an edge $\{u, v\}$ creates vertex v in G_U if v is not present in G_U yet.
3. Finally, color a vertex in G_U black if it comes from U, or white otherwise.

We now build a structure of Lemma 9 on G_U, which uses $\tilde{O}(|E_U|^2/\lambda^2)$ space where E_U is the set of edges in G_U. By Property P4-1, each edge $\{u, v\}$ of G can be added to the E_U of $\tilde{O}(1)$ subsets $U \in \mathcal{C}$. It thus follows that $\sum_{U\in\mathcal{C}} |E_U| = \tilde{O}(m)$. The structures of all $U \in \mathcal{C}$ occupy $\tilde{O}(m^2/\lambda^2)$ space in total.

Query. Consider now a (Problem-1) query with interval q. By Property P4-2, in $\tilde{O}(1)$ time we can pick $h = \tilde{O}(1)$ members $U_1, ..., U_h$ from $\mathcal{C}$ to partition V_q. For each $i \in [1, h]$, issue a colored range wedge counting query with interval q on G_{U_i}. We return the sum of the h queries' outputs. The overall query time is $h \cdot \tilde{O}(\lambda) = \tilde{O}(\lambda)$.

To verify correctness, first observe that every *wedge*(u, v, w) counted by the colored query on G_{U_i} satisfies: $A_u \in q$, $A_w \in q$ (definition of colored range wedge counting), and $A_v \in q$ (because v being black means $v \in U_i \subseteq V_q$). Conversely, every occurrence *wedge*(u, v, w) satisfying $\{A_u, A_v, A_w\} \subseteq q$ is counted only once: by the colored query on G_{U_i} where U_i is the only subset (among all $i \in [1, h]$) containing v. Indeed, for any U_j with $j \neq i$, v is either absent in G_{U_j} or is white; in neither case can the wedge be counted. Correctness now follows.

5 Problem 2: Arbitrary Subgraphs

We now proceed to tackle Problem 2 for an arbitrary query pattern Q. We will, in fact, solve the range join problem defined in Section 1.2.2. As shown in Appendix D, it is relatively easy to convert our structure to prove Theorem 5.

For a relation $R \in \mathcal{R}$ (recall that $\mathcal{R}$ is the set of input relations; see Section 1.2.2) its scheme, *scheme*(R), is the set of attributes in R. Let $\mathcal{X} = \bigcup_{R\in\mathcal{R}}$ *scheme*(R). The input size N can now be expressed as $\sum_{R\in\mathcal{R}} |R|$. We will assume, w.l.o.g., that (i) the relations in $\mathcal{R}$ have distinct schemes, (ii) N is a power of 2, and (iii) each attribute $X \in \mathcal{X}$ has a domain *dom*(X) comprising the integers in $[1, N]$. Given an interval $q = [x_1, x_2]$, a query lists every t in *join*$(\mathcal{R})$ – the natural join result on $\mathcal{R}$ – satisfying $t[X] \in q$ for all $X \in \mathcal{X}$, where $t[X]$ is the tuple's value under attribute X. We want to design a structure of small space to answer such queries with a small delay.

[7] It suffices to build a binary search tree T on the vertices' attribute values. Each node in T defines a subset in $\mathcal{C}$, which consists of every $v \in V$ whose attribute A_v is stored in the node's subtree. It is well known (see, e.g., [46]) that, for any interval q, there exist $O(\log n)$ *canonical nodes* in T whose subtrees are disjoint and together contain all and only the attribute values in q. Those nodes can be found in $O(\log n)$ time and satisfy Property P4-2 with respect to V_q.

It will be convenient to work with a hypergraph $\mathcal{G} = (\mathcal{X}, \mathcal{E})$ where $\mathcal{E} = \{scheme(R) \mid R \in \mathcal{R}\}$. Given an edge $e \in \mathcal{E}$, we use R_e to denote the (only) relation $R \in \mathcal{R}$ whose scheme is e. For a function W that assigns a non-negative weight $W(e)$ to every $e \in \mathcal{E}$, its *lump-sum* is $\sum_{e \in \mathcal{E}} W(e)$. The function W is a *fractional edge covering* if $\sum_{e \in \mathcal{E}: X \in e} W(e) \geq 1$ holds on every attribute $X \in \mathcal{X}$. The *fractional edge covering number* ρ^* of $\mathcal{G}$ is the smallest lump-sum of all fractional edge coverings. Henceforth, we will use W to represent an optimal assignment function with lump-sum ρ^*.

The section's main result is:

▶ **Theorem 10.** *For the range join problem (see Section 1.2.2), given any $\Delta \geq 1$, there is a structure of $\tilde{O}(N + N^{\rho^*}/\Delta)$ space that answers a query with an $\tilde{O}(\Delta)$ delay.*

5.1 A Generalization of the AGM Bound

The classical AGM bound [4] states that $|join(\mathcal{R})| \leq \prod_{e \in \mathcal{E}} |R_e|^{W(e)}$. Next, we will present a more general version of this inequality.

Set $d = |\mathcal{X}|$ and impose an arbitrary ordering on the d attributes: $X_1, X_2, ..., X_d$. Given intervals $I_1, I_2, ..., I_d$ where $I_i \subseteq dom(X_i)$ for each $i \in [1, d]$, define $B(I_1, ..., I_d)$ as the d-dimensional box $I_1 \times ... \times I_d$. For a relation $R \in \mathcal{R}$, we use $R \ltimes B(I_1, ..., I_d)$ to represent the set of tuples $t \in R$ such that $t[X_i] \in I_i$ for every i satisfying $X_i \in scheme(R)$.

We prove in Appendix C:

▶ **Lemma 11.** *Let $\boldsymbol{\mathcal{I}}_i$, $i \in [1, d]$, be a set of disjoint intervals in $dom(X_i)$. Then:*

$$\sum_{I_1 \in \boldsymbol{\mathcal{I}}_1} \sum_{I_2 \in \boldsymbol{\mathcal{I}}_2} ... \sum_{I_d \in \boldsymbol{\mathcal{I}}_d} \prod_{e \in \mathcal{E}} |R_e \ltimes B(I_1, ..., I_d)|^{W(e)} \leq \prod_{e \in \mathcal{E}} |R_e|^{W(e)}. \tag{1}$$

To see how (1) captures the AGM bound, consider the special $\boldsymbol{\mathcal{I}}_i$ with size $|dom(X_i)|$, namely, each interval in $\boldsymbol{\mathcal{I}}_i$ is a value in $dom(X_i)$ and vice versa. Thus, $|R_e \ltimes B(I_1, ..., I_d)|$ is either 0 or 1 such that the left hand side of (1) is precisely $|join(\mathcal{R})|$. The real power of (1), however, comes from allowing $\boldsymbol{\mathcal{I}}_i$ to be an arbitrary set of disjoint intervals, a feature crucial for us to prove Theorem 10.

A remark is in order about why Lemma 11 is not trivial. It would be if the term $\prod_{e \in \mathcal{E}} |R_e \ltimes B(I_1, ..., I_d)|^{W(e)}$ in (1) was replaced by the output size of the join on the relations in $\{R_e \ltimes B(I_1, ..., I_d) \mid e \in \mathcal{E}\}$. By the AGM bound, the term $\prod_{e \in \mathcal{E}} |R_e \ltimes B(I_1, ..., I_d)|^{W(e)}$ is an *upper bound* on the size of the join $\{R_e \ltimes B(I_1, ..., I_d) \mid e \in \mathcal{E}\}$. The non-trivial goal is to show that the summation of all those *upper* bounds (i.e., the left hand side of (1)) still cannot exceed $\prod_{e \in \mathcal{E}} |R_e|^{W(e)}$.

5.2 Range Join

This subsection serves as a proof of Theorem 10. Given an $\ell \geq 0$, we call an interval a *level-ℓ dyadic interval* if it has the form $[i \cdot 2^\ell + 1,\ (i+1) \cdot 2^\ell]$ for some integer $i \geq 0$. Because N is a power of 2, for each $\ell \in [0, \log_2 N]$, we can partition $[1, N]$ into $N/2^\ell$ disjoint level-ℓ dyadic intervals.

A *dyadic combination* is a sequence of d dyadic intervals $(I_1, ..., I_d)$; recall that $d = |\mathcal{X}|$. The combination defines a (natural) join instance on the relations in $\{R_e \ltimes B(I_1, ..., I_d) \mid e \in \mathcal{E}\}$. We will denote the instance as $\mathcal{R}_{I_1, ..., I_d}$. Define

$$\text{AGM}(I_1, ..., I_d) \quad = \quad \prod_{e \in \mathcal{E}} |R_e \ltimes B(I_1, ..., I_d)|^{W(e)}. \tag{2}$$

The AGM bound assures us that $|join(\mathcal{R}_{I_1, ..., I_d})| \leq \text{AGM}(I_1, ..., I_d)$.

Structure. A dyadic combination $(I_1, ..., I_d)$ with a non-empty $join(\mathcal{R}_{I_1,...,I_d})$ is said to be *heavy* if $\text{AGM}(I_1, ..., I_d) > \Delta$, or *light* otherwise. For each heavy combination, we build a structure of [20] that can enumerate the tuples in $join(\mathcal{R}_{I_1,...,I_d})$ with an $\tilde{O}(\Delta)$ delay. The structure's space is bounded by $O(\text{AGM}(I_1, ..., I_d)/\Delta)$.[8]

We argue that the structures on all the heavy (dyadic) combinations use $\tilde{O}(N^{\rho^*}/\Delta)$ space in total. Fix d arbitrary level numbers $\ell_1, ..., \ell_d$ each between 0 and $\log_2 N$. For $i \in [1, d]$, let $\boldsymbol{\mathcal{I}}_i$ be the set of all level-ℓ_i dyadic intervals. The total space occupied by the structures of all heavy combinations $(I_1, ..., I_d) \in \boldsymbol{\mathcal{I}}_1 \times ... \times \boldsymbol{\mathcal{I}}_d$ is

$$\frac{1}{\Delta} \sum_{(I_1,...,I_d) \in \boldsymbol{\mathcal{I}}_1 \times ... \times \boldsymbol{\mathcal{I}}_d} \text{AGM}(I_1, ..., I_d). \tag{3}$$

up to an $\tilde{O}(1)$ factor. The above includes a term for every light combination but such terms can only over-estimate the space. Each $\boldsymbol{\mathcal{I}}_i$ is a set of disjoint intervals in $dom(X_i)$. Applying the definition in (2) and Lemma 11, we can see that (3) is bounded by N^{ρ^*}/Δ, noticing that the right hand side of (1) is at most N^{ρ^*}.

In the above analysis, we have fixed a set of $\ell_1, ..., \ell_d$. As each ℓ_i has $O(\log N)$ choices, all together there are $O(\log^d N) = \tilde{O}(1)$ different sets of $\ell_1, ..., \ell_d$. We can now conclude that the overall space is $\tilde{O}(N^{\rho^*}/\Delta)$.

Finally, we need a hash table to check in constant time whether a dyadic combination is heavy. The hash table occupies $\tilde{O}(N^{\rho^*}/\Delta)$ space because our earlier analysis implies a bound $\tilde{O}(N^{\rho^*}/\Delta)$ on the number of heavy dyadic combinations. The overall space of our entire structure is therefore $\tilde{O}(N + N^{\rho^*}/\Delta)$, where the term $\tilde{O}(N)$ counts the space for storing the relations of $\mathcal{R}$.

Query. Consider a range join query with interval $q = [x_1, x_2]$. We consider, w.l.o.g., that x_1 and x_2 are integers in $[1, N]$. In $\tilde{O}(1)$ time, we can partition the box $B(\underbrace{q, ..., q}_{t})$ into $O(\log^d N) = \tilde{O}(1)$ disjoint boxes, each in the form $B(I_1, ..., I_d)$ where $(I_1, ..., I_d)$ is a dyadic combination; we say that $(I_1, ..., I_d)$ is *canonical* for q. The query result is

$$\bigcup_{\text{canonical } (I_1, ..., I_d)} join(\mathcal{R}_{I_1,...,I_d}).$$

The results $join(\mathcal{R}_{I_1,...,I_d})$ of all the canonical $(I_1, ..., I_d)$ are disjoint. If a canonical $(I_1, ..., I_d)$ is heavy, we enumerate $join(\mathcal{R}_{I_1,...,I_d})$ with an $\tilde{O}(\Delta)$ delay using the structure of [20] on $(I_1, ..., I_d)$. Otherwise, we apply a worst-case optimal join algorithm [39, 40, 48] to compute $join(\mathcal{R}_{I_1,...,I_d})$. The algorithm finishes in $\tilde{O}(\text{AGM}(I_1, ..., I_d))$ time, which is $\tilde{O}(\Delta)$ by definition of light dyadic combination. Our algorithm guarantees a delay of $\tilde{O}(\Delta)$. This completes the proof of Theorem 10.

Remark. In [36], Khamis et al. used dyadic intervals in their algorithm for one-off computation of $join(\mathcal{R})$. Their main technical issue was to select "good" dyadic boxes (i.e., boxes of the form $B(I_1, ..., I_d)$) to cover the tuples in $join(\mathcal{R})$ once. That issue is non-existent in our context, where the primary obstacle is to argue that the total space given in (3) is affordable. We overcame the obstacle using Lemma 11, which, though perpahs no longer surprising given all the existing variations of the AGM bound, deserves a careful treatment that, we believe, has not appeared before.

[8] Strictly speaking, the space should also account for the relations in $\mathcal{R}_{I_1,...,I_d}$. In our context, it suffices to store the relations of $\mathcal{R}$ once and generate the relations in $\mathcal{R}_{I_1,...,I_d}$ when answering a query. Appendix D has additional details about [20].

6 Problem 2: Triangles

This section will describe a structure for Problem 2 under $Q =$ triangle. We will first attack, in Section 6.1 and 6.2, two fundamental problems whose solutions are vital to establishing Theorem 6, the proof of which is presented in Section 6.3.

6.1 The Range Triangle Edges Problem

This subsection will discuss the following standalone problem.

> **Range Triangle Edges (RTE).** Let G be an undirected graph with m edges. Given an interval $q = [x_1, x_2]$, a query returns: (i) all the edges appearing in at least one triangle of G_q; and (ii) $\Theta(m^*)$ triangles where m^* is the number of edges reported in (i).

We will develop a structure of $O(m)$ space that can answer a query in $\tilde{O}(m^*)$ time. Furthermore, the query can enumerate the m^* edges and the $\Theta(m^*)$ triangles both with a delay Δ.

Let us represent a triangle occurrence in G as *triangle*(u, v, w) where u, v, and w are the triangle's vertices. Ordering is important: we will always adhere to the convention $A_u < A_v < A_w$. Given an interval q, we denote by E_q^* the set of edges showing up in at least one triangle of G_q. Hence, $m^* = |E_q^*|$. If *triangle*(u, v, w) appears in G_q, we call $\{u, v\}$ a type-1 edge, $\{v, w\}$ a type-2 edge, and $\{u, w\}$ a type-3 edge. The total number of edges of all three types is between m^* and $3m^*$.[9]. Next, we explain how to extract the edges of each type in G_q.

Type 1 and 2. We will discuss only type 1 because type 2 is symmetric. For each edge $\{u, v\}$ in G (assume, w.l.o.g., $A_u < A_v$), identify a *sentinel* vertex w^* for $\{u, v\}$ as follows:

- $w^* =$ null if G has no occurrence of the form *triangle*(u, v, w);
- otherwise, w^* has the smallest attribute among all the vertices w making a triangle occurrence *triangle*(u, v, w) in G.

Consider any interval $q = [x_1, x_2]$. Observe that $\{u, v\}$ is a type-1 edge for q if and only if $x_1 \le A_u$ and $A_{w^*} \le x_2$. This motivates us to convert type-1 edge retrieval to range reporting on 2D points (introduced in Section 2). Towards the purpose, create a set P of points, which has a point (A_u, A_{w^*}) for every $\{u, v\}$ whose sentinel w^* is not null. Attach edge $\{u, v\}$ to the point (A_u, A_{w^*}) so that the former can be fetched along with the latter. The size of P is at most m. Given $q = [x_1, x_2]$, we can find all the type-1 edges by enumerating the points of P inside the rectangle $[x_1, \infty) \times (-\infty, x_2]$. Hence, we can store P in a Chazelle's structure (see Section 2) that has $O(|P|) = O(m)$ space and ensures an $\tilde{O}(1)$ delay in reporting the type-1 edges of any q.

Type 3. A similar approach works for type 3. Let $\{u, w\}$ be an edge appearing in at least one occurrence *triangle*(u, v, w) in G. It is a type-3 edge of $q = [x_1, x_2]$ if and only if $x_1 \le A_u$ and $A_w \le x_2$. By adapting the earlier discussion in a straightforward manner, we conclude that there is a structure of $O(m)$ space allowing us to retrieve all the type-3 edges with an $\tilde{O}(1)$ delay.

[9] Even for the same q, an edge can be of different types in various triangle occurrences.

Listing $\Theta(m^*)$ Triangles. The above has explained how to retrieve E_q^*, but an RTE query still needs to report $\Theta(m^*)$ triangles. Next, we remedy the issue by slightly modifying our solution so far.

Recall that, in dealing with type 1, we attached the edge $\{u, v\}$ to the point (A_u, A_{w^*}) generated from the edge. Now, we attach $\mathit{triangle}(u, v, w^*)$ to (A_u, A_{w^*}) as well. This way, when (A_u, A_{w^*}) is found, we obtain both $\{u, v\}$ and $\mathit{triangle}(u, v, w^*)$ for free. After applying the same idea to type-2 and type-3, we can assert that, whenever the query algorithm finds a type-1, -2, or -3 edge, it must have also found a triangle in G_q. Therefore, the algorithm can report the triangles in G_q with an $\tilde{O}(1)$ delay, although the same triangle may be reported up to three times[10]. By applying the duplicate-removal technique in Section 2, we now have an algorithm that can enumerate $\Theta(m^*)$ distinct triangles with an $\tilde{O}(1)$ delay. The number of distinct triangles reported is at least $m^*/3$ and at most $3m^*$.

6.2 The Small-Delay Triangle Listing Problem

In this subsection, we will concentrate on a standalone problem defined as follows.

> **Small-Delay Triangle Listing (SDTL).** *G is an undirected graph with m edges, each of which appears in at least one triangle. We are given $\Omega(m)$ free triangles and $O(m)$ forbidden triangles. Design an algorithm to enumerate all the triangles of G – except for the forbidden ones – with a small delay (free triangles must be enumerated). No preprocessing is allowed.*

We will settle the problem with an algorithm of delay $\tilde{O}(m^{\frac{\omega-1}{\omega+1}})$.

Suppose that G has OUT triangles in total. Our starting point is an algorithm of Bjorklund et al. [11] which is able to list k triangles in $\alpha \cdot m^{\frac{3(\omega-1)}{\omega+1}} k^{\frac{3-\omega}{\omega+1}}$ time, where $\alpha = \tilde{O}(1)$, for a parameter $k \in [\Omega(m), \mathrm{OUT}]$. As far as the algorithm of [11] is concerned, we can consider OUT known because it can be found in $O(m^{2\omega/(\omega+1)})$ time [3] which is $O(m^{\frac{3(\omega-1)}{\omega+1}} k^{\frac{3-\omega}{\omega+1}})$. The algorithm of Bjorklund et al. does not have a small delay, but we will turn it into one that does.

We run the algorithm of Bjorklund et al. [11] with geometrically-increasing k and, in each run, report only some, but not all, of the triangles. How many triangles are reported in each run is decided strategically to keep the delay small. Let S_{no}^0 be the set of forbidden triangles and S_{yes}^0 the set of free triangles in the beginning. Set $k_0 = |S_{no}^0| + |S_{yes}^0|$. When running the algorithm of [11] for the i-th time, we set its parameter k to $k_i = \min\{3^i k_0, \mathrm{OUT}\}$. We enforce the invariant that, when run i starts, there are always a set S_{no}^{i-1} of forbidden triangles and a set S_{yes}^{i-1} of free triangles. The set S_{yes}^{i-1} will be reported with a small delay during the i-th run (details to be clarified shortly).

Specifically, suppose that the i-th run finds a set S_{raw}^i of k_i triangles (some of which have been output in previous runs). We generate the forbidden and free sets for the next run as follows:

$$S_{no}^i = S_{no}^{i-1} \cup S_{yes}^{i-1} \text{ and then } S_{yes}^i \quad = \quad S_{raw}^i \setminus S_{no}^i.$$

[10] An occurrence $\mathit{triangle}(u, v, w)$ can be reported only when $\{u, v\}$, $\{v, w\}$, or $\{u, w\}$ is output as a type-1, -2, or -3 edge, respectively.

Run i finishes in $\alpha \cdot m^{\frac{3(\omega-1)}{\omega+1}} k_i^{\frac{3-\omega}{\omega+1}}$ time. We instruct the run to output a triangle from S_{yes}^{i-1} every

$$\alpha \cdot \frac{m^{\frac{3(\omega-1)}{\omega+1}} k_i^{\frac{3-\omega}{\omega+1}}}{|S_{yes}^{i-1}|} \tag{4}$$

atomic operations. We will show $|S_{yes}^{i-1}| = \Omega(k_i)$, with which the delay in (4) can be bounded as:

$$\tilde{O}\left(\frac{m^{\frac{3(\omega-1)}{\omega+1}}}{k_i^{\frac{2\omega-2}{\omega+1}}}\right) = \tilde{O}\left(m^{\frac{\omega-1}{\omega+1}}\right) \tag{5}$$

where the equality used $k_i \geq k_0 = \Omega(m)$.

For $i = 1$, $|S_{yes}^{i-1}| = \Omega(k_0)$ follows directly from the definition of the SDTL problem (i.e., we have $\Omega(m)$ free triangles to start with). To prove $|S_{yes}^{i-1}| = \Omega(k_i)$ for $i \geq 2$, we derive:

$$|S_{no}^{i-1}| \leq |S_{no}^0| + |S_{yes}^0| + \sum_{j=1}^{i-2} |S_{raw}^j| = k_0 + \sum_{j=1}^{i-2} 3^j \cdot k_0 = \sum_{j=0}^{i-2} 3^j \cdot k_0 < \frac{3^{i-1}k_0}{2}.$$

Therefore:

$$|S_{yes}^{i-1}| \quad \geq \quad |S_{raw}^{i-1}| - |S_{no}^{i-1}| > k_{i-1} - 3^{i-1}k_0/2 = k_{i-1}/2 = \Omega(k_i).$$

We now conclude that the delay of our algorithm is as given in (5).

6.3 Proof of Theorem 6

We are ready to explain how to solve Problem 2 with Q = triangle. In preprocessing, we build an RTE structure (Section 6.1) on G. Now, consider a (Problem-2) query with interval q. We start by issuing an RTE query to retrieve E_q^*, i.e., the set of edges appearing in at least one triangle of G_q. This, in effect, generates G_q^*, which is the subgraph of G_q induced by the edges in E_q^*. In addition, the RTE query has also enumerated a set S of $\Theta(m^*)$ triangles in G_q, where $m^* = |E_q^*|$. The size of S falls in $[\frac{m^*}{3}, 3m^*]$.

Our remaining mission is to enumerate the triangles in G_q^* that are outside S. Note that G_q^* is a graph with m^* edges and at least $\Theta(m^*)$ triangles. This motivates us to convert the mission to the SDTL problem, which has been solved in Section 6.2. However, the SDTL problem requires $\Theta(m^*)$ free triangles and $O(m^*)$ forbidden triangles as part of the input. Unfortunately, we do not seem to have these triangles at the moment.

We overcome this obstacle by, interestingly, dividing S into S_{yes} and S_{no}, such that S_{yes} (resp. S_{no}) serves as the set of free (resp. forbidden) triangles. Recall that the RTE query algorithm, denoted as $\mathcal{A}$, is designed to enumerate an edge in E_q^* with a delay $\Delta = \tilde{O}(1)$ and a triangle in S also with a delay Δ. Therefore, it must finish within $t_{max} = \max\{\Delta \cdot (|E_q^*| + 1), \Delta \cdot (|S| + 1)\} \leq \Delta \cdot (3m^* + 1)$ time. We can now apply the buffering technique in Section 2 with $\alpha = 18$ to turn $\mathcal{A}$ into an algorithm that outputs a triangle at the end of each epoch, which has a length 18Δ. The total number of epochs is at most $\frac{t_{max}}{18\Delta} \leq \frac{3m^*+1}{18}$. Thus, when $\mathcal{A}$ finishes, we have output at most $(3m^* + 1)/18$ triangles, whereas the buffer B (defined in Section 2) still has at least $|S| - \frac{3m^*+1}{18} = \Theta(m^*)$ triangles. We can, thus, set S_{yes} to the content of B when $\mathcal{A}$ finishes, and S_{no} to the set of triangles already output.

We can now apply the SDTL algorithm on G_q^* and, thus, complete the proof of Theorem 6.

7 Problem 2: Near-Constant Delays

This section will focus on two instances of Problem 2 where it is possible to achieve $\tilde{O}(1)$ delays with space substantially smaller than Theorem 5. We will discuss first $Q = \ell$-star in Section 7.1 and then $Q = 2\ell$-cycle in Section 7.2. We will focus on explaining how to enumerate a perhaps-not-distinct occurrence with an $\tilde{O}(1)$ delay, while ensuring each occurrence to be output only a constant number of times. Owing to the duplicate-removal method in Section 2, we can modify the algorithms to enumerate only *distinct* occurrences with $\tilde{O}(1)$ delays.

7.1 ℓ-Stars

Recall that an ℓ-star is a tree with only one non-leaf node, which we will refer to as the star's *center*. Consider a query with interval q. We refer to a node u as a *q-center* if G_q has at least one ℓ-star occurrence with u as the center. Once u is found, it becomes a trivial matter to enumerate all the ℓ-stars having u as the center with an $\tilde{O}(1)$ delay. Specifically, we can first (use a binary search tree to) retrieve all the neighbors v of u in G satisfying $A_v \in q$. From those neighbors, any ℓ distinct vertices form an ℓ-star together with u (as the center). It is rudimentary to ensure an $\tilde{O}(1)$ delay in enumerating all those stars.

Next, we concentrate on designing a structure to enumerate the q-centers with an $\tilde{O}(1)$ delay. Consider an arbitrary ℓ-star in G with center u. Sort the star's $\ell + 1$ vertices in ascending order of attribute and look for the position of u. If u is the r-th smallest, we will refer to the star as a *rank-r ℓ-star* and u as a *rank-r q-center*.

Now, fix an $r \in [1, \ell + 1]$. We will describe a structure to support the following operation:

Given an interval q, find all the rank-r q-centers, i.e., all vertices $u \in V$ s.t. G_q has a rank-r ℓ-star with u as the center.

Consider any rank-r ℓ-star in G having u as the center. Let us write out the star's vertices as $v_1, ..., v_{r-1}, u, v_{r+1}, ..., v_\ell$ in ascending order of attribute. For a $q = [x_1, x_2]$, the ℓ-star appears in G_q if and only if $x_1 \le A_{v_1}$ and $A_{v_\ell} \le x_2$. Refer to v_1 as a *left r-sentinel* of u and to v_ℓ as a *right r-sentinel* of u. From all the left r-sentinels of u (one from each rank-r ℓ-star with center u), identify the one v_1^* with the largest attribute. Similarly, from all the right r-sentinels of u, identify the one v_ℓ^* with the smallest attribute. Observe that u is a rank-r q-center if and only if $x_1 \le A_{v_1^*}$ and $A_{v_\ell^*} \le x_2$. We can therefore convert the retrieval of rank-r q-centers into range reporting on 2D points (review Section 2), in the same way as illustrated in Section 6.1. Following Section 2, we can create a Chazelle's structure on n points – each point created for a vertex $u \in V$ in the way explained – that has $O(n)$ space and, given any q, can list the rank-r q-centers with an $\tilde{O}(1)$ delay. This completes the proof of Theorem 7.

7.2 2ℓ-Cycles

We will start with an assumption: all queries specify a fixed $q = (-\infty, \infty)$, namely, there is effectively only one query, which enumerates all the 2ℓ-cycles in G. The assumption allows us to explain the core ideas with the minimum technical details and will be removed eventually.

Queries with $q = (-\infty, \infty)$. Given a 2ℓ-cycle occurrence, we refer to the vertex u in the cycle having the smallest attribute as the occurrence's *anchor*. Let v be the vertex in the cycle such that cutting the cycle at u and v gives two ℓ-paths connecting u and v. We will refer to v as the occurrence's *inverse anchor*, the pair (u, v) as an *anchor pair*, and the two aforementioned paths as *cycle ℓ-paths*. The number of cycle ℓ-paths is at most $\#P_\ell$ (recall that $\#P_\ell$ is the total number of ℓ-paths).

The problem may appear deceivingly simple: can't we answer a query by simply concatenating, for each anchor pair (u, v), every two cycle ℓ-paths from u to v? This does not work because the two cycle ℓ-paths may share common vertices other than u and v, in which case the concatenation does not yield a 2ℓ-cycle! This motivates a crucial notion: two cycle ℓ-paths are *interior disjoint* if they (i) have the same anchor pair (u, v), and (ii) do not share any common vertex except u and v. Concatenating two cycle ℓ-paths from u to v spawns a 2ℓ-cycle if and only if those paths are interior disjoint. The challenge we are facing at this moment is the following problem.

> Design a structure to support the following operation: given a cycle ℓ-path π from anchor u to inverse anchor v, list all the cycle ℓ-paths interior disjoint with π with an $\tilde{O}(1)$ delay.

We will overcome the challenge with a structure of $\tilde{O}(\#P_\ell)$ space.

Our main observation is that the operation can be converted to range reporting on $(\ell-1)$-dimensional points (review Section 2). To explain, let us consider any cycle ℓ-path π from anchor u to inverse anchor v. After excluding u and v, the path has $\ell - 1$ vertices, which we list as $w_1, w_2, ..., w_{\ell-1}$ in ascending order of attribute[11]. Convert π into an $(\ell-1)$-dimensional point $(A_{w_1}, ..., A_{w_{\ell-1}})$. Let $P_{u,v}$ be the set of points thus obtained from all the cycle ℓ-paths with (u, v) as the anchor pair.

Now, consider another cycle ℓ-path π' from u to v. List the vertices of π' other than u and v as $w'_1, w'_2, ..., w'_{\ell-1}$ also in ascending order of attribute. If π' is interior disjoint with π, each $A_{w'_i}$ $(i \in [1, \ell-1])$ must fall in one of the ℓ open intervals:

$$(-\infty, A_{w_1}), (A_{w_1}, A_{w_2}), ..., (A_{w_{\ell-2}}, A_{w_{\ell-1}}), (A_{w_{\ell-1}}, \infty). \tag{6}$$

Therefore, $(A_{w'_1}, ..., A_{w'_{\ell-1}})$ – the point converted from π' – must fall in one of the following $\ell^{\ell-1} = O(1)$ rectangles: $q_1 \times q_2 \times ... \times q_{\ell-1}$, where each q_i $(i \in [1, \ell-1])$ is taken independently from one of the intervals in (6). As per Section 2, by creating a range tree on $P_{u,v}$ of $\tilde{O}(|P_{u,v}|)$ space, we can enumerate all the points in such a rectangle with an $\tilde{O}(1)$ delay.

The conclusion from the above is that, for each anchor pair (u, v), we can create a range tree of $\tilde{O}(|P_{u,v}|)$ space which, given any cycle ℓ-path cycle π from u to v, permits the enumeration of every cycle ℓ-path π', which is interior disjoint with π, with an $\tilde{O}(1)$ delay. The structures of all the anchor pairs use in total $\sum_{\text{anc. pair }(u,v)} \tilde{O}(|P_{u,v}|) = \tilde{O}(\#P_\ell)$ space.

With the challenge conquered, listing all the 2ℓ-cycles becomes an easy matter. We simply look at each cycle ℓ-path π, retrieve every ℓ-path π' interior disjoint with π, and make a cycle by concatenating π and π'. The delay in cycle reporting is $\tilde{O}(1)$ (each 2ℓ-cycle can be reported twice).

Arbitrary Queries. Next, we remove the constraint $q = (-\infty, \infty)$ and tackle queries with arbitrary q. A new issue now arises: a query can no longer afford to look at all the cycle ℓ-paths. We say that a cycle ℓ-path from anchor u to inverse anchor v *contributes* to G_q if it makes a 2ℓ-cycle in G_q with another interior disjoint cycle ℓ-path. We need a way to list only the contributing cycle ℓ-paths.

[11] The order should not be confused with the order by which the vertices appear in π.

Fix any cycle ℓ-path π with anchor pair (u, v). Let S_π be the set of 2ℓ-cycles in G that include π and have (u, v) as the anchor pair. Take an arbitrary cycle from S_π. By definition of anchor, u has the smallest attribute among the cycle's vertices. Let w be the vertex in the cycle with the largest attribute. For $q = [x_1, x_2]$, the cycle appears in G_q if and only if $x_1 \le A_u$ and $A_w \le x_2$. Let w^* be the vertex with the smallest attribute among all such w's. It becomes evident that π contributes to the G_q of $q = [x_1, x_2]$ if and only if $x_1 \le A_u$ and $A_{w^*} \le x_2$. We can therefore convert the retrieval of contributing cycle ℓ-paths to range reporting on 2D points, using the method in Section 6.1. The resulting structure (a Chazelle's structure) stores a point converted from every cycle ℓ-path and uses $O(\#P_\ell)$ space. Give any q, we can list the cycle ℓ-paths contributing to G_q with an $\tilde{O}(1)$ delay.

Suppose that we have found a contributing cycle ℓ-path π with anchor pair (u, v). As before, we proceed to find the cycle ℓ-paths π' interior disjoint with π. The new requirement here, however, is that π' needs to be contributing as well. Recall that, in the $q = (-\infty, \infty)$ scenario, we converted the task to range reporting on $(\ell - 1)$-dimensional points. To deal with arbitrary $q = [x_1, x_2]$, we will increase the dimension by one.

To explain, in a fashion like before, let us list out the vertices of π – after excluding u and v – as $w_1, ..., w_{\ell-1}$ in ascending order of attribute. Denote by $w_{\max}$ the vertex in π with the largest attribute ($w_{\max}$ can be v). Convert π to an ℓ-dimensional point $(A_{w_1}, ..., A_{w_{\ell-1}}, A_{w_{\max}})$. Let $(A_{w'_1}, ..., A_{w'_{\ell-1}}, A_{w'_{\max}})$ be the point converted from π' in the same manner. As we already know $A_u \in [x_1, x_2]$ (recall that π is a contributing path), π' is a path we want if and only if it satisfies the conditions below:

- $A_{w'_i}$ $(1 \le i \le \ell - 1)$ falls in one of the ℓ intervals in (6);
- $A_{w'_{\max}} \le x_2$.

Thus, the point $(A_{w'_1}, ..., A_{w'_{\ell-1}}, A_{w'_{\max}})$ must fall in one of the following $\ell^{\ell-1} = O(1)$ rectangles: $q_1 \times q_2 \times ... \times q_{\ell-1} \times (-\infty, x_2]$, where each q_i $(i \in [1, \ell - 1])$ is an interval taken independently from (6). By the above reasoning, for each anchor pair (u, v), we create a set $P_{u,v}$ of ℓ-dimensional points, each converted from a cycle ℓ-path with anchor pair (u, v), and then build a range tree on $P_{u,v}$. The range trees of all anchor pairs use $\sum_{\text{anc. pair }(u,v)} \tilde{O}(|P_{u,v}|) = \tilde{O}(\#P_\ell)$ space in total.

We now elaborate on the overall algorithm for answering a (Problem-2) query with parameter q. First, enumerate all the cycle ℓ-paths contributing to G_q with an $\tilde{O}(1)$ delay; call this the *outer enumeration*. Every time such a path π – say with anchor pair (u, v) – is obtained, we suspend outer enumeration and utilize the range tree on $P_{u,v}$ to find all the paths π' discussed previously with an $\tilde{O}(1)$ delay. Upon the delivery of a π', concatenate it with π and output the 2ℓ-cycle obtained. After exhausting all such π', we resume outer enumeration. This concludes the proof of Theorem 8.

References

1 Josh Alman and Virginia Vassilevska Williams. A refined laser method and faster matrix multiplication. In *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 522–539, 2021.

2 Noga Alon, Raphael Yuster, and Uri Zwick. Color-coding. *Journal of the ACM (JACM)*, 42(4):844–856, 1995.

3 Noga Alon, Raphael Yuster, and Uri Zwick. Finding and counting given length cycles. *Algorithmica*, 17(3):209–223, 1997.

4 Albert Atserias, Martin Grohe, and Daniel Marx. Size bounds and query plans for relational joins. *SIAM Journal on Computing*, 42(4):1737–1767, 2013.

5 Guillaume Bagan, Arnaud Durand, and Etienne Grandjean. On acyclic conjunctive queries and constant delay enumeration. In *Computer Science Logic*, pages 208–222, 2007.

6 Michael J Bannister, Christopher DuBois, David Eppstein, and Padhraic Smyth. Windows into relational events: Data structures for contiguous subsequences of edges. In *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 856–864, 2013.

7 Jon Louis Bentley. Decomposable searching problems. *Information Processing Letters (IPL)*, 8(5):244–251, 1979.

8 Suman K. Bera, Noujan Pashanasangi, and C. Seshadhri. Near-linear time homomorphism counting in bounded degeneracy graphs: The barrier of long induced cycles. In *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2315–2332, 2021.

9 Christoph Berkholz, Jens Keppeler, and Nicole Schweikardt. Answering conjunctive queries under updates. In *Proceedings of ACM Symposium on Principles of Database Systems (PODS)*, pages 303–318, 2017.

10 Andreas Bjorklund, Petteri Kaski, and Lukasz Kowalik. Counting thin subgraphs via packings faster than meet-in-the-middle time. *ACM Transactions on Algorithms*, 13(4):48:1–48:26, 2017.

11 Andreas Bjorklund, Rasmus Pagh, Virginia Vassilevska Williams, and Uri Zwick. Listing triangles. In *Proceedings of International Colloquium on Automata, Languages and Programming (ICALP)*, pages 223–234, 2014.

12 Nofar Carmeli and Markus Kroll. On the enumeration complexity of unions of conjunctive queries. *ACM Transactions on Database Systems (TODS)*, 46(2):5:1–5:41, 2021.

13 Nofar Carmeli, Shai Zeevi, Christoph Berkholz, Benny Kimelfeld, and Nicole Schweikardt. Answering (unions of) conjunctive queries using random access and random-order enumeration. In *Proceedings of ACM Symposium on Principles of Database Systems (PODS)*, pages 393–409, 2020.

14 Farah Chanchary and Anil Maheshwari. Time windowed data structures for graphs. *J. Graph Algorithms Appl.*, 23(2):191–226, 2019.

15 Farah Chanchary, Anil Maheshwari, and Michiel Smid. Querying relational event graphs using colored range searching data structures. *Discrete Applied Mathematics*, 286:51–61, 2020.

16 Bernard Chazelle. A functional approach to data structures and its use in multidimensional searching. *SIAM Journal of Computing*, 17(3):427–462, 1988.

17 N. Chiba and T. Nishizeki. Arboricity and subgraph listing algorithms. *SIAM Journal of Computing*, 14(1):210–223, 1985.

18 Radu Curticapean, Holger Dell, and Dániel Marx. Homomorphisms are a good basis for counting small subgraphs. In *Proceedings of ACM Symposium on Theory of Computing (STOC)*, pages 210–223, 2017.

19 Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, 3rd edition, 2008.

20 Shaleen Deep and Paraschos Koutris. Compressed representations of conjunctive query results. In *Proceedings of ACM Symposium on Principles of Database Systems (PODS)*, pages 307–322, 2018.

21 Arnaud Durand. Fine-grained complexity analysis of queries: From decision to counting and enumeration. In *Proceedings of ACM Symposium on Principles of Database Systems (PODS)*, pages 331–346, 2020.

22 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.

23 Arnaud Durand, Nicole Schweikardt, and Luc Segoufin. Enumerating answers to first-order queries over databases of low degree. In *Proceedings of ACM Symposium on Principles of Database Systems (PODS)*, pages 121–131, 2014.

24 David Eppstein. Arboricity and bipartite subgraph listing algorithms. *Information Processing Letters (IPL)*, 51(4):207–211, 1994.

25 David Eppstein. Subgraph isomorphism in planar graphs and related problems. *J. Graph Algorithms Appl.*, 3(3):1–27, 1999.

26 David Eppstein, Maarten Loffler, and Darren Strash. Listing all maximal cliques in sparse graphs in near-optimal time. In *International Symposium on Algorithms and Computation (ISAAC)*, volume 6506, pages 403–414, 2010.

27 Peter Floderus, Miroslaw Kowaluk, Andrzej Lingas, and Eva-Marta Lundell. Detecting and counting small pattern graphs. *SIAM J. Discret. Math.*, 29(3):1322–1339, 2015.

28 Fedor V. Fomin, Daniel Lokshtanov, Venkatesh Raman, Saket Saurabh, and B. V. Raghavendra Rao. Faster algorithms for finding and counting subgraphs. *Journal of Computer and System Sciences (JCSS)*, 78(3):698–706, 2012.

29 Ehud Friedgut. Hypergraphs, entropy, and inequalities. *Am. Math. Mon.*, 111(9):749–760, 2004.

30 Pierre-Louis Giscard, Nils M. Kriege, and Richard C. Wilson. A general purpose algorithm for counting simple cycles and simple paths of any length. *Algorithmica*, 81(7):2716–2737, 2019.

31 Isaac Goldstein, Tsvi Kopelowitz, Moshe Lewenstein, and Ely Porat. Conditional lower bounds for space/time tradeoffs. In *Algorithms and Data Structures Workshop (WADS)*, pages 421–436. Springer, 2017.

32 Isaac Goldstein, Moshe Lewenstein, and Ely Porat. On the hardness of set disjointness and set intersection with bounded universe. In *International Symposium on Algorithms and Computation (ISAAC)*, pages 7:1–7:22, 2019.

33 Chinh T. Hoang, Marcin Kaminski, Joe Sawada, and R. Sritharan. Finding and listing induced paths and cycles. *Discrete Applied Mathematics*, 161(4-5):633–641, 2013.

34 Ahmet Kara, Milos Nikolic, Dan Olteanu, and Haozhe Zhang. Trade-offs in static and dynamic evaluation of hierarchical queries. In *Proceedings of ACM Symposium on Principles of Database Systems (PODS)*, pages 375–392, 2020.

35 Wojciech Kazana and Luc Segoufin. Enumeration of first-order queries on classes of structures with bounded expansion. In *Proceedings of ACM Symposium on Principles of Database Systems (PODS)*, pages 297–308, 2013.

36 Mahmoud Abo Khamis, Hung Q. Ngo, Christopher Ré, and Atri Rudra. Joins via geometric resolutions: Worst-case and beyond. In *Proceedings of ACM Symposium on Principles of Database Systems (PODS)*, pages 213–228, 2015.

37 Ton Kloks, Dieter Kratsch, and Haiko Müller. Finding and counting small induced subgraphs efficiently. *Information Processing Letters (IPL)*, 74(3-4):115–121, 2000.

38 Jaroslav Nesetril and Svatopluk Poljak. On the complexity of the subgraph problem. *Commentationes Mathematicae Universitatis Carolinae*, 26(2):415–419, 1985.

39 Hung Q. Ngo, Ely Porat, Christopher Re, and Atri Rudra. Worst-case optimal join algorithms. *Journal of the ACM (JACM)*, 65(3):16:1–16:40, 2018.

40 Hung Q. Ngo, Christopher Re, and Atri Rudra. Skew strikes back: new developments in the theory of join algorithms. *SIGMOD Rec.*, 42(4):5–16, 2013.

41 Dan Olteanu and Jakub Zavodny. Size bounds for factorised representations of query results. *ACM Transactions on Database Systems (TODS)*, 40(1):2:1–2:44, 2015.

42 Saladi Rahul. Improved bounds for orthogonal point enclosure query and point location in orthogonal subdivisions in $\mathbb{R}^3$. In *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 200–211, 2015.

43 Nicole Schweikardt, Luc Segoufin, and Alexandre Vigny. Enumeration for FO queries over nowhere dense graphs. In *Proceedings of ACM Symposium on Principles of Database Systems (PODS)*, pages 151–163, 2018.

44 Luc Segoufin. Constant delay enumeration for conjunctive queries. *SIGMOD Rec.*, 44(1):10–17, 2015.

45 Luc Segoufin and Alexandre Vigny. Constant delay enumeration for FO queries over databases with local bounded expansion. In *Proceedings of International Conference on Database Theory (ICDT)*, volume 68, pages 20:1–20:16, 2017.

46 Yufei Tao. Algorithmic techniques for independent query sampling. In *Proceedings of ACM Symposium on Principles of Database Systems (PODS)*, 2022.

47 Yufei Tao and Ke Yi. Intersection joins under updates. *Journal of Computer and System Sciences (JCSS)*, 124:41–64, 2022.

48 Todd L. Veldhuizen. Triejoin: A simple, worst-case optimal join algorithm. In *Proceedings of International Conference on Database Theory (ICDT)*, pages 96–106, 2014.

49 Duncan J. Watts and Steven H. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393:440–442, 1998.

50 Virginia Vassilevska Williams and Ryan Williams. Finding, minimizing, and counting weighted subgraphs. *SIAM Journal of Computing*, 42(3):831–854, 2013.

A Correctness of the Reduction in Section 3.1

In our construction, S_i ($i \in [1, s]$) corresponds to two set vertices with attribute values i and $i + s$, respectively. To facilitate derivation, we make a copy of each set: define $S_i = S_{i-s}$ for each $i \in [s + 1, 2s]$. In the rest of the proof, we hold the view that each S_i ($i \in [1, 2s]$) corresponds to only one set vertex, the one with attribute value i.

Consider a wedge occurrence with vertices u, v, and w where the edges are $\{u, v\}$ and $\{v, w\}$. We classify it as one of the two types below:

- (type e-s-e) u and w are element vertices and v is a set vertex;
- (type s-e-s) u and w are set vertices and v an element vertex.

▶ **Lemma 12.** *For any interval $q = [x, y]$ satisfying $1 \le x < s + 1/2 < y \le 2s$, we have*

- *the number of e-s-e wedges in G_q is $\sum_{i\in[x,y]} \binom{|S_i|}{2}$;*
- *the number of s-e-s wedges in G_q is $\sum_{i\in[x,y],j\in[i+1,y]} |S_i \cap S_j|$.*

Proof. To prove the first bullet, define an *e-s-e tuple* as (e_1, S_i, e_2) where $i \in q$ and e_1 and e_2 are distinct elements in S_i. The number of such tuples is $\sum_{i=x}^{y} \binom{|S_i|}{2}$. Our construction ensures a one-one correspondence between e-s-e tuples and e-s-e wedges in G_q.

To prove the second bullet, define an *s-e-s tuple* as (S_i, e, S_j) where $x \le i < j \le y$ and $e \in S_i \cap S_j$. The number of such tuples is $\sum_{i\in[x,y],j\in[i+1,y]} |S_i \cap S_j|$. Our construction ensures a one-one correspondence between s-e-s tuples and s-e-s wedges in G_q. ◀

To find out whether $S_a \cap S_b$ is empty, our reduction issues four Problem-1 queries with intervals $q_1 = [a, s + b]$, $q_2 = [a + 1, s + b]$, $q_3 = [a, s + b - 1]$, and $q_4 = [a + 1, s + b - 1]$, respectively. The above lemma is applicable to all these intervals. For $i \in [1, 4]$, let c'_i (resp. c''_i) be the number of e-s-e (resp. s-e-s) wedges in G_{q_i}; this means that c_i, the total number of wedges in G_{q_i}, equals $c'_i + c''_i$. According to Lemma 12, we have:

$$\begin{aligned}
& c'_1 - c'_2 - c'_3 + c'_4 \\
= \; & \sum_{i\in[a,s+b]} \binom{|S_i|}{2} - \sum_{i\in[a+1,s+b]} \binom{|S_i|}{2} - \sum_{i\in[a,s+b-1]} \binom{|S_i|}{2} \\
& + \sum_{i\in[a+1,s+b-1]} \binom{|S_i|}{2} \\
= \; & 0
\end{aligned}$$

and

$$
\begin{aligned}
& c''_1 - c''_2 - c''_3 + c''_4 \\
= & \left(\sum_{\substack{i\in[a,s+b] \\ j\in[i+1,s+b]}} |S_i \cap S_j| - \sum_{\substack{i\in[a+1,s+b] \\ j\in[i+1,s+b]}} |S_i \cap S_j| \right) - \\
& \left(\sum_{\substack{i\in[a,s+b-1] \\ j\in[i+1,s+b-1]}} |S_i \cap S_j| - \sum_{\substack{i\in[a+1,s+b-1] \\ j\in[i+1,s+b-1]}} |S_i \cap S_j| \right) \\
= & \sum_{j\in[a+1,s+b]} |S_a \cap S_j| - \sum_{j\in[a+1,s+b-1]} |S_a \cap S_j| \\
= & |S_a \cap S_{s+b}| \\
= & |S_a \cap S_b|.
\end{aligned}
$$

We thus conclude that $c_1 - c_2 - c_3 + c_4 = |S_a \cap S_b|$.

B Proof of Lemma 9

Let us first consider a variant of the set disjointness problem.

Weighted Set Intersection Size. We have $s \geq 2$ sets S_1, S_2, ..., S_s. Each S_i ($i \in [1, s]$) is associated with a function $weight_{S_i}$ which assigns to each element $e \in S_i$ a value $weight_{S_i}(e)$. Given distinct set ids $a, b \in [1, s]$, a query returns

$$size(S_a, S_b) = \sum_{e\in S_a\cap S_b} weight_{S_a}(e) \cdot weight_{S_b}(e). \tag{7}$$

Let $N = \sum_{i=1}^{s} |S_i|$. For any $\lambda \in [1, \sqrt{N}]$, it is straightforward to build a structure of $O(N^2/\lambda^2)$ space answering a query in $O(\lambda)$ time. Call S_i ($i \in [1, s]$) a *large* set if $|S_i| > \lambda$, or a *small* set otherwise. The number of large sets is at most N/λ. For each pair $(i, j) \in [1, s] \times [1, s]$, $i \neq j$, such that S_i and S_j are both large, we store $size(S_i, S_j)$; the space needed is $O(N^2/\lambda^2)$. Given a query with parameters a and b, return $size(S_a, S_b)$ directly if S_a and S_b are both large. Otherwise, assume, w.l.o.g., that S_a is small. We compute $S_a \cap S_b$ in $O(\lambda)$ time using a hash table (for each $e \in S_a$, check if $e \in S_b$). The result $size(S_a, S_b)$ can then be obtained easily.

Equipped with the above, next we describe a structure for the colored range wedge counting problem to prove Lemma 9.

Structure. First obtain a canonical collection $\mathcal{C}$ of V (defined in Section 4) satisfying $\sum_{U\in\mathcal{C}} |U| = \tilde{O}(n)$. For each $U \in \mathcal{C}$ – recall that U is a subset of V – construct a weighted set as follows:

- S_U = the set of black vertices adjacent to at least one vertex in U;
- for each $b \in S_U$, $weight_{S_U}(b)$ = the number of vertices in U adjacent to b.

These weighted sets constitute an instance of the weighted set intersection size problem. Build a structure described earlier on the instance using the given parameter λ. The lemma below implies that the structure occupies $\tilde{O}(m^2/\lambda^2)$ space.

▶ **Lemma 13.** $\sum_{U \in \mathcal{C}} |S_U| = \tilde{O}(m)$.

Proof. Each $b \in S_U$ is adjacent to a vertex $u \in U$. Pay a dollar to the edge $\{b, u\}$ for each such pair (b, u). Since an edge can receive a dollar only if it has a vertex in U, it can receive up to two dollars[12]. $|S_U|$ is no more than the number of dollars paid. Do the above for all $U \in \mathcal{C}$. Each edge in G can receive $\tilde{O}(1)$ dollars in total because every vertex appears in $\tilde{O}(1)$ subsets in $\mathcal{C}$ (Property P4-1 of $\mathcal{C}$; see Section 4). ◀

For any distinct $U, U' \in \mathcal{C}$, define $\mathit{size}(S_U, S_{U'})$ as in (7). On the other hand, for each $U \in \mathcal{C}$, define

$$\mathit{size}(S_U, S_U) \quad = \quad \sum_{b \in S_U} \binom{\mathit{weight}_{S_U}(b)}{2}.$$

We store the value $\mathit{size}(S_U, S_U)$ for all U. The total space is $\tilde{O}(m^2/\lambda^2)$.

Before proceeding, the reader should note the following subtle fact about the function $\mathit{size}(.,.)$:

Fact B-1: $\mathit{size}(S_U, S_{U'})$ is the number of occurrences $\mathit{wedge}(u, v, w)$ in G such that $u \in S_U$, $w \in S_{U'}$, and v is black.

The fact holds even if $U = U'$.

Query. Given a query with interval q, in $\tilde{O}(1)$ time we can pick $h = \tilde{O}(1)$ members $U_1, ..., U_h$ from $\mathcal{C}$ that form a partition of V_q (Property P4-2 of $\mathcal{C}$). The query returns

$$\sum_{i,j \in [1,h]: i \le j} \mathit{size}(S_{U_i}, S_{U_j}). \tag{8}$$

Each $\mathit{size}(S_{U_i}, S_{U_j})$ is either explicitly stored or can be obtained from the weighted set intersection size structure in $O(\lambda)$ time. The overall query time is therefore $\tilde{O}(\lambda)$.

Fact B-1 and $U_1, ..., U_h$ forming a partition of V_q assure us that (8) counts only occurrences $\mathit{wedge}(u, v, w)$ in G such that $A_u \in q$, $A_w \in q$, and v is black. To complete the correctness argument, we still need to show that (8) counts every such occurrence exactly once. Indeed, there exist unique $a, b \in [1, h]$ such that $a \le b$, $u \in U_a$, and $w \in U_b$. The wedge is counted only by the term in (8) with $i = a$ and $j = b$.

C Proof of Lemma 11

Let us first review Hölder's Inequality. Fix some positive integers α and β. Let

- $x_{i,j}$, for each $i \in [1, \alpha]$ and $j \in [1, \beta]$, be non-negative real numbers;
- y_j, for each $j \in [1, \beta]$, be non-negative real numbers satisfying $\sum_{j=1}^{\beta} y_j \ge 1$.

Under the convention $0^0 = 0$, Hölder's inequality states that:

$$\sum_{i=1}^{\alpha} \prod_{j=1}^{\beta} x_{i,j}^{y_j} \le \prod_{j=1}^{\beta} \left(\sum_{i=1}^{\alpha} x_{i,j} \right)^{y_j}. \tag{9}$$

A proof can be found in [29].

[12] Two is possible: this happens when b and u are both black and both appear in U.

We now return to the context of Lemma 11. Given any $j \in [1, d-1]$ and $(I_1, I_2, ..., I_j) \in \boldsymbol{\mathcal{I}}_1 \times ...\boldsymbol{\mathcal{I}}_j$, we will use $B(I_1, I_2, ..., I_j)$ as a short-form for the d-dimensional box

$$B(I_1, ..., I_j, dom(X_{j+1}), ..., dom(X_d)).$$

As a special case, define $B(\emptyset) = B(dom(X_1), ..., dom(X_d))$.

▶ **Lemma 14.** *For any $j \in [1, d]$, we have*

$$\sum_{I_j \in \boldsymbol{\mathcal{I}}_j} \prod_{e \in \mathcal{E}} |R_e \ltimes B(I_1, ..., I_j)|^{W(e)} \le \prod_{e \in \mathcal{E}} |R_e \ltimes B(I_1, ..., I_{j-1})|^{W(e)}.$$

Proof. Define

$$\mathcal{E}_j = \{e \in \mathcal{E} \mid X_j \in e\}.$$

Since $\sum_{e \in \mathcal{E}_j} W(e) \ge 1$ (W is a fractional edge covering), from Hölder's inequality (9) we have

$$\begin{aligned} & \sum_{I_j \in \boldsymbol{\mathcal{I}}_j} \prod_{e \in \mathcal{E}_j} |R_e \ltimes B(I_1, ..., I_j)|^{W(e)} \\ \le\; & \prod_{e \in \mathcal{E}_j} \Big(\sum_{I_j \in \boldsymbol{\mathcal{I}}_j} |R_e \ltimes B(I_1, ..., I_j)| \Big)^{W(e)} \\ \le\; & \prod_{e \in \mathcal{E}_j} \Big| R_e \ltimes B\big(I_1, ..., I_{j-1}, dom(X_j)\big) \Big|^{W(e)} \\ =\; & \prod_{e \in \mathcal{E}_j} |R_e \ltimes B(I_1, ..., I_{j-1})|^{W(e)} \end{aligned} \tag{10}$$

where the second inequality used the fact that $\boldsymbol{\mathcal{I}}_j$ is a set of disjoint intervals in $dom(X_j)$.

For each $e \in \mathcal{E} \setminus \mathcal{E}_j$, $R_e \ltimes B(I_1, ..., I_j)$ does not depend on I_j and can be rewritten as $R_e \ltimes B(I_1, ..., I_{j-1})$. We can thus derive:

$$\begin{aligned} & \sum_{I_j \in \boldsymbol{\mathcal{I}}_j} \prod_{e \in \mathcal{E}} |R_e \ltimes B(I_1, ..., I_j)|^{W(e)} \\ =\; & \sum_{I_j \in \boldsymbol{\mathcal{I}}_j} \Big(\prod_{e \in \mathcal{E} \setminus \mathcal{E}_j} |R_e \ltimes B(I_1, ..., I_j)|^{W(e)} \cdot \prod_{e \in \mathcal{E}_j} |R_e \ltimes B(I_1, ..., I_j)|^{W(e)} \Big) \\ =\; & \prod_{e \in \mathcal{E} \setminus \mathcal{E}_j} |R_e \ltimes B(I_1, ..., I_j)|^{W(e)} \cdot \sum_{I_j \in \boldsymbol{\mathcal{I}}_j} \prod_{e \in \mathcal{E}_j} |R_e \ltimes B(I_1, ..., I_j)|^{W(e)} \\ \le\; & \prod_{e \in \mathcal{E} \setminus \mathcal{E}_j} |R_e \ltimes B(I_1, ..., I_{j-1})|^{W(e)} \cdot \prod_{e \in \mathcal{E}_j} |R_e \ltimes B(I_1, ..., I_{j-1})|^{W(e)} \\ =\; & \prod_{e \in \mathcal{E}} |R_e \ltimes B(I_1, ..., I_{j-1})|^{W(e)}. \end{aligned}$$

where the inequality used (10). ◀

We can prove Lemma 11 with d applications of Lemma 14:

$$\begin{aligned}
& \sum_{I_1 \in \mathcal{I}_1} \dots \sum_{I_d \in \mathcal{I}_d} \prod_{e \in \mathcal{E}} |R_e \ltimes B(I_1, \dots, I_d)|^{W(e)} \\
\leq & \sum_{I_1 \in \mathcal{I}_1} \dots \sum_{I_{d-1} \in \mathcal{I}_{d-1}} \prod_{e \in \mathcal{E}} |R_e \ltimes B(I_1, \dots, I_{d-1})|^{W(e)} \\
\leq & \sum_{I_1 \in \mathcal{I}_1} \dots \sum_{I_{d-2} \in \mathcal{I}_{d-2}} \prod_{e \in \mathcal{E}} |R_e \ltimes B(I_1, \dots, I_{d-2})|^{W(e)} \\
\leq & \dots \\
\leq & \sum_{I_1 \in \mathcal{I}_1} \prod_{e \in \mathcal{E}} |R_e \ltimes B(I_1)|^{W(e)} \\
\leq & \prod_{e \in \mathcal{E}} |R_e|^{W(e)}.
\end{aligned}$$

D Proof of Theorem 5

The reader should read this proof after having finished Section 5. The basic idea is to convert Problem 2 to range join. Let $\mathcal{X}$ (resp. $\mathcal{E}$) be the set of vertices (resp. edges) in the pattern graph Q. The reader should not confuse $\mathcal{X}$ and $\mathcal{E}$ with V and E: the latter two are defined on the data graph G. For each edge $e \in \mathcal{E}$, construct a relation R_e with two attributes by inserting, for each edge $\{u, v\}$ in G, two tuples (u, v) and (v, u). This defines a join instance $\mathcal{R} = \{R_e \mid e \in \mathcal{E}\}$ with input size $N = 2m \cdot |\mathcal{E}| = O(m)$.

Every occurrence of Q corresponds to a constant number of tuples in $join(\mathcal{R})$. Motivated by this, given a Problem-2 query with interval q, we issue a range join query on $\mathcal{R}$ with q, which guarantees retrieving all the occurrences. The issue, however, is that not every tuple in $join(\mathcal{R})$ gives rise to an occurrence. To see this, consider Q = 4-cycle and, hence, $\mathcal{R}$ has four relations with schemes (X_1, X_2), (X_2, X_3), (X_3, X_4), and (X_4, X_1), respectively. Let $\{u, v\}$ be an arbitrary edge in E; tuples (u, v), (v, u), (u, v), and (v, u) exist in the four relations, respectively. Thus, $join(\mathcal{R})$ contains a tuple (u, v, u, v) that does not correspond to any occurrence.

The issue can be eliminated by slightly modifying the structure of [20], which we review next. Consider an arbitrary set $\mathcal{R}$ of relations (with any number of attributes) defined in Section 5. Deep and Koutris [20] proved the existence of a set $\mathcal{B}$ of boxes such that:

- each box has the form $B(I_1, \dots, I_d)$ where I_i is an interval in $dom(X_i)$ for $i \in [1, d]$;
- the boxes are disjoint and their union is $B(dom(X_1), dom(X_2), \dots, dom(X_d))$;
- for each box $B(I_1, \dots, I_d)$, the join instance $\mathcal{R}_{I_1, \dots, I_d}$ has a non-empty result;
- each box $B(I_1, \dots, I_d)$ satisfies $\mathrm{AGM}(I_1, \dots, I_d) \leq \Delta$;
- $|\mathcal{B}| = O(N^{\rho^*}/\Delta)$.

The structure of [20] simply stores $\mathcal{B}$ itself and uses $O(N^{\rho^*}/\Delta)$ space[13]. To enumerate $join(\mathcal{R})$, the algorithm of [20] looks at each $B(I_1, \dots, I_d) \in \mathcal{B}$ and applies a worst-case optimal join algorithm [39, 40, 48] to compute $join(\mathcal{R}_{I_1, \dots, I_d})$ in $\tilde{O}(\mathrm{AGM}(I_1, \dots, I_d)) = \tilde{O}(\Delta)$ time. This guarantees a delay of $\tilde{O}(\Delta)$.

We now adapt the structure to list all the occurrences of Q in G (fixing $q = (-\infty, \infty)$). Construct $\mathcal{R}$ from G and Q as before. Apply [20] to find a set $\mathcal{B}$ with all the properties explained earlier. Then, inspect each box $B(I_1, \dots, I_d) \in \mathcal{B}$ in turn and remove it from $\mathcal{B}$ if

[13] Obviously, the relations of $\mathcal{R}$ also need to be stored separately.

all the occurrences of Q producible from $join(\mathcal{R}_{I_1,\ldots,I_d})$ can already be produced from the boxes inspected earlier. The size of $\mathcal{B}$ can only decrease and therefore is still bounded by $O(N^{\rho^*}/\Delta)$. To find the occurrences, apply a worst-case optimal join algorithm on each box in $\mathcal{B}$. As each box generates at least one new occurrence, we guarantee a delay of $\tilde{O}(\Delta)$.

To support (Problem-2) queries with arbitrary q, use the adapted structure to replace that of [20] in the solution presented in Section 5.2. All the analysis still holds through. We thus complete the proof of Theorem 5.

Constant-Delay Enumeration for SLP-Compressed Documents

Martín Muñoz ✉
Pontificia Universidad Católica de Chile, Santiago, Chile
Millennium Institute for Foundational Research on Data, Santiago, Chile

Cristian Riveros ✉
Pontificia Universidad Católica de Chile, Santiago, Chile
Millennium Institute for Foundational Research on Data, Santiago, Chile

Abstract

We study the problem of enumerating results from a query over a compressed document. The model we use for compression are straight-line programs (SLPs), which are defined by a context-free grammar that produces a single string. For our queries we use a model called Annotated Automata, an extension of regular automata that allows annotations on letters. This model extends the notion of Regular Spanners as it allows arbitrarily long outputs. Our main result is an algorithm which evaluates such a query by enumerating all results with output-linear delay after a preprocessing phase which takes linear time on the size of the SLP, and cubic time over the size of the automaton. This is an improvement over Schmid and Schweikardt's result [25], which, with the same preprocessing time, enumerates with a delay which is logarithmic on the size of the uncompressed document. We achieve this through a persistent data structure named Enumerable Compact Sets with Shifts which guarantees output-linear delay under certain restrictions. These results imply constant-delay enumeration algorithms in the context of regular spanners. Further, we use an extension of annotated automata which utilizes succinctly encoded annotations to save an exponential factor from previous results that dealt with constant-delay enumeration over vset automata. Lastly, we extend our results in the same fashion Schmid and Schweikardt did [26] to allow complex document editing while maintaining the constant-delay guarantee.

2012 ACM Subject Classification Theory of computation → Database theory

Keywords and phrases SLP compression, query evaluation, enumeration algorithms

Digital Object Identifier 10.4230/LIPIcs.ICDT.2023.7

Funding This work was funded by ANID – Millennium Science Initiative Program – Code `ICN17_002`.

1 Introduction

A *constant-delay enumeration algorithm* is an efficient solution to an enumeration problem: given an instance of the problem, the algorithm performs a preprocessing phase to build some indices, to then continue with an enumeration phase where it retrieves each output, one by one, taking constant-delay between consecutive outcomes. These algorithms provide a strong guarantee of efficiency since a user knows that, after the preprocessing phase, she will access the output as if we have already computed them. For these reasons, constant-delay algorithms have attracted researchers' attention, finding sophisticated solutions to several query evaluation problems. Starting with Durand and Grandjean's work [14], researchers have found constant-delay algorithms for various classes of conjunctive queries [6, 10], FO queries over sparse structures [19, 27], and MSO queries over words and trees [5, 2].

The enumeration problem over documents (i.e., strings) has been studied extensively under the framework of document spanners [15]. A constant-delay algorithm for evaluating deterministic regular spanners was first presented in [16] and extended to non-deterministic in [3]. After these results, people have studied the enumeration problem of document spanners in the context of ranked enumeration [12, 8], nested documents [22], or grammars [23, 4].

Recently, Schmid and Schweikardt [25, 26] studied the evaluation problem for regular spanners over a document compressed by a Straight-line Program (SLP). In this setting, one encodes a document through a context-free grammar that produces a single string (i.e., the document itself). This mechanism allows highly compressible documents, in some instances allowing logarithmic space compared to the uncompressed copy. The enumeration problem consists now of evaluating a regular spanner over an SLP-compressed document. In [25], the authors provided a logarithmic-delay (over the uncompressed document) algorithm for the problem, and in [26], they extended this setting to edit operations over SLP documents, maintaining the delay. In particular, these works left open whether one can solve the enumeration problem of regular spanners over SLP-compressed documents with a constant-delay guarantee.

This paper aims to extend the understanding of the evaluation problem over SLP-compressed documents in several directions. First, we study the evaluation problem of *annotated automata* (AnnA) over SLP-compressed documents. These automata are a general model for defining regular enumeration problems, which strictly generalizes the model of extended variable-set automaton used in [25]. Second, we provide an output-linear delay enumeration algorithm for the problem of evaluating an unambiguous AnnA over an SLP-compressed document. In particular, this result implies a constant-delay enumeration algorithm for evaluating extended variable-set automaton, giving a positive answer to the open problem left in [25]. Third, we can show that we can extend this result to what we call a *succinctly* annotated automaton, a generalization of AnnA whose annotations are succinctly encoded by an enumeration scheme. We can develop an output-linear delay enumeration algorithm for this model, showing a constant-delay algorithm for sequential (non-extended) vset automata, strictly generalizing the work in [25]. Finally, we show that one can maintain these algorithmic results when dealing with complex document editing as in [26].

The main technical result is to show that the data structure presented in [22], called Enumerable Compact Set (ECS), can be extended to deal with shift operators (called Shift-ECS). This extension allows us to compactly represent the outputs and "shift" the results in constant time, which is to add or substract a common value to all elements in a set. Then, by using matrices with Shift-ECS nodes, we can follow a bottom-up evaluation of the annotated automaton over the grammar (similar to [25]) to enumerate all outputs with output-linear delay. The combination of annotated automata and Shift-ECSs considerably simplifies the algorithm presentation, reaching a better delay bound.

Organization of the paper. In Section 2 we introduce the setting and its corresponding enumeration problem. In Section 3, we present our data structure for storing and enumerating the outputs, and in Section 4 we show the evaluation algorithm. Section 5 offers the application of the algorithmic results to document spanners and Section 6 shows how to extend these results to deal with complex document editing. We finish the paper with future work in Section 7.

2 Setting and main problem

In this section, we present the setting and state the main result. First, we define straight-line programs, which we will use for the compressed representation of input documents. Then we introduce the definition of annotated automaton, an extension of regular automata to produce outputs. We use annotated automata as our computational model to represent queries over documents. By combining both formalisms, we state the main enumeration problem and main technical result.

Documents. Given a finite alphabet Σ, a document d over Σ (or just a document) is a string $d = a_1 a_2 \dots a_n \in \Sigma^*$. Given documents d_1 and d_2, we write $d_1 \cdot d_2$ (or just $d_1 d_2$) for the concatenation of d_1 and d_2. We denote by $|d| = n$ the length of the document $d = a_1 \dots a_n$ (i.e., the number of symbols) and by ε the document of length 0. We use Σ^* to denote the set of all documents, and Σ^+ for all documents with one or more symbols. To simplify the notation, in the sequel we will use d for a document, and a or b for a symbol in Σ.

SLP compression. A context-free grammar is a tuple $G = (N, \Sigma, R, S_0)$, where N is a non-empty set of non-terminals, Σ is finite alphabet, $S_0 \in N$ is the start symbol and $R \subseteq N \times (N \cup \Sigma)^+$ is the set of rules. As a convention, the rule $(A, w) \in R$ is commonly written as $A \to w$, and Σ and N are called terminal and non-terminal symbols, respectively. A context-free grammar $S = (N, \Sigma, R, S_0)$ is a *straight-line program* (SLP) if R is a total function from N to $(N \cup \Sigma)^+$ and the directed graph $(N, \{(A, B) \mid (A, w) \in R \text{ and } B \text{ appears in } w\})$ is acyclic. For every $A \in N$, let $R(A)$ be the unique $w \in (N \cup \Sigma)^+$ such that $(A, w) \in R$, and for every $a \in \Sigma$ let $R(a) = a$. We extend R to a morphism $R^* : (N \cup \Sigma)^* \to \Sigma^*$ recursively such that $R^*(d) = d$ when d is a document, and $R^*(\alpha_1 \dots \alpha_n) = R^*\big(R(\alpha_1) \cdot \ldots \cdot R(\alpha_n)\big)$, where $\alpha_i \in (N \cup \Sigma)$, for $i \leq n$. By our definition of SLP, $R^*(A)$ is in Σ^+, and uniquely defined for each $A \in N$. Then we define the document encoded by S as $\operatorname{doc}(S) = R^*(S_0)$.

▶ **Example 1.** Let $S = (N, \Sigma, R, S_0)$ be a SLP with $N = \{S_0, A, B\}$, $\Sigma = \{\mathsf{a}, \mathsf{b}, \mathsf{r}\}$, and $R = \{S_0 \to A\mathsf{r}BABA, A \to \mathsf{ba}, B \to A\mathsf{ra}\}$. We then have that $\operatorname{doc}(A) = \mathsf{ba}$, $\operatorname{doc}(B) = \mathsf{bara}$ and $\operatorname{doc}(S) = \operatorname{doc}(S_0) = \mathsf{barbarababaraba}$, namely, the string represented by S.

We define the size of an SLP $S = (N, \Sigma, R, S_0)$ as $|S| = \sum_{A \in N} |R(A)|$, namely, the sum of the lengths of the right-hand sides of all rules. It is important to note that an SLP S can encode a document $\operatorname{doc}(S)$ such that $|\operatorname{doc}(S)|$ is exponentially larger with respect to $|S|$. For this reason, SLPs stay as a commonly used data compression scheme [28, 20, 24, 11], and they are often studied in particular because of their algorithmic properties; see [21] for a survey. In this paper, we consider SLP compression to represent documents and use the formalism of annotated automata for extracting relevant information from the document.

Annotated automata. An *annotated automaton* (AnnA for short) is a finite state automaton where we label transitions with annotations. Formally, it is a tuple $\mathcal{A} = (Q, \Sigma, \Omega, \Delta, q_0, F)$ where Q is a state set, Σ is an input alphabet, Ω is an output alphabet, $q_0 \in Q$ and $F \subseteq Q$ are the initial state and final set of states, respectively, and: $\Delta \subseteq Q \times \Sigma \times Q \cup Q \times (\Sigma \times \Omega) \times Q$ is the transition relation, which contains *read transitions* of the form $(p, a, q) \in Q \times \Sigma \times Q$, and *read-write transitions* of the form $(p, (a, \sigma), q) \in Q \times (\Sigma \times \Omega) \times Q$.

Similarly to transducers [7], a symbol $a \in \Sigma$ is an input symbol that the machine reads and $\sigma \in \Omega$ is a symbol that indicates what the machine prints in an output tape. A run ρ of $\mathcal{A}$ over a document $d = a_1 a_2 \dots a_n \in \Sigma^*$ is a sequence of the form $\rho = q_0 \xrightarrow{b_1} q_1 \xrightarrow{b_2} \dots \xrightarrow{b_n} q_n$ such that for each $i \in [1, n]$ one of the following holds: (1) $b_i = a_i$ and there is a transition $(q_{i-1}, a_i, q_i) \in \Delta$ or (2) $b_i = (a_i, \sigma)$ and there is a transition $(q_{i-1}, (a_i, \sigma), q_i) \in \Delta$. We say that ρ is accepting if $q_n \in F$.

We define the *annotation* of ρ as $\mathsf{ann}(\rho) = \mathsf{ann}(b_1) \cdot \ldots \cdot \mathsf{ann}(b_n)$ such that $\mathsf{ann}(b_i) = (\sigma, i)$ if $b_i = (a, \sigma)$, and $\mathsf{ann}(b_i) = \varepsilon$ otherwise, for each $i \in [1, n]$. Given an annotated automaton $\mathcal{A}$ and a document $d \in \Sigma^*$, we define the set $[\![\mathcal{A}]\!](d)$ of all outputs of $\mathcal{A}$ over d as: $[\![\mathcal{A}]\!](d) = \{\mathsf{ann}(\rho) \mid \rho \text{ is an accepting run of } \mathcal{A} \text{ over } d\}$. Note that each output in $[\![\mathcal{A}]\!](d)$ is a sequence of the form $(\sigma_1, i_1) \dots (\sigma_k, i_k)$ for some $k \leq n$ where $i_1 < i_2 < \ldots < i_k$ and each (σ_j, i_j) means that position i_j is annotated with the symbol σ_j.

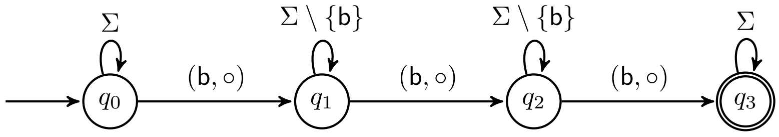


Figure 1 Example of an annotated automaton.

▶ **Example 2.** Consider an AnnA $\mathcal{A} = (Q, \Sigma, \Omega, \Delta, q_0, F)$ where $Q = \{q_0, q_1, q_2, q_3\}$, $\Sigma = \{\mathsf{a}, \mathsf{b}, \mathsf{r}\}$, $\Omega = \{\circ\}$, and $F = \{q_3\}$. We define Δ as the set of transitions that are depicted in Figure 1. For the document $d = \mathsf{barbarababaraba}$ the set $[\![A]\!](d)$ contains the results $(\circ, 1)(\circ, 4)(\circ, 8)$, $(\circ, 4)(\circ, 8)(\circ, 10)$ and $(\circ, 8)(\circ, 10)(\circ, 14)$. Intuitively, $\mathcal{A}$ selects triples of b which are separated by characters other than b.

Annotated automata are the natural regular counterpart of annotated grammars introduced in [4]. Moreover, it is the generalization and simplification of similar automaton formalisms introduced in the context of information extraction [15, 23], complex event processing [18, 17], and enumeration in general [8, 22]. In Section 5, we show how we can reduce the automaton model of document spanners, called a variable-set automaton, into a (succinctly) annotated automaton, generalizing the setting in [25].

As for other automata models, the notion of an unambiguous automaton is crucial for removing duplicate outputs. We say that an AnnA $\mathcal{A} = (Q, \Sigma, \Omega, \Delta, q_0, F)$ is *unambiguous* if for every $d \in \Sigma^*$ and every $\nu \in [\![\mathcal{A}]\!](d)$ there is exactly one accepting run ρ of $\mathcal{A}$ over d such that $\nu = \mathsf{ann}(\rho)$. On the other hand, we say that $\mathcal{A}$ is *deterministic* if Δ is a partial function of the form $\Delta : (Q \times \Sigma \cup Q \times (\Sigma \times \Omega)) \to Q$. Note that a deterministic AnnA is also unambiguous. The definition of unambiguous is in line with the notion of unambiguous annotated grammar [4] (see also [22]), and determinism with the idea of I/O-determinism used in [16, 8, 18]. As usual, one can easily show that for every AnnA $\mathcal{A}$ there exists an equivalent deterministic AnnA and, therefore, an equivalent unambiguous AnnA.

▶ **Lemma 3.** *For every annotated automaton $\mathcal{A}$ there exists a deterministic annotated automaton $\mathcal{A}'$ such that $[\![\mathcal{A}]\!](d) = [\![\mathcal{A}']\!](d)$ for every $d \in \Sigma^*$.*

Regarding the expressiveness of annotated automata, one can note that they have the same expressive power as MSO formulas with monadic second-order free variables. We refer the reader to [22] for an equivalent result in the context of nested documents.

Main problem and main result. We are interested in the problem of evaluating annotated automata over an SLP-compressed document, namely, to enumerate all the annotations over the document represented by an SLP. Formally, we define the main evaluation problem of this paper as follows. Let $\mathcal{C}$ be a class of AnnA (e.g. unambiguous AnnA).

Problem:	SLPEnum[$\mathcal{C}$]
Input:	an AnnA $\mathcal{A} \in \mathcal{C}$ and an SLP S
Output:	Enumerate $[\![\mathcal{A}]\!](\mathrm{doc}(S))$.

As it is common for enumeration problems, we want to impose an efficiency guarantee regarding the preprocessing of the input (e.g., $\mathcal{A}$ and S) and the delay between two consecutive outputs. For this purpose, one often divides the work of the enumeration algorithm into two phases: first, a *preprocessing phase* in which it receives the input and produces some object D (e.g., a collection of indices) which encodes the expected output and, second, an *enumeration phase* which extracts the results from D. We say that such an algorithm has f*-preprocessing time* if there exists a constant c such that, for every input $\mathcal{I}$, the time for the preprocessing phase of $\mathcal{I}$ is bounded by $c \cdot f(|\mathcal{I}|)$. Instead, we say that the algorithm has *output-linear delay* if there exists a constant d such that whenever the enumeration phase delivers the sequence of outputs $O_1, \ldots, O_\ell$, the time for producing the next output O_i is bounded by $c \cdot |O_i|$ for every $i \leq \ell$. As is expected, we assume here the computational model of Random Access Machines (RAM) with uniform cost measure and addition and subtraction as basic operations [1]. For a formal presentation of the output-linear delay guarantee, we refer the reader to [16]. As it is commonly done on algorithms over SLPs and other compression schemes, we assume that the registers in the underlying RAM-model allow for constant-time arithmetical operations over positions in the *uncompressed* document (i.e., they have $\mathcal{O}(\log |\operatorname{doc}(S)|)$ size).

The notion of output-linear delay is a refinement of the better-known constant-delay bound, which requires that each output has a constant size (i.e., concerning the input). Since even the document encoded by an SLP can be of exponential length, it is more reasonable in our setting to use the output-linear delay guarantee.

The following is the main technical result of this work.

▶ **Theorem 4.** *Let $\mathcal{C}$ be the class of all unambiguous AnnAs. Then one can solve the problem* SLPENUM$[\mathcal{C}]$ *with linear preprocessing time and output-linear delay. Specifically, there exists an enumeration algorithm that runs in $|\mathcal{A}|^3 \times |S|$-preprocessing time and output-linear delay for enumerating $[\![\mathcal{A}]\!](\operatorname{doc}(S))$ given an unambiguous AnnA $\mathcal{A}$ and a SLP S.*

We dedicate the rest of the paper to presenting the enumeration algorithm of Theorem 4. In Section 4 we explain the preprocessing phase of the algorithm. Before that, in the next section, we explain how Enumerable Compact Sets with Shifts work, which is the data structure used in the preprocessing phase to store outputs.

3 Enumerable compact sets with shifts

We present here the data structure, called Enumerable Compact Sets with Shifts, to compactly store the outputs of evaluating an annotated automaton over a straight-line program. This structure extends Enumerable Compact Sets (ECS) introduced in [22] (we note that a similar data structure for constant-delay enumeration was previously proposed in [2]). Indeed, people have also used ECS extensions in [4, 9]. This new version extends ECS by introducing a shift operator, which helps compactly move all outputs' positions with a single call. Although the shift nodes require a revision of the complete ECS model, it simplifies the evaluation algorithm in Section 4 and achieves output-linear delay for enumerating all outputs. For completeness of presentation, this section goes through all main details as in [22].

The structure. Let Ω be an output alphabet such that Ω has no elements in common with $\mathbb{Z}$ or $\{\cup, \odot\}$ (i.e., $\Omega \cap \mathbb{Z} = \emptyset$ and $\Omega \cap \{\cup, \odot\} = \emptyset$). We define an *Enumerable Compact Set with Shifts* (Shift-ECS) as a structure $\mathcal{D} = (\Omega, V, \ell, r, \lambda)$ such that V is a finite sets of nodes, $\ell\colon V \to V$ and $r\colon V \to V$ are the *left* and *right* partial functions, and $\lambda\colon V \to \Omega \cup \mathbb{Z} \cup \{\cup, \odot\}$ is a labeling function. We assume that $\mathcal{D}$ forms an acyclic graph (i.e., $(V, \{(v, \ell(v)), (v, r(v)) \mid v \in V\})$ is acyclic). Further, for every node $v \in V$, $\ell(v)$ is defined iff $\lambda(v) \in \mathbb{Z} \cup \{\cup, \odot\}$, and $r(v)$ is defined iff $\lambda(v) \in \{\cup, \odot\}$. Notice that, by definition, nodes labeled by Ω are bottom nodes in the acyclic structure formed by $\mathcal{D}$, and nodes labeled by $\mathbb{Z}$ or $\{\cup, \odot\}$ are inner nodes. Here, $\mathbb{Z}$-nodes are unary operators (i.e., $r(\cdot)$ is not defined over them), and $\cup$-nodes or $\odot$-nodes are binary operators. Indeed, we say that $v \in V$ is a *bottom node* if $\lambda(v) \in \Omega$, a *product node* if $\lambda(v) = \odot$, a *union node* if $\lambda(v) = \cup$, and a *shift node* if $\lambda(v) \in \mathbb{Z}$. Finally, we define $|\mathcal{D}| = |V|$.

The outputs retrieved from a Shift-ECS are strings of the form $(\mho_1, i_1)(\mho_2, i_2)\ldots(\mho_\ell, i_\ell)$, where $\mho_j \in \Omega$ and $i_j \in \mathbb{Z}$. To build them, we use the *shifting function* $\mathsf{sh} : (\Omega \times \mathbb{Z}) \times \mathbb{Z} \to (\Omega \times \mathbb{Z})$ such that $\mathsf{sh}((\mho, i), k) = (\mho, i + k)$. We extend this function to strings over $\Omega \times \mathbb{Z}$ such that $\mathsf{sh}((\mho_1, i_1)\ldots(\mho_\ell, i_\ell), k) = (\mho_1, i_1 + k)\ldots(\mho_\ell, i_\ell + k)$ and to set of strings such that $\mathsf{sh}(L, k) = \{\mathsf{sh}(\nu, k) \mid \nu \in L\}$ for every $L \subseteq (\Omega \times \mathbb{Z})^*$.

Each node $v \in V$ of a Shift-ECS $\mathcal{D} = (\Omega, V, \ell, r, \lambda)$ defines a set of output strings. Specifically, we associate a set of strings $[\![\mathcal{D}]\!](v)$ recursively as follows: (1) $[\![\mathcal{D}]\!](v) = \{(\mho, 1)\}$ whenever $\lambda(v) = \mho \in \Omega$, (2) $[\![\mathcal{D}]\!](v) = [\![\mathcal{D}]\!](\ell(v)) \cup [\![\mathcal{D}]\!](r(v))$ whenever $\lambda(v) = \cup$, (3) $[\![\mathcal{D}]\!](v) = [\![\mathcal{D}]\!](\ell(v)) \cdot [\![\mathcal{D}]\!](r(v))$ whenever $\lambda(v) = \odot$, where $L_1 \cdot L_2 = \{w_1 \cdot w_2 \mid w_1 \in L_1 \text{ and } w_2 \in L_2\}$, and (4) $[\![\mathcal{D}]\!](v) = \mathsf{sh}([\![\mathcal{D}]\!](\ell(v)), \lambda(v))$ whenever $\lambda(v) \in \mathbb{Z}$.

▶ **Example 5.** Suppose $\Omega = \{x, y\}$. Consider the Shift-ECS $\mathcal{D} = (\Omega, V, \ell, r, \lambda)$ where $V = \{v_1, v_2, v_3, v_4, v_5\}$, $\ell(v_1) = v_4$, $r(v_1) = v_2$, $\ell(v_2) = v_3$, $\ell(v_3) = v_4$, $r(v_3) = v_5$, $\lambda(v_1) = \odot$, $\lambda(v_2) = +2$, $\lambda(v_3) = \cup$, $\lambda(v_4) = x$ and $\lambda(v_5) = y$. We show this Shift-ECS in Figure 2 (a). The sets of words $[\![\mathcal{D}]\!]$ associated to each node are thus: $[\![\mathcal{D}]\!](v_4) = \{(x, 1)\}$, $[\![\mathcal{D}]\!](v_5) = \{(y, 1)\}$, $[\![\mathcal{D}]\!](v_3) = \{(x, 1), (y, 1)\}$, $[\![\mathcal{D}]\!](v_2) = \{(x, 3), (y, 3)\}$ and $[\![\mathcal{D}]\!](v_1) = \{(x, 1)(x, 3), (x, 1)(y, 3)\}$.

Enumeration. Given that every node of a Shift-ECS represents a set of strings, we are interested in enumerating them with output-linear delay. Specifically, we focus on the following problem. Let $\mathcal{C}$ be a class of Enumerable Compact Sets with Shifts.

Problem:	ShiftECSEnum[$\mathcal{C}$]
Input:	a Shift-ECS $\mathcal{D} \in \mathcal{C}$ and a node v of $\mathcal{D}$
Output:	Enumerate $[\![\mathcal{D}]\!](v)$.

The plan then is to provide an enumeration algorithm with output-linear delay for ShiftECSEnum[$\mathcal{C}$] and some helpful class $\mathcal{C}$. A reasonable strategy to enumerate the set $[\![\mathcal{D}]\!](v)$ is to do a traversal on the structure while accumulating the shift values in the path to each leaf. However, to be able to do this without repetitions and output-linear delay, we need to guarantee two conditions: first, that one can obtain every output from $\mathcal{D}$ in only one way and, second, union and shift nodes are *close* to an output node (i.e., a bottom node or a product node), in the sense that we can always reach them in a bounded number of steps. To ensure that these conditions hold, we impose two restrictions for an ECS.

For the first restriction, we say that an ECS $\mathcal{D}$ is *duplicate-free* if the following hold: (1) for every union node v in $\mathcal{D}$ it holds that $[\![\mathcal{D}]\!](\ell(v))$ and $[\![\mathcal{D}]\!](r(v))$ are disjoint and (2) for every product node v and for every $w \in [\![\mathcal{D}]\!](v)$, there exists a unique way to decompose $w = w_1 \cdot w_2$ such that $w_1 \in [\![\mathcal{D}]\!](\ell(v))$ and $w_2 \in [\![\mathcal{D}]\!](r(v))$.

For the second restriction, we define *k-bounded* Shift-ECS. Given a Shift-ECS $\mathcal{D}$, define the (left) output-depth of a node $v \in V$, denoted by $\mathsf{odepth}_{\mathcal{D}}(v)$, recursively as follows: $\mathsf{odepth}_{\mathcal{D}}(v) = 0$ whenever $\lambda(v) \in \{\odot\} \cup \Omega$, and $\mathsf{odepth}_{\mathcal{D}}(v) = \mathsf{odepth}_{\mathcal{D}}(\ell(v)) + 1$ whenever $\lambda(v) \in \{\cup\} \cup \mathbb{Z}$. Then, for $k \in \mathbb{N}$ we say that $\mathcal{D}$ is k-bounded if $\mathsf{odepth}_{\mathcal{D}}(v) \leq k$ for all $v \in V$.

▶ **Proposition 6.** *Fix $k \in \mathbb{N}$. Let $\mathcal{C}_k$ be the class of all duplicate-free and k-bounded Shift-ECSs. Then one can solve the problem ShiftECSEnum[$\mathcal{C}_k$] with output-linear delay and without preprocessing (i.e. constant preprocessing time).*

Operations. The next step is to provide a set of operations that allow extending a Shift-ECS $\mathcal{D}$ in a way that maintains k-boundedness. Fix a Shift-ECS $\mathcal{D} = (\Omega, V, \ell, r, \lambda)$. Then for any $\vartheta \in \Omega$, $v_1, \ldots, v_4, v \in V$ and $k \in \mathbb{Z}$, we define the operations:

$$\begin{aligned} \mathsf{add}(\vartheta) &\rightarrow v' & \qquad \mathsf{prod}(v_1, v_2) &\rightarrow v' \\ \mathsf{union}(v_3, v_4) &\rightarrow v' & \qquad \mathsf{shift}(v, k) &\rightarrow v' \end{aligned}$$

such that $[\![\mathcal{D}]\!](v') := \{(\vartheta, 1)\}$; $[\![\mathcal{D}]\!](v') := [\![\mathcal{D}]\!](v_1) \cdot [\![\mathcal{D}]\!](v_2)$; $[\![\mathcal{D}]\!](v') := [\![\mathcal{D}]\!](v_3) \cup [\![\mathcal{D}]\!](v_4)$; and $[\![\mathcal{D}]\!](v') := \mathsf{sh}([\![\mathcal{D}]\!](v), k)$, respectively. Here we assume that the union and prod respect properties (1) and (2) of a duplicate-free Shift-ECS, namely, $[\![\mathcal{D}]\!](v_3)$ and $[\![\mathcal{D}]\!](v_4)$ are disjoint and, for every $w \in [\![\mathcal{D}]\!](v_1) \cdot [\![\mathcal{D}]\!](v_2)$, there exists a unique way to decompose $w = w_1 \cdot w_2$ such that $w_1 \in [\![\mathcal{D}]\!](v_1)$ and $w_2 \in [\![\mathcal{D}]\!](v_2)$.

Strictly speaking, each operation above should receive as input the data structure $\mathcal{D}$, and output a fresh node v' plus a new data structure $\mathcal{D}' = (\Omega, V', \ell', r', \lambda')$ such that $\mathcal{D}'$ is an extension of $\mathcal{D}$, namely, $\mathsf{obj} \subseteq \mathsf{obj}'$ for every $\mathsf{obj} \in \{V, \ell, r, \lambda\}$ and $v' \in V' \setminus V$. Note that we assume that each operation can only extend the data structure with new nodes and that old nodes are immutable after each operation. For simplification, we will not explicitly refer to $\mathcal{D}$ on the operations above, although they modify $\mathcal{D}$ directly by adding new nodes.

To define the above operations, we impose further restrictions on the structure below the operations' input nodes to ensure k-boundedness. Towards this goal, we introduce the notion of safe nodes. We say that a node $v \in V$ is *safe* if v is a shift node and either $\ell(v)$ is an output node (i.e., a bottom or product node), or $u = \ell(v)$ is an union node, $\mathsf{odepth}_{\mathcal{D}}(u) = 1$, and $r(u)$ is a shift node with $\mathsf{odepth}_{\mathcal{D}}(r(u)) \leq 2$. In other words, v is safe if it is a shift node over an output node or over a union node with an output on the left and a shift node on the right, whose output depth is less or equal to 2. The trick then is to show that all operations over Shift-ECSs receive only safe nodes and always output safe nodes. As we will see, safeness will be enough to provide a light structural restriction on the operations' input nodes in order to maintain k-boundedness after each operation.

Next, we show how to implement each operation assuming that every input node is safe. In fact, the cases of add and shift are straightforward. For $\mathsf{add}(\vartheta) \rightarrow v'$ we extend $\mathcal{D}$ with two fresh nodes v' and u such that $\lambda(u) = \vartheta$, $\lambda(v') = 0$, and $\ell(v') = u$. In other words, we hang a fresh 0-shift node v' over a fresh ϑ-node u, and output v'. For $\mathsf{shift}(v, k) \rightarrow v'$, add the fresh node v' to $\mathcal{D}$, and set $\ell(v') = \ell(v)$ and $\lambda(v') = \lambda(v) + k$. One can easily check that in both cases the node v' represents the desired set, is safe, and k-boundedness is preserved.

To show how to implement $\mathsf{prod}(v_1, v_2) \rightarrow v'$, recall that v_1 and v_2 are safe and, in particular, both are shift nodes. Then we need to extend $\mathcal{D}$ with fresh nodes v', v'', and v''' such that $\ell(v') = v''$, $\ell(v'') = \ell(v_1)$, $r(v'') = v'''$, $\ell(v''') = \ell(v_2)$, $\lambda(v') = \lambda(v_1)$, $\lambda(v'') = \odot$ and $\lambda(v''') = \lambda(v_2) - \lambda(v_1)$. Figure 2(b) shows a diagram of this gadget. One can easily check that v' represents the product of v_1 and v_2, v' is safe, and the new version of $\mathcal{D}$ is k-bounded whenever $\mathcal{D}$ is also k-bounded.

The last operation is $\mathsf{union}(v_3, v_4) \rightarrow v'$. For the sake of presentation, we only provide the construction for the most involved case, which is when both $u_3 = \ell(v_3)$ and $u_4 = \ell(v_4)$ are union nodes. We show the gadget for this case in Figure 2(c). This construction has several interesting properties. First, one can check that $[\![\mathcal{D}]\!](v') = [\![\mathcal{D}]\!](v_3) \cup [\![\mathcal{D}]\!](v_4)$ since each shift value is carefully constructed so that the accumulated shift value from v' to each node remains unchanged. Thus, the semantics is well-defined. Second, union can be computed in constant time in $|\mathcal{D}|$ given that we only need to add a fixed number of fresh nodes. Furthermore, the produced node v' is safe, although some of the new nodes are not necessarily safe. Finally, the new $\mathcal{D}$ is 3-bounded whenever $\mathcal{D}$ is 3-bounded. To see this, we

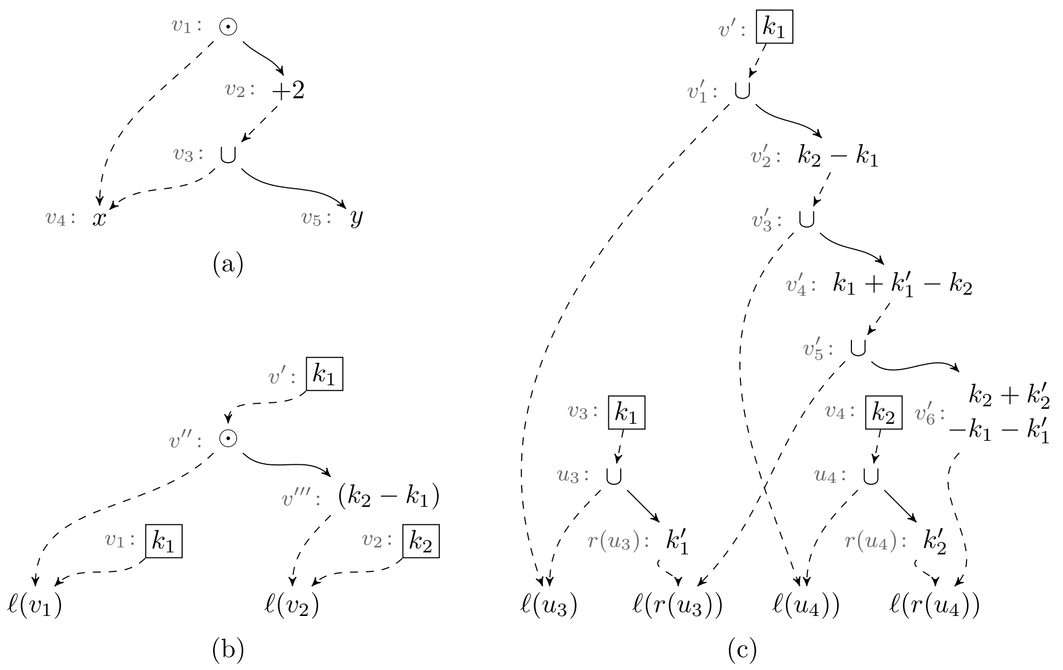


Figure 2 (a) An example of a Shift-ECS with output alphabet $\{x, y\}$. (b) Gadget for $\mathsf{prod}(\mathcal{D}, v_1, v_2, k)$. (c) Gadget for $\mathsf{union}(\mathcal{D}, v_3, v_4)$. We use dashed and solid edges for the left and right mappings, respectively. Node names are in grey at the left of each node. In (b) and (c), square nodes are the input and output nodes of each operation.

first have to notice that $\ell(u_3)$ and $\ell(u_4)$ are output nodes, and that $\mathsf{odepth}(\ell(r(u_3))) \leq 1$ and $\mathsf{odepth}(\ell(r(u_4))) \leq 1$. We can check the depth of each node going from the bottom to the top: $\mathsf{odepth}(v'_6) \leq 2$, $\mathsf{odepth}(v'_5) \leq 2$, $\mathsf{odepth}(v'_4) \leq 3$, $\mathsf{odepth}(v'_3) \leq 1$, $\mathsf{odepth}(v'_2) \leq 2$, $\mathsf{odepth}(v'_1) \leq 1$ and $\mathsf{odepth}(v') \leq 2$.

By the previous discussion, if we start with a Shift-ECS $\mathcal{D}$ which is 3-bounded (in particular, empty) and we apply the add, prod, union and shift operators between safe nodes (which also produce safe nodes), then the result is 3-bounded as well. Furthermore, the data structure is fully-persistent [13]: for every node v in $\mathcal{D}$, $[\![\mathcal{D}]\!](v)$ is immutable after each operation. Finally, by Proposition 6, the result can be enumerated with output-linear delay.

▶ **Theorem 7.** *The operations* add*,* prod*,* union *and* shift *take constant time and are fully persistent. Furthermore, if we start from an empty Shift-ECS $\mathcal{D}$ and apply these operations over safe nodes, the result node v' is always a safe node and the set $[\![\mathcal{D}]\!](v)$ can be enumerated with output-linear delay (without preprocessing) for every node v.*

The empty- and ε-nodes. The last step of constructing our model of Shift-ECS is the inclusion of two special nodes that produce the empty set and the empty string, called empty- and ε-nodes, respectively.

We start with the empty node, which is easier to incorporate into a Shift-ECS. Consider a special node $\perp$ and include it on every Shift-ECS $\mathcal{D}$, such that $[\![\mathcal{D}]\!](\perp) = \emptyset$. Then extend the operations prod, union, and shift accordingly to the empty set, namely, $\mathsf{prod}(v_1, v_2) \to \perp$ whenever v_1 or v_2 is equal to $\perp$, $\mathsf{union}(v, \perp) = \mathsf{union}(\perp, v) \to v$, and $\mathsf{shift}(\perp, k) \to \perp$ for every nodes v_1, v_2, v, and $k \in \mathbb{Z}$. It is easy to check that one can include the $\perp$-node into Shift-ECSs without affecting the guarantees of Theorem 7.

The other special node is the ε-node. Let ε denote a special node, included on every Shift-ECS $\mathcal{D}$, such that $[\![\mathcal{D}]\!](\varepsilon) = \{\varepsilon\}$. With these new nodes in a Shift-ECS, we need to revise our notions of output-depth, duplicate-free, and k-boundedness to change the enumeration algorithm, and to extend the operations add, prod, union, and shift over so-called ε-safe nodes (i.e., the extension of safe nodes with ε). Given space restrictions, we omit the details on how to implement these ε-nodes and how we can preserve Proposition 6 and Theorem 7.

For the rest of the paper, we assume that a Shift-ECS is a tuple $\mathcal{D} = (\Omega, V, \ell, r, \lambda, \bot, \varepsilon)$ where we define $\Omega, V, \ell, r, \lambda$ as before, and $\bot, \varepsilon \in V$ are the empty and ε nodes, respectively. Further, we assume that ℓ, r, and λ are extended accordingly, namely, $\ell(v)$ and $r(v)$ are not defined whenever $v \in \{\bot, \varepsilon\}$, and $\lambda : V \to \Omega \cup \mathbb{Z} \cup \{\cup, \odot, \bot, \varepsilon\}$ such that $\lambda(v) = \bot$ ($\lambda(v) = \varepsilon$) iff $v = \bot$ ($v = \varepsilon$, resp.). Finally, we can extend Theorem 7 for the Shift-ECS extension as follows.

▶ **Theorem 8.** *The operations* add, prod, union *and* shift *over Shift-ECS extended with bot- and ε-nodes take constant time. Furthermore, if we start from an empty Shift-ECS $\mathcal{D}$ and apply* add, prod, union, *and* shift *over ε-safe nodes, the resulting node v' is always an ε-safe node, and the set $[\![\mathcal{D}]\!](v)$ can be enumerated with output-linear delay without preprocessing for every node v.*

4 Evaluation of annotatated automata over SLP-compressed strings

This section shows our algorithm for evaluating an annotated automaton over an SLP-compressed document. This evaluation is heavily inspired by the preprocessing phase in [25], as it primarily adapts the algorithm to the Shift-ECS data structure. In a nutshell, we keep matrices of Shift-ECS nodes, where each matrix represents the outputs of all partial runs of the annotated automaton over fragments of the compressed strings. We extend the operations of Shift-ECS over matrices of nodes, which will allow us to compose matrices, and thus compute sequences of compressed strings. Then the algorithm proceeds in a dynamic programming fashion, where matrices are computed bottom-up for each non-terminal symbol. Finally, the start symbol of the SLP will contain all the outputs. The result of this process is that each matrix entry succinctly represents an output set, can be operated in constant time, and can be enumerated with output-linear delay.

Matrices of nodes. The main ingredient for the evaluation algorithm are matrices of nodes for encoding partial runs of annotated automata. To formalize this notion, fix an unambiguous AnnA $\mathcal{A} = (Q, \Sigma, \Omega, \Delta, q_0, F)$ and a Shift-ECS $\mathcal{D} = (\Omega, V, \ell, r, \lambda, \bot, \varepsilon)$. We define a *partial run* ρ of $\mathcal{A}$ over a document $d = a_1 a_2 \dots a_n \in \Sigma^*$ as a sequence $\rho = p_0 \xrightarrow{b_1} \dots \xrightarrow{b_n} p_n$ such that either $b_i = a_i$ and $(q_{i-1}, a_i, q_i) \in \Delta$, or $b_i = (a_i, o)$ and $(q_{i-1}, (a_i, o), q_i) \in \Delta$. Additionally, we say that the partial run ρ is from state p to state q if $p_0 = p$ and $p_n = q$. In other words, partial runs are almost equal to runs, except they can start at any state p.

For the algorithm, we use the set of all $Q \times Q$ matrices where entry $M[p, q]$ is a node in V for every $p, q \in Q$. Intuitively, each node $M[p, q]$ represents all outputs $[\![\mathcal{D}]\!](M[p, q])$ of partial runs from state p to state q, which can be enumerated with output-linear delay by Theorem 8. Intuitively, $M[p, q] = \bot$ represents that there is no run from p to q, and $M[p, q] = \varepsilon$ represents that there is a single run without outputs.

To combine matrices over $\mathcal{D}$-nodes, we define two operations. The first operation is the *matrix multiplication* over the semiring $(2^{(\Omega \times \mathbb{Z})^*}, \cup, \cdot, \emptyset, \{\varepsilon\})$ but represented over $\mathcal{D}$. Formally, let $Q = \{q_0, \dots, q_{m-1}\}$ with $m = |Q|$. Then, for two $m \times m$ matrices M_1 and M_2, we define $M_1 \otimes M_2$ such that for every $p, q \in Q$:

Algorithm 1 The enumeration algorithm of an unambiguous AnnA $\mathcal{A} = (Q, \Sigma, \Omega, \Delta, q_0, F)$ over an SLP $S = (N, \Sigma, R, S_0)$.

```
 1: procedure EVALUATION($\mathcal{A}, S$)
 2:     Initialize $\mathcal{D}$ as an empty Shift-ECS
 3:     NONTERMINAL($S_0$)
 4:     $v \leftarrow \perp$
 5:     for each $q \in F$ do
 6:         $v \leftarrow \mathsf{union}(v, M_{S_0}[q_0, q])$
 7:     ENUMERATE($v, \mathcal{D}$)

 8: procedure TERMINAL($a$)
 9:     $M_a \leftarrow \{[p, q] \rightarrow \perp \mid p, q \in Q\}$
10:     for each $(p, (a, \vartheta), q) \in \Delta$ do
11:         $M_a[p, q] \leftarrow \mathsf{union}(M_a[p, q], \mathsf{add}(\vartheta))$
12:     for each $(p, a, q) \in \Delta$ do
13:         $M_a[p, q] \leftarrow \mathsf{union}(M_a[p, q], \varepsilon)$
14:     $\mathrm{len}_a \leftarrow 1$

15: procedure NONTERMINAL($X$)
16:     $M_X \leftarrow \{[p, q] \rightarrow \perp \mid p, q \in Q, p \neq q\} \cup$
                 $\{[p, q] \rightarrow \varepsilon \mid p, q \in Q, p = q\}$
17:     $\mathrm{len}_X \leftarrow 0$
18:     for $i = 1$ to $|R(X)|$ do
19:         $Y \leftarrow R(X)[i]$
20:         if $M_Y$ is not defined then
21:             if $Y \in \Sigma$ then
22:                 TERMINAL($Y$)
23:             else
24:                 NONTERMINAL($Y$)
25:         $M_X \leftarrow M_X \otimes \mathsf{shift}(M_Y, \mathrm{len}_X)$
26:         $\mathrm{len}_X \leftarrow \mathrm{len}_X + \mathrm{len}_Y$
```

$$(M_1 \otimes M_2)[p, q] := \mathsf{union}_{i=0}^{m-1}\Big(\mathsf{prod}\big(M_1[p, q_i], M_2[q_i, q]\big)\Big)$$

where $\mathsf{union}_{i=0}^{m-1} E_i := \mathsf{union}(\ldots \mathsf{union}(\mathsf{union}(E_1, E_2), E_3) \ldots, E_m)$. In other words, the node $(M_1 \otimes M_2)[p, q]$ represents the set $\bigcup_{i=0}^{m-1} \big([\![\mathcal{D}]\!](M_1[p, q_i]) \cdot [\![\mathcal{D}]\!](M_2[q_i, q])\big)$.

The second operation for matrices is the extension of the *shift operation*. Formally, $\mathsf{shift}(M, k)[p, q] := \mathsf{shift}(M[p, q], k)$ for a matrix M, $k \in \mathbb{Z}$, and $p, q \in Q$. Since each operation over $\mathcal{D}$ takes constant time, overall multiplying M_1 with M_2 takes time $\mathcal{O}(|Q|^3)$ and shifting M with k takes time $\mathcal{O}(|Q|^2)$.

The algorithm. We present the evaluation algorithm for the SLPENUM problem in Algorithm 1. As expected, the main procedure EVALUATION receives as input an unambiguous annotated automaton $\mathcal{A} = (Q, \Sigma, \Omega, \Delta, q_0, F)$ and an SLP $S = (N, \Sigma, R, S_0)$, and enumerates all outputs in $[\![\mathcal{A}]\!](\mathrm{doc}(S))$. To simplify the notation, in Algorithm 1 we assume that $\mathcal{A}$ and S are globally defined, and we can access them in any subprocedure. Similarly, we use a Shift-ECS $\mathcal{D}$, and matrix M_X and integer len_X for every $X \in N \cup \Sigma$, which can globally be accessed at any place as well.

The main purpose of the algorithm is to compute M_X and len_X. On one hand, M_X is a $Q \times Q$ matrix where each node entry $M_X[p, q]$ represents all outputs of partial runs from p to q. On the other hand, len_X is the length of the string $R^*(X)$ (i.e., the string produced from X). Both M_X and len_X start undefined, and we compute them recursively, beginning from the non-terminal symbol S_0 and by calling the method NONTERMINAL(S_0) (line 3). After M_{S_0} was computed, we can retrieve the set $[\![\mathcal{A}]\!](S)$ by taking the union of all partial run's outputs from the initial state q_0 to a state $q \in F$, and storing it in node v (lines 4–6). Finally, we can enumerate $[\![\mathcal{A}]\!](S)$ by enumerating all outputs represented by v (line 7).

The workhorses of the evaluation algorithm are procedures NONTERMINAL and TERMINAL in Algorithm 1. The former computes matrices M_X recursively whereas the latter is in charge of the base case M_a for a terminal $a \in \Sigma$. For computing this base case, we can start with M_a with all entries equal to the empty node $\perp$ (line 9). Then if there exists a read-write transition $(p, (a, \vartheta), q) \in \Delta$, we add an output node ϑ to $M_a[p, q]$, by making the union between the current node at $M_a[p, q]$ with the node $\mathsf{add}(\delta)$ (line 11). Also, if a read transition $(p, a, q) \in \Delta$ exists, we do the same but with the ε-node (line 13). Finally, we set the length of len_a to 1, and we have covered the base case.

For the recursive case (i.e., procedure NonTerminal(X)), we start with a sort of "identity matrix" M_X where all entries are set up to the empty-node except the ones where $p = q$ that are set up to the ε-node, and the value $\text{len}_X = 0$ (lines 16–17). Then we iterate sequentially over each symbol Y of $R(X)$, where we use $R(X)[i]$ to denote the i-th symbol of $R(X)$ (lines 18–19). If M_Y is not defined, then we recursively compute Terminal(Y) or NonTerminal(Y) depending on whether Y is in Σ or not, respectively (lines 20–24). The matrix M_Y is memoized (by having the check in line 20 to see if it is defined or not) so we need to compute it at most once. After we have retrieved M_Y, we can compute all outputs for $R(X)[1] \dots R(X)[i]$ by multiplying the current version of M_X (i.e., the outputs of $R(X)[1] \dots R(X)[i - 1]$), with the matrix M_Y shifted by the current length len_X (line 25). Finally, we update the current length of X by adding len_Y (line 26).

Regarding correctness, the algorithm follows a direct matrix evaluation over the SLP grammar, where its correctness depends on the Shift-ECS $\mathcal{D}$. Notice that, although all operations over nodes are not necessarily duplicate-free, we know that the runs from the initial state q_0 to the final states are unambiguous. Then the operations used for the final output are duplicate-free. Regarding performance, the main procedure calls NonTerminal or Terminal at most once for every symbol. Note that after making all calls to Terminal, each transition in Δ is seen exactly once, and NonTerminal takes time at most $\mathcal{O}(|R(X)| \times |Q|^3)$ not taking into account the calls inside. Overall, the preprocessing time is $\mathcal{O}(|\mathcal{A}| + |S| \times |Q|^3)$.

▶ **Theorem 9.** *Algorithm 1 enumerates the set $[\![\mathcal{A}]\!](S)$ correctly for every unambiguous AnnA $\mathcal{A}$ and every SLP-compressed document S, with output-linear delay and after a preprocessing phase that takes time $\mathcal{O}(|\mathcal{A}| + |S| \times |Q|^3)$.*

We want to finish by noticing that, contrary to [25], our evaluation algorithm does not need to modify the grammar S into Chomsky's normal form (CNF) since we can evaluate $\mathcal{A}$ over S directly. Although passing S into CNF can be done in linear time over S [25], this step can incur an extra cost, which we can avoid in our approach.

5 Applications in regular spanners

It was already shown in [4] that working with annotations directly and then providing a reduction from a spanner query to an annotation query is sometimes more manageable. In this section we will do just that: starting from a document-regular spanner pair $(d, \mathcal{M})$, we will show how to build a document-annotated automaton pair $(d', \mathcal{A})$ such that $\mathcal{M}(d) = [\![\mathcal{A}]\!](d')$. Although people have studied various models of regular spanners in the literature, we will focus here on sequential variable-set automata (VA) [15] and sequential extended VA [16]. The latter, which we handle first, is essentially the model that the work of Schmid and Schweikardt used in their results. In the second half of the section we reduce the former to *succinctly* annotated automata, an extension of AnnA that allows output symbols to be stored concisely. These reductions imply constant-delay enumeration for the spanner tasks.

Variable-set automata. Consider a document $d = a_1 \dots a_n$ over an input alphabet Σ. A *span* of d is a pair $[i, j\rangle$ with $1 \leq i \leq j \leq n + 1$. We define the substring of $[i, j\rangle$ by $d[i, j\rangle = a_i \dots a_{j-1}$. We also consider a finite set of variables $\mathcal{X}$ and we define a *mapping* as a partial function that maps some of these variables to spans. We define a *document spanner* as a function assigning every input document d to a set of mappings [15].

A *variable-set automaton* (VA for short) is a tuple $\mathcal{A} = (Q, \Sigma, \mathcal{X}, \Delta, q_0, F)$ where Q is a set of states, $q_0 \in Q$, $F \subseteq Q$, and Δ consists of *read transitions* $(p, a, q) \in Q \times \Sigma \times Q$ and *variable transitions* $(p, \vdash^x, q)$ or $(p, \dashv^x, q)$ where $p, q \in Q$ and $x \in \mathcal{X}$. The symbols $\vdash^x$ and $\dashv^x$ are referred to as *variable markers* of x, where $\vdash^x$ is *opening* and $\dashv^x$ is *closing*. Given a document $d = a_1 \dots a_n \in \Sigma^*$ a configuration of $\mathcal{A}$ is a pair (q, i) where $q \in Q$ and $i \in [1, n+1]$. A run ρ of $\mathcal{A}$ over d is a sequence $\rho = (q_0, i_0) \xrightarrow{\sigma_1} (q_1, i_1) \xrightarrow{\sigma_2} \cdots \xrightarrow{\sigma_m} (q_m, i_m)$ where $i_0 = 1$, $i_m = n+1$, and for each $j \in [0, m-1]$, $(q_j, \sigma_{j+1}, q_{j+1}) \in \Delta$ and either (1) $\sigma_{j+1} = a_{i_j}$ and $i_{j+1} = i_j + 1$, or (2) $\sigma_{j+1} \in \{\vdash^x, \dashv^x \mid x \in \mathcal{X}\}$ and $i_{j+1} = i_j$. We say that ρ is *accepting* if $q_m \in F$ and that it is *valid* if variables are non-repeating, and they are opened and closed correctly. If ρ is accepting and valid, we define the mapping μ^ρ which maps $x \in \mathcal{X}$ to the span $[i_j, i_k\rangle$ iff $\sigma_j = \vdash^x$ and $\sigma_k = \dashv^x$. We say that $\mathcal{A}$ is *sequential* if every accepting run is also valid. Finally, define the document spanner $[\![\mathcal{A}]\!]$ as the function $[\![\mathcal{A}]\!](d) = \{\mu^\rho \mid \rho$ is an accepting and valid run of $\mathcal{A}$ over $d\}$. Like in AnnAs, we say $\mathcal{A}$ is *unambiguous* if for each mapping $\mu \in [\![\mathcal{A}]\!](d)$ there is exactly one accepting run ρ of $\mathcal{T}$ over d such that $\mu^\rho = \mu$.

Extended VA. For the sake of presentation, we will skip a formal definition for extended VA, and we refer the reader to [16]. These are automata in which the transitions read either letters in Σ or sets of markers from $\{\vdash^x, \dashv^x \mid x \in \mathcal{X}\}$. Runs are defined as sequences which alternate between transitions that read letters and sets, and a mapping associated to a run is defined as one would expect, where $x \in \mathcal{X}$ is mapped to the span $[i, j\rangle$ iff $\vdash^x$ is in the i-th set of the run, and $\dashv^x$ is in the j-th set in the run. We define sequential and unambiguous extended VA analogously to VA.

To illustrate the reduction from sequential extended VA to annotated automata, consider a document $d = \texttt{aab}$, and a run of some extended VA with variable set $\mathcal{X} = \{x, y\}$ over d:

$$\rho = q_0 \xrightarrow{\emptyset} q_0 \xrightarrow{\texttt{a}} q_1 \xrightarrow{\{\vdash^x, \dashv^x, \vdash^y\}} p_1 \xrightarrow{\texttt{a}} q_2 \xrightarrow{\emptyset} q_2 \xrightarrow{\texttt{b}} q_3 \xrightarrow{\{\dashv^y\}} p_3$$

This run defines the mapping μ which assigns $\mu(x) = [2, 2\rangle$ and $\mu(y) = [2, 4\rangle$. To translate this run to the annotated automata model, first we append an end-of-document character to d, and then we "push" the marker sets one transition to the right. We then obtain a run of some annotated automaton with output set $\Omega = 2^{\{\vdash^x, \dashv^x \mid x \in \mathcal{X}\}}$ over the document $d' = \texttt{aab\#}$:

$$\rho' = q_0' \xrightarrow{\texttt{a}} q_1' \xrightarrow{(\texttt{a}, \{\vdash^x, \dashv^x, \vdash^y\})} q_2' \xrightarrow{\texttt{b}} q_3' \xrightarrow{(\texttt{\#}, \{\dashv^y\})} q_4'$$

The annotation of this run would then be $\nu = (2, \{\vdash^x, \dashv^x, \vdash^y\})(4, \{\dashv^y\})$, from where the mapping μ can be extracted directly. The reduction from extended VA into annotated automata operates in a similar fashion: the read transitions are kept, and for each pair of transitions $(p, S, q), (q, a, r)$ in the former, a transition $(p, (a, S), r)$ is added to the latter.

The equivalence between mappings and annotations is formally defined as follows: For some document d of size n, a mapping μ from $\mathcal{X}$ to spans in d is equivalent to an annotation $\nu = (S_1, i_1) \dots (S_m, i_m)$ iff $S_j = \{\vdash^x \mid \mu(x) = [i_j, k\rangle\} \cup \{\dashv^x \mid \mu(x) = [k, i_j\rangle\}$ for every $j \leq m$.

▶ **Proposition 10.** *For any unambiguous sequential extended VA $\mathcal{A}$ with state set Q and transition set Δ, there exists an AnnA $\mathcal{A}'$ of size $\mathcal{O}(|Q| \times |\Delta|)$ such that for every document d, each mapping $\mu \in [\![A]\!](d)$ is equivalent to some unique $\nu \in [\![\mathcal{A}']\!](d\texttt{\#})$ and vice versa.*

Combining Proposition 10 and Theorem 4, we get a constant-delay algorithm for evaluating an unambiguous sequential extended VA over a document, proving the extension of the result in [25]. Notice that the result in [25] is for *deterministic* VA, where here we generalize this result for the unambiguous case plus constant-delay.

Succinctly annotated automata. For the next result, we need an extension to annotated automata which features succinct representations of sets of annotations.

A *succinct enumerable representation scheme* (SERS) is a tuple $\mathcal{S} = (\mathcal{R}, \Omega, |\cdot|, \mathcal{L}, \mathcal{E})$ made of an infinite set of representations $\mathcal{R}$, and an infinite set of annotations Ω. It includes a function $|\cdot|$ that indicates, for each $r \in \mathcal{R}$ and $\vartheta \in \Omega$, the sizes $|r|$ and $|\vartheta|$, i.e., the number of units needed to store r and ϑ in the underlying computational model (e.g. the RAM model). The function $\mathcal{L}$ maps each element $r \in \mathcal{R}$ to some finite non-empty set $\mathcal{L}(r) \subseteq \Omega$. Lastly, there is an algorithm $\mathcal{E}$ which enumerates the set $\mathcal{L}(r)$ with output-linear delay for every $r \in \mathcal{R}$. Intuitively, a SERS provides us with representations to encode sets of annotations. Moreover, there is the promise of the enumeration algorithm $\mathcal{E}$ where we can recover all the annotations with output-linear delay. This representation scheme allows us to generalize the notion of annotated automaton for encoding an extensive set of annotations in the transitions.

Fix a SERS $\mathcal{S} = (\mathcal{R}, \Omega, |\cdot|, \mathcal{L}, \mathcal{E})$. A Succinctly Annotated Automaton over $\mathcal{S}$ (sAnnA for short) is a tuple $\mathcal{T} = (Q, \Sigma, \Omega, \Delta, q_0, F)$ where all sets are defined like in AnnA, except that in Δ read-write transitions are of the form $(p, (a, r), q) \in Q \times (\Sigma \times \mathcal{R}) \times Q$. That is, transitions are now annotated by a representation r which encodes *sets* of annotations in Ω. For a read-write transition $t = (p, (a, r), q)$, we define its size as $|t| = |r| + 1$ and for a read transition $t = (p, a, q)$ we define its size as $|t| = 1$. A run ρ over a document $d = a_1 \dots a_n$ is also defined as a sequence $\rho = q_0 \xrightarrow{b_1} q_1 \xrightarrow{b_2} \dots \xrightarrow{b_n} q_n$ with the same specifications as in AnnA with the difference that it either holds that $b_i = a_i$, or $b_i = (a_i, r)$ for some representation r. We now define the *set of annotations* of ρ as: $\mathsf{ann}(\rho) = \mathsf{ann}(b_1, 1) \cdot \ldots \cdot \mathsf{ann}(b_n, n)$ such that $\mathsf{ann}(b_i, i) = \{(\vartheta, i) \mid \vartheta \in \mathcal{L}(r)\}$, if $b_i = (a, r)$, and $\mathsf{ann}(b_i, i) = \{\varepsilon\}$ otherwise. The set $[\![\mathcal{T}]\!](d)$ is defined as the union of sets $\mathsf{ann}(\rho)$ for all accepting runs ρ of $\mathcal{T}$ over d. In this model, we say that $\mathcal{T}$ is unambiguous if for every document d and every annotation $\nu \in [\![\mathcal{T}]\!](d)$ there exists only one accepting run ρ of $\mathcal{T}$ over d such that $\nu \in \mathsf{ann}(\rho)$. Finally, we define the size of Δ as $|\Delta| = \sum_{t \in \Delta} |t|$, and the size of $\mathcal{T}$ as $|\mathcal{T}| = |Q| + |\Delta|$.

This annotated automata extension allows for representing output sets more compactly. Moreover, given that we can enumerate the set of annotations with output-linear delay, we can compose it with Theorem 4 to get an output-linear delay algorithm for the whole set.

▶ **Theorem 11.** *Fix a SERS $\mathcal{S}$. There exists an enumeration algorithm that, given an unambiguous sAnnA $\mathcal{T}$ over $\mathcal{S}$ and an SLP S, it runs in $|\mathcal{T}|^3 \times |S|$-preprocessing time and output-linear delay for enumerating $[\![\mathcal{T}]\!](\mathrm{doc}(S))$.*

The purpose of sAnnA is to encode sequential VA succinctly. Indeed, as shown in [16], representing sequential VA with extended VA has an exponential blow-up in the number of variables that cannot be avoided. Therefore, the reduction from Proposition 10 cannot work directly. Instead, we can use a Succinctly Annotated Automaton over some specific SERS to translate every sequential VA into the annotation world efficiently.

▶ **Proposition 12.** *There exists an SERS $\mathcal{S}$ such that for any unambiguous sequential VA $\mathcal{A}$ with state set Q and transition set Δ there exists a sAnnA $\mathcal{T}$ over $\mathcal{S}$ of size $\mathcal{O}(|Q| \times |\Delta|)$ such that for every document d, each mapping $\mu \in [\![\mathcal{A}]\!](d)$ is equivalent to some unique $\nu \in [\![\mathcal{A}]\!](d\#)$ and vice versa. Furthermore, the number of states in $\mathcal{T}$ is in $\mathcal{O}(|Q|)$.*

By Proposition 12 and Theorem 11 we prove the extension of the output-linear delay algorithm for unambiguous sequential VA.

6 Constant delay-preserving complex document editing

In this section, we show that the results obtained by Schmid and Schweikardt [26] regarding enumeration over document databases and complex document editing still hold, maintaining the same time bounds in doing these edits, but allowing output-linear delay. We also include a refinement of the result for whenever the edits needed are limited to the concatenation of two documents. To be precise, we will give an overview of the following theorem.

▶ **Theorem 13.** *Let $D = \{d_1, \dots, d_m\}$ be a document database that is represented by an SLP S in normal form. Let $\mathcal{A}_1, \dots, \mathcal{A}_k$ be unambiguous sequential variable-set automata. When given the query data structures for S and $\mathcal{A}_1, \dots, \mathcal{A}_k$, and a CDE-expression φ over D, we can construct an extension S' of S and new query data structures for S' and $\mathcal{A}_1, \dots, \mathcal{A}_k$, and a new non-terminal $\tilde{A}$ of S', such that $\text{doc}(\tilde{A}) = \mathsf{eval}(\varphi)$.*

- *If φ contains operations other than concat, we require S to be strongly balanced. Then, S' is also strongly balanced, and this construction can be done in time $\mathcal{O}(k \cdot |\varphi| \cdot \log |d^*|)$ with data complexity where $|d^*| = |\max_\varphi(D)|$.*
- *If φ only contains concat, then this can be done in $\mathcal{O}(k \cdot |\varphi|)$ with data-complexity.*

Afterwards, upon input of any $d \in \mathsf{docs}(S')$ (represented by a non-terminal of S') and any $i \in [1, m]$, the set $[\![\mathcal{A}_i]\!](d)$ can be enumerated with constant-delay.

Note that a similar result was proved in [26] but with extended VA instead of VA, and with logarithmic delay instead of constant-delay. The rest of this section will be dedicated to define the concepts we have not yet introduced, and show how the techniques presented in [26] allow us to obtain this result.

Normal form, balanced and rootless SLPs. We define a *rootless SLP* as a triple $S = (N, \Sigma, R)$, where N is a set of non-terminals, Σ is the set of terminals, and R is a set of rules. Rootless SLPs are defined as SLPs with the difference that there is no starting symbol, and thus $\text{doc}(S)$ is not defined. Instead, we define $\text{doc}(A)$ for each $A \in N$ as $\text{doc}(A) = R^*(A)$. We say that S is in *Chomsky normal form* if every rule in R has the form $A \to a$ or $A \to BC$, where $a \in \Sigma$ and $A, B, C \in N$. Also, we say that S is *strongly balanced* if for each rule $A \to BC$, the value $\mathsf{ord}(B) - \mathsf{ord}(C)$ is either -1, 0 or 1, where $\mathsf{ord}(X)$ is the maximum distance from X to any terminal in the derivation tree.

Document Databases. A *document database* over Σ is a finite collection $D = \{d_1, \dots, d_m\}$ of documents over Σ. Document databases are represented by a rootless SLP as follows. For an SLP $S = (N, \Sigma, R)$, let $\mathsf{docs}(S) = \{\text{doc}(A) \mid A \in N\}$ be the set of documents represented by S. The rootless SLP S is a representation for a document database D if $D \subseteq \mathsf{docs}(S)$.

For a document database D, it is assumed that a rootless SLP S that represents D is in normal form and strongly balanced. It is also assumed that for each nonterminal A for which its rule has the form $A \to BC$, the values $|\text{doc}(A)|$, $\mathsf{ord}(A)$ and nonterminals B and C are accessible in constant time. All these values can be precomputed with a linear-time pass over S. We call S along with constant-time access to these values *the basic data structure for S*.

Complex Document Editing. As in [26], given a document database $D = \{d_1, \dots, d_m\}$ our goal is to create new documents by a sequence of text-editing operations. Here we introduce the notion of a CDE-expression over D, which is defined by the following syntax:

$$\varphi := d_\ell, \ell \in [1, m] \mid \mathsf{concat}(\varphi, \varphi) \mid \mathsf{extract}(\varphi, i, j) \mid \mathsf{delete}(\varphi, i, j) \mid \mathsf{insert}(\varphi, \varphi, k) \mid \mathsf{copy}(\varphi, i, j, k)$$

where the values i, j are valid *positions*, and k is a valid *gap*. The semantics of these operations, called *basic operations*, works as follows:

$$\begin{aligned} \mathsf{concat}(d, d') &= d \cdot d' & \mathsf{insert}(d, d', k) &= d[1, k\rangle \cdot d' \cdot d[k, |d| + 1\rangle \\ \mathsf{extract}(d, i, j) &= d[i, j + 1\rangle & \mathsf{delete}(d, i, j) &= d[1, i\rangle \cdot d[j + 1, |d| + 1\rangle \\ \mathsf{copy}(d, i, j, k) &= \mathsf{insert}(d, d[i, j + 1\rangle, k) \end{aligned}$$

We write $\mathsf{eval}(\varphi)$ for the document obtained by evaluating φ on D according to these semantics. For an operation $\mathsf{extract}(\varphi, i, j)$, $\mathsf{delete}(\varphi, i, j)$, $\mathsf{insert}(\varphi, \psi, k)$, or $\mathsf{copy}(\varphi, i, j, k)$, i, j are valid positions if $i, j \in [1, |\mathsf{eval}(\varphi)|]$, and k is a valid gap if $k \in [1, |\mathsf{eval}(\varphi)| + 1]$. We define $|\varphi|$ as the number of basic operations in φ. To adding these new documents in the database we will use the notion of extending a rootless SLP. A rootless SLP $S' = (N', \Sigma, R')$ is called an extension of S if S' is in normal form, $N \subseteq N'$, and $R'(A) = R(A)$ for every $A \in N$. In this context, we call $N' \setminus N$ the *set of new non-terminals*. We define the *maximum intermediate document size* $|\max_\varphi(D)|$ induced by a CDE-expression φ on a document database D as the maximum size of $\mathsf{eval}(\psi)$ for any sub-expression ψ of φ (i.e., any substring ψ of φ that matches the CDE syntax).

Having defined most of the concepts mentioned in Theorem 13, we can introduce the following Theorem in [26], which will be instrumental in the final proof.

▶ **Theorem 14** ([26], Theorem 4.3). *Let D be a document database represented by a strongly balanced rootless SLP S in normal form. When given the basic data structure for S and a CDE-expression φ over D, we can construct a strongly balanced extension S' of S, along with its basic data structure, and a non-terminal $\tilde{A}$ of S' such that $\mathrm{doc}(\tilde{A}) = \mathsf{eval}(\varphi)$. This construction takes time $\mathcal{O}(|\varphi| \cdot \log|\max_\varphi(D)|)$. In particular, the number of new non-terminals $|N' \setminus N|$ is in $\mathcal{O}(|\varphi| \cdot \log|\max_\varphi(D)|)$.*

For the second bullet point in Theorem 13, we use the following fact, given without proof:

▶ **Observation 15.** *Let D be a document database that is represented by a rootless SLP S in normal form. Given the basic data structure for S and a CDE-expression φ over D which only mentions concat, we can construct an extension S' of S, along with its basic data structure, and a nonterminal $\tilde{A}$ of S' such that $\mathrm{doc}(\tilde{A}) = \mathsf{eval}(\varphi)$. This construction takes time $\mathcal{O}(|\varphi|)$. In particular, the number of new non-terminals $|N' \setminus N|$ is in $\mathcal{O}(|\varphi|)$.*

The query data structure. The structure we will use is the one produced in Theorem 11. This structure is built by an algorithm that receives an SLP S, an unambiguous sAnnA $\mathcal{T}$, and produces a (succinct) Shift-ECS $\mathcal{D}$ indexed by the matrices M_A, for each non-terminal A in S. These matrices store nodes $v = M_A[p, q]$ such that $[\![\mathcal{D}]\!](v)$ contains all partial annotations from a path of $\mathcal{T}$ which starts p, ends in q, and reads the string $\mathrm{doc}(A)$. Note that, although the algorithm receives a "rooted" SLP, it can be adapted quite easily to rootless SLPs by adding a node v_A for each non-terminal A in S, built as $v_A = \mathsf{union}_{q \in F}(M_A[q_0, q])$ (the same construction that was done for S_0 in the algorithm).

We define *the query data structure for S and $\mathcal{T}$* as the mentioned succinct Shift-ECS along with constant-time access to every index $M_A[p, q]$ for states p and q and non-terminal A. Note that for each A it holds that $[\![\mathcal{D}]\!](v_A) = [\![\mathcal{A}]\!](\mathrm{doc}(A))$. In particular, if S represents a document database D, then for each $d \in D$ there is a v in $\mathcal{D}$ for which $[\![\mathcal{D}]\!](v) = [\![\mathcal{A}]\!](d)$. Recall that for every node $v \in \mathcal{D}$, the set $[\![\mathcal{D}]\!](v)$ can be enumerated with output-linear delay.

▶ **Lemma 16.** *Let S be an SLP in normal form and an extension S' of S with new non-terminals $\tilde{N} = N' \setminus N$. Also, let $\mathcal{T}$ be an unambiguous sAnnA and assume we are given the query data structure for S and $\mathcal{A}$, and the basic data structure for S'. We can construct the query data structure for S' and $\mathcal{T}$ in $\mathcal{O}(|\mathcal{A}|^3 \cdot |\tilde{N}|)$ time.*

To prove Theorem 13, we first reduce the variable-set automata $\mathcal{A}_1, \dots \mathcal{A}_k$ to sAnnAs $\mathcal{T}_1, \dots, \mathcal{T}_k$ using the construction of Proposition 12. Note, however, that this reduction requires the input document to be modified as well. This can be solved by adding a non-terminal $A_{\#}$ for each $A \in N$, and a rule $A_{\#} \to AH$, where H is a new non-terminal with the rule $H \to \#$. Then, in the query data structure for S and $\mathcal{T}$, the nodes v_A are defined over the matrices $M_{A_{\#}}$ instead. That way, when the user chooses a document $d \in \mathsf{docs}(S)$ and a variable set automata $\mathcal{A}_i$, she can be given the set $[\![\mathcal{T}_i]\!](d\#)$ as output. Note that this has no influence in the time bounds given so far for the edit, except for a factor that is linear in $|\tilde{N}|$.

It can now be seen that the result follows from Theorem 14, Observation 15, and Lemma 16. The fact that for each $d \in \mathsf{docs}(S')$ the set $[\![\mathcal{A}]\!](d)$ can be enumerated with output-linear delay follows from the definition of the query data structure for S' and $\mathcal{A}$.

7 Future work

One natural direction for future work is to study which other compression schemes allow output-linear delay enumeration for evaluating annotated automata. To the best of our knowledge, the only model for compressed data in which spanner evaluation has been studied is SLPs. However, other models (such as some based on run-length encoding) allow better compression rates and might be more desirable results in practice.

Regarding the Shift-ECS data structure, it would be interesting to see how further one could extend the data structure while still allowing output-linear delay enumeration. Another aspect worth studying is whether there are enumeration results in other areas that one can improve using Shift-ECS. Lastly, it would be interesting to study whether one can apply fast matrix multiplication techniques to Algorithm 1 to improve the running time to sub-cubic time in the number of states.

References

1 Alfred V Aho and John E Hopcroft. *The design and analysis of computer algorithms*. Pearson Education India, 1974.

2 Antoine Amarilli, Pierre Bourhis, Louis Jachiet, and Stefan Mengel. A circuit-based approach to efficient enumeration. In *ICALP*, volume 80, pages 111:1–111:15, 2017.

3 Antoine Amarilli, Pierre Bourhis, Stefan Mengel, and Matthias Niewerth. Constant-delay enumeration for nondeterministic document spanners. *ACM Trans. Database Syst.*, 46(1):2:1–2:30, 2021.

4 Antoine Amarilli, Louis Jachiet, Martin Muñoz, and Cristian Riveros. Efficient enumeration for annotated grammars. In *PODS*, pages 291–300, 2022.

5 Guillaume Bagan. MSO queries on tree decomposable structures are computable with linear delay. In *CSL*, pages 167–181, 2006.

6 Guillaume Bagan, Arnaud Durand, and Etienne Grandjean. On acyclic conjunctive queries and constant delay enumeration. In *CSL*, pages 208–222, 2007.

7 Jean Berstel. *Transductions and context-free languages*. Springer-Verlag, 2013.

8 Pierre Bourhis, Alejandro Grez, Louis Jachiet, and Cristian Riveros. Ranked enumeration of MSO logic on words. In *ICDT*, volume 186, pages 20:1–20:19, 2021.

9 Marco Bucchi, Alejandro Grez, Andrés Quintana, Cristian Riveros, and Stijn Vansummeren. CORE: a complex event recognition engine. *VLDB*, 15(9):1951–1964, 2022.

10 Nofar Carmeli, Shai Zeevi, Christoph Berkholz, Benny Kimelfeld, and Nicole Schweikardt. Answering (unions of) conjunctive queries using random access and random-order enumeration. In *PODS*, pages 393–409, 2020.

11 Francisco Claude and Gonzalo Navarro. Self-indexed grammar-based compression. *Fundam. Informaticae*, 111(3):313–337, 2011.

12 Johannes Doleschal, Benny Kimelfeld, Wim Martens, and Liat Peterfreund. Weight annotation in information extraction. *Log. Methods Comput. Sci.*, 18(1), 2022.

13 James R. Driscoll, Neil Sarnak, Daniel Dominic Sleator, and Robert Endre Tarjan. Making data structures persistent. In *STOC*, pages 109–121, 1986.

14 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.

15 Ronald Fagin, Benny Kimelfeld, Frederick Reiss, and Stijn Vansummeren. Document spanners: A formal approach to information extraction. *J. ACM*, 62(2):12:1–12:51, 2015.

16 Fernando Florenzano, Cristian Riveros, Martín Ugarte, Stijn Vansummeren, and Domagoj Vrgoc. Efficient enumeration algorithms for regular document spanners. *ACM Trans. Database Syst.*, 45(1):3:1–3:42, 2020.

17 Alejandro Grez and Cristian Riveros. Towards streaming evaluation of queries with correlation in complex event processing. In *ICDT*, volume 155, pages 14:1–14:17, 2020.

18 Alejandro Grez, Cristian Riveros, Martín Ugarte, and Stijn Vansummeren. A formal framework for complex event recognition. *ACM Trans. Database Syst.*, 46(4):1–49, 2021.

19 Wojciech Kazana and Luc Segoufin. First-order query evaluation on structures of bounded degree. *Log. Methods Comput. Sci.*, 7(2), 2011.

20 John C. Kieffer and En-Hui Yang. Grammar-based codes: A new class of universal lossless source codes. *IEEE Trans. Inf. Theory*, 46(3):737–754, 2000.

21 Markus Lohrey. Algorithmics on slp-compressed strings: A survey. *Groups Complex. Cryptol.*, 4(2):241–299, 2012.

22 Martin Muñoz and Cristian Riveros. Streaming enumeration on nested documents. In *ICDT*, volume 220, pages 19:1–19:18, 2022.

23 Liat Peterfreund. Grammars for document spanners. In *ICDT*, volume 186, pages 7:1–7:18, 2021.

24 Wojciech Rytter. Application of lempel-ziv factorization to the approximation of grammar-based compression. In *CPM*, volume 2373, pages 20–31, 2002.

25 Markus L. Schmid and Nicole Schweikardt. Spanner evaluation over slp-compressed documents. In *PODS*, pages 153–165, 2021.

26 Markus L. Schmid and Nicole Schweikardt. Query evaluation over slp-represented document databases with complex document editing. In *PODS*, pages 79–89, 2022.

27 Nicole Schweikardt, Luc Segoufin, and Alexandre Vigny. Enumeration for FO queries over nowhere dense graphs. In *PODS*, pages 151–163, 2018.

28 James A. Storer and Thomas G. Szymanski. Data compression via textual substitution. *J. ACM*, 29(4):928–951, 1982.

Degree Sequence Bound for Join Cardinality Estimation

Kyle Deeds
University of Washington, Seattle, WA, USA

Dan Suciu
University of Washington, Seattle, WA, USA

Magda Balazinska
University of Washington, Seattle, WA, USA

Walter Cai
University of Washington, Seattle, WA, USA

Abstract

Recent work has demonstrated the catastrophic effects of poor cardinality estimates on query processing time. In particular, underestimating query cardinality can result in overly optimistic query plans which take orders of magnitude longer to complete than one generated with the true cardinality. Cardinality bounding avoids this pitfall by computing an upper bound on the query's output size using statistics about the database such as table sizes and degrees, i.e. value frequencies. In this paper, we extend this line of work by proving a novel bound called the Degree Sequence Bound which takes into account the full degree sequences and the max tuple multiplicity. This work focuses on the important class of Berge-Acyclic queries for which the Degree Sequence Bound is tight. Further, we describe how to practically compute this bound using a functional approximation of the true degree sequences and prove that even this functional form improves upon previous bounds.

2012 ACM Subject Classification Information systems → Query optimization; Information systems → Query planning; Theory of computation → Database query processing and optimization (theory); Theory of computation → Data modeling

Keywords and phrases Cardinality Estimation, Cardinality Bounding, Degree Bounds, Functional Approximation, Query Planning, Berge-Acyclic Queries

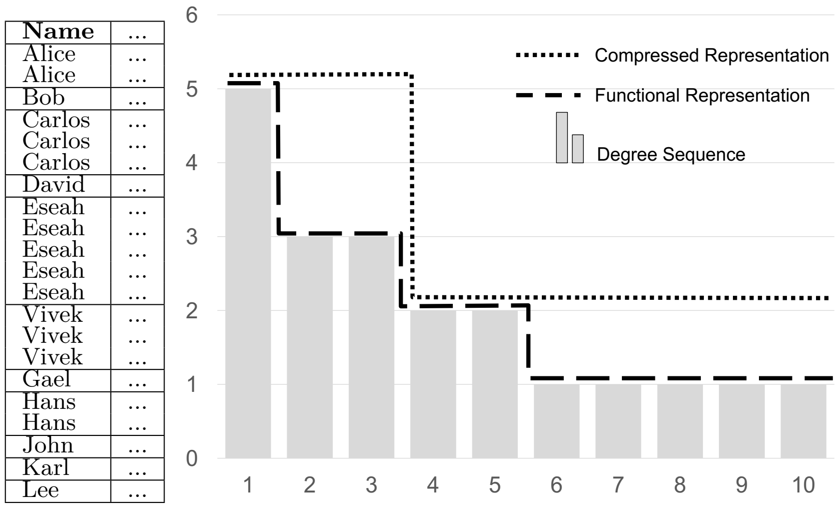
Digital Object Identifier 10.4230/LIPIcs.ICDT.2023.8

Related Version *Full Version*: `https://arxiv.org/pdf/2201.04166` [4]

Funding This work is supported by National Science Foundation grants NSF IIS 1907997 and NSF-BSF 2109922.

1 Introduction

The weakest link in a modern query processing engine is the *cardinality estimator*. There are several major decisions where the system needs to estimate the size of a query's output: the optimizer uses the estimate to compute an effective query plan; the scheduler needs the estimate to determine how much memory to allocate for a hash table and to decide whether to use a main-memory or an out-of-core algorithm; a distributed system needs the estimate to decide how many servers to reserve for subsequent operations. Today's systems estimate the cardinality of a query by making several strong and unrealistic assumptions, such as uniformity and independence. As a result, the estimates for multi-join queries commonly have relative errors up to several orders of magnitude. An aggravating phenomenon is that cardinality estimators consistently underestimate (this is a consequence of the independence assumption), and this leads to wrong decisions for the most expensive queries [15, 3, 10]. A significant amount of effort has been invested in the last few years into using machine



Name	...
Alice	...
Alice	...
Bob	...
Carlos	...
Carlos	...
Carlos	...
David	...
Eseah	...
Eseah	...
Eseah	...
Eseah	...
Eseah	...
Vivek	...
Vivek	...
Vivek	...
Gael	...
Hans	...
Hans	...
John	...
Karl	...
Lee	...

Figure 1 The degree sequence of `Name`. The first rank represents `Eseah` whose degree is 5, the next two ranks are for `Carlos` and `Vivek` whose degrees are 3. The degree sequence can be represented compactly using a staircase functions, and even more compactly using lossy compression.

learning for cardinality estimation [20, 22, 23, 24, 21, 16, 17], but this approach still faces several formidable challenges, such as the need for large training sets, the long training time of complex models, and the lack of guarantees about the resulting estimates.

An alternative approach to estimating the cardinality is to compute an *upper bound* for the size of the query answer. This approach originated in the database theory community, through the pioneering results by Grohe and Marx [8] and Atserias, Grohe, and Marx [1]. They described an elegant formula, now called the AGM bound, that gives a tight upper bound on the query result in terms of the cardinalities of the input tables. This upper bound was improved by the *polymatroid bound*, which takes into account both the cardinalities, and the degree constraints and includes functional dependencies as a special case [7, 13, 14, 18]. In principle, an upper bound could be used by a query optimizer in lieu of a cardinality estimator and, indeed, this idea was recently pursued by the systems community, where the upper bound appears under various names such as bound sketch or pessimistic cardinality estimator [3, 11]. In this paper, we will call it a *cardinality bound*. As expected, a cardinality bound prevents query optimizers from choosing disastrous plans for the most expensive queries [3], however, their relative error is often much larger than that of other methods [19, 6]. While the appeal of a guaranteed upper bound is undeniable, in practice overly pessimistic bounds are unacceptable.

In this paper, we propose a new upper bound on the query size based on *degree sequences*. By using a slightly larger memory footprint, this method has the potential to achieve much higher accuracy than previous bounds. Given a relation R, an attribute X, and a value $u \in \Pi_X(R)$, the *degree* of u is the number of tuples in R with u in the X attribute, formally $d^{(u)} = |\sigma_{X=u}(R)|$. The *degree sequence* of an attribute X in relation R is the sorted sequence of all degrees for the values of that attribute, $d^{(u_1)} \geq d^{(u_2)} \geq \cdots \geq d^{(u_n)}$. Going forward, we drop any reference to values and instead refer to degrees by their index in this sequence, also called their *rank*, i.e. $d_1 \geq \cdots \geq d_n$.[1] A degree sequence can easily be computed

[1] Note that the degree sequence is very similar to a rank-frequency distribution in the probability literature and has been extensively used in graph analysis [2, 9].

offline, and can be compressed effectively, with a good space/accuracy tradeoff due to its monotonicity; see Fig. 1 for an illustration. Degree sequences offer more information on the database instance than the statistics used by previous upper bounds. For example, the AGM bound uses only the cardinality of the relations, which is $\sum_i d_i$, while the extension to degree constraints [14] uses the cardinality, $\sum_i d_i$, and the maximum degree, d_1.

For this new bound we had to develop entirely new techniques over those used for the AGM and the polymatroid bounds. Previous techniques are based on information theory. If some relation $R(X, Y)$ has cardinality N, then any probability space over R has an entropy that satisfies $H(XY) \leq \log N$; if the degree sequence of the attribute X is $d_1 \geq d_2 \geq \ldots$, then $H(Y|X) \leq \log d_1$. Both the AGM and the polymatroid bound start from such constraints on the entropy. Unfortunately, these constraints do not extend to degree sequences, because H is ignorant of $d_2, d_3, \ldots$ Information theory gives us only three degrees of freedom, namely $H(XY), H(X), H(Y)$, while the degree sequence has an arbitrary number of degrees of freedom. Rather than using information theory, our new framework models relations as tensors, and formulates the upper bound as a linear optimization problem. This framework is restricted to *Berge-acyclic, fully conjunctive queries* [5] (reviewed in Sec. 2); throughout the paper we will assume that queries are in this class. As we explain in Appendix A.1 [4] these are the most common queries found in applications.

The Worst-Case Instance. Our main result (Theorems 3.2 and 4.1) is a tight cardinality bound given the degree sequences of all relations. This bound is obtained by evaluating the query on a *worst-case instance* that satisfies those degree constraints.[2] Intuitively, each relation of the worst-case instance is obtained by matching the highest degree values in the different columns, and the same principle is applied across relations. For example, consider the join $R(X, \ldots) \bowtie S(X, \ldots)$, where the degree sequences of $R.X$ and $S.X$ are $a_1 \geq a_2 \geq \cdots$ and $b_1 \geq b_2 \geq \cdots$ respectively. The true cardinality of the join is $\sum_i a_i b_{\tau(i)}$ for some unknown permutation τ , while the maximum cardinality is[3] $\sum_i a_i b_i$, and is obtained when the highest degree values match. Our degree sequence bound holds even when the input relations are allowed to be bags. Furthermore, we prove (Theorem 4.6) that this bound is always below the AGM and polymatroid bounds, although the latter restrict the relations to be sets. To prove this we had to develop a new, explicit formula for the polymatroid bound for Berge-acyclic queries, which is of independent interest (Theorem 4.3).

Compact Representation. A full degree sequence is about as large as the relation instance, while cardinality estimators need to run in sub-linear time. Fortunately, a degree sequence can be represented compactly using a piece-wise constant function, called a *staircase function*, as illustrated in Fig. 1. Our next result, Theorem 5.2, is an algorithm for the degree sequence bound that runs in quasi-linear time (i.e. linear plus a logarithmic factor) in the size of the representation, independent of the size of the instance. The algorithm makes some rounding errors (Lemma 5.1), hence its output may be slightly larger than the exact bound, however we prove that it is still lower than the AGM and polymatroid bounds (Theorem 5.5). The algorithm can be used in conjunction with a compressed representation of the degree sequence. By using few buckets and upper-bounding the degree sequence one can trade off the memory size and estimation time for accuracy. At one extreme, we could upper bound

[2] In graph theory, the problem of computing a graph satisfying a given degree sequence is called the *realization problem.*

[3] For example, if $a_1 \geq a_2$, $b_1 \geq b_2$, then $a_1b_1 + a_2b_2 \geq a_1b_2 + a_2b_1$.

the entire sequence using a single bucket with the constant d_1, at the other extreme we could keep the complete sequence. Neither the AGM bound nor the polymatroid bound have this tradeoff ability.

Max Tuple Multiplicity. Despite using more information than previous upper bounds, our bound can still be overly pessimistic, because it needs to match the most frequent elements in all attributes. For example, suppose a relation has two attributes whose highest degrees are a_1 and b_1 respectively. Its worst-case instance is a bag and must include some tuple that occurs $\min(a_1, b_1)$ times. Usually, a_1 and b_1 are large, since they represent the frequencies of the worst heavy hitters in the two columns, but in practice they rarely occur together $\min(a_1, b_1)$ times. To avoid such worst-case matchings, we use one additional piece of information on each base table: the max multiplicity over all tuples, denoted B. Usually, B is significantly smaller than the largest degrees, and, by imposing it as an additional constraint, we can significantly improve the query's upper bound; in particular, when $B = 1$ then the relation is restricted to be a set. Our main results in Theorems 3.2 and 4.1 extend to max tuple multiplicities, but in some unexpected ways. The worst-case relation, while still *tight*, is not a conventional relation: it may have tuples that occur more than B times, and, when the relation has 3 or more attributes it may even have tuples with negative multiplicities. Nevertheless, these rather unconventional worst-case relations provide an even better degree sequence bound than by ignoring B.

▶ **Example 1.1.** To give a taste of our degree-sequence bound, consider the full conjunctive query $Q(\cdots) = R(X, \cdots) \bowtie S(X, Y, \cdots) \bowtie T(Y, \cdots)$, where we omit showing attributes that appear in only one of the relations. Alternatively, we can write $Q(X, Y) = R(X) \bowtie S(X, Y) \bowtie T(Y)$ where R, S, T are bags rather than sets. Assume the following degree sequences:

$$\boldsymbol{d}^{(R.X)} = (3, 2, 2) \qquad \boldsymbol{d}^{(T.Y)} = (2, 1, 1, 1) \qquad \boldsymbol{d}^{(S.X)} = (5, 1) \qquad \boldsymbol{d}^{(S.Y)} = (3, 2, 1) \tag{1}$$

The AGM bound uses only the cardinalities, which are:

$$|R| = 7 \qquad\qquad |S| = 6 \qquad\qquad |T| = 5$$

The AGM bound[4] is $|R| \cdot |S| \cdot |T| = 210$. The extension to degree constraints in [14] uses in addition the maximum degrees:

$$\texttt{deg}(R.X) = 3 \qquad \texttt{deg}(S.X) = 5 \qquad \texttt{deg}(S.Y) = 3 \qquad \texttt{deg}(T.Y) = 2$$

and the bound is the minimum between the AGM bound and the following quantities:

$$|R| \cdot \texttt{deg}(S.X) \cdot \texttt{deg}(T.Y) = 7 \cdot 5 \cdot 2 = 70$$
$$\texttt{deg}(R.X) \cdot |S| \cdot \texttt{deg}(T.Y) = 3 \cdot 6 \cdot 2 = 36$$
$$\texttt{deg}(R.X) \cdot \texttt{deg}(S.Y) \cdot |T| = 3 \cdot 3 \cdot 5 = 45$$

Thus, the degree-constraint bound is improved to 36.

Our new bound is given by the answer to the query on the worst-case instance of the relations R, S, T, shown here together with their multiplicities (recall that they are bags):

$$R = \begin{array}{|c|}\hline a \\ b \\ c \\ \hline\end{array}\begin{array}{c} 3 \\ 2 \\ 2 \end{array}, \qquad S = \begin{array}{|c|c|}\hline a & u \\ a & v \\ b & w \\ \hline\end{array}\begin{array}{c} 3 \\ 2 \\ 1 \end{array}, \qquad T = \begin{array}{|c|}\hline u \\ v \\ w \\ z \\ \hline\end{array}\begin{array}{c} 2 \\ 1 \\ 1 \\ 1 \end{array},$$

[4] Recall that each of the three relations has private variables, e.g. $R(X, U), S(X, Y, V, W), T(Y, Z)$. The only fractional edge cover is $1, 1, 1$.

The three relations have the required degree sequences, for example $S.X$ consists of 5 a's and 1 b, thus has degree sequence $(5, 1)$. Notice the matching principle: we assumed that the most frequent element in $R.X$ and $S.X$ are the same value a, and that the most frequent values in $S.X$ and in $S.Y$ occur together. On this instance, we compute the query and obtain the answer Q.

$Q =$

a	u	$3 \cdot 3 \cdot 2 = 18$
a	v	$3 \cdot 2 \cdot 1 = 6$
b	w	$2 \cdot 1 \cdot 1 = 2$

$S' =$

a	u	2
a	v	2
a	w	1
b	u	1

The upper bound is the size of the answer on this instance, which is $18 + 6 + 2 = 26$, and it improves over 36. Here, the improvement is relatively minor, but this is a consequence of the short example. In practice, degree sequences often have a long tail, i.e. with a few large leading degrees $d_1, d_2, \ldots$ followed by very many small degrees $d_m, d_{m+1}, \ldots, d_n$ (with a large n). In that case the improvements of the new bound can be very significant.

Suppose now that we have one additional information about S: every tuple occurs at most $B = 2$ times. Then we need to reduce the multiplicity of (a, u), and the new worst-case instance, denoted S', is the following relation which decreases the cardinality bound to 25.

2 Problem Statement

Tensors. In this paper, it is convenient to define tensors using a named perspective, where each dimension is associated with a variable. We write variables with capital letters $X, Y, \ldots$ and sets of variables with boldface, $\boldsymbol{X}, \boldsymbol{Y}, \ldots$ We assume that each variable X has an associated finite domain $D_X \stackrel{\text{def}}{=} [n_X]$ for some number $n_X \geq 1$. For any set of variables $\boldsymbol{X}$ we denote by $D_{\boldsymbol{X}} \stackrel{\text{def}}{=} \prod_{Z \in \boldsymbol{X}} D_Z$. We use lower case for values, e.g. $z \in D_Z$ and boldface for tuples, e.g. $\boldsymbol{x} \in D_{\boldsymbol{X}}$. An $\boldsymbol{X}$-*tensor*, or simply a tensor when $\boldsymbol{X}$ is clear from the context, is $\boldsymbol{M} \in \mathbb{R}^{D_{\boldsymbol{X}}}$. We say that $\boldsymbol{M}$ has $|\boldsymbol{X}|$ dimensions. Given two $\boldsymbol{X}$-tensors $\boldsymbol{M}, \boldsymbol{N}$, we write $\boldsymbol{M} \leq \boldsymbol{N}$ for the component-wise order ($M_{\boldsymbol{x}} \leq N_{\boldsymbol{x}}$, for all $\boldsymbol{x}$). If $\boldsymbol{X}, \boldsymbol{Y}$ are two sets of variables, then we denote their union by $\boldsymbol{XY}$. If, furthermore, $\boldsymbol{X}, \boldsymbol{Y}$ are disjoint, and $\boldsymbol{x} \in D_{\boldsymbol{X}}, \boldsymbol{y} \in D_{\boldsymbol{Y}}$, then we denote by $\boldsymbol{xy} \in D_{\boldsymbol{XY}}$ the concatenation of the two tuples.

▶ **Definition 2.1.** Let $\boldsymbol{M}, \boldsymbol{N}$ be an $\boldsymbol{X}$-tensor, and a $\boldsymbol{Y}$-tensor respectively. Their tensor product is the following $\boldsymbol{XY}$-tensor:

$$\forall \boldsymbol{z} \in D_{\boldsymbol{XY}} : \qquad (\boldsymbol{M} \otimes \boldsymbol{N})_{\boldsymbol{z}} \stackrel{\text{def}}{=} M_{\pi_{\boldsymbol{X}}(\boldsymbol{z})} \cdot N_{\pi_{\boldsymbol{Y}}(\boldsymbol{z})} \tag{2}$$

If $\boldsymbol{X}, \boldsymbol{Y}$ are disjoint and $\boldsymbol{M}$ is an $\boldsymbol{XY}$-tensor then we define its $\boldsymbol{X}$-summation to be the following $\boldsymbol{Y}$-tensor:

$$\forall \boldsymbol{y} \in D_{\boldsymbol{Y}} : \qquad (\texttt{SUM}_{\boldsymbol{X}}(\boldsymbol{M}))_{\boldsymbol{y}} \stackrel{\text{def}}{=} \sum_{\boldsymbol{x} \in D_{\boldsymbol{X}}} M_{\boldsymbol{xy}} \tag{3}$$

If $\boldsymbol{M}, \boldsymbol{N}$ are $\boldsymbol{XY}$ and $\boldsymbol{YZ}$ tensors, where $\boldsymbol{X}, \boldsymbol{Y}, \boldsymbol{Z}$ are disjoint sets of variables, then their dot product is the $\boldsymbol{XZ}$-tensor:

$$\forall \boldsymbol{x} \in D_{\boldsymbol{X}}, \boldsymbol{z} \in D_{\boldsymbol{Z}} : \qquad (\boldsymbol{M} \cdot \boldsymbol{N})_{\boldsymbol{xz}} \stackrel{\text{def}}{=} \texttt{SUM}_{\boldsymbol{Y}}(\boldsymbol{M} \otimes \boldsymbol{N})_{\boldsymbol{xz}} = \sum_{\boldsymbol{y} \in D_{\boldsymbol{Y}}} M_{\boldsymbol{xy}} N_{\boldsymbol{yz}} \tag{4}$$

In other words, in this paper we use $\otimes$ like a natural join. For example, if $\boldsymbol{M}$ is an IJ-tensor (i.e. a matrix) and $\boldsymbol{N}$ is an KL-tensor, then $\boldsymbol{M} \otimes \boldsymbol{N}$ is the Kronecker product; if $\boldsymbol{P}$ is an IJ-tensor (like $\boldsymbol{M}$) then $\boldsymbol{M} \otimes \boldsymbol{P}$ is the element-wise product. The dot product sums out the common variables, for example if $\boldsymbol{a}$ is a J-tensor, then $\boldsymbol{M} \cdot \boldsymbol{a}$ is the standard matrix-vector multiplication, and its result is an I-tensor. The following is easily verified. If $\boldsymbol{M}$ is an $\boldsymbol{X}$-tensor, $\boldsymbol{N}$ is a $\boldsymbol{Y}$-tensor and $\boldsymbol{X}, \boldsymbol{Y}$ are disjoint sets of variables, then:

$$\forall \boldsymbol{X}_0 \subseteq \boldsymbol{X}, \forall \boldsymbol{Y}_0 \subseteq \boldsymbol{Y} : \qquad \texttt{SUM}_{\boldsymbol{X}_0 \boldsymbol{Y}_0}(\boldsymbol{M} \otimes \boldsymbol{N}) = \texttt{SUM}_{\boldsymbol{X}_0}(\boldsymbol{M}) \otimes \texttt{SUM}_{\boldsymbol{Y}_0}(\boldsymbol{N}) \tag{5}$$

Permutations. A permutation on $D = [n]$ is a bijective function $\sigma : D \to D$; the set of permutations on D is denoted S_D, or simply S_n. If $\boldsymbol{D} = D_1 \times \cdots \times D_k$ then we denote by $S_{\boldsymbol{D}} \stackrel{\text{def}}{=} S_{D_1} \times \cdots \times S_{D_k}$. Given an $\boldsymbol{X}$-tensor $\boldsymbol{M} \in \mathbb{R}^{D_{\boldsymbol{X}}}$ and permutations $\boldsymbol{\sigma} \in S_{D_{\boldsymbol{X}}}$, the *$\boldsymbol{\sigma}$-permuted $\boldsymbol{X}$-tensor* is $\boldsymbol{M} \circ \boldsymbol{\sigma} \in \mathbb{R}^{D_{\boldsymbol{X}}}$:

$$\forall \boldsymbol{x} \in D_{\boldsymbol{X}} : \qquad (\boldsymbol{M} \circ \boldsymbol{\sigma})_{\boldsymbol{x}} \stackrel{\text{def}}{=} \boldsymbol{M}_{\boldsymbol{\sigma}(\boldsymbol{x})}$$

Sums are invariant under permutations, for example if $\boldsymbol{a}, \boldsymbol{b} \in \mathbb{R}^{D_Z}$ are Z-vectors and $\sigma \in S_{D_Z}$, then $(\boldsymbol{a} \circ \sigma) \cdot (\boldsymbol{b} \circ \sigma) = \boldsymbol{a} \cdot \boldsymbol{b}$, because $\sum_{i \in D_Z} a_{\sigma(i)} b_{\sigma(i)} = \sum_{i \in D_Z} a_i b_i$.

Queries. A full conjunctive query Q is:

$$Q(\boldsymbol{X}) = \bowtie_{R \in \boldsymbol{R}} R(\boldsymbol{X}_R) \tag{6}$$

where $\boldsymbol{R} \stackrel{\text{def}}{=} \boldsymbol{R}(Q)$ denotes the set of its relations, $\boldsymbol{X}$ is a set of variables, and $\boldsymbol{X}_R \subseteq \boldsymbol{X}$ for each relation $R \in \boldsymbol{R}$. The *incidence graph* of Q is the following bipartite graph: $T \stackrel{\text{def}}{=} (\boldsymbol{R} \cup \boldsymbol{X}, E \stackrel{\text{def}}{=} \{(R, Z) \mid Z \in \boldsymbol{X}_R\})$. It can be shown that Q is *Berge-acyclic* [5] iff its incidence graph is an undirected tree (see Appendix A.1 [4]). Unless otherwise stated, all queries in this paper are assumed to be full, Berge-acyclic conjunctive queries. We use bag semantics for query evaluation, and represent an *instance* of a relation $R \in \boldsymbol{R}$ by an $\boldsymbol{X}_R$-tensor, $\boldsymbol{M}^{(R)}$, where $M^{(R)}_t$ is defined to be the multiplicity of the tuple $t \in D_{\boldsymbol{X}_R}$ in the bag R. The number of tuples in the answer to Q is:

$$|Q| = \texttt{SUM}_{\boldsymbol{X}} \left(\bigotimes_{R \in \boldsymbol{R}} \boldsymbol{M}^{(R)} \right) \tag{7}$$

▶ **Example 2.2.** Consider the following query:

$$Q(X, Y, Z, U, V, W) = R(X, Y) \bowtie S(Y, Z, U) \bowtie T(U, V) \bowtie K(Y, W)$$

Its incidence graph is $T = (\{R, \ldots, K\} \cup \{X, \ldots, W\}, \{(R, X), (R, Y), (S, Y), \ldots, (K, W)\})$ and is an undirected tree. An instance of $R(X, Y)$ is represented by a matrix $\boldsymbol{M}^{(R)} \in \mathbb{R}^{D_X \times D_Y}$, where $M^{(R)}_{xy}$ = the number of times the tuple (x, y) occurs in R. Similarly, S is represented by a tensor $\boldsymbol{M}^{(S)} \in \mathbb{R}^{D_Y \times D_Z \times D_U}$. The size of the query's output is:

$$|Q| = \sum_{x,y,z,u,v,w} M^{(R)}_{xy} M^{(S)}_{yzu} M^{(T)}_{uv} M^{(K)}_{yw}$$

Degree Sequences. We denote by $\mathbb{R}_+ \stackrel{\text{def}}{=} \{x \mid x \in \mathbb{R}, x \geq 0\}$ and we say that a vector $f \in \mathbb{R}^{[n]}_+$ is non-increasing if $f_{r-1} \geq f_r$ for $r \in [2, \ldots, n]$.

▶ **Definition 2.3.** Fix a set of variables $\boldsymbol{X}$, with domains D_Z, $Z \in \boldsymbol{X}$. A *degree sequence* associated with the dimension $Z \in \boldsymbol{X}$ is a non-increasing vector $\boldsymbol{f}^{(Z)} \in \mathbb{R}^{D_Z}_+$. We call the index r the *rank*, and $f^{(Z)}_r$ the *degree at rank* r. An $\boldsymbol{X}$-tensor $\boldsymbol{M}$ is *consistent* w.r.t. $\boldsymbol{f}^{(Z)}$ if:

$$\texttt{SUM}_{\boldsymbol{X} - \{Z\}}(\boldsymbol{M}) \leq \boldsymbol{f}^{(Z)} \tag{8}$$

$\boldsymbol{M}$ is consistent with a tuple of degree sequences $\boldsymbol{f}^{(\boldsymbol{X})} \stackrel{\text{def}}{=} (\boldsymbol{f}^{(Z)})_{Z \in \boldsymbol{X}}$, if it is consistent with every $\boldsymbol{f}^{(Z)}$. Furthermore, given $B \in \mathbb{R}_+ \cup \{\infty\}$, called the *max tuple multiplicity*, we say that $\boldsymbol{M}$ is *consistent* w.r.t. B if $\boldsymbol{M}_t \leq B$ for all $t \in D_{\boldsymbol{X}}$. We denote:

$$\mathcal{M}_{\boldsymbol{f}^{(\boldsymbol{X})},B} \stackrel{\text{def}}{=} \{\boldsymbol{M} \in \mathbb{R}^{D_{\boldsymbol{X}}} \mid \boldsymbol{M} \text{ is consistent with } \boldsymbol{f}^{(\boldsymbol{X})}, B\}$$
$$\mathcal{M}^+_{\boldsymbol{f}^{(\boldsymbol{X})},B} \stackrel{\text{def}}{=} \{\boldsymbol{M} \in \mathbb{R}_+^{D_{\boldsymbol{X}}} \mid \boldsymbol{M} \text{ is non-negative and consistent with } \boldsymbol{f}^{(\boldsymbol{X})}, B\} \tag{9}$$

For a simple illustration consider two degree sequences $\boldsymbol{f} \in \mathbb{R}^{[m]}, \boldsymbol{g} \in \mathbb{R}^{[n]}$. $\mathcal{M}_{\mathbf{f},\mathbf{g},\infty}$ is the set of matrices $\boldsymbol{M}$ whose row-sums and column-sums are $\leq \boldsymbol{f}$ and $\leq \boldsymbol{g}$ respectively; $\mathcal{M}^+_{\mathbf{f},\mathbf{g},\infty}$ is the subset of non-negative matrices; $\mathcal{M}^+_{\mathbf{f},\mathbf{g},B}$ is the subset of matrices that also satisfy $M_{ij} \leq B$, $\forall i,j$.

Problem Statement. Fix a query Q. For each relation R, we are given a set of degree sequences $\boldsymbol{f}^{(R,\boldsymbol{X}_R)} \stackrel{\text{def}}{=} \left(\boldsymbol{f}^{(R,Z)}\right)_{Z \in \boldsymbol{X}_R}$, and a tuple multiplicity $B^{(R)} \in \mathbb{R}_+ \cup \{\infty\}$. We are asked to find the maximum size of Q over all database instances consistent with all degree sequences and tuple multiplicities. To do this, we represent a relation instance R by an unknown tensor $\boldsymbol{M}^{(R)} \in \mathcal{M}^+_{\mathbf{f}^{(R,\boldsymbol{X}_R)},B^{(R)}}$ and an unknown set of permutations $\boldsymbol{\sigma}^{(R)} \in S_{D_{\boldsymbol{X}_R}}$, and solve the following problem:

▶ **Problem 1** (Degree Sequence Bound)**.** Solve the following optimization problem:

$$\text{Maximize: } |Q| = \texttt{SUM}_{\boldsymbol{X}} \left(\bigotimes_{R \in \boldsymbol{R}} (\boldsymbol{M}^{(R)} \circ \boldsymbol{\sigma}^{(R)}) \right) \tag{10}$$
$$\text{Where: } \forall R \in \boldsymbol{R},\ \boldsymbol{\sigma}^{(R)} \in S_{D_{\boldsymbol{X}_R}},\ \boldsymbol{M}^{(R)} \in \mathcal{M}^+_{\boldsymbol{f}^{(R,\boldsymbol{X}_R)},B^{(R)}}$$

This is a non-linear optimization problem: while the set $\mathcal{M}^+$ defined in Eq. (9) is a set of linear constraints, the objective (10) is non-linear. In the rest of the paper we describe an explicit formula for the degree sequence bound, which is optimal (i.e. tight) when $B^{(R)} = \infty$, for all R, and is optimal in a weaker sense in general.

▶ **Example 2.4.** Continuing Example 1.1, the four degree sequences in (1) correspond to the variables in each relation $R.X$, $S.X$, $S.Y$, and $T.Y$. Since $S.X$ has a shorter degree sequence than $R.X$, we pad it with a 0, so it becomes $\boldsymbol{d}^{(S.X)} = (5,1,0)$; similarly for $\boldsymbol{d}^{(S.Y)}$. Instead of values c,b,a, we use indices $1,2,3$, similarly u,v,w,z becomes $1,2,3,4$. For example, $S = \begin{array}{|c|c|c} 3 & 1 & 3 \\ 3 & 2 & 2 \\ 2 & 3 & 1 \end{array}$ is isomorphic to the instance in Example 1.1. It is represented by $\boldsymbol{M} \circ (\sigma, \tau)$ where the matrix $\boldsymbol{M} = \begin{pmatrix} 3 & 2 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$, (its row-sums are $5,1,0$ and column-sums are $3,2,1,0$, as required) and the permutations are, in two-line notation, $\sigma \stackrel{\text{def}}{=} \begin{pmatrix} 1 & 2 & 3 \\ 3 & 2 & 1 \end{pmatrix}$ and $\tau \stackrel{\text{def}}{=}$ the identity. Similarly, the relations R, T, are represented by vectors $\boldsymbol{a}, \boldsymbol{b}$ and permutations θ, ρ. The bound of Q is the maximum value of $\sum_{i=1,3} \sum_{j=1,4} M_{\sigma(i)\tau(j)} a_{\theta(i)} b_{\rho(j)}$, where $\boldsymbol{M}, \boldsymbol{a}, \boldsymbol{b}$ are consistent with the given degree sequences, and $\sigma, \tau, \theta, \rho$ are permutations. This is a special case of Eq. (10).

3 The Star Query

We start by computing the degree sequence bound for a *star query*, which is defined as:

$$Q_{\texttt{star}} = S(X_1, \ldots, X_d) \bowtie R^{(1)}(X_1) \bowtie \cdots \bowtie R^{(d)}(X_d) \tag{11}$$

Assume that the domain of each variable X_p is $[n_p]$ for some $n_p > 0$, and denote by $[\boldsymbol{n}] \stackrel{\text{def}}{=} [n_1] \times \cdots \times [n_d]$. Later, in Sec. 4, we will use the bound for $Q_{\texttt{star}}$ as a building block to compute the degree sequence bound of a general query Q. There, S will be one of the relations of the query, for which we know the degree sequences $\boldsymbol{f}^{(X_p)} \in \mathbb{R}_+^{[n_p]}$, $p = 1, \ldots, d$ and tuple bound B, while the unary relations $R^{(1)}, \ldots, R^{(d)}$ will be results of subqueries, which are unknown. The instance of each $R^{(p)}$ is given by an unknown vector $\boldsymbol{a}^{(p)} \in \mathbb{R}_+^{[n_p]}$, which we can assume w.l.o.g. to be non-increasing, by permuting the domain of X_p in both S and in $R^{(p)}$. Therefore, S will be represented by $\boldsymbol{M} \circ \boldsymbol{\sigma}$, where $\boldsymbol{M} \in \mathcal{M}^+_{\boldsymbol{f}^{(\boldsymbol{X})},B}$ is some tensor and $\boldsymbol{\sigma}$ some permutation, and the size of $Q_{\texttt{star}}$ is:

$$|Q_{\texttt{star}}| = \sum_{(i_1,\ldots,i_d)\in[\boldsymbol{n}]} (\boldsymbol{M} \circ \boldsymbol{\sigma})_{i_1\cdots i_d} \cdot a^{(X_1)}_{i_1} \cdots a^{(X_d)}_{i_d} \tag{12}$$

Equivalently: $|Q_{\texttt{star}}| = \texttt{SUM}_{\boldsymbol{X}} \left((\boldsymbol{M} \circ \boldsymbol{\sigma}) \otimes \bigotimes_p \boldsymbol{a}^{(X_p)} \right) = (\boldsymbol{M} \circ \boldsymbol{\sigma}) \cdot \boldsymbol{a}^{(X_1)} \cdots \boldsymbol{a}^{(X_d)}$.

Our goal is to find the unknown $\boldsymbol{M} \circ \boldsymbol{\sigma}$ for which $|Q_{\texttt{star}}|$ is maximized, no matter what the unary relations are. It turns out that $\boldsymbol{\sigma}$ can always be chosen the identity permutation, thus it remains to find the optimal $\boldsymbol{M}$, which we denote by $\boldsymbol{C}$. This justifies:

▶ **Problem 2** (Worst-Case Tensor). Fix $\boldsymbol{f}^{(\boldsymbol{X})}$, B. Find a tensor $\boldsymbol{C} \in \mathcal{M}_{\boldsymbol{f}^{(\boldsymbol{X})},\infty}$ such that, for all $\boldsymbol{\sigma} \in S_{[\boldsymbol{n}]}$, $\boldsymbol{M} \in \mathcal{M}^+_{\boldsymbol{f}^{(\boldsymbol{X})},B}$, and all non-increasing vectors $\boldsymbol{a}^{(X_1)} \in \mathbb{R}_+^{[n_1]}, \ldots, \boldsymbol{a}^{(X_d)} \in \mathbb{R}_+^{[n_d]}$:

$$(\boldsymbol{M} \circ \boldsymbol{\sigma}) \cdot \boldsymbol{a}^{(X_1)} \cdots \boldsymbol{a}^{(X_d)} \leq \boldsymbol{C} \cdot \boldsymbol{a}^{(X_1)} \cdots \boldsymbol{a}^{(X_d)} \tag{13}$$

In the rest of this section we describe the solution $\boldsymbol{C}$. If all entries in $\boldsymbol{C}$ are ≥ 0 and $\leq B$, then $\boldsymbol{C} \in \mathcal{M}^+_{\boldsymbol{f}^{(\boldsymbol{X})},B}$ and, by setting $\boldsymbol{M} \stackrel{\text{def}}{=} \boldsymbol{C}$ and $\boldsymbol{\sigma} \stackrel{\text{def}}{=}$ the identity permutations, the relation S represented by $\boldsymbol{M} \circ \boldsymbol{\sigma}$ maximizes $|Q_{\texttt{star}}|$, achieving our goal. But, somewhat surprisingly, we found that sometimes this worst-case $\boldsymbol{C}$ has entries $> B$ or < 0, yet it still achieves our goal of a tight upper bound for $|Q_{\texttt{star}}|$. This is why we allow $\boldsymbol{C} \in \mathcal{M}_{\boldsymbol{f}^{(\boldsymbol{X})},\infty}$.

Let Δ_Z denote the *discrete derivative* of an $\boldsymbol{X}$-tensor w.r.t. a variable $Z \in \boldsymbol{X}$, and Σ_Z denote the *discrete integral*. Formally, if $\boldsymbol{a} \in \mathbb{R}^{[n]}$ is a Z-vector, then, setting $a_0 \stackrel{\text{def}}{=} 0$:

$$\forall i \in [n]: \qquad (\Delta_Z \boldsymbol{a})_i \stackrel{\text{def}}{=} a_i - a_{i-1} \qquad (\Sigma_Z \boldsymbol{a})_i = \sum_{j=1,i} a_j \tag{14}$$

Notice that:

$$\Sigma_Z(\Delta_Z \boldsymbol{a}) = \Delta_Z(\Sigma_Z \boldsymbol{a}) = \boldsymbol{a} \qquad \texttt{SUM}_Z(\Delta_Z \boldsymbol{a}) = a_n \tag{15}$$

The subscript in Δ, Σ indicates on which variable they act. For example, if $\boldsymbol{M}$ is an XYZ-tensor, then $(\Delta_Y \boldsymbol{M})_{xyz} \stackrel{\text{def}}{=} M_{xyz} - M_{x(y-1)z}$. One should think of the three operators $\Delta_X, \Sigma_X, \texttt{SUM}_X$ as analogous to the continuous operators $\frac{d\cdots}{dx}$, $\int \cdots dx$, $\int_0^n \cdots dx$.

▶ **Definition 3.1.** The *value* tensor, $\boldsymbol{V}^{\boldsymbol{f}^{(\boldsymbol{X})},B} \in \mathbb{R}_+^{[\boldsymbol{n}]}$, is defined by the following linear optimization problem:

$$\forall \boldsymbol{m} \in [\boldsymbol{n}] : \qquad V_{\boldsymbol{m}}^{\boldsymbol{f}^{(\boldsymbol{X})},B} \stackrel{\text{def}}{=} \text{Maximize: } \sum_{\boldsymbol{s} \le \boldsymbol{m}} M_{\boldsymbol{s}} \qquad \text{Where: } \boldsymbol{M} \in \mathcal{M}^+_{\boldsymbol{f}^{(\boldsymbol{X})},B} \tag{16}$$

The *worst-case* tensor, $\boldsymbol{C}^{\boldsymbol{f}^{(\boldsymbol{X})},B} \in \mathbb{R}^{[\boldsymbol{n}]}$, is defined as:

$$\boldsymbol{C}^{\boldsymbol{f}^{(\boldsymbol{X})},B} \stackrel{\text{def}}{=} \Delta_{X_1} \cdots \Delta_{X_d} \boldsymbol{V}^{\boldsymbol{f}^{(\boldsymbol{X})},B} \tag{17}$$

We will drop the superscripts when clear from the context, and write simply $\boldsymbol{V}, \boldsymbol{C}$. Our main result in this section is:

▶ **Theorem 3.2.** *Let $\boldsymbol{f}^{(\boldsymbol{X})}, B$ be given as above, and let $\boldsymbol{V}, \boldsymbol{C}$ defined by* (16)-(17)*. Then:*

1. *$\boldsymbol{C}$ is a solution to Problem 2, i.e. $\boldsymbol{C} \in \mathcal{M}_{\boldsymbol{f}^{(\boldsymbol{X})},\infty}$ and it satisfies Eq.* (13)*. Furthermore, it is tight in the following sense: there exists a tensor $\boldsymbol{M} \in \mathcal{M}^+_{\boldsymbol{f}^{(\boldsymbol{X})},B}$ and non-increasing vectors $\boldsymbol{a}^{(p)} \in \mathbb{R}_+^{[n_p]}$, $p = 1, d$, such that inequality* (13) *(with $\boldsymbol{\sigma}$ the identity) is an equality.*
2. *If there exists any solution $\boldsymbol{C}' \in \mathcal{M}^+_{\boldsymbol{f}^{(\boldsymbol{X})},B}$ to Problem 2, then $\boldsymbol{C}' = \boldsymbol{C}$.*
3. *When the number of dimensions is $d = 2$ then $\boldsymbol{C}$ is integral and non-negative. If $d \ge 3$, $\boldsymbol{C}$ may have negative entries.*
4. *If $B < \infty$, then $\boldsymbol{C}$ may not be consistent with B, even if $d = 2$.*
5. *For any non-increasing vectors $\boldsymbol{a}^{(X_p)} \in \mathbb{R}_+^{[n_p]}$, $p = 2, d$, the vector $\boldsymbol{C} \cdot \boldsymbol{a}^{(X_2)} \cdots \boldsymbol{a}^{(X_d)}$ is in $\mathbb{R}_+^{[n_1]}$ and non-increasing.*
6. *Assume $B = \infty$. Then the following holds:*

$$\forall \boldsymbol{m} \in [\boldsymbol{n}] : \qquad V_{\boldsymbol{m}} = \min\left(F_{m_1}^{(X_1)}, \ldots, F_{m_d}^{(X_d)}\right) \tag{18}$$

where $F_r^{(X_p)} \stackrel{def}{=} \sum_{j \le r} f_j^{(X_p)}$ is the CDF associated to the PDF $\boldsymbol{f}^{(X_p)}$, for $p = 1, d$. Moreover, $\boldsymbol{C}$ can be computed by Algorithm 1, which runs in time $\mathbf{O}(\sum_p n_p)$. This further implies that $\boldsymbol{C} \ge 0$, in other words $\boldsymbol{C} \in \mathcal{M}^+_{\boldsymbol{f}^{(\boldsymbol{X})},\infty}$.

Algorithm 1 Efficient construction of $\boldsymbol{C}$ when $B = \infty$.

```
$\forall p = 1, d : s_p \leftarrow 1$;     $\boldsymbol{C} = 0$;
while $\forall p : s_p < n_p$ do
  $p_{min} \leftarrow \arg\min_p(f_{s_p}^{(X_p)})$     $d_{min} \leftarrow \min_p(f_{s_p}^{(X_p)})$
  $C_{s_1,\ldots,s_d} \leftarrow d_{min}$
  $\forall p = 1, d : f_{s_p}^{(X_p)} \leftarrow f_{s_p}^{(X_p)} - d_{min}$
  $s_{p_{min}} \leftarrow s_{p_{min}} + 1$
end while
return $C$
```

In a nutshell, the theorem asserts that the tensor $\boldsymbol{C}$ defined in (17) is the optimal solution to Problem 2; this is stated in item 1. Somewhat surprisingly, $\boldsymbol{C}$ may be inconsistent w.r.t. B, and may even be negative. When that happens, then, by item 2, no consistent solution exists to Problem 2, hence we have to make do with $\boldsymbol{C}$. In that case $\boldsymbol{C}$ may not represent a traditional bag S, for example if it has entries < 0. However, this will not be a problem for computing the degree sequence bound in Sec. 4, because all we need is to compute the product $\boldsymbol{C} \cdot \boldsymbol{a}^{(X_2)} \cdots \boldsymbol{a}^{(X_d)}$, which we need to be non-negative, and non-increasing: this is guaranteed by item 5. The last item gives more insight into $\boldsymbol{V}$ and, by extension, into $\boldsymbol{C}$. Recall that $V_{\boldsymbol{m}}$, defined by (16), is the largest possible sum of values of a consistent $m_1 \times m_2 \times \cdots \times m_d$ tensor $\boldsymbol{M}$. Since the sum in each hyperplane $X_1 = r$ of $\boldsymbol{M}$ is $\leq f_r^{(X_1)}$, it follows that $\sum_{\boldsymbol{s} \leq \boldsymbol{m}} M_{\boldsymbol{s}} \leq \sum_{r=1,m_1} f_r^{(X_1)} \stackrel{\text{def}}{=} F_{m_1}^{(X_1)}$. Repeating this argument for each dimension X_p implies that $V_{\boldsymbol{m}} \leq \min_{p=1,d}(F_{m_p}^{(X_p)})$. Item 6 states that this becomes an equality, when $B = \infty$.

▶ **Example 3.3.** Suppose that we want to maximize $\boldsymbol{a}^T \cdot \boldsymbol{M} \cdot \boldsymbol{b}$, where $\boldsymbol{M}$ is a 3×4 matrix with degree sequences $\boldsymbol{f} = (6, 3, 1)$ and $\boldsymbol{g} = (4, 3, 2, 1)$; assume $B = \infty$. The vectors $\boldsymbol{a}, \boldsymbol{b}$ are non-negative and non-increasing, but otherwise unknown. The theorem asserts that this product is maximized by the worst-case matrix $\boldsymbol{C}$. We show here the matrices $\boldsymbol{C}$ and $\boldsymbol{V}$ defined by (16) and (17), together with degree sequences $\boldsymbol{f}, \boldsymbol{g}$ next to $\boldsymbol{C}$, and the cumulative sequences $\boldsymbol{F} = \Sigma \boldsymbol{f}, \boldsymbol{G} = \Sigma \boldsymbol{g}$ next to $\boldsymbol{V}$:

$$\boldsymbol{C} = \begin{array}{c|cccc} & 4 & 3 & 2 & 1 \\ \hline 6 & 4 & 2 & 0 & 0 \\ 3 & 0 & 1 & 2 & 0 \\ 1 & 0 & 0 & 0 & 1 \end{array} \qquad \boldsymbol{V} = \begin{array}{c|cccc} & 4 & 7 & 9 & 10 \\ \hline 6 & 4 & 6 & 6 & 6 \\ 9 & 4 & 7 & 9 & 9 \\ 10 & 4 & 7 & 9 & 10 \end{array}$$

We can check that $V_{m_1 m_2} = \min(F_{m_1}, G_{m_2})$; for example $V_{31} = \min(10, 4) = 4$. The worst-case matrix $\boldsymbol{C}$ is defined as the second discrete derivative of $\boldsymbol{V}$, more precisely $C_{m_1 m_2} = V_{m_1 m_2} - V_{m_1-1, m_2} - V_{m_1, m_2-1} + V_{m_1-1, m_2-1}$. Alternatively, $\boldsymbol{C}$ can be computed greedily, using Algorithm 1: start with $C_{11} \leftarrow \min(f_1, g_1) = 4$, decrease both f_1, g_1 by 4, set the rest of column 1 to 0 (because now $g_1 = 0$) and continue with C_{12}, etc. Another important property, which we will prove below in the Appendix (Eq. 35 [4]), is that, for all m_1, m_2, $\sum_{i \leq m_1, j \leq m_2} C_{ij} = V_{m_1 m_2}$; for example $\sum_{i \leq 2, j \leq 3} C_{ij} = 4 + 2 + 1 + 2 = 9 = V_{23}$.

While the proof of Theorem 3.2 provides interesting insight into the structure of the degree sequence bound, it is not necessary for understanding the remainder of the paper and requires the introduction of additional notation and machinery. Therefore, for the sake of space and clarity, we omit it from the main text and instead include a proof of each item in the appendix Section A.2 [4].

4 The Berge-Acyclic Query

We now turn to the general problem 1. Fix a Berge-acyclic query Q with relations $\boldsymbol{R} \stackrel{\text{def}}{=} \boldsymbol{R}(Q)$, degree sequences $\boldsymbol{f}^{R,Z}$, and max tuple multiplicities $B^{(R)}$ as in problem 1.

4.1 The Degree Sequence Bound

▶ **Theorem 4.1.** *For any tensors $\boldsymbol{M}^{(R)} \in \mathcal{M}^+_{\boldsymbol{f}^{(R,\boldsymbol{X}_R)}, B^{(R)}}$ and permutations $\boldsymbol{\sigma}^{(R)}$, for $R \in \boldsymbol{R}$, the following holds:*

$$\mathit{SUM}_{\boldsymbol{X}}\left(\bigotimes_{R \in \boldsymbol{R}} (\boldsymbol{M}^{(R)} \circ \boldsymbol{\sigma}^{(R)})\right) \leq \mathit{SUM}_{\boldsymbol{X}}\left(\bigotimes_{R \in \boldsymbol{R}} \boldsymbol{C}^{\boldsymbol{f}^{(R,\boldsymbol{X}_R)}, B^{(R)}}\right) \stackrel{\text{def}}{=} \mathit{DSB}(Q) \tag{19}$$

where $\boldsymbol{C}^{\boldsymbol{f}^{(R,\boldsymbol{X}_R)}, B^{(R)}}$ is the worst-case tensor from Def. 3.1.

The theorem simply says that the upper bound to the query Q can be computed by evaluating Q on the worst case instances, represented by the worst case tensors $\boldsymbol{C}^{\boldsymbol{f}^{(R,\boldsymbol{X}_R)}, B^{(R)}}$. We call this quantity the *degree sequence bound* and denote it by $\mathit{DSB}(Q)$. When all max

Algorithm 2 Computing $DSB(Q) = \texttt{SUM}_{\boldsymbol{X}} \left(\bigotimes_{R \in \boldsymbol{R}} \boldsymbol{C}^{\boldsymbol{f}^{(R,\boldsymbol{X}_R)},B^{(R)}} \right)$.

```
for each variable $X \in \boldsymbol{X}$ and non-root relation $R \in \boldsymbol{R}$, $R \neq \texttt{root}$, in bottom-up order do
    $\boldsymbol{a}^{(X)} \stackrel{\text{def}}{=} \bigotimes_{R \in \text{children}(X)} \boldsymbol{w}^{(R)}$    // element-wise product
    $\boldsymbol{w}^{(R)} \stackrel{\text{def}}{=} \boldsymbol{C}^{\boldsymbol{f}^{(R,\boldsymbol{X}_R)},B^{(R)}} \cdot \boldsymbol{a}^{(X_2)} \cdots \boldsymbol{a}^{(X_k)}$  // where $\boldsymbol{X}_R = (X_1, \ldots, X_k)$, $X_1 = \texttt{parent}(R)$
end for
return $C^{\boldsymbol{f}^{(\texttt{root},\boldsymbol{X}_{\texttt{ROOT}})},B^{(\texttt{ROOT})}} \cdot \boldsymbol{a}^{(X_1)} \cdot \boldsymbol{a}^{(X_2)} \cdots \boldsymbol{a}^{(X_k)}$
```

tuple multiplicities $B^{(R)}$ are ∞, then the bound is tight, because in that case every worst-case tensor $\boldsymbol{C}^{\boldsymbol{f}^{(R,\boldsymbol{X}_R)},\infty}$ is in $\mathcal{M}^+_{\boldsymbol{f}^{(R,\boldsymbol{X}_R)},\infty}$ (by Th. 3.2 item 6); otherwise the bound may not be tight, but it is locally tight, in the sense of Th. 3.2 item 1.

Before we sketch the main idea of the proof, we note that an immediate consequence is that the degree sequence bound can be computed using a special case of the FAQ algorithm [12]. We describe this briefly in Algorithm 2. Recall that the incidence graph of Q is a tree T. Choose an arbitrary relation $\texttt{ROOT} \in \boldsymbol{R}(Q)$ and designate it as root, then make T a directed tree by orienting all its edges away from the root. Denote by $\texttt{parent}(R) \in \boldsymbol{X}_R$ the parent node of a relation $R \neq \texttt{ROOT}$, associate an X-vector $\boldsymbol{a}^{(X)}$ to each variable X, and a $\texttt{parent}(R)$-vector $\boldsymbol{w}^{(R)}$ to each relation name R, then compute these vectors by traversing the tree bottom-up, as shown in Algorithm 2. Notice that, when X is a leaf variable, then $\text{children}(X) = \emptyset$ and $\boldsymbol{a}^{(X)} = (1, 1, \ldots, 1)^T$; similarly, if $R(X)$ is leaf relation of arity 1 with variable X, then $\boldsymbol{w}^{(R)}$ is the degree sequence of its variable, because $\boldsymbol{w}^{(R)} = \boldsymbol{C}^{(\boldsymbol{f}^{(R,X)},B^{(R)})} = \boldsymbol{f}^{(R,X)}$. We provide an example in [4], Appendix A.3. It follows:

▶ **Corollary 4.2.** *The degree sequence bound $DSB(Q)$ can be computed in time polynomial in the size of the largest domain (data complexity).*

In the rest of this section we sketch the proof of Theorem 4.1, mostly to highlight the role of item 5 of Theorem 3.2, and defer the formal details to Appendix A.3 [4]. Fix tensors $\boldsymbol{M}^{(R)}$ and permutations $\boldsymbol{\sigma}^{(R)}$, for each $R \in \boldsymbol{R}$. Choose one relation, say $S \in \boldsymbol{R}$, assume it has k variables $X_1, \ldots, X_k$, then write the LHS of (19) as:

$$\texttt{SUM}_{\boldsymbol{X}_S} \left(\left(\boldsymbol{M}^{(S)} \circ \boldsymbol{\sigma}^{(S)} \right) \otimes \boldsymbol{b}_1 \otimes \cdots \otimes \boldsymbol{b}_k \right) \tag{20}$$

where each $\boldsymbol{b}_p$ is a tensor expression sharing only variable X_p with S, where we sum out all variables except X_p (using Eq. (5)). Compute the vectors $\boldsymbol{b}_p$ first, sort them in non-decreasing order, let τ_p be the permutation that sorts $\boldsymbol{b}_p$, and $\boldsymbol{\tau} \stackrel{\text{def}}{=} (\tau_1, \ldots, \tau_k)$. Then (20) equals:

$$\texttt{SUM}_{\boldsymbol{X}_S} \left(\left(\boldsymbol{M}^{(S)} \circ \boldsymbol{\sigma}^{(R)} \circ \boldsymbol{\tau} \right) \otimes (\boldsymbol{b}_1 \circ \tau_1) \otimes \cdots \otimes (\boldsymbol{b}_k \circ \tau_k) \right) \tag{21}$$

because sums are invariant under permutations. Since each $\boldsymbol{b}_p \circ \tau_p$ is sorted, by item 1 of Theorem 3.2, the expression above is $\leq$ to the expression obtained by replacing $\boldsymbol{M}^{(S)} \circ \boldsymbol{\sigma}^{(S)} \circ \boldsymbol{\tau}$ with the worst-case tensor $\boldsymbol{C}^{f^{(S,\boldsymbol{X}_S)},B^{(S)}}$. Thus, every tensor could be replaced by the worst-case tensor, albeit at the cost of applying some new permutations τ_p to other expressions. To avoid introducing these permutations, we proceed as follows. We choose an orientation of the tree T, as in Algorithm 2, then prove inductively, bottom-up the tree, that each tensor $\boldsymbol{M} \circ \boldsymbol{\sigma}$ can be replaced by the worst-case tensor $\boldsymbol{C}$ without decreasing the LHS of (19), *and* that the resulting vector (in the bottom-up computation) is sorted. To prove this, we re-examine Eq. (20), assuming X_1 is the parent variable of S. By induction, all the tensors occurring in $\boldsymbol{b}_2, \ldots, \boldsymbol{b}_k$ have already been replaced with worst-case tensors, *and* their results are non-increasing vectors. Then, in Eq. (21) it suffices to apply the permutation τ to the parent expression $\boldsymbol{b}_1$ (which still has the old tensors $\boldsymbol{M} \circ \boldsymbol{\sigma}$), use item 1 of Theorem 3.2 to replace $\boldsymbol{M}^{(S)} \circ \boldsymbol{\sigma}^{(S)} \circ \boldsymbol{\tau}$ by $\boldsymbol{C}^{f^{(S,\boldsymbol{X}_S)},B^{(S)}}$, and, finally, use item 5 of Theorem 3.2 to prove that the result returned by the node S is a non-decreasing vector, as required.

4.2 Connection to the AGM and Polymatroid Bounds

We prove now that $DSB(Q)$ is always below the AGM [1] and the polymatroid bounds [14, 18].

The AGM bound is expressed in terms of the cardinalities of the relations. For each relation R, let N_R be an upper bound on its cardinality. Then the AGM bound is $AGM(Q) \overset{\text{def}}{=} \min_{\boldsymbol{w}} \prod_R N_R^{w_R}$, where the vector $\boldsymbol{w} = (w_R)_{R \in \boldsymbol{R}}$ ranges over the fractional edge covers of the hypergraph associated to Q. If a database instance satisfies $|R| \leq N_R$ for all R, then the size of the query is $|Q| \leq AGM(Q)$, and this bound is tight, i.e. there exists an instance for which we have equality.

The polymatroid bound uses both the cardinality constraints N_R and the maximum degrees. The general bound in [14] considers maximum degrees for any subset of variables, but throughout this paper we restrict to degrees of single variables, in which case the polymatroid bound is expressed in terms of the quantities N_R and $f_1^{(R,X)}$, one for each relation R and each of its variables X. The AGM bound is the special case when $f_1^{(R,X)} = N_R$ for all R. We review the general definition of the polymatroid bound in [4], Appendix A.4, but will mention that no closed formula is known for polymatroid bound, similar to the AGM bound. We give here the first such closed formula, for the case of Berge-acyclic queries. Let Q be a Berge-acyclic query with incidence graph T (which is a tree). Choose an arbitrary relation $\texttt{ROOT} \in \boldsymbol{R}(Q)$ to designate as the root of T, and for each other relation R, denote by $Z_R \overset{\text{def}}{=} \texttt{parent}(R)$, i.e. its unique variable pointing up the tree. Denote by:

$$PB(Q, \texttt{ROOT}) \overset{\text{def}}{=} N_{\texttt{ROOT}} \prod_{R \neq \texttt{ROOT}} f_1^{(R,Z_R)} \tag{22}$$

One can immediately check that the query answer on any database instance consistent with the statistics satisfies $|Q| \leq PB(Q, \texttt{ROOT})$. A *cover* of Q is set $\boldsymbol{W} = \{Q_1, Q_2, \ldots, Q_m\}$, for some $m \geq 1$, where each Q_i is a connected subquery of Q, and each variable of Q occurs in at least one Q_i, and we denote by:

$$PB(\boldsymbol{W}) \overset{\text{def}}{=} \prod_{i=1,m} \min_{\texttt{ROOT} \in \boldsymbol{R}(Q_i)} PB(Q_i, \texttt{ROOT}_i) \tag{23}$$

Since $|Q| \leq |Q_1| \cdot |Q_2| \cdots |Q_m|$, we also have $|Q| \leq PB(\boldsymbol{W})$. We prove in [4], Appendix A.4:

▶ **Theorem 4.3.** *The polymatroid bound of a Berge-acyclic query Q is $PB(Q) \overset{def}{=} \min_{\boldsymbol{W}} PB(\boldsymbol{W})$, where $\boldsymbol{W}$ ranges over all covers.*

▶ **Example 4.4.** Let $Q = R(X,Y), S(Y,Z), T(Z,U), K(U,V)$. Then $PB(Q,S) = f_1^{(R,Y)} N_S f_1^{(T,Z)} f_1^{(K,U)}$, $PB(\{R, TK\}) = N_R \cdot \min\left(N_T f^{(K,U)}, f^{(T,U)} N_k\right)$, and $PB(\{R,T,K\}) = N_R N_T N_K$.

If we restrict the formula to the AGM bound, i.e. all max degrees are equal to the cardinalities, $f_1^{(R,X)} = N_R$, then Eq. (22) becomes $\prod_{R \in \boldsymbol{R}(Q)} N_R$, while the polymatroid bound (23) becomes $\min_{\boldsymbol{W}} \prod_{R \in \boldsymbol{W}} N_R$, where $\boldsymbol{W}$ ranges over integral covers of Q. In particular, the AGM bound of a Berge-acyclic query can be obtained by restricting to integral edge covers, although this property fails for α-acylic queries. For example, consider the query

$R(X,Y), S(Y,Z), T(Z,X), K(X,Y,Z)$; when $|R| = |S| = |T| = |K|$ then the AGM bound is obtained by the edge cover $0,0,0,1$, but when $|R| = |S| = |T| \ll |K|$ one needs the fractional cover $1/2, 1/2, 1/2, 0$. Next, we prove next that the degree sequence bound is always better.

▶ **Lemma 4.5.**

(1) *For any choice of root relation, $\texttt{ROOT} \in \boldsymbol{R}(Q)$: $DSB(Q) \leq PB(Q, \texttt{ROOT})$.*

(2) *For any cover $Q_1, \ldots, Q_m$ of Q, $DSB(Q) \leq DSB(Q_1) \cdots DSB(Q_m)$*

Proof. (1) Referring to Algorithm 2, we prove by induction on the tree that, for all $R \neq \texttt{ROOT}$, and every index i, $w_i^{(R)} \leq \prod_{S \in \texttt{tree}(R)} f_1^{(S,Z_S)}$. In other words, each element of the vector $\boldsymbol{w}^{(R)}$ is $\leq$ the product of all max degrees in the subtree rooted at R. Assuming this holds for all children of R, consider the definition of $\boldsymbol{w}^{(R)}$ in Algorithm 2. By induction hypothesis, for each vector $\boldsymbol{a}^{(X_p)}$ we have $a_{i_p}^{(X_p)} \leq \prod_{S \in \texttt{tree}(X_p)} f_1^{(S,Z_S)}$, a quantity that is independent of the index i_p, and therefore we obtain the following:

$$w_{i_1}^{(R)} = \left(\boldsymbol{C}^{\boldsymbol{f}^{(R,\boldsymbol{X}_R)},B^{(R)}} \cdot \boldsymbol{a}^{(X_2)} \cdots \boldsymbol{a}^{(X_k)}\right)_{i_1} \leq \left(\sum_{i_2 i_3 \cdots i_k} C_{i_1 i_2 i_3 \cdots i_k}^{\boldsymbol{f}^{(R,\boldsymbol{X}_R)},B^{(R)}}\right) \cdot \prod_{S \in \texttt{tree}(R), S \neq R} f_1^{(S,Z_S)}$$

and we use the fact that $\sum_{i_2 i_3 \cdots i_k} C_{i_1 i_2 \cdots i_k}^{\boldsymbol{f}^{(R,\boldsymbol{X}_R)},B^{(R)}} \leq f_{i_1}^{(R,X_1)}$ because, by Theorem 3.2 item 1, $\boldsymbol{C}^{\boldsymbol{f}^{(R,\boldsymbol{X}_R)},B^{(R)}}$ is consistent with the degree sequence $f_1^{(R,X_1)}$, and, finally, $f_{i_1}^{(R,X_1)} \leq f_1^{(R,X_1)}$. This completes the inductive proof. The algorithm returns $C^{\boldsymbol{f}^{(\texttt{root},\boldsymbol{X}_{\texttt{ROOT}})},B^{(\texttt{ROOT})}} \cdot \boldsymbol{a}^{(X_1)} \cdot \boldsymbol{a}^{(X_2)} \cdots \boldsymbol{a}^{(X_k)} \leq \texttt{SUM}(C^{\boldsymbol{f}^{(\texttt{root},\boldsymbol{X}_{\texttt{ROOT}})},B^{(\texttt{ROOT})}}) \cdot \prod_{R \neq \texttt{ROOT}} f_1^{(R,Z_R)} \leq |\texttt{ROOT}| \cdot \prod_{R \neq \texttt{ROOT}} f_1^{(R,Z_R)}$, which is $= PB(Q, \texttt{ROOT})$, as required.

(2) We prove the statement only for $m = 2$ (the general case is similar) and show that $DSB(Q) \leq DSB(Q_1) \cdot DSB(Q_2)$. Since DSB is the query answer on the worst case instance, we need to show that $|Q_1 \bowtie Q_2| \leq |Q_1| \cdot |Q_2|$. This is not immediately obvious because the worst case instance may have negative multiplicities. Let X be the unique common variable of Q_1, Q_2, and let $\boldsymbol{a}$, $\boldsymbol{b}$ be the X-vectors representing the results of Q_1 and Q_2 respectively. It follows from Theorem 3.2 item 5 that $\boldsymbol{a}, \boldsymbol{b}$ are non-negative, therefore, $|Q| = \sum_i a_i b_i \leq (\sum_i a_i)(\sum_i b_i) = |Q_1| \cdot |Q_2|$. ◀

Our discussion implies:

▶ **Theorem 4.6.** *Let Q be a Berge-acyclic query. We denote by $DSB(Q, \boldsymbol{f}, \boldsymbol{B})$ the DSB computed on the statistics $\boldsymbol{f} \overset{def}{=} (\boldsymbol{f}^{R,Z})_{R \in \boldsymbol{R}(Q), Z \in \boldsymbol{X}_R}$ and $\boldsymbol{B} \overset{def}{=} (B^{(R)})_{R \in \boldsymbol{R}(Q)}$. Then:*

$$|Q| \leq DSB(Q, \boldsymbol{f}, \boldsymbol{1}) \leq DSB(Q, \boldsymbol{f}, \boldsymbol{B}) \leq DSB(Q, \boldsymbol{f}, \infty) \leq PB(Q) \leq AGM(Q) \tag{24}$$

where $|Q|$ is the answer to the query on an database instance consistent with the given statistics.

Recall that both AGM and PB bounds are defined over set semantics only. While the AGM bound is tight, the PB bound is known to not be tight in general, and it is open whether it is tight for Berge-acyclic queries. Our degree sequence bound under either set or bag semantics improves over PB and, in the case of bag semantics ($B = \infty$) DSB is tight.

5 Functional Representation

A degree sequence requires, in general, $\Omega(n)$ space, where $n = \max_{X \in \boldsymbol{X}} n_X$ is the size of the largest domain, while cardinality estimators require sublinear space and time. However, a degree sequence can be *represented* compactly, using a staircase function as illustrated in

Fig. 1. In this section we show how the degree sequence bound, DSB, be approximated in quasi-linear time in the size of the functional representation. We call this approximate bound FDSB, show that DSB $\leq$ FDSB $\leq PB$, and show that the staircase functions can be further compressed, allowing a tradeoff between the memory size and computation time on one hand, and accuracy of the FDSB on the other hand. We restrict our discussion to $B^{(R)} = \infty$.

In this section we denote a vector element by $F(i)$ rather than F_i. For a non-decreasing vector $\boldsymbol{F} \in \mathbb{R}_+^{[n]}$, we denote by $\boldsymbol{F}^{-1} : \mathbb{R}_+ \to \mathbb{R}_+$ any function satisfying the following, for all v, $0 \leq v \leq F(n)$: if $F(i) < v$ then $i < F^{-1}(v)$, and if $F(i) > v$ then $i > F^{-1}(v)$. Such a function always exists[5], but is not unique. Then:

▶ **Lemma 5.1.** *Let $\boldsymbol{F}_1 \in \mathbb{R}_+^{[n_1]}, \ldots, \boldsymbol{F}_d \in \mathbb{R}_+^{[n_d]}$ be non-decreasing vectors satisfying $F_1(0) = 0$ and, for all $p = 1, d$, $F_1(n_1) \leq F_p(n_p)$. Let $a_1 \in \mathbb{R}_+^{[n_1]}, \ldots, a_d \in \mathbb{R}_+^{[n_d]}$ be non-increasing vectors. Denote by $\boldsymbol{C}, \boldsymbol{w}$ the following tensor and vector:*

$$C_{i_1 \cdots i_d} \overset{def}{=} \Delta_{i_1} \cdots \Delta_{i_d} \max(F_1(i_1), \ldots, F_d(i_d)) \tag{25}$$

$$w(i_1) \overset{def}{=} \sum_{i_2=1}^{n_2} \cdots \sum_{i_d=1}^{n_d} C_{i_1 \cdots i_d} \prod_{p \in [2,d]} a_p(i_p) \tag{26}$$

Then the following inequalities hold:

$$w(i_1) \geq (\Delta_{i_1} F_1(i_1)) \prod_{p \in [2,d]} a_p \left(\left\lfloor F_p^{-1}(F_1(i_1)) \right\rfloor + 1 \right) \tag{27}$$

$$w(i_1) \leq (\Delta_{i_1} F_1(i_1)) \prod_{p \in [2,d]} a_p \left(\left\lceil F_p^{-1}(F_1(i_1 - 1)) \right\rceil \right) \tag{28}$$

We give the proof in Appendix [4]. The lemma implies that, in Algorithm 2, we can use inequality (28) to upper bound the computation $w^{(R)} = \boldsymbol{C} \cdot \boldsymbol{a}^{(X_2)} \cdots \boldsymbol{a}^{(X_k)}$. Indeed, in that case each $F_p(r) \overset{\text{def}}{=} \sum_{i=1,r} f_p(r)$ is the cdf of a degree sequence f_p, hence $F_p(0) = 0$ and $F_p(n_p) =$ the cardinality of R, while the tensor $\boldsymbol{C}$ is described in item 6 of Theorem 3.2, hence the assumptions of the lemma hold.

We say that a vector $\boldsymbol{f} \in \mathbb{R}_+^n$ is *represented* by a function $\hat{f} : \mathbb{R}_+ \to \mathbb{R}_+$ if $f(i) = \hat{f}(i)$ for all $i = 1, n$. A function $\hat{f}$ is a *staircase function with s steps*, in short an *s-staircase*, if there exists dividers $m_0 \overset{\text{def}}{=} 0 < m_1 < \cdots < m_s \overset{\text{def}}{=} n$ such that $\hat{f}(x)$ is a nonnegative constant on each interval $\{x \mid m_{q-1} < x \leq m_q\}$, $q = 1, s$. The sum or product of an s_1-staircase with an s_2-staircase is an $(s_1 + s_2)$-staircase. We denote the summation of a staircase $\hat{f}(x)$ as $\hat{F}(x) = \int_0^x \hat{f}(t)dt$ which is then an increasing piecewise-linear function. Its standard inverse $\hat{F}^{-1} : \mathbb{R}_+ \to \mathbb{R}_+$ is also increasing and piecewise-linear. If $\hat{F}$ represents the vector F, then $\hat{F}^{-1}$ is an inverse F^{-1} of that vector (as discussed above).

Fix a Berge-acyclic query Q, and let each degree sequence $\boldsymbol{f}^{(R,Z)}$ be represented by some $s_{R,Z}$-staircase $\hat{f}^{R,Z}$, and we denote by $\hat{F}^{(R,Z)}$ its summation. Fix any relation $\texttt{ROOT} \in \boldsymbol{R}(Q)$ to designated as root. The *Functional Degree Sequence Bound at* ***ROOT***, $FDSB(Q, \texttt{ROOT})$, is the value returned by Algorithm 3. This algorithm is identical to Algorithm 2, except that it replaces both $\boldsymbol{w}^{(R)}$ with a functional upper bound justified by the inequality 28 of Lemma 5.1, and similarly for the returned result. All functions $\hat{\boldsymbol{a}}^{(X)}$ and $\hat{\boldsymbol{w}}^{(R)}$ are staircase functions, and can be computed in linear time, plus a logarithmic time need for a binary search to lookup a segment in a staircase. Using this, we prove the following in Appendix A.6 [4]:

[5] E.g. define it as follows: if $\exists i$ s.t. $F(i-1) < v < F(i)$ then set $F^{-1}(v) \overset{\text{def}}{=} i - 1/2$, otherwise set $F^{-1}(v) = i$ for some arbitrary i s.t. $F(i) = v$.

Algorithm 3 $FDSB(Q, \texttt{ROOT})$.

```
for each variable $X \in \boldsymbol{X}$ and non-root relation $R \in \boldsymbol{R}$, $R \neq \texttt{root}$, in bottom-up order do
    $\hat{\boldsymbol{a}}^{(X)} \stackrel{\text{def}}{=} \bigotimes_{R \in \text{children}(X)} \hat{\boldsymbol{w}}^{(R)}$
    $\forall i_1 : \quad \hat{w}^{(R)}(i_1) \stackrel{\text{def}}{=} \left(\hat{f}^{(R,X_1)}(i_1)\right) \prod_{p \in [2,d]} a^{(X_p)} \left(\max(1, (\hat{F}^{(R,X_p)})^{-1}(\hat{F}^{(R,X_1)}(i_1 - 1)))\right)$
end for
return $\sum_{i=1,|\texttt{ROOT}|} \prod_{p=1,k} a^{(X_p)}(\max(1, (F^{\texttt{ROOT},X_p})^{-1}(i-1)))$
```

▶ **Theorem 5.2.**

(1) *$FDSB(Q, ROOT) \geq DSB(Q)$.*

(2) *$FDSB(Q, ROOT)$ can be computed in time $T_{FDSB} \stackrel{def}{=} \tilde{O}(m \cdot \sum_{R,Z}(arity(R) \cdot s_{R,Z}))$, where $\tilde{O}$ hides a logarithmic term, and $m = |\boldsymbol{R}(Q)|$ is the number of relations in Q.*

The theorem says that $FDSB(Q, \texttt{ROOT})$ is still an upper bound on $|Q|$, and can be computed in quasi-linear time in the size of the functional representations of the degree sequences. Next, we check if $FDSB$ is below the polymatroid bound. Consider the computation of $\hat{w}^{(R)}(i_1)$ by the algorithm. On one hand $\hat{f}^{(R,X_1)}(i_1) \leq \hat{f}^{(R,X_1)}(1)$; on the other hand $a^{(X_p)}(\max(1, \ldots)) \leq a^{(X_p)}(1)$. This allows us to prove (inductively on the tree, in [4], Appendix A.7):

▶ **Lemma 5.3.** *$FDSB(Q, ROOT) \leq PB(Q, ROOT)$, where PB is defined in (22).*

When we proved $DSB \leq PB$ in Lemma 4.5, we used two properties of DSB: $DSB(Q, \texttt{ROOT})$ is independent of the choice of $\texttt{ROOT}$, and $DSB(Q_1 \bowtie \cdots \bowtie Q_m) \leq DSB(Q_1) \cdots DSB(Q_m)$, for any cover $\boldsymbol{W} = \{Q_1, \ldots, Q_m\}$. Both hold because $DSB(Q)$ is standard query evaluation: it is independent of the query plan (i.e. choice of $\texttt{ROOT}$) and it can only increase if we remove join conditions. But $FDSB$ is no longer standard query evaluation and these properties may fail. For that reason we introduce a stronger functional degree sequence bound:

$$FDSB(Q) = \min_{\boldsymbol{W}} \prod_{i=1,m} \min_{ROOT \in \boldsymbol{R}(Q)} FDSB(Q_i, ROOT) \tag{29}$$

where $\boldsymbol{W}$ range over the covers of Q. We prove in Appendix A.6.1 [4]:

▶ **Theorem 5.4.** *$FDSB(Q)$ can be computed in time $O(2^m \cdot (2^m + m \cdot T_{FDSB}))$ (where T_{FDSB} is defined in Theorem 5.2).*

Mirroring our results from Theorem 4.6, we prove the following in Appendix A.8 [4]:

▶ **Theorem 5.5.** *Suppose Q is a Berge-acyclic query. Then the following hold:*

$$|Q| \leq FDSB(Q) \leq PB(Q) \leq AGM(Q) \tag{30}$$

Together, Theorems 5.4 and 5.5 imply that we can compute in quasi-linear time in the size of the representation an upper bound to the query Q that is guaranteed to improve over the polymatroid bound. In practice, we expect this bound to be significantly lower than the polymatroid bound, because it accounts for the entire degree sequence $\boldsymbol{f}$, not just f_1.

Finally, we show that one can tradeoff the size of the representation for accuracy, by simply choosing more coarse staircase approximations of the degree sequences. They only need to be non-increasing, and lie above the true degree sequences.

▶ **Theorem 5.6.** *Fix a query Q, let $\boldsymbol{f}^{(R,Z)}, B^{(R)}$ be statistics as in Problem 1, and let U be the cardinality bound defined by* (10)*. Let $\hat{\boldsymbol{f}}^{(R,Z)}, \hat{B}^{(R)}$ be a new set of statistics, and $\hat{U}$ the resulting cardinality bound. If $\boldsymbol{f}^{(R,\boldsymbol{X}_R)} \leq \hat{\boldsymbol{f}}^{(R,\boldsymbol{X}_R)}$ and $B^{(R)} \leq \hat{B}^{(R)}$ for all R, $Z \in X_R$, then $U \leq \hat{U}$.*

Proof. The proof follows immediately from the observation that the set of feasible solutions can only increase (see Def. 2.3): $\mathcal{M}^+_{\boldsymbol{f}^{(R,\boldsymbol{X}_R)},B^{(R)}} \subseteq \mathcal{M}^+_{\hat{\boldsymbol{f}}^{(R,\boldsymbol{X}_R)},\hat{B}^{(R)}}$. ◀

6 Conclusions

We have described the *degree sequence bound* of a conjunctive query, which is an upper bound on the size of its answer, given in terms of the degree sequences of all its attributes. Our results apply to Berge-acyclic queries, and strictly improve over previously known AGM and polymatroid bounds [1, 14]. On one hand, our results represent a significant extension, because they account for the full degree sequences rather than just cardinalities or just the maximum degrees. On the other hand, they apply only to a restricted class of acyclic queries, although, we argue, this class is the most important for practial applications. While the full degree sequence can be as large as the entire data, we also described how to approximate the cardinality bound very efficiently, using compressed degree sequences. Finally, we have argued for using the max tuple multiplicity for each relation, which can significantly improve the accuracy of the cardinality bound.

References

1 Albert Atserias, Martin Grohe, and Dániel Marx. Size bounds and query plans for relational joins. In *49th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2008, October 25-28, 2008, Philadelphia, PA, USA*, pages 739–748. IEEE Computer Society, 2008. doi:10.1109/FOCS.2008.43.

2 Douglas Bauer, Haitze J Broersma, Jan van den Heuvel, Nathan Kahl, A Nevo, E Schmeichel, Douglas R Woodall, and Michael Yatauro. Best monotone degree conditions for graph properties: a survey. *Graphs and combinatorics*, 31(1):1–22, 2015.

3 Walter Cai, Magdalena Balazinska, and Dan Suciu. Pessimistic cardinality estimation: Tighter upper bounds for intermediate join cardinalities. In Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska, editors, *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, pages 18–35. ACM, 2019. doi:10.1145/3299869.3319894.

4 Kyle Deeds, Dan Suciu, Magda Balazinska, and Walter Cai. Degree sequence bound for join cardinality estimation. *arXiv preprint*, 2022. arXiv:2201.04166.

5 Ronald Fagin. Degrees of acyclicity for hypergraphs and relational database schemes. *J. ACM*, 30(3):514–550, 1983. doi:10.1145/2402.322390.

6 Amir Gilad, Shweta Patwa, and Ashwin Machanavajjhala. Synthesizing linked data under cardinality and integrity constraints. In Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava, editors, *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, pages 619–631. ACM, 2021. doi:10.1145/3448016.3457242.

7 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.

8 Martin Grohe and Dániel Marx. Constraint solving via fractional edge covers. In *Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2006, Miami, Florida, USA, January 22-26, 2006*, pages 289–298. ACM Press, 2006. URL: http://dl.acm.org/citation.cfm?id=1109557.1109590.

9 S Louis Hakimi and Edward F Schmeichel. Graphs and their degree sequences: A survey. In *Theory and applications of graphs*, pages 225–235. Springer, 1978.

10 Yuxing Han, Ziniu Wu, Peizhi Wu, Rong Zhu, Jingyi Yang, Liang Wei Tan, Kai Zeng, Gao Cong, Yanzhao Qin, Andreas Pfadler, et al. Cardinality estimation in dbms: A comprehensive benchmark evaluation. *arXiv preprint arXiv:2109.05877*, 2021.

11 Axel Hertzschuch, Claudio Hartmann, Dirk Habich, and Wolfgang Lehner. Simplicity done right for join ordering. In *11th Conference on Innovative Data Systems Research, CIDR 2021, Virtual Event, January 11-15, 2021, Online Proceedings*. www.cidrdb.org, 2021. URL: `http://cidrdb.org/cidr2021/papers/cidr2021_paper01.pdf`.

12 Mahmoud Abo Khamis, Hung Q. Ngo, and Atri Rudra. FAQ: questions asked frequently. 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 13–28. ACM, 2016. `doi:10.1145/2902251.2902280`.

13 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`.

14 Mahmoud Abo Khamis, Hung Q. Ngo, and Dan Suciu. What do shannon-type inequalities, submodular width, and disjunctive datalog have to do with one another? 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 429–444. ACM, 2017. `doi:10.1145/3034786.3056105`.

15 Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. How good are query optimizers, really? *Proc. VLDB Endow.*, 9(3):204–215, 2015. `doi:10.14778/2850583.2850594`.

16 Jie Liu, Wenqian Dong, Dong Li, and Qingqing Zhou. Fauce: Fast and accurate deep ensembles with uncertainty for cardinality estimation. *Proc. VLDB Endow.*, 14(11):1950–1963, 2021. URL: `http://www.vldb.org/pvldb/vol14/p1950-liu.pdf`, `doi:10.14778/3476249.3476254`.

17 Parimarjan Negi, Ryan C. Marcus, Andreas Kipf, Hongzi Mao, Nesime Tatbul, Tim Kraska, and Mohammad Alizadeh. Flow-loss: Learning cardinality estimates that matter. *Proc. VLDB Endow.*, 14(11):2019–2032, 2021. URL: `http://www.vldb.org/pvldb/vol14/p2019-negi.pdf`, `doi:10.14778/3476249.3476259`.

18 Hung Q. Ngo. Worst-case optimal join algorithms: Techniques, results, and open problems. In Jan Van den Bussche and Marcelo Arenas, editors, *Proceedings of the 37th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, Houston, TX, USA, June 10-15, 2018*, pages 111–124. ACM, 2018. `doi:10.1145/3196959.3196990`.

19 Yeonsu Park, Seongyun Ko, Sourav S. Bhowmick, Kyoungmin Kim, Kijae Hong, and Wook-Shin Han. G-CARE: A framework for performance benchmarking of cardinality estimation techniques for subgraph matching. In David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo, editors, *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, pages 1099–1114. ACM, 2020. `doi:10.1145/3318464.3389702`.

20 Ji Sun, Guoliang Li, and Nan Tang. Learned cardinality estimation for similarity queries. In Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava, editors, *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, pages 1745–1757. ACM, 2021. `doi:10.1145/3448016.3452790`.

21 Xiaoying Wang, Changbo Qu, Weiyuan Wu, Jiannan Wang, and Qingqing Zhou. Are we ready for learned cardinality estimation? *Proc. VLDB Endow.*, 14(9):1640–1654, 2021. URL: `http://www.vldb.org/pvldb/vol14/p1640-wang.pdf`, `doi:10.14778/3461535.3461552`.

22 Peizhi Wu and Gao Cong. A unified deep model of learning from both data and queries for cardinality estimation. In Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava, editors, *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, pages 2009–2022. ACM, 2021. doi:10.1145/3448016.3452830.

23 Zongheng Yang, Amog Kamsetty, Sifei Luan, Eric Liang, Yan Duan, Xi Chen, and Ion Stoica. Neurocard: One cardinality estimator for all tables. *Proc. VLDB Endow.*, 14(1):61–73, 2020. doi:10.14778/3421424.3421432.

24 Rong Zhu, Ziniu Wu, Yuxing Han, Kai Zeng, Andreas Pfadler, Zhengping Qian, Jingren Zhou, and Bin Cui. FLAT: fast, lightweight and accurate method for cardinality estimation. *Proc. VLDB Endow.*, 14(9):1489–1502, 2021. URL: http://www.vldb.org/pvldb/vol14/p1489-zhu.pdf, doi:10.14778/3461535.3461539.

Absolute Expressiveness of Subgraph-Based Centrality Measures

Andreas Pieris
University of Edinburgh, UK
University of Cyprus, Nicosia, Cyprus

Jorge Salas
Pontificia Universidad Católica de Chile, Santiago, Chile
University of Edinburgh, UK

Abstract

In graph-based applications, a common task is to pinpoint the most important or "central" vertex in a (directed or undirected) graph, or rank the vertices of a graph according to their importance. To this end, a plethora of so-called centrality measures have been proposed in the literature. Such measures assess which vertices in a graph are the most important ones by analyzing the structure of the underlying graph. A family of centrality measures that are suited for graph databases has been recently proposed by relying on the following simple principle: the importance of a vertex in a graph is relative to the number of "relevant" connected subgraphs surrounding it; we refer to the members of this family as subgraph-based centrality measures. Although it has been shown that such measures enjoy several favourable properties, their absolute expressiveness remains largely unexplored. The goal of this work is to precisely characterize the absolute expressiveness of the family of subgraph-based centrality measures by considering both directed and undirected graphs. To this end, we characterize when an arbitrary centrality measure is a subgraph-based one, or a subgraph-based measure relative to the induced ranking. These characterizations provide us with technical tools that allow us to determine whether well-established centrality measures are subgraph-based. Such a classification, apart from being interesting in its own right, gives useful insights on the structural similarities and differences among existing centrality measures.

2012 ACM Subject Classification Mathematics of computing → Graph theory; Information systems → Graph-based database models

Keywords and phrases Graph centrality measures, ranking, expressiveness

Digital Object Identifier 10.4230/LIPIcs.ICDT.2023.9

Related Version *Full Version*: `https://arxiv.org/abs/2206.06137`

Funding *Andreas Pieris*: Pieris was supported by the EPSRC grant EP/S003800/1 "EQUID".
Jorge Salas: Salas was supported by ANID – Millennium Science Initiative Program – Code ICN17_002.

1 Introduction

Graphs are well-suited for representing complex networks such as biological networks, cognitive and semantic networks, computer networks, and social networks, to name a few. In many applications that involve (directed or undirected) graphs, a crucial task is to pinpoint the most important or "central" vertex in a graph, or rank the vertices of a graph according to their importance. Indeed, these graph-theoretic tasks naturally appear in many different contexts, for example, finding people who are more likely to spread a disease in the event of an epidemic [4], highlighting cancer genes in proteomic data [7], assessing the importance of websites by search engines [12], identifying influencers in social networks [6], and many more.

To this end, a plethora of centrality measures have been proposed that assess the importance of a vertex in a graph [3, 11]. Centrality measures have been also studied in a principled way with the aim of providing axiomatic characterizations via structural properties over certain classes of graphs; see, e.g., [8, 18, 19].

It is not surprising that centrality measures have been also considered in the context of graph-structured data. Major graph database management systems such as Neo4j[1] and TigerGraph[2], have already adopted and implemented several centrality measures and algorithms in their Graph Data Science library such as Eigenvector [2], PageRank [12], Closeness [15], and many others. Moreover, applications of centrality measures have recently emerged in the context of knowledge graphs for entity linking [9], and Semantic Web search engines where ranking results is a central task [5].

Several existing centrality measures rely on the following intuitive principle: the importance of a vertex in a graph is relative to the number of *connected subgraphs* (e.g., triangles, paths, or cliques) surrounding it. We refer to such measures as *subgraph-based*. Interestingly, subgraph-based centrality measures are of particular interest for graph-structured data since a connected subgraph can be understood as the potential graph patterns occurring in a graph database. Consider, for example, a property graph G, which is essentially a finite directed graph, and a language L of basic graph patterns [1]. The evaluation of a query Q from L over G, denoted $Q(G)$, is the set of vertices of G that comply with the graph pattern expressed by Q. It is reasonable to assume that the more queries Q's from L such that $v \in Q(G)$ exist, the more important v is in G (relative to L). This way of defining the importance of a vertex follows the general principle discussed above, where the relevant connected subgraphs are the basic graph patterns from the language L.

A framework for defining and studying subgraph-based centrality measures has been recently introduced by Riveros and Salas [14], where the importance of a vertex is defined as the logarithm of the number of connected subgraphs surrounding it. As explicitly discussed in [14], the choice of applying the logarithmic function is purely for technical simplicity, and one could adopt any function, which we call filtering function, that leads to a richer family of subgraph-based centrality measures. Note that [14] considered only undirected graphs, but we can naturally define subgraph-based centrality measures over directed graphs. The main outcome of the analysis performed in [14] is that subgraph-based centrality measures satisfy desirable theoretical properties, typically called axioms, provided that the underlying family of connected subgraphs enjoys certain properties.

Despite the thorough analysis performed in [14], the absolute expressiveness of the family of subgraph-based centrality measures remains largely unexplored. Our main objective is to delineate the limits of the family of subgraph-based measures for both directed and undirected graphs. More precisely, we would like to understand when an arbitrary centrality measure is a subgraph-based one, or when it induces the same ranking as a subgraph-based one.

Our Contributions. Our contributions can be summarized as follows:

- In Section 4, we provide a precise characterization of when an arbitrary centrality measure is subgraph-based. More precisely, we isolate a "bounded value" property P over centrality measures, which essentially states that the total number of distinct values that can be assigned to vertices surrounded by a certain number of connected subgraphs is bounded, and then show that a measure can be expressed as a subgraph-based one iff it enjoys P.

[1] `https://neo4j.com/docs/graph-data-science/current/algorithms/centrality/`
[2] `https://docs.tigergraph.com/graphml/current/centrality-algorithms/`

- We then proceed in Section 5 to characterize when an arbitrary centrality measure induces the same ranking as a subgraph-based measure. In this case, we isolate a "graph coloring" property P over centrality measures, and then show that a centrality measure can be expressed as a subgraph-based one relative to the induced ranking iff it enjoys P.
- In Section 6, we focus on the family of monotonic subgraph-based measures, i.e., subgraph-based measures with a monotonic filtering function, and provide analogous characterizations via refined properties in the spirit of the "bounded value" property discussed above. An interesting finding is that in the case of connected graphs, *every* centrality measure can be expressed as a monotonic subgraph-based measure relative to the induced ranking.
- We finally proceed in Section 7 to determine if established measures (such as PageRank, Eigenvector, and many others) are (monotonic) subgraph-based (relative to the induced ranking). Such a classification, apart from being interesting in its own right, provides insights on the structural similarities and differences among the considered measures.

Clarification Remark. In the rest of the paper, due to space constraints and for the sake of clarity, we focus on undirected graphs, but all the notions and results can be transferred to the case of directed graphs under the standard notion of weak connectedness.

2 Preliminaries

We recall the basics on undirected graphs and graph centrality measures. In the rest of the paper, we assume the countable infinite set $\mathbf{V}$ of *vertices*. For $n > 0$, let $[n] = \{1, \ldots, n\}$.

Undirected Graphs. An *undirected graph* (or simply *graph*) G is a pair (V, E), where V is a finite non-empty subset of $\mathbf{V}$ (the set of *vertices of* G), and $E \subseteq \{\{u, v\} \mid u, v \in V\}$ (the set of *edges of* G). For notational convenience, given a graph G, we write $V(G)$ and $E(G)$ for the set of its vertices and edges, respectively. We denote by $\mathbf{G}$ the set of all graphs, and by $\mathbf{VG}$ the set of vertex-graph pairs $\{(v, G) \in \mathbf{V} \times \mathbf{G} \mid v \in V(G)\}$. The *neighbourhood* of a vertex $v \in V(G)$ in G, denoted $N_G(v)$, is the set $\{u \in V(G) \mid \{u, v\} \in E(G)\}$. For $u \in N_G(v)$, we say that v and u are *adjacent* in G. For a vertex $v \in \mathbf{V}$, we write G_v for the graph $(\{v\}, \emptyset)$.

A *subgraph* of a graph G is a graph G' such that $V(G') \subseteq V(G)$ and $E(G') \subseteq E(G)$; we write $G' \subseteq G$ to indicate that G' is a subgraph of G. Note that the binary relation $\subseteq$ over graphs forms a partial order. We denote by $\mathrm{Sub}(G)$ all the subgraphs of G, that is, the set of graphs $\{G' \mid G' \subseteq G\}$. Given a set of vertices $S \subseteq V(G)$, the subgraph of G *induced* by S, denoted $G[S]$, is the subgraph G' of G such that $V(G') = S$ and $E(G') = \{\{u, v\} \in E(G) \mid u, v \in S\}$.

A *path* in G is a sequence of vertices $\pi = v_0, v_1, \ldots, v_n$, for $n \geq 0$, such that $\{v_i, v_{i+1}\} \in E(G)$ for every $0 \leq i < n$. We further say that π is a path from v_0 to v_n. The length of π, denoted $|\pi|$, is the number of edges in π, i.e., n. By convention, there exists a path of length 0 from a vertex to itself. The *distance* between two vertices $u, v \in V(G)$ in G, denoted $d_G(u, v)$, is defined as the length of a shortest path from u to v in G; if there is no path, then $d_G(u, v) = \infty$. We denote by $S_G(u, v)$ the set of all the shortest paths from u to v in G, that is, the set $\{\pi \mid \pi \text{ is a path from } u \text{ to } v \text{ in } G \text{ with } |\pi| = d_G(u, v)\}$.

A graph G is *connected* if, for every two distinct vertices $u, v \in V(G)$, there exists a path from u to v. We denote by $\mathsf{A}(v, G)$ the set of all connected subgraphs of G that contain v, that is, the set $\{G' \subseteq G \mid v \in V(G') \text{ and } G' \text{ is connected}\}$. By abuse of notation, we may treat $\mathsf{A}(\cdot, \cdot)$ as a function of the form $\mathbf{VG} \to \mathcal{P}(\mathbf{G})$; as usual, $\mathcal{P}(S)$ denotes the powerset of a set S. A *connected component* (or simply *component*) of G is an induced subgraph $G[S]$

of G, where $S \subseteq V(G)$, such that $G[S]$ is connected, and, for every $v \in V(G) \setminus S$, there is no path in G from v to a vertex of S. It is clear that whenever G is connected, the only component of G is G itself. We denote by $\mathrm{Comp}(G)$ all the components of G, that is, the set of graphs $\{G' \mid G' \text{ is a component of } G\}$. Let $K_v(G)$ be the set of vertices of the component of G containing the vertex v.

Two graphs G_1 and G_2 are *isomorphic*, denoted $G_1 \simeq G_2$, if there exists a bijective function $h : V(G_1) \to V(G_2)$ such that $\{v, u\} \in E(G_1)$ iff $\{h(v), h(u)\} \in E(G_2)$. Furthermore, given the vertices $v_1 \in V(G_1)$ and $v_2 \in V(G_2)$, we say that the pairs (v_1, G_1) and (v_2, G_2) are isomorphic, denoted $(v_1, G_1) \simeq (v_2, G_2)$, if $G_1 \simeq G_2$ witnessed by h and $h(v_1) = v_2$.

Centrality Measures. A centrality measure assigns a score to a vertex v in a graph G, which reflects the importance of v in G. In other words, we adopt the standard assumption that the higher the score of a vertex v in G, the more important or "central" v is in G. Furthermore, it is typically assumed that the values assigned by a measure to the vertices of a graph do not depend on the names of the vertices, but only on the structure of the graph. In other words, two isomorphic vertices occurring in isomorphic graphs should be assigned the same importance; the latter property is usually called *closure under isomorphism* or *anonymity*. The formal definition of the notion of centrality measure follows:

▶ **Definition 1** (Centrality Measure). *A* centrality measure *(or simply* measure*) is a function* $\mathsf{C} : \mathbf{VG} \to \mathbb{R}$ *such that, for every two pairs* $(v_1, G_1) \in \mathbf{VG}$ *and* $(v_2, G_2) \in \mathbf{VG}$*,* $(v_1, G_1) \simeq (v_2, G_2)$ *implies* $\mathsf{C}(v_1, G_1) = \mathsf{C}(v_2, G_2)$*.* ⌟

We proceed to recall three known centrality measures that will be used throughout the paper; more centrality measures are discussed in Section 7.

Stress. This is a well-known centrality measure introduced in the 1950s [16]. It measures the centrality of a vertex by counting the number of shortest paths that go via that vertex. For a graph G and a vertex $v \in V(G)$, let $S_G^v(u, w)$ be the set of paths $\{\pi \in S_G(u, w) \mid \pi \text{ contains } v\}$. The *stress centrality* of v in G is defined as follows:

$$\mathsf{Stress}(v, G) \;=\; \sum_{u,w \in V(G)\setminus\{v\}} |S_G^v(u, w)|\,.$$

All-Subgraphs. This measure was recently introduced in the context of graph databases [14]. It states that a vertex is more central if it participates in more connected subgraphs. Formally, given a graph G and a vertex $v \in V(G)$, the *all-subgraphs centrality* of v in G is

$$\mathsf{All\text{-}Subgraphs}(v, G) \;=\; \log_2 |\mathsf{A}(v, G)|.$$

Closeness. This is a well-known measure introduced back in the 1960s [15]. It is usually called a geometrical measure since it relies on the distance inside a graph. It essentially states that the closer a vertex is to everyone in the graph the more central it is. Formally, given a graph G and a vertex $v \in V(G)$, the *closeness centrality* of v in G is the ratio

$$\mathsf{Closeness}(v, G) = \frac{1}{\sum_{u \in K_v(G)} d_G(v, u)}.$$

Let us clarify that we define the sum of distances inside a component of G since the distance between two vertices in different components of G is by definition infinite.

3 Subgraph-based Centrality Measures

As already discussed in the Introduction, a natural way of measuring the importance of a vertex in a graph is to count the relevant connected subgraphs surrounding it, and then apply a certain filtering function from the non-negative integers to the reals on top of the count. Of course, the relevant subgraphs and the adopted filtering function are determined by the intention of the centrality measure. Interestingly, both the stress and the all-subgraphs centrality measures are actually based on this principle. Let us elaborate further on this. Consider a graph G and a vertex $v \in V(G)$:

- For the stress centrality, the important subgraphs for v in G are the shortest paths that go via v in G, and the filtering function is $f_{\times 2}(x) = 2x$ since each shortest path is counted twice. In other words, with G_π being the graph that corresponds to a path π,

$$\mathsf{Stress}(v, G) \;=\; f_{\times 2}\left(\left|\bigcup_{u,w \in V(G)\setminus\{v\}} \{G_\pi \mid \pi \in S^v_G(u, w)\}\right|\right).$$

- For the all-subgraphs centrality, the important subgraphs for v in G are the connected subgraphs of G that contain v, that is, the set $\mathsf{A}(v, G)$, and the filtering function is $\log_2$. Indeed, by definition, we have that

$$\mathsf{All\text{-}Subgraphs}(v, G) \;=\; \log_2 |\mathsf{A}(v, G)|.$$

We proceed to formalize the above simple principle, originally introduced in [14], which gives rise to a family of centrality measures, and then highlight our main research questions.

Subgraph-based Centrality Measures. We first need a mechanism that allows us to specify what are the important subgraphs for a vertex v in a graph G. This is done via the notion of *subgraph family*, which is defined as a function from vertex-graph pairs to sets of graphs that is closed under isomorphism, that is, a function $\mathsf{F} : \mathbf{VG} \to \mathcal{P}(\mathbf{G})$ such that:

- for every $(v, G) \in \mathbf{VG}$, $\mathsf{F}(v, G) \subseteq \mathsf{A}(v, G)$, that is, F assigns to each $(v, G) \in \mathbf{VG}$ a set of connected subgraphs of G surrounding v, and
- for every two pairs $(v_1, G_1) \in \mathbf{VG}$ and $(v_2, G_2) \in \mathbf{VG}$ such that $(v_1, G_1) \simeq (v_2, G_2)$ witnessed by h, there exists a bijection $\mu : \mathsf{F}(v_1, G_1) \to \mathsf{F}(v_2, G_2)$ such that, for every $G' \in \mathsf{F}(v_1, G_1)$, $\mu(G') = (\{h(v) \mid v \in V(G')\}, \{\{h(v), h(u)\} \mid (v, u) \in G'\})$.

We also need the notion of *filtering function*, which, as said above, is simply a function of the form $f : \mathbb{N} \to \mathbb{R}$. We are now ready to define subgraph-based centrality measures:

▶ **Definition 2** ($\langle \mathsf{F}, f\rangle$-measure). *Consider a subgraph family* F *and a filtering function* f*. The* $\langle \mathsf{F}, f\rangle$-measure *is the function* $\mathsf{C}\langle \mathsf{F}, f\rangle : \mathbf{VG} \to \mathbb{R}$ *such that, for every pair* $(v, G) \in \mathbf{VG}$*, it holds that* $\mathsf{C}\langle \mathsf{F}, f\rangle(v, G) = f(|\mathsf{F}(v, G)|)$. ⌟

Since, by definition, subgraph families are closed under isomorphism, it is straightforward to see that each $\langle \mathsf{F}, f\rangle$-measure defines a valid centrality measure.

▶ **Lemma 3.** *For a subgraph family* F *and a filtering function* f*, it holds that the* $\langle \mathsf{F}, f\rangle$-measure *is a centrality measure.*

We say that a centrality measure C is a subgraph-based centrality measure if there are a subgraph family F and a filtering function f such that C coincides with the $\langle \mathsf{F}, f\rangle$-measure, i.e., for every pair $(v, G) \in \mathbf{VG}$, $\mathsf{C}(v, G) = \mathsf{C}\langle \mathsf{F}, f\rangle(v, G)$. Coming back to our discussion on stress and all-subgraph centralities, assuming that S is the subgraph family such that

$$\mathsf{S}(v, G) \ = \bigcup_{u,w \in V(G) \setminus \{v\}} \{G_\pi \mid \pi \in S^v_G(u, w)\}\,,$$

it is straightforward to verify that

$$\mathsf{Stress} \ = \ \mathsf{C}\langle \mathsf{S}, f_{\times 2}\rangle \quad \text{and} \quad \mathsf{All\text{-}Subgraphs} = \mathsf{C}\langle \mathsf{A}, \log_2\rangle.$$

Main Research Questions. Having the family of subgraph-based centrality measures in place, the natural question that comes up concerns its absolute expressive power. In other words, we are interested in the following research question:

▶ **Question I. When is a centrality measure a subgraph-based centrality measure?**

One may wonder whether the above question is conceptually trivial in the sense that every centrality measure can be expressed as a subgraph-based centrality measure by choosing the subgraph family and the filtering function in the proper way as done for Stress and All-Subgraphs. It turns out that there are measures that are *not* subgraph-based.

▶ **Proposition 4.** *There is a centrality measure that is not a subgraph-based measure.*

Proof. Consider the centrality measure C such that, for every $(v, G) \in \mathbf{VG}$, it holds that $\mathsf{C}(v, G) \ = \ |V(G)|$, i.e., it simply assigns to each vertex v in a graph G the number of vertices occurring in G. It suffices to show that C is not subgraph-based even if we focus on the set of graphs $\mathbf{G}^\star$ consisting of $G_1 = (\{u_1\}, \emptyset)$, $G_2 = (\{u_2, v_2\}, \emptyset)$, and $G_3 = (\{u_3, v_3, w_3\}, \emptyset)$. By contradiction, assume that C is a subgraph-based measure over $\mathbf{G}^\star$. Thus, there exists a subgraph family F and a filtering function f such that, for every $G \in \mathbf{G}^\star$ and $v \in V(G)$, $\mathsf{C}(v, G) = \mathsf{C}\langle \mathsf{F}, f\rangle(v, G)$. We observe that:

1. For every $(v, G) \in \mathbf{V} \times \mathbf{G}^\star$ with $v \in V(G)$, it holds that $\mathsf{C}\langle \mathsf{F}, f\rangle(v, G) \in \{1, 2, 3\}$, i.e., we have three distinct values. This follows by the definition of $\mathsf{C} = \mathsf{C}\langle \mathsf{F}, f\rangle$.
2. For every $(v, G) \in \mathbf{V} \times \mathbf{G}^\star$, it holds that $|\mathsf{F}(v, G)| \in \{0, 1\}$, i.e., we have two possible sizes for the sets of connected subgraphs.

Now, by the pigeonhole principle, we can safely conclude that there are two distinct pairs $(v, G), (u, G') \in \mathbf{V} \times \mathbf{G}^\star$ with $|\mathsf{F}(v, G)| = |\mathsf{F}(u, G')|$ such that $\mathsf{C}\langle \mathsf{F}, f\rangle(v, G) \neq \mathsf{C}\langle \mathsf{F}, f\rangle(u, G')$. But this contradicts the fact that f is a function, and the claim follows. ◀

As we shall see, not only artificial measures as the one employed in the proof of Proposition 4, but also well-known centrality measures from the literature (such as Closeness) are *not* subgraph-based. We are going to prove such inexpressibility results by using the technical tools developed towards answering Question I.

In several applications that involve graphs, we are more interested in the relative than the absolute importance of a vertex in a graph. More precisely, we are interested in the ranking of the vertices of a graph induced by a measure C, and not in the absolute value assigned to a vertex by C. This brings us to the next technical notion:

▶ **Definition 5** (Induced Ranking). *Let* C *be a centrality measure. The* ranking induced by C*, denoted* $\mathrm{Rank}(\mathsf{C})$*, is the binary relation*

$$\{((u, G), (v, G)) \mid u, v \in V(G) \text{ and } \mathsf{C}(u, G) \leq \mathsf{C}(v, G)\}$$

over $\mathbf{VG}$*.* C *is a* subgraph-based centrality measure relative to the induced ranking *if there are a subgraph family* F *and a filtering function* f *with* $\mathrm{Rank}(\mathsf{C}) = \mathrm{Rank}(\mathsf{C}\langle \mathsf{F}, f\rangle)$. ⌟

Interestingly, although the measure employed in the proof of Proposition 4 is not subgraph-based, it is easy to show that it is a subgraph-based measure relative to the induced ranking. In particular, by defining the subgraph family F as $\mathsf{F}(v, G) = \{G_v\}$, for every $(v, G) \in \mathbf{VG}$, and the filtering function as the identity, it is not difficult to see that $\mathrm{Rank}(\mathsf{C}) = \mathrm{Rank}(\mathsf{C}\langle\mathsf{F}, f\rangle)$. This observation brings us to our next research question:

▶ **Question II. When is a centrality measure a subgraph-based centrality measure relative to the induced ranking?**

As we shall see, the above question is conceptually non-trivial, i.e., there are measures that are *not* subgraph-based measures relative to the induced ranking. In particular, we will see that there are well-established measures (such as Closeness) that are *not* subgraph-based centrality measures relative to the induced ranking. Such inexpressibility results are shown by exploiting the tools developed towards answering Question II.

4 Characterizing Subgraph-based Centrality Measures

We proceed to provide an answer to Question I. More precisely, our goal is to isolate a structural property P over centrality measures that precisely characterizes subgraph-based measures, that is, for an arbitrary measure C, C is a subgraph-based measure iff C enjoys P. Interestingly, the desired property can be somehow extracted from the proof of Proposition 4. The crucial intuition provided by that proof is that the absolute expressiveness of subgraph-based measures is tightly related to the amount of connected subgraphs that are available for assigning different centrality values to vertices. In other words, a measure that assigns "too many" values among vertices that are surrounded by "too few" connected subgraphs cannot be expressed as a subgraph-based measure. We proceed to formalize this intuition.

We first collect all the different values assigned by a centrality measure C to the vertices of a graph G that are surrounded by a bounded number of connected subgraphs of G. In particular, for $n > 0$, we define the set of real values

$$\mathrm{Val}^n_G(\mathsf{C}) \;=\; \{\mathsf{C}(v, G) \mid v \in V(G) \text{ and } |\mathsf{A}(v, G)| \leq n\}\,.$$

We can then easily collect all the values assigned by C to the vertices of $\mathbf{V}$ that are surrounded by a bounded number of connected subgraphs in some graph. In particular, for $n > 0$,

$$\mathrm{Val}^n(\mathsf{C}) \;=\; \bigcup_{G \in \mathbf{G}} \mathrm{Val}^n_G(\mathsf{C}).$$

We now define the following property over centrality measures:

▶ **Definition 6** (Bounded Value Property). *A measure* C *enjoys the* bounded value property *if, for every* $n > 0$, $|\mathrm{Val}^n(\mathsf{C})| \leq n + 1$. ⌟

The bounded value property captures the key intuition discussed above. It actually bounds the number of different values that can be assigned among vertices that are surrounded by a limited number of connected subgraphs; hence the name "bounded value property". Observe that the measure C devised in the proof of Proposition 4 does not enjoy the bounded value property; indeed, $|\mathrm{Val}^1(\mathsf{C})| \geq 3 > 2$. Interestingly, the bounded value property is all we need towards a precise characterization of subgraph-based measures.

▶ **Theorem 7.** *Consider a centrality measure* C. *The following statements are equivalent:*
1. C *is a subgraph-based centrality measure.*
2. C *enjoys the bounded value property.*

Proof. ($1 \Rightarrow 2$) By contradiction, assume that C does not enjoy the bounded value property, namely there exists an integer $n \geq 1$ such that $|\mathrm{Val}^n(\mathsf{C})| > n+1$. By hypothesis, C is a subgraph-based centrality measure, and thus, there exist a subgraph family F and a filtering function f such that the following holds: for every $(v, G) \in \mathbf{VG}$, $\mathsf{C}(v, G) = \mathsf{C}\langle \mathsf{F}, f\rangle(v, G)$. We now define the set

$$B_n \;=\; \{|\mathsf{F}(v, G)| \mid (v, G) \in \mathbf{VG} \text{ and } |\mathsf{A}(v, G)| \leq n\}\,.$$

Clearly, $|B_n| \leq n+1$ since $\mathsf{F}(v, G) \subseteq \mathsf{A}(v, G)$. Let $h : \mathrm{Val}^n(\mathsf{C}) \to B_n$ be such that

$$h(\mathsf{C}(v, G)) \;=\; |\mathsf{F}(v, G)|.$$

By the pigeonhole principle, h is not injective, i.e., there exist $\mathsf{C}(v_1, G_1)$ and $\mathsf{C}(v_2, G_2)$ such that $\mathsf{C}(v_1, G_1) \neq \mathsf{C}(v_2, G_2)$ but $|\mathsf{F}(v_1, G_1)| = |\mathsf{F}(v_2, G_2)|$. This contradicts the fact that $\mathsf{C}(v_1, G_1) = f(|\mathsf{F}(v_1, G_1)|) \neq f(|\mathsf{F}(v_2, G_2)|) = \mathsf{C}(v_2, G_2)$, and the claim follows.

($2 \Rightarrow 1$) The goal is to show that there exist a subgraph family F and a filtering function f such that, for every $(v, G) \in \mathbf{VG}$, $\mathsf{C}(v, G) = \mathsf{C}\langle \mathsf{F}, f\rangle(v, G)$. We start by defining a total order $\preceq_{\mathsf{C}}$ over the set of values $\mathrm{Val}(\mathsf{C}) = \bigcup_{i=1}^{\infty} \mathrm{Val}^i(\mathsf{C})$. By definition, for every $n, m > 0$ such that $n \leq m$, it holds that $\mathrm{Val}^n(\mathsf{C}) \subseteq \mathrm{Val}^m(\mathsf{C})$. In other words, as we increase the integer n we are adding new values to the set $\mathrm{Val}^n(\mathsf{C})$. We can now define the binary relation $\preceq_{\mathsf{C}}$ over $\mathrm{Val}(\mathsf{C})$ as follows: for each $a, b \in \mathrm{Val}(\mathsf{C})$, if there exists n such that $a \in \mathrm{Val}^n(\mathsf{C})$ but $b \notin \mathrm{Val}^n(\mathsf{C})$ then $a \preceq_{\mathsf{C}} b$, if not, then $a \preceq_{\mathsf{C}} b$ if $a \leq b$. It is easy to see that $\preceq_{\mathsf{C}}$ is a total order over $\mathrm{Val}(\mathsf{C})$, and thus, it is a total order over $\mathrm{Val}^n(\mathsf{C})$ for each $n > 0$. For notational convenience, in the rest of the proof we assume that $\mathrm{Val}(\mathsf{C}) = \{a_1, a_2, a_3, \ldots\}$ and $a_1 \preceq_{\mathsf{C}} a_2 \preceq_{\mathsf{C}} a_3 \preceq_{\mathsf{C}} \cdots$.

By exploiting the total order $\preceq_{\mathsf{C}}$ over $\mathrm{Val}(\mathsf{C})$, we proceed to define a subgraph family F. Consider an arbitrary pair $(v, G) \in \mathbf{VG}$, and let $n = |\mathsf{A}(v, G)|$. By hypothesis, C enjoys the bounded value property, which in turn implies that $|\mathrm{Val}^n(\mathsf{C})| \leq n+1$. Therefore, $\mathsf{C}(v, G)$, which belongs to $\{a_1, a_2, ..., a_{|\mathrm{Val}^n(\mathsf{C})|}\}$, is equal to $\mathrm{Val}^n(\mathsf{C})$. We further observe that $\mathsf{A}(v, G)$ is a finite set, and we let $\mathsf{A}(v, G) = \{S_1, S_2, \ldots, S_n\}$. Here we assume an arbitrary order for $\mathsf{A}(v, G)$ that has the following property: for every pair (v', G') with $(v, G) \simeq (v', G')$, assuming that $\mathsf{A}(v', G') = \{S'_1, S'_2, \ldots, S'_n\}$, it holds that $(v, S_i) \simeq (v'_i, S'_i)$ for every $i \in \{1, \ldots, n\}$. The subgraph family F is defined as follows:

$$\mathsf{C}(v, G) = a_i \quad \text{implies} \quad \mathsf{F}(v, G) = \{S_1, ..., S_{i-1}\}.$$

This is indeed a subgraph family since $\mathsf{F}(v, G) \subseteq \mathsf{A}(v, G)$, while the chosen order for $\mathsf{A}(v, G)$ and the fact that C is (by definition) closed under isomorphism ensures closure under isomorphism. Notice that $|\mathsf{F}(v, G)| = i - 1$ for $i \in \{1, \ldots, |\mathrm{Val}^n(\mathsf{C})| + 1\}$. Finally, we define the filtering function $f : \mathbb{N} \to \mathrm{Val}(\mathsf{C})$ as follows: for each $i \in \mathbb{N}$,

$$f(i) \;=\; a_{i+1}.$$

We proceed to show that F and f capture our intention, that is, for every $(v, G) \in \mathbf{VG}$, $\mathsf{C}(v, G) = \mathsf{C}\langle \mathsf{F}, f\rangle(v, G)$, which will establish Theorem 7 . Let $n = |\mathsf{A}(v, G)|$. If $\mathsf{C}(v, G) = a_i \in \mathrm{Val}^n(\mathsf{C})$, then $|\mathsf{F}(v, G)| = i-1$. Therefore, $f(|\mathsf{F}(v, G)|) = \mathsf{C}\langle \mathsf{F}, f\rangle(v, G) = \mathsf{C}(v, G)$. Conversely, if $\mathsf{C}\langle \mathsf{F}, f\rangle(v, G) = a_i$, then $|\mathsf{F}(v, G)| = i - 1$, and thus, by construction, $\mathsf{C}(v, G) = a_i$. ◀

The above characterization, apart from giving a definitive answer to Question I, it provides a useful tool for establishing inexpressibility results. To show that a centrality measure C is not a subgraph-based measure it suffices to show that there exists an integer $n > 0$ such that $|\mathrm{Val}^n(\mathsf{C})| > n+1$. For example, we can show that $|\mathrm{Val}^5(\mathsf{Closeness})| > 6$, and therefore:

▶ **Proposition 8.** Closeness *is not a subgraph-based measure.*

Without Theorem 7 in place, it is completely unclear how one can prove that $\mathsf{Closeness}$ (or any other established measure) is not a subgraph-based measure. More inexpressibility results concerning well-established centrality measures are discussed in Section 7.

5 Characterizing Subgraph-based Measures Relative to the Induced Ranking

We now focus on Question II. Our goal is to isolate a structural property P over centrality measures that precisely characterizes subgraph-based measures relative to the induced ranking, i.e., for an arbitrary measure C, C is subgraph-based relative to the induced ranking iff C enjoys the property P. It turns out that P can be defined by exploiting a certain notion of graph coloring relative to a centrality measure.

Graph Colorings. The high-level idea is to consider the sizes of the available subgraph families that can be assigned to a vertex v in a graph G, i.e., the set of integers $\{0, \ldots, |\mathsf{A}(v,G)|\}$, as available colors. We can then refer to a precoloring of $\mathbf{VG}$ (i.e., of all the possible graphs) as a function $pc : \mathbf{VG} \to \mathbb{N}$ that assigns to each vertex v in a graph G only available colors from $\{0, \ldots, |\mathsf{A}(v,G)|\}$. Then, the goal is to isolate certain properties of such a precoloring of $\mathbf{VG}$ that leads to the desired characterization, i.e., a measure C is subgraph-based relative to the induced ranking iff there exists a precoloring of $\mathbf{VG}$ that enjoys the properties in question. Such a characterization tells us that for a centrality measure being subgraph-based relative to the induced ranking is tantamount to the fact that there are enough colors (i.e., sizes of sugbraph families, but *without* considering their actual topological structure) that allow us to color $\mathbf{VG}$ in a valid way, namely in a way that the crucial properties are satisfied. We proceed to formalize the above discussion about colorings.

Given a set $S \subseteq \mathbf{VG}$, a *precoloring of* S is a function $pc : S \to \mathbb{N}$ such that, for every $(v, G) \in S$, $pc(v, G) \in \{0, \ldots, |\mathsf{A}(v,G)|\}$. The first key property of such a precoloring states that the values assigned by a measure C to the vertices of a graph G should be respected, i.e., vertices with different centrality values get different colors. This is formalized as follows:

▶ **Definition 9** (Non-Uniform C-Injectivity). *Consider a set $S \subseteq \mathbf{VG}$, and a precoloring $pc : S \to \mathbb{N}$ of S. Given a centrality measure C, we say that pc is* non-uniformly C-injective *if, for every $(u, G), (v, G) \in S$, $\mathsf{C}(u, G) \neq \mathsf{C}(v, G)$ implies $pc(u, G) \neq pc(v, G)$.* ⌟

The term non-uniform in the above definition refers to the fact that C-injectivity is only enforced inside a certain graph, and not across all the graphs mentioned in S, i.e., it might be the case that a non-uniformly C-injective precoloring of S assigns to $(u, G), (v, G')$, where $G \neq G'$ and $\mathsf{C}(u, G) \neq \mathsf{C}(v, G')$, the same color.

The second key property of a precoloring S states that S should be consistent with the induced ranking, not only inside a certain graph, but also among different graphs mentioned in S. In other words, if (u, G) comes before (v, G) and (u', G') comes before (v', G'), then one of the following should hold: (u, G) and (v', G') get different colors, or (u', G) and (v, G') get different colors. This is formalized as follows:

▶ **Definition 10** (C-Consistency). *Consider a set $S \subseteq \mathbf{VG}$, and a precoloring $pc : S \to \mathbb{N}$ of S. Given a measure C, we say that pc is* C-consistent *if, for every $(u, G), (v, G), (u', G'), (v', G') \in S$, the following holds: if $\mathsf{C}(u, G) < \mathsf{C}(v, G)$ and $\mathsf{C}(u', G') < \mathsf{C}(v', G')$, then $pc(u, G) \neq pc(v', G')$ or $pc(u', G) \neq pc(v, G')$.* ⌟

Putting together the above two properties over precolorings, we get the notion of C-colorability of a set $S \subseteq \mathbf{VG}$:

▶ **Definition 11** (C-Colorability). *We say that a set $S \subseteq \mathbf{VG}$ is* C-colorable*, for some measure* C*, if there exists a precoloring of S that is non-uniformly* C*-injective and* C*-consistent.* ⌟

The Characterization. Interestingly, C-colorability is all we need towards the desired characterization, namely a measure C is subgraph-based relative to the induced ranking iff $\mathbf{VG}$ (i.e., all possible graphs) is C-colorable. We further show that the C-colorability of $\mathbf{VG}$ is equivalent to the C-colorability of every finite set $S \subsetneq \mathbf{VG}$. The latter, apart from being interesting in its own right, it provides a tool that is more convenient than the C-colorability of $\mathbf{VG}$ for classifying measures as subgraph-based relative to the induced ranking.

▶ **Theorem 12.** *Consider a centrality measure* C*. The following statements are equivalent:*
1. C *is a subgraph-based centrality measure relative to the induced ranking.*
2. *Every finite set $S \subsetneq \mathbf{VG}$ is* C*-colorable.*
3. $\mathbf{VG}$ *is* C*-colorable.*

To show the above characterization, it suffices to establish the sequence of implications (1) ⇒ (2) ⇒ (3) ⇒ (1). The implication (1) ⇒ (2) is a rather easy one and its full proof is given below. The proofs of the implications (2) ⇒ (3) and (3) ⇒ (1) are more interesting and we discuss their key ingredients below.

Implication (1) ⇒ (2)

Since, by hypothesis, C is a subgraph-based measure relative to the induced ranking, there are a subgraph family F and a filtering function f such that $\mathrm{Rank}(\mathsf{C}) = \mathrm{Rank}(\mathsf{C}\langle \mathsf{F}, f\rangle)$. Given a finite set $S \subsetneq \mathbf{VG}$, we define the function $pc_S : S \to \mathbb{N}$ as follows: for every $(v, G) \in S$, $pc_S(v, G) = |\mathsf{F}(v, G)|$. It is clear that pc_S is a precoloring of S since, by definition, $\mathsf{F}(v, G) \subseteq \mathsf{A}(v, G)$, and thus, $pc_S(v, G) \in \{0, \ldots, |\mathsf{A}(v, G)|\}$. It remains to show that pc_S is non-uniformly C-injective and C-consistent, which in turn implies that S is C-colorable:

Non-uniformly C-injective. Since $\mathrm{Rank}(\mathsf{C}) = \mathrm{Rank}(\mathsf{C}\langle \mathsf{F}, f\rangle)$, for every $(u_1, G), (u_2, G) \in S$, it holds that $\mathsf{C}(u_1, G) \neq \mathsf{C}(u_2, G)$ iff $\mathsf{C}\langle \mathsf{F}, f\rangle(u_1, G) \neq \mathsf{C}\langle \mathsf{F}, f\rangle(u_2, G)$. Therefore, $pc_S(u_1, G) = |\mathsf{F}(u_1, G)| \neq |\mathsf{F}(u_2, G)| = pc_S(u_2, G)$, and the claim follows.

C-consistent. By contradiction, assume that there are $(v_1, G_1), (v_2, G_1), (u_1, G_2)$ and (u_2, G_2) such that $\mathsf{C}(v_1, G_1) < \mathsf{C}(v_2, G_1)$ and $\mathsf{C}(u_1, G_2) > \mathsf{C}(u_2, G_2)$ but $pc_S(u_1, G_2) = pc_S(v_1, G_1)$ and $pc_S(v_2, G_1) = pc_S(u_2, G_2)$. Therefore, $|\mathsf{F}(v_1, G_1)| = |\mathsf{F}(u_1, G_2)|$ and $|\mathsf{F}(v_2, G_1)| = |\mathsf{F}(u_2, G_2)|$. Consequently, using the fact that $\mathrm{Rank}(\mathsf{C}) = \mathrm{Rank}(\mathsf{C}\langle \mathsf{F}, f\rangle)$, $\mathsf{C}\langle \mathsf{F}, f\rangle(v_1, G_1) < \mathsf{C}\langle \mathsf{F}, f\rangle(v_2, G_1) = \mathsf{C}\langle \mathsf{F}, f\rangle(u_2, G_2) < \mathsf{C}\langle \mathsf{F}, f\rangle(u_1, G_2) = \mathsf{C}\langle \mathsf{F}, f\rangle(v_1, G_1)$, which is clearly a contradiction, and the claim follows.

Implication (2) ⇒ (3)

The proof of this implication heavily relies on an old result that goes back in 1949 by Rado [13] known as *Rado's Selection Principle.* We write $\mathcal{P}_{\mathsf{fin}}(A)$ for the finite powerset of a set A, i.e., the set that collects all the *finite* subsets of A. Furthermore, given a function $f : A \to B$, we write $f_{|C}$ for the restriction of f to $C \subseteq A$.

▶ **Theorem 13** (Rado's Selection Principle). *Let A and B be arbitrary sets. Assume that, for each $C \in \mathcal{P}_{\mathsf{fin}}(A)$, f_C is a function $C \to B$ (a so-called "local function"). Assume further that, for every $x \in A$, the set $\{f_C(x) \mid C \in \mathcal{P}_{\mathsf{fin}}(A)$ and $x \in C\}$ is finite. Then, there is a function $f : A \to B$ (a so-called "global function") such that, for every $C \in \mathcal{P}_{\mathsf{fin}}(A)$, there is $D \in \mathcal{P}_{\mathsf{fin}}(A)$ with $C \subsetneq D$ and $f_{|C} = f_{D|C}$.*

Several proofs and applications of Rado's Theorem can be found in [10]. We proceed to discuss how it is used to prove (2) $\Rightarrow$ (3). By hypothesis, for each $S \in \mathcal{P}_{\mathsf{fin}}(\mathbf{VG})$, there exists a precoloring of S, i.e., a function $pc_S : S \to \mathbb{N}$ that is non-uniformly C-injective and C-consistent. Since, for every $(v,G) \in \mathbf{VG}$, $\mathsf{A}(v,G)$ is finite, we can conclude that the following holds: for every $(v,G) \in \mathbf{VG}$, the set $\{pc_S(v,G) \mid S \in \mathcal{P}_{\mathsf{fin}}(\mathbf{VG}) \text{ and } (v,G) \in S\}$ is finite. This allows us to apply Theorem 13 with $A = \mathbf{VG}$ and $B = \mathbb{N}$. Therefore, there exists a function $f : \mathbf{VG} \to \mathbb{N}$ such that, for every $S \in \mathcal{P}_{\mathsf{fin}}(\mathbf{VG})$, there exists $S' \in \mathcal{P}_{\mathsf{fin}}(\mathbf{VG})$ with $S \subsetneq S'$ and $f_{|S} = pc_{S'|S}$. Interestingly, by exploiting the latter property of the function f guaranteed by Theorem 13, and the fact that, for each $S \in \mathcal{P}_{\mathsf{fin}}(\mathbf{VG})$, pc_S is a precoloring of S that is non-uniformly C-injective and C-consistent, it is not difficult to show that f is a precoloring of $\mathbf{VG}$ that is non-uniformly C-injective and C-consistent, and item (3) follows.

Implication (3) $\Rightarrow$ (1)

We finally discuss the proof of the last implication. The goal is to devise a subgraph family F and a filtering function f such that $\mathrm{Rank}(\mathsf{C}) = \mathrm{Rank}(\mathsf{C}\langle \mathsf{F}, f\rangle)$, which in turn proves item (1). By hypothesis, there exists a precoloring pc of $\mathbf{VG}$ that is non-uniformly C-injective and C-consistent. We define F in such way that, for every $(v,G) \in \mathbf{VG}$, $|\mathsf{F}(v,G)| = pc(v,G)$; note that such a subgraph family exists since $pc(v,G) \in \{0,\ldots,|\mathsf{A}(v,G)|\}$. Now, defining the filtering function f is a non-trivial task. Let R_{pc} be the relation

$$\{(i,j) \in \mathbb{N} \times \mathbb{N} \mid \text{ there are } (u,G),(v,G) \text{ in } \mathbf{VG} \text{ such that } \mathsf{C}(u,G) < \mathsf{C}(v,G), pc(u,G) = i, \text{ and } pc(v,G) = j\}.$$

The fact that pc is non-uniformly C-injective allows us to conclude that R_{pc} is irreflexive. Moreover, the C-consistency of pc implies that R_{pc} is asymmetric. Observe now that if we extend R_{pc} into a total order $R^{\star}_{pc}$ over $\mathbb{N}$, and then show that $R^{\star}_{pc}$ can be embedded into a carefully chosen countable subset N of $\mathbb{R}$, then we obtain the desired filtering function f, which assigns real numbers to the sizes of the subgraph families assigned to the pairs of $\mathbf{VG}$ by F as dictated by the embedding of $R^{\star}_{pc}$ into $N \subsetneq \mathbb{R}$. Let us now briefly discuss how this is done. The binary relation R_{pc} is first extended into the strict partial order R^{+}_{pc} by simply taking its transitive closure. Now, the fact that R^{+}_{pc} can be extended into a total order $R^{\star}_{pc}$ over $\mathbb{N}$ follows by the *order-extension principle* (a.k.a. *Szpilrajn Extension Theorem*), shown by Szpilrajn in 1930 [17], which essentially states that every partial order can be extended into a total order. Finally, the fact that $R^{\star}_{pc}$ can be embedded into $N \subsetneq \mathbb{R}$ is shown via the *back-and-forth method*, a technique for showing isomorphism between countably infinite structures satisfying certain conditions.

A Bounded-Value-Like Property. An interesting question is whether we can isolate a property in the spirit of the bounded value property (see Definition 6) that can characterize subgraph-based measures relative to the induced ranking. Despite our efforts, we have not managed to provide an answer to this question. On the other hand, we succeeded in isolating a bounded-value-like property that is a necessary condition for a measure being subgraph-based relative to the induced ranking. It is clear that the bounded value property is not enough towards a necessary condition since, as discussed in Section 3, there is a measure (see the one devised in the proof of Proposition 4) that is not subgraph-based, which means that it does not enjoy the bounded value property, but it is subgraph-based relative to the induced ranking. On the other hand, to our surprise, a *non-uniform* version of the bounded value property leads to the desired necessary condition. Let us make this more precise. The ranking induced by a measure C compares only the values of vertices of the same graph; a pair $((u, G), (v, G'))$, where $G \neq G'$, will never appear in $\text{Rank}(\mathsf{C})$. This led us to conjecture that for characterizing subgraph-based measures relative to the induced ranking, it suffices to bound the number of different values that can be assigned among vertices *inside the same graph* that are surrounded by a limited number of connected subgraphs. This leads to the non-uniform version of the bounded value property:

▶ **Definition 14** (Non-Uniform Bounded Value Property). *A measure* C *enjoys the* non-uniform bounded value property *if, for every* $n > 0$ *and* $G \in \mathbf{G}$, $|\text{Val}^n_G(\mathsf{C})| \leq n + 1$. ⌟

We can then show the following implication:

▶ **Proposition 15.** *Consider a centrality measure* C*. If there exists a precoloring of* $\mathbf{VG}$ *that is non-uniformly* C*-injective, then* C *enjoys the non-uniform bounded value property.*

Proof. Let pc be the non-uniform C-injective precoloring of $\mathbf{VG}$, which exists by hypothesis. Consider an arbitrary graph G and an integer $n > 0$. We define the set

$$S_n \;=\; \{pc(v, G) \mid |\mathsf{A}(v, G)| \leq n\}.$$

In simple words, S_n collects all the colors assigned by pc to vertices with at most n connected subgraphs surrounding them. We then have that $|\text{Val}^n_G(\mathsf{C})| \leq |S_n|$ since pc is non-uniformly C-injective. Since pc is a precoloring, $S_n \subseteq \{0, \ldots, n\}$, and thus, $|S_n| \leq n + 1$. This in turn implies that $|\text{Val}^n_G(\mathsf{C})| \leq n + 1$, and the claim follows. ◀

By combining Theorem 12 and Proposition 15, we get the following corollary, which states that the non-uniform bounded value property leads to the desired necessary condition:

▶ **Corollary 16.** *If a centrality measure is a subgraph-based measure relative to the induced ranking, then it enjoys the non-uniform bounded value property.*

The question whether the non-uniform bounded value property is also a sufficient condition is negatively settled by the next result:

▶ **Proposition 17.** *There exists a centrality measure that is not a subgraph-based measure relative to the induced ranking, but it enjoys the non-uniform bounded value property.*

Let us stress that Corollary 16 equips us with a convenient tool for showing that a measure C is not a subgraph-based measure relative to the induced ranking: it suffices to show that there is $n > 0$ and a graph G such that $|\text{Val}^n_G(\mathsf{C})| > n + 1$. In the case of closeness, we can show that there exists a graph G such that $|\text{Val}^5_G(\mathsf{Closeness})| > 6$, which in turn implies that:

▶ **Proposition 18.** Closeness *is not a subgraph-based measure relative to the induced ranking.*

More inexpressibility results of the above form concerning established centrality measures are presented and discussed in Section 7.

Connected Graphs. The proof of Proposition 8 establishes that Closeness is not a subgraph-based measure even if we concentrate on connected graphs. On the other hand, the proof of Proposition 18 heavily relies on the fact that the employed graphs are not connected. This observation led us ask ourselves whether Closeness is a subgraph-based measure relative to the induced ranking if we consider only connected graphs. It turned out that, for connected graphs, not only Closeness, but *every* measure is subgraph-based relative to the induced ranking. We proceed to formalize this discussion.

Let $\mathbf{VCG} = \{(v, G) \in \mathbf{VG} \mid G \text{ is connected}\}$. For an arbitrary centrality measure C, its version that operates only on connected graphs is defined as the function $\mathsf{ConC} : \mathbf{VCG} \to \mathbb{R}$ such that, for every $(v, G) \in \mathbf{VCG}$, $\mathsf{C}(v, G) = \mathsf{ConC}(v, G)$, i.e., it is the restriction of C over $\mathbf{VCG}$. We then say that ConC is a subgraph-based measure (resp., subgraph-based measure relative to the induced ranking) if there exist a subgraph family F and a filtering function f such that $\mathsf{ConC} = \mathsf{ConC}\langle\mathsf{F}, f\rangle$ (resp., $\mathrm{Rank}(\mathsf{C}) \cap \mathbf{VCG}^2 = \mathrm{Rank}(\mathsf{C}\langle\mathsf{F}, f\rangle) \cap \mathbf{VCG}^2$). We can then establish the following result:

▶ **Theorem 19.** *Consider a centrality measure* C. *It holds that* ConC *is a subgraph-based measure relative to the induced ranking.*

Proof. We are going to define a subgraph family F and a filtering function f such that $\mathrm{Rank}(\mathsf{C}) \cap \mathbf{VCG}^2 = \mathrm{Rank}(\mathsf{C}\langle\mathsf{F}, f\rangle) \cap \mathbf{VCG}^2$, which in turn implies that ConC is a subgraph-based measure relative to the induced ranking, as needed. Consider an arbitrary connected graph G. We first observe that, for every $v \in V(G)$, it holds that $|\mathsf{A}(v, G)| \geq |V(G)|$ since every path from v to any other vertex in G is a connected subgraph containing v. We then define the equivalence relation $\equiv_G$ over $V(G)$ as follows: $v \equiv_G u$ if $\mathsf{C}(v, G) = \mathsf{C}(u, G)$. Let $V(G)/_{\equiv_G} = \{C_1, \ldots, C_m\}$ be the equivalence classes of $\equiv_G$. We can assume, without loss of generality, that, for every $i, j \in [m]$, with $C_i = [v]_{\equiv_G}$ and $C_j = [u]_{\equiv_G}$, $i < j$ implies $\mathsf{C}(v, G) < \mathsf{C}(u, G)$. We then define the subgraph family F in such a way that, for every vertex $v \in V(G)$, $|\mathsf{F}(v, G)| = i - 1$ if $[v]_{\equiv_G} = C_i$.[3] Note that such a subgraph family F always exists since, as discussed above, $|\mathsf{A}(v, G)| \geq |V(G)|$, but we have that $|V(G)/_{\equiv_G}| \leq |V(G)|$. Note also that we can ensure that F is closed under isomorphism by using the same idea as in the proof of Theorem 7. Finally, we define the filtering function f in such a way that, for every $i \in \{0, \ldots, m-1\}$, $f(i) = i + 1$. It is now not difficult to verify that indeed $\mathrm{Rank}(\mathsf{C}) \cap \mathbf{VCG}^2 = \mathrm{Rank}(\mathsf{C}\langle\mathsf{F}, f\rangle) \cap \mathbf{VCG}^2$, and the claim follows. ◀

As discussed above, $\mathsf{ConCloseness}$ is not a subgraph-based measure (this is implicit in the proof of Proposition 8), whereas $\mathsf{ConCloseness}$ is a subgraph-based measure relative to the induced ranking (follows from Theorem 19). This reveals a striking difference between the two notions of expressiveness, that is, being subgraph-based or being subgraph-based realtive to the induced ranking, when focussing on connected graphs.

We conclude this section by stressing that Theorem 19 provides a unifying framework for all centrality measures in a practically relevant setting: connected graphs and induced ranking. Indeed, graphs in real-life scenarios, although might be non-connected, they typically consists of one dominant connected component and several small components that are usually neglected as, by default, the most important vertex appears in the dominant component. Moreover, in real-life graph-based applications, we are typically interested in the induced ranking rather than the absolute centrality values assigned to vertices.

6 Monotonic Filtering Functions

Until now, we considered arbitrary filtering functions without any restrictions. On the other hand, the filtering functions $f_{\times 2}$ and $\log_2$ used to express Stress and $\mathsf{All\text{-}Subgraphs}$, respectively, as subgraph-based measures are monotonic; formally, a filtering function f is *monotonic* if, for all $x, y \in \mathbb{N}$, $x \leq y$ implies $f(x) \leq f(y)$. It is natural to ask Questions I and

[3] Note that for pairs (u, G'), where G' is a non-connected graph, we can simply define $\mathsf{F}(u, G')$ as the empty set since it is irrelevant what F does over non-connected graphs.

II for *monotonic subgraph-based centrality measures*, i.e., subgraph-based centrality measures $\mathsf{C}\langle\mathsf{F}, f\rangle$ where f is monotonic. Needless to say, one can study a plethora of different families of subgraph-based centrality measures that use filtering functions with certain properties (e.g., linear functions, logarithmic functions, etc.). However, such a thorough analysis is beyond the scope of this work, and it remains the subject of future research.

Monotonic Subgraph-based Measures. We first give a result analogous to Proposition 4, showing that not all subgraph-based measures are monotonic, and thus, the bounded value property is not the answer to Question I in the case of monotonic subgraph-based measures.

▶ **Proposition 20.** *There is a subgraph-based centrality measure that is not monotonic.*

Proof. Let G_1 be the graph with just one isolated node $(\{v_1\}, \emptyset)$, and G_2 be the graph $(\{v_1, v_2, v_3\}, \{\{v_2, v_3\}\})$. Consider the (partial) function $\mathsf{C} : \mathbf{VG} \to \mathbb{R}$ defined as follows:

$$\mathsf{C}(v, G) \;=\; \begin{cases} 1 & G = G_2 \text{ and } v \in \{v_2, v_3\} \\ 2 & G = G_2 \text{ and } v = v_1 \\ 3 & G = G_1 \text{ and } v = v_1. \end{cases}$$

It is easy to see that C can be extended to a proper centrality measure $\hat{\mathsf{C}}$: for every pair $(u, G') \in \mathbf{VG}$ such that $(v, G) \simeq (u, G')$, where $(v, G) \in \{(v_1, G_1), (v_1, G_2), (v_2, G_2), (v_3, G_2)\}$, let $\hat{\mathsf{C}}(u, G') = \mathsf{C}(v, G)$, and in any other case let $\hat{\mathsf{C}}(u, G') = 1$. We first show that $\hat{\mathsf{C}}$ is a subgraph-based measure. Notice that, for every vertex $v \in \mathbf{V}$, $\hat{\mathsf{C}}(v, G_v) = \mathsf{C}(v_1, G_1) = 3$. Hence, we have only two options concerning the set of connected subgraphs assigned to the vertices of G_2 by a subgraph family, and the filtering function, which are the following: with G_{uv} being the single-edge graph $(\{u, v\}, \{\{u, v\}\})$, either

$$\mathsf{F}_1(v, G) \;=\; \begin{cases} \emptyset & v = v_1 \text{ and } G = G_1 \\ \{G_{v_1}\} & v = v_1 \text{ and } G = G_2 \\ \{G_{v_2}, G_{v_2v_3}\} & v = v_2 \text{ and } G = G_2 \\ \{G_{v_3}, G_{v_2v_3}\} & v = v_3 \text{ and } G = G_2 \end{cases}$$

with $f_1(0) = 3$, $f_1(1) = 2$ and $f_1(2) = 1$, or

$$\mathsf{F}_2(v, G) \;=\; \begin{cases} \{G_{v_1}\} & v = v_1 \text{ and } G = G_1 \\ \emptyset & v = v_1 \text{ and } G = G_2 \\ \{G_{v_2}, G_{v_2v_3}\} & v = v_2 \text{ and } G = G_2 \\ \{G_{v_3}, G_{v_2v_3}\} & v = v_3 \text{ and } G = G_2 \end{cases}$$

with $f_2(0) = 2$, $f_2(1) = 3$ and $f_2(2) = 1$. We can now extend F_1 and F_2 into subgraph families that are closed under isomorphism as follows: for every $(u, G') \in \mathbf{VG}$ with $(v, G) \simeq (u, G')$, if $(v, G) \in \{(v_1, G_1), (v_1, G_2), (v_2, G_2), (v_3, G_2)\}$, then $\mathsf{F}_1(v, G) \simeq \mathsf{F}_1(u, G')$ and $\mathsf{F}_2(v, G) \simeq \mathsf{F}_2(u, G')$, otherwise, $\mathsf{F}_1(u, G') = \emptyset$ and $\mathsf{F}_2(u, G') = \{G_u\}$. It is clear that $\hat{\mathsf{C}} = \mathsf{C}\langle\mathsf{F}_1, f_1\rangle = \mathsf{C}\langle\mathsf{F}_2, f_2\rangle$. Observe, however, that both f_1 and f_2 are not monotonic functions. ◀

The proof of Proposition 20 essentially tells us that the key reason why the subgraph-based measure $\hat{\mathsf{C}}$ is not monotonic is because the maximum centrality value is assigned to a vertex surrounded by few connected subgraphs. To formalize this intuition, we first collect all the different values x assigned by a measure C to the vertices of a graph G that are surrounded by "too many" connected subgraphs such that x does not exceed the maximum value assigned by C to the vertices of G surrounded by "too few" connected subgraphs. More precisely, for an integer $n > 0$, we define the set of values

$$\mathrm{BVal}_G^n(\mathsf{C}) \;=\; \left\{ x \in \bigcup_{m>0} \mathrm{Val}_G^m(\mathsf{C}) \mid x \notin \mathrm{Val}_G^n(\mathsf{C}) \;\text{ and }\; x < \max \mathrm{Val}_G^n(\mathsf{C}) \right\}.$$

We then define the set of values

$$\mathrm{BVal}^n(\mathsf{C}) \;=\; \bigcup_{G \in \mathbf{G}} \mathrm{BVal}_G^n(\mathsf{C}).$$

We can now define a refined version of the bounded value property, which provides a better upper bound for $|\mathrm{Val}^n(\mathsf{C})|$:

▶ **Definition 21** (Monotonic Bounded Value Property). *A centrality measure* C *enjoys the* monotonic bounded value property *if, for every* $n > 0$, $|\mathrm{Val}^n(\mathsf{C})| \leq n + 1 - |\mathrm{BVal}^n(\mathsf{C})|$. ⌟

It is not difficult to see that the measure C devised in the proof of Proposition 20 does not enjoy the monotonic bounded value property. Indeed, $\mathrm{Val}^1(\mathsf{C}) = \{1, 3\}$ and $\mathrm{BVal}^1 = \{2\}$, and thus, $|\mathrm{Val}^1(\mathsf{C})| = 2 > 1$. The above refinement of the bounded value property is all we need to get a precise characterization of monotonic subgraph-based measures; hence the name "monotonic bounded balue property".

▶ **Theorem 22.** *Consider a centrality measure* C. *The following statements are equivalent:*
1. C *is a monotonic subgraph-based centrality measure.*
2. C *enjoys the monotonic bounded value property.*

Induced Ranking. Concerning the expressiveness of monotonic subgraph-based centrality measures relative to the induced ranking, we can show that the non-uniform version of the monotonic bounded value property provides a precise characterization.

▶ **Definition 23** (Non-Uniform Monotonic Bounded Value Property). *A centrality measure* C *enjoys the* non-uniform monotonic bounded value property *if, for every integer* $n > 0$ *and graph* $G \in \mathbf{G}$, *it holds that* $|\mathrm{Val}_G^n(\mathsf{C})| \leq n + 1 - |\mathrm{BVal}_G^n(\mathsf{C})|$. ⌟

We can then establish the following characterization that is in striking difference with Theorem 12, which shows that the non-uniform bounded value property is only a necessary condition (but not a sufficient condition) for a centrality measure being subgraph-based relative to the induced ranking.

▶ **Theorem 24.** *Consider a centrality measure* C. *The following statements are equivalent:*
1. C *is a monotonic subgraph-based centrality measure relative to the induced ranking.*
2. C *enjoys the non-uniform monotonic bounded value property.*

Connected Graphs. Recall that the family of subgraph-based measures relative to the induced ranking provides a unifying framework for all centrality measures whenever we concentrate on connected graphs (see Theorem 19). Interestingly, a careful inspection of the proof of Theorem 19 reveals that this holds even for the family of monotonic subgraph-based measures relative to the induced ranking.

▶ **Theorem 25.** *Consider a centrality measure* C. *It holds that* ConC *is a monotonic subgraph-based measure relative to the induced ranking.*

Table 1 Subgraph-based Measures.

Measure	Absolute Values	Induced Ranking
Stress	✓	✓
All-Subgraphs	✓	✓
Degree	✓	✓
Cross-Clique	✓	✓
Closeness	$\times[\mathit{trees}]$	$\times$ and $\checkmark[\mathit{con}]$
Harmonic	$\times[\mathit{trees}]$	$\times$ and $\checkmark[\mathit{con}]$
PageRank	$\times[\mathit{trees}]$	$\times$ and $\checkmark[\mathit{con}]$
Eigenvector	$\times[\mathit{trees}]$	? and $\checkmark[\mathit{con}]$
Betweenness	? and $\checkmark[\mathit{trees}]$	? and $\checkmark[\mathit{con}]$

Table 2 Monotonic Subgraph-based Measures.

Measure	Absolute Values	Induced Ranking
$\star$	as in Table 1	as in Table 1
Betweenness	$\times[\mathit{con}]$ and $\checkmark[\mathit{trees}]$	$\times$ and $\checkmark[\mathit{con}]$

7 Classification

We proceed to determine whether existing measures belong to the family of (monotonic) subgraph-based measures (relative to the induced ranking) by exploiting the technical tools provided by the results of the previous sections. Such a classification, apart from being interesting in its own right, will provide insights on the structural similarities and differences among existing centrality measures. To this end, we focus on established measures from the literature and provide a rather complete classification depicted in Tables 1 and 2; due to space constraints, the formal definitions of the considered measures are omitted. The second (resp., third) column determines whether the measure C stated in the first column is subgraph-based (resp., subgraph-based relative to the induced ranking); ✓ means that it is, $\times$ means that it is not, $\times[\mathit{trees}]$ means that it is not even for trees, $\checkmark[\mathit{con}]$ means that it is over connected graphs, $\checkmark[\mathit{trees}]$ means that it is over trees, and ? means that it is open. Concerning Table 2, $\star$ refers to any measure considered in Table 1 apart from Betweenness, and $\times[\mathit{con}]$ means that the respective measure (i.e., Betweenness) is not monotonic subgraph-based even for connected graphs. Note that Table 2 is identical to Table 1, apart from Betweenness, which is provably not monotonic subgraph-based (relative to the induced ranking).

We would like to remark that the result $\checkmark[\mathit{con}]$ for Eigenvector in both tables holds for a broader class of graphs than connected graphs. Moreover, we can show that Betweenness is a (monotonic) subgraph-based measure (relative to the induced ranking) for a class of graphs that captures the class of trees and is incomparable to the class of connected graphs. For the sake of readability, we state our expressibility results only for trees and connected graphs.

Take-home Messages. We highlight the key take-home messages of the above classification, which we believe provide further insights concerning the centrality measures in question:

1. If we focus on the induced ranking rather than the absolute values over connected graphs, then the family of monotonic subgraph-based measures should be understood as a unifying framework that incorporates every other measure.
2. Our classification excludes a priori the adoption of certain centrality measures (e.g., Closeness, Harmonic, etc.) in applications where the importance of a vertex should be measured based on the connected subgraphs surrounding it.

3. Betweenness, which computes the percentage of the shortest paths in a graph going through a vertex, is of different nature compared to all the other measures. Notably, although it looks similar to Stress, it behaves in a significantly different way. The relationship of Betweenness with (monotonic) subgraph-based measures deserves further investigation.
4. There is a notable difference between the two feedback measures considered in our classification, namely PageRank and Eigenvector, that deserves further exploration. As mentioned above, Eigenvector is a (monotonic) subgraph-based measure relative to the induced ranking over a broader class $\mathcal{C}$ of graphs than connected graphs, whereas PageRank is provable *not* a subgraph-based measure over the class $\mathcal{C}$.

A Note on Directed Graphs. As discussed in the clarification remark at the end of the Introduction, although our analysis (including the classification of this section) focused on undirected graphs, all the notions and results can be transferred to directed graphs under the notion of weak connectedness. The only exception is the negative result $\times$[*trees*] for Eigenvector in Tables 1 and 2. Although we can show that for directed graphs, Eigenvector is not a (monotonic) subgraph-based centrality measure, it remains open whether this holds even for directed trees (i.e., directed graphs whose underlying undirected graph is a tree).

8 Conclusions

We have provided a rather complete picture concerning the absolute expressiveness of the family of (monotonic) subgraph-based centrality measures (relative to the induced ranking) by establishing precise characterizations. We have also presented a detailed classification of standard centrality measures by using the tools provided by the aforementioned characterizations. Although our development focused on undirected graphs, all the notions and results can be transferred to directed graphs under the standard notion of weak connectedness.

We would like to stress that the machinery on graph colorings, introduced in Section 5, can be used to provide characterizations for all the families considered in the paper, and not only for the family of subgraph-based measures relative to the induced ranking. For example, we can show that a measure C is subgraph-based iff there exists a precoloring of **VG** that is uniformly C-injective; the latter is defined as non-uniform C-injectivity with the difference that C-injectivity is enforced across all the graphs (not only inside a certain graph).

The obvious question that remains open is whether we can isolate a bounded-value-like property that characterizes subgraph-based measures relative to the induced ranking. We believe that our coloring-based characterization (Theorem 24) is a useful tool towards such a bounded-value-like characterization. Finally, towards a deeper understanding of subgraph-based measures, one should perform a more refined analysis by focussing on restricted classes of subgraph families and filtering functions that enjoy desirable structural properties.

References

1 Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan L. Reutter, and Domagoj Vrgoc. Foundations of modern query languages for graph databases. *ACM Comput. Surv.*, 50(5):68:1–68:40, 2017.

2 Phillip Bonacich. Power and centrality: A family of measures. *American journal of sociology*, 92(5):1170–1182, 1987.

3 Stephen P. Borgatti and Martin G. Everett. A graph-theoretic perspective on centrality. *Soc. Networks*, 28(4):466–484, 2006.

4 Zoltán Dezső and Albert-László Barabási. Halting viruses in scale-free networks. *Phys. Rev. E*, 65:055103, 2002.

5 Aidan Hogan, Andreas Harth, Jürgen Umbrich, Sheila Kinsella, Axel Polleres, and Stefan Decker. Searching and browsing linked data with SWSE: the semantic web search engine. *J. Web Semant.*, 9(4):365–401, 2011.

6 Xinyu Huang, Dongming Chen, Dongqi Wang, and Tao Ren. Identifying influencers in social networks. *Entropy*, 22(4):450, 2020.

7 Gábor Iván and Vince Grolmusz. When the Web meets the cell: using personalized PageRank for analyzing protein interaction networks. *Bioinformatics*, 27(3):405–407, 2010.

8 Mitri Kitti. Axioms for centrality scoring with principal eigenvectors. *Social Choice and Welfare*, 46(3):639–653, 2016.

9 José-Lázaro Martínez-Rodríguez, Aidan Hogan, and Ivan López-Arévalo. Information extraction meets the semantic web: A survey. *Semantic Web*, 11(2):255–335, 2020.

10 Leonid Mirsky. *Transversal Theory: An Account of Some Aspects of Combinatorial Mathematics*. Academic Press, 1971.

11 Mark Newman. *Networks*. Oxford University Press, 2018.

12 Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, 1999.

13 Richard Rado. Axiomatic treatment of rank in infinite sets. *Canad. J. Math.*, pages 337–343, 1949.

14 Cristian Riveros and Jorge Salas. A family of centrality measures for graph data based on subgraphs. In *ICDT*, pages 23:1–23:18, 2020.

15 Gert Sabidussi. The centrality index of a graph. *Psychometrika*, 31(4):581–603, 1966.

16 Alfonso Shimbel. Structural parameters of communication networks. *Bull. Math. Biophysics*, 15:501–507, 1953.

17 Edward Szpilrajn. Sur l'extension de l'ordre partiel. *Fundamenta Matematicae*, 16:386–389, 1930.

18 René van den Brink and Robert P. Gilles. Measuring domination in directed networks. *Social Networks*, 22(2):141–157, 2000.

19 Tomasz Was and Oskar Skibski. Axiomatization of the pagerank centrality. In *IJCAI*, pages 3898–3904, 2018.

Diversity of Answers to Conjunctive Queries

Timo Camillo Merkl
TU Wien, Austria

Reinhard Pichler
TU Wien, Austria

Sebastian Skritek
TU Wien, Austria

Abstract

Enumeration problems aim at outputting, without repetition, the set of solutions to a given problem instance. However, outputting the entire solution set may be prohibitively expensive if it is too big. In this case, outputting a small, sufficiently diverse subset of the solutions would be preferable. This leads to the Diverse-version of the original enumeration problem, where the goal is to achieve a certain level d of diversity by selecting k solutions. In this paper, we look at the Diverse-version of the query answering problem for Conjunctive Queries and extensions thereof. That is, we study the problem if it is possible to achieve a certain level d of diversity by selecting k answers to the given query and, in the positive case, to actually compute such k answers.

2012 ACM Subject Classification Information systems → Data management systems

Keywords and phrases Query Answering, Diversity of Solutions, Complexity, Algorithms

Digital Object Identifier 10.4230/LIPIcs.ICDT.2023.10

Related Version *Full Version*: `https://arxiv.org/abs/2301.08848` [28]

Funding This work was supported by the Austrian Science Fund (FWF) project P30930-N35.

1 Introduction

The notion of *solutions* is ubiquitous in Computer Science and there are many ways of defining computational problems to deal with them. Decision problems, for instance, may ask if the set of solutions is non-empty or test for a given candidate if it indeed is a solution. Search problems aim at finding a concrete solution and counting problems aim at determining the number of solutions. In recent time, *enumeration problems*, which aim at outputting, without repetition, the set of solutions to a given problem instance have gained a lot of interest, which is, for instance, witnessed by two recent Dagstuhl seminars on this topic [8, 16]. Also in the Database Theory community, enumeration problems have played a prominent role on the research agenda recently, see e.g., [3, 24, 27]. Here, the natural problem to consider is query answering with the answers to a given query constituting the "solutions" to this problem.

It is well known that even seemingly simple problems, such as answering an acyclic Conjunctive Query, can have a huge number of solutions. Consequently, specific notions of tractability were introduced right from the beginning of research on enumeration problems [22] to separate the computational intricacy of a problem from the mere size of the solution space. However, even with these refined notions of tractability, the usefulness of flooding the user with tons of solutions (many of them possibly differing only minimally) may be questionable. If the solution space gets too big, it would be more useful to provide an overview by outputting a "meaningful" subset of the solutions. One way of pursuing this goal is to randomly select solutions (also known as "sampling") as was, for instance, done in [4, 9]. In fact, research on sampling has a long tradition in the Database community [11] – above all with the goal of supporting more accurate cardinality estimations [25, 26, 36].

A different approach to providing a "meaningful" subset of the solution space aims at outputting a small *diverse* subset of the solutions. This approach has enjoyed considerable popularity in the Artificial Intelligence community [5, 15, 29] – especially when dealing with Constraint Satisfaction Problems (CSPs) [20, 21, 30]. For instance, consider a variation of the car dealership example from [20]. Suppose that I models the preferences of a customer and $\mathcal{S}(I)$ are all cars that match these restrictions. Now, in a large dealership, presenting all cars in $\mathcal{S}(I)$ to the customer would be infeasible. Instead, it would be better to go through a rather small list of cars that are significantly different from each other. With this, the customer can point at those cars which the further discussion with the clerk should concentrate on.

Due to the inherent hardness of achieving the maximal possible diversity [20], the Database community – apart from limited exceptions [13] – focused on heuristic and approximation methods to find diverse solutions (see [37] for an extensive survey). Also, in contrast to the present work, there diversification is usually treated as a post-processing task that is applied to a set of solutions after materializing it.

The goal of our work is therefore to broaden the understanding of the theoretical boundaries of diverse query answering and develop complementary exact algorithms. More specifically, we want to analyze diversity problems related to answering Conjunctive Queries (CQs) and extensions thereof. As pointed out in [21], to formalize the problems we are thus studying, we, first of all, have to fix a notion of *distance* between any two solutions and an *aggregator* to combine pairwise distances to a *diversity measure* for a set of solutions. For the distance between two answer tuples, we will use the Hamming distance throughout this paper, that is, counting the number of positions on which two tuples differ. As far as the choice of an aggregator f is concerned, we impose the general restriction that it must be computable in polynomial time. As will be detailed below, we will sometimes also consider more restricted cases of f. Formally, for a class $\mathcal{Q}$ of queries and diversity measure δ that maps k answer tuples to an aggregated distance, we will study the following problem Diverse-$\mathcal{Q}$:

Diverse-$\mathcal{Q}$

Input: A database instance I, query $Q \in \mathcal{Q}$, and integers k and d.

Question: Do there exist pairwise distinct answers $\gamma_1, \dots, \gamma_k \in Q(I)$ such that $\delta(\gamma_1, \dots, \gamma_k) \geq d$?

That is, we ask if a certain level d of diversity can be achieved by choosing k pairwise distinct answers to a given query Q over the database instance I. We refer to $\{\gamma_1, \dots, \gamma_k\}$ as the desired *diversity set.* As far as the notation is concerned, we will denote the Hamming distance between two answers γ, γ' by $\Delta(\gamma, \gamma')$. With diversity measure δ, we denote the aggregated Hamming distances of all pairs of k answer tuples for an arbitrary, polynomial-time computable aggregate function f. That is, let $f\colon \bigcup_{k \geq 1} \mathbb{N}^{\frac{k(k-1)}{2}} \to \mathbb{R}$ and let $d_{i,j} = \Delta(\gamma_i, \gamma_j)$ for $1 \leq i < j \leq k$. Then we define $\delta(\gamma_1, \dots, \gamma_k) := f((d_{i,j})_{1 \leq i < j \leq k})$. Moreover, we write δ_{sum}, δ_{min}, and δ_{mon} if the aggregator f is the minimum, the sum, or an arbitrary (polynomial-time computable) monotone function, i.e., $f(d_1, \dots, d_N) \leq f(d'_1, \dots, d'_N)$ whenever $d_i \leq d'_i$ holds for every $i \in \{1, \dots, N\}$ with $N = \frac{k(k-1)}{2}$. The corresponding diversity problems are denoted by Diverse$_{\mathsf{sum}}$-$\mathcal{Q}$, Diverse$_{\mathsf{min}}$-$\mathcal{Q}$, and Diverse$_{\mathsf{mon}}$-$\mathcal{Q}$, respectively.

When we prove upper bounds on the complexity of several variations of the Diverse-$\mathcal{Q}$ problem (in the form of membership in some favorable complexity class), we aim at the most general setting, i.e., the Diverse-$\mathcal{Q}$ problem for an arbitrary, polynomial-time computable aggregation function. However, in some cases, the restriction to sum, min, or mon will be needed in order to achieve the desired upper bound on the complexity. In contrast, to prove lower bounds (in the form of hardness results), we aim at restricted cases (in particular,

$\mathsf{Diverse}_{\mathsf{sum}}$-$\mathcal{Q}$ and $\mathsf{Diverse}_{\mathsf{min}}$-$\mathcal{Q}$). These hardness results, of course, carry over to the more general cases. Somehow surprisingly, our analyses will reveal differences in the complexity between the seemingly similar cases $\mathsf{Diverse}_{\mathsf{sum}}$-$\mathcal{Q}$ and $\mathsf{Diverse}_{\mathsf{min}}$-$\mathcal{Q}$.

We will analyze several variations of the Diverse-$\mathcal{Q}$ problem: as mentioned above, we study various aggregator functions. Moreover, we consider several query classes $\mathcal{Q}$ – starting with the class CQ of Conjunctive Queries and then extending our studies to the classes UCQ and $\mathsf{CQ}^{\neg}$ of unions of CQs and CQs with negation. In one case, we will also look at the class FO of all first-order queries. Recall that, for combined complexity and query complexity, even the question if an answer tuple exists at all is NP-complete for CQs [10]. We therefore mostly restrict our study to acyclic CQs (ACQs, for short) with the corresponding query classes ACQ and UACQ, allowing only ACQs and unions of ACQs, respectively. For CQs with negation, query answering remains NP-complete even if we only allow ACQs [32]. Hence, for $\mathsf{CQ}^{\neg}$ we have to impose a different restriction. We thus restrict ourselves to CQs with bounded treewidth. Finally note that, even if we have formulated Diverse-$\mathcal{Q}$ as a decision problem, we also care about actually computing k solutions in case of a yes-answer.

We aim at a thorough complexity analysis of the Diverse-$\mathcal{Q}$ problem from various angles. We thus mainly consider the problem parameterized by the size k of the diversity set. In the non-parameterized case (i.e., if k is simply part of the input) we assume k to be given in unary representation. This assumption is motivated by the fact that for binary representation of k, the size k of the diversity set can be exponentially larger than the input: this contradicts the spirit of the diversity approach which aims at outputting a *small* (not an exponentially big) number of diverse solutions. As is customary in the Database world, we will distinguish combined, query, and data complexity.

Summary of results.

- We start our analysis of the Diverse-$\mathcal{Q}$ problem with the class of ACQs and study data complexity, query complexity, and combined complexity. With the size k of the diversity set as the parameter, we establish XP-membership for combined complexity, which is strengthened to FPT-membership for data complexity. The XP-membership of combined complexity is complemented by a W[1]-lower bound of the $\mathsf{Diverse}_{\mathsf{sum}}$-ACQ and $\mathsf{Diverse}_{\mathsf{min}}$-ACQ problems. For the non-parameterized case, we show that even the data complexity is NP-hard.
- The FPT-result of data complexity is easily extended to unions of ACQs. Actually, it even holds for arbitrary FO-queries. However, rather surprisingly, we show that the combined complexity and even query complexity of the $\mathsf{Diverse}_{\mathsf{sum}}$-UACQ and $\mathsf{Diverse}_{\mathsf{min}}$-UACQ problems are NP-complete. The hardness still holds, if the size k of the diversity set is 2 and the UACQs are restricted to unions of 2 ACQs.
- Finally, we study the Diverse-$\mathcal{Q}$ problem for the class $\mathsf{CQ}^{\neg}$. As was mentioned above, the restriction to ACQs is not even enough to make the query answering problem tractable. We, therefore, study the Diverse-$\mathsf{CQ}^{\neg}$ problem by allowing only classes of CQs of bounded treewidth. The picture is then quite similar to the Diverse-ACQ problem, featuring analogous XP-membership, FPT-membership, W[1]-hardness, and NP-hardness results.

Structure. We present some basic definitions and results in Section 2. In particular, we will formally introduce all concepts of parameterized complexity (complexity classes, reductions) relevant to our study. We then analyze various variants of the Diverse-$\mathcal{Q}$ problem, where $\mathcal{Q}$ is the class of CQs in Section 3, the class of unions of CQs in Section 4, and the class of CQs with negation in Section 5, respectively. Some conclusions and directions for future work are given in Section 6. Due to the space limit, most proofs are only sketched. Full details are provided in [28].

2 Preliminaries

Basics. We assume familiarity with relational databases. For basic notions such as schema, (arity of) relation symbols, relations, (active) domain, etc., the reader is referred to any database textbook, e.g., [1]. A CQ is a first-order formula of the form $Q(X) := \exists Y \bigwedge_{i=1}^{\ell} A_i$ with free variables $X = (x_1, \dots, x_m)$ and bound variables $Y = (y_1, \dots, y_n)$ such that each A_i is an atom with variables from $x_1, \dots, x_m, y_1, \dots, y_n$. An answer to such a CQ $Q(X)$ over a database instance (or simply "database", for short) I is a mapping $\gamma\colon X \to \mathit{dom}(I)$ which can be extended to a mapping $\bar{\gamma}\colon (X \cup Y) \to \mathit{dom}(I)$ such that instantiating each variable $z \in (X \cup Y)$ to $\bar{\gamma}(z)$ sends each atom A_i into the database I. We write $\mathit{dom}(I)$ to denote the (finite, active) domain of I. By slight abuse of notation, we also refer to the tuple $(\gamma(X) = \gamma(x_1), \dots, \gamma(x_m))$ as an answer (or an answer tuple). A UCQ is a disjunction $\bigvee_{i=1}^{N} Q_i(X)$, where all Q_i's are CQs with the same free variables. The set of answers of a UCQ is the union of the answers of its CQs. In a CQ with negation, we allow the A_i's to be either (positive) atoms or literals (i.e., negated atoms) satisfying a safety condition, i.e., every variable has to occur in some positive atom. An answer to a CQ with negation $Q(X)$ over a database I has to satisfy the condition that each positive atom is sent to an atom in the database while each negated atom is not. The set of answers to a query Q over a database I is denoted by $Q(I)$.

For two mappings α and α' defined on variable sets Z and Z', respectively, we write $\alpha \cong \alpha'$ to denote that the two mappings coincide on all variables in $Z \cap Z'$. If this is the case, we write $\alpha \cap \alpha'$ and $\alpha \cup \alpha'$ to denote the mapping obtained by restricting α and α' to their common domain or by combining them to the union of their domains, respectively. That is, $(\alpha \cap \alpha')(z) = \alpha(z)$ for every $z \in Z \cap Z'$ and $(\alpha \cup \alpha')(z)$ is either $\alpha(z)$ if $z \in Z$ or $\alpha'(z)$ otherwise. For $X \subseteq Z$ and $z \in Z$, we write $\alpha|_X$ and $\alpha|_z$ for the mapping resulting from the restriction of α to the set X or the singleton $\{z\}$, respectively. Also, the Hamming distance between two mappings can be restricted to a subset of the positions (or, equivalently, of the variables): by $\Delta_X(\alpha, \alpha')$ we denote the number of variables in X on which α and α' differ.

Acyclicity and widths. In a landmark paper [34], Yannakakis showed that query evaluation is tractable (combined complexity) if restricted to *acyclic* CQs. A CQ is acyclic if it has a *join tree*. Given a CQ $Q(X) := \exists Y \bigwedge_{i=1}^{\ell} A_i$ with $\mathit{At}(Q(X)) = \{A_i : 1 \leq i \leq \ell\}$, a join tree of $Q(X)$ is a triple $\langle T, \lambda, r\rangle$ such that $T = (V(T), E(T))$ is a rooted tree with root r and $\lambda\colon V(T) \to \mathit{At}(Q(X))$ is a node labeling function that satisfies the following properties:

1. The labeling λ is a bijection.
2. For every $v \in X \cup Y$, the set $T_v = \{t \in V(T) : v \text{ occurs in } \lambda(t)\}$ induces a subtree $T[T_v]$.

Testing if a given CQ is acyclic and, in case of a yes-answer, constructing a join tree is feasible in polynomial time by the GYO-algorithm, named after the authors of [19, 35].

Another approach to making CQ answering tractable is by restricting the *treewidth* (*tw*), which is defined via *tree decompositions* [31]. Treewidth does not generalize acyclicity, i.e., a class of acyclic CQs can have unbounded *tw*. We consider *tw* here only for CQs with negation. Let $Q(X) := \exists Y \bigwedge_{i=1}^{\ell} L_i$, be a CQ with negation, i.e., each L_i is a (positive or negative) literal. Moreover, let $\mathit{var}(L_i)$ denote the variables occurring in L_i. A tree decomposition of $Q(X)$ is a triple $\langle T, \chi, r\rangle$ such that $T = (V(T), E(T))$ is a rooted tree with root r and $\chi\colon V(T) \to 2^{X \cup Y}$ is a node labeling function with the following properties:

1. For every L_i, there exists a node $t \in V(T)$ with $\mathit{var}(L_i) \subseteq \chi(t)$.
2. For every $v \in X \cup Y$, the set $T_v = \{t \in V(T) : v \in \chi(t)\}$ induces a subtree $T[T_v]$.

The sets $\chi(t)$ of variables are referred to as "bags" of the tree decomposition T. The width of a tree decomposition is defined as $\max_{t \in V(T)}(|\chi(t)| - 1)$. The treewidth of a CQ with negation Q is the minimum width of all tree decompositions of Q. For fixed ω, it is feasible in linear time w.r.t. the size of the query Q to decide if $tw(Q) \leq \omega$ holds and, in case of a yes-answer, to actually compute a tree decomposition of width $\leq \omega$ [6].

Complexity. We follow the categorization of the complexity of database tasks introduced in [33] and distinguish combined/query/data complexity of the Diverse-$\mathcal{Q}$ problem. That is, for data complexity, we consider the query Q as arbitrarily chosen but fixed, while for query complexity, the database instance I is considered fixed. In case of combined complexity, both the query and the database are considered as variable parts of the input.

We assume familiarity with the fundamental complexity classes P (polynomial time) and NP (non-deterministic polynomial time). We study the Diverse-$\mathcal{Q}$ problem primarily from a parameterized complexity perspective [14]. An instance of a *parameterized problem* is given as a pair (x, k), where x is the actual problem instance and k is a parameter – usually a non-negative integer. The effort for solving a parameterized problem is measured by a function that depends on both, the size $|x|$ of the instance and the value k of the parameter. The asymptotic worst-case time complexity is thus specified as $\mathcal{O}(f(n, k))$ with $n = |x|$.

The parameterized analogue of *tractability* captured by the class P is *fixed-parameter tractability* captured by the class FPT of fixed-parameter tractable problems. A problem is in FPT, if it can be solved in time $\mathcal{O}(f(k) \cdot n^c)$ for some computable function f and constant c. In other words, the run time only depends polynomially on the size of the instance, while a possibly exponential explosion is confined to the parameter. In particular, if for a class of instances, the parameter k is bounded by a constant, then FPT-membership means that the problem can be solved in polynomial time. This also applies to problems in the slightly less favorable complexity class XP, which contains the problems solvable in time $\mathcal{O}(n^{f(k)})$.

Parameterized complexity theory also comes with its own version of reductions (namely "FPT-reductions") and hardness theory based on classes of fixed-parameter intractable problems. An FPT-reduction from a parameterized problem P to another parameterized problem P' maps every instance (x, k) of P to an equivalent instance (x', k') of P', such that k' only depends on k (i.e., independent of x) and the computation of x' is in FPT (i.e., in time $\mathcal{O}(f(k) \cdot |x|^c)$ for some computable function f and constant c). For *fixed-parameter intractability*, the most prominent class is W[1]. It has several equivalent definitions, for instance, W[1] is the class of problems that allow for an FPT-reduction to the Independent Set problem parameterized by the desired size k of an independent set. We have FPT $\subseteq$ W[1] $\subseteq$ XP. It is a generally accepted assumption in parameterized complexity theory that FPT $\neq$ W[1] holds – similar but slightly stronger than the famous P $\neq$ NP assumption in classical complexity theory, i.e., FPT $\neq$ W[1] implies P $\neq$ NP, but not vice versa.

3 Diversity of Conjunctive Queries

3.1 Combined and Query Complexity

We start our study of the Diverse-ACQ problem by considering the combined complexity and then, more specifically, the query complexity. We will thus present our basic algorithm in Section 3.1.1, which allows us to establish the XP-membership of this problem. We will then prove W[1]-hardness in Section 3.1.2 and present some further improvements of the basic algorithm in Section 3.1.3.

3.1.1 Basic Algorithm

Our algorithm for solving Diverse-ACQ is based on a dynamic programming idea analogous to the Yannakakis algorithm. Given a join tree $\langle T, \lambda, r\rangle$ and database I, the Yannakakis algorithm decides in a bottom-up traversal of T at each node $t \in V(T)$ and for each answer α to the single-atom query $\lambda(t)$ if it can be extended to an answer to the CQ consisting of all atoms labeling the nodes in the complete subtree T' rooted at t. It then stores this (binary) information by either keeping or dismissing α. Our algorithm for Diverse-ACQ implements a similar idea. At its core, it stores *k-tuples* $(\alpha_1, \ldots, \alpha_k)$ of answers to the single-atom query $\lambda(t)$, each k-tuple describing a set of (partial) diversity sets. We extend this information by the various vectors $(d_{i,j})_{1\leq i<j\leq k}$ of Hamming distances that are attainable by possible extensions $(\gamma_1, \ldots, \gamma_k)$ to the CQ consisting of the atoms labeling the nodes in T'.

In the following, we consider an ACQ $Q(X) := \exists Y \bigwedge_{i=1}^{\ell} A_i$ where each atom is of the form $A_i = R_i(Z_i)$ for some relation symbol R_i and variables $Z_i \subseteq X \cup Y$. For an atom $A = R(Z)$ and a database instance I, define $A(I)$ as the set of mappings $\{\alpha\colon Z \to \mathit{dom}(I) : \alpha(Z) \in R^I\}$. We extend the definition to sets (or conjunctions) $\psi(Z)$ of atoms $A_i(Z_i)$ with $Z_i \subseteq Z$. Then $\psi(I)$ is the set of mappings $\{\alpha\colon Z \to \mathit{dom}(I) : \alpha(Z_i) \in R_i^I$ for all $R_i(Z_i) \in \psi(Z)\}$. Let $\langle T, \lambda, r\rangle$ be a join tree. For a subtree T' of T we define $\lambda(T') = \{\lambda(t) : t \in V(T')\}$ and, by slight abuse of notation, we write $t(I)$ and $T'(I)$ instead of $\lambda(t)(I)$ and $\lambda(T')(I)$. Now consider T' to be a subtree of T with root t. For tuples $e \in \{(\alpha_1, \ldots, \alpha_k, (d_{i,j})_{1\leq i<j\leq k}) : \alpha_1, \ldots, \alpha_k \in t(I), d_{i,j} \in \{0, \ldots, |X|\}$ for $1 \leq i < j \leq k\}$, we define $\mathit{ext}_{T'}(e) = \{(\gamma_1, \ldots, \gamma_k) : \gamma_1, \ldots, \gamma_k \in T'(I)$ s.t. $\alpha_i \cong \gamma_i$ for $1 \leq i \leq k$ and $\Delta_X(\gamma_i, \gamma_j) = d_{i,j}$ for $1 \leq i < j \leq k\}$.

Intuitively, our algorithm checks for each such tuple e whether there exist extensions γ_i of α_i that (a) are solutions to the subquery induced by T' and (b) exhibit $d_{i,j}$ as their pairwise Hamming distances. If this is the case, the tuple e is kept, otherwise, e is dismissed. In doing so, the goal of the algorithm is to compute sets $D_{T'}$ that contain exactly those e with $\mathit{ext}_{T'}(e) \neq \emptyset$. Having computed D_T (i.e., for the whole join tree), Diverse-ACQ can now be decided by computing for each $e \in D_T$ the diversity measure from the values $d_{i,j}$.

To do so, in a first phase, at every node $t \in V(T)$, we need to compute and store the set $D_{T'}$ (for T' being the complete subtree rooted in t). We compute this set by starting with some set D_t and updating it until eventually, it is equal to $D_{T'}$. In addition, to every entry e in every set D_t, we maintain a set $\rho_{D_t}(e)$ containing provenance information on e. Afterwards, in the recombination phase, the sets $D_{T'}$ and $\rho_{D_t}(\cdot)$ are used to compute a diversity set with the desired diversity – if such a set exists.

Algorithm 1. Given $Q(X)$, I, $\langle T, \lambda, r\rangle$, k, d, and a diversity measure δ defined via some aggregate function f, the first phase proceeds in three main steps:

- **Initialization:** In this step, for every node $t \in V(T)$, initialize the set D_t as

  $$D_t = \{(\alpha_1, \ldots, \alpha_k, (d_{i,j})_{1\leq i<j\leq k}) : \alpha_i \in t(I), d_{i,j} = \Delta_X(\alpha_i, \alpha_j)\}.$$

  That is, D_t contains one entry for every combination $\alpha_1, \ldots, \alpha_k \in t(I)$, and each value $d_{i,j}$ $(1 \leq i < j \leq k)$ is the Hamming distance of the mappings $\alpha_i|_X$ and $\alpha_j|_X$.
  For every $e \in D_t$, initialize $\rho_{D_t}(e)$ as the empty set.

- **Bottom-Up Traversal:** Set the status of all non-leaf nodes in T to "not-ready" and the status of all leaf nodes to "ready". Then repeat the following action until no "not-ready" node is left: Pick any "not-ready" node t that has at least one "ready" child node t'.

Update D_t to D'_t as

$$\begin{aligned} D'_t = \{(\alpha_1, \dots, \alpha_k, (\bar{d}_{i,j})_{1\leq i<j\leq k}) : {} & (\alpha_1, \dots, \alpha_k, (d_{i,j})_{1\leq i<j\leq k}) \in D_t, \\ & (\alpha'_1, \dots, \alpha'_k, (d'_{i,j})_{1\leq i<j\leq k}) \in D_{t'}, \\ & \alpha_i \cong \alpha'_i \text{ for } 1 \leq i \leq k, \\ & \bar{d}_{i,j} = d_{i,j} + d'_{i,j} - \Delta_X(\alpha_i \cap \alpha'_i, \alpha_j \cap \alpha'_j) \\ & \qquad\qquad\qquad \text{for } 1 \leq i < j \leq k\}. \end{aligned}$$

Expressed in a more procedural style: Take every entry $e \in D_t$ and compare it to every entry $e' \in D_{t'}$. If the corresponding mappings $\alpha_i \in D_t$ and $\alpha'_i \in D_{t'}$ agree on the shared variables, the new set D'_t contains an entry $\bar{e}$ with the mappings α_i from e and the Hamming distances computed from e and e' as described above.
Set $\rho_{D'_t}(\bar{e}) = \rho_{D_t}(e) \cup \{(t', e')\}$. If the same entry $\bar{e}$ is created from different pairs (e, e'), choose an arbitrary one of them for the definition of $\rho_{D'_t}(\bar{e})$.
Finally, change the status of t' from "ready" to "processed". The status of t becomes "ready" if the status of all its child nodes is "processed" and remains "not-ready" otherwise.

- **Finalization:** Once the status of root r is "ready", remove all $(\alpha_1, \dots, \alpha_k, (d_{i,j})_{1\leq i<j\leq k}) \in D_r$ where $f((d_{i,j})_{1\leq i<j\leq k}) < d$. To ensure that all answers in the diversity set are pairwise distinct, also remove all entries where $d_{i,j} = 0$ for some (i, j) with $1 \leq i < j \leq k$. If, after the deletions, D_r is empty, then there exists no diversity set of size k with a diversity of at least d. Otherwise, at least one such diversity set exists.

Clearly, the algorithm is well-defined and terminates. The following theorem states that the algorithm decides Diverse-ACQ and gives an upper bound on the run time.

▶ **Theorem 1.** *The Diverse-ACQ problem is in XP (combined complexity) when parameterized by the size k of the diversity set. More specifically, for an ACQ $Q(X)$, a database I, and integers k and d, Algorithm 1 decides the Diverse-ACQ problem in time $\mathcal{O}\big(|R^I|^{2k} \cdot (|X|+1)^{k(k-1)} \cdot pol(|Q|, k)\big)$ where R^I is the relation from I with the most tuples and $pol(|Q|, k)$ is a polynomial in $|Q|$ and k.*

For any node t in the join tree, D_t denotes the data structure manipulated by Algorithm 1. On the other hand, for the complete subtree T' rooted at t, $D_{T'}$ denotes the goal of our computation, namely the set of tuples $e = (\alpha_1, \dots, \alpha_k, (d_{i,j})_{1\leq i<j\leq k})$ with $ext_{T'}(e) \neq \emptyset$. The key to the correctness of Algorithm 1 is to show that, on termination of the bottom-up traversal, $D_t = D_{T'}$ indeed holds for every node t in the join tree.

We briefly discuss the run time of the algorithm. $|R^I|^{2k} \cdot (|X| + 1)^{k(k-1)}$ represents $|D_t|^2$, where $|D_t|$ is the maximal number of entries e in any D_t during an execution of the algorithm: $|R^I|$ restricts the number of mappings α_i in any $t(I)$, each $d_{i,j}$ can take at most $|X| + 1$ different values (being the Hamming distance of mappings with at most $|X|$ variables), giving $(|X| + 1)^{\frac{k(k-1)}{2}}$ different tuples $(d_{i,j})_{1\leq i<j\leq k}$. $|D_t|^2$ is because the bottom-up traversal can be implemented via a nested loop, dominating the run time of the initialization and finalization steps. The polynomial factor $pol(|Q|, k)$ represents the computation of $\frac{k(k-1)}{2}$ Hamming distances between at most $|var(A)|$ variables (i.e., $k^2 \cdot |var(A)|$ where A is the atom in Q with the most variables), the number of nodes (i.e., $|Q|$), and the computation of the aggregate function f (i.e., some polynomial $pol_f(|X|, k)$ depending on $|X|$ and k).

Theorem 1 shows that the first phase of the algorithm *decides* in XP the existence of a diversity set with a given diversity. Computing a witness diversity set now means computing one element $(\gamma_1, \dots, \gamma_k) \in ext_T(e)$ for some $e \in D_T$ with $f((d_{i,j})_{1\leq i<j\leq k}) \geq d$ and $d_{i,j} \neq 0$

for all i, j. Similarly to the construction of an answer tuple by the Yannakakis algorithm for CQs, we can compute an arbitrary element from $ext_T(e)$ by making use of the information stored in the final sets $\rho_{D_t}(e)$. By construction, for every node $t \in V(T)$ and every entry $e \in D_{T'}$, the final set $\rho_{D_t}(e)$ contains exactly one pair (t', e') for every child node t' of t. Moreover, for the mappings $\alpha_1, \dots, \alpha_k$ from e and $\alpha'_1, \dots, \alpha'_k$ from e', $\alpha_i \cong \alpha'_i$ holds for all $1 \leq i \leq k$, hence $\alpha_i \cup \alpha'_i$ are again mappings. Thus, to compute the desired witness $(\gamma_1, \dots, \gamma_k) \in ext_T(e)$ for the chosen $e \in D_T$, start with $(\alpha_1, \dots, \alpha_k)$ from e, take all (t', e') from $\rho_{D_r}(e)$, extend each α_i with α'_i from e', and repeat this step recursively.

3.1.2 W[1]-Hardness

Having proved XP-membership combined complexity of the Diverse-ACQ problem in Theorem 1, we now show that, for two important aggregators sum and min, a stronger result in the form of FPT-membership is very unlikely to exist. More specifically, we prove W[1]-hardness for query complexity and, hence, also for combined complexity in these cases.

▶ **Theorem 2.** *The problems Diverse$_{\mathsf{sum}}$-ACQ and Diverse$_{\mathsf{min}}$-ACQ, parameterized by the size k of the diversity set, are W[1]-hard. They remain W[1]-hard even if all relation symbols are of arity at most two and $Q(X)$ contains no existential variables.*

Proof sketch. The proof is by simultaneously reducing Independent Set parameterized by the size of the independent set to both Diverse$_{\mathsf{sum}}$-ACQ and Diverse$_{\mathsf{min}}$-ACQ. The only difference between the two reductions will be in how we define the diversity threshold d.

Let (G, s) be an arbitrary instance of Independent Set with $V(G) = \{v_1, \dots, v_n\}$ and $E(G) = \{e_1, \dots, e_m\}$. We define an instance $\langle I, Q, k, d\rangle$ of Diverse$_{\mathsf{sum}}$-ACQ and Diverse$_{\mathsf{min}}$-ACQ, respectively, as follows. The schema consists of a relation symbol R of arity one and m relation symbols $R_1, \dots, R_m$ of arity two. The CQ $Q(X)$ is defined as

$$Q(v, x_1, \dots, x_m) := R(v) \wedge R_1(v, x_1) \wedge \dots \wedge R_m(v, x_m)$$

and the database instance I with $dom(I) = \{0, 1, \dots, n\}$ is

$$R^I = \{(i) : v_i \in V(G)\} \text{ and}$$
$$R_j^I = \{(i, i) : v_i \text{ is not incident to } e_j\} \cup \{(i, 0) : v_i \text{ is incident to } e_j\} \text{ for all } j \in \{1, \dots, m\}.$$

Finally, set $k = s$ and $d = \binom{k}{2} \cdot (m + 1)$ for Diverse$_{\mathsf{sum}}$-ACQ and $d = m + 1$ for Diverse$_{\mathsf{min}}$-ACQ, respectively. Clearly, this reduction is feasible in polynomial time, and the resulting problem instances satisfy all the restrictions stated in the theorem.

The correctness of this reduction depends on two main observations. First, for each $i \in \{1, \dots, n\}$, independently of G, there exists exactly one solution $\gamma_i \in Q(I)$ with $\gamma_i(v) = i$, and these are in fact the only solutions in $Q(I)$. Thus, there is a natural one-to-one association between vertices $v_i \in V(G)$ and solutions $\gamma_i \in Q(I)$. And, second, the desired diversities $d = \binom{k}{2} \cdot (m + 1)$ in case of sum and $d = m + 1$ in case of min, respectively, can only be achieved by k solutions that pairwise differ on all variables. ◀

3.1.3 Speeding up the Basic Algorithm

Algorithm 1 works for any polynomial-time computable diversity measures δ. To compute the diversity at the root node, we needed to distinguish between all the possible values for $d_{i,j}$ $(1 \leq i < j \leq k)$, which heavily increases the size of the sets D_t. However, specific diversity measures may require less information as will now be exemplified for δ_{sum}.

▶ **Theorem 3.** *The* Diverse$_{\mathsf{sum}}$*-ACQ problem is in* FPT *query complexity when parameterized by the size k of the diversity set. More specifically,* Diverse$_{\mathsf{sum}}$*-ACQ for an ACQ $Q(X)$, a database instance I, and integers k and d, can be solved in time $\mathcal{O}(|R^I|^{2k} \cdot 2^{k(k-1)} \cdot pol(|Q|, k))$, where R^I is the relation from I with the most tuples and $pol(|Q|, k)$ is a polynomial in $|Q|$ and k.*

Proof sketch. Note that $pol(|Q|, k)$ is the same as in Theorem 1. For query complexity, the size $|R^I|$ of a relation in I is considered as constant. Hence, the above-stated upper bound on the asymptotic complexity indeed entails FPT-membership. To prove this upper bound, the crucial property is that for a collection of mappings $\gamma_1, \dots, \gamma_k$ over variables Z, the equality $\delta_{\mathsf{sum}}(\gamma_1, \dots, \gamma_k) = \sum_{z \in Z} \delta_{\mathsf{sum}}(\gamma_1|_z, \dots, \gamma_k|_z)$ holds. The reason we had to explicitly distinguish all possible values $(\alpha_1, \dots, \alpha_k, (d_{i,j})_{1 \le i < j \le k})$ in the basic algorithm is that, in general, given two collections $(\gamma_1, \dots, \gamma_k)$ and $(\gamma'_1, \dots, \gamma'_k)$ of mappings that agree on the shared variables, we cannot derive $\delta(\hat{\gamma}_1, \dots, \hat{\gamma}_k)$ for $\hat{\gamma}_i = \gamma_i \cup \gamma'_i$ from $\delta(\gamma_1, \dots, \gamma_k)$ and $\delta(\gamma'_1, \dots, \gamma'_k)$. In contrast, for δ_{sum}, this is possible. Hence, in principle, it suffices to store in $D_{T'}$ for each collection $(\alpha_1, \dots, \alpha_k)$ with $\alpha_i \in t(I)$ (t being the root of T') such that there exists $\gamma_i \in T'(I)$ with $\gamma_i \cong \alpha_i$ (for all $1 \le i \le k$) the value

$$d_{T'}(\alpha_1, \dots, \alpha_k) = \max_{\substack{\gamma_1, \dots, \gamma_k \in T'(I) \\ \text{s.t. } \gamma_i \cong \alpha_i \text{ for all } i}} \delta_{\mathsf{sum}}(\gamma_1|_X, \dots, \gamma_k|_X).$$

I.e., each entry in $D_{T'}$ now is of the form $(\alpha_1, \dots, \alpha_k, v)$ with $v = d_{T'}(\alpha_1, \dots, \alpha_k)$. In the bottom-up traversal step of the algorithm, when updating some D_t to D'_t by merging $D_{t'}$, for every entry $(\alpha_1, \dots, \alpha_k, v) \in D_t$ there exists an entry $(\alpha_1, \dots, \alpha_k, \bar{v}) \in D'_t$ if and only if there exists at least one $(\alpha'_1, \dots, \alpha'_k, v') \in D_{t'}$ such that $\alpha_i \cong \alpha'_i$ for $1 \le i \le k$. Then $\bar{v}$ is

$$\bar{v} = \max_{\substack{(\alpha'_1, \dots, \alpha'_k, v') \in D_{t'} \\ \text{s.t. } \alpha_i \cong \alpha'_i \text{ for all } i}} (v + v' - \delta_{\mathsf{sum}}((\alpha_1 \cap \alpha'_1)|_X, \dots, (\alpha_k \cap \alpha'_k)|_X)).$$

In order to make sure that the answer tuples in the final diversity set are pairwise distinct, the following additional information must be maintained at each $D_{T'}$: from the partial solutions $\alpha_1, \dots, \alpha_k$ it is not possible to determine whether the set of extensions $\gamma_1, \dots, \gamma_k$ contains duplicates or not. Thus, similar to the original values $d_{i,j}$ describing the pairwise diversity of partial solutions, we now include binary values $b_{i,j}$ for $1 \le i < j \le k$ that indicate whether extensions γ_i and γ_j of α_i and α_j to $var(T')$ differ on at least one variable of X ($b_{i,j} = 1$) or not in order to be part of $ext_{T'}(e)$. This increases the maximal size of $D_{T'}$ to $|R^I|^{2k} \cdot 2^{k(k-1)}$. The bottom-up traversal step can be easily adapted to consider in the computation of $\bar{v}$ for an entry in D'_t only those entries from D_t and $D_{t'}$ that are consistent with the values of $b_{i,j}$, giving the stated run time. ◀

Actually, if we drop the condition that the answer tuples in the final diversity set must be pairwise distinct, the query complexity of Diverse$_{\mathsf{sum}}$-ACQ can be further reduced. Clearly, in this case, we can drop the binary values $b_{i,j}$ for $1 \le i < j \le k$ from the entries in $D_{T'}$, which results in a reduction of the asymptotic complexity to $\mathcal{O}(|R^I|^{2k} \cdot pol(|Q|, k))$. At first glance, this does not seem to strengthen the FPT-membership result. However, a further, generally applicable improvement (not restricted to a particular aggregate function and not restricted to query complexity) is possible via the observation that the basic algorithm computes (and manages) redundant information: for an arbitrary node $t \in V(T)$ and set D_t, if D_t contains an entry of the form $(\alpha_1, \dots, \alpha_k, \dots)$, then D_t also contains entries of the form $(\alpha_{\pi(1)}, \dots, \alpha_{\pi(k)}, \dots)$ for all permutations π of $(1, \dots, k)$. But we are ultimately interested in *sets* of answer tuples and do not distinguish between permutations of the members inside a set. Keeping these redundant entries made the algorithm conceptually simpler and had no significant impact on the run times (especially since we assume k to be small compared to the size of the relations in I). However, given the improvements for $\mathsf{Diverse}_{\mathsf{sum}}$-ACQ from Theorem 3 and dropping the binary values $b_{i,j}$ for $1 \leq i < j \leq k$ from the entries in D_t, we can get a significantly better complexity classification:

▶ **Theorem 4.** *The problem $\mathsf{Diverse}_{\mathsf{sum}}$-ACQ is in P (query complexity) when the diversity set may contain duplicates and k is given in unary.*

Proof sketch. To remove redundant rows from the sets D_t, we introduce some order $\preceq$ on partial solutions $\alpha \in t(I)$ for each $t \in V(T)$ (e.g. based on some order on the tuples in $\lambda(t)^I$), and only consider such collections $\alpha_1, \ldots, \alpha_k \in t(I)$ where $\alpha_1 \preceq \cdots \preceq \alpha_k$ together with the value $d_{T'}(\alpha_1, \ldots, \alpha_k)$. Applying some basic combinatorics and assuming the size of I (and thus of $t(I)$) to be constant, we get that the number of entries in any D_t is in $\mathcal{O}(k^{|t(I)|-1})$. Using this upper bound for the size of $|D_t|$ instead of $|R^I|^k$ we get a polynomial run time. ◀

3.2 Data Complexity

We now inspect the data complexity of Diverse-ACQ both from the parameterized and non-parameterized point of view. For the parameterized case, we will improve the XP-membership result from Theorem 1 (for combined complexity) to FPT-membership for arbitrary monotone aggregate functions. Actually, by considering the query as fixed, we now allow arbitrary FO-queries, whose evaluation is well-known to be feasible in polynomial time (data complexity) [33]. Thus, as a preprocessing step, we can evaluate Q and store the result in a table R^I. We may therefore assume w.l.o.g. that the query is of the form $Q(x_1, \ldots, x_m) := R(x_1, \ldots, x_m)$ and the database I consists of a single relation R^I.

To show FPT-membership, we apply a problem reduction that allows us to iteratively reduce the size of the database instance until it is bounded by a function of m and k, i.e., the query and the parameter. Let $X = \{x_1, \ldots, x_m\}$ and define $\binom{X}{s} := \{Z \subseteq X : |Z| = s\}$ for $s \in \{0, \ldots, m\}$. Moreover, for every assignment $\alpha\colon Z \to \mathit{dom}(I)$ with $Z \subseteq X$ let $Q(I)_\alpha := \{\gamma \in Q(I)\colon \gamma \cong \alpha\}$, i.e., the set of answer tuples that coincide with α on Z. The key to our problem reduction is applying the following reduction rule $\mathbf{Red}_t$ for $t \in \{1, \ldots, m\}$ exhaustively in order $\mathbf{Red}_1$ through $\mathbf{Red}_m$:

($\mathbf{Red}_t$) If for some $\alpha\colon Z \to \mathit{dom}(I)$ with $Z \in \binom{X}{m-t}$, the set $Q(I)_\alpha$ has at least $t!^2 \cdot k^t$ elements, then do the following: select (arbitrarily) $t \cdot k$ solutions $\Gamma \subseteq Q(I)_\alpha$ that pairwisely differ on all variables in $X \setminus Z$. Then remove the tuples corresponding to assignments $Q(I)_\alpha \setminus \Gamma$ from R^I.

The intuition of the reduction rule is best seen by looking at $\mathbf{Red}_1$. Our ultimate goal is to achieve maximum diversity by selecting k answer tuples. Now suppose that we fix $m-1$ positions in the answer relation R^I. In this case, if there are at least k different values in the m-th component, the maximum is actually achieved by selecting k such tuples. But then there is no need to retain further tuples with the same values in the $m-1$ fixed positions. This can be generalized to fixing fewer positions but the intuition stays the same. When fixing $m-t$ positions, there is also no need to retain all different value combinations in the remaining t positions. Concretely, if there exists at least $t!^2 \cdot k^t$ different value combinations, there also exist $t \cdot k$ tuples with pairwise maximum Hamming distance on the remaining positions and it is sufficient to only keep those.

With the reduction rule $\mathbf{Red}_t$ at our disposal, we can design an FPT-algorithm (data complexity) for Diverse$_{\mathsf{mon}}$-ACQ and, more generally, for the Diverse$_{\mathsf{mon}}$-FO problem:

▶ **Theorem 5.** *The problem Diverse$_{\mathsf{mon}}$-FO is in FPT data complexity when parameterized by the size k of the diversity set. More specifically, an instance $\langle I, Q, k, d\rangle$ of Diverse$_{\mathsf{mon}}$-FO with m-ary FO-query Q can be reduced in polynomial time (data complexity) to an equivalent instance $\langle I', Q', k, d\rangle$ of Diverse$_{\mathsf{mon}}$-FO of size $\mathcal{O}(m!^2 \cdot k^m)$.*

Proof sketch. As mentioned above, we can transform in polynomial time any (fixed) FO-query into the form $Q(x_1, \ldots, x_m) = R(x_1, \ldots, x_m)$ over a database I with a single relation R^I. The reduction to an equivalent problem instance of size $\mathcal{O}(m!^2 \cdot k^m)$ is then achieved by applying $\mathbf{Red}_1$ through $\mathbf{Red}_m$ to I in this order exhaustively. The crucial property of the reduction rule $\mathbf{Red}_t$ with $t \in \{1, \ldots, m\}$ is as follows:

Claim A. *Let $t \in \{1, \ldots, m\}$ and suppose that all sets $Q(I)_{\alpha'}$ with $\alpha' \colon Z' \to dom(I)$ and $Z' \in \binom{X}{m-(t-1)}$ have cardinality at most $(t-1)!^2 \cdot k^{t-1}$. Then the reduction rule $\mathbf{Red}_t$ is well-defined and safe. That is:*

- "well-defined". *If for some $\alpha : Z \to dom(I)$ with $Z \in \binom{X}{m-t}$, the set $Q(I)_\alpha$ has at least $t!^2 \cdot k^t$ elements, then there exist at least $t \cdot k$ solutions $\Gamma \subseteq Q(I)_\alpha$ that pairwisely differ on all variables in $X \setminus Z$.*
- "safe". *Let I_{old} denote the database instance before an application of $\mathbf{Red}_t$ and let I_{new} denote its state after applying $\mathbf{Red}_t$. Let $\gamma_1, \ldots, \gamma_k$ be pairwise distinct solutions in $Q(I_{old})$. Then there exist pairwise distinct solutions $\gamma'_1, \ldots, \gamma'_k$ in $Q(I_{new})$ with $\delta(\gamma'_1, \ldots, \gamma'_k) \geq \delta(\gamma_1, \ldots, \gamma_k)$, i.e., the diversity achievable before deleting tuples from the database can still be achieved after the deletion.*

Note that a naive greedy algorithm always finds a witnessing Γ and the existence of this greedy algorithm implies the well-definedness. The safety follows from the fact that each γ_i that is removed, i.e., $\gamma_i \in Q(I)_\alpha \setminus \Gamma$, can be replaced by a $\gamma'_i \in \Gamma$ that is kept. Concretely, we can pick $\gamma'_i \in \Gamma$ such that $\delta(\ldots, \gamma'_i, \ldots) \geq \delta(\ldots, \gamma_i, \ldots)$. ◀

We now study the data complexity of the Diverse-ACQ problem in the non-parameterized case, i.e., the size k of the diversity set is part of the input and no longer considered as the parameter. It will turn out that this problem is NP-hard for the two important aggregator functions sum and min. Our NP-hardness proof will be by reduction from the Independent Set problem, where we restrict the instances to graphs of degree at most 3. It was shown in [2] that this restricted problem remains NP-complete.

▶ **Theorem 6.** *The problems Diverse$_{\mathsf{sum}}$-ACQ and Diverse$_{\mathsf{min}}$-ACQ are NP-hard data complexity. They are NP-complete if the size k of the diversity set is given in unary.*

Proof sketch. The NP-membership is immediate: compute $Q(I)$ (which is feasible in polynomial time when considering the query as fixed), then guess a subset $S \subseteq Q(I)$ of size k and check in polynomial time that S has the desired diversity.

For the NP-hardness, we define the query Q independently of the instance of the Independent Set problem as $Q(x_1, x_2, x_3, x_4, x_5) := R(x_1, x_2, x_3, x_4, x_5)$, i.e., the only relation symbol R has arity 5. Now let (G, s) be an instance of Independent Set where each vertex of G has degree at most 3.

Let $V(G) = \{v_1, \ldots, v_n\}$ and $E(G) = \{e_1, \ldots, e_m\}$. Then the database I consists of a single relation R^I with n tuples (= number of vertices in G) over the domain $dom(I) = \{\mathbf{free}_1, \ldots, \mathbf{free}_n, \mathbf{taken}_1, \ldots, \mathbf{taken}_m\}$. The i-th tuple in R^I will be denoted $(e_{i,1}, \ldots, e_{i,5})$. For each $v_i \in V(G)$, the values $e_{i,1}, \ldots, e_{i,5} \in dom(I)$ are defined by an iterative process:

1. The iterative process starts by initializing all $e_{i,1}, \dots, e_{i,5}$ to $\mathbf{free}_i$ for each $v_i \in V(G)$.
2. We then iterate through all edges $e_j \in E(G)$ and do the following: Let v_i and $v_{i'}$ be the two incident vertices to e_j and let $t \in \{1, \dots, 5\}$ be an index such that $e_{i,t}$ and $e_{i',t}$ both still have the values $\mathbf{free}_i$ and $\mathbf{free}_{i'}$, respectively. Then set both $e_{i,t}$ and $e_{i',t}$ to $\mathbf{taken}_j$.

In the second step above when processing an edge e_j, such an index t must always exist. This is due to the fact that, at the moment of considering e_j, the vertex v_i has been considered at most twice (the degree of v_i is at most 3) and thus, for at least three different values of $t \in \{1, \dots, 5\}$, the value $e_{i,t}$ is still set to $\mathbf{free}_i$. Analogous considerations apply to vertex $v_{i'}$ and thus, for at least 3 values of $t \in \{1, \dots, 5\}$, we have $e_{i',t} = \mathbf{free}_{i'}$. Hence, by the pigeonhole principle, there exists $t \in \{1, \dots, 5\}$ with $e_{i,t} = \mathbf{free}_i$ and $e_{i',t} = \mathbf{free}_{i'}$.

After the iterative process, the database I is defined by $R^I = \{(e_{i,1}, e_{i,2}, e_{i,3}, e_{i,4}, e_{i,5}) : i = 1, \dots, n\}$. Moreover, the size of the desired diversity set is set to $k = s$ and the target diversity is set to $d_{\mathsf{sum}} = 5 \cdot \frac{k \cdot (k-1)}{2}$ and $d_{\mathsf{min}} = 5$ in the case of the $\mathsf{Diverse}_{\mathsf{sum}}$-$\mathsf{ACQ}$ and $\mathsf{Diverse}_{\mathsf{min}}$-$\mathsf{ACQ}$ problems, respectively. The resulting problem instances of $\mathsf{Diverse}_{\mathsf{sum}}$-$\mathsf{ACQ}$ and $\mathsf{Diverse}_{\mathsf{min}}$-$\mathsf{ACQ}$ are thus of the form $\langle I, Q, k, d_{\mathsf{sum}} \rangle$ and $\langle I, Q, k, d_{\mathsf{min}} \rangle$, respectively.

The reduction is clearly feasible in polynomial time. Its correctness hinges on the observation that the desired diversities $d_{\mathsf{sum}} = 5 \cdot \frac{k \cdot (k-1)}{2}$ and $d_{\mathsf{min}} = 5$ can only be reached by k answer tuples that pairwisely differ in all 5 positions. ◀

4 Diversity of Unions of Conjunctive Queries

We now turn our attention to UCQs. Of course, all hardness results proved for CQs and ACQs in Section 3 carry over to UCQs and UACQs, respectively. Moreover, the FPT-membership result from Theorem 5 for general FO-formulas of course also includes UCQs. It remains to study the query complexity and combined complexity of UACQs. It turns out that the union makes the problem significantly harder than for ACQs. We show next that Diverse-UACQ is NP-hard (for the aggregators sum and min) even in a very restricted setting, namely a union of two ACQs and with desired size $k = 2$ of the diversity set.

The proof will be by reduction from a variant of the List Coloring problem, which we introduce next: A *list assignment* C assigns each vertex v of a graph G a list of colors $C(v) \subseteq \{1, \dots, l\}, l \in \mathbb{N}$. Then a *coloring* is a function $c : V(G) \to \{1, \dots, l\}$ and it is called C*−admissible* if each vertex $v \in V(G)$ is colored in a color of its list, i.e., $c(v) \in C(v)$, and adjacent vertices $u, v \in E(G)$ are colored with different colors, i.e., $c(u) \neq c(v)$. Formally, the problem is defined as follows:

List Coloring

Input: A graph G, an integer $l \in \mathbb{N}$, and a list assignment $C : V(G) \to 2^{\{1,\dots,l\}}$.

Question: Does there exist a C-admissible coloring $c : V(G) \to \{1, \dots, l\}$?

Clearly, List Coloring is a generalization of 3-Colorability and, hence, NP-complete. It was shown in [12], that the List Coloring problem remains NP-hard even when assuming that each vertex of G has degree 3, G is bipartite, and $l = 3$. This restriction will be used in the proof of the following theorem.

▶ **Theorem 7.** *The problems $\mathsf{Diverse}_{\mathsf{sum}}$-UACQ and $\mathsf{Diverse}_{\mathsf{min}}$-UACQ are NP-hard query complexity (and hence, also combined complexity). They remain NP-hard even if the desired size of the diversity set is bounded by 2 and the UACQs are restricted to containing at most two conjuncts and no existential variables. The problems are NP-complete if the size k of the diversity set is given in unary.*

Proof sketch. The NP-membership in case of k given in unary is immediate: guess k assignments to the free variables of query Q, check in polynomial time that they are solutions, and verify in polynomial time that their diversity is above the desired threshold.

For the NP-hardness, first observe that δ_{sum} and δ_{min} coincide if we only allow two solutions. Hence, we may use a single diversity function δ to prove the NP-hardness for both Diverse$_{\mathsf{sum}}$-UACQ and Diverse$_{\mathsf{min}}$-UACQ.

For our problem reduction, we consider a fixed database I over a fixed schema, which consists of 9 relation symbols

$$R_{\{1\}}, R_{\{2\}}, R_{\{3\}}, R_{\{1,2\}}, R_{\{1,3\}}, R_{\{2,3\}}, R_{\{1,2,3\}}, S, S'.$$

The relations of the database are defined as follows:

$$\begin{aligned}
R^I_{\{1\}} &= \{(1,1,1)\} & R^I_{\{1,2\}} &= \{(1,1,1),(2,2,2)\}\\
R^I_{\{2\}} &= \{(2,2,2)\} & R^I_{\{1,3\}} &= \{(1,1,1),(3,3,3)\}\\
R^I_{\{3\}} &= \{(3,3,3)\} & R^I_{\{2,3\}} &= \{(2,2,2),(3,3,3)\}\\
R^I_{\{1,2,3\}} &= \{(1,1,1),(2,2,2),(3,3,3)\} & S^I &= \{(0)\} \qquad S'^I = \{(1)\}
\end{aligned}$$

Now let $\langle G, l, C\rangle$ be an arbitrary instance of List Coloring, where each vertex of G has degree 3, G is bipartite, and $l = 3$. That is, G is of the form $G = (V \cup V', E)$ for vertex sets V, V' and edge set E with $V = \{v_1, \ldots, v_n\}$, $V' = \{v'_1, \ldots, v'_n\}$, and $E = \{e_1, \ldots, e_{3n}\}$. Note that $|V| = |V'|$ and $|E| = 3 \cdot |V|$ as each vertex in G has degree three and G is bipartite.

From this we construct a UACQ Q as follows: we use the $3n + 1$ variables $x_1, \ldots, x_{3n}, y$ in our query. For each $i \in \{1, \ldots, n\}$, we write $e_{j_{i,1}}, e_{j_{i,2}}, e_{j_{i,3}}$ to denote the three edges incident to the vertex v_i. Analogously, we write $e_{j'_{i,1}}, e_{j'_{i,2}}, e_{j'_{i,3}}$ to denote the three edges incident to the vertex v'_i.

The UACQ Q is then defined as $Q(x_1, \ldots, x_{3n}, y) := \varphi \vee \psi$ with

$$\varphi = \bigwedge_{i=1}^{n} R_{C(v_i)}(x_{j_{i,1}}, x_{j_{i,2}}, x_{j_{i,3}}) \wedge S(y),$$

$$\psi = \bigwedge_{i=1}^{n} R_{C(v'_i)}(x_{j'_{i,1}}, x_{j'_{i,2}}, x_{j'_{i,3}}) \wedge S'(y).$$

Moreover, we set the target diversity to $d = 3n + 1$ and we are looking for $k = 2$ solutions to reach this diversity. Observe that each variable appears exactly once in φ and once in ψ, which makes both formulas trivially acyclic. Furthermore, Q contains no existential variables.

The intuition of the big conjunction in φ (resp. ψ) is to "encode" for each vertex v_i (resp. v'_i) the 3 edges incident to this vertex in the form of the 3 x-variables with the corresponding indices. The relation symbol chosen for each vertex v_i or v'_i depends on the color list for this vertex. For instance, if $C(v_1) = \{2, 3\}$ and if v_1 is incident to the edges e_4, e_6, e_7, then the first conjunct in the definition of φ is of the form $R_{\{2,3\}}(x_4, x_6, x_7)$. Note that the order of the variables in this atom is irrelevant since the R-relations contain only tuples with identical values in all 3 positions. Intuitively, this ensures that a vertex (in this case v_1) gets the same color (in this case color 2 or 3) in all its incident edges (in this case e_4, e_6, e_7). ◀

5 Diversity of Conjunctive Queries with Negation

Lastly, we consider CQs$^\neg$. As was recalled in Section 1, the restriction to acyclicity is not sufficient to ensure tractable answering of CQs$^\neg$ [32]. In the following, we thus restrict ourselves to queries of bounded treewidth when analyzing the Diverse-CQ$^\neg$ problem.

The data complexity case has already been settled for arbitrary FO-formulas in Theorem 5. Hence, of course, also Diverse-CQ$^\neg$ is in FPT data complexity and NP-hard in the non-parameterized case. Moreover, we observe that the query used in the proof of Theorem 2 has a treewidth of one. Hence, it is clear that also Diverse-CQ$^\neg$ is W[1]-hard combined complexity for queries with bounded treewidth. It remains to study the combined complexity, for which we describe an XP-algorithm next.

Our algorithm is based on so-called *nice* tree decompositions – a normal form introduced in [23]. A nice tree decomposition only allows leaf nodes plus three types of inner nodes: introduce nodes, forget nodes, and join nodes. An *introduce node* t has a single child t' with $\chi(t) = \chi(t') \cup \{z\}$ for a single variable z. Similarly, a *forget node* t has a single child t' with $\chi(t') = \chi(t) \cup \{z\}$ for a single variable z. Finally, a *join node* t has two child nodes t_1, t_2 with $\chi(t) = \chi(t_1) = \chi(t_2)$. It was shown in [23] that every tree decomposition can be transformed in linear time into a nice tree decomposition without increasing the width.

The intuition of the present algorithm is very similar to the intuition of Algorithm 1 presented in Section 3.1.1. That is, both algorithms maintain information on tuples of k partial solutions in a set D_t. Concretely, these tuples are again of the form $(\alpha_1, \dots, \alpha_k, (d_{i,j})_{1 \le i<j \le k})$. This time, however, partial solutions α_i are not assignments that satisfy concrete atoms but arbitrary assignments defined on $\chi(t)$. Nevertheless, a tuple gets added to D_t if and only if it is possible to extend the partial solutions to mappings $\gamma_1, \dots, \gamma_k$ that (a) satisfy the query associated to the subtree rooted in t and (b) for $1 \le i < j \le k$ the distance between γ_i and γ_j is exactly $d_{i,j}$.

Formally, for a CQ$^\neg$ $Q(X) := \exists Y \bigwedge_{i=1}^n L_i(X,Y)$ and nice tree decomposition $\langle T, \chi, r\rangle$ of Q we define for $t \in V(T)$ the subquery

$$Q_t = \bigwedge_{\substack{i=1,\dots,n \\ var(L_i) \subseteq \chi(t)}} L_i,$$

i.e., Q_t contains those literals of Q whose variables are covered by $\chi(t)$.

Algorithm 2. Given $Q(X)$, I, k, d, a nice tree decomposition $\langle T, \chi, r\rangle$ of minimum width, and a diversity measure δ defined via some aggregate function f, the algorithm proceeds in two main steps: First, the sets D_t are computed bottom-up for each $t \in V(T)$, and then, it is determined from D_r whether the diversity threshold d can be met. For the bottom-up step, the type of t determines how D_t is computed:

- **Leaf Node:** For a leaf node $t \in V(T)$ we create D_t as

$$\begin{aligned} D_t = \{(\alpha_1, \dots, \alpha_k, (d_{i,j})_{1\le i<j\le k}) : {} & \alpha_1, \dots, \alpha_k \colon \chi(t) \to dom(I), \\ & \alpha_1, \dots, \alpha_k \text{ satisfy } Q_t, \\ & d_{i,j} = \Delta_X(\alpha_i, \alpha_j), 1 \le i < j \le k\}. \end{aligned}$$

  Hence, we exhaustively go through all possible variable assignments $\alpha_1, \dots, \alpha_k \colon \chi(t) \to dom(I)$, keep those which satisfy the query Q_t, and record their pairwise diversities.

- **Introduce Node:** For an introduce node $t \in V(T)$ with child $c \in V(T)$ which introduces the variable $z \in \chi(t) \setminus \chi(c)$, we create D_t as

$$\begin{aligned} D_t = \{(\alpha_1 \cup \beta_1, \dots, \alpha_k \cup \beta_k, (d'_{i,j})_{1\le i<j\le k}) : {} & (\alpha_1, \dots, \alpha_k, (d_{i,j})_{1\le i<j\le k}) \in D_c, \\ & \beta_1, \dots, \beta_k \colon \{z\} \to dom(I), \\ & \alpha_1 \cup \beta_1, \dots, \alpha_k \cup \beta_k \text{ satisfy } Q_t, \\ & d'_{i,j} = d_{i,j} + \Delta_X(\beta_i, \beta_j), 1 \le i < j \le k\}. \end{aligned}$$

Thus, we extend the domain of the local variable assignments in D_c by z. We do this by exhaustively going through all $e \in D_c$ in combination with all $\beta_1, \dots, \beta_k \colon \{z\} \to dom(I)$, check if the extensions $\alpha_1 \cup \beta_1, \dots, \alpha_k \cup \beta_k$ satisfy all literals for which all variables are covered, and, if this is the case, add the diversity achieved on the z-variable.

- **Forget Node:** For a forget node $t \in V(T)$ with child $c \in V(T)$ we create D_t as

$$D_t = \{(\alpha_1|_{\chi(t)}, \dots, \alpha_k|_{\chi(t)}, (d_{i,j})_{1 \le i < j \le k}) : (\alpha_1, \dots, \alpha_k, (d_{i,j})_{1 \le i < j \le k}) \in D_c\}.$$

- **Join Node:** For a join node $t \in V(T)$ with children $c_1, c_2 \in V(T)$ we create D_t as

$$\begin{aligned} D_t = \{(\alpha_1, \dots, \alpha_k, (d_{i,j})_{1 \le i < j \le k}) : {} & (\alpha_1, \dots, \alpha_k, (d'_{i,j})_{1 \le i < j \le k}) \in D_{c_1}, \\ & (\alpha_1, \dots, \alpha_k, (d''_{i,j})_{1 \le i < j \le k}) \in D_{c_2}, \\ & d_{i,j} = d'_{i,j} + d''_{i,j} - \Delta_X(\alpha_i, \alpha_j), 1 \le i < j \le k\}. \end{aligned}$$

  In this step, we match rows of D_{c_1} with rows of D_{c_2} that agree on the local variable assignments and simply combine the diversities achieved in the two child nodes while subtracting the diversity counted twice.

For the second step, the algorithm goes through all $(\alpha_1, \dots, \alpha_k, (d_{i,j})_{1 \le i < j \le k}) \in D_r$ and removes those tuples where $d_{i,j} = 0$ for at least one $1 \le i < j \le k$ or $f((d_{i,j})_{1 \le i < j \le k}) < d$. Then, the algorithm returns "yes" if the resulting set is non-empty and otherwise "no".

Clearly, the algorithm is well-defined and terminates. The next theorem states that the algorithm decides Diverse-CQ$^\neg$, and discusses its run time.

▶ **Theorem 8.** *For a class of CQs$^\neg$ of bounded treewidth, the problem Diverse-CQ$^\neg$ is in XP when parameterized by the size k of the diversity set. More specifically, let $Q(X)$ be from a class of CQs$^\neg$ which have treewidth $\le \omega$. Then, for a database instance I and integers k, d, Algorithm 2 solves Diverse-CQ$^\neg$ in time $\mathcal{O}(dom(I)^{2 \cdot k \cdot (\omega+1)} \cdot (|X|+1)^{k(k-1)} \cdot pol(|Q|, k))$, where $pol(|Q|, k)$ is a polynomial in $|Q|$ and k.*

Proof sketch. We briefly sketch how to arrive at the given run time. Note that the core ideas are similar to the ones of Algorithm 1. Firstly, for the bottom-up traversal, $dom(I)^{2 \cdot k \cdot (\omega+1)} \cdot (|X|+1)^{k(k-1)}$ is a bound for $|D_t|^2$. Thus, for each node t, we can simply use (nested) loops for the exhaustive searches and, as the checks only require polynomial time, compute each D_t in the required time bound. Then, evaluating f also only requires polynomial time and has to be applied at most $|D_r|$ many times. Lastly, computing an appropriate tree decomposition in the required time bound is possible due to [6] and [23]. ◀

We conclude this section by again stressing the analogy with Algorithm 1 for ACQs: First, we have omitted from our description of Algorithm 2 how to compute a concrete witnessing diversity set in the case of a yes-answer. This can be done exactly as in Algorithm 1 by maintaining the same kind of provenance information. And second, it is possible to speed up the present algorithm by applying the same kind of considerations as in Section 3.1.3. It is thus possible to reduce the query complexity to FPT for the diversity measure δ_{sum} and even further to P if we allow duplicates in the diversity set.

6 Conclusion and Future Work

In this work, we have had a fresh look at the Diversity problem of query answering. For CQs and extensions thereof, we have proved a collection of complexity results, both for the parameterized and the non-parameterized case. To get a chance of reaching tractability or at least fixed-parameter tractability (when considering the size k of the diversity set as the parameter), we have restricted ourselves to acyclic CQs and CQs with bounded treewidth, respectively. It should be noted that the restriction to acyclic CQs is less restrictive than it may seem at first glance. Indeed, our upper bounds (in particular, the XP- and FPT-membership results in Section 3) are easily generalized to CQs of bounded hypertree-width [18]. Moreover, recent empirical studies of millions of queries from query logs [7] and thousands of queries from benchmarks [17] have shown that CQs typically have hypertree-width at most 3.

For the chosen settings, our complexity results are fairly complete. The most obvious gaps left for future work are concerned with the query complexity of ACQs and CQs with negation of bounded treewidth. For the parameterized case, we have XP-membership but no fixed-parameter intractability result in the form of W[1]-hardness. And for the non-parameterized case, it is open if the problems are also NP-hard as we have shown for the data complexity. Moreover, for future work, different settings could be studied. We mention several modifications below.

First, different parameterizations might be of interest. We have only considered the parameterization by the size k of the diversity set. Adding the hypertree-width (for Diverse-ACQ) and the treewidth (for Diverse-CQ$^{\neg}$) to the parameter would leave our XP-membership results unchanged. On the other hand, different parameterizations such as the threshold d on the diversity are left for future work.

Another direction for future work is motivated by a closer look at our FPT- and XP-membership results: even though such parameterized complexity results are generally considered as favorable (in particular, FPT), the run times are exponential in the parameter k. As we allow larger values of k, these run times may not be acceptable anymore. It would therefore be interesting to study the diversity problem also from an approximation point of view – in particular, contenting oneself with an approximation of the desired diversity.

A further modification of our settings is related to the choice of a different distance measure between two answer tuples and different aggregators. As far as the distance measure is concerned, we have so far considered data values as untyped and have therefore studied only the Hamming distance between tuples. For numerical values, one might of course take the difference between values into account. More generally, one could consider a metric on the domain, which then induces a metric on tuples that can be used as a distance measure. As far as the aggregator is concerned, we note that most of our upper bounds apply to arbitrary (polynomial-time computable) aggregate functions. As concrete aggregators, we have studied sum and min. This seems quite a natural choice since, for a fixed number k of answer tuples, avg behaves the same as sum and count makes no sense. Moreover, max is unintuitive if we want to achieve diversity *above* some threshold. However, a problem strongly related to Diversity is Similarity [15], where one is interested in finding solutions close to each other. In this case, max (and again sum) seems to be the natural aggregator. We leave the study of Similarity for future work.

References

1 Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995. URL: `http://webdam.inria.fr/Alice/`.

2 Paola Alimonti and Viggo Kann. Hardness of approximating problems on cubic graphs. In Gian Carlo Bongiovanni, Daniel P. Bovet, and Giuseppe Di Battista, editors, *Algorithms and Complexity, Third Italian Conference, CIAC '97, Rome, Italy, March 12-14, 1997, Proceedings*, volume 1203 of *Lecture Notes in Computer Science*, pages 288–298. Springer, 1997. doi:10.1007/3-540-62592-5_80.

3 Antoine Amarilli, Louis Jachiet, Martin Muñoz, and Cristian Riveros. Efficient enumeration for annotated grammars. In Leonid Libkin and Pablo Barceló, editors, *PODS '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, pages 291–300. ACM, 2022. `doi:10.1145/3517804.3526232`.

4 Marcelo Arenas, Luis Alberto Croquevielle, Rajesh Jayaram, and Cristian Riveros. Efficient logspace classes for enumeration, counting, and uniform generation. In *Proc. PODS 2019*, pages 59–73. ACM, 2019. `doi:10.1145/3294052.3319704`.

5 Julien Baste, Michael R. Fellows, Lars Jaffke, Tomás Masarík, Mateus de Oliveira Oliveira, Geevarghese Philip, and Frances A. Rosamond. Diversity of solutions: An exploration through the lens of fixed-parameter tractability theory. *Artif. Intell.*, 303:103644, 2022. `doi:10.1016/j.artint.2021.103644`.

6 Hans L. Bodlaender. A linear-time algorithm for finding tree-decompositions of small treewidth. *SIAM J. Comput.*, 25(6):1305–1317, 1996. `doi:10.1137/S0097539793251219`.

7 Angela Bonifati, Wim Martens, and Thomas Timm. An analytical study of large SPARQL query logs. *VLDB J.*, 29(2-3):655–679, 2020. `doi:10.1007/s00778-019-00558-9`.

8 Endre Boros, Benny Kimelfeld, Reinhard Pichler, and Nicole Schweikardt. Enumeration in data management (dagstuhl seminar 19211). *Dagstuhl Reports*, 9(5):89–109, 2019. `doi:10.4230/DagRep.9.5.89`.

9 Nofar Carmeli, Shai Zeevi, Christoph Berkholz, Benny Kimelfeld, and Nicole Schweikardt. Answering (unions of) conjunctive queries using random access and random-order enumeration. In *Proc. PODS 2020*, pages 393–409. ACM, 2020. `doi:10.1145/3375395.3387662`.

10 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`.

11 Surajit Chaudhuri and Rajeev Motwani. On sampling and relational operators. *IEEE Data Eng. Bull.*, 22(4):41–46, 1999. URL: `http://sites.computer.org/debull/99dec/surajit.ps`.

12 Miroslav Chlebík and Janka Chlebíková. Hard coloring problems in low degree planar bipartite graphs. *Discret. Appl. Math.*, 154(14):1960–1965, 2006. `doi:10.1016/j.dam.2006.03.014`.

13 Ting Deng and Wenfei Fan. On the complexity of query result diversification. *ACM Trans. Database Syst.*, 39(2):15:1–15:46, 2014. `doi:10.1145/2602136`.

14 Rodney G. Downey and Michael R. Fellows. *Parameterized Complexity*. Monographs in Computer Science. Springer, 1999. `doi:10.1007/978-1-4612-0515-9`.

15 Thomas Eiter, Esra Erdem, Halit Erdogan, and Michael Fink. Finding similar/diverse solutions in answer set programming. *Theory Pract. Log. Program.*, 13(3):303–359, 2013. `doi:10.1017/S1471068411000548`.

16 Henning Fernau, Petr A. Golovach, and Marie-France Sagot. Algorithmic enumeration: Output-sensitive, input-sensitive, parameterized, approximative (dagstuhl seminar 18421). *Dagstuhl Reports*, 8(10):63–86, 2018. `doi:10.4230/DagRep.8.10.63`.

17 Wolfgang Fischl, Georg Gottlob, Davide Mario Longo, and Reinhard Pichler. Hyperbench: A benchmark and tool for hypergraphs and empirical findings. *ACM J. Exp. Algorithmics*, 26:1.6:1–1.6:40, 2021. `doi:10.1145/3440015`.

18 Georg Gottlob, Nicola Leone, and Francesco Scarcello. Hypertree decompositions and tractable queries. *J. Comput. Syst. Sci.*, 64(3):579–627, 2002. `doi:10.1006/jcss.2001.1809`.

19 Marc H. Graham. On The Universal Relation. Technical report, University of Toronto, 1979.

20 Emmanuel Hebrard, Brahim Hnich, Barry O'Sullivan, and Toby Walsh. Finding diverse and similar solutions in constraint programming. In Manuela M. Veloso and Subbarao Kambhampati, editors, *Proceedings, The Twentieth National Conference on Artificial Intelligence and the Seventeenth Innovative Applications of Artificial Intelligence Conference, July 9-13, 2005, Pittsburgh, Pennsylvania, USA*, pages 372–377. AAAI Press / The MIT Press, 2005. URL: `http://www.aaai.org/Library/AAAI/2005/aaai05-059.php`.

21 Linnea Ingmar, Maria Garcia de la Banda, Peter J. Stuckey, and Guido Tack. Modelling diversity of solutions. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*, pages 1528–1535. AAAI Press, 2020. URL: https://aaai.org/ojs/index.php/AAAI/article/view/5512.

22 David S. Johnson, Christos H. Papadimitriou, and Mihalis Yannakakis. On generating all maximal independent sets. *Inf. Process. Lett.*, 27(3):119–123, 1988. doi:10.1016/0020-0190(88)90065-8.

23 Ton Kloks. *Treewidth, Computations and Approximations*, volume 842 of *Lecture Notes in Computer Science*. Springer, 1994. doi:10.1007/BFb0045375.

24 Yasuaki Kobayashi, Kazuhiro Kurita, and Kunihiro Wasa. Linear-delay enumeration for minimal steiner problems. In Leonid Libkin and Pablo Barceló, editors, *PODS '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, pages 301–313. ACM, 2022. doi:10.1145/3517804.3524148.

25 Viktor Leis, Bernhard Radke, Andrey Gubichev, Alfons Kemper, and Thomas Neumann. Cardinality estimation done right: Index-based join sampling. In *Proc. CIDR 2017*. www.cidrdb.org, 2017. URL: http://cidrdb.org/cidr2017/papers/p9-leis-cidr17.pdf.

26 Feifei Li, Bin Wu, Ke Yi, and Zhuoyue Zhao. Wander join and XDB: online aggregation via random walks. *ACM Trans. Database Syst.*, 44(1):2:1–2:41, 2019. doi:10.1145/3284551.

27 Carsten Lutz and Marcin Przybylko. Efficiently enumerating answers to ontology-mediated queries. In Leonid Libkin and Pablo Barceló, editors, *PODS '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, pages 277–289. ACM, 2022. doi:10.1145/3517804.3524166.

28 Timo Camillo Merkl, Reinhard Pichler, and Sebastian Skritek. Diversity of answers to conjunctive queries. *CoRR*, arXiv:2301.08848, 2023. doi:10.48550/arXiv.2301.08848.

29 Alexander Nadel. Generating diverse solutions in SAT. In Karem A. Sakallah and Laurent Simon, editors, *Theory and Applications of Satisfiability Testing - SAT 2011 - 14th International Conference, SAT 2011, Ann Arbor, MI, USA, June 19-22, 2011. Proceedings*, volume 6695 of *Lecture Notes in Computer Science*, pages 287–301. Springer, 2011. doi:10.1007/978-3-642-21581-0_23.

30 Thierry Petit and Andrew C. Trapp. Finding diverse solutions of high quality to constraint optimization problems. In Qiang Yang and Michael J. Wooldridge, editors, *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*, pages 260–267. AAAI Press, 2015. URL: http://ijcai.org/Abstract/15/043.

31 Neil Robertson and Paul D. Seymour. Graph minors. III. planar tree-width. *J. Comb. Theory, Ser. B*, 36(1):49–64, 1984. doi:10.1016/0095-8956(84)90013-3.

32 Marko Samer and Stefan Szeider. Algorithms for propositional model counting. *J. Discrete Algorithms*, 8(1):50–64, 2010. doi:10.1016/j.jda.2009.06.002.

33 Moshe Y. Vardi. The complexity of relational query languages (extended abstract). In Harry R. Lewis, Barbara B. Simons, Walter A. Burkhard, and Lawrence H. Landweber, editors, *Proceedings of the 14th Annual ACM Symposium on Theory of Computing, May 5-7, 1982, San Francisco, California, USA*, pages 137–146. ACM, 1982. doi:10.1145/800070.802186.

34 Mihalis Yannakakis. Algorithms for acyclic database schemes. In *Very Large Data Bases, 7th International Conference, September 9-11, 1981, Cannes, France, Proceedings*, pages 82–94. IEEE Computer Society, 1981.

35 C. T. Yu and M. Z. Özsoyoglu. An algorithm for tree-query membership of a distributed query. In *The IEEE Computer Society's Third International Computer Software and Applications Conference, COMPSAC 1979*, pages 306–312, 1979.

36 Zhuoyue Zhao, Robert Christensen, Feifei Li, Xiao Hu, and Ke Yi. Random sampling over joins revisited. In *Proc. SIGMOD 2018*, pages 1525–1539. ACM, 2018. `doi:10.1145/3183713.3183739`.

37 Kaiping Zheng, Hongzhi Wang, Zhixin Qi, Jianzhong Li, and Hong Gao. A survey of query result diversification. *Knowl. Inf. Syst.*, 51(1):1–36, 2017. `doi:10.1007/s10115-016-0990-4`.

The Complexity of the Shapley Value for Regular Path Queries

Majd Khalil
Technion, Haifa, Israel

Benny Kimelfeld
Technion, Haifa, Israel

Abstract

A path query extracts vertex tuples from a labeled graph, based on the words that are formed by the paths connecting the vertices. We study the computational complexity of measuring the contribution of edges and vertices to an answer to a path query, focusing on the class of conjunctive regular path queries. To measure this contribution, we adopt the traditional Shapley value from cooperative game theory. This value has been recently proposed and studied in the context of relational database queries and has uses in a plethora of other domains.

We first study the contribution of edges and show that the exact Shapley value is almost always hard to compute. Specifically, it is #P-hard to calculate the contribution of an edge whenever at least one (non-redundant) conjunct allows for a word of length three or more. In the case of regular path queries (i.e., no conjunction), the problem is tractable if the query has only words of length at most two; hence, this property fully characterizes the tractability of the problem. On the other hand, if we allow for an approximation error, then it is straightforward to obtain an efficient scheme (FPRAS) for an additive approximation. Yet, a multiplicative approximation is harder to obtain. We establish that in the case of conjunctive regular path queries, a multiplicative approximation of the Shapley value of an edge can be computed in polynomial time if and only if all query atoms are finite languages (assuming non-redundancy and conventional complexity limitations). We also study the analogous situation where we wish to determine the contribution of a vertex, rather than an edge, and establish complexity results of similar nature.

2012 ACM Subject Classification Theory of computation → Data provenance

Keywords and phrases Path queries, regular path queries, graph databases, Shapley value

Digital Object Identifier 10.4230/LIPIcs.ICDT.2023.11

Related Version *Full Version*: `https://arxiv.org/abs/1412.2221` [15]

Funding This work was supported by the Israel Science Foundation (ISF), Grant 768/19, and the German Research Foundation (DFG) Project 412400621 (DIP program).

1 Introduction

Graph databases arise in common applications where the underlying data is a network of entities, especially when connectivity and path structures are of importance. Such usage spans many fields, including the Semantic Web [2], social networks [10], biological networks [20, 38], data provenance [1], fraud detection [30], recommendation engines [37], and many more. In its simplest form, a graph database is a finite, directed, edge-labeled graph. Vertices represent entities and edges represent binary relationships of different types (labels) between entities. Query mechanisms for graph databases enable the retrieval of parts of the graph according to patterns of connections between vertices.

A canonical example of a graph query is the Regular Path Query (RPQ) [5, 7, 8, 36]. An RPQ qualifies paths using a regular expression over the edge labels. When evaluated on a graph, the answers are source-target pairs of vertices that are connected by a path that conforms to the regular expression. This allows users to inspect complex connections in graphs by enabling them to form queries that match arbitrarily long paths. An important generalization of the class of RPQs is the class of *Conjunctive Regular Path Queries* (CRPQs) that extend regular path queries to conjunctions of atoms, each being an RPQ that should hold between two specified variables [7, 8].

Being simple and expressive, RPQs and CRPQs are an integral part of popular graph query languages for graphs, such as GraphLog, Cypher, XPath, and SPARQL. Therefore, they motivate and give rise to much research effort, including the study of some natural computational problems and variations thereof [11, 22, 23, 26]: What is the complexity of deciding whether an RPQ matches a path from a given vertex to another (what we refer to as *Boolean query evaluation*)? Can we efficiently count and enumerate these paths? Is a given CRPQ contained in another given CRPQ? The combined complexity of Boolean query evaluation is in polynomial time for RPQs and NP-complete for CRPQs [4]. Data complexity, however, is NLogspace-complete for both [3]. The containment problem for RPQs is Pspace-complete, and for CRPQs, it is Expspace-hard [6, 11].

In this paper, we focus on the problem of *quantifying the responsibility and contribution* of different components in the graph, namely edges and vertices, to an answer to the CRPQ (and RPQ in particular). This problem has been studied in the context of queries on relational databases, and our motivation here is the same as in the relational context: we wish to provide the database user with an explanation of *why* (or what in the database led to that) we got a specific answer; when many combinations of data items can lead to an answer, and the lineage is too large or complex, we wish to quantify the contribution of individual items in order to distinguish the more important from the less important to the answer [9].

How does one quantify the contribution of a database item to a query answer? In the relational model, several definitions and frameworks have been proposed for measuring the contribution of a tuple. For example, Meliou et al. [25] defined the responsibility of a tuple t as, roughly, the inverse of the minimal number of tuples needed to be removed in order to make t counterfactual (i.e., the query answer is determined by the existence of t); this measure is an adaptation of earlier notions of formal causality by Halpern and Pearl [13]. *Causal effect* is another measure proposed by Salimi et al. [31]: if the database is probabilistic and each tuple has independently the probability $1/2$ of existence, how does the probability of the answer change if we assume the existence or absence of t? Lastly, and most relevant to our work, recent work has studied the adoption of the *Shapley value* as a responsibility measure [9, 16, 18, 29].

The Shapley value is a formula for wealth distribution in a cooperative game [32]. In databases, the conceptual application is straightforward: the tuples are the players who play the game of answering the Boolean (or numerical) query; hence, the wealth function is the result of the query [16]. The Shapley value has a plethora of applications, including profit sharing between ISPs [21], influence measurement in social network analysis [28], determining the most important genes for specific body functions [27], and identifying key players in terrorist networks [34], to name a few. Closer to databases is a recent application to model checking for measuring the influence of formula components [24]. As another example, in machine learning, the SHAP score [19] has been used for measuring the contribution of each feature to the prediction, and it is essentially the Shapley value with the features as players. This value was also used for quantifying the responsibility that every tuple has on the *inconsistency* of a knowledge base [14, 39] and a database [18]. The Shapley value is often intractable to calculate, and particularly, the execution cost might grow exponentially with the number of players. Hence, past research has been investigating islands of tractability and approximation algorithms.

Contribution. We study the complexity of computing the Shapley value of edges and vertices for CRPQs over graph databases. In the remainder of this section and throughout the paper, we focus on edges (and discuss vertices in Section 6). Computing the Shapley value of an edge e then boils down to answering the following question. If we eliminate all edges and add them back one by one in uniformly random order, what is the probability that e is a *counterfactual cause* (i.e., its inclusion is necessary and sufficient [25]) for the answer at hand? As done in previous work in the context of relational databases [18, 25], we view the graph as consisting of two types of edges: *endogenous* edges and *exogenous* edges. The endogenous edges are the ones that we consider for reasoning on responsibility, and they are the players of the game. The exogenous edges constitute external knowledge that we take for granted, and so, they are not players in the cooperative game (and not eliminated at the beginning of the probabilistic process).

To be more precise, an instance of our problem involves a query q (e.g., an RPQ or a CRPQ), an input graph G, an answer tuple $\vec{t}$ of vertices of G, and an edge e whose contribution to $\vec{t}$ we seek to measure. We adopt the yardstick of *data complexity* [35] where we consider the query q as fixed. Hence, each fixed query q is associated with a distinct computational problem that takes as input G, $\vec{t}$, and e.

We first show that the exact computation of the Shapley value is almost always hard. Specifically, it is sufficient for the CRPQ to have a non-redundant atom (i.e., a conjunct associated with a regular language) with a word of length three or more for the computation to be #P-hard ($\mathrm{FP}^{\#\mathrm{P}}$-complete). In addition, for RPQs (i.e., single-atom CRPQs), we complete this hardness condition to a full dichotomy by showing that the Shapley value can be computed in polynomial time if the language contains only words of length at most two.

Next, we study the complexity of approximation. In our context, we adopt a standard yardstick of tractable approximation, namely FPRAS (Fully Polynomial-Time Approximation Scheme). An approximation of the Shapley value of an edge to a CRPQ can be computed via a straightforward Monte Carlo (average-over-samples) estimation of the probability that we previously defined. This estimation guarantees an additive (or absolute) approximation. However, we are also interested in a multiplicative (or relative) approximation.

We establish a dichotomy that classifies CRPQs into a class where there is a multiplicative FPRAS and the complementing class where there cannot be any such FPRAS under conventional complexity assumptions. Specifically, if the CRPQ contains an atom (non-redundant atom) with an infinite regular language, then (any) multiplicative approximation is intractable since it is already NP-complete to determine whether the Shapley value is nonzero. In every other case (assuming no redundant atoms), an additive FPRAS can also be used to obtain a multiplicative FPRAS, due to the *gap property*, previously established in the relational model [18, 29]: if the Shapley value is nonzero, then it is at least the reciprocal of a polynomial. Note that this is contrasting the situation with relational conjunctive queries, where there is always a multiplicative FPRAS [16]. Intuitively, this is true since, unlike the case of conjunctive queries, in the case of RPQs and CRPQs we do not necessarily have any fixed upper bound on the minimal number of tuples (edges) that need to be present for e to become a counterfactual cause.

Moving on from edges to vertices, the complexity situation remains quite similar. In particular, it is generally hard to compute the exact Shapley value of a vertex: it is sufficient for the CRPQ to have a non-redundant atom that contains a word of length four or more for the computation to be hard. For RPQs, we establish that the family of tractable queries for edges is also tractable for vertices. Yet, for vertices, a gap remains and we do not complete a full classification. For approximate evaluation, we establish the same dichotomy as for edges.

Organization. The rest of the paper is organized as follows. We introduce some basic terminology in Section 2. In Section 3, we formally define how the Shapley value is applied in our setting for edges in graph databases. We study the complexity of computing exact Shapley values for CRPQs in Section 4, and investigate approximations in Section 5. In Section 6, we present the complexity results for the case when measuring the contribution of vertices instead of edges. We conclude and discuss directions for future work in Section 7. For lack of space, some of the proofs are omitted and can be found in the full version of the paper [15].

2 Preliminaries

We begin by setting some terminology and notation that we use throughout the paper.

Graphs and Path Queries

We use Σ to denote a finite alphabet (i.e., set of symbols) that is used for labeling edges of graphs. A *word* is a finite sequence of symbols from Σ. As usual, Σ^* denotes the set of all words. A *language* L is a (finite or infinite) subset of Σ^*. By a slight abuse of notation, we may identify a language L with a representation of L such as a regular expression or a finite-state automaton. A *regular expression* is defined as follows: $\emptyset$, ϵ, and $\sigma \in \Sigma$ represent the empty language, the empty word, and the symbol σ, respectively; and if r and s are regular expressions, then $(r \mid s)$ and $(r \cdot s)$ and (r^*) are also regular expressions, denoting union, concatenation, and Kleene star, respectively. We sometimes omit parentheses and dots when there is no risk of ambiguity (so we may write rs instead of $(r \cdot s)$, for instance). The language $L(r)$ that r accepts (recognizes) is defined as usual. We abbreviate by Σ^* the regular expression that accepts every word. A *deterministic finite automaton* (DFA) A is a tuple $(Q, \Sigma, \delta, q_0, F)$, where Q is a finite set of states, Σ is a finite alphabet, $\delta \colon Q \times \Sigma \to Q$ is the transition function, q_0 is the initial state, F is the set of accepting states. By $\delta^*(w)$ we denote the state that the automaton reaches after reading w, starting from the initial state. The automaton *accepts* a word w if $\delta^*(w) \in F$. We again use $L(A)$ to denote the language that A recognizes. (Recall that the classes of regular expressions and DFAs coincide in their expressive power.)

By a *graph* we mean an edge-labeled directed graph $G = (V, E)$ where V is the finite set of vertices and $E \subseteq V \times V$ is the set of edges (v, u), each with a label $\mathit{lbl}(v, u)$. We will consistently denote by n and m the number of vertices and edges, respectively; that is, $n = |V|$ and $m = |E|$. A path p from the vertex u to the vertex v in G is a sequence $p = (v_0, v_1), (v_1, v_2), \dots, (v_{k-1}, v_k)$ of edges in G such that $u = v_0$ and $v = v_k$. By $|p|$ we denote the length k of p, and by $\mathit{lbl}(p)$ we denote the word $\mathit{lbl}(v_0, v_1) \cdots \mathit{lbl}(v_{k-1}, v_k)$. If $G = (V, E)$ is a graph and $E' \subseteq E$ is a set of edges, then we denote by $G[E']$ the subgraph $G' = (V, E')$ of G. In other words, $G[E']$ is obtained from G by removing every edge in $E \setminus E'$.

A *path query* q has the form (x, L, y) where x and y are variables and L is a language. When evaluated on a graph G, it returns the set $q(G)$ of all pairs (s, t) such that s and t are vertices of G and there exists a path p from s to t with $\mathit{lbl}(p) \in L$. An answer (s, t) is viewed as an assignment of s and t to x and y, respectively; this will become important later when we combine multiple path queries. For convenience, we may view q as a function that takes G, s and t as input, where $q[s, t](G) = 1$ if (s, t) is an answer and $q[s, t](G) = 0$ otherwise. As a special case, a *regular path query* (RPQ) is such that L is a regular language, defined via a regular expression r or an automaton A. We sometimes use the shorthand L for the query (x, L, y), or r in the case of a regular expression.

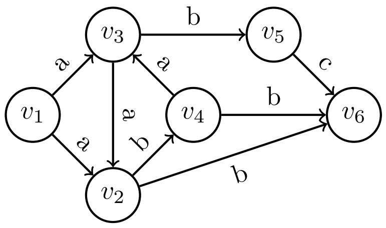


Figure 1 The graph of our running example. In the paper, we denote the edge (v_i, v_j) as e_{ij}.

▶ **Example 1.** Figure 1 depicts the graph G over $\Sigma = \{a, b, c\}$ for our running example. We show a few examples of RPQs on G.

- $q_1 = \Sigma^*$. This query tests whether there is a path from s to t in G. For example, we have that $q_1[v_1, v_2](G) = 1$, and $q_1[v_1, v_6](G) = 1$, since there are paths from v_1 to both v_2 and v_6. In contrast, $q_1[v_3, v_1](G) = 0$ since there is no path from v_3 to v_1.
- $q_2 = abc$. This query tests whether there is a path from s to t in G that matches the word abc. For example, we have that $q_2[v_1, v_6](G) = 1$, as there is a path $v_1 \to v_3 \to v_5 \to v_6$ that matches abc. But $q_2[v_3, v_5](G) = 0$, as the only path from v_3 to v_5 consists of a single edge labeled b.
- $q_3 = ab^*$. This query tests whether there is a path from s to t in G that matches regular expression ab^*. We have $q_3[v_1, v_6](G) = 1$ due to the path $v_1 \to v_2 \to v_4 \to v_6$, or alternatively, $v_1 \to v_2 \to v_6$, that match ab^*. But we have $q_2[v_3, v_5](G) = 0$, as the only path from v_3 to v_5 consists of a single edge with label b, which is not a match for ab^*.

We will later use these queries to illustrate additional concepts in the paper. ⌟

Conjunctive Regular Path Queries

A *conjunctive regular path query*, CRPQ for short, is a conjunction of RPQs with possibly shared variables. More precisely, a CRPQ q has the form

$$q[x_1, \dots, x_k] = \bigwedge_{i=1}^{m} (y_i, r_i, z_i) \tag{1}$$

where each y_i and z_i is a variable from $\{x_1, \dots, x_k\}$ and each r_i is a regular expression. The RPQ (y_i, r_i, z_i) is also referred to as the ith atom of q and is denoted by q_i. As before, when evaluated on a graph G, we denote by $q(G)$ the set of all of assignments $(u_1, \dots, u_k)$ to $(x_1, \dots, x_k)$, such that all atoms are satisfied. We also denote the assignment $(u_1, \dots, u_k)$ as a function $\mu : \{x_1, \dots, x_k\} \to \{u_1, \dots, u_k\}$ such that $\mu(x_i) = u_i$ for $i = 1, \dots, k$. We use a numeric notation similarly to RPQs, that is, $q[u_1, \dots, u_k](G) = 1$ if $(u_1, \dots, u_k)$ is an answer and $q[u_1, \dots, u_k](G) = 0$ otherwise.

▶ **Example 2.** Let us look at the query $q[x_1, x_2, x_3] = (x_1, a^*, x_2) \wedge (x_2, b^*, x_3)$. When evaluated on a graph, this query returns triplets (u_1, u_2, u_3) such that there is a path from u_1 to u_2 of edges labeled a, and from u_2 to u_3 of edges labeled b. In our running example (Figure 1), we have that $q[v_1, v_2, v_6](G) = 1$, as there is a path $v_1 \to v_2$ that matches a^*, and a path $v_2 \to v_4 \to v_6$ that matches b^*. Yet, $q[v_1, v_3, v_6](G) = 0$, since every match from v_3 to v_6 contains a label that is not b. ⌟

An atom q_j is *redundant* if its removal from q results in a query that is equivalent to q. Formally, denote by $q^{\setminus j}$ the CRPQ that is obtained from q by removing the jth atom.

$$q^{\setminus j}[x_1, \dots, x_k] = \bigwedge_{i=1; i \neq j}^{m} (y_i, r_i, z_i)$$

Then the jth atom is *redundant* if $q \equiv q^{\setminus j}$, that is, $q(G) = q^{\setminus j}(G)$ for all graphs G.

▶ **Example 3.** Let us look at $q[x_1, x_2, x_3] = (x_1, a, x_2) \wedge (x_2, b, x_3) \wedge (x_1, a^*b^*, x_3)$. In this query, the third atom is redundant according to our definition, as removing it does not change the result set on any graph. Intuitively, if the first two queries return true then so does the third, thus the third atom does not add any restriction to the conjunction. ⌟

We later refer to the following obvious (and standard) observation.

▶ **Observation 4.** *Let q be a CRPQ. If the ith atom is non-redundant, then there exists a graph G and assignment μ to $(x_1, \dots, x_k)$ such that $q_j[\mu(y_j), \mu(z_j)](G) = 1$ for $j \neq i$ and $q_i[\mu(y_i), \mu(z_i)](G) = 0$.*

In the sequel, we say that q is *without redundancy* if every atom of q is non-redundant. Note that every CRPQ q can be made one without redundancy (while preserving equivalence) by repeatedly removing redundant atoms.

The Shapley Value

Let A be a finite set of players. A cooperative game is a function $v\colon P(A) \to \mathbb{R}$, where $P(A)$ is the power set of A (containing all subsets of A), such that $v(\emptyset) = 0$. For $S \subseteq A$, the value $v(S)$ represents a value, such as wealth, jointly obtained by S when the players of S cooperate. The Shapley value for the player a is defined to be:

$$\text{Shapley}(A, v, a) = \frac{1}{|A|!} \sum_{\pi \in \Pi_A} (v(\pi_a \cup \{a\}) - v(\pi_a)). \tag{2}$$

Here, Π_A is the set of all possible permutations over the players in A, and for each permutation π we denote by π_a the set of players that appear before a in the permutation. Alternatively, the Shapley value can be written as follows:

$$\text{Shapley}(A, v, a) = \sum_{B \subseteq A \setminus \{a\}} \frac{|B|!(|A| - |B| - 1)!}{|A|!} (v(B \cup \{a\}) - v(B)).$$

Intuitively, the Shapley value of a player a is the expected contribution of a to the value $v(B)$ where B is a set of players chosen by randomly (and uniformly) selecting players one by one without replacement. The Shapley value is known to be unique up to some rationality axioms that we omit here (c.f. [32]).

3 The Shapley Value of Edges

Throughout the paper, we focus on the Shapley value of edges of the input graph G. Later, in Section 6, we also discuss the extension of our results to the Shapley value of vertices.

Given a CRPQ q, our goal is to quantify the contribution of edges in the input graph G to an answer $\vec{u}$ for q. We adopt the convention that, for the sake of measuring contribution, the database is viewed as consisting of two types of data items – we reason about the contribution of the *endogenous* items while we take for granted the existence of the *exogenous* items (that serve as out-of-game background) [16, 25, 31]. Hence, in our setup, we view the graph as consisting of two types of edges: *endogenous edges* and *exogenous edges*. For a graph $G = (V, E)$, we denote by E_n and E_x the sets of endogenous and exogenous edges, respectively, and we assume that E is the disjoint union of E_n and E_x.

Our goal is to quantify the contribution of an edge $e \in E_\mathsf{n}$ to an answer $\vec{u} = (u_1, \dots u_k)$ of the query q, that is, to the fact that $q[\vec{u}](G) = 1$. To this end, we view the situation as a cooperative game where the players are the endogenous edges. The Shapley value of an edge $e \in E_\mathsf{n}$ in this setting will be denoted by $\mathrm{Shapley}\langle q\rangle(G, \vec{u}, e)$.

$$\mathrm{Shapley}\langle q\rangle(G, \vec{u}, e) \stackrel{\mathrm{def}}{=} \mathrm{Shapley}(E_\mathsf{n}, v_q, e)$$

where the function Shapley is as defined in Equation (2) and v_q is the numerical function that takes as input a subset of the endogenous edges and is defined as follows:

$$v_q(B) \stackrel{\mathrm{def}}{=} q[\vec{u}](G[B \cup E_\mathsf{x}]) - q[\vec{u}](G[E_\mathsf{x}])$$

In particular, $v_q(\emptyset) = 0$. Put differently, we have the following.

$$\mathrm{Shapley}\langle q\rangle(G, \vec{u}, e) = \sum_{B \subseteq E_\mathsf{n} \setminus \{e\}} \frac{|B|!(|E_\mathsf{n}| - |B| - 1)!}{|E_\mathsf{n}|!} \Big(q[\vec{u}](G[B \cup E_\mathsf{x} \cup \{e\}]) - q[\vec{u}](G[B \cup E_\mathsf{x}])\Big) \quad (3)$$

For a CRPQ q, the computational problem $\mathsf{CRPQShapley}\langle q\rangle$ is that of computing the Shapley value of a given edge:

Problem $\mathsf{CRPQShapley}\langle q\rangle$	
Parameter:	CRPQ q
Input:	Graph G, vertex vector $\vec{u} = (u_1, \dots, u_k)$, endogenous edge e
Goal:	Compute $\mathrm{Shapley}\langle q\rangle(G, \vec{u}, e)$

When q has only one atom, and is in fact an RPQ (x, r, y) with r being a regular expression, we may replace q with r in the notation and write $\mathrm{Shapley}\langle r\rangle(G, s, t, e)$ and $\mathsf{RPQShapley}\langle r\rangle$ with the meaning of $\mathrm{Shapley}\langle q\rangle(G, s, t, e)$ and $\mathsf{RPQShapley}\langle q\rangle$, respectively.

▶ **Example 5.** Considering the running example of Figure 1, assume that all edges are endogenous. Let us first compute the contribution of the edges to the answer (v_2, v_6) to b^*.

- The edge e_{26} changes $q[v_2, v_6](G)$ from 0 to 1 if and only if it is selected first or second among $\{e_{24}, e_{26}, e_{46}\}$. This event happens with probability 2/3, so

  $$\mathrm{Shapley}\langle b^*\rangle(G, v_2, v_6, e_{26}) = 2/3\,.$$

- For e_{24} to increase the value, it should be selected before e_{26} and after e_{46}, and this happens with probability 1/6. Hence, $\mathrm{Shapley}\langle b^*\rangle(G, v_2, v_6, e_{24}) = 1/6$.
- Similarly to e_{24}, $\mathrm{Shapley}\langle b^*\rangle(G, v_2, v_6, e_{46}) = 1/6$.
- Every other edge is irrelevant to the answer (v_2, v_6), and so, its Shapley value is zero.

Note that the sum of the Shapley values of all edges is 1, which is no coincidence, since in general the Shapley value over all players sums up to the overall wealth of the entire set of players [32]. Following are additional examples.

- Shapley$\langle abc\rangle(G, v_1, v_6, e)$. Any edge that is not on the only path that matches abc, namely $p\colon v_1 \to v_3 \to v_5 \to v_6$, will have the Shapley value of zero. For edges on the path p, the computations are similar to each other and they all have the same Shapley value. For one of them to change the query result, it needs to appear after both other edges in the permutation of E_{n}. This happens in $\frac{9!}{3}$ of the overall 9! permutations. So we have:

  $$\text{Shapley}\langle abc\rangle(G, v_1, v_6, e_{13}) = \text{Shapley}\langle abc\rangle(G, v_1, v_6, e_{35}) \\ = \text{Shapley}\langle abc\rangle(G, v_1, v_6, e_{56}) = \frac{1}{3}\,.$$

  If we assume that e_{13} is exogenous, then the other two edges will *split* the contribution evenly. Then we get:

  $$\text{Shapley}\langle abc\rangle(G, v_1, v_6, e_{35}) = \text{Shapley}\langle abc\rangle(G, v_1, v_6, e_{56}) = \frac{1}{2}\,.$$

- Shapley$\langle ab^*\rangle(G, v_1, v_6, e)$. There are two paths that match the regular expression ab^* (as we have seen in Example 1). Again, any edge that is not on any of these paths has the Shapley value zero. But now, the contributions of the remaining edges is not equal since, for instance, e_{12} is on both paths so we expect it to have higher contribution than the others. For the edge e_{26} to change the query result, it needs to appear after edge e_{12} but before at least one of e_{24} and e_{46}. Permutations where this happens are either permutations where e_{26} appears after e_{12} and one of e_{24} and e_{46} but before the other one, and there are $2 \cdot \sum_{i=0}^{5}(i+2)!\binom{5}{i}(8-i-2)! = \frac{2}{12}\cdot 9!$ such permutations. This is also possible in permutations where e_{26} appears after e_{12} but before both e_{24} and e_{46}, and there are $\sum_{i=0}^{5}(i+1)!\binom{5}{i}(8-i-1)! = \frac{1}{12}\cdot 9!$ such permutations. There are an overall of 9! possible permutations, so,

  $$\text{Shapley}\langle ab^*\rangle(G, v_1, v_6, e_{26}) = \frac{1}{4}\,.$$

  For the two edges e_{24} and e_{46}, the computations are similar to each other. For one of them to change the query, it needs to appear after e_{12} and the other one, but before e_{26}. Similar to before, there are $\sum_{i=0}^{5}(i+2)!\binom{5}{i}(8-i-2)! = \frac{1}{12}\cdot 9!$ permutations where this happens, so,

  $$\text{Shapley}\langle ab^*\rangle(G, v_1, v_6, e_{24}) = \text{Shapley}\langle ab^*\rangle(G, v_1, v_6, e_{46}) = \frac{1}{12}\,.$$

  For the last edge e_{12}, it needs to appear after e_{26} or after both e_{24}, e_{46}. Permutations where this happens are either permutations where e_{12} appears second after e_{26}, and these account for $\frac{1}{4}\cdot\frac{1}{3}\cdot 9! = \frac{1}{12}\cdot 9!$ of all permutations, or permutations where e_{12} appears third or forth, and these account for $\frac{1}{2}\cdot 9!$ of all permutations. So overall we get that

  $$\text{Shapley}\langle ab^*\rangle(G, v_1, v_6, e_{12}) = \frac{7}{12}\,.$$

  Note that, again, $\sum_{e\in E_{\mathsf{n}}} \text{Shapley}\langle ab^*\rangle(G, v_1, v_6, e) = 1$, as expected. ⌟

4 The Complexity of Exact Computation

In this section, we study the complexity of CRPQShapley$\langle q\rangle$, where the goal is to compute the exact Shapley value of an edge. Note that the query q is fixed in the analysis, hence, every q defines a separate computational problem CRPQShapley$\langle q\rangle$. The following theorem show that CRPQShapley$\langle q\rangle$ is computationally intractable for almost *every* CRPQ q, except for limited cases. We prove the theorem later, in Section 4.1.

▶ **Theorem 6** (Hardness). *Let q be a CRPQ. If q has a non-redundant atom with a language that contains a word of length three or more, then* $\mathsf{CRPQShapley}\langle q\rangle$ *is* $\mathrm{FP}^{\#\mathrm{P}}$*-complete.*

Recall that $\mathrm{FP}^{\#\mathrm{P}}$ is the class of functions computable in polynomial time with an oracle to a problem in #P (e.g., counting the number of satisfying assignments of a propositional formula). This class is considered intractable, and above the polynomial hierarchy (Toda's theorem [33]).

The question of whether the condition of Theorem 6 is necessary for hardness remains open. Yet, we can show that it is, indeed, necessary, in the case of a single atom (RPQ):

▶ **Theorem 7** (Tractability). *Let q be an RPQ with the regular expression r. If every word in $L(r)$ is of length at most two, then* $\mathsf{RPQShapley}\langle q\rangle$ *is solvable in polynomial time.*

Proof. We give a polynomial-time algorithm for computing $\mathsf{RPQShapley}\langle r\rangle$ where $L = L(r)$ consists of words of length at most two. We denote by $\mathcal{M}(G,k)$ the set of all subsets $E' \subseteq E_{\mathsf{n}}$ of size k such that $G[E_{\mathsf{x}} \cup E']$ contains a path of L from s to t. We have the following from Equation (3):

$$\mathsf{RPQShapley}\langle r\rangle(G,s,t,e) = \sum_{k=0}^{m'-1} \frac{k!(m'-k-1)!}{m'!}|\mathcal{M}(G_e,k)| - \sum_{k=0}^{m'-1} \frac{k!(m'-k-1)!}{m'!}|\mathcal{M}(G\setminus e,k)|.$$

Here, G_e is the same as G, except for e that is exogenous instead of endogenous, $G \setminus e$ is the graph G with the exclusion of e, and $m' = |E_{\mathsf{n}}|$. This shows that the computation of $\mathsf{RPQShapley}\langle r\rangle(G,s,t,e)$ reduces efficiently to computing $|\mathcal{M}(G,k)|$, that is, counting the subsets of E_{n} (of endogenous edges) of size k that, when added to E_{x}, connects s to t via a path that matches a word in $w \in L$.

We assume that L does not contain the empty word. This is without loss of generality, for the following reason. If L contains the empty word ϵ, then either $s = t$ and e has the Shapley value zero (since it is irrelevant), or $s \neq t$ and we can ignore the empty word of L.

We now show that $|\mathcal{M}(G,k)|$ can be computed in polynomial time when $L(r)$ consists of words of length at most two. First, let us observe that we can compute $|\mathcal{M}(G,k)|$ by computing the complement set $|\overline{\mathcal{M}(G,k)}|$ which is defined similarly but for subsets of length k where there is *no* path in L:

$$|\mathcal{M}(G,k)| = \binom{m'}{k} - |\overline{\mathcal{M}(G,k)}|\,.$$

So, it suffices to show how to compute $|\overline{\mathcal{M}(G,k)}|$.

For a subset of endogenous edges to be in $\overline{\mathcal{M}(G,k)}$, it should not connect, with E_{x}, any path from s to t matching $w \in L$ (i.e., matching one of $w_0,\dots,w_l$). In other words, it should not connect any path of length one matching some w_i with $|w_i| = 1$, or any path of length two matching some w_i with $|w_i| = 2$. This partitions the set of endogenous edges into three categories:

- $\mathsf{Permitted}$: Edges that are not part of any path that matches L.
- $\mathsf{Forbidden}$: Edges that connect s to t without needing any other endogenous edge, either because they have a label that constitutes a word w_i, or because they connect a path of length two together with an exogenous edge.
- $\mathsf{On2Path}$: All other edges, that is, the edges that belong to pairs of endogenous edges that are needed together in order to connect s to t through a word in L.

Observe that following. First, the three sets Permitted, Forbidden and On2Path are pairwise disjoint (by definition). Second, On2Path can be partitioned into |On2Path|/2 pairwise-disjoint pairs, each constitutes a path with a word in L. (Note that our data model does not allow for parallel edges.)

It follows that to construct a set of k edges in $\overline{\mathcal{M}(G,k)}$, we can select $i \leq k$ edges from Permitted, then $k-i$ pairs from the |On2Path|/2 pairs, and then one edge from each pair. Hence, we get:

$$|\overline{\mathcal{M}(G,k)}| = \sum_{i=0}^{k} \binom{|\mathsf{Permitted}|}{i} \cdot \binom{\frac{|\mathsf{On2Path}|}{2}}{k-i} \cdot 2^{k-i}. \tag{4}$$

Finally, observe that we can compute each of Permitted, Forbidden and On2Path in polynomial time, and we can then compute Equation (4) in polynomial time. This concludes the proof. ◀

Hence, we get a full classification for RPQs:

▶ **Corollary 8.** *Let q be an RPQ with the regular expression r. Assuming* P $\neq$ NP*, the following are equivalent:*
1. RPQShapley$\langle q \rangle$ *is solvable in polynomial time.*
2. *Every word in $L(r)$ is of length at most two.*

In the remainder of this section, we prove the hardness side (Theorem 6).

4.1 Proof of Hardness

Membership in $\mathrm{FP}^{\#\mathrm{P}}$ is straightforward from the definition of the Shapley value in Equation (2). Indeed, Shapley$\langle q \rangle(G, \vec{u}, e)$ can be computed using an oracle to the problem of counting the permutations over the edge set such that e changes the evaluation from zero (false) to one (true). For the $\mathrm{FP}^{\#\mathrm{P}}$-hardness, we prove it in a sequence of reductions. We begin with hardness for the special case where the language consists of a single three-letter word. For that, we will use a result by Livshits et al. [17] on the computation of Shapley values for facts (tuples) in relational databases. We use that to prove hardness for the general case of a language with one or more words of length at least three, even when restricted to simple graphs.

We first recall the result of Livshits et al. [17]. They considered relational databases D where some of the facts are endogenous and the rest exogenous. As in our notation, the corresponding subsets of D are denoted by D_{n} and D_{x}, respectively. For a Boolean query q that maps every database into $\{0,1\}$, they defined the Shapley value of a fact similarly to the way we define the Shapley value of an edge: the endogenous facts are the players and the query is the wealth function:

$$\mathrm{Shapley}\langle q \rangle(D, f) = \mathrm{Shapley}(D_{\mathsf{n}}, v_{db}, f)$$

where $v_{db}(E) = q(E \cup D_{\mathsf{x}}) - q(D_{\mathsf{x}})$. They established a complete classification of the class of conjunctive queries without self-joins into tractable and intractable queries for the computation of the Shapley value. What is relevant to us is that the following conjunctive query is $\mathrm{FP}^{\#\mathrm{P}}$-hard:

$$Q_{\mathsf{RST}}()\colon \exists x, y[R(x) \wedge S(x,y) \wedge T(y)]$$

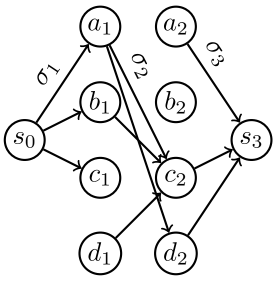
We define a special kind of graphs that will help us in some of the proofs. A graph $G = (V, E)$ is called a *leveled graph* if there exists a split of the vertex set into levels $V_0, \dots, V_k$, such that:

R
a
b
c

S	
a	c
a	d
b	c
d	c

T
a
c
d

(a) Input database D.



(b) Reduction graph G.

Figure 2 An example for the construction in the reduction of the proof of Lemma 9.

1. The set of vertices V is the disjoint union of $V_0, \dots, V_k$.
2. Every edge is from a vertex of some level V_i to a vertex of V_{i+1}.

From the hardness of the Shapley value for Q_{RST}, it is easy to prove the following.

▶ **Lemma 9.** *Let $\sigma_i \in \Sigma$ for $i = 1, 2, 3$.* RPQShapley$\langle \sigma_1\sigma_2\sigma_3 \rangle$ *is* $\mathrm{FP}^{\#\mathrm{P}}$*-hard, even when restricted to leveled graphs.*

The proof (given in the long version of the paper) is via the reduction illustrated in Figure 2. Next, we have the following generalization of Lemma 9.

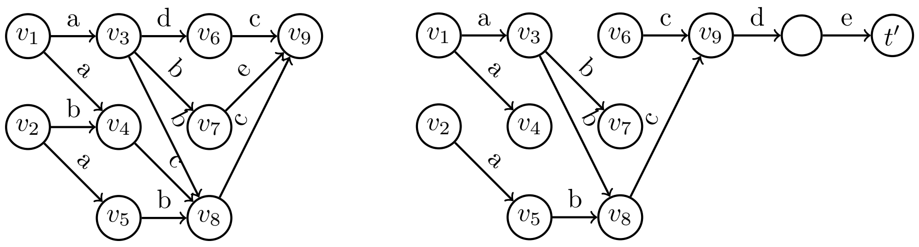
▶ **Lemma 10.** *Let r be a regular expression. If there exists a word in $L(r)$ of length at least three, then* RPQShapley$\langle r \rangle$ *is* $\mathrm{FP}^{\#\mathrm{P}}$*-hard, even when restricted to leveled graphs.*

The reduction from the problem of Lemma 9 to that of Lemma 10 is illustrated in Figure 3 (and explained in the full version). With Lemma 10, we can prove Theorem 6.

Proof of Theorem 6. We know that q has a non-redundant atom q_i such that $L(r_i)$ contains a word of length at least three. We reduce RPQShapley$\langle r_i \rangle$ on leveled graphs (Lemma 10) to CRPQShapley$\langle q \rangle$. Given an input leveled graph G, source vertex s, target vertex t and edge e for RPQShapley$\langle r_i \rangle$, we construct an input instance G^* for CRPQShapley$\langle q \rangle$.

Since the ith atom is non-redundant, we can use Observation 4 and conclude that there exists a graph G_i and assignment $\vec{v}$ to $\vec{x}$ such that all RPQ atoms return true except for the ith atom; that is, we have that $q_j[s_j, t_j](G_i) = 1$ for every $j \neq i$ and $q_i[s_j, t_j](G_i) = 0$. Here, s_j and t_j are the vertices assigned to the variables y_j and z_j, respectively, from Equation (1).

We assume that G and G_i are disjoint. We construct a new graph G^* by adding G to G_i, and merging s and t into s_i and t_i, respectively. Hence, in G^*, the vertex s_i has all edges that it has in G_i, in addition to the outgoing edges that s has in G. Similarly, in G^*, the



(a) Input graph G. **(b)** Graph G' constructed in the reduction.

Figure 3 An example for the reduction in Lemma 10, for a regular expression that accepts the word *abcde*, source vertex $s = v_1$, target vertex $t = v_9$.

vertex t_i has all of the edges that it has in G_i, in addition to the incoming edges that t has in G. In G^*, we set all of the edges of G_i to be exogenous ones. Moreover, G^* and G have the same set of endogenous edges.

To complete the proof, observe that $q[\vec{v}](G^*)$ is equal to $q_i[s,t](G)$. To see that, observe that G^* has a matching path for the jth atom for all $j \neq i$ (since G_i does). Recall also that there are no paths from s_i to t_i matching r_i in G_i. Hence, from our construction of G^* (and in particular given that s_i and t_i are not part of any cycle), we get that every path from s_i to t_i that matches r_i should be fully contained in G. Hence, G^* has a q_i path from s_i to t_i if and only if G has a q_i path from s to t.

Let E_{x} and E_{n} be the sets of exogenous and endogenous edges of G, respectively, and let E^*_{x} be the set of exogenous edges of G^*. We can extend the above argument to conclude that

$$q[\vec{v}](G^*[E^*_{\mathsf{x}} \cup E']) = q_i[s,t](G[E_{\mathsf{x}} \cup E'])$$

for all subsets E' of E_{n}, since every edge of G_i is exogenous in G^*. From that we can now conclude that $\text{Shapley}\langle r_i\rangle(G,s,t,e) = \text{Shapley}\langle q\rangle(G^*,\vec{v},e)$, as claimed. ◀

This completes the proof of the hardness side of Theorem 6.

5 Complexity of Approximation

We now study the complexity of approximating $\mathsf{CRPQShapley}\langle q\rangle$. We aim for a *fully polynomial randomized approximation scheme*, or FPRAS for short. Formally, an FPRAS for a numeric function f is a randomized algorithm $A(x,\epsilon,\delta)$, where x is an input for f and $\epsilon,\delta \in (0,1)$, such that $A(x,\epsilon,\delta)$ returns an ϵ-approximation of $f(x)$ with probability $1-\delta$ (where the probability is over the randomness of A) in time polynomial in x, $1/\epsilon$ and $\log(1/\delta)$. We distinguish between an *additive* FPRAS:

$$\Pr\left[f(x) - \epsilon \leq A(x,\epsilon,\delta) \leq f(x) + \epsilon\right] \geq 1 - \delta$$

and a *multiplicative* FPRAS:

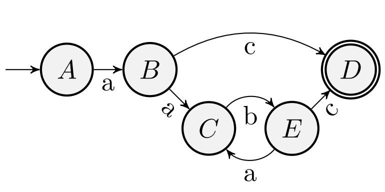
$$\Pr\left[\frac{f(x)}{1+\epsilon} \leq A(x,\epsilon,\delta) \leq (1+\epsilon)f(x)\right] \geq 1 - \delta.$$

5.1 Results

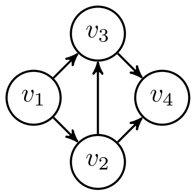
There is a simple Monte-Carlo algorithm that guarantees an additive approximation for the Shapley value on any CRPQ, and we show that it also serves as a multiplicative FPRAS in some cases. In this section, we establish a dichotomy in the complexity of multiplicative approximation for the class of CRPQs. We note that here and later on, we sometimes give results for general CRPQs, yet without redundancy. These results generalize to CRPQs with redundant atoms by application to any CRPQ obtained by repeatedly eliminating redundancy (as mentioned in Section 2).

▶ **Theorem 11.** *Let q be a CRPQ without redundancy. If $L(r_i)$ is finite for every atom r_i of q, then $\mathsf{CRPQShapley}\langle q\rangle$ has a multiplicative FPRAS. Otherwise, $\mathsf{CRPQShapley}\langle q\rangle$ has no multiplicative approximation (of any ratio) or else NP ⊆ BPP.*

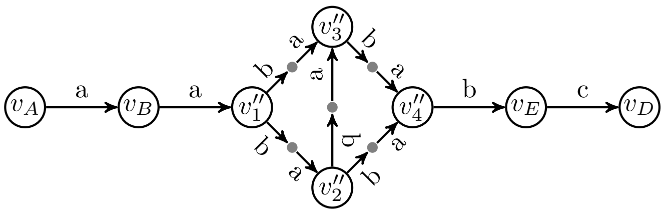
In the remainder of this section, we prove Theorem 11, starting with the hardness side (Section 5.2) and moving on to the FPRAS algorithm (Section 5.3).



(a) The DFA for regular expression $a(a+b)^*c$.



(b) Input graph G.



(c) The graph G' of the reduction for input instance (G, v_1, v_4, e).

Figure 4 An example for the construction in the reduction of the proof of Lemma 15.

5.2 Proof of Hardness

For the hardness, we use a direct consequence of the characterization of Fortune, Hopcroft and Wyllie [12] of the *subgraph homeomorphism problem*:

▶ **Proposition 12.** *It is* NP*-complete to determine, given a graph G, vertices s and t, and edge e, whether e lies on any simple path from s to t.*

Equipped with Proposition 12, we can now show the hardness for Σ^* using the following characterization of when the Shapley value of an edge is nonzero.

▶ **Lemma 13.** *Let G be a graph where all edges are endogenous. Let s and t be two vertices of G, and e an edge of G.* $\text{Shapley}\langle\Sigma^*\rangle(G, s, t, e) > 0$ *if and only if e belongs to a simple path from s to t.*

Proof. Denote by q the RPQ (x, Σ^*, y). We handle separately each direction of the claim. If $\text{Shapley}\langle\Sigma^*\rangle(G, s, t, e) > 0$, then it follows from the definition of the Shapley value that there exists a subset E' of the edges such that $q[s,t](G[E']) = 0$ and $q[s,t](G[E' \cup \{e\}]) = 1$. Let $G' = G[E' \cup \{e\}]$. Then G' contains a simple path from s to t. If this simple path does not contain e, then it is a simple path in $G[E']$, which contradicts the fact that $q[s,t](G[E']) = 0$. Conversely, suppose that e lies on a simple path P from s to t in G. If the random selection of Shapley selects precisely all edges of P except for e, then the addition of e would change the result from 0 to 1. Hence, the Shapley value is nonzero. ◀

Hence, from Proposition 12 and Lemma 13 we conclude the following corollary, which proves the hardness part of Theorem 11 for the language Σ^*.

▶ **Corollary 14.** *It is* NP*-complete to determine whether* $\text{Shapley}\langle\Sigma^*\rangle(G, s, t, e) > 0$*, given G, s, t and e, even if all edges of G are assumed to be endogenous.*

Next, we generalize Corollary 14 from Σ^* to any arbitrary infinite regular language r.

▶ **Lemma 15.** *Let r be a regular expression. If $L(r)$ is infinite, then it is* NP*-complete to determine whether* $\text{Shapley}\langle r\rangle(G, s, t, e) > 0$.

Proof sketch. It is straightforward to show that the problem is in NP, as any subset of endogenous edges that adding e to it connects a matching path serves as a witness and can be verified in polynomial time. We will prove NP-hardness by showing a reduction from the problem of determining whether $\text{Shapley}\langle\Sigma^*\rangle(G, s, t, e) > 0$ where all edges are endogenous, and then apply Corollary 14. Given an input instance (G, s, t, e), we will show how to construct an instance (G', s', t', e') for our problem so that $\text{Shapley}\langle\Sigma^*\rangle(G, s, t, e) > 0$ if an only if $\text{Shapley}\langle r\rangle(G', s', t', e') > 0$.

Since $L(r)$ is infinite, we know that its corresponding DFA A has at least one cycle. We find a path from an initial state to an accepting state that passes through a state s of A that participates in a cycle. We will denote the path by: $l : s_0 \to \ldots \to s_i \to \ldots \to s_k$ where $s_i = s$.

We assumed s is a part of a cycle. Let us denote the labels along the cycle starting, from s, by $w_o = \sigma_0 \ldots \sigma_c$. The graph G' is constructed from G so that every path from s' to t' matches r in the following way. The graph G' consist of three subgraphs, as illustrated in Figure 4.

- A copy of the graph G where every edge is replaced with a fresh path of c edges with labels matching w_o. We denote the correspondent of each vertex v of G as v''. Hence, s and t become s'' and t'', respectively. All of the edges in this part are endogenous.
- A copy of the path $s_0 \to \ldots \to s_i$, with the same labels as in the DFA A, where we identify s_i with s''. The edges of this part are all exogenous.
- The path $s_i \to \ldots \to s_k$ with the same labels as in the DFA A, there we now identify s_i with t''. The edges of this part are all exogenous.

We now define s' to be the copy of s_0, we define t' to be the copy of s_k, and we choose as e' any edge along the path that replaces e.

From here it is easy to prove that e belongs to a simple path of G from s to t if and only if e' belongs to a simple path of G' from s' to t', and from there we conclude similarly to Lemma 13 that $\text{Shapley}\langle\Sigma^*\rangle(G, s, t, e) > 0$ if and only if $\text{Shapley}\langle r\rangle(G', s', t', e') > 0$. ◀

Finally, we extend Lemma 15 from RPQs to CRPQs similarly to the way we proved Theorem 6.

▶ **Lemma 16.** *Let q be a CRPQ without redundancy. If $L(r_i)$ is infinite for some atom r_i of q, then determining whether* $\text{Shapley}\langle q\rangle(G, \vec{u}, e) > 0$ *is* NP*-complete.*

This completes the proof of the hardness side of Theorem 11. Next, we will prove the positive side.

5.3 Proof of Tractability

We now show that for every CRPQ where the hardness condition of Theorem 11 does not hold, a multiplicative FPRAS exists. We start by showing that in this case, the *gap property* (as defined by Livshits et al. [17]) holds: if the Shapley value is nonzero, then it must be at least the reciprocal of a polynomial.

▶ **Lemma 17.** *Let q be a fixed CRPQ without redundancy. If $L(r_i)$ is finite for every atom r_i of q, then there exists a polynomial p such that* $\text{Shapley}\langle q\rangle(G, \vec{u}, e)$ *is either zero or at least* $1/p(|G|)$.

Proof. If there is no subset E' of E_{n} such that adding e to $E' \cup E_{\mathsf{x}}$ changes the value of query q from false to true, then $\text{Shapley}\langle q\rangle(G, \vec{u}, e) = 0$. Otherwise, let E' be a minimal such set. Then $|E'| \leq k_1 + \ldots + k_m$, where each k_i is the length of the longest word in $L(r_i)$; the language for the i-th atom in q, as at worst case, the paths match the longest word for each RPQ. Since each $L(r_i)$ is finite, every k_i is a finite constant, and so, $k = k_1 + \ldots + k_m$ is a constant.

Returning to the definition of the Shapley value (Equation (2)), the probability of selecting a permutation π such that π_e is exactly $E' \setminus \{e\}$ is

$$\frac{(|E|-1)!(m'-|E|)!}{m'!} \geq \frac{(m'-k)!}{m'!}$$

where $m' = |E_{\mathsf{n}}|$. Hence, we have that

$$\text{Shapley}\langle q\rangle(G, \vec{u}, e) \geq \frac{(m'-k))!}{m'!} = \frac{1}{(m'-k+1)\cdot\ldots\cdot m'}\,.$$

This completes the proof. ◀

Similarly to Livshits et al. [17], we can use the gap property to show that an additive FPRAS can be turned into a multiplicative FPRAS.

▶ **Lemma 18.** *Let q be a CRPQ without redundancy. If $L(r_i)$ is finite for every atom r_i of q, then* CRPQShapley$\langle q\rangle$ *has both an additive and a multiplicative FPRAS.*

Proof. Using the Chernoff-Hoeffding bound, we can get an additive FPRAS of the value Shapley$\langle q\rangle(G, \vec{u}, e)$, by simply taking the ratio of successes over $O(\log(1/\delta)/\epsilon^2)$ trials of the following experiment:

- Select a random permutation $(e_1, \ldots, e_m)$ over the set of endogenous edges E_{n}.
- Suppose that $e = e_i$, and let $E_{i-1} = \{e_1, \ldots, e_{i-1}\}$. If $q[\vec{u}](G[E_{i-1} \cup E_{\mathsf{x}} \cup \{e\}]) = 1$ and $q[\vec{u}](G[E_{i-1} \cup E_{\mathsf{x}}]) = 0$, then report "success," otherwise, report "failure."

From Lemma 17 (the gap property), we conclude that in order to get a multiplicative ϵ-approximation, it suffices to apply an additive ϵ'-approximation where $1/\epsilon'$ is polynomial in the size of G and in $1/\epsilon$. ◀

5.4 Open Problem: Directed Acyclic Graphs

It is worth noting that the proof of hardness fails when the graph is acyclic, as it relies on Proposition 12. The lemma states that it is NP-complete to determine whether a given graph G has a simple path from a given source vertex s to a given target vertex t through a given edge e. While this is true in the case of a general graph G, the problem is easily solvable in polynomial time when the graph is acyclic, since every path is simple. This leaves open the question of whether we can have a multiplicative FPRAS for the Shapley value even for RPQs with an infinite language. We leave this question for future investigation. Note, however, that for the exact computation there is no change even when restricted to DAGs, since the reductions constructed DAGs.

6 Shapley Value of Vertices

In this section, we discuss the complexity of the Shapley value for *vertices*, rather than *edges*, of the graph. Similarly to the case for edges, our goal is to quantify the contribution of vertices in the input graph to an answer of a path query. So, now, the graph consists of two types of vertices: *endogenous vertices* and *exogenous vertices.*

For a graph $G = (V, E)$, we denote by V_{n} and V_{x} the sets of endogenous and exogenous vertices, respectively, and we assume that V is the disjoint union of V_{n} and V_{x}. If U is a set of vertices, then $G[U]$ denotes the subgraph of G that is induced by U; hence, the vertex set of $G[U]$ is U and the edge set of $G[U]$ consists of every edge of G with both endpoints in U. We denote by $\text{Shapley}\langle q\rangle(G, \vec{u}, w)$ the Shapley value of a vertex $w \in V_{\mathsf{n}}$, that is:

$$\text{Shapley}\langle q\rangle(G, \vec{u}, w) \stackrel{\text{def}}{=} \text{Shapley}(V_{\mathsf{n}}, v_q^{\mathsf{v}}, w)$$

where the function Shapley is as defined in Equation (2) and v_q^{v} is defined as follows:

$$v_q^{\mathsf{v}}(B) \stackrel{\text{def}}{=} q[\vec{u}](G[B \cup V_{\mathsf{x}}]) - q[\vec{u}](G[V_{\mathsf{x}}])$$

We denote by $\mathsf{CRPQShapley}^{\mathsf{v}}\langle q\rangle$ and $\mathsf{RPQShapley}^{\mathsf{v}}\langle r\rangle$ the computational problems that correspond to the ones defined earlier for the Shapley values of edges. We now state the results that we establish with some notes on the changes that should be made in the proofs.

6.1 Complexity of Exact Computation

We can show the following regarding the exact computation of the Shapley value of a graph vertex.

▶ **Theorem 19.** *The following hold for a CRPQ q.*

1. *If q has a non-redundant atom with a language that contains a word of length four or more, then* $\mathsf{CRPQShapley}^{\mathsf{v}}\langle q\rangle$ *is* $\mathrm{FP}^{\#\mathrm{P}}$*-complete.*
2. *If q is an RPQ with the regular expression r, and every word in $L(r)$ is of length at most two, then* $\mathsf{RPQShapley}^{\mathsf{v}}\langle q\rangle$ *is solvable in polynomial time.*

Note that we leave a gap in the classification of RPQs. Theorem 19 states that if there exists a word of length four or more, then the problem is hard, and if all words are of length at most two, then the problem is solvable in polynomial time. The case where there are words of length three but not longer remains an open problem (as opposed to the case of edges where we had a full dichotomy on RPQs due to Corollary 8).

The proof of the hardness part is almost the same as the proof of Theorem 6 for the case of edges. We begin with hardness for the special case where the regular language (or any language) consists of a single four-letter word (rather than three in the case of edges). For that, we use the same result by Livshits et al. [16] on the computation of Shapley values for facts in relational databases. We then continue with the same sequence of reductions as done for edges to get the hardness for a general CRPQ. The proof of the tractability part is also similar to the proof of Theorem 7.

6.2 Complexity of Approximation

For calculating the Shapley value approximately, we get the exact same dichotomy for vertices as Theorem 11 for edges.

▶ **Theorem 20.** *Let q be a CRPQ without redundancy. If $L(r_i)$ is finite for every atom r_i of q, then* $\mathsf{CRPQShapley}^{\mathsf{v}}\langle q\rangle$ *has a multiplicative FPRAS. Otherwise,* $\mathsf{CRPQShapley}^{\mathsf{v}}\langle q\rangle$ *has no multiplicative approximation (of any ratio) or else NP $\subseteq$ BPP.*

The proof is also similar to that of Theorem 11. We establish an FPRAS through a straightforward additive approximation and the gap property. For hardness, we know from Fortune, Hopcroft and Wyllie [12] that it is NP-complete to determine whether a vertex

v lies on a simple path from s to t in a given graph G, and from that we conclude that deciding whether $\text{Shapley}\langle\Sigma^*\rangle(G, s, t, v) > 0$ is also NP-complete. From there we continue with a sequence of reductions that is similar to what we have for the case of edges.

6.3 Summary

We conclude that the complexity for both exact computation and approximation of the Shapley value of vertices is very similar to the case of edges. It is generally hard to compute exact values; it is sufficient for the CRPQ to have an atom that is non-redundant and contains a word of length four or more for the computation to be hard, while for RPQs we identify that the tractable family of queries for edges is also tractable for vertices. For approximation, we have an identical dichotomy for the existence of a multiplicative FPRAS.

7 Concluding Remarks

This work continues the research of responsibility and contribution in databases. We presented the graph-database perspective where the queries are (conjunctive) regular path queries, and the responsibility measure is the Shapley value. We investigated the data complexity of the Shapley value of edges in the graph. For the exact computation, we showed that it is generally hard, while we also showed a specific family of CRPQs where the computation can be done in polynomial time. This is not a full dichotomy for the class of CRPQs, but we establish a dichotomy for the class of RPQs. It remains an open problem whether the condition we have for hardness defines a full dichotomy on CRPQs. We have also studied the complexity of computing an approximation of the Shapley value in the form of an FPRAS. An additive FPRAS is easy to achieve using Monte-Carlo sampling, while a multiplicative approximation is harder. We showed a family of CRPQs where the gap property holds, and hence, an additive FPRAS can be transformed into a multiplicative one. These are the CRPQs where every atom has a finite language. For the other CRPQs, we showed that it is hard to obtain any multiplicative approximation (assuming no redundant atoms). Thus, we achieved a dichotomy on CRPQs for the case of approximation. Finally, we showed that the complexity picture is quite similar (up to a small gap) if we compute the Shapley value of vertices rather than edges.

Several problems remain open. We still do not have a full coverage of all CRPQs for the exact computation of Shapley values. In the case of vertices, we still have a gap already for RPQs. In addition, the proof of the hardness of approximation in Section 5.2 is not valid when the input graph is acyclic; this raises the question of whether there are better opportunities of efficient approximations when the problem is restricted to acyclic graphs. It is also interesting to understand the impact on complexity of adopting other semantics for RPQ evaluation, such as simple paths and shortest paths [22]. Another direction is investigating richer path languages, for example, allowing existentially quantified variables in the query, or negated atoms.

References

1 Manish Kumar Anand, Shawn Bowers, and Bertram Ludäscher. Techniques for efficiently querying scientific workflow provenance graphs. In *EDBT*, volume 426, pages 287–298. ACM, 2010. doi:10.1145/1739041.1739078.

2 Marcelo Arenas and Jorge Pérez. Querying semantic web data with SPARQL. In Maurizio Lenzerini and Thomas Schwentick, editors, *PODS*, pages 305–316. ACM, 2011. doi:10.1145/1989284.1989312.

3 Pablo Barceló Baeza. Querying graph databases. In Richard Hull and Wenfei Fan, editors, *PODS*, pages 175–188. ACM, 2013. doi:10.1145/2463664.2465216.

4 Pablo Barceló, Leonid Libkin, Anthony W Lin, and Peter T Wood. Expressive languages for path queries over graph-structured data. *ACM Transactions on Database Systems (TODS)*, 37(4):1–46, 2012.

5 Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Moshe Y. Vardi. Rewriting of regular expressions and regular path queries. In *PODS*, pages 194–204. ACM, 1999. doi:10.1145/303976.303996.

6 Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Moshe Y. Vardi. Containment of conjunctive regular path queries with inverse. In *KR*, pages 176–185. Morgan Kaufmann, 2000.

7 Mariano P. Consens and Alberto O. Mendelzon. GraphLog: a visual formalism for real life recursion. In *PODS*, pages 404–416. ACM, 1990. doi:10.1145/298514.298591.

8 Isabel F. Cruz, Alberto O. Mendelzon, and Peter T. Wood. A graphical query language supporting recursion. In *SIGMOD*, pages 323–330. ACM, 1987. doi:10.1145/38713.38749.

9 Daniel Deutch, Nave Frost, Benny Kimelfeld, and Mikaël Monet. Computing the Shapley value of facts in query answering. In *SIGMOD*, pages 1570–1583. ACM, 2022.

10 Wenfei Fan. Graph pattern matching revised for social network analysis. In *ICDT*, pages 8–21. ACM, 2012. doi:10.1145/2274576.2274578.

11 Daniela Florescu, Alon Y. Levy, and Dan Suciu. Query containment for conjunctive queries with regular expressions. In Alberto O. Mendelzon and Jan Paredaens, editors, *PODS*, pages 139–148. ACM, 1998. doi:10.1145/275487.275503.

12 Steven Fortune, John E. Hopcroft, and James Wyllie. The directed subgraph homeomorphism problem. *Theor. Comput. Sci.*, 10:111–121, 1980. doi:10.1016/0304-3975(80)90009-2.

13 Joseph Y. Halpern and Judea Pearl. Causes and explanations: A structural-model approach: Part 1: Causes. In *UAI*, pages 194–202, 2001.

14 Anthony Hunter and Sébastien Konieczny. On the measure of conflicts: Shapley inconsistency values. *Artif. Intell.*, 174(14):1007–1026, 2010.

15 Majd Khalil and Benny Kimelfeld. The complexity of the Shapley value for regular path queries, 2022.

16 Ester Livshits, Leopoldo E. Bertossi, Benny Kimelfeld, and Moshe Sebag. Query games in databases. *SIGMOD Rec.*, 50(1):78–85, 2021.

17 Ester Livshits, Leopoldo E. Bertossi, Benny Kimelfeld, and Moshe Sebag. The Shapley value of tuples in query answering. *Log. Methods Comput. Sci.*, 17(3), 2021.

18 Ester Livshits and Benny Kimelfeld. The Shapley value of inconsistency measures for functional dependencies. In *ICDT*, volume 186 of *LIPIcs*, pages 15:1–15:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.ICDT.2021.15.

19 Scott M Lundberg and Su-In Lee. A unified approach to interpreting model predictions. In *Advances in Neural Information Processing Systems 30*, pages 4765–4774. Curran Associates, Inc., 2017.

20 Artem Lysenko, Irina A. Roznovat, Mansoor Saqi, Alexander Mazein, Christopher J. Rawlings, and Charles Auffray. Representing and querying disease networks using graph databases. *BioData Min.*, 9:23, 2016. doi:10.1186/s13040-016-0102-8.

21 Richard T. B. Ma, Dah-Ming Chiu, John Chi-Shing Lui, Vishal Misra, and Dan Rubenstein. Internet economics: The use of Shapley value for ISP settlement. *IEEE/ACM Trans. Netw.*, 18(3):775–787, 2010. doi:10.1109/TNET.2010.2049205.

22 Wim Martens and Tina Trautner. Evaluation and enumeration problems for regular path queries. In *ICDT*, volume 98 of *LIPIcs*, pages 19:1–19:21. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018. doi:10.4230/LIPIcs.ICDT.2018.19.

23 Wim Martens and Tina Trautner. Dichotomies for evaluating simple regular path queries. *ACM Trans. Database Syst.*, 44(4):16:1–16:46, 2019. doi:10.1145/3331446.

24 Corto Mascle, Christel Baier, Florian Funkev, Simon Jantsch, and Stefan Kiefer. Responsibility and verification: Importance value in temporal logics. In *2021 36th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–14. IEEE, 2021.

25 Alexandra Meliou, Wolfgang Gatterbauer, Katherine F. Moore, and Dan Suciu. The complexity of causality and responsibility for query answers and non-answers. *Proc. VLDB Endow.*, 4(1):34–45, 2010.

26 Alberto O. Mendelzon and Peter T. Wood. Finding regular simple paths in graph databases. *SIAM J. Comput.*, 24(6):1235–1258, 1995. doi:10.1137/S009753979122370X.

27 Stefano Moretti, Fioravante Patrone, and Stefano Bonassi. The class of microarray games and the relevance index for genes. *Top*, 15(2):256–280, 2007.

28 Ramasuri Narayanam and Yadati Narahari. A Shapley value-based approach to discover influential nodes in social networks. *IEEE Trans Autom. Sci. Eng.*, 8(1):130–147, 2011. doi:10.1109/TASE.2010.2052042.

29 Alon Reshef, Benny Kimelfeld, and Ester Livshits. The impact of negation on the complexity of the Shapley value in conjunctive queries. In Dan Suciu, Yufei Tao, and Zhewei Wei, editors, *PODS*, pages 285–297. ACM, 2020. doi:10.1145/3375395.3387664.

30 Gorka Sadowski and Philip Rathle. Fraud detection: Discovering connections with graph databases. *White Paper-Neo Technology-Graphs are Everywhere*, 13, 2014.

31 Babak Salimi, Leopoldo E. Bertossi, Dan Suciu, and Guy Van den Broeck. Quantifying causal effects on query answering in databases. In *TaPP*. USENIX Association, 2016.

32 Lloyd S Shapley. A value for n-person games. In Harold W. Kuhn and Albert W. Tucker, editors, *Contributions to the Theory of Games II*, pages 307–317. Princeton University Press, Princeton, 1953.

33 Seinosuke Toda. PP is as hard as the polynomial-time hierarchy. *SIAM J. Comput.*, 20(5):865–877, 1991.

34 Tjeerd van Campen, Herbert Hamers, Bart Husslage, and Roy Lindelauf. A new approximation method for the Shapley value applied to the WTC 9/11 terrorist attack. *Soc. Netw. Anal. Min.*, 8(1):3:1–3:12, 2018. doi:10.1007/s13278-017-0480-z.

35 Moshe Y. Vardi. The complexity of relational query languages (extended abstract). In *STOC*, pages 137–146. ACM, 1982. doi:10.1145/800070.802186.

36 Mihalis Yannakakis. Graph-theoretic methods in database theory. In *PODS*, pages 230–242, 1990. doi:10.1145/298514.298576.

37 Ningning Yi, Chunfang Li, Xin Feng, and Minyong Shi. Design and implementation of movie recommender system based on graph database. In *WISA*, pages 132–135. IEEE, 2017. doi:10.1109/WISA.2017.34.

38 Byoung-Ha Yoon, Seon-Kyu Kim, and Seon-Young Kim. Use of graph database for the integration of heterogeneous biological data. *Genomics & informatics*, 15(1):19, 2017.

39 Bruno Yun, Srdjan Vesic, Madalina Croitoru, and Pierre Bisquert. Inconsistency measures for repair semantics in OBDA. In *IJCAI*, pages 1977–1983. ijcai.org, 2018.

How Do Centrality Measures Choose the Root of Trees?

Cristian Riveros
Pontificia Universidad Católica de Chile, Santiago, Chile
Millennium Institute for Foundational Research on Data, Santiago, Chile

Jorge Salas
Pontificia Universidad Católica de Chile, Santiago, Chile
Millennium Institute for Foundational Research on Data, Santiago, Chile
University of Edinburgh, UK

Oskar Skibski
University of Warsaw, Poland

Abstract

Centrality measures are widely used to assign importance to graph-structured data. Recently, understanding the principles of such measures has attracted a lot of attention. Given that measures are diverse, this research has usually focused on classes of centrality measures. In this work, we provide a different approach by focusing on classes of graphs instead of classes of measures to understand the underlying principles among various measures. More precisely, we study the class of trees. We observe that even in the case of trees, there is no consensus on which node should be selected as the most central. To analyze the behavior of centrality measures on trees, we introduce a property of *tree rooting* that states a measure selects one or two adjacent nodes as the most important, and the importance decreases from them in all directions. This property is satisfied by closeness centrality but violated by PageRank. We show that, for several centrality measures that root trees, the comparison of adjacent nodes can be inferred by *potential functions* that assess the quality of trees. We use these functions to give fundamental insights on rooting and derive a characterization explaining why some measure root trees. Moreover, we provide an almost linear time algorithm to compute the root of a graph by using potential functions. Finally, using a family of potential functions, we show that many ways of tree rooting exist with desirable properties.

2012 ACM Subject Classification Information systems → Data management systems; Computing methodologies → Network science; Networks → Network structure; Theory of computation → Data structures and algorithms for data management

Keywords and phrases Databases, centrality measures, data centrality, graph theory, tree structures

Digital Object Identifier 10.4230/LIPIcs.ICDT.2023.12

Related Version *Full Version*: https://arxiv.org/abs/2112.13736

Funding C. Riveros and J. Salas were supported by ANID - Millennium Science Initiative Program - Code ICN17_002. Jorge Salas and Oskar Skibski were supported by the Polish National Science Centre Grant No. 2018/31/B/ST6/03201.

1 Introduction

Centrality measures are fundamental tools for network analysis. They are used in a plethora of applications from various areas of science, such as finding people who are more likely to spread a disease in the event of an epidemic [10] or highlighting cancer genes in proteomic data [15]. In all these applications, people use centrality measures to assess graph data and rank which nodes are the most relevant by only using the graph's structure.

Database systems have also incorporated centrality measures to rank nodes in graph data. The first significant use of a centrality measure in a data management problem is by the Google search engine, which successfully introduced PageRank [22] for assessing the importance of a website on the web. Indeed, today graph databases, like Neo4j [1] or TigerGraph [2], natively support centrality algorithms for ranking nodes or analyzing the graph structure. Furthermore, new applications for centrality measures have emerged over knowledge graphs for entity linking [19] or semantic web search engines where ranking results is a core task [14]. More generally, centrality measures play a central role in *network science*, where they are one of the main algorithmic metrics for analyzing graphs [20].

Although more than 200 measures have been proposed in the literature today, it is less clear how one can compare them. Precisely, these measures assess the position of a vertex in the network based on various elements (e.g., degree, number of paths, eigenvector), which makes it hard to compare their properties. Since people propose dozens of new measures every year, choosing a measure for a specific application becomes more and more challenging. In recent years, efforts at understanding and explaining differences between existing measures have intensified, making the foundational aspects of centrality measures an active research topic [5, 26, 24, 27]. Given the diversity of these measures, people have usually followed an axiomatic approach by considering classes of measures like game-theoretical [27] or motif-based [24] measures. Until now, no research has focused on graph classes to classify different centrality measures.

This paper aims to understand how centrality measures operate on trees (i.e., acyclic undirected graphs), as this is a crucial graph family where we can compare different centrality measures and understand their underlying principles. Indeed, even in the case of trees, centrality measures vary significantly as different measures could indicate other vertices as the most central in the same tree. For example, consider a line graph: while the middle vertex (or vertices) is ranked first according to most centrality measures, Google's PageRank puts the second and the second-to-last vertices at the top of the ranking. This simple example shows that not only do different measures select different vertices, but even one measure may select several vertices from different parts of the tree as the most important. Moreover, although PageRank diverges, other centrality measures (e.g., closeness or all-subgraph) coincide following the same underlying principles.

A natural question here is why we should start considering centrality measures over the class of trees. We see several reasons for studying the principal aspects of centrality measures over trees. First, undirected trees are arguably the simplest and non-trivial graph class to study centrality measures. Indeed, trees have a more amenable structure than general graphs, given that, among other properties, every edge is a cutting edge, and there is a unique path between any two nodes. Second, every general result on centrality measures should include trees, and it should answer similar questions to the one studied in this paper. Thus, understanding centrality measures over trees could guide the study of other graph families. Last, trees are a graph structure ubiquitous in computer science and databases. Therefore, studying centrality measures over trees could be helpful for the application of centrality measures in data management and other areas (see Section 8 for some extensions and possible applications of centrality measures over trees).

What principal aspect of centrality measures can we study over the class of trees? In this paper, we focus on understanding one of the main questions over trees: how centrality measures choose the most central nodes in trees, why some measures define a single important node (usually called the root), and why others do not. For answering these questions, the main contributions are as follows:

1. We introduce the *tree rooting property* which states that a measure selects one or two adjacent vertices in a tree as the most important, and the importance decreases from them in all directions. We found that closeness, eccentricity, and all-subgraphs centralities (see Section 2) satisfy this rooting property but often rank different vertices at the top. Instead, measures like degree and betweenness do not satisfy this property. We call the vertex (or two vertices) ranked first the "root" and say that such measures *root trees*.
2. To understand what distinguishes measures that root trees we focus on the question: how to choose which out of two adjacent vertices is more central? We observe that most centrality measures, including all standard ones that root trees, answer this question by comparing subtrees of both vertices. More precisely, we introduce a framework of *potential functions* that assess the quality or "potential" of an arbitrary tree. Now, we show that most centrality measures admit a potential function such that the vertex which subtree has higher potential value is considered more central.
3. We show that if a centrality measure has a potential function, then it roots trees if, and only if, the potential function is *symmetric*. This property means that the potential of a tree is larger than the potential of any proper complete subtree. In particular, the potential function of closeness, eccentricity, and all-subgraphs centrality are symmetric, but the potential functions of degree and betweenness centralities are not; as a result, the latter centralities do not root trees.
4. We use our framework of potential functions to understand better the class of measures that root trees. More specifically, we show three applications of potential functions. Our first application is to study efficient algorithms for computing a root when centrality measures have potential functions and root trees. By exploiting symmetric potential functions, we show that, given a tree T, we can compute the most central vertex in time $\mathcal{O}(|T|\log(|T|))$ whenever one can calculate the potential function locally. Interestingly, this general algorithm works independently of the centrality measures and only depends on the potential function.
5. Our second application of potential functions is to understand desirable properties over tree rooting measures. Although a centrality measure could root trees, it can behave inconsistently. For instance, a rooting centrality measure could choose the root of a tree T close to a leaf when the size of T is even and far from the leaves when the size of T is odd. Therefore, we study when centrality measures consistently root trees through their potential functions. We propose a monotonicity property that imposes additional consistency conditions on how the root is selected and characterizes which potential functions satisfy this property.
6. Our last application shows how to design and build new potential functions that consistently root trees. Specifically, we present infinitely many constructive potential functions that satisfy all properties discussed so far. In particular, the algorithm for finding the root applies to any of them. We believe that this family of measures is interesting in its own right and can be used in several data-driven scenarios.

Related work. Our work could be included into a broad literature that focuses on the analysis of theoretical properties of centrality measures. The classic approach is to analyze standard centrality measures with respect to simple desirable properties. Different properties have been considered over the years [25, 21, 6, 5]. Most of them, however, are very simple (e.g., invariance under automorphism) or not satisfied by most measures. Similarly, in our work, we propose several properties specific for trees. Focusing on trees allows us to identify meaningful properties shared by many measures based on completely different principles.

Another approach is to create a common framework for large classes of centrality measures. In such frameworks, centrality measures are presented as a function of some underlying structure of the graph. Hence, the emphasis is focused on the differences between these functions and their implications for various measures defined under such framework. In this spirit, classes of measures based on distances [12], nodal statistics [4], coalitional game theory [27], subgraphs [24] and vitality functions [26] have been analyzed in the literature. On the opposite, our framework of potential functions can be considered as an approach focused on classes of graphs instead of classes of measures. Yet another approach, less related to our work, is to focus on one or several similar measures and provide their full axiomatic characterization. In this spirit, axiomatizations of PageRank [29], eigenvector [17], beta-measure and degree [28] and many more have been developed in the literature.

There is also a line of research that studies methods for solely choosing one most central vertex in a tree that does not necessarily come from the centrality analysis (see, e.g., [23] for an overview). However, such methods coincide with the top vertices selected by centrality measures that we consider in our work. In particular, the center of a tree coincides with eccentricity, and the median, as well as the centroid, coincides with closeness centrality.

2 Preliminaries

Undirected graphs. In this paper, we consider *finite undirected graphs* of the form $G = (V, E)$ where V is a finite non-empty set and $E \subseteq 2^V$ such that $|e| = 2$ for all $e \in E$. For convenience, given a graph $G = (V, E)$ we use $V(G)$ for indicating the set of *vertices* V and $E(G)$ for the set of *edges* E. We write $N_G(v)$ for the *neighbourhood* of v in G, namely, $N_G(v) \subseteq V(G)$ such that $u \in N_G(v)$ if, and only if, $\{u, v\} \in E(G)$. If this is the case, we say that u and v are *adjacent*.

We say that a graph $G' = (V', E')$ is a *subgraph* of G, denoted $G' \subseteq G$, if $V' \subseteq V$ and $E' \subseteq E$. Note that $\subseteq$ forms a partial order between graphs. For a sequence of graphs $G_1 = (V_1, E_1), \ldots, G_m = (V_m, E_m)$ we denote by $\cup_{i=1}^m G_i$ the graph (V, E) such that $V = \bigcup_{i=1}^m V_i$ and $E = \bigcup_{i=1}^m E_i$. Given a graph G and an edge $e = \{u, v\}$, we write $G + e$ to be the new graph G with the additional edge e, formally, $V(G + e) = V(G) \cup e$ and $E(G + e) = E(G) \cup \{e\}$.

From now on, fix an enumerable set $\mathcal{V}$ of vertices. We define the *set of all graphs* using vertices from $\mathcal{V}$ as $\mathcal{G}$. Further, we define the set $\mathcal{VG}$ as all pairs *vertex-graph* (v, G) such that $v \in V(G)$ and $G \in \mathcal{G}$. In the sequel, for a pair (v, G) we assume that $(v, G) \in \mathcal{VG}$, unless stated otherwise. We say that graphs G_1 and G_2 are *isomorphic*, denoted by $G_1 \cong G_2$, if there exists an bijective function (*isomorphism*) $f : V(G_1) \to V(G_2)$ such that $\{u, v\} \in E(G_1)$ if, and only if, $\{f(u), f(v)\} \in E(G_2)$. We also say that (v_1, G_1) and (v_2, G_2) are isomorphic, denoted by $(v_1, G_1) \cong (v_2, G_2)$, if there exists an isomorphism f between G_1 and G_2 and $f(v_1) = v_2$. Note that $\cong$ is an equivalence relation over $\mathcal{G}$ and over $\mathcal{VG}$.

A *path* in G is a sequence of vertices $\pi = v_0, \ldots, v_n$ such that $\{v_i, v_{i+1}\} \in E$ for every $i < n$, and we say that π is a *path* from v_0 to v_n. We say that π is *simple* if $v_i \neq v_j$ for every $0 \leq i < j < n$. From now on, we usually assume that paths are simple unless stated otherwise. We define the *length* of π as $|\pi| = n$. We agree that v_0 is the *trivial path* of length 0 from v_0 to itself. Given $R, R' \subseteq V(G)$, we say that $\pi = v_0, \ldots, v_n$ is a path from R to R' if $v_0 \in R$, $v_n \in R'$, and $v_i \notin R \cup R'$ for every $i \in [1, n-1]$. We say that a graph G is *connected* if there exists a path between every pair of vertices.

Centrality measures. A *centrality measure*, or just a *measure*, is any function $C : \mathcal{VG} \to \mathbb{R}$ that assigns a score $C(v, G)$ to v depending on its graph G. Here, we use the standard assumption that, the higher the score $C(v, G)$, the more important or "central" is v in G. We also assume that every centrality measure is *closed under isomorphism*, namely, if $(v_1, G_1) \cong (v_2, G_2)$ then $C(v_1, G_1) = C(v_2, G_2)$, which is a standard assumption in the literature [8, 25].

Next, we recall four centrality measures that we will regularly use as examples: *degree*, *closeness*, *eccentricity*, and *all-subgraphs* centralities. During this work, we also mention *betweenness* [11], *decay* [16], *PageRank* [22], and *eigenvector* [7] centralities. Given that we do not use them directly, we refer the reader to the corresponding works for a definition.

Degree centrality. Degree centrality is probably the most straightforward measure. Basically, the bigger the neighborhood of a vertex (i.e., adjacent vertices), the more central it is in the graph. Formally, we define *degree centrality* as follows: $\textsc{Degree}(v, G) = |N_G(v)|$.

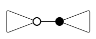
Closeness centrality. For every graph G and vertices $v, u \in V(G)$ we define the *distance* between v and u in G as $d_G(v, u) = |\pi_{v,u}|$ where $\pi_{v,u}$ is a shortest path from v to u in G. Then *closeness centrality* [25] is defined as: $\textsc{Closeness}(v, G) = 1/\sum_{u \in K_v(G)} d_G(v, u)$ where $K_v(G)$ is the connected component of G containing v. Closeness is usually called a geometrical measure because it is based on the distance inside a graph. The intuition behind closeness centrality is simple: the closer a vertex is to everyone in the component (i.e., $\sum_{u \in K_v(G)} d_G(v, u)$ is small) the more important it is.

Eccentricity centrality. Another important notion in graph theory is radius. In simple words, we can define the center of a graph G as the vertex that minimizes the maximum distance in G. Formally, v is the center of G if it minimizes $\max_{u \neq v \in V(G)} d_G(v, u)$. Then, the radius of G is defined as the maximum distance from the center. Now, *eccentricity measure* [13] is precisely the one centrality that selects the center of a graph as the most important vertex, defined as $\textsc{Eccentricity}(v, G) = 1/\max_{u \in V(G)} d_G(v, u)$.

All-subgraphs centrality. Given a graph $G = (V, E)$ and a vertex $v \in V$, we denote by $\mathcal{A}(v, G)$ the set of all connected subgraphs of G that contain v, formally, $\mathcal{A}(v, G) = \{S \subseteq G \mid v \in V(S) \text{ and } S \text{ is connected}\}$. Then *all-subgraphs centrality* [24] of v in G is defined as: $\textsc{AllSubgraphs}(v, G) = \log_2 |\mathcal{A}(v, G)|$. All-subgraphs centrality was recently proposed in [24], proving that it satisfies several desirable properties as a centrality measure. Intuitively, it says that a vertex will be more relevant in a graph if it has more connected subgraphs surrounding it.

Undirected Trees. This paper is about *undirected trees* (or just trees), so we use some special notation for them. Specifically, we say that a graph T is a *tree* if it is connected and for every $u, v \in V(T)$ there exists a unique path that connects u with v in T. We usually use T to denote a tree. Further, we say that $v \in V(T)$ is a *leaf* of T if $|N_T(v)| = 1$. If v is a leaf and $N_T(v) = \{u\}$, then we say that u is the *parent* of v. Note that trees are a special class of undirected graphs, and all previous definitions apply. In particular, we can use and apply centrality measures over trees.

We say that T' is a *subtree* of T if $T' \subseteq T$ and T' is a tree. We also say that T' is a *complete subtree* of T if $T' \subseteq T$ and there is at most one vertex in $V(T')$ connected to some vertex in $V(T) \setminus V(T')$, namely, $|\{v \in V(T') \mid \exists u \in V(T) \setminus V(T'). \{u, v\} \in E(T)\}| \leq 1$.

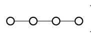
The following notation will be useful in the paper to decompose trees. Given a tree T and two adjacent vertices $u, v \in V(T)$, we denote by $T_{u,v}$ the maximum subtree of T that contains u and not v. For example, if $T =$ , then $T_{\circ,\bullet} =$ and $T_{\bullet,\circ} =$.

Finally, we consider some special trees to give examples or show some properties of centrality measures. For a vertex v we define $G_v = (\{v\}, \emptyset)$, and for an edge e we define $G_e = (e, \{e\})$, namely, the graphs with *one isolated vertex* v or *one isolated edge* e, respectively. Similarly, for any $n \geq 1$ we write L_n for the *line* with n vertices where $V(L_n) = \{0, \ldots, n-1\}$ and $E(L_n) = \{\{i, i+1\} \mid 0 \leq i < n-1\}$.

3 Tree rooting centrality measures

We start by giving a formal definition of when a centrality measure C roots trees. For this, let C be a centrality measure. We define the set $\textsc{Max}_C(T)$ to be the set of most central vertices with respect to C in a tree T, namely, $v \in \textsc{Max}_C(T)$ iff $C(u,T) \leq C(v,T)$ for every $u \in V(T)$.

▶ **Definition 1.** *We say that a centrality measure C* roots trees *if for every tree T, the set of most central vertices $\textsc{Max}_C(T)$ consists of one vertex or two adjacent vertices. Moreover, for every $u \notin \textsc{Max}_C(T)$ if $u_0 u_1 \ldots u_n$ is the unique path from $\textsc{Max}_C(T)$ to u, then $C(u_i, T) > C(u_{i+1}, T)$ for every $i \in [0, n-1]$.*

In the following, if a centrality measure roots trees, we also say that it satisfies the *tree rooting property* (i.e., Definition 1). We can motivate this property as follows. We treat vertices with the highest centrality as "roots". For the first part of the definition, we assume there is a single source of importance – one vertex or two adjacent vertices. We allow two adjacent vertices to be the roots, as in some graphs, due to their symmetrical structure, it is impossible to indicate one most central vertex. This is, for example, the case of a line with an even number of vertices (e.g., ). In such a scenario the edge between both can be considered the real root of the tree. For the second part, we assume that the centrality should decrease from the root through branches. This restriction aligns with the intuition that the closer a vertex is to the root, the more central it is.

We continue by giving examples of measures that root trees, and some others that do not.

▶ **Example 2.** Closeness, eccentricity, and all-subgraphs all root trees. It was already noticed [18] that over trees, closeness and eccentricity define at most two maximum vertices, and both are connected. On the other hand, it is more subtle to show that all-subgraphs centrality roots trees. This fact, however, will follow from the framework developed in Section 4. As an illustration, in Figure 1 we show how closeness, eccentricity, and all-subgraphs behave over the same tree. One can verify that each measure declares one vertex with the maximum centrality (this vertex is marked with a black star). Moreover, the centrality decreases through the branches (the lower the centrality the whiter the color). It is interesting that, although the three measures root trees, they declare different vertices as the most central.

▶ **Example 3.** One can easily check that degree centrality does not root trees. Indeed, the last tree at Figure 1 is an example where degree centrality declares two maximum vertices, and they are not adjacent. Indeed, all measures presented in Section 2, except closeness, eccentricity, and all-subgraphs, do not root trees. For all of them, the last tree at Figure 1 is a counterexample where they violate the tree rooting property. At Figure 2, we show a table that summarize which centrality measures considered in this paper root trees.

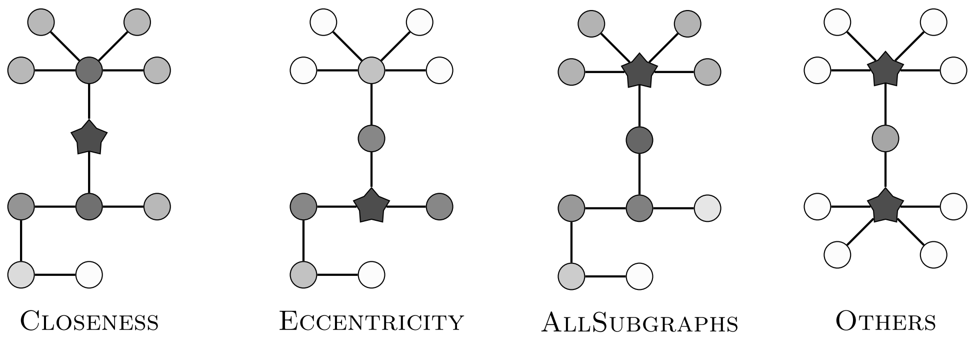


Figure 1 The first three trees exemplify how closeness, eccentricity, and all-subgraphs centralities root trees. We mark the most central vertex with a black star. The colors show how the centrality value decreases through the branches (i.e., whiter vertices are less central). The fourth tree is a counter-example that shows why other centralities do not root trees.

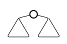
An important consequence of assigning a root to a tree is that each vertex has a parent (except for the root). Here, the unique path from the root to the vertex defines its parent. Another possibility would be to use the centrality measure to find the neighbour with higher centrality, and declare it as the parent. We capture this intuition in the following property.

▶ **Definition 4.** *We say that a measure C satisfies the* at-most-one-parent *property if for every tree T and $v \in V(T)$ there exists at most one neighbour $u \in N_T(v)$ such that $C(v,T) \leq C(u,T)$.*

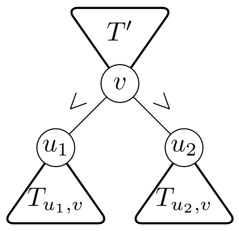
Clearly, if a centrality measure C roots trees, then it also satisfies at-most-one-parent. The other direction is also true, providing an alternative characterization for tree rooting.

▶ **Proposition 5.** *A centrality measure roots trees if, and only if, it satisfies the at-most-one-parent property.*

Notice that tree rooting is a global property over a tree but, instead, at-most-one-parent is a local property of the neighbourhoods of a tree, which is easier to prove for a centrality measure. Indeed, in the next section we use this alternative definition to prove our main characterization for tree rooting.

In some trees the root is uniquely characterized solely by the tree rooting property. This happens because every centrality must be closed under isomorphism which implies that isomorphic vertices have the same centrality. For example, one can verify that for the line L_n the roots must be the set $\{\lfloor\frac{n-1}{2}\rfloor, \lceil\frac{n-1}{2}\rceil\}$. Indeed, every vertex $i \leq \frac{n-1}{2}$ is isomorphic with the vertex $n-1-i$ in L_n. Then, if i is the root, then $n-1-i$ must be the root as well. Given that vertices i and $n-1-i$ are not connected if $i < \frac{n-1}{2}$, we get that every centrality that roots trees must declare $\{\lfloor\frac{n-1}{2}\rfloor, \lceil\frac{n-1}{2}\rceil\}$ as the root. This idea can be extended to all symmetric trees: if T is a tree with a vertex $\circ$ that connects two isomorphic subtrees (i.e., $T =$ ), then $\circ$ must be the root of T. We generalize this property as follows.

▶ **Definition 6.** *We say that a centrality measure C is* symmetric over trees *if for every tree T, vertex v, and different neighbors $u_1, u_2 \in N_T(v)$ such that $(u_1, T_{u_1,v}) \cong (u_2, T_{u_2,v})$, then $C(u_1,T) < C(v,T)$ and $C(u_2,T) < C(v,T)$.*



Measures	Can root?	Potential function?
Closeness	Yes	Yes
Eccentricity	Yes	Yes
All-Subgraphs	Yes	Yes
Degree	No	Yes
Betweenness	No	Yes
Decay	No	Yes
PageRank	No	?
EigenVector	No	?

Figure 2 At the left, a graphic illustration of the symmetry property over a tree T. Subtrees $T_{u_1,v}$ and $T_{u_2,v}$ are isomorphic, and subtree T' represents the rest of T hanging from v. At the right, a table summarizes which centrality measures root trees and which one admits a potential function.

Figure 2 (left) is a graphical representation of the symmetry property. It generalizes the previously discussed intuition by considering any pair of isomorphic subtrees in a (not necessarily symmetric) tree. Interestingly, every centrality measure that roots trees must be symmetric over trees.

▶ **Proposition 7.** *If a centrality measure C roots trees, then C is symmetric over trees.*

A symmetric centrality measure may not root trees. For example, a centrality measure could root trees with non-trivial automorphisms but not root trees when there is none. Despite this, symmetry property is crucial for finding a characterization for tree rooting, as we will show in the next section.

4 Potential functions

What have in common closeness, eccentricity, and all-subgraphs centralities? What is the fundamental property so they can root trees? A crucial ingredient for understanding the connection between these measures is what we call a *potential function* for a centrality measure.

▶ **Definition 8.** *Given a centrality measure C, we say that $f : \mathcal{VG} \to \mathbb{R}$ is a potential function for C if f is closed under isomorphism on $\mathcal{VG}$ and, for every tree T and every adjacent vertices $u, v \in V(T)$, it holds that $C(u,T) \leq C(v,T)$ if, and only if, $f(u, T_{u,v}) \leq f(v, T_{v,u})$.*

A potential function is a function that measures the "potential" of every *rooted* tree, i.e., a tree with one node selected and the assessment depends on the selection. Now, a centrality measure admits some potential function if the comparison between two adjacent vertices is determined by the potential of their corresponding subtrees. Interestingly, in the following examples we show that several centrality measures admit a potential function.

▶ **Example 9.** Degree, closeness, eccentricity, all-subgraphs centralities have the following potential functions:

$$\begin{array}{lcll}
f_{\mathrm{d}}(v,T) & := & |N_T(v)| & (\textit{degree}) \\
f_{\mathrm{c}}(v,T) & := & |V(T)| & (\textit{closeness}) \\
f_{\mathrm{e}}(v,T) & := & \max_{u \in V(T)} d_T(v,u) & (\textit{eccentricity}) \\
f_{\mathrm{a}}(v,T) & := & |\mathcal{A}(v,T)| & (\textit{all-subgraphs})
\end{array}$$

Let us verify that each function above is a potential function for its corresponding measure. Take an arbitrary tree T and two adjacent vertices u and v. It is straightforward to check that f_{d} is a potential function for degree centrality. Indeed, if u has a smaller degree in its subtree (i.e., without considering the common edge), then it also has a smaller centrality.

For closeness centrality, the potential function f_{c} is simply the number of vertices in a tree. To see this, note that vertex u has a distance smaller by one than v to all vertices from $T_{u,v}$; analogously, vertex v has a distance smaller by one than u to all vertices from $T_{v,u}$. As a result, out of both vertices, the one with the larger subtree has the smaller sum of distances which results in the higher closeness centrality.

For eccentricity the potential function f_{e} is the height of a tree, i.e., the distance to the farthest vertex. This is because if subtree $T_{u,v}$ is higher than $T_{v,u}$, then vertex u has smaller distance to the farthest vertex in T which results in the higher eccentricity. Interestingly, f_{e} is an inverse of eccentricity.

Finally, for all-subgraphs centrality the potential function f_{a} is the number of subgraphs that contain vertex v. The reason is that we have $|\mathcal{A}(u,T)| = |\mathcal{A}(u,T_{u,v})| + |\mathcal{A}(u,T_{u,v})| \cdot |\mathcal{A}(v,T_{v,u})|$ and, symmetrically, $|\mathcal{A}(v,T)| = |\mathcal{A}(v,T_{v,u})| + |\mathcal{A}(u,T_{u,v})| \cdot |\mathcal{A}(v,T_{v,u})|$. This implies that if u has more subgraphs in $T_{u,v}$ than v in $T_{v,u}$, then it also has higher all-subgraphs centrality. Hence, as in the case of degree centrality, the potential function coincides with the centrality itself.

One can also show that betweenness and decay centralities have a potential function. As we will see later, there are centrality measures that do not have potential functions. For the particular case of PageRank and EigenVector it is not clear whether they admit a potential function. The table at Figure 2 (right) summarizes which centrality measures have a potential function.

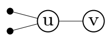
A potential function determines the centrality order between two adjacent vertices, but it does not imply the relation between non-adjacent vertices. Although this information is weaker than the centrality measure itself, it is exactly what we need to understand the centrality measures that root trees. Precisely, which measures with potential functions root trees? To answer this question, we first need to capture the symmetry property through the lens of potential functions.

▶ **Lemma 10.** *Let C be a centrality measure and f a potential function for C. Then C is symmetric over trees if, and only if, for every tree T and every pair of adjacent vertices $u, v \in V(T)$, it holds that $f(u, T_{u,v}) < f(v, T)$.*

By the previous result, we will call potential functions with this property *symmetric*. Next, we show that symmetric potential functions characterize the tree rooting property.

▶ **Theorem 11.** *Let C be a centrality measure that admits a potential function f. Then C roots trees if, and only if, f is symmetric.*

▶ **Example 12.** Continuing with Example 9 we can check that the potential functions f_{c}, f_{e}, and f_{a} for closeness, eccentricity, and all-subgraphs, respectively, are symmetric. Indeed, for all these functions a subtree always has less potential than the whole tree. For this reason, $f_{\mathrm{x}}(u, T_{u,v}) < f_{\mathrm{x}}(v, T)$ for $\mathrm{x} \in \{\mathrm{c}, \mathrm{e}, \mathrm{a}\}$ since $T_{u,v}$ is a subgraph of T. By Theorem 11, this proves that closeness, eccentricity, and all-subgraphs root trees.

In turn, the potential functions of degree centrality (as well as betweenness and decay centralities) are not symmetric and, therefore, do not root trees. For example, for degree centrality we can take $T =$ 

, and check that $f_{\mathrm{d}}(u, T_{u,v}) > f_{\mathrm{d}}(v, T)$. Therefore, f_{d} is not symmetric.

We showed that all standard centrality measures that root trees can be defined through potential functions. The natural question is: is it true for all measures that root trees? In the following result, we show that this is not the case and there exists a measure that roots a tree, but does not have a potential function.

▶ **Proposition 13.** *There exists a centrality measure that roots trees but does not admit a potential function.*

Potential functions and Theorem 11 explain why some measures root trees and others do not. In the following sections, we use this framework to further understand the tree rooting centrality measures in terms of algorithms, consistency, and the design of new measures.

5 An algorithm to find the root

Even though not every tree rooting measure has a potential function, having one gives us some essential properties that we can exploit. In particular, having a symmetric potential function implies finding the root of any tree in $\mathcal{O}(n\log(n))$-time under some assumptions on the efficiency to compute the potential function. Notice that the naive approach of computing the centrality for each vertex separately and then choosing the one with the highest centrality runs in quadratic time (i.e., by assuming that computing the centrality of a single vertex takes linear time). Instead, the algorithm presented here runs in $\mathcal{O}(n\log(n))$ for every centrality measure that admits a (locally-computable) symmetric potential function.

The main intuition behind this algorithm is based on the following property satisfied by centrality measures with a symmetric potential function.

▶ **Proposition 14.** *Let C be a centrality measure that has a symmetric potential function f. Let T be any tree, $w_1, w_n \in V(T)$, and $w_1w_2...w_n$ be the unique path connecting w_1 to w_n in T. Whenever $f(w_1, T_{w_1,w_2}) \leq f(w_n, T_{w_n,w_{n-1}})$ then $C(w_1, T) \leq C(w_2, T)$.*

In other words, Proposition 14 says that if the potential of the subtree hanging from w_1 is less than the potential of the subtree hanging from w_n, then a root should be closer to the adjacent vertex of w_1 that is towards the direction of w_n. This result gives us a way to traverse a tree, starting from the leaves and going up until we find the root. More specifically, starting from the leaves, by Proposition 14 we can compare the potential of two opposite complete subtrees. Then, vertices with higher potential in their subtree indicate the direction of higher centrality. When we finally reach two connected vertices, the vertex with higher (or equal) potential is a root.

Algorithm 1 implements the above intuition based on Proposition 14. It receives as input a tree T and a symmetric potential function f, and outputs a root of T with respect to f. For implementing Algorithm 1 we need two data structures, denoted by H and Q. The first data structure H is a *key-value map* (i.e., a Hash-table), where a key can be any vertex $v \in V(T)$ and its value is a subset of $N_T(v)$. We denote the value of v (i.e., a key) in H by $H[v]$. By some abuse of notation, when $H[v]$ is a single vertex, we write $u \leftarrow H[v]$ to retrieve and store this vertex in u. The second data structure Q is a *priority-Queue*. For $v \in V(T)$ and $p \in \mathbb{R}$ we write $Q.\texttt{insert}(v, p)$ to insert v in Q with priority p. We also write $v \leftarrow Q.\texttt{pull}()$ to remove the vertex with the lowest priority from Q, and store it in v. For both structures, these operations can be implemented in $\mathcal{O}(\log(n))$-time where n is the number of inserted objects [9].

Algorithm 1 starts by initializing Q as empty and H with all key-value pairs $(v, N_T(v))$ (lines 2-3). Then, it runs over all leaves v of T and inserts it into Q with priority $f(v, G_v)$ where G_v is the tree with an isolated vertex v (lines 4-5). As we already mentioned, the

Algorithm 1 Find a root given a tree.

```
Input: A non-trivial tree $T$ and a symmetric potential function $f$.
Output: The most central vertex of $T$ according to $f$.
1  Function Find-a-root($T, f$):
2      $Q \leftarrow$ Empty-queue
3      $H \leftarrow \{(v, N_T(v)) \mid v \in V(T)\}$
4      foreach $v$ leaf of $T$ do
5          $Q$.insert($v, f(v, G_v)$)
6      while $Q.size() > 1$
7          $v \leftarrow Q$.pull()
8          $u \leftarrow H[v]$
9          $H[u] \leftarrow H[u] \setminus \{v\}$
10         if $|H[u]| = 1$ then
11             $w \leftarrow H[u]$
12             $Q$.insert($u, f(u, T_{u,w})$)
13     return $Q$.pull()
```

intuition is to start from all leaves v and use its potential (as a single vertex) for comparing it with other vertices. Then we loop while the number of elements in Q is greater than 1. Recall that any non-trivial tree has at least two leaves, and therefore the algorithm reaches line 6 with $Q.\mathtt{size}() \geq 2$ for the first time. Instead, if T is trivial, we return the single vertex directly in line 13.

We remove the vertex with the lowest priority from Q in each iteration and store it in v (line 7). This step discards v as a possible root (by Proposition 14) and moves towards its "parent" represented by $u \leftarrow H[v]$ (line 8). Given that we discarded v, we remove v as a child of u, where $H[u]$ contains the current children of u (line 9). Indeed, when $|H[u]| = 1$ (line 10) this means that we have reached u, its parent is $w \leftarrow H[u]$ (line 11) and its complete subtree $T_{u,w}$ hanging from u must be evaluated with f, and inserted in Q (line 12). An important invariant during the while-loop is that any vertex v in Q satisfies $|H[v]| = 1$ (except at the end of the last iteration). Conceptually, if $H[v] = \{w\}$, this invariant means that w is the parent of v and we are using the potential of the subtree $(v, T_{v,w})$ for comparing v with other vertices. Then when v is the vertex with the lowest priority on Q, it means that other vertices beat it, and a root must be towards its parent.

Finally, when there is only one vertex left in Q, it beats all other vertices, and it should be one of the roots. It is necessary to mention that if T has two roots, we could also output the second root by slightly modifying the algorithm.

Regarding time complexity, the reader can check that the for- and while-loops take linear time on $|T|$. Each operation over H and Q take at most $\log(|T|)$ steps, and overall it sum up to $\mathcal{O}(|T| \log(|T|))$ if computing f takes constant time.

Of course, the previous assumption is not always true, given that f can be any symmetric potential function. To solve this, we say that f is *locally-computable* if, for every T and $u \in V(T)$, $f(u, T)$ can be computed in $\mathcal{O}(k)$-time from the values of its k-neighbors, namely, from $f(u_1, T_{u_1,v}), \ldots, f(u_k, T_{u_k,v})$ where $N_T(u) = \{u_1, \ldots, u_k\}$. Note that by book-keeping the values $f(u_1, T_{u_1,v}), \ldots, f(u_k, T_{u_k,v})$ of the neighbors of u, we can compute $f(u, T_{u,w})$ (line 12) in $\mathcal{O}(|N_T(u)|)$-time. If we sum this extra time over all vertices, it only adds $\mathcal{O}(|T|)$-steps to the total running time of the algorithm.

▶ **Proposition 15.** *Given a tree T and a symmetric potential function f, Algorithm 1 returns a root of T with respect to f. Moreover, if f is locally-computable, the algorithms runs in $\mathcal{O}(|T| \cdot \log(|T|))$-time.*

The reader can easily check that the potential functions of closeness, eccentricity, and all-subgraphs are locally-computable (see also Section 7), and then Algorithm 1 can be used for any of these measures to find the root of any tree in $\mathcal{O}(|T| \cdot \log(|T|))$.

6 Consistent rooting

The tree rooting property fixes the "shape" of a centrality measure in every possible tree. However, it does not impose any relation between roots in different trees. As a result, even a small change (e.g., adding a leaf) may move the root arbitrarily since there might be no relation between the roots in a tree and the altered tree. To give an example, a centrality measure may be defined in one way for trees with odd number of vertices, but in a completely different way for trees with even number of vertices.

To this end, we propose a notion of *consistency*. Consistency states that if we add a leaf to the tree, then the root may move only in its direction.

▶ **Definition 16.** *We say that a centrality measure C* consistently roots trees *if it roots trees and for every tree T and vertices $u, v \in V(T)$, $w \notin V(T)$ such that $u \in \text{Max}_C(T)$ it holds $\text{Max}_C(T + \{v, w\}) \subseteq \pi_{u,w} \cup \text{Max}_C(T)$, where $\pi_{u,w}$ is the path between u and w in $T + \{v, w\}$.*

We can verify that closeness, eccentricity, and all-subgraphs centralities all consistently root trees.

Consistency is a property of measures that root trees. However, it can be also interpreted using a natural property for arbitrary centrality measures that we call *monotonicity*. Monotonicity states that if vertex v has a higher (or equal) centrality than its neighbour u in a tree, then this fact will not change if we add a leaf on the side of vertex v.

▶ **Definition 17.** *We say that a centrality measure C is* monotonic *if for every tree T, vertices $v, u, w \in T$ such that $\{v, u\} \in E(T)$, $w \in T_{u,v}$ and vertex $w' \notin V(T)$ if $C(v, T) < C(u, T)$, then $C(v, T + \{w, w'\}) < C(u, T + \{w, w'\})$.*

Monotonicity is in fact a general property satisfied by many centrality measures, including all geometric centralities such as closeness and decay. The following result ties both concepts: monotonicity and consistency of rooting.

▶ **Proposition 18.** *Let C be a centrality measure that roots trees. Then C consistently roots trees if, and only if, it is monotonic.*

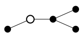
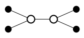
Let us turn our attention to the relation between tree rooting and potential functions. If a centrality that roots trees admits a potential function, then it must be consistent to some extent. However, as it turns out, it might not consistently root trees, as we show in the following counterexample.

▶ **Example 19.** Consider the following ad-hoc centrality measure:

$$C(v, T) = \text{Eccentricity}(v, T) - (1/|T|^2) \cdot \text{Closeness}(v, T).$$

Intuitively, if two vertices in a tree have different eccentricity, then their eccentricity differs by more than $1/|T|^2$. Also, $\text{Closeness}(v, T) \in (0, 1]$. Hence, we have $C(u, T) < C(v, T)$ if, and only if, u has a smaller eccentricity than v or equal eccentricity, but higher closeness.

It is easy to verify that the following potential function corresponds to C: $f(v, T) = h(v, T) - 1/|T|$, where $h(v, T)$ is the distance from v to the farthest vertex in T.

To show that consistency is violated consider trees  and  (vertices with the highest centrality are marked with white color). Consistency states that in the second tree the root should stay on the left-hand side which is not the case here.

To characterize which of the centrality measures with potential function root trees consistently, we look at the restriction that monotonicity imposes on the potential function.

▶ **Proposition 20.** *Let C be a centrality measure that admits a potential function f. Then C is monotonic iff for every tree T, subtree T' of T, and $v \in V(T')$ it holds $f(v, T) \geq f(v, T')$.*

In particular, the potential function from Example 19 violates this condition, as adding vertices to a tree without increasing its height decreases the value of the potential function.

Now we can summarize rooting results and consistency regarding potential functions as one of our principal theorems.

▶ **Theorem 21.** *Let C be a centrality measure that admits a potential function f. Then C is monotonic and symmetric if, and only if, for every tree T, proper subtree T' of T and vertices $v \in V(T)$, $u \in V(T')$ it holds $f(v, T) \geq f(u, T')$ and $f(v, T) > f(u, T')$ if $u \neq v$.*

7 Families of potential functions

In this section, we apply the previous results by showing how to design potential functions that consistently root trees. Specifically, using the following results, we can derive an infinite family of potential functions. This family shows infinite ways to root trees with good characteristics, namely, that are consistent and computable in $\mathcal{O}(n \log(n))$ time. In the following, we recall some standard definitions of monoids, to then define potential functions through them.

A *monoid* (over $\mathbb{R}$) is a triple $(\mathbb{M}, *, \mathbb{1})$ where $\mathbb{M} \subseteq \mathbb{R}$, $*$ is a binary operation over $\mathbb{M}$, $*$ is associative, and $\mathbb{1} \in \mathbb{M}$ is the identity of $*$ (i.e., $r * \mathbb{1} = \mathbb{1} * r = r$). We further assume that monoids are commutative, namely, $*$ is commutative. Examples of (commutative) monoids are $(\mathbb{R}_{\geq 0}, +, 0)$ and $(\mathbb{R}_{\geq 1}, \times, 1)$, where we use $\mathbb{R}_{\geq c}$ for all reals greater or equal than c. For the sake of presentation, in the following we will usually refer the monoid

▶ **Definition 22.** *Given a monoid $(\mathbb{M}, *, \mathbb{1})$ and $\ell : \mathbb{M} \to \mathbb{M}$, we define the potential function $f_{*,\ell}$ recursively as follows:*

1. *For a vertex v, we define $f_{*,\ell}(v, G_v) = \mathbb{1}$, where G_v is the tree with an isolated vertex v.*
2. *For a tree T and a leaf $v \in V(T)$ hanging from its parent $u \in V(T)$, we define $f_{*,\ell}(v, T) = \ell(f_{*,\ell}(u, T_{u,v}))$. In other words, we apply ℓ to the potential function of the subtree rooted at u. We call ℓ the* leaf-function *of $f_{*,\ell}$.*
3. *Given two trees T_1 and T_2 with $V(T_1) \cap V(T_2) = \{v\}$, we define $f_{*,\ell}(v, T_1 \cup T_2) = f_{*,\ell}(v, T_1) * f_{*,\ell}(v, T_2)$.*

A potential function f is constructive *if there exists a monoid $*$ and a leaf-function ℓ such that $f = f_{*,\ell}$.*

Notice that $f_{*,\ell}(v, T)$ is uniquely determined by the three cases above. Specifically, suppose that $u_1, \ldots, u_k \in N_T(v)$ are the neighbors of v on T. Then we can decompose T by considering all subtrees $T_{u_i,v} + \{u_i, v\}$ and compute $f_{*,\ell}(v, T)$ recursively:

$$f_{*,\ell}(v, T) = \ell\big(f_{*,\ell}(u_1, T_{u_1,v})\big) * \ldots * \ell\big(f_{*,\ell}(u_k, T_{u_k,v})\big)$$

until we reach a single vertex. Furthermore, $f_{*,\ell}$ is closed under isomorphism over $\mathcal{VG}$ given that $*$ is associative and commutative. Thus, we conclude that $f_{*,\ell}$ is well-defined and could work as a potential function. In addition, $f_{*,\ell}$ is locally-computable since $f_{*,\ell}(v,T)$ can be computed from its k-neighbors.

▶ **Example 23.** All potential functions presented in Example 9 are constructive by considering the following monoids and leaf-functions:

$(\mathbb{R}_{\geq 0}, +, 0)$	$\ell(x) = 1$	(*degree*)
$(\mathbb{R}_{\geq 1}, a+b-1, 1)$	$\ell(x) = x+1$	(*closeness*)
$(\mathbb{R}_{\geq 0}, \max, 0)$	$\ell(x) = x+1$	(*eccentricity*)
$(\mathbb{R}_{\geq 1}, \times, 1)$	$\ell(x) = x+1$	(*all-subgraphs*)

It is easy to check that each monoid and leaf-function defines the corresponding potential function of the above measures.

One advantage of the previous definition is that it shows a way for constructing potential functions. Moreover, we can study which properties are necessary over $*$ and ℓ to guarantee that $f_{*,\ell}$ consistently root trees. Towards this goal, we recall some standard definitions for monoids and functions. A function f is called *monotonic* if $x \leq y$, implies $f(x) \leq f(y)$ for every x, y. A monoid $(\mathbb{M}, *, \mathbb{1})$ is called *ordered* if $x \leq y$, implies $x * z \leq y * z$ for every $x, y, z \in \mathbb{M}$. Further, it is called *positively ordered* if in addition $\mathbb{1} \leq x$ for every $x \in \mathbb{M}$.

▶ **Lemma 24.** *Let $(\mathbb{M}, *, \mathbb{1})$ be a monoid and $\ell : \mathbb{M} \to \mathbb{M}$ a leaf-function. The potential function $f_{*,\ell}$ consistently roots trees whenever (1) $x < \ell(x)$ for every x, (2) ℓ is monotonic, and (3) $(\mathbb{M}, *, \mathbb{1})$ is positively ordered.*

For example, the monoids and leaf-functions of closeness, eccentricity, and all-subgraphs (see Example 23), satisfy properties (1) to (3) and, as we know, they consistently root trees. On the other hand, degree's leaf-function does not satisfy (1), and therefore, it does not root trees.

Lemma 24 shows sufficient conditions over $*$ and ℓ to consistently root trees. To get a necessary condition, we need to add some technical restrictions, and to slightly weaken conditions (2) and (3). Towards this goal, let $\text{Range}_{*,\ell}$ be the range of $f_{*,\ell}$. Define $\bar{*}$ and $\bar{\ell}$ to be the monoid $(\mathbb{M}, *, \mathbb{1})$ and function ℓ restricted to $\text{Range}_{*,\ell}$. For two values x and y, we say that x is a *subtree-value* of y if there exist T and T' such that T is a subtree of T', $f_{*,\ell}(u,T) = x$, and $f_{*,\ell}(u,T') = y$ for some $u \in V(T)$. Then we say that $\bar{\ell}$ is *monotonic over subtrees* if $x \leq y$ and x is a subtree-value of y implies that $\bar{\ell}(x) \leq \bar{\ell}(y)$. Similarly, we say that $\bar{*}$ is *positively ordered over subtrees*, if $x \leq y$ and x is a subtree-value of y, then $x * z \leq y * z$ for every $z \in \text{Range}_{*,\ell}$, and $\mathbb{1} \leq x$ for every $x \in \text{Range}_{*,\ell}$.

▶ **Theorem 25.** *Let $(\mathbb{M}, *, \mathbb{1})$ be a monoid and $\ell : \mathbb{M} \to \mathbb{M}$ a leaf-function. The potential function $f_{*,\ell}$ consistently roots trees if, and only if, (1) $x < \bar{\ell}(x)$ for every $x \in \text{Range}_{*,\ell}$, (2) $\bar{\ell}$ is monotonic over subtrees, and (3) $\bar{*}$ is positively ordered over subtrees.*

Theorem 25 and, specifically, Lemma 24 give the ingredients to design potential functions that consistently root trees and, further, we have an algorithm to find the root in $\mathcal{O}(n\log(n))$. For instance, take a triple $(a,b,c) \in \mathbb{R}^3$. Then define the monoid $(\mathbb{R}_{\geq c}, *_c, c)$ and leaf-function $\ell_{a,b}$ such that: $x *_c y := \frac{x \cdot y}{c}$ and $\ell_{a,b}(x) := a \cdot x + b$. For example, if we consider $a = b = c = 1$, we get the monoid and leaf-function for the potential function of all-subgraph centrality (see Example 23). Interestingly, one can verify that, if $a \geq 1$, $b > 0$ and $c > 0$, then $*_c$ is a monoid. Moreover, $*_c$ and $\ell_{a,b}$ satisfy properties (1) to (3) of Lemma 24 and we get the following result.

▶ **Proposition 26.** *For every $(a, b, c) \in \mathbb{R}^3$ with $a \geq 1$, $b > 0$, and $c > 0$, the potential function $f_{*_c,\ell_{a,b}}$ consistently root trees.*

Finally, we want to know if we can get different roots for different values (a, b, c). In other words, is it the case that for every different triples (a, b, c) and (a', b', c') there exists a tree T such that the root of T according to $f_{*_c,\ell_{a,b}}$ is different to one chosen by $f_{*_{c'},\ell_{a',b'}}$? The next result shows that $\{f_{*_c,\ell_{a,b}} \mid c \geq 1, b > 0, c > 0\}$ is indeed an infinity family of different potential functions for tree rooting.

▶ **Proposition 27.** *There exists an infinite set $S \subseteq \mathbb{R}^3$ such that for every $(a, b, c), (a', b', c') \in S$, there exists a tree T where the roots of T according to $f_{*_c,\ell_{a,b}}$ are not the same as roots of T according to $f_{*_{c'},\ell_{b',c'}}$.*

8 Discussion

We end the paper by discussing some extensions and applications.

Extensions to other classes of graphs. An obvious question is whether one can generalize the know-how acquired on trees to other classes of graphs. We agree that further research is needed to extend potential functions to new graph families. Nevertheless, we see some exciting directions in which our work can be extended. In particular, the idea of potential functions extends to arbitrary graphs by considering the endpoints of a bridge (i.e., a cut edge) instead of adjacent vertices of a tree. More in detail, let G be a connected graph and $\{u, v\}$ be an edge such that G_u, G_v are two connected components of $G - \{u, v\}$ that contain u, v, respectively. We say that $f : \mathcal{VG} \to \mathbb{R}$ is a *graph* potential function for C if for every such graph G it holds $C(u, G) \leq C(v, G)$ if, and only if, $f(u, G_u) \leq f(v, G_v)$. This property generalizes (tree) potential functions, as every two adjacent nodes in a tree form a bridge. As it turns out, all centrality measures listed on Figure 2 that have (tree) potential functions (degree, closeness, betweenness, eccentricity, all-subgraphs, decay) have identical graph potential functions. For example, $\textsc{Eccentricity}(v, G) \leq \textsc{Eccentricity}(u, G)$ if and only if $f_e(v, G_v) \leq f_e(u, G_u)$ for f_e defined in Example 9. Interestingly, some centrality measures have identical potential functions on trees but different ones on general graphs (for example, closeness and random-walk closeness centralities).

Applications. In this work, we focused on the foundational aspects of understanding centrality measures over trees, and we left for future work the application in the context of data management. Given the axiomatic approach of our work and given that tree structures are ubiquitous in data management, we believe that potential functions and their implications on rooting trees could find several exciting applications. In the following, we present some possible applications of this work in data management scenarios.

In conjunctive query answering, the class of acyclic queries is of particular interest, given that each query has a join tree that permits efficient evaluation in linear time on data complexity [3]. For this, the so-called Yannakakis algorithm [30] performs a bottom-up traversal of the join tree for filtering the tuples that will not be part of the output. In particular, the different ways one can root the join tree gives rise to several individual computational schedules to obtain the same results [3]. Here, choosing the root of a join tree by using some specific potential function could lead to improving existing join evaluation algorithms in practice. We leave for future work on how one can use this principle for query evaluation in the presence of join trees.

Another possible application is in the context of tree-structured data, like XML or JSON documents. Although this data is usually rooted, assessing the most crucial node using a centrality measure can lead to a better understanding of the document's structure. For instance, given a tree-structured document, one could measure the difference between the root provided by potential function and the original root and see how this difference affects query evaluation, document representation, or other metrics.

Finally, in a broader sense, one could see centrality measures over graphs as an instance of a general database problem: find the most central data object in the data model given its underlying structure. The data model could be a relational database, an object-oriented database, an RDF database, or even a tree-structure database. For all these cases, the principle should be the same: the more relevant the data object is for its data model, the more central it should be. The present work could be seen as the first step toward this direction, namely, understanding data centrality in the case of trees. We leave for future work on how to extend this line of research to other classes of graphs or data models.

References

1 Neo4j: Centrality. `https://neo4j.com/docs/graph-data-science/current/algorithms/centrality/`.

2 Tigergraph: Centrality algorithms. `https://docs.tigergraph.com/graph-ml/current/centrality-algorithms/`.

3 Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of databases*, volume 8. Addison-Wesley Reading, 1995.

4 Francis Bloch, Matthew O. Jackson, and Pietro Tebaldi. Centrality measures in networks. *arXiv preprint*, 2019. `arXiv:1608.05845`.

5 Paolo Boldi, Alessandro Luongo, and Sebastiano Vigna. Rank monotonicity in centrality measures. *Network Science*, 5(4):529–550, 2017.

6 Paolo Boldi and Sebastiano Vigna. Axioms for centrality. *Internet Mathematics*, 10(3-4):222–262, 2014.

7 Phillip Bonacich. Factoring and weighting approaches to status scores and clique identification. *Journal of mathematical sociology*, 2(1):113–120, 1972.

8 Ulrik Brandes. *Network analysis: methodological foundations*, volume 3418. Springer Science & Business Media, 2005.

9 Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2009.

10 Zoltán Dezső and Albert-László Barabási. Halting viruses in scale-free networks. *Physical Review E*, 65:055103, 2002.

11 Linton C Freeman. A set of measures of centrality based on betweenness. *Sociometry*, pages 35–41, 1977.

12 Manuj Garg. Axiomatic foundations of centrality in networks. *Available at SSRN 1372441*, 2009.

13 Per Hage and Frank Harary. Eccentricity and centrality in networks. *Social networks*, 17(1):57–63, 1995.

14 Aidan Hogan, Andreas Harth, Jürgen Umbrich, Sheila Kinsella, Axel Polleres, and Stefan Decker. Searching and browsing linked data with swse: The semantic web search engine. *Journal of web semantics*, 9(4):365–401, 2011.

15 Gábor Iván and Vince Grolmusz. When the web meets the cell: using personalized PageRank for analyzing protein interaction networks. *Bioinformatics*, 27(3):405–407, 2010.

16 Matthew O Jackson. *Social and economic networks*. Princeton university press, 2010.

17 Mitri Kitti. Axioms for centrality scoring with principal eigenvectors. *Social Choice and Welfare*, 46(3):639–653, 2016.

18 Dirk Koschützki, Katharina Anna Lehmann, Leon Peeters, Stefan Richter, Dagmar Tenfelde-Podehl, and Oliver Zlotowski. Centrality indices. In *Network analysis*, pages 16–61. Springer, 2005.

19 Jose L Martinez-Rodriguez, Aidan Hogan, and Ivan Lopez-Arevalo. Information extraction meets the semantic web: a survey. *Semantic Web*, 11(2):255–335, 2020.

20 Mark Newman. *Networks*. Oxford university press, 2018.

21 Juhani Nieminen. On the centrality in a directed graph. *Social Science Research*, 2(4):371–378, 1973.

22 Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab, 1999.

23 K Brooks. Reid. Centrality measures in trees. In *Advances in Interdisciplinary Applied Discrete Mathematics*, pages 167–197. World Scientific, 2011.

24 Cristian Riveros and Jorge Salas. A family of centrality measures for graph data based on subgraphs. In Carsten Lutz and Jean Christoph Jung, editors, *ICDT*, volume 155, pages 23:1–23:18, 2020.

25 Gert Sabidussi. The centrality index of a graph. *Psychometrika*, 31(4):581–603, 1966.

26 Oskar Skibski. Vitality indices are equivalent to induced game-theoretic centralities. In Zhi-Hua Zhou, editor, *IJCAI*, pages 398–404. International Joint Conferences on Artificial Intelligence Organization, 2021.

27 Oskar Skibski, Tomasz P. Michalak, and Talal Rahwan. Axiomatic characterization of game-theoretic centrality. *Journal of Artificial Intelligence Research*, 62:33–68, 2018.

28 René van den Brink and Robert P. Gilles. Measuring domination in directed networks. *Social Networks*, 22(2):141–157, 2000.

29 Tomasz Wąs and Oskar Skibski. Axiomatization of the PageRank centrality. In *IJCAI*, pages 3898–3904. International Joint Conferences on Artificial Intelligence Organization, 2018.

30 Mihalis Yannakakis. Algorithms for acyclic database schemes. In *VLDB*, volume 81, pages 82–94, 1981.

Size Bounds and Algorithms for Conjunctive Regular Path Queries

Tamara Cucumides
University of Antwerp, Belgium
Pontificia Universidad Católica de Chile, Santiago, Chile

Juan Reutter
Pontificia Universidad Católica de Chile, Santiago, Chile
Millennium Institute for Foundational Research on Data (IMFD), Santiago, Chile

Domagoj Vrgoč
University of Zagreb, Croatia
Pontificia Universidad Católica de Chile, Santiago, Chile

Abstract

Conjunctive regular path queries (CRPQs) are one of the core classes of queries over graph databases. They are join intensive, inheriting their structure from the relational setting, but they also allow arbitrary length paths to connect points that are to be joined. However, despite their popularity, little is known about what are the best algorithms for processing CRPQs. We focus on worst-case optimal algorithms, which are algorithms that run in time bounded by the worst-case output size of queries, and have been recently deployed for simpler graph queries with very promising results. We show that the famous bound on the number of query results by Atserias, Grohe and Marx can be extended to CRPQs, but to obtain tight bounds one needs to work with slightly stronger cardinality profiles. We also discuss what algorithms follow from our analysis. If one pays the cost for fully materializing graph queries, then the techniques developed for conjunctive queries can be reused. If, on the other hand, one imposes constraint on the working memory of algorithms, then worst-case optimal algorithms must be adapted with care: the order of variables in which queries are processed can have striking implications on the running time of queries.

2012 ACM Subject Classification Information systems → Query languages

Keywords and phrases graph databases, regular path queries, worst-case optimal algorithms

Digital Object Identifier 10.4230/LIPIcs.ICDT.2023.13

Funding This work is supported by: ANID – Millennium Science Initiative Program – Code ICN17_002 and ANID Fondecyt Regular project number 1221799.

1 Introduction

Graph patterns form the basis of most query languages for graph databases [1]. Consequently, there has been a lot of progress in terms of pattern query answering, either by porting and optimizing relational techniques into a graph context [15, 11, 12], or by implementing worst-case optimal algorithms over graphs, which run in time given by the AGM bound of queries [14, 10, 2], or even with a mix of both approaches [8].

However, the main focus has been so far on simple graph patterns, or *conjunctive queries* (CQs), which are matched to the queried database. But one of the key aspects that differentiate graph and relational databases is the need for answering path queries, which are usually integrated into graph patterns to form so called conjunctive regular path queries (CRPQs). CRPQs form an important use case for graph patterns [1], but so far we know little about algorithms that can compute answers of these queries.

Consider for instance the CRPQ in Figure 1. We assume in this paper the standard relational representation of graphs using one binary relation per edge label. Namely, each edge label a results in a relation R_a containing all pairs (v, v') connected by an a-labelled edge in the graph. Then Q_1 features a triple join, but one of the relations we are joining is given by expression a^+, which corresponds to the transitive closure of the relation R_a. How should one compute this query? One approach is to first *materialize* the answers of all path queries, after which we have a simple graph pattern or CQ over these materialized relations, whose answers we already know how to compute [16, 13]. In our case, this means computing the transitive closure R_a^+ of R_a, as a virtual relation, and then compute the (relational) triple join $R_a^+(x,y) \wedge R_b(y,z) \wedge R_c(z,x)$, treating now a^+ as if it was a standard relation. Is this efficient? Let us assume for simplicity that the cardinality of R_a, R_b and R_c is N. Then, the virtual relation R_a^+ may have up to N^2 tuples. If we use a worst-case algorithm for the task of computing the triple-join, we can get the answers in $O(N^2)$, which also encompass the time taken to build the virtual relation R_a^+ for dealing with a^+. As we shall see, the $O(N^2)$ bound also corresponds to the maximum number of tuples that may be in the answer of this query, so our algorithm can be dubbed *worst-case optimal.* In this case, the approach seems plausible, at least in terms of worst-case asymptotic complexity.

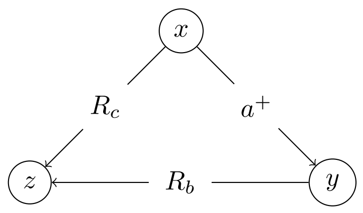


Figure 1 $Q_1(x,y,z) \leftarrow a^+(x,y) \wedge R_b(y,z) \wedge R_c(x,z)$.

On the other hand, our strategy of materializing transitive closure (or more generally, any path query) can be quite costly, as R_a^+ may have up to N^2 tuples itself, which need to be stored in memory. Thus, it is natural to ask if there is any way of computing the answers for this query in an optimal way, and in such a way that we do not pay the cost of fully materializing all path queries. And perhaps more importantly, what happens with other CRPQs? Do we have a worst-case optimal algorithm for every CRPQ? Does it necessarily involve materializing all path queries beforehand?

In this paper we provide answers to these questions. We study bounds on the maximum size of the answer of a CRPQ, given certain cardinality information about the graph. We use these bounds to investigate optimal algorithms for CRPQs, either in full generality, or with additional memory constraints. Our main contributions are as follows.

1. Regarding output bounds for CRPQs, we first observe that **the bound obtained by materializing RPQs and applying the standard AGM bound on the resulting query is not tight**. For example, consider the query Q_2 in Figure 2 below:

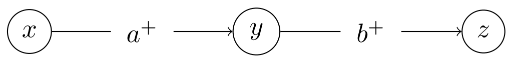


Figure 2 $Q_2(x,y,z) \leftarrow a^+(x,y) \wedge b^+(y,z)$.

If $|R_a| = |R_b| = N$ then the answers to a^+ and b^+ may have up to N^2 tuples. Thus, applying the usual AGM bound over the CQ resulting from materializing both expressions into relations gives an upper bound of $O(N^4)$. This is of course not tight: since $|R_a| = |R_b| = N$, the number of possible data values in any relation is also bounded by N, so the total number of tuples in the answer is $O(N^3)$. One can show that this bound is actually tight.

2. We can obtain much more precise bounds for Q_2 if we also take into account the cardinality of the first and second attributes of both R_a and R_b. For example, if we assume that the cardinality of the projection of R_a and R_b over the first or second attributes is bounded by M, then the number of tuples in the output of Q_2 is in $O(M^3)$. And we can generalize this for every CRPQ: **We provide bounds on the number of tuples in the answer of any CRPQ, over any graph satisfying the same cardinalities of relations and each of their attributes**. Our upper bound is based on an extension of the linear program used to show the AGM bound. Consider for example query $Q_3(x, y, z) \leftarrow a^+(x, y) \wedge b^+(y, z) \wedge R_c(x, z)$ in Figure 3.

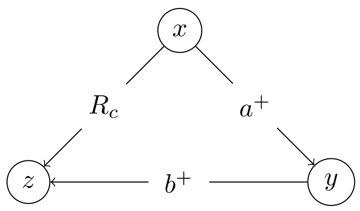


Figure 3 $Q_3(x, y, z) \leftarrow a^+(x, y) \wedge b^+(y, z) \wedge R_c(x, z)$.

Let R_a^s be the projection on the first attribute of R_a, R_a^e the projection on the second attribute (and analogously for R_b). Then the answers of Q_3 over a given graph with relations R_a, R_b, R_c are bounded by 2^{ρ^*}, where ρ^* is the solution of the following program.

$$\begin{aligned}
\text{minimize} \quad & u^{R_c} \log|R_c| + u_x^{a^+} \log|R_a^s| + u_y^{a^+} \log|R_a^e| + u_y^{b^+} \log|R_b^s| + u_z^{b^+} \log|R_b^e| \\
\text{where} \quad & u^{R_c} + u_x^{a^+} \geq 1 \\
& u^{R_c} + u_z^{b^+} \geq 1 \\
& u_y^{a^+} + u_y^{b^+} \geq 1 \\
& u^{R_c}, u_x^{a^+}, u_y^{a^+}, u_y^{b^+}, u_z^{b^+} \geq 0
\end{aligned} \tag{1}$$

This is a generalization of the AGM linear program [4], in which now we can also assign weights to the starting and ending points of RPQs, which receive their own variables ($u_x^{a^+}$ and $u_y^{a^+}$ for a^+, $u_y^{b^+}$ and $u_z^{b^+}$ for b^+). Assume that the cardinality of R_a^s, R_a^e, R_b^s and R_b^e is M, and the cardinality of R_c is N, with $N \leq M^2$. Then, an optimal solution for this query is $u^{R_c} = 1$, $u_y^{a^+} = u_y^{b^+} = \frac{1}{2}$, and $u_x^{a^+} = u_z^{b^+} = 0$. Intuitively, this means assigning *full weight* to the $R_c(x, z)$ atom of the query, and evenly dividing the weights for vertex y. This makes sense, because the answers of Q are always bounded by MN: for each tuple (u, v) in R_c there are at most M nodes connected to u and v by means of the expressions a^+ and b^+.

3. Now that we know how to bound the answers of CRPQ, the next question is to look for worst-case optimal algorithms for them: an algorithm for a query Q is worst-case optimal if, on input graph G, the answers of Q over G are computed in time bounded by the maximum number of tuples in the answer of Q over any graph with the same cardinalities of all the relations as G. Unfortunately we show that, under usual complexity assumptions, **there are CRPQs for which no worst-case optimal algorithm exists.**

4. Two strategies stand off when thinking about computing the answers of CRPQs. The first we already mentioned: materialize every path query as a virtual relation, and then apply a worst case optimal algorithm such as e.g. Leapfrog Trie-join [16]. For some queries, such as the triangle query in Figure 3, this strategy appears to be as optimal as one can be, at least in terms of computation time in the worst case. However, the memory requirements are quite high, as materialized path queries can be of quadratic size in terms of the number of nodes in the graph. On the other hand, one can immediately perform Leapfrog Trie-join on the graph as if it was a relational database, and whenever one needs pairs of the form (a, x) connected by a path query r, one computes it on demand, say by doing a Breadth First Search (BFS) over the relation. Assuming we do not cache intermediate results, this strategy has no significant memory requirements, but it may incur in chained searches on the graph, and end up being slower than materialization. At a first glance, it would appear that we have a strict time/memory tradeoff when computing this type of queries. But is this the best we can do? As it turns out, by carefully planning how RPQs are instantiated within worst case optimal algorithms, **we provide an algorithm that can compute the answers of many CRPQs under the same running time as an algorithm based on full materialization of path queries, but requiring only linear memory, in terms of the nodes of the graph**.

2 Preliminaries

Graph databases. A *graph database* is usually defined in the theoretical literature as a directed edge-labelled graph [5, 18]. More formally, if Σ is a finite alphabet of edge labels, a graph database over Σ is a pair (V, E), where V is a finite set of nodes, and $E \subseteq V \times \Sigma \times V$. An alternative way of viewing a graph database is through its relational representation. Namely, if Σ is a finite labelling alphabet, a graph database $G = (V, E)$ over Σ can be given as a relational database over the schema $\{R_a\}_{a\in\Sigma}$ of binary relations. Intuitively, $R_a(v, v')$ holds if and only if $(v, a, v') \in E$; that is, if there is an a-labelled edge between v and v'. Throughout the paper we will often switch between these two representations. For a binary relation R_a, with $a \in \Sigma$, we denote with R_a^s the projection of R_a onto its first attribute. Similarly we define R_a^e as the projection of of R_a onto the second attribute. In the remainder of the paper we will often use the term graph when referring to a graph database.

Queries over graph databases. Path queries are usually given as regular expressions, under the name of Regular Path Queries, or RPQs. An RPQ r selects, in a graph G, all pairs (u, v) of nodes that are connected via edge labels forming a word in the language of r. We denote this set of pairs as $[\![r]\!]_G$, see Table 1 for the definition. We assume RPQs are given both by regular expressions or automata, and freely switch between these representations.

Conjunctive regular path queries (CRPQs) [1, 5], are simply conjunctions of path queries. In order to exploit what is known about size bounds for relational CQs, we separate the expressions in our CRPQ into two sets: (i) the expressions consisting of a single letter (which are thus equivalent to an ordinary CQ); and (ii) regular expressions whose languages contain more than a single letter. Therefore, we define a conjunctive regular path query over a graph database to be given by an expression

$$Q(\overline{x}) \leftarrow \bigwedge_{i=1}^{\ell} R_{a_i}(y_i, z_i) \wedge \bigwedge_{i=\ell+1}^{k} r_i(y_i, z_i) \tag{2}$$

Table 1 Semantics of RPQs, for $a \in \Sigma$, and r, r_1 and r_2 arbitrary RPQs. The symbol $\circ$ denotes the composition of binary relations.

$[\![\varepsilon]\!]_G$	$=$	$\{(u,u) \mid u \text{ is a node id in } G\}$	$[\![a]\!]_G$	$=$	$\{(u,v) \mid (u,a,v) \in G\}$
$[\![r_1 \cdot r_2]\!]_G$	$=$	$[\![r_1]\!]_G \circ [\![r_2]\!]_G$	$[\![r_1 + r_2]\!]_G$	$=$	$[\![r_1]\!]_G \cup [\![r_2]\!]_G$
$[\![r^+]\!]_G$	$=$	$[\![r]\!]_G \cup [\![r \circ r]\!]_G \cup [\![r \circ r \circ r]\!]_G \cup \cdots$	$[\![r^*]\!]_G$	$=$	$[\![\varepsilon]\!]_G \cup [\![r^+]\!]_G$

where $a_i \in \Sigma$, r_i is a regular expression whose language is not equal to a single one letter word over Σ, and $\overline{x} = \{x_1, \dots, x_n\} \subseteq \{y_1, z_1, \dots, y_k, z_k\}$ is a set of output variables. A CRPQ without such regular expressions is simply a *conjuntive query* (CQ). Further, a CRPQ is *full* if every variable y_i, z_i is also mentioned in $\bar{x}$, and it is ε-free if none of the expressions r_i admit ε in their language. The expression to the right of the arrow is the *body* of query Q.

The semantics of a CRPQ Q, over a graph G is given via homomorphisms [1]. Namely, a mapping $\mu : \{x_1, \dots, x_n\} \to V$ is an output of Q over G when μ can be extended to the variables of Q in such a way that for each $i \in \{1, \dots \ell\}$ $R_{a_i}(\mu(y_i), \mu(z_i))$ holds, and for each $i \in \{\ell+1, \dots k\}$, $(\mu(y_i), \mu(z_i)) \in [\![r_i]\!]_G$. We denote the set of all outputs with $\text{Eval}(Q, G)$. A CRPQ Q is compatible with a graph G if the graph features all relations mentioned in Q.

Cardinality Profiles. For a given graph database G, we use r^s to denote the number of nodes in G that can participate as starting points for a path labelled by r in G: it corresponds to the union of each R^s of each relation R that labels a transition out of the initial state of the automaton for r. Likewise, r^e is the union of each R^e of each relation that labels a transition into a final state of the automaton for r.

In order to reason about bounds on graph databases, we always assume access to some basic statistics about the size of relations in the graph. Formally, the *cardinality profile* of a graph database G over Σ with respect to a query Q includes the following cardinalities:

- $|V|$ the total number of nodes;
- For each atom $R_a(y, z)$ in Q, the number $|R_a|$ of a-labelled edges;
- For each atom $r(y, z)$ in Q, with r a regular expression, the number r^s of starting points and r^e of final points participating in r.

To avoid extra notation, we also assume that a graph database G contains, in addition to the edge relation, every unary relation of the form r^s or r^e. Notice one can always add these unary relations in linear time.

The AGM bound. Atserias, Grohe and Marx [4] link the size bound of a relational join query to the optimal solution to a given linear program. In graph terms, let $Q(x_1, \dots, x_n) \leftarrow \bigwedge R_{a_i}(y_i, z_i)$ be a full conjunctive query without self-joins, i.e, in which each a_i is different, and let G be a graph database where the size of each R_{a_i} is N_i. Atserias et. al. [4] show that an optimal bound is achieved by considering the following linear program:

$$\begin{array}{lll} \text{minimize} & \displaystyle\sum_{i=1}^{n} u^{R_{a_i}}\ logN_i & \\ \text{where} & \displaystyle\sum_{\text{i : } x \text{ appears in atom } R_{a_i}} u^{R_{a_i}} \geq 1 & \text{for each variable } x \text{ in } Q \\ & u^{R_{a_i}} \geq 0 & \text{for } i = 1, \dots, m \end{array} \tag{3}$$

Let us denote by $\rho^*(Q, D)$ the optimal value of $\sum_{i=1}^{n} u^{R_{a_i}}\ \log N_i$. The AGM bound [4] can then be stated as follows.

▶ **Theorem 1** (AGM bound). *Let Q be a full CQ without self joins, D a database instance and $\rho^*(Q,D)$ the optimal solution of the associated linear program (3). Then,*

$$|Eval(Q,D)| \leq 2^{\rho^*(Q,D)}.$$

Furthermore, there are arbitrary large instances D for which we have $|Eval(Q,D)| = 2^{\rho^*(Q,D)}$.

We remark that all the results in this paper refer to *data complexity*, and thus the size of CRPQs is treated as a constant throughout our analysis.

3 Size bounds for CRPQs

Path queries provide an interesting challenge when studying size bounds. Every path query is a relation in itself, but in the worst case, a query like $a^+(x,y)$ may end up connecting all nodes in R_a^s with all nodes in R_a^e, thus invoking a quadratic jump in terms of the size of the potential nodes matching to x and to y. For this reason, tight output bounds must take into account the number of nodes that can participate as the starting point and the ending point of the expressions mentioned in the queries. We show how to construct a modified linear program, extending that of [4], that we use to provide our size bounds.

3.1 Motivation: underlying flat CQs

To see the intuition for our linear program, let us come back to query $Q_3(x,y,z)$ in Figure 3, and consider a graph G. In order to bound the size of $\text{Eval}(Q_3,G)$, we reason in terms of the size of $[\![a^+]\!]_G$. In the worst possible case, we have that $[\![a^+]\!]_G = R_a^s \times R_a^e$, that is, any node from R_a^e can be reached from any node from R_a^s. It is then easy to see that the answers in the evaluation $\text{Eval}(Q_3,G)$ will always be contained in what we call the *flat* CQ

$$\mathit{flat}(Q_3)(x,y,z) \;\leftarrow\; R_a^s(x) \wedge R_a^e(y) \wedge R_b^s(y) \wedge R_b^e(z) \wedge R_c(x,z),$$

in which every path query is replaced by the cross product of two unary relations, the possible starting nodes and the possible ending nodes. In fact, assuming each of R_a^s, R_a^e, R_b^s and R_b^e are unary relations in G, we have that $|\text{Eval}(Q_3,G)| \leq |\text{Eval}(\mathit{flat}(Q_3),G)|$, and this holds for any graph G compatible with Q. Now $\mathit{flat}(Q_3)$ is a full CQ without self-joins, and we know how to bound its output [4], which immediately results in an upper bound for Q_3.

Interestingly, the focus on flat conjunctive queries has another intuitive reading. Coming back to query Q_2 from Figure 2, its flat version is simply a cross product of unary relations

$$Q_2(x,y,z) \;\leftarrow\; R_a^s(x) \wedge R_a^e(y) \wedge R_b^s(y) \wedge R_b^e(z).$$

For a graph G in which all of R_a^s, R_a^e, R_b^s and R_b^e have N nodes, we verify that $|\text{Eval}(Q,G)| \leq N^3$. This cubic bound is, in a sense, the most crude upper bound one could get for a conjunctive query: it is simply the cross product of every vertex matching for x, y and z. It just happens that when the labels joining x and y, and y and z are path queries, this crude bound ends up being realistic.

But is it tight? We can show it is, and our size bounds end up enjoying several good properties proved before for full join queries [4] or conjunctive queries [9]. Moving from this simple example to arbitrary CRPQs, however, is not that easy, and we proceed in several steps. In section 3.2 we start with a fragment of CRPQs for which the proof is simpler, and the bounds much more elegant. This fragment corresponds to full CRPQs, without self-joins or any repetition of labels between atoms, and whose RPQs are defined by ε-free expressions that admit at least one word of length 2. We call this fragment *Simple* CRPQs, and the reason for starting with this fragment is that we can define the general upper and lower bounds exactly in the same way they were defined for simple CQs by Atserias et al. in their seminal paper[4]. We then extend our results to arbitrary CRPQs defined by ε-free expressions, with the only caveat that our lower bound is now up to a constant that depends on the query. We finish with CRPQs that may use expressions including ε, such as a^*, which is one of the most common path query occurring in practice [6]. We deal with them by separating into ε and ε-free parts, which we can then treat independently.

3.2 Simple CRPQs

To state our first result, we provide a formal definition of the aforementioned simple fragment. A *simple* CRPQ is a *full* CRPQ of the form $Q(\overline{x}) \leftarrow \bigwedge_{i=1}^{\ell} R_{a_i}(y_i, z_i) \wedge \bigwedge_{i=\ell+1}^{k} r_i(y_i, z_i)$ with the following properties:

- Each relation R_{a_i} appears only once in Q (no self joins);
- All regular expressions r_i are ε-free and they contain a word of length at least 2;
- If r and r' are two different regular expressions in Q, then the set of all labels in the first or last position of any word in the language of r is disjoint to that of r'.

As we hinted in the introduction, the idea is to extend the linear program of AGM with one *vertex* variable for each endpoint of every atom $r(x, y)$ in the query, which are then constrained in the same fashion as edge variables. Alternatively, one can directly construct the program for the corresponding flat query: it happens to be exactly the same program.

▶ **Theorem 2** (Bound for simple CRPQs). *Assume that the query* $Q(\overline{x}) \leftarrow \bigwedge_{i=1}^{\ell} R_{a_i}(y_i, z_i) \wedge \bigwedge_{i=\ell+1}^{k} r_i(y_i, z_i)$ *is a simple CRPQ. Then for any graph G we have that*

$$|Eval(Q, G))| \leq 2^{\rho^*(Q,G)}$$

where $\rho^*(Q, G)$ is the optimal solution of the following linear program:

$$\begin{aligned} &\textit{minimize} && \sum_{i=1}^{\ell} u^{R_{a_i}} \ \log |R_{a_i}| \ + \sum_{i=\ell+1}^{k} (u_{y_i}^{r_i} \log |r_i^s| + u_{z_i}^{r_i} \log |r_i^e|) \\ &\textit{where} && \sum_{i:x=y_i \vee i:x=z_i} u^{R_{a_i}} + \sum_{i:x=y_i} u_{y_i}^{r_i} + \sum_{i:x=z_i} u_{z_i}^{r_i} \geq 1 \quad \textit{for } x \in \overline{x} \\ &&& u^{R_{a_i}} \geq 0 \quad \textit{for } i \in [1, \ell] \\ &&& u_{y_i}^{r_i}, u_{z_i}^{r_i} \geq 0 \quad \textit{for } i \in [\ell+1, k] \end{aligned} \tag{4}$$

Furthermore there are arbitrarily large instances for which

$$|Eval(Q, G))| \geq 2^{\rho^*(Q,G)}.$$

The upper bound. The upper bound can be obtained using flat CQs. Let $Q(\overline{x})$ be a simple CRPQ. Its underlying *flat* query *flat*(Q) is the conjunctive query defined as:

$$\mathit{flat}(Q)(\overline{x}) \leftarrow \bigwedge_{i=1}^{\ell} R_{a_i}(y_i, z_i) \wedge \bigwedge_{i=\ell+1}^{k} r_i^s(y_i) \wedge r_i^e(z_i)$$

Recall we assume for simplicity that each r^s and r^e is an unary predicate already present in G. The following is now easy to check:

▶ **Lemma 3.** *$Eval(Q, G) \subseteq Eval(flat(Q), G)$, with Q a simple CRPQ.*

Since the linear programs of both *flat*(Q) (as in [4]) and Q (as in the statement of Theorem 2) coincide, and $2^{\rho^*(flat(Q),G)}$ is an upper bound for Eval(*flat*(Q), G), this immediately proves the upper bound of Theorem 2.

The lower bound. We will prove the lower bound by constructing an instance out of the dual program for Q. Let us first illustrate the tightness of the bound via the means of an example. Consider again query $Q_3(x, y, z) \leftarrow a^+(x, y), b^+(y, z), R_c(x, z)$.

The linear program for this query is as seen in (1) and the corresponding dual is:

$$\begin{aligned}
&\text{maximize: } v_x + v_y + v_z\\
&\text{subject to: } v_x + v_z \leq \log |R_c|\\
&\qquad v_x \leq \log |(a^+)^s| \qquad v_y \leq \log |(a^+)^e|\\
&\qquad v_y \leq \log |(b^+)^s| \qquad v_z \leq \log |(b^+)^e|\\
&\qquad v_x, v_y, v_z \geq 0
\end{aligned}$$

Consider an optimal solution $\overline{x}$ for the primal and (for duality) a solution to the dual (v_x, v_y, v_z) such that $\rho^*(Q, D) = v_x + v_y + v_z$. Now we want to build an instance G such that Eval$(Q, G) = 2^{\rho^*(Q,G)}$ with $|(a^+)^s| = 2^{v_x}$, $|(a^+)^e| = 2^{v_y}$, $|R_b^s| = 2^{v_y}$, $|(b^+)^e| = 2^{v_z}$ and $|R_c| = 2^{v_x+v_z}$. The instance is defined as follows,

- We have a special vertex $\star$ and 3 sets of vertices: $|V_x| = 2^{v_x}$, $|V_y| = 2^{v_y}$, $|V_z| = 2^{v_z}$ such that $V_x \cap V_y \cap V_z = \{\star\}$
- Add edges (x, c, z) for every pair of nodes $(x, z) \in V_x \times V_z$
- Add edges $(x, a, \star)$ for every $x \in V_x$ and edges $(\star, a, y)$ for $y \in V_y$
- Finally, add edges $(y, b, \star)$ for $y \in V_y$ and $(\star, b, z)$ for $z \in V_z$.

By the dual restrictions, we can check that the cardinalities are equal or smaller than we wanted (if they're smaller we can add random edges as this can only increase the number of tuples of Eval(Q, G))). Also we can check that $|\text{Eval}(Q, G)| = 2^{v_x+v_y+v_z}$ since we have all tuples (x, y, z) with $x \in V_x$, $y \in V_y$ and $z \in V_z$ in the result. We conclude that $|\text{Eval}(Q, G)| = 2^{\rho^*(Q,G)}$.

Now we formalize this construction for any simple CRPQ:

Proof of Theorem 2, lower bound. As before, we use the dual program of equation (4)

$$\begin{aligned}
&\text{maximize: } \sum_{x \in \overline{x}} v_x\\
&\text{subject to: } v_{y_i} + v_{z_i} \leq \log |R_{a_i}|, \qquad i = 1, \ldots, \ell\\
&\qquad v_{y_i} \leq \log |r_i^s|, \qquad i = \ell + 1, \ldots, k\\
&\qquad v_{z_i} \leq \log |r_i^e|, \qquad i = \ell + 1, \ldots, k\\
&\qquad v_x \geq 0, \qquad x \in \overline{x}
\end{aligned}$$

Consider an instance with cardinalities $|R_{a_i}| = N_i$ for $i \in [1, \ell]$, $|r_j^s| = N_j^s$ and $|r_j^e| = N_j^e$ for $j \in [\ell + 1, k]$. By duality, for any solution $\overline{u}$ to the primal and $\overline{v}$ for the dual, we have that

$$\sum_{i=1}^{\ell} u^{R_{a_i}} \log |R_{a_i}| + \sum_{i=\ell+1}^{k} (u_{y_i}^{r_i} \log |r_i^s| + u_{z_i}^{r_i} \log |r_i^e|) \geq \sum_{x \in \overline{x}} v_x,$$

with equality when the solutions are optimal. Let us assume that all N_i, N_i^s and N_i^e are of the form 2^{L_i} for some $L_i \in \mathbb{N}$ so the optimal solution of both the primal and dual are rational. Let $\overline{v}$ be the dual solution and write each v_x as p_x/q. Then $\overline{p}$ is an optimal solution to the linear program with cardinalities N_i^q. Now we present a graph database G with $|R_i| = N_i^q$, $|r_i^s| = (N_i^s)^q$ and $|r_i^e| = (N_i^e)^q$ such that $|\text{Eval}(Q,G)| \geq 2^{\rho^*(Q,G)}$.

- The vertices of G is the union of sets $V_x = \{1, \dots, 2^{v_x}\}$ for each $x \in \overline{x}$. Also consider a vertex $\star$ that is part of every V_x.
- For every atom $R_{a_i}(y_i, z_i)$ in Q, add to G one edge (u, a_i, v) for every pair (u, v) in $V_{y_i} \times V_{z_i}$.
- For every atom $r_i(y_i, z_i)$ in Q, choose an arbitrary word $\pi_i = a_{i_1} \dots a_{i_N}$ of length at least 2 in the language of r_i and
  - Add to G the edges $(u, a_{i_1}, \star)$ for each $u \in V_{y_i}$.
  - Add to G edges $(\star, a_{i_j} \star)$ for every $j \in [2, N-1]$.
  - Add to G the edges $(\star, a_{i_N}, v)$ for each $v \in V_{z_i}$.

From the construction we verify that:

$$\begin{aligned}
|R_{a_i}| &= 2^{v_{y_i}+v_{z_i}} && \leq 2^{q \log N_i} = N_i^q && \forall i \in [1, \ell] \\
|r_i^s| &= 2^{v_{y_i}} && \leq 2^{q \log N_i^s} = (N_i^s)^q && \forall i \in [\ell+1, k] \\
|r_i^e| &= 2^{v_{z_i}} && \leq 2^{q \log N_i^e} = (N_i^e)^q && \forall i \in [\ell+1, k]
\end{aligned}$$

Further, we also verify that $\text{Eval}(Q,G)$ contains all tuples $t \in V_{x_1} \times \cdots \times V_{x_n}$. Now we add random edges and vertices such that $|R_i| = N_i^q$, $|r_i^s| = (N_i^s)^q$ and $|r_i^e| = (N_i^e)^q$. We now have a graph G with the desired cardinality profile for which:

$$|\text{Eval}(Q,G)| \geq \prod_{i=1}^{l} |R_{a_i}|^{u^{R_{a_i}}} \prod_{i=l+1}^{m} |r_i^s|^{u_{y_i}^{r_i}} \cdot |r_i^e|^{u_{z_i}^{r_i}} = 2^{\sum_{x \in \overline{x}} v_x} \qquad ◀$$

As in Atserias et al., we also show that the instances satisfying the lower bound can be constructed with a certain degree of regularity, in which all cardinalities are equal.

▶ **Corollary 4.** *Given a simple CRPQ Q, we can build an arbitrarily large instance G such that $|Eval(Q,G))| \geq 2^{\rho^*(Q,G)}$ with $|R_{a_i}| = |r_j^s| = |r_j^e|$ for every relation i and j such that $u^{R_{a_i}} > 0$, $u_{y_j}^{r_j} > 0$ and $u_{z_j}^{r_j} > 0$.*

Unfortunately, not every combination of cardinalities of relations and vertices can be shown to produce tight bounds. However, as in [4], we can show the following: Let Q be a simple CRPQ and G a graph. Then there exists a graph G' with the same cardinalities as G in all vertices and relations mentioned in Q, such that $\text{Eval}(Q, G')| \geq 2^{\rho^*(Q,G)-n}$, where n is the number of attributes of Q. As for CQs, this is essentially the best we can get.

3.3 Bound for arbitrary ε-free CRPQs

Gottlob et al. study how to go from relational join queries to CQs [9], and the same techniques can be used for obtaining size bounds for ε-free CRPQs, even if they feature projections, repetition of variables, or expressions allowing only words of size 1. Bounds remain tight, except this time they are tight up to a factor that does depend on the query (but not the data) in a polynomial way. We first show how to handle arbitrary full CRPQs that are ε-free (and not just the simple ones), and then move to CRPQs that project out some variables.

From full to simple CRPQs. We first show that for a full CRPQ Q that is also ε-free, and a graph database G compatible with Q, we can construct a simple CRPQ Q', and an instance G' compatible with Q' such that $\text{Eval}(Q,G) = \text{Eval}(Q',G')$. The translation from Q to Q' has to deal with repeated labels/relations, and also with expressions that accept only words of length 1. For this, we first, replace every appearance of a relation R_a or label a in any atom of Q with a fresh relation or label not used elsewhere in the query. Next, replace any atom of the form $r_i(y_i,z_i)$ where $r_i = (a_1|a_2|\dots|a_k)$ (i.e. an expression accepting only words of size 1), with an atom $R_{r_i}(y_i,z_i)$, where R_{r_i} is a fresh relation. Translation from a graph G compatible with Q, to a graph G' compatible with Q' is constructed by assigning every copy of R_a (introduced in the construction of Q') the same tuples as R_a, and by assigning to R_r, for an expression $r = (a_1|a_2|\dots|a_k)$, the tuples in the union of all $R_{a_1},\dots,R_{a_k}$. Other relations are the same as in G. We call (Q',G') the *simplified* version of (Q,G)

▶ **Proposition 5** (full CRPQs). *Consider a full CRPQ of form (2) in which every r_i is ε-free. For this query we have that $|Eval(Q,G)| = |Eval(Q',G')| \leq 2^{\rho^*(Q',G')}$, where Q' and G' the simplified version of Q and G. Furthermore, one can construct arbitrarily large instances G such that $|Eval(Q,G)|2^{p(|Q|)} \geq 2^{\rho^*(Q',G')}$ where $p(|Q|)$ is a polynomial that depends exclusively on Q.*

Bounds for projections of full, ε-free CRPQs. Consider a (non-full) ε-free CRPQ of the form

$$P(\overline{x}_0) \;\leftarrow\; Q(\overline{x}), \tag{5}$$

with $\overline{x}_0 \subsetneq \overline{x}$, and where Q is full and ε-free. From our previous result, we know that $\text{Eval}(Q,G)$ is always bounded by $2^{\rho^*(Q',G')}$, where Q' and G' constructed as above. As in [9], we consider a relaxation of the linear program for Q', in which we only keep those restrictions that refer to variables of Q (and Q') that are in $\overline{x_0}$. Formally, we denote by $2^{\rho^*_{\overline{x}_0}(Q',G')}$ the optimal solution of a modified linear program for Q' and G', where in the restrictions of (4) we only consider those referring to $\overline{x_0}$. We then have:

▶ **Proposition 6** (Queries with projections [9]). *Given an CRPQ P of the form (5) then for every graph database instance G we have that $|Eval(P,G)| \leq 2^{\rho^*_{\overline{x}_0}(Q',G')}$. Moreover, there are arbitrarily large instances G such that $|Eval(P,G)|2^{p(|Q|)} \geq 2^{\rho^*_{\overline{x}_0}(Q',G')}$, where $p(|P|)$ is a polynomial that depends exclusively on P.*

3.4 Dealing with ε

As we have mentioned, the evaluation of the expression ε over a graph $G = (V,E)$ contains the *diagonal* $D = \{(v,v) \mid v \in V\}$. Thus, the evaluation of expressions containing ε, such as a^*, are somehow the union of two different sets of results. On one hand there is the ε-free part, that we know how to deal with, and on the other there is ε, which behaves more like a relation, albeit drawing pairs only from the diagonal D.

The expression ε. Consider the triangle query $Q_4(x,y,z) \;\leftarrow\; R_a(x,y) \wedge R_b(y,z) \wedge \varepsilon(x,z)$, featuring two edge labels and the regular expression ε. One can check that Q_4 is equivalent to $R_a(x,y) \wedge R_b(y,z) \wedge \varepsilon^s(x) \wedge \varepsilon^e(z) \wedge x = z$. What we have done is to produce an analogue of the *flat* version of CRPQs, and we use the equalities to force the flat part to map only to the diagonal. We further transform this query by noting that $\varepsilon^s = \varepsilon^e = V$, and *chasing* away the equality, obtain the query $R_a(x,y) \wedge R_b(y,x) \wedge V(x)$, which always produces the same number of tuples as Q_4. Hence, dealing with epsilon involves (1) transforming every atom $\varepsilon(x, y)$ into two unary atoms $V(x), V(y)$ (to be interpreted as V), plus the corresponding equality $x = y$, and (2) chasing away such equalities. It is not difficult to see that both of these operations do not alter the size of the outputs of queries; the transformation always yields an equivalent query, save for the case when the arity of the query is reduced when chasing the equalities.

Formally, assume that Q is a CRPQ, and let $Q^{\setminus\varepsilon}$ be the query in which each atom $\varepsilon(x, y)$ is replaced for the construct $V(x) \wedge V(y) \wedge x = y$. Assuming V is interpreted as the set of vertices in every graph $G = (V, E)$, we have:

▶ **Lemma 7.** *For every CRPQ Q and graph G, $Eval(Q, G) = Eval(Q^{\setminus\varepsilon}, G)$*

Further, let Q be a CRPQ with equalities, i.e, additional atoms of the form $x = y$, where both x and y appear in a non-equality atom in Q. Let chase(Q) be CRPQ resulting by repeatedly replacing variable y for variable x for each atom $x = y$ in the query. We have:

▶ **Lemma 8.** *For every CRPQ Q and graph G, $|Eval(Q, G)| = |Eval(chase(Q), G)|$*

In order to formally state the bound for queries with ε, we use again query Q' and graph G' constructed in the previous subsection, as well as the solution $2^{\rho^*_{\overline{x}_0}(Q',G')}$ for the modified linear program for Q' and G'.

▶ **Proposition 9.** *Let $P(\overline{x_0})$ be a CRPQ in which every regular expression is either ε, or is ε-free, and G a graph, and assume that the body of chase($P^{\setminus\varepsilon}$) is of the form $Q(\overline{x})$, with $\overline{x_0} \subseteq \overline{x}$. Then for every graph database instance G we have that $|Eval(P, G)| \leq 2^{\rho^*_{\overline{x}_0}(Q',G')}$. Moreover, there are arbitrarily large instances G such that $|Eval(P, G)|2^{p(|P|)} \geq 2^{\rho^*_{\overline{x}_0}(Q',G')}$, where $p(|P|)$ is a polynomial that depends exclusively on P.*

Arbitrary RPQs. Arbitrary RPQs such as a^* are not so easy to deal with, as they represent, somehow, the union of the diagonal database and an ε-free CRPQ. Consequently, we will look into *splitting* CRPQs into parts with ε and parts without it. For a given CRPQ Q, let $r_{\ell_1}, \dots, r_{\ell_p}$ be the RPQs in Q that accept ε. We define the family of *split* queries $Q[S]$, for $S \subseteq \{\ell_1, \dots, \ell_p\}$, as follows. For each $r_\ell, \ell \in \{\ell_1, \dots, \ell_p\}$, find a decomposition $r_\ell = \varepsilon + \hat{r}_\ell$, where $\hat{r}_\ell$ is ε-free. Then atom $r_\ell(y_\ell, z_\ell)$ is replaced by $\hat{r}_\ell(y_\ell, z_\ell)$, if $\ell \in S$, or by $K_\ell(y_\ell) \wedge K_\ell(z_\ell) \wedge y_\ell = z_\ell$, where K_ℓ is a fresh relation symbol, if $\ell \notin S$.

Now augment any graph G to make it compatible with any $Q[S]$ by adding relation $K_\ell = \{a \mid a \notin \hat{r}^s_\ell \cap \hat{r}^e_\ell\}$ for each $\ell \in \{\ell_1, \dots, \ell_p\}$. It is not too difficult to prove that $|\text{Eval}(Q, G)| \leq \sum_{S \subseteq \{\ell_1,\dots,\ell_p\}} |\text{Eval}(\mathit{flat}(Q[S]), G)|$, and we can further turn this property into an output bound for queries[1].

▶ **Proposition 10.** *Let Q be a CRPQ. For any graph G we have that $|Eval(Q, G)| \leq \sum_{S \subseteq \{\ell_1,\dots,\ell_p\}} 2^{\hat{\rho}*(Q[S],G)}$, where $Q[S]$ are queries* split *from Q, and $2^{\hat{\rho}*(Q[S],G)}$ is the size output bound shown for $Q[S]$, in Proposition 9. Moreover, there are arbitrarily large graphs for which this bound is tight.*

One important caveat of this result is that the instances showing that the bound is tight work by constructing graphs G in which, for every expression $r_\ell = \varepsilon + \hat{r}_\ell$, we verify that $[\![\varepsilon]\!]_G \subseteq [\![\hat{r}_\ell]\!]_G$.

[1] For CRPQs with equalities, *flat*(Q) is defined just as before, all equalities are maintained.

4 WCO algorithms for CRPQs

In this section we deal with algorithms for computing CRPQs. Ideally, one would expect an algorithm that runs in the worst-case optimal bound from Theorem 2 (and subsequent generalizations). We call such an algorithm worst-case optimal, or wco algorithm for short. Unfortunately, as we review below, bounds from Casel and Schmid [7] directly imply that such algorithms do not exist under usual complexity assumptions. In the light of this, we establish a baseline which amounts to first computing all the answers to the regular expressions mentioned in our query, materializing them, and running a classical wco algorithm (e.g. GenericJoin [13]) on these materialized relations. We show that a modification of the GenericJoin algorithm of [13] can approach the optimal performance of our baseline for many CRPQs. As is usual in algorithms for relational/graph queries, we will assume all our queries to be full.

4.1 WCO algorithms for CRPQs may not exist

Casel and Schmid show lower bounds for the problem of evaluating a single RPQ [7]. Specifically, for a graph $G = (V, E)$, and a (regular path) query $Q(x, y) \leftarrow r(x, y)$, they prove that any algorithm capable of evaluating Q over G in time $O(|V|^{\omega} f(|Q|))$ can also be used to solve the *Boolean Matrix Multiplication* (BMM) problem: given two square matrices A and B of size n, compute the product matrix $A \times B$, in time $O(n^{\omega})$. In particular, this means that a quadratic algorithm for computing path queries does not exist unless the BMM hypothesis is false, and if we accept the weaker *combinatorial* BMM hypothesis [17], then no subcubic algorithm exists for computing Q. Since the answers to Q are clearly bounded by $|V|^2$, then we cannot hope for a worst-case optimal algorithm in this case.

A natural question is what happens with CRPQs that mix both path queries and relations in their edges. Perhaps the relations help soften the underlying complexity of the problem? Unfortunately, this is not the case. To see this, consider query $Q(x, y, z) \leftarrow R_a(x, y) \wedge S_b(y, z) \wedge r(x, z)$, where r is any regular expression. Given a graph G in which $|R_a| = |S_b| = n$, our results tell us that the answer of Q over G contains at most $O(n^2)$ tuples, and thus a worst-case algorithm must evaluate Q in time $O(n^2)$. But this algorithm can then be used to compute the answers for r over a graph $G = (V_G, E_G)$, where V_G contains at least n nodes $v_1, \ldots, v_n$. For this, we construct a graph database $G' = (V_G \cup \{1\}, E_{G'})$, where $R_a = \{(v_i, 1) \mid 1 \leq i \leq n\}$, $S_b = \{(1, v_i) \mid 1 \leq i \leq n\}$ and where the rest of the relations are as in G. Then a tuple $(v_i, 1, v_j)$ is in $\mathrm{Eval}(Q, G')$ if and only if (v_i, v_j) is an answer to r on G.

▶ **Proposition 11.** *An algorithm capable of computing the answers of every simple CRPQ Q over a graph G in time $O(2^{\rho^*(Q,G)})$ refutes the BMM hypothesis.*

Having ruled out the possibility of worst-case optimal algorithms, let us review what can we do with existing techniques.

As our baseline, we establish a rather naive algorithm, called FullMaterialization, which evaluates a CRPQ Q over a graph database G as follows:

1. Compute the answer of each RPQ r appearing in Q over G.
2. Materialize all of these binary relations and add them to G.
3. Use a (relational) wco algorithm (e.g. GenericJoin [13]) to compute the query answer.

In the final step, each RPQ is now simply treated as a relation that we have previously computed. This algorithm runs in time bounded by the time to compute the RPQs from Q, and the AGM bound of the query. However, the algorithm may require memory that is quadratic in terms of the nodes in the graph, to be able to store the results of RPQs.

While reasonable, this algorithm has practical issues: the quadratic memory footprint may be too big to store in memory, and we may be performing useless computations because most pairs in the answers of RPQs may not even match to the remainder of patterns. Memory usage may be alleviated by clever usage of compact data structures, as in e.g. [3], but we take a different approach.

In what follows, we impose that algorithms may only use $O(|V|)$ memory, for $G = (V, E)$. Since Proposition 11 rules out strict wco algorithms, our goal is to devise algorithms that are capable of achieving the running time of FullMaterialization, but using just linear memory (in data complexity). To analyse the running time of the algorithm, we first introduce some notation. For a CRPQ Q and a graph database G, with $\mathrm{AGM}(Q, G)$ we denote the bound for maximal size of $\mathrm{Eval}(Q, G')$, over all graphs G' that have the same cardinality profiles as G (this includes both the cardinalities of all the relations, as well as the projections on starting and ending points of these). The time complexity of FullMaterialization for a query Q, over a graph $G = (V, E)$, is bounded by $O(|V|^3 + \mathrm{AGM}(Q, G))$, where the cubic factor accounts to materializing all the RPQs in Q.

4.2 GenericJoin for CRPQs

In order to avoid materializing relations which are potentially quadratic in the size of the graph, we can utilize a simple idea: compute RPQs on-demand, the first time such an answer is needed. For this, we will adapt the (relational) wco algorithm GenericJoin of [13], so that it processes regular relations as needed. As we will see, this approach gives us good running time even when the memory is constrained, and can actually run under the FullMaterialization time bounds for a broad class of queries. For CRPQs, however, the order of variables we work with has striking implications on the efficiency of the algorithm.

If $Q(\overline{x}) \leftarrow \bigwedge_{i=1}^{\ell} R_{a_i}(y_i, z_i) \wedge \bigwedge_{i=\ell+1}^{k} r_i(y_i, z_i)$ is a full CRPQ, and G a graph database, then Algorithm 1 defines GenericJoinCRPQ(Q, G), a generalization of the GenericJoin wco algorithm from the relational setting to graphs and (full) CRPQs. Similarly as in [13], we assume an order on the variables of Q, and start to recursively strip one variable at a time. For a selected variable, we compute all the nodes that can be bound to this variable (line 5). Then we iterate over these nodes one by one, compute RPQs as needed, adding them to the database (lines 9–11 and 12–14), and proceed recursively (line 15). For the base case when we have only one variable, we simply complete the missing values (line 4).

Analysis. So how does this algorithm compare to FullMaterialization? Well, this is heavily dependent on the CRPQ we are processing. As an example, consider again the triangle query with two RPQs, $Q_3(x, y, z) \leftarrow a^+(x, y) \wedge b^+(y, z) \wedge R_c(x, z)$ as in Figure 3, and consider a graph G in which $|R_c| = N$ and all starting and ending points of RPQs a^+ and b^+ have cardinality M. Here FullMaterialization runs in time $O(M^3 + MN)$, but with quadratic memory (the first part of the sum is for computing answers of RPQs, the second part is the max number of outputs of the query). On the other hand, GenericJoinCRPQ achieves the same bound, but using only linear memory. To see this, let us assume the first chosen variable is y. As per line 5, we first iterate over all possible vertices v in $L = {b^+}^s \cap {a^+}^e$. For each such value, we compute sets $a^+[v] = \{v' \mid (v', v) \in [\![a^+]\!]_G\}$ and $b^+[v] = \{v' \mid (v, v') \in [\![b^+]\!]_G\}$, storing these in memory and adding them to $\hat{G}$ (here $\hat{G}$ is the augmented graph storing these relations). We then process the query $\hat{Q}(x, v, z) \leftarrow a^+[v](x) \wedge b^+[v](z) \wedge R_c(x, z)$ over the augmented graph $\hat{G}$. This query does not feature regular expressions, so we can compute its answers using GenericJoin$(\hat{Q}, \hat{G})$ from [13]. Further, the AGM bound for $\hat{Q}(x, v, z)$ is N, so the algorithm computes the answers in $O(N)$. Thus, the total running time is in

Algorithm 1 GenericJoinCRPQ(Q, G).

```
1:        ▷ $Q$ May have unary relations of the form $r[v]$, from previous recursive iterations.
2:  $A \leftarrow \emptyset$
3:  if $|\overline{x}| = 1$ then
4:      return Eval$(Q, G)$
5:  Pick a variable $x \in \overline{x}$        ▷ We compute into $L$ nodes that can potentially map to $x$
6:
        $$L \leftarrow \bigcap_{R(x,z)\in Q} R^s \bigcap_{R(y,x)\in Q} R^e \bigcap_{r(x,z)\in Q} r^s \bigcap_{r(y,x)\in Q} r^e \bigcap_{r[v](x)\in Q} r[v]$$
7:  for $v \in L$ do
8:      $\hat{Q} \leftarrow Q[x/v]$, $\hat{G} \leftarrow G$                ▷ We instantiate $x$ to $v$ in $\hat{Q}$
9:      for each atom $r(v, z) \in \hat{Q}$ do   ▷ Compute answers to $r(v, z)$, store them in $r[v](z)$
10:         $\hat{G} \leftarrow G \cup r[v]$, with $r[v] = \{v' \mid (v, v') \in [\![r]\!]_G\}$
11:         replace $r(v, z)$ for $r[v](z)$ in $\hat{Q}$
12:     for each atom $r(y, v) \in \hat{Q}$ do   ▷ Compute answers to $r(y, v)$, store them in $r[v](y)$
13:         $\hat{G} \leftarrow G \cup r[v]$, with $r[v] = \{v' \mid (v', v) \in [\![r]\!]_G\}$
14:         replace $r(y, v)$ for $r[v](y)$ in $\hat{Q}$
15:     $A[v] \leftarrow$ GenericJoinCRPQ$(\hat{Q}, \hat{G})$
16:     $A \leftarrow A \cup \{v\} \times A[v]$
17: return $A$
```

$O(|L| \cdot (M^2 + N)) = O(M \cdot (M^2 + N))$. Again, the first part of the sum is for computing the answers of the path queries, the second part for evaluating $\hat{Q}$. Importantly, this uses linear memory, as we refresh $a^+[v]$ and $b^+[v]$ after each new value in L.

So far good news, we managed to avoid quadratic memory at virtually no cost. Unfortunately, we cannot avoid it for all queries. Let us consider the triangle query but now with three RPQs: $Q(x, y, z) \leftarrow a^+(x, y) \wedge b^+(y, z) \wedge c^+(x, z)$. The cardinalities of all starting and endpoints will be N and let us assume that the first chosen variable is y so the computation goes as in the example above, except that $\hat{Q}(x, v, z) \leftarrow a^+[v](x) \wedge b^+[v](z) \wedge c^+(x, z)$ will still have one more RPQ to compute and therefore the running time will be in $O(N \cdot (N^2 + N^3))$. It is easy to see that all possible orders for this query will result in the same algorithm: for this query we cannot avoid having to nest at least the computation of two RPQs.

In the best case, thus, GenericJoinCRPQ does run in the sought after FullMaterialization time bounds. But for certain queries and orderings, the algorithm resorts to computing each RPQ on demand, which implies a much slower $O(\text{AGM}(Q, G) \cdot |V|^2)$ bound.

Queries for which GenericJoinCRPQ is efficient. As we have seen, the problem in our algorithm is that nesting the evaluation of RPQs is often too costly, and sends us above the FullMaterialization bound. As it turns out, we can characterize the types of queries for which the nesting can be avoided, and introduce a version of GenericJoinCRPQ that takes advantage of this structure.

For this, we will require the query Q is such that its RPQ components form a bipartite graph. More formally, assume that we have a full CRPQ $Q(\overline{x}) \leftarrow \bigwedge_{i=1}^{\ell} R_{a_i}(y_i, z_i) \wedge \bigwedge_{i=\ell+1}^{k} r_i(y_i, z_i)$. We will say that Q is *RPQ-bipartite*, if the graph $G_r(Q) = (V_r, E_r)$, with $V_r = \bigcup_{i=\ell+1}^{k} \{y_i, z_i\}$, and $E_r = \{(y_i, z_i) \mid i = \ell + 1, \ldots, k\}$, is bipartite. We call the graph

Algorithm 2 GenericJoinCRPQ-Bipartite$(Q, G, \overline{x}_1)$.

```
1: $A \leftarrow \emptyset$
2: if $|\overline{x}| = 1$ then
3:     return Eval$(Q, G)$
4: $L \leftarrow$ GenericJoin$(Q_{\overline{x}_1}, G)$
5: for $\mathbf{t}_{\overline{x}_1} \in L$ do
6:     for $i \in [\ell + 1, k]$ do
7:         if $y_i \in \overline{x}_1$ then                         ▷ processing $r_i(y_i, z_i)$
8:             $r_i[v] \leftarrow \{v' \mid (v, v') \in [\![r_i]\!]_G\}$
9:             Replace $r_i(y_i, z_i)$ in $\hat{Q}$ for $r_i[v](z_i)$
10:        else                                              ▷ bipartite implies $z_i \in \overline{x}_1$
11:            $r_i[v] \leftarrow \{v' \mid (v', v) \in [\![r_i]\!]_G\}$
12:            Replace $r_i(y_i, z_i)$ in $\hat{Q}$ for $r_i[v](y_i)$
13:        $\hat{G} \leftarrow G \cup r_i[v]$
14:    $A[\mathbf{t}_{\overline{x}_1}] \leftarrow$ GenericJoin$(\hat{Q}, \hat{G})$
15:    $A \leftarrow A \cup \{\mathbf{t}_{\overline{x}_1}\} \times A[\mathbf{t}_{\overline{x}_1}]$
16: return $A$
```

$G_r(Q)$ the RPQ-graph of Q. Assume that Q is RPQ-bipartite and let $\overline{x}_1, \overline{x} - \overline{x}_1$ be a bipartiton of the RPQ-graph of Q. Then evaluating Q over a graph database G can be done via Algorithm 2, which generalizes GenericJoinCRPQ so that it takes the advantage of the bipartite structure of Q. Here for a CRPQ Q, and a set of variables $\overline{x}_1$, with $Q_{\overline{x}_1}$ we denote the CRPQ Q restricted to conjuncts using only the variables in $\overline{x}_1$. Notice that, given that $\overline{x}_1$ partitions the RPQ-graph of Q, the query $Q_{\overline{x}_1}$ contains only relations and no RPQs.

Algorithm GenericJoinCRPQ-Bipartite generalizes Algorithm 1 by taking the first partition of vertices to be a partition that forms a bipartition in the RPQ-graph of the query. This allows us to instantiate the starting vertices from which all the RPQs in Q will be computed. Intuitively, the existence of a bipartition in the RPQ-graph of the query allows us to divide the query into two subqueries with no RPQs and by this avoid having to compute nested RPQs.

In order to show that the algorithm is correct and to analyse its running time, we decompose the algorithm in three parts:

1. First, we compute the tuples $\mathbf{t}_{\overline{x}_1}$ in the answer of $Q_{\overline{x}_1}$ using the relational GenericJoin (line 4).
2. For every tuple $\mathbf{t}_{\overline{x}_1}$ we compute all the associated regular expressions (lines 5–13).
3. We compute the rest of the join (involving the variables in $\overline{x} - \overline{x}_1$ with the relational GenericJoin (line 14).

In the worst case, we must perform $\mathrm{AGM}(Q_{\overline{x}_1}, G_{\overline{x}_1})$ computations of every regular expression r_i. Therefore, the total cost is in $O(\mathrm{AGM}(Q_{\overline{x}_1}, G_{\overline{x}_1}) \times |V|^2)$ (the $|V|^2$ being the cost of computing the RPQs). Next, we also need to evaluate the remaining (conjunctive) query over variables $\overline{x} - \overline{x}_1$. This takes time in $O(\mathrm{AGM}(Q_{\overline{x}-\overline{x}_1}, G_{\overline{x}-\overline{x}_1}))$. We obtain the following.

▶ **Theorem 12.** *Let $Q(\overline{x})$ be a CRPQ such that its RPQ-graph is bipartite, and let $\overline{x}'$, $\overline{x}''$ be an RPQ-bipartition, with $|\overline{x}'| \leq |\overline{x}''|$. Then the running time of* GenericJoinCRPQ-Bipartite *over Q and a graph $G = (V, E)$ is*

$$AGM(Q_{\overline{x}'}, G) \cdot |V|^2 + AGM(Q_{\overline{x}''}, G).$$

In order to reach the running time of FullMaterialization we need the query to be even further restricted. In particular, if the bipartition is such that one side contains a single variable, then the algorithm is equivalent to fixing a vertex in this variable, computing all the RPQs in Q from this vertex (by the property of bipartition, no other vertex exists), and then joining the rest using GenericJoin. This gives us the following.

▶ **Corollary 13.** *When the RPQ-graph of a CRPQ Q is bipartite and it admits a partition $\overline{x}'$, $\overline{x}''$ with $\min\{|\overline{x}'|, |\overline{x}''|\} = 1$, the running time of GenericJoinCRPQ-Bipartite is equal to FullMaterialization.*

Hence, for these types of CRPQs we can achieve running time of FullMaterialization using only linear memory. It is not difficult to show that GenericJoinCRPQ-Bipartite does not run under the FullMaterialization bound when queries are not of this specific shape. In general, we conjecture that this bound (under memory constraints) is not attainable when graphs are not RPQ-bipartite; solving this problem opens up an interesting line of work into space-time tradeoffs for computing the answers of a CRPQ.

5 Conclusions and future work

Our paper provides techniques for understanding size bounds of CRPQs, and makes use of these techniques to inform better algorithms for evaluating CRPQs. Our work also opens up several lines of work regarding CRPQs, size bounds and algorithms. A first important problem is to verify that GenericJoinCRPQ-Bipartite works well in practice, and enjoys as big success as standard worst-case optimal algorithms in graph databases. Of course, moving beyond RPQ-bipartite queries would require either new algorithms, or proving that the bounds offered by GenericJoinCRPQ cannot be improved. Further, there are several questions regarding tight bounds for complex classes of queries. In particular, our bounds for CRPQs with ε or RPQs accepting ε are only shown for very structured graphs where all relations share the same vertices, and it would be good to show that the bound remains to hold under arbitrary cardinalities.

References

1 Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan L. Reutter, and Domagoj Vrgoc. Foundations of Modern Query Languages for Graph Databases. *ACM Comput. Surv.*, 50(5):68:1–68:40, 2017.

2 Diego Arroyuelo, Aidan Hogan, Gonzalo Navarro, Juan L. Reutter, Javiel Rojas-Ledesma, and Adrián Soto. Worst-case optimal graph joins in almost no space. In Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava, editors, *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, pages 102–114. ACM, 2021.

3 Diego Arroyuelo, Aidan Hogan, Gonzalo Navarro, and Javiel Rojas-Ledesma. Time-and space-efficient regular path queries on graphs. *arXiv preprint*, 2021. `arXiv:2111.04556`.

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 Pablo Barceló Baeza. Querying graph databases. In *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 175–188, 2013.

6 Angela Bonifati, Wim Martens, and Thomas Timm. An analytical study of large SPARQL query logs. *VLDB J.*, 29(2-3):655–679, 2020.

7 Katrin Casel and Markus L. Schmid. Fine-grained complexity of regular path queries. In *24th International Conference on Database Theory, ICDT 2021, March 23-26, 2021, Nicosia, Cyprus*, pages 19:1–19:20, 2021.

8 Michael J. Freitag, Maximilian Bandle, Tobias Schmidt, Alfons Kemper, and Thomas Neumann. Adopting worst-case optimal joins in relational database systems. *Proc. VLDB Endow.*, 13(11):1891–1904, 2020.

9 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.

10 Aidan Hogan, Cristian Riveros, Carlos Rojas, and Adrián Soto. A worst-case optimal join algorithm for SPARQL. In *The Semantic Web – ISWC 2019 – 18th International Semantic Web Conference, Auckland, New Zealand, October 26-30, 2019, Proceedings, Part I*, pages 258–275, 2019.

11 Thomas Neumann and Gerhard Weikum. Rdf-3x: a risc-style engine for rdf. *Proceedings of the VLDB Endowment*, 1(1):647–659, 2008.

12 Thomas Neumann and Gerhard Weikum. The rdf-3x engine for scalable management of rdf data. *The VLDB Journal*, 19(1):91–113, 2010.

13 Hung Q. Ngo, Christopher Ré, and Atri Rudra. Skew strikes back: new developments in the theory of join algorithms. *SIGMOD Rec.*, 42(4):5–16, 2013.

14 Dung Nguyen, Molham Aref, Martin Bravenboer, George Kollias, Hung Q Ngo, Christopher Ré, and Atri Rudra. Join processing for graph patterns: An old dog with new tricks. In *Proceedings of the GRADES'15*, pages 1–8. ACM, 2015.

15 Jena Team. TDB Documentation, 2021. URL: `https://jena.apache.org/documentation/tdb/`.

16 Todd L. Veldhuizen. Triejoin: A simple, worst-case optimal join algorithm. In Nicole Schweikardt, Vassilis Christophides, and Vincent Leroy, editors, *Proc. 17th International Conference on Database Theory (ICDT), Athens, Greece, March 24-28, 2014*, pages 96–106. OpenProceedings.org, 2014.

17 Virginia Vassilevska Williams and R Ryan Williams. Subcubic equivalences between path, matrix, and triangle problems. *Journal of the ACM (JACM)*, 65(5):1–38, 2018.

18 Peter T. Wood. Query languages for graph databases. *SIGMOD Rec.*, 41(1):50–60, 2012.

Uniform Reliability for Unbounded Homomorphism-Closed Graph Queries

Antoine Amarilli
LTCI, Télécom Paris, Institut Polytechnique de Paris, France

Abstract

We study the uniform query reliability problem, which asks, for a fixed Boolean query Q, given an instance I, how many subinstances of I satisfy Q. Equivalently, this is a restricted case of Boolean query evaluation on tuple-independent probabilistic databases where all facts must have probability 1/2. We focus on graph signatures, and on queries closed under homomorphisms. We show that for any such query that is *unbounded*, i.e., not equivalent to a union of conjunctive queries, the uniform reliability problem is #P-hard. This recaptures the hardness, e.g., of s-t connectedness, which counts how many subgraphs of an input graph have a path between a source and a sink.

This new hardness result on uniform reliability strengthens our earlier hardness result on probabilistic query evaluation for unbounded homomorphism-closed queries [2]. Indeed, our earlier proof crucially used facts with probability 1, so it did not apply to the unweighted case. The new proof presented in this paper avoids this; it uses our recent hardness result on uniform reliability for non-hierarchical conjunctive queries without self-joins [3], along with new techniques.

2012 ACM Subject Classification Theory of computation → Database query processing and optimization (theory)

Keywords and phrases Uniform reliability, #P-hardness, probabilistic databases

Digital Object Identifier 10.4230/LIPIcs.ICDT.2023.14

Related Version *Full Version*: `https://arxiv.org/abs/2209.11177` [1]

Funding Partially supported by the ANR project ANR-18-CE23-0003-02 ("CQFD").

Acknowledgements I am grateful to Mikaël Monet, Charles Paperman, and Martin Retaux for helpful discussions about this research. Thanks to the reviewers for their helpful feedback.

1 Introduction

A long line of research [14] has investigated how to extend relational databases with probability values. The most common probabilistic model, called *tuple-independent databases* (TID), annotates each fact of the input database with an independent probability of existence. The *probabilistic query evaluation* (PQE) problem then asks for the probability that a fixed Boolean query is true in the resulting product distribution on possible worlds. The PQE problem has been historically studied for conjunctive queries (CQs) and unions of conjunctive queries (UCQs). This study led to the dichotomy result of Dalvi and Suciu [5], which identifies a class of *safe UCQs* for which the problem can be solved in PTIME:

▶ **Theorem 1.1** ([5]). *Let Q be a UCQ. Consider the PQE problem for Q which asks, given a TID I, to compute the probability that Q holds on I. This problem is in PTIME if Q is safe, and #P-hard otherwise.*

This result has been extended in several ways, to apply to some queries featuring negation [6], disequality ($\neq$) joins [10], or inequality ($<$) joins [11]. More recently, two new directions have been explored. First, our work with Ceylan [2] extended the study from UCQs to the broader class of *homomorphism-closed* queries. This class captures recursive queries such as regular path queries (RPQs) or Datalog (without inequalities or negation). In [2], we focused on homomorphism-closed queries that were *unbounded*, i.e., not equivalent to a UCQ. We showed that PQE is #P-hard for *any* such query, though for technical reasons the result only applies to graphs, i.e., arity-two signatures. This extended the above dichotomy to the full class of homomorphism-closed queries (on arity-two signatures).

Second, the dichotomy has been extended from PQE to restricted problems which do not allow arbitrary probabilities on the TID. Kenig and Suciu [8] have shown that the dichotomy of [5] still held for the so-called *generalized model counting* problem, where the allowed probabilities on tuples are only 0 (the tuple is missing), 1/2, or 1; this is in contrast with the original proof of the dichotomy, which uses arbitrary probabilities. Our result in [2] already held for the generalized model counting problem. What is more, for a subclass of the unsafe queries, they showed that hardness still held for the *model counting problem*, where the probabilities are either 0 or 1/2. Independently, with Kimelfeld [3], we have shown hardness of the same problem for the incomparable class of non-hierarchical CQs without self-joins. Rather than model counting, we called this the *uniform reliability* (UR) problem, following the terminology in the work of Grädel, Gurevich, and Hirsch [7].

In our opinion, this uniform reliability problem is interesting even outside of the context of probabilistic databases: we simply ask, for a fixed query Q, given a database instance I, *how many* subinstances of I satisfy Q. The UR problem also relates to computing the *causal effect* and *Shapley values* in databases [13, 9, 3]. What is more, UR for homomorphism-closed queries captures existing counting problems on graphs, such as *st-connectedness* [15] which asks how many subgraphs of an input graph contain a path between a source and a sink.

The ultimate goal of these two lines of work would be to classify the complexity of *uniform reliability*, across *all homomorphism-closed queries*. Specifically, one can conjecture:

► **Conjecture 1.2.** *Let Q be a homomorphism-closed query on an arbitrary signature. The uniform reliability problem for Q is in PTIME if Q is a safe UCQ, and #P-hard otherwise.*

To establish this, there are three obstacles to overcome. First, in the case where Q is a UCQ, one would need to establish the hardness of UR for all unsafe UCQs, extending the work of Kenig and Suciu [8]. Second, when Q is unbounded, one would need to adapt the methods of [2] to apply to UR rather than PQE. Third, the methods of [2] would need to be extended from graph signatures to arbitrary arity signatures.

Result statement. In this paper, we address the second difficulty and show the following, which extends the main result of [2] from PQE to UR, and brings us closer to Conjecture 1.2:

► **Theorem 1.3** (Main result). *Let Q be an unbounded homomorphism-closed query on an arity-two signature. The uniform reliability problem for Q is #P-hard.*

The proof of this result has the same high-level structure as in [2], but there are significant new technical challenges to overcome. In particular, we now reduce from different problems, whose hardness rely (among other things) on the hardness of uniform reliability for the query $R(x), S(x, y), T(y)$, shown in [3]. The impossibility to assign a probability of 1 to facts also makes reductions much more challenging: intuitively, as all facts can now be missing, there is no longer a clear connection between the possible worlds of the source problem and the possible worlds of the database built in the reduction. We use multiple tools to work around this, for instance a *saturation* technique that creates a large but polynomial number of copies of some facts and argues that their absence is sufficiently unlikely to be negligible. As saturation cannot apply to unary facts, we also need to identify so-called *critical models*, a more elaborate variant of a notion in [2], minimizing carefully-chosen weight criteria.

We give a high-level structure of the proof below as it is presented in the rest of the paper, and comment in more detail on how the techniques relate to our earlier work [2].

Paper structure. We give preliminaries and the formal definition of UR in Section 2, along with the two problems from which we reduce: one problem on bipartite graphs from [3], and one variant of a connectivity problem of [15]. We show that they are #P-hard in [1].

We then review notions from [2] in Section 3: the *dissociation* operation on instances, and the notion of a *tight edge*, which makes the query false when we apply dissociation to it. We invoke a result from [2] showing that tight edges always exist for unbounded queries. This is the only place where we use the unboundedness of the query, and is unfortunately the only result from [2] that can be used as-is. Some other notions are reused and extended from [2] but they are always re-defined and re-proved in a self-contained way in the present paper.

We then present in Section 4 the notion of a *critical model*, as a model of the query which is *subinstance-minimal* and features a tight edge which is minimal by optimizing three successive quantities: *weight*, *extra weight*, and *lexicographic weight*. The notion of *weight* is from [2], the two other notions relate to *side weight* from [2] but significantly extend it. We show in this section that a query having a model with a tight edge also has a critical model.

We then move on to the hardness proof. As in [2], there are two cases: a *non-iterable* case where we reduce from the problem on bipartite graphs, and an *iterable* case where we reduce from the connectivity problem. In Section 5, we formally define the notion of iteration (essentially identical to the notion in [2]) and show hardness when there is a non-iterable critical model. The coding used in the reduction extends that of [2] with the *saturation* technique of creating a large number of copies of some elements. There are many new technical challenges, e.g., proving that a polynomial number of copies suffices to make the absence of the facts sufficiently unlikely, and justifying that all the other facts are "necessary" for a query match, using in particular subinstance-minimality and the notion of extra weight.

Last, in Section 6, we show hardness in the case where all critical models are iterable. We first show that such models can be repeatedly iterated, and that the measure of *extra weight* must be zero in this case, allowing us to focus on the more precise criterion of lexicographic weight. Then we define the coding, which is similar to [2] up to technical modifications. The reduction does not use saturation but argues that all facts are "necessary" using the notion of lexicographic weight and a new *explosion* structure.

We then conclude in Section 7. The complete proofs are given in the full version [1].

2 Preliminaries and Problem Statement

Instances. We consider an *arity-two relational signature* σ consisting of *relations* with an associated *arity*, where the maximal arity of the signature is assumed to be 2. A σ-*instance* (or just *instance*) is a set of *facts*, i.e., expressions of the form $R(a, b)$ where a and b are constants and $R \in \sigma$. We assume without loss of generality that all relations in σ are binary, i.e., have arity two. Indeed, if there are unary relations U, we can simply code them with a binary relation U', replacing facts $U(a)$ by $U'(a, a)$ in instances, and modifying the query to interpret $U'(a, a)$ as $U(a)$ and to ignore facts $U'(a, b)$ with $a \neq b$: this is similar to Theorem 8.4 of [2]. Accordingly, we call a fact $R(a, b)$ *unary* if $a = b$, otherwise it is *binary*.

The *domain* $\text{dom}(I)$ of an instance I is the set of constants occurring in I. A *homomorphism* from I to an instance I' is a function $h\colon \text{dom}(I) \to \text{dom}(I')$ such that, for each fact $R(a, b)$ of I, the fact $R(h(a), h(b))$ is in I'. We say that I' is a *subinstance* of I, written $I' \subseteq I$, if I' is a subset of the facts of I; we then have $\text{dom}(I') \subseteq \text{dom}(I)$.

Queries. A *query* Q over σ is a Boolean function over σ-instances which we always assume to be *homomorphism-closed*, i.e., if Q returns true on I and I has a homomorphism to an instance I' then Q also returns true on I'. When Q returns true on I we call I a *model* of Q, or say that I *satisfies* Q (written $I \models Q$); otherwise I *violates* Q. Any homomorphism-closed query Q is *monotone*, i.e., if I satisfies Q and $I \subseteq I'$ then I' satisfies Q. A *subinstance-minimal model* of Q is a model I of Q such that no strict subinstance of I satisfies Q.

We focus on *unbounded queries*, i.e., queries having an infinite number of subinstance-minimal models. Examples of well-studied homomorphism-closed query languages include *conjunctive queries* (CQs), *unions of CQs* (UCQs), *regular path queries* (RPQs), and *Datalog* without inequalities or negations. The queries defined by Datalog or RPQs are unbounded unless they are equivalent to a UCQ (i.e., non-recursive Datalog); more generally a query is either unbounded or equivalent to a UCQ.

UR and PQE problems. In this paper, we study *uniform reliability* (UR). The problem UR(Q) for a fixed query Q is the following: we are given as input an instance I, and we must return how many subinstances of I satisfy Q, i.e., the number $|\{I' \subseteq I \mid I' \models Q\}|$. Note that we have no general upper bound on the complexity of this problem, as we allow queries to be arbitrarily complex or even undecidable to evaluate, e.g., "there is a path $R(x_1), S(x_1, x_2), \ldots, S(x_{n-1}, x_n), T(x_n)$ where n is the index of a Turing machine that halts".

We will sometimes consider the generalization of UR called *probabilistic query evaluation* (PQE). The PQE(Q) problem for a fixed query Q asks, given an instance I and a probability distribution $\pi \colon I \to [0,1]$ mapping each fact of I to a rational in $[0,1]$, to determine the total probability of the subinstances of I satisfying Q, when each fact $F \in I$ is drawn independently from the others with the probability $\pi(F)$. Formally, we must compute:

$$\sum_{I' \subseteq I \text{ s.t. } I' \models Q} \prod_{F \in I'} \pi(F) \times \prod_{F' \in I \setminus I'} (1 - \pi(F)).$$

The UR problem is a special case of PQE where the function π maps all facts to $1/2$, up to renormalization, i.e., multiplying by $2^{|I|}$. We will sometimes abusively talk about UR as the problem of computing that probability, because this probabilistic phrasing makes it more convenient, e.g., to reason about conditional probabilities, or about negligible probabilities.

Hard problems. The goal of this paper is to show Theorem 1.3. We will establish #P-hardness using *polynomial-time Turing reductions* [4] (see [2] for details). Specifically, we reduce from one of two #P-hard problems, depending on the query. In [2], we reduce from the problems #PP2DNF and U-ST-CON (undirected source-to-target connectivity), which are shown to be #P-hard in [12]. In this paper, given our focus on UR, we reduce from variants of these problems: the *λ, μ, ν-variable-clause-variable probabilistic #PP2DNF problem* and the *ϕ, η-vertex-edge probabilistic U-ST-CON problem.* We first define the first problem:

▶ **Definition 2.1.** *Let $0 < \lambda, \nu < 1$ and $0 < \mu \leq 1$ be fixed probabilities. The λ, μ, ν-*variable-clause-variable probabilistic #PP2DNF problem *(or for brevity λ, μ, ν-*#PP2DNF*) is the following: given a bipartite graph $(U \cup V, E)$ with $E \subseteq U \times V$, we ask for the probability that we keep an edge and its two incident vertices, where vertices of U have probability λ to be kept, edges of E have probability μ to be kept, and vertices of V have probability ν to be kept, all these choices being independent. Formally, we must compute:*

$$\sum_{\substack{(U', E', V') \subseteq U \times E \times V \\ E' \cap (U' \times V') \neq \emptyset}} \lambda^{|U|'} \times (1-\lambda)^{|U| - |U|'} \times \mu^{|E|'} \times (1-\mu)^{|E| - |E|'} \times \nu^{|V|'} \times (1-\nu)^{|V| - |V|'}$$

The name #PP2DNF is because of the link to positive partitioned 2-DNF formulas, which we do not need here. We can show that λ, μ, ν-#PP2DNF is #P-hard, by adapting the proof in [3] which shows the hardness of uniform reliability for the query $R(x), S(x, y), T(y)$:

▶ **Proposition 2.2** ([3]). *For any fixed $0 < \lambda, \nu < 1$ and $0 < \mu \leq 1$, the problem λ, μ, ν-#PP2DNF is #P-hard.*

We now define the second problem:

▶ **Definition 2.3.** *Let $0 < \phi \leq 1$ and $0 < \eta < 1$ be fixed probabilities. The ϕ, η-*vertex-edge-probabilistic U-ST-CON problem *(or for brevity ϕ, η-*U-ST-CON*) is the following: given an undirected graph $G = (V, E)$ and source and sink vertices $r, s \in V$ with $r \neq s$, we ask for the probability that we keep a subset of edges and vertices containing a path that connects r and s (in particular keeping r and s), where vertices have probability ϕ to be kept and edges have probability η to be kept, all these choices being independent. Formally, we must compute:*

$$\sum_{\substack{V' \subseteq V, E' \subseteq E \\ r \text{ and } s \text{ connected in } (V',\, E'_{|V'})}} \phi^{|V|'} \times (1 - \phi)^{|V| - |V|'} \times \eta^{|E|'} \times (1 - \eta)^{|E| - |E|'}$$

This intuitively combines features of the undirected source-to-target edge-connectedness and node-connectedness problems of [15]. With standard techniques and some effort, we can show that ϕ, η-U-ST-CON is #P-hard (see the full version [1]):

▶ **Proposition 2.4.** *For any fixed $0 < \phi \leq 1$ and $0 < \eta < 1$, the problem ϕ, η-U-ST-CON is #P-hard.*

3 Basic Techniques: Dissociation, Tight Edges

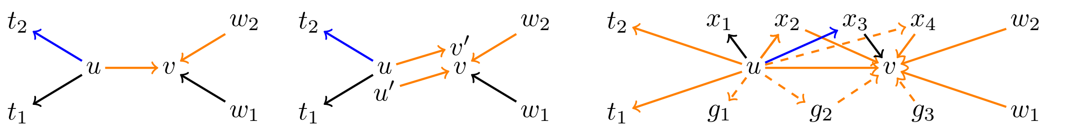
Having presented the hard problems, we now recall the notion of *edges* and how we *copy* them, and the *dissociation* operation introduced in [2]. We also present *tight edges* and re-state the result of [2] showing that unbounded queries have models with tight edges.

Edges and copies. An *edge* e in an instance I is an ordered pair (u, v) of distinct elements of $\text{dom}(I)$ such that there is at least one fact of I using both u and v, i.e., of the form $R(u, v)$ or $R(v, u)$, hence non-unary. The *covering facts* of e in I is the non-empty set of these facts. Note that (u, v) is an edge iff (v, u) is, and they have the same covering facts.

We call $e = (u, v)$ a *non-leaf edge* if I contains facts using u but not v (called *left-incident facts*) and facts using v but not u (called *right-incident facts*). An example is shown in Figure 1a (with no unary facts). The left-incident and right-incident facts are called together the *incident facts*; note that they may include unary facts.

In this paper we will often modify instances I by *copying* an edge $e = (u, v)$ of I to some other ordered pair (u', v') of elements. This means that we modify I to add, for each covering fact F of e, the fact obtained by replacing u by u' and v by v'. Note that, if u' and v' are both fresh, or if $u' = u$ and v' is fresh or $v' = v$ and u' is fresh, then the result of this process has a homomorphism back to I. Clearly, copying (u, v) on (u', v') is equivalent to copying (v, u) on (v', u') (but different from copying, say, (u, v) on (v', u')). Note that copying an edge does *not* copy its incident facts, though our constructions will often separately copy some of them.

▶ **Example 3.1.** In the instance $I = \{R(a), S(a, b), S'(b, a), T(b)\}$, copying (a, b) on (a, b') for a fresh element b' means adding the facts $S(a, b'), S'(b', a)$.



(a) A non-leaf edge e with incident facts.
(b) The dissociation of e.
(c) An edge and various kinds of incident facts. See Example 4.10.

Figure 1 Examples of Section 3 and 4.

Dissociation. One basic operation on instances is *dissociation*, which replaces one edge by two copies connected to each endpoint:

▶ **Definition 3.2.** *Let I be an instance and $e = (u, v)$ be a non-leaf edge of I. The* dissociation *of e in I is obtained by modifying I to add two fresh elements u' and v', copying e to (u', v) and to (u, v'), and then removing the covering facts of e.*

The process is illustrated in Figures 1a and 1b. Note the following immediate observation:

▷ Claim 3.3. The dissociation of an edge in I has a homomorphism back to I.

Tight edges. We can then define a *tight edge* as one whose dissociation breaks the query:

▶ **Definition 3.4.** *A non-leaf edge (u, v) in an instance I is* tight *for the query Q if I satisfies Q but the dissociation of (u, v) in I does not.*

We use a result of [2] which shows that unbounded queries must have a model with a tight edge. This is the only point where we use the unboundedness of the query.

▶ **Theorem 3.5** (Theorem 6.6 in [2]). *Any unbounded query has a model with a tight edge.*

We give a proof sketch for completeness (see [2] for the proof):

Proof sketch. As the query Q is unbounded, it has infinitely many minimal models: let I be a sufficiently large one. Iteratively dissociate the non-leaf edges of I until none remain (this always terminates), and let I' be the result. If I' violates Q, then some dissociation broke Q, i.e., was applied to a tight edge in a model of Q. Otherwise, I' has no non-leaf edges and satisfies Q. We can then show thanks to the simple structure of I' that it has a constant-sized subset that satisfies Q, and deduce that Q already holds on a constant-sized subinstance of I. As I is large, this contradicts the minimality of I. ◀

Thus, in the sequel, we fix the query Q and assume that it has a model with a tight edge. Note that some bounded queries may also have a tight edge, e.g., the prototypical unsafe CQ $R(x), S(x, y), T(y)$; our results in this paper thus also apply to some bounded queries.

4 Minimality and Critical Models

In this section, we refine the notion of a tight edge to impose minimality criteria and get to the notion of *critical models*. We define three successive minimality criteria, which we present intuitively here before formalizing them in the rest of this section. The first is called *weight* and counts the covering facts; the *critical weight* Θ is the minimal weight of a tight edge. Having defined Θ, we restrict our attention to *clean* tight edges e, whose incident facts do not include so-called *garbage facts*, i.e., strict subsets of the covering facts of e. The second criterion is *extra weight* and counts the incident facts that are not isomorphic to the covering facts; the *critical extra weight* Ξ is the minimal extra weight of a tight edge of weight Θ. The third criterion is *lexicographic weight* and counts the other left-incident and right-incident facts, ordered lexicographically: the *critical lexicographic weight* Λ is the minimal lexicographic weight of a tight edge of weight Θ and extra weight Ξ.

We then define a *critical model* as a *subinstance-minimal* model with a clean tight edge that optimizes these three weights in order, and show that such models exist.

Weight. The *weight* was defined in [2], but unlike in [2] we do not count unary facts:

▶ **Definition 4.1.** *The* weight *of an edge $e = (u, v)$ in an instance I is the number of covering facts of e (it is necessarily greater than 0).*

▶ **Example 4.2.** The weight of (a, b) in $I = \{R(b), T(b, c), S(b, a), S'(b, a), U(a, b)\}$ is 3.

The minimal weight of a tight edge across all models is an intrinsic characteristic of Q, called the *critical weight*:

▶ **Definition 4.3.** *The* critical weight *of the query Q, written $\Theta \geq 1$, is the minimum, across all models I of Q and tight edges e of I, of the weight of e in I.*

The point of the critical weight is that edges with weight less than Θ can never be tight:

▷ Claim 4.4. Let I be a model of Q and $e = (u, v)$ be a non-leaf edge of I. If the weight of e is less than Θ, then the dissociation of e in I is also a model of Q.

▶ **Example 4.5.** The bounded CQ $Q' : R(x), S(x, y), S'(x, y), T(y)$ has critical weight 2, as witnessed by the model $I' = \{R(a), S(a, b), S'(a, b), T(b)\}$ with a tight non-leaf edge (a, b) of weight 2 and the inexistence of a model with a tight non-leaf edge of weight 1.

As Q' has critical weight 2, in any model I of Q', if we have an edge $e = (u, v)$ with only one covering fact using both u and v, we know that dissociating e cannot make Q' false.

Having defined Θ, to simplify further definitions, we introduce the notion of a *clean* edge as one that does not have incident facts achieving strict subsets of its covering facts:

▶ **Definition 4.6.** *Let I be an instance, let $e = (u, v)$ be an edge of I, and let $C \subseteq I$ be the covering facts of e. For any edge (u, t), if its covering facts are isomorphic to a strict subset of C when renaming t to v, then we call these left-incident facts* left garbage facts*. Likewise, the* right garbage facts *are the right-incident facts that are covering facts of edges (w, v) that are isomorphic to a strict subset of C when renaming w to u.*

We call e clean *if it has no left or right garbage facts (called collectively* garbage facts*).*

▶ **Example 4.7.** In the instance $I = \{S(a, b'), U(a), S(a, b), S'(b, a), T(c, b), S(c, b), S'(d, b), S'(b, e), S(f, b)\}$, the left garbage facts of the edge (a, b) are $\{S(a, b')\}$ on the edge (a, b'), and the right garbage facts are $\{S'(b, e)\}$ on the edge (e, b) and $\{S(f, b)\}$ on the edge (f, b). Note that there are no garbage facts on the edge (b, c), because the covering facts $\{T(c, b), S(c, b)\}$ of this edge are not isomorphic to a strict subset of the covering facts of (a, b). Further note that there are no garbage facts on the edge (d, b), because the covering facts $\{S'(d, b)\}$ are not isomorphic to a strict subset of the covering facts of (a, b) when renaming d to a.

We will always be able to ensure that tight edges with critical weight are clean, justifying that we restrict our attention to clean tight edges in the sequel:

▷ Claim 4.8. If Q has a model with a tight edge, then it has a model with a clean tight edge of weight Θ.

Proof sketch. We find a model with a tight edge of weight Θ by definition of Θ. Then, any edges with garbage facts have weight $< \Theta$, so they can be dissociated using Claim 4.4 and homomorphically merged to e. At the end of this process, e is clean and is still tight. ◁

Extra weight. We further restrict tight edges e by limiting their number of incident facts, similarly to the notion of *side weight* in [2]. However, in this paper, we additionally partition the incident facts between so-called *extra facts* and *copy facts*. Intuitively, our reductions will use codings that introduce copies of the edge e, and the *extra facts* are those that can be "distinguished" from incident copies of e added in codings; by contrast copy facts are non-unary facts in edges that are isomorphic copies of e and therefore "indistinguishable".

We want to minimize the number of extra facts, to intuitively ensure that they are all "necessary", in the sense that a copy of e missing an incident extra fact can be dissociated. Let us formally define the extra facts: among the non-garbage incident facts, they are those that are part of a so-called *triangle* (i.e., involve an element occurring both in a left-incident in a right-incident fact), those which are unary, or those which are a covering fact of an edge whose covering facts are not isomorphic to the covering facts of e.

▶ **Definition 4.9.** *Let I be an instance with an edge $e = (u, v)$, and let $C \subseteq I$ be the covering facts of e. An element $w \in \mathrm{dom}(I)$ forms a* triangle *with e if both (u, w) and (v, w) are edges.*

Let (u', v') be some edge of I. We call (u', v') a copy *of (u, v) if the covering facts of (u', v') are isomorphic to C by the isomorphism mapping u' to u and v' to v.*

We partition the non-garbage left-incident facts of (u, v) between:

- *The* left copy facts*, i.e., the binary facts involving u and an element v' such that (u, v') is a copy of (u, v) and v' does* not *form a triangle with e: we call v' a* left copy element *of e.*
- *The* left extra facts*, which comprise all other non-garbage left-incident facts, namely:*
  - *The unary facts on u.*
  - *The non-garbage binary facts involving u and some element x such that:*
    - *the element x forms a triangle with e; or*
    - *the covering facts of the edge (u, x) are not isomorphic to C.*

We partition the non-garbage right-incident facts into right extra facts *and* right copy facts *with* right copy elements *in a similar way. Note that, as we prohibit triangles, the left copy elements and right copy elements are disjoint. We talk of the* copy elements*,* copy facts*,* extra facts *of e to denote both the left and right kinds.*

▶ **Example 4.10.** Consider the instance of Figure 1c and the edge $e = (u, v)$. The covering facts C of e are represented as an orange edge, and the other orange edges represent edges which are copies of e. The left and right copy elements are respectively t_1 and t_2 and w_1 and w_2. The dashed orange edges represent edges whose covering facts are a strict subset of C, i.e., they are garbage facts. The extra facts include unary facts (not pictured), facts with x_1 (the black edge (u, x_1) represents non-garbage facts not isomorphic to C), and facts with x_2, x_3, and x_4 (which form triangles).

Note that garbage facts are neither extra facts nor copy facts, and are ignored in the definition above except in that they may help form triangles. This does not matter: thanks to Claim 4.8, garbage facts will only appear in intermediate steps of some proofs. We can now define the *critical extra weight* as the minimal extra weight of a tight edge with weight Θ:

▶ **Definition 4.11.** *The* critical extra weight *of Q, written $\Xi \geq 0$, is the minimum across all models I of Q and tight edges e of I of weight Θ, of the number of extra facts of e in I.*

▶ **Example 4.12.** Continuing Example 4.5, the query Q' had critical extra weight 2, as witnessed by I'. The query $Q'' : R(x), S(x, y), S(x', y), S(x', y'), T(y')$, has critical weight 1 and critical extra weight 0, as witnessed by the model $I'' = \{R(a), S(a, b), S(a', b), S(a', b'), T(b')\}$ where the edge (a', b) is tight and has weight 1 and extra weight 0.

Again, the definition of critical extra weight clearly ensures:

▷ Claim 4.13. Let I be a model of Q and $e = (u, v)$ be a non-leaf edge. If e has weight Θ and extra weight $< \Xi$, then the dissociation of e in I is also a model of Q.

Lexicographic weight. We then impose a third minimality requirement on tight edges e, which is needed in Section 6 (but unused in Section 5). The intuition is that we want to limit the number of copy elements. Specifically, we minimize first the number τ of left copy elements, then the number ω of right copy elements, hence the name *lexicographic weight*. This is why, when choosing a tight edge, we also choose an orientation (i.e., choosing (u, v) as a tight edge is different from choosing (v, u)):

▶ **Definition 4.14.** *Let I be an instance with an edge $e = (u, v)$. Let τ be the number of left copy elements and ω be the number of right copy elements of e. The* lexicographic weight *of e is the ordered pair (τ, ω). We order these ordered pairs lexicographically, i.e., $(\tau, \omega) < (\tau', \omega')$ with $\tau, \tau', \omega, \omega' \in \mathbb{N}$ iff $\tau < \tau'$ or $\tau = \tau'$ and $\omega < \omega'$.*

The critical lexicographic weight *Λ of Q is the minimum, over all models I of Q and all tight edges of e with weight Θ and extra weight Ξ, of the lexicographic weight of e.*

Note that minimizing the lexicographic weight does not always minimize the total number of copy facts[1], e.g., $(1, 3) < (2, 1)$ but $1 + 3 > 2 + 1$. However, it is always the case that removing a copy fact of an edge e causes the lexicographic weight of e to decrease (and does not cause the extra weight to increase, as the remaining covering facts of the edge are garbage facts).

Again, we have:

▷ Claim 4.15. Let I be a model of Q and $e = (u, v)$ be a non-leaf edge with weight Θ, extra weight Ξ, and lexicographic weight $< \Lambda$. Then, the dissociation of e in I is also a model of Q.

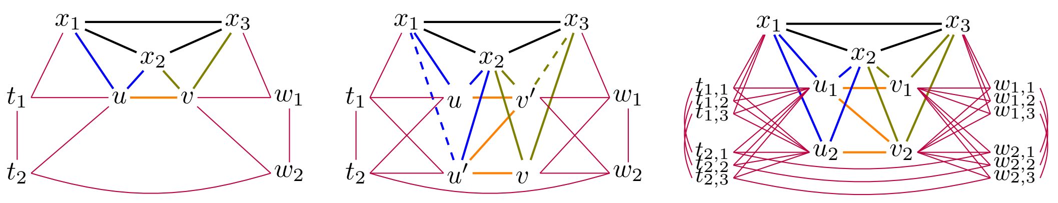
Critical models. We now define *critical models* (significantly refining the so-called *minimal tight patterns* of [2]). A critical model I is intuitively a model of Q with a clean tight edge e that achieves the minimum of our three weight criteria, and where we additionally impose that I is subinstance-minimal. For convenience we also specify a choice of incident facts in the critical model, but this choice is arbitrary, i.e., we can pick any pair of a left-incident fact and right-incident fact.

▶ **Definition 4.16.** *A* critical model *$(I, e, F_{\mathrm{L}}, F_{\mathrm{R}})$ is a model I of Q which is subinstance-minimal, a clean tight edge e of I having weight Θ, extra weight Ξ, and lexicographic weight Λ, and a left-incident fact $F_{\mathrm{L}} \in I$ and a right-incident fact $F_{\mathrm{R}} \in I$ of e.*

We can now claim that critical models exist:

▶ **Proposition 4.17.** *If a query Q has a model with a tight edge, then it has a critical model.*

[1] Minimizing the total number of copy facts, or minimizing along the componentwise partial order on $\mathbb{N} \times \mathbb{N}$, would suffice almost everywhere in the proof except in part of Section 6.



(a) Example critical model M. **(b)** Iteration of M. **(c)** 3-saturated coding $I_{G,3}$ in M of $G = (\{1,2\}, \{(1,1),(1,2),(2,2)\})$.

Figure 2 Examples of Section 5 and illustration of the notation.

Proof sketch. The existence of models with tight edges achieving the critical weights is by definition, cleanliness can be imposed by the process used to prove Claim 4.8, and subinstance-minimality can easily be imposed by picking some minimal subset of facts of the model that satisfy the query. ◀

5 Hardness with a Non-Iterable Critical Model

Having defined critical models, we now start our hardness proof. As in [2], we will distinguish two cases, based on whether we can break Q with an *iteration* process on a critical model.

▶ **Definition 5.1.** *Let $M = (I, e, F_\mathrm{L}, F_\mathrm{R})$ be a critical model, let $e = (u, v)$, and let C be the covering facts of e. Let A and B be the set of the left-incident and right-incident facts of e in I, respectively. The* iteration *of M is obtained by modifying I in the following way:*
- *Add fresh elements u' and v', copy e on (u, v'), (u', v'), (u', v), and remove the facts of C.*
- *Create a copy of the facts of $A \setminus \{F_\mathrm{L}\}$ where we replace u by u'.*
- *Create a copy of the facts of $B \setminus \{F_\mathrm{R}\}$ where we replace v by v'.*

▶ **Example 5.2.** Consider the critical model in Figure 2a, with edge (u, v) and where F_L and F_R are binary facts respectively using u and x_1 and v and x_3. Its iteration is shown in Figure 2b, with dashed edges representing edges where F_L and F_R are missing.

A *non-iterable* critical model M is one whose iteration no longer satisfies the query; otherwise M is *iterable*. In this section, we show hardness when there is a non-iterable critical model:

▶ **Proposition 5.3.** *Assume that Q has a non-iterable critical model. Then the uniform reliability problem for Q is #P-hard.*

We prove this result in the rest of this section.

Fixing notation. Fix the critical model $M = (I, e, F_\mathrm{L}, F_\mathrm{R})$ and let $e = (u, v)$ be the tight clean edge. We must introduce some notation to talk about the incident facts of e in I, which is summarized in Figure 2a. As e is clean, we know that its incident facts are either extra facts or copy facts – there are no garbage facts.

Let $C \subseteq I$ be the covering facts of e in I (in orange on the picture), with $|C| = \Theta$. Let $X = \{x_1, \ldots, x_k\}$ be the elements different from u and v with which one of u or v has a (non-unary) extra fact or has one of the two facts F_L and F_R. Note that some of the elements in X may have facts with both u and v (i.e., triangles), like x_2 in the picture. We may have $k = 0$, specifically when F_L and F_R are unary facts and any other extra facts are unary.

Further let $T = \{t_1, \dots, t_\tau\}$ be the left copy elements of e not in X, and let $W = \{w_1, \dots, w_\omega\}$ be the right copy elements of e not in X, with T and W disjoint (because copy elements cannot form triangles). We exclude elements of X because, if F_L (resp., F_R) is a copy fact, then X contains exactly one left copy element (resp., exactly one right copy element)[2]. Also note that we may have $\tau = \omega = 0$, i.e., if there are no copy facts except possibly those of the edges of F_L and of F_R.

To recapitulate, the incident facts of e in I only involve elements from $X \sqcup T \sqcup W$. Specifically, they are the unary facts on u, the unary facts on v, the non-unary extra facts (which involve one of $\{u, v\}$ and one element of X), the facts F_L and F_R which respectively involve u and v and (if they are non unary) one element of X, and the other left and right copy facts forming isomorphic copies of e as edges (u, t_j) with $1 \leq j \leq \tau$ and (w_i, v) with $1 \leq i \leq \omega$. Notice again how, if F_L or F_R are copy facts, then these notations handle them as extra facts along with any other covering facts of their edge. Note that our description of the incident facts of e does not describe the facts that may exist between elements of $X \sqcup T \sqcup W$, and indeed these may be arbitrary (some are pictured in Figure 2a).

Coding bipartite graphs. We will reduce from our variant of #PP2DNF (Definition 2.1) by using M to code a bipartite graph $G = (U \sqcup V, E)$. Intuitively, we will create one copy u_i of u for each vertex i of U, one copy v_j of v for each vertex j of V, and copy the edge e on (u_i, v_j) for each edge (i, j) of E. The reason why we distinguish X and T and W is because we will handle them differently. For the incident facts of e that are unary or involve elements of X, we will create one single copy of them for each u_i and each v_j. Indeed, we will show that edges (u_i, v_j) that are missing one such incident fact can be dissociated (if an extra fact is missing, using Claim 4.13) or mapped in a specific way in the iteration (if one of F_L or F_R is a copy fact and we are missing one of the covering facts of their edge). For the (copy) facts involving $T \sqcup W$, we will copy them (using the fact that they are binary) by creating a large number q of copies of $T \sqcup W$. This *saturation* process will in fact create a large number of copies of all facts involving some element of $T \sqcup W$, which we call the *saturated facts*.

Let us accordingly define the *saturated coding* of a bipartite graph in M:

▶ **Definition 5.4.** *Let $G = (U \sqcup V, E)$ be a non-empty bipartite graph, and assume without loss of generality that $U = \{1, \dots, n\}$ and $V = \{1, \dots, m\}$.*

Let $q > 0$ be some integer. The q-saturated coding *of G in M, written $I_{G,q}$, is the instance defined by modifying I in the following way:*

- *For all $1 \leq p \leq q$, create fresh elements $T_p = \{t_{1,p}, \dots, t_{\tau,p}\}$ and $W_p = \{w_{1,p}, \dots, w_{\omega,p}\}$. Identify $t_j = t_{j,1}$ for $1 \leq j \leq \tau$ and $w_i = w_{i,1}$ for $1 \leq i \leq \omega$.*
- *Letting Φ be the set of the saturated facts, for each $1 \leq p \leq q$, create a copy of Φ where each element t_j is replaced by $t_{j,p}$ and each element w_i is replaced by $w_{i,p}$.*
- *Create elements $u_1, \dots, u_n$ and $v_1, \dots, v_n$, where we identify $u = u_1$ and $v = v_1$.*
- *Create a copy of all incident facts of e for all u_i and v_j. Formally, let A and B be the set of the left-incident and right-incident facts of e* in the current model *(i.e., involving the $t_{j,p}$ and $w_{i,p}$): note that A (resp., B) contains in particular F_L (resp., F_R) and any unary facts on u (resp., on v). For each $1 \leq i \leq n$, create a copy of the facts of A replacing u by u_i, and for each $1 \leq j \leq m$ create a copy of the facts of B replacing v by v_j.*
- *Copy e (i.e., C) on (u_i, v_j) for each $(i, j) \in E$, and remove the facts of C if $(u_1, v_1) \notin E$.*

[2] Because of this, in general (τ, ω) may be less than the critical lexicographic weight Λ.

The saturated coding process is illustrated in Figure 2c. Note that the process is in polynomial time if the value q is polynomial in the size $|G|$ of the input bipartite graph.

Understanding the coding. Letting $G = (U \sqcup V, E)$ be a non-empty bipartite graph and writing $U = \{1, \ldots, n\}$ and $V = \{1, \ldots, m\}$, we study the coding $I_{G,q}$ to relate subsets of $I_{G,q}$ to subsets of $U \times E \times V$. For this, we partition the facts of $I_{G,q}$ in five kinds (see Figure 2c):

- The *base facts* (pictured in black), which are the facts that do not involve any of the elements $u_1, \ldots, u_n$, $v_1, \ldots, v_m$ or any element of $\bigsqcup_{1 \leq p \leq q} T_p \sqcup W_p$ (but they may involve elements of X). These facts are precisely the facts of I that do not involve the elements u or v or any element of $T \sqcup W$, and they are unchanged in the coding.
- The *saturated facts* (in purple), i.e., the facts involving some element of $T_p \sqcup W_p$ for some $1 \leq p \leq q$. These facts exist in q copies, and some (corresponding to facts of I between u or v and an element of $T \sqcup W$) have been further copied n times (if they involve u) or m times (if they involve v).
- The *non-saturated left-incident facts* (in blue) of each vertex $i \in U$, which are the facts which involve u_i and do not involve the $T_p \sqcup W_p$, i.e., are unary or involve an element of X. These facts include in particular one copy of F_{L}.
- The *non-saturated right-incident facts* (in green) of each vertex $j \in V$, that involve v_j and not the $T_p \sqcup W_p$, i.e., are unary or involve an element of X; they include one copy of F_{R}.
- The *copy of* e (in orange) for each edge $(i, j) \in E$, which is on the edge (u_i, v_j) of $I_{G,q}$.

The last three kinds are what we are interested in for the reduction, but the first two kinds need to be dealt with. We will show that the base facts must all be present to satisfy the query, and that each edge has some copy of the saturated facts with high probability.

Base facts. We say that a subinstance of $I_{G,q}$ is *well-formed* if all base facts are present, and *ill-formed* if at least one is missing. The following is easy to see by subinstance-minimality of I:

▶ **Proposition 5.5.** *The ill-formed subinstances do not satisfy the query.*

Hence, the number of subinstances of $G_{I,q}$ satisfying the query is the number of well-formed subinstances that do. Thus, in the sequel, we only consider well-formed subinstances.

Saturated facts. For the saturated facts, we will intuitively define *valid* subinstances where, for each ordered pair of vertices $(i, j) \in U \times V$, considering the copies u_i and v_j of u and v, there is a complete copy of the saturated facts that are "relevant" to them. More precisely, looking back at the original instance I, and considering the facts of I involving an element of $T \sqcup W$, there are of two types. The first type are the facts that do not involve u or v, i.e., they only involve elements of $T \sqcup W$ and possibly of $\mathrm{dom}(I) \setminus \{u, v\}$. Each such fact has been copied q times in $I_{G,q}$, and the copy numbered $1 \leq p \leq q$ uses one or two elements of $T_p \sqcup W_p$. The second type are the facts involving u or v in I (they cannot involve both). These facts have been copied $n \times q$ or $m \times q$ times in $I_{G,q}$, each copy using one element of $T_p \sqcup W_p$ for some $1 \leq p \leq q$ and one u_i for some $1 \leq i \leq n$ or one v_j for some $1 \leq j \leq m$. What we require of a valid subinstance $J \subseteq I_{G,q}$ is that, for each pair of vertices $(i, j) \in U \times V$, we have in J some copy $1 \leq p \leq q$ containing all facts of the first type and all facts of the second type involving u_i and v_j:

▶ **Definition 5.6.** *We partition the saturated facts of $I_{G,q}$ in q copies: formally, the p-th* saturated copy *for $1 \leq p \leq q$ is the subset of the saturated facts of $I_{G,q}$ that involve some element of $T_p \sqcup W_p$. A* saturation index *for $I_{G,q}$ is a function $\iota \colon U \times V \to \{1, \ldots, q\}$.*

For $J \subseteq I_{G,q}$, we say that J is valid *for ι if, for each $(i, j) \in U \times V$, letting $p := \iota(i, j)$, considering the facts of the p-th saturated copy, J contains all such facts that are:*

- *of the first type, i.e., J contains all facts of $I_{G,q}$ that involve some element of $T_p \sqcup W_p$ and do not involve any elements of $\{u_{i'} \mid 1 \leq i' \leq n\} \sqcup \{v_{j'} \mid 1 \leq j' \leq m\}$;*
- *of the second type and involve u_i or v_j, i.e., J contains all facts of $I_{G,q}$ that involve some element of $T_p \sqcup W_p$ and involve either u_i or v_j.*

We call J valid *if there is a saturation index for which it is valid; otherwise J is* invalid.

Note that, for each choice of ordered pair $(i, j) \in U \times V$, the required facts can be found in a different saturated copy $\iota(i, j)$, i.e., we do not require that there is a p such that J contains all facts of the p-th saturated copy. Indeed this stronger requirement would be too hard to ensure: intuitively, the number of facts required for each (i, j) is constant (it only depends on I), but the number of facts in the p-th saturated copy depends on G (it is linear in $|U| \times |V|$).

We now show that we can pick a number q of copies which is polynomial in the input G, but makes it very unlikely that a random subinstance is invalid. Thanks to this, we do not need to know which ones of the invalid subinstances satisfy Q. Indeed, the proportion of subinstances of $I_{G,q}$ that satisfy Q will be the proportion of valid subinstances that do, up to an error which is much less than the probability of any valid subinstance and can be eliminated by rounding:

▶ **Lemma 5.7.** *There is a polynomial P_M depending on the critical model M such that, for any non-empty bipartite graph $G = (U \sqcup V, E)$, letting $\chi := |U| + |V| + |E|$ be the size of G and defining $q := P_M(\chi)$, the proportion of subinstances of $I_{G,q}$ that are invalid is strictly less than $2^{-(\chi|I|+1)}$.*

Thanks to this, we focus on the well-formed subinstances J where we keep some subset of the saturated facts making J valid. We now fix q to the value of Lemma 5.7, and build $I_{G,q}$ in polynomial time in the input bipartite graph G (with the critical model M being fixed).

Good and bad subinstances. Let us now study the status of the last three kinds of facts:

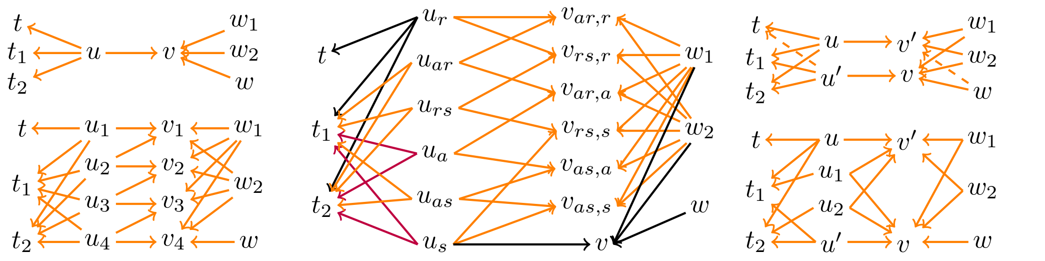
▶ **Definition 5.8.** *Let $J \subseteq I_{G,q}$. For $1 \leq i \leq n$ (resp., $1 \leq j \leq m$), the vertex $i \in U$ (resp., $j \in V$) is* complete *in J if all its non-saturated left-incident facts (resp., non-saturated right-incident facts) are present in J, and* incomplete *otherwise. The edge $(i, j) \in E$ is* complete *in J if all covering facts of (u_i, v_j) in $I_{G,q}$ are present in J, and* incomplete *otherwise. We call J* good *if there is an edge $(i, j) \in E$ with (i, j), i, and j complete, and* bad *otherwise.*

We now claim that, among the well-formed valid subinstances, the good ones satisfy the query, and the bad ones do not. This is easy to see for good subinstances:

▶ **Proposition 5.9.** *For any good valid well-formed subinstance $J \subseteq I_{G,q}$, there is a homomorphism from I to J.*

Proof sketch. As J is well-formed all base facts are present, and J is valid for some saturation index ι. Let $(i, j) \in E$ be an edge witnessing that J is good. The homomorphism maps $T \sqcup W$ to $T_{\iota(i,j)} \sqcup W_{\iota(i,j)}$, maps u to u_i and v to v_j, and is the identity otherwise. ◀

For bad subinstances, we show with much more effort that they do not satisfy the query:



(a) Example critical model M (top), 4-step iteration (bottom). **(b)** The coding I_G of a graph G in M: $G = (\{a, r, s\}, \{\{r, s\}, \{a, r\}, \{a, s\}\})$. **(c)** Fine dissociation (top) and explosion (bottom) of M.

Figure 3 Examples of Section 6 and illustration of the notation.

▶ **Proposition 5.10.** *Any bad subinstance $J \subseteq I_{G,q}$ does not satisfy the query.*

Proof sketch. It suffices to study the case with no saturation, i.e., $q = 1$. We dissociate incomplete edges with Claim 4.4, and dissociate complete edges missing at least one incident extra fact with Claim 4.13, which does not break Q. Then we show how to map this homomorphically to the iteration I' of M, by mapping complete vertices to u and v in the dissociation, and mapping the vertices which are missing facts of the edges of F_{L} or F_{R} to u' and v' respectively (after dissociating these edges if F_{L} or F_{R} are copy facts). This contradicts the assumption that M was non-iterable, i.e., that I' violates Q. ◀

This establishes that the status of Q on a valid well-formed subinstance J depends on whether J is good or bad, i.e., depends on which of the last three kinds of facts were kept in J. Now, the subsets of these facts are clearly in correspondence with the subsets of $U \times E \times V$ for the λ, μ, ν-#PP2DNF problem (see Definition 2.1), for some choice of constant probabilities λ, μ, ν. Further, a subset of $U \times E \times V$ is counted in λ, μ, ν-#PP2DNF if and only if the corresponding subset of the last three kinds of facts yields a good subinstance. As the ill-formed subinstances are easy to count, and the invalid ones are negligible, we can conclude the reduction and establish Proposition 5.3. The complete proof is given in [1].

6 Hardness when all Critical Models are Iterable

In this last section, we show hardness in the case where all critical models are iterable:

▶ **Proposition 6.1.** *Assume that Q has a critical model and that all critical models of Q are iterable. Then the uniform reliability problem for Q is #P-hard.*

A first observation is that, in this case, we have $\Xi = 0$, by contraposition of the following:

▷ Claim 6.2. If the critical extra weight is > 0, then Q has a non-iterable critical model.

Proof sketch. Take a critical model $M = (I, e, F_{\mathrm{L}}, F_{\mathrm{R}})$ with $e = (u, v)$ and one of $F_{\mathrm{L}}, F_{\mathrm{R}}$ an extra fact. The edge (u', v') in the iteration of M has weight Θ and extra weight $< \Xi$, so we can dissociate it without breaking Q and merge the two resulting copies. This yields the so-called *fine dissociation* (illustrated in Figure 3c, and formally defined in the full version of this paper [1]), which violates Q. ◁

Hence, in the rest of the section, we assume $\Xi = 0$, and fix an iterable critical model $M = (I, e, F_{\mathrm{L}}, F_{\mathrm{R}})$. All incident facts of $e = (u, v)$ in I are copy facts, so we let $t, t_1, \ldots, t_{\tau-1}$ be the left copy elements and $w, w_1, \ldots, w_{\omega-1}$ be the right copy elements, where t and w are the elements that occur in F_{L} and F_{R} respectively (the choice of F_{L} and F_{R} from now on only matters in that it distinguishes two copy elements t and w). The lexicographic weight of e in I is thus $\Lambda = (\tau, \omega)$ with $\tau, \omega \geq 1$. We let C be the covering facts of e in I. See Figure 3a.

n-step iteration. Let us now define the *n-step iteration* of M. It is related to iteration in [2], but specialized to the case where $\Xi = 0$, i.e., all incident facts are copy facts.

▶ **Definition 6.3.** *For $n > 0$, the* n-step iteration *of M is obtained by modifying I:*

- *Create elements $u_1, \ldots, u_n$ and $v_1, \ldots, v_n$, where we identify u and u_1 and v_n and v.*
- *For all $1 \leq i, j \leq n$, copy e on $(u_i, t_{j'})$ and $(w_{i'}, v_j)$ for all $1 \leq j' < \tau$ and $1 \leq i' < \omega$.*
- *For all $1 \leq i \leq n$, copy e on (u_i, v_i) for all $1 \leq i \leq n$ and on (u_{i+1}, v_i) for all $1 \leq i < n$.*
- *Remove the facts of C, except in the trivial case where $n = 1$.*

The iteration is illustrated in Figure 3a. Note that the 1-step iteration is exactly I. Further, the 2-step iteration resembles the iteration in Section 5, but omits some incomplete copies of (u, t) and (w, v) (i.e., the dashed edges in Figure 2b): as t and w are copy elements these facts would be garbage facts so the difference is inessential.

We now show that, if the iteration process of Section 5 cannot break Q on any critical model, then Q must also be satisfied in the n-step iteration of any critical model M for any $n > 0$. This proposition summarizes how we use the hypothesis that all critical models are iterable:

▶ **Proposition 6.4.** *Let Q be a query that has a critical model. Assume that all critical models for Q are iterable. Then $\Xi = 0$ and, for any critical model M of Q, for any $n > 0$, the n-iteration of M satisfies Q; further it is a subinstance-minimal model of Q.*

Proof sketch. Intuitively, the n-step iteration can be achieved by repeatedly performing the iteration from Section 5. A tedious point in the proof is to show that subinstance-minimality is preserved throughout this process. ◀

Coding. We explain how to code an undirected graph to reduce from ϕ, η-U-ST-CON for some $0 < \phi \leq 1$ and $0 < \eta < 1$ (see Definition 2.3): this time no saturation is needed. Proposition 6.4 will then intuitively show that some paths in the coding make Q true.

▶ **Definition 6.5.** *Let $G = (V, E)$ be an undirected graph with source r and sink s, with $r \neq s$. The* coding I_G *of G in M is the instance defined by modifying I in the following way:*

- *For all $a \in V$, create a fresh element u_a, and copy $(u, t_{j'})$ on $(u_a, t_{j'})$ for all $1 \leq j' < \tau$.*
- *We identify u to u_r, so u_r also occurs in another copy of e, namely the edge (u_r, t).*
- *For each edge $\pi = \{a, b\} \in E$, create fresh elements $u_\pi, v_{\pi,a}, v_{\pi,b}$, copy $(u, t_{j'})$ on $(u_\pi, t_{j'})$ for all $1 \leq j' < \tau$, copy $(w_{i'}, v)$ on $(w_{i'}, v_{\pi,\beta})$ for all $1 \leq i' < \omega$ and $\beta \in \{a, b\}$, and copy (u, v) on $(u_a, v_{\pi,a})$, $(u_\pi, v_{\pi,a})$, $(u_\pi, v_{\pi,b})$, and $(u_b, v_{\pi,b})$.*
- *Copy (u, v) on (u_s, v), and then remove the facts of C.*

An example is given in Figure 3b, shortening the vertex names for readability. The coding I_G can clearly be built in polynomial time in G. We partition the facts of I_G in four kinds:

- The *base facts* (not pictured), i.e., the facts involving no element of $\{u_a \mid a \in V\} \cup \{v_{\pi,\beta} \mid \pi \in E, \beta \in \pi\} \cup \{v\}$.
- The *supplementary base facts* (in black), i.e., the covering facts of (u_r, t) and $(u_r, t_{j'})$ for $1 \leq j' < \tau$, and the covering facts of (u_s, v) and (w, v) and $(w_{i'}, v)$ for $1 \leq i' < \omega$.

- The *vertex facts* (in purple) of each vertex $a \in V \setminus \{r\}$, i.e., the covering facts of $(u_a, t_{j'})$ for $1 \leq j' < \tau$.
- The *edge facts* (in orange) of each edge $\pi = \{a, b\}$ of E, i.e., all covering facts and incident facts of $(u_\pi, v_{\pi,a})$ and $(u_\pi, v_{\pi,b})$, including the covering facts of $(u_a, v_{\pi,a})$ and $(u_b, v_{\pi,b})$.

Similarly to Section 5, the base facts of I_G are precisely the facts of I that do not involve u or v. A subinstance $J \subseteq I_G$ is *well-formed* if it contains all base facts and supplementary base facts, and *ill-formed* otherwise. We can then use subinstance-minimality to show:

▷ Claim 6.6. The ill-formed subinstances do not satisfy the query.

Now, consider a well-formed subinstance $J \subseteq I_G$. A vertex $a \in V$ is *complete* in J if all vertex facts of a are present, and *incomplete* otherwise; and an edge $\pi \in E$ is *complete* in J if all its edge facts of π are present, and *incomplete* otherwise. A *complete path* in J is a path connecting r and s in G such that all traversed edges and vertices are complete in J (except r, for which completeness was not defined). We say that J is *good* if it has a complete path, and *bad* otherwise. We can easily see that good subinstances satisfy the query, because they contain an iterate of M and we can use Proposition 6.4:

▷ Claim 6.7. For any good well-formed subinstance $J \subseteq I_G$, there is a homomorphism from the $(2n + 1)$-step iteration of M to J, where n is the length of a complete path in J.

It is again far more challenging to show the other claim:

▷ Claim 6.8. Any bad subinstance $J \subseteq I_G$ does not satisfy the query.

Proof sketch. We dissociate all copies of e that are missing a fact or are of the form $(u_\beta, v_{\pi,\beta})$ and are missing an incident fact with some element $w_{i'}$. Then, we map the result by a homomorphism h to a structure called the *explosion* (illustrated in Figure 3c and formally defined in the full version of this paper [1]), which intuitively reflects all maximal strict subsets of the $\{t_1, \ldots, t_{\tau-1}\}$, and violates Q (by considering the lexicographic weight of its edges). We define h along the cut of G defined by considering the vertices reachable from r via a complete path. ◁

We then show hardness by reducing from ϕ, η-U-ST-CON for well-chosen constant probabilities ϕ and η (up to assuming that the source vertex r is always kept) and thus conclude the reduction, establishing Proposition 6.1. Together with Proposition 5.3, as Q has a critical model by Proposition 4.17 and Theorem 3.5, we have shown our main result (Theorem 1.3).

7 Conclusion

We have proved the intractability of uniform reliability for unbounded homomorphism-closed queries on arity-two signatures. We have not investigated the related problem of *weighted uniform reliability* [3], which is the restricted case of probabilistic query evaluation where we impose that all facts of the input TID must have some fixed probability different from 1/2. We expect that our hardness result should extend to this problem when the fixed probability is the same across all relations (and is different from 0 and 1). It seems more challenging to understand the setting where the fixed probability can depend on the relation, in particular if we can require some relations to be be deterministic, i.e., only have tuples with probability 1. In this setting, some unbounded homomorphism-closed queries would become tractable (e.g., Datalog queries that involve only the deterministic relations), and it is not clear what one can hope to show.

Coming back to the problem of (non-weighted) uniform reliability, an ambitious direction for future work would be to extend our intractability result towards Conjecture 1.2. The two remaining obstacles are the case of unbounded queries on arbitrary signatures, which we intend to study in future work; and the case of bounded queries, i.e., UCQs, where the general case is left open by Kenig and Suciu [8].

Other natural extensions include the study of queries satisfying weaker requirements than closure under homomorphisms; or other notions of possible worlds, e.g., induced subinstances; or other notions of intractability, e.g., the inexistence of lineages in tractable circuit classes from knowledge compilation. Another broad question is whether the techniques developed here have any connection to other areas of research, e.g., constraint satisfaction problems (CSPs).

References

1 Antoine Amarilli. Uniform reliability for unbounded homomorphism-closed graph queries. Full version with proofs, 2023. `arXiv:2209.11177`.

2 Antoine Amarilli and İsmail İlkan Ceylan. The dichotomy of evaluating homomorphism-closed queries on probabilistic graphs. *LMCS*, 2021. `arXiv:1910.02048`, `doi:10.46298/lmcs-18(1:2)2022`.

3 Antoine Amarilli and Benny Kimelfeld. Uniform reliability of self-join-free conjunctive queries. *LMCS*, 18(4), 2022. `arXiv:1908.07093`, `doi:10.46298/lmcs-18(4:3)2022`.

4 Stephen A. Cook. The complexity of theorem-proving procedures. In *Proc. STOC*, 1971. URL: `https://www.cs.toronto.edu/~sacook/homepage/1971.pdf`.

5 Nilesh N. Dalvi and Dan Suciu. The dichotomy of probabilistic inference for unions of conjunctive queries. *Journal of the ACM*, 59(6):30, 2012. URL: `https://homes.cs.washington.edu/~suciu/jacm-dichotomy.pdf`.

6 Robert Fink and Dan Olteanu. Dichotomies for queries with negation in probabilistic databases. *TODS*, 41(1), 2016. URL: `http://www.cs.ox.ac.uk/people/Dan.Olteanu/papers/fo-tods16.pdf`.

7 Erich Grädel, Yuri Gurevich, and Colin Hirsch. The complexity of query reliability. In *Proc. PODS*, 1998. URL: `https://www.researchgate.net/profile/Yuri_Gurevich2/publication/2900852_The_Complexity_of_Query_Reliability/links/0c96053321102376cd000000/The-Complexity-of-Query-Reliability.pdf`.

8 Batya Kenig and Dan Suciu. A dichotomy for the generalized model counting problem for unions of conjunctive queries. In *Proc. PODS*, 2021. `arXiv:2008.00896`.

9 Ester Livshits, Leopoldo E. Bertossi, Benny Kimelfeld, and Moshe Sebag. The Shapley value of tuples in query answering. In *Proc. ICDT*, volume 155, 2020. `doi:10.4230/LIPIcs.ICDT.2020.20`.

10 Dan Olteanu and Jiewen Huang. Using OBDDs for efficient query evaluation on probabilistic databases. In *Proc. SUM*, volume 5291, 2008. URL: `https://www.cs.ox.ac.uk/people/dan.olteanu/papers/oh-sum08.pdf`.

11 Dan Olteanu and Jiewen Huang. Secondary-storage confidence computation for conjunctive queries with inequalities. In *Proc. SIGMOD*, 2009. URL: `https://www.cs.ox.ac.uk/people/dan.olteanu/papers/oh-sigmod09.pdf`.

12 J. Scott Provan and Michael O. Ball. The complexity of counting cuts and of computing the probability that a graph is connected. *SIAM Journal on Computing*, 12(4), 1983.

13 Babak Salimi, Leopoldo E. Bertossi, Dan Suciu, and Guy Van den Broeck. Quantifying causal effects on query answering in databases. In *TAPP*, 2016. `arXiv:1603.02705`.

14 Dan Suciu, Dan Olteanu, Christopher Ré, and Christoph Koch. *Probabilistic databases*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2011.

15 Leslie Gabriel Valiant. The complexity of computing the permanent. *TCS*, 8(2):189–201, 1979. `doi:10.1016/0304-3975(79)90044-6`.

Approximation and Semantic Tree-Width of Conjunctive Regular Path Queries

Diego Figueira
Univ. Bordeaux, CNRS, Bordeaux INP, LaBRI, UMR5800, F-33400 Talence, France

Rémi Morvan
Univ. Bordeaux, CNRS, Bordeaux INP, LaBRI, UMR5800, F-33400 Talence, France

Abstract

We show that the problem of whether a query is equivalent to a query of tree-width k is decidable, for the class of Unions of Conjunctive Regular Path Queries with two-way navigation (UC2RPQs). A previous result by Barceló, Romero, and Vardi [5] has shown decidability for the case $k = 1$, and here we show that decidability in fact holds for any arbitrary $k > 1$. The algorithm is in 2ExpSpace, but for the restricted but practically relevant case where all regular expressions of the query are of the form a^* or $(a_1 + \cdots + a_n)$ we show that the complexity of the problem drops to Π_2^p.

We also investigate the related problem of approximating a UC2RPQ by queries of small tree-width. We exhibit an algorithm which, for any fixed number k, builds the maximal under-approximation of tree-width k of a UC2RPQ. The maximal under-approximation of tree-width k of a query q is a query q' of tree-width k which is contained in q in a maximal and unique way, that is, such that for every query q'' of tree-width k, if q'' is contained in q then q'' is also contained in q'.

2012 ACM Subject Classification Information systems → Query languages for non-relational engines; Theory of computation → Parameterized complexity and exact algorithms; Theory of computation → Database query processing and optimization (theory)

Keywords and phrases graph databases, conjunctive regular path queries, semantic optimization, tree-width, containment, approximation

Digital Object Identifier 10.4230/LIPIcs.ICDT.2023.15

Related Version *Full Version*: `https://arxiv.org/abs/2212.01679`

☞ This pdf contains internal links: clicking on a notion leads to its *definition.*[1]

1 Introduction

Graph databases are abstracted as edge-labeled directed graphs $G = \langle V(G), E(G)\rangle$, where nodes of $V(G)$ represent entities and labeled edges $E(G) \subseteq V(G) \times \mathbb{A} \times V(G)$ represent relations between these entities, with $\mathbb{A}$ being a fixed finite alphabet. For instance, Figure 1 depicts a graph database, whose nodes are authors and papers, on the alphabet $\mathbb{A} = \{\text{wrote}, \text{advised}\}$. Edges $x \xrightarrow{\text{wrote}} y$ indicate that the person x wrote the paper y, while edges $x \xrightarrow{\text{advised}} y$ indicate that person x was the Ph.D. advisor of person y.

Being a subclass of relational databases, graph databases can be queried by the predominant query language of *conjunctive queries*, a.k.a. CQs, which consists of the closure under projection of conjunctions of atoms of the form $x \xrightarrow{a} y$ for some letter $a \in \mathbb{A}$. For instance, the conjunctive query

$$\gamma_1(x, y) = x \xrightarrow{\text{wrote}} z \wedge y \xrightarrow{\text{wrote}} z$$

[1] This result was achieved by using the `knowledge` package and its companion tool `knowledge-clustering`.

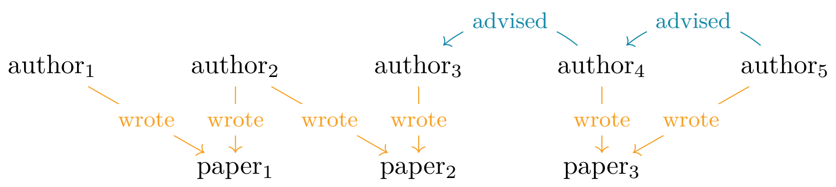


Figure 1 A graph database with eight nodes and eight edges on a two-letter alphabet.

returns, when evaluated on the graph database G defined in Figure 1, all pairs of nodes (u, v) such that u is a co-author of v. Each variable not appearing in the left-hand side of the definition of a conjunctive query (in this example, z) is existentially quantified. Note that every CQ can be seen as a graph database, where each atom is an edge; hence, we sometimes use graph database terminology for CQs.

The expressive power of CQs is somewhat limited, since CQs cannot express, for example, transitive closure. Since the ability to navigate paths is of importance in many graph database scenarios, most modern graph query languages support, as a central querying mechanism, conjunctive regular path queries, or CRPQs for short. CRPQs are defined analogously to conjunctive queries, except that their atoms are now of the form $x \xrightarrow{L} y$ where L is an arbitrary regular language over the alphabet $\mathbb{A}$. For instance the evaluation of the CRPQ $\gamma_2(x, y) = x \xrightarrow{\text{wrote}} z \wedge z' \xrightarrow{\text{wrote}} z \wedge y \xrightarrow{(\text{advised})^*} z'$ on G yields every pair of persons (u, v) such that u is a co-author of a "scientific descendant" of v.

Formally, a *CRPQ* γ is defined as a tuple $\bar{z} = (z_1, \ldots, z_n)$ of *output variables*[2] together with a conjunction of *atoms* of the form $\bigwedge_{j=1}^{m} x_j \xrightarrow{L_j} y_j$, where each L_j is a regular language. The set of all variables occurring in γ, namely[3] $\{z_1, \ldots, z_n\} \cup \{x_1, y_1, \ldots, x_m, y_m\}$, is denoted by *vars*$(\gamma)$. Given a database G, we say that $(u_1, \ldots, u_n)$ *satisfies* γ on G if there is a mapping $f\colon \mathit{vars}(\gamma) \to V(G)$ such that $u_i = f(z_i)$ for all $1 \leqslant i \leqslant n$, and for each $1 \leqslant j \leqslant m$, there exists a path from $f(x_i)$ to $f(y_i)$ in G, labelled by a word from L_i (if the path is empty, the labelled word is ε). The *evaluation* of γ on G is then the set of all tuples that satisfy γ. For example, $(\text{author}_2, \text{author}_5)$ satisfies γ_2 on the graph database G of Figure 1 via the function that maps x to author$_2$, y to author$_5$, z to paper$_2$, and z' to author$_3$.

The language of CRPQ can be extended to navigate edges in both directions. Consider the database $G^\pm$ obtained from G by adding, for every edge $x \xrightarrow{a} y$ in G, an extra edge $y \xrightarrow{a^-} x$. We obtain a graph database on the alphabet $\mathbb{A}^\pm = \mathbb{A} \dot{\cup} \mathbb{A}^-$ where $\mathbb{A}^- = \{a^- \mid a \in \mathbb{A}\}$. We then define the syntax of a *CRPQ with two-way navigation*, or *C2RPQ*, as a CRPQ on the alphabet $\mathbb{A}^\pm$. Its *evaluation* is defined as the evaluation of the CRPQ on $G^\pm$. For instance, the evaluation of the C2RPQ $\gamma_3(x, y) = x \xrightarrow{(\text{wrote}\cdot\text{wrote}^-)^*} y$ on the graph database of Figure 1 returns all pairs of individuals linked by a chain of co-authorship. It includes $(\text{author}_1, \text{author}_3)$ or $(\text{author}_1, \text{author}_1)$ but not $(\text{author}_1, \text{author}_4)$. If a query has no output variables we call it *Boolean*, and its evaluation can either be the set $\{()\}$, in which case we say that G satisfies the query, or the empty set $\{\}$. For example, G satisfies $\gamma_4() = x \xrightarrow{\text{wrote}} y$ if, and only if, the database contains one author together with the paper they wrote. We denote the set of atoms of a C2RPQ γ by $\text{Atoms}(\gamma)$, and by $\|\gamma\|$ we denote its number of atoms, i.e., $|\text{Atoms}(\gamma)|$.

[2] For technical reasons (see the definition of expansion) we allow for a variable to appear multiple times.
[3] We neither assume disjointness nor inclusion between $\{z_1, \ldots, z_n\}$ and $\{x_1, y_1, \ldots, x_m, y_m\}$

Finally, a *union of CQs* (*UCQs*) [resp. *union of CRPQs* (*UCRPQs*), resp. *union of C2RPQs* (*UC2RPQs*)] is defined as a finite set of CQs [resp. CRPQs, resp. C2RPQs], whose tuples of output variables have all the same arity. The evaluation of a union is defined as the union of its evaluations, for instance:

$$\Gamma_5 = \gamma_5^1(x,y) \vee \gamma_5^2(x,y) \text{ where } \gamma_5^1(x,y) = x \xrightarrow{\text{wrote}} y \text{ and } \gamma_5^2(x,y) = x \xleftarrow{\text{advised}} z \wedge z \xrightarrow{\text{wrote}} y$$

evaluates to the set of pairs (x,y) such that y is a paper written by either x or their advisor. *Infinitary unions* are defined analogously, except that we allow for potentially infinite unions.

For a more detailed introduction to CRPQs, we refer the reader to [10]. For a more general introduction to different query languages for graph databases – including CRPQs – see [6], and for a more practical approach, see [1].

Given two UC2RPQ Γ and Γ', we say that Γ is *contained* in Γ', denoted by $\Gamma \subseteq \Gamma'$ if for every graph database G, for every tuple $\bar{u}$ of G, if $\bar{u}$ satisfies Γ on G, then so does Γ'. The *containment problem* for UC2RPQs is the problem of, given two UC2RPQs Γ and Γ', to decide if $\Gamma \subseteq \Gamma'$. When Γ is contained in Γ' and vice versa, we say that Γ and Γ' are *semantically equivalent*, denoted by $\Gamma \equiv \Gamma'$. The *evaluation problem* for UC2RPQ is the problem of, given a C2RPQ γ, a graph database G and a tuple $\bar{u}$ of elements of G, whether $\bar{u}$ satisfies γ on G.

Queries of small tree-width

It is known that the evaluation problem for UC2RPQ is NP-complete, just as for conjunctive queries [9]. However, queries whose underlying structure looks like a tree – formally, queries of bounded tree-width – can be evaluated in polynomial time.

Tree-width is a measure of how much a graph differs from a tree, introduced by Arnborg and Proskurowski [2]. Formally, a *tree decomposition* of a C2RPQ γ is a pair $(T, \mathbf{v})$ where T is a tree and $\mathbf{v} : V(T) \to \wp(vars(\gamma))$ is a function that associates to each node of T, called *bag*, a set of variables of γ. When $x \in \mathbf{v}(b)$ we shall say that the bag $b \in V(T)$ *contains* the variable x. Further, it must satisfy the following three properties:

- each variable x of γ is contained in at least one bag of T;
- for each atom $x \xrightarrow{L} y$ of γ, there is at least one bag of T that contains both x and y; and
- for each variable x of γ, the set of bags of T containing x is a connected subset of $V(T)$.

The *width* of $(T, \mathbf{v})$ is the maximum of $|\mathbf{v}(b)| - 1$ when b ranges over $V(T)$. The *tree-width* of γ is the minimum of the width of all tree decompositions of γ. We denote by $\mathcal{T}w_k$ the set of all C2RPQ of tree-width at most k. The tree-width of a UC2RPQ is simply the maximum of the tree-width of its C2RPQs. An example of tree decomposition of width 2 is given in Figure 3 on Page 12. For a gentle introduction to tree-width, see [14, §3.6].

▶ **Proposition 1.1** (Folklore, see e.g. [15, Theorem IV.3]). *For each $k \geqslant 1$, the evaluation problem for UC2RPQs of tree-width at most k is in polynomial time.*

In practice, graph databases tend to be huge and often changing, while queries are in comparison very small. This motivates the following question, given some natural $k \geqslant 1$:

Given a UC2RPQ Γ, is it equivalent to a UC2RPQ Γ' of tree-width at most k?
That is, does it have *semantic tree-width* at most k?

This problem is called the *semantic tree-width k problem*. Should it be decidable in a constructive way – that is, decidable, and if the answer is positive, we can compute a witnessing Γ' from Γ – , then one could, once and for all, compute Γ' from Γ and, whenever one wants to evaluate Γ on a database, evaluate Γ' instead.

▶ **Example 1.2.** Consider the following CRPQs, where $\bar{x} = (x_0, x_1, y, z)$:

$$\gamma(\bar{x}) \mathrel{\hat{=}} \begin{tikzcd} x_0 \arrow[rr, "c"] \arrow[dr, "a"] \arrow[ddr, bend right, "a(bb)^+"'] & & x_1 \arrow[dl, "a"'] \arrow[ddl, bend left, "ab(bb)^*"] \\ & y \arrow[d, "b^+"] & \\ & z & \end{tikzcd} \qquad \delta(\bar{x}) \mathrel{\hat{=}} \begin{tikzcd} x_0 \arrow[rr, "c"] \arrow[dr, "a"] \arrow[ddr, bend right, "a(bb)^*"'] & & x_1 \arrow[dl, "a"'] \\ & y \arrow[d, "b(bb)^*"] & \\ & z & \end{tikzcd} \qquad \delta'(\bar{x}) \mathrel{\hat{=}} \begin{tikzcd} x_0 \arrow[rr, "c"] \arrow[dr, "a"] & & x_1 \arrow[dl, "a"'] \arrow[ddl, bend left, "ab(bb)^*"] \\ & y \arrow[d, "(bb)^+"'] & \\ & z & \end{tikzcd}$$

The underlying graph of $\gamma(\bar{x})$ being the directed 4-clique, $\gamma(\bar{x})$ has tree-width 3. We claim that $\gamma(\bar{x})$ is equivalent to the UCRPQ $\delta(\bar{x}) \vee \delta'(\bar{x})$, and hence has semantic tree-width 2.

Indeed, given a graph database satisfying $\gamma(\bar{x})$ via some mapping μ, it suffices to make a case disjunction on whether the number of b-labelled atoms in the path from $\mu(y)$ to $\mu(z)$ is even or odd. In the first case, the atom $x_0 \xrightarrow{a(bb)^+} z$ becomes redundant since we can deduce the existence of such a path from the conjunction $x \xrightarrow{a} y \xrightarrow{(bb)^+} z$, and hence the database satisfies $\delta(\bar{x})$ via μ. Symmetrically, in the second case, the atom $x_1 \xrightarrow{b(bb)^*} z$ becomes redundant, and the database satisfies $\delta'(\bar{x})$ via μ. Thus, $\gamma(\bar{x})$ is contained, and hence equivalent (the other containment being trivial), to the UCRPQ $\delta(\bar{x}) \vee \delta'(\bar{x})$ of tree-width 2.

For conjunctive queries, the semantic tree-width k problem can be effectively decided quite easily – in fact, CQs enjoy the effective existence of unique minimal queries [9, Theorem 12] which happen to also minimize the tree-width. For CRPQs and UC2RPQs, the question is far more challenging, and it has only been solved for the case $k = 1$ by Barceló, Romero, and Vardi [5, Theorem 6.1]. We solve the problem for every other $k > 1$, left open in [5, §7] [15, §VI-(3)]:

▶ **Theorem 1.3.** *For each $k \geqslant 1$, the semantic tree-width k problem is decidable. Moreover, it lies in* 2ExpSpace *and is* ExpSpace*-hard.*

Amusingly, our proof for $k > 1$ cannot be stretched to capture the case $k = 1$: the two approaches seem to be intrinsically incompatible.

▶ Remark 1.4. To simplify proofs, we assume that all the regular languages are described via non-deterministic finite automata (NFA) instead of regular expressions, which does not affect any of our complexity bounds. However, for readability all our examples will be given in terms of regular expressions.

Moreover, we also show that for any class $\mathcal{L}$ of regular languages over $\mathbb{A}^{\pm}$ satisfying some mild hypothesis ("closure under sublanguages"), if $\Gamma \in \text{UC2RPQ}(\mathcal{L})$ has semantic tree-width $k > 1$, then Γ is equivalent to a UC2RPQ($\mathcal{L}$) of tree-width at most k, where UC2RPQ($\mathcal{L}$) denotes the class of all UC2RPQs whose atoms are all labelled by languages from $\mathcal{L}$. In other words, if a query can be defined with labels in $\mathcal{L}$, and if this query is equivalent to a query of small tree-width, then it is also equivalent to a query of small tree-width with labels in $\mathcal{L}$.

For a NFA $\mathcal{A}$ and two states q, q' thereof, we denote by $\mathcal{A}[q, q']$ the *sublanguage* of $\mathcal{A}$ recognized when considering q as initial state and $\{q'\}$ as set of final states. We say that $\mathcal{L}$ is *closed under sublanguages* if (i) it contains every language of the form $\{a\}$, where $a \in \mathbb{A}$ is any (positive) letter such that either a or a^- occur in a word of a language of $\mathcal{L}$, and (ii) for every language $L \in \mathcal{L}$ there exists a NFA $\mathcal{A}_L$ such that every sublanguage $\mathcal{A}_L[q, q']$ distinct from $\varnothing$ and $\{\varepsilon\}$ belongs to $\mathcal{L}$.

To the best of our knowledge, all classes of regular expressions that have been considered in the realm of regular path queries (see, e.g., [11, §1]) are closed under sublanguages. In particular, this is the case for the class $\big\{\{a_1 + \ldots + a_n\} \mid a_1, \ldots, a_n \in \mathbb{A}\big\} \cup \big\{a^* \mid a \in \mathbb{A}\big\}$, which will be our focus of study in Section 6. Moreover, even if some class $\mathcal{L}$ is not closed under sublanguages, such as for example $\{(aa)^*\}$, then it is contained is a class closed under sublanguages – $\{a, a(aa)^*, (aa)^*\}$ in this example – , whose size[4] is polynomial in the size of the original class. See the full version for more examples.

▶ **Theorem 1.5.** *Assume that $\mathscr{L}$ is closed under sublanguages. For any query $\Gamma \in$ UC2RPQ($\mathscr{L}$) and $k > 1$, the following are equivalent:*

1. *Γ is equivalent to an infinitary union of conjunctive queries of tree-width at most k;*
2. *Γ has semantic tree-width at most k;*
3. *Γ is equivalent to a* UC2RPQ($\mathscr{L}$) *of tree-width at most k.*

The implications (3) $\Rightarrow$ (2) $\Rightarrow$ (1) immediately follow from the definition of the semantic tree-width. On the other hand, the implications (1) $\Rightarrow$ (2) and (2) $\Rightarrow$ (3) are surprising, since they are both trivially false when $k = 1$. We defer the proof of this claim to Section 2 (see Remark 2.5) as we first need a few tools to manipulate CRPQs.

The proofs of both Theorems 1.3 and 1.5 rely on our key lemma (Lemma 3.8), which states essentially that every UC2RPQ has a computable "maximal under approximation" by a UC2RPQ of tree-width k. Formally, the key lemma has the following corollary:

▶ **Corollary 3.9.** *For each $k > 1$ and for each class $\mathscr{L}$ closed under sublanguages, for each query $\Gamma \in$* UC2RPQ($\mathscr{L}$)*, there exists $\Gamma' \in$* UC2RPQ($\mathscr{L}$) *of tree-width k such that $\Gamma' \subseteqq \Gamma$, and for every $\Delta \in$* UC2RPQ*, if Δ has tree-width k and $\Delta \subseteqq \Gamma$, then $\Delta \subseteqq \Gamma'$. Moreover, Γ' is computable from Γ in* ExpSpace.

The proof of our key lemma spans over Sections 3–5: in Section 3, we introduce necessary notions to formally state it, and deduce Theorems 1.3 and 1.5 from it; in Section 4 we introduce the central notion of tagged tree decompositions of C2RPQs homomorphisms, and building on it, we finally describe the constructions used to prove the key lemma in Section 5.

Finally, in Section 6, given the high complexity of semantic tree-width k problem, we focus on the case of CRPQs using some simple regular expressions (SRE), and show that the complexity of this problem is much lower:

▶ **Theorem 6.1.** *For $k > 1$, the semantic tree-width k problem for* UCRPQ(SRE) *is in Π_2^p.*

A discussion on differences with Barceló, Romero and Vardi's contributions and open questions are left for Section 7.

2 Homomorphisms, refinements, and expansions

Before attacking the statement of our key lemma in Section 3, we first give a few elementary definitions on C2RPQs in this section. A *homomorphism* f from a C2RPQ $\gamma(x_1, \ldots, x_m)$ to a C2RPQ $\gamma'(y_1, \ldots, y_m)$ is a mapping from $vars(\gamma)$ to $vars(\gamma')$ such that $f(x) \xrightarrow{L} f(y)$ is an atom of γ' for every atom $x \xrightarrow{L} y$ of γ, and further $f(x_i) = y_i$ for every i. Such a homomorphism h is *strong onto* if for every atom $x' \xrightarrow{L} y'$ of γ' there is an atom $x \xrightarrow{L} y$ of γ such that $f(x) = x'$ and $f(y) = y'$. We write $\gamma \xrightarrow{hom} \gamma'$ if there is a homomorphism from γ to γ', and $\gamma \overset{hom}{\twoheadrightarrow} \gamma'$ if there is a strong onto homomorphism. It is easy to see that if $\gamma \xrightarrow{hom} \gamma'$ then $\gamma' \subseteqq \gamma$, and in the case where γ, γ' are CQs this is an "if and only if" [9, Lemma 13].

[4] Defined as the sum of the number of states of the minimal automaton of the languages of $\mathscr{L}$.

Some intuitions on maximal under-approximations. Given a conjunctive query γ, the union of all conjunctive queries that are contained in γ is semantically equivalent to the union $\bigvee\{\gamma' \mid \gamma \xrightarrow{hom}\!\!\!\!\!\rightarrow \gamma'\}$. Naturally, this statement borders on the trivial since γ' belongs to this union. It becomes interesting when we add a restriction: given a class $\mathcal{C}$ of CQs (to which γ may not belong) closed under subqueries, then $\bigvee\{\gamma' \in \mathcal{C} \mid \gamma \xrightarrow{hom}\!\!\!\!\!\rightarrow \gamma'\}$ is the maximal under-approximation[5] of γ by finite unions of conjunctive queries of $\mathcal{C}$[6]. As a consequence, we deduce that for each $k \geqslant 1$, the maximal under-approximation of a CQ by a finite union of CQs of tree-width at most k is computable, and hence we can effectively decide if some CQ is equivalent to a query of tree-width at most k. For more details on approximations of CQs, see [3].

Unfortunately, these results cannot be straightforwardly extended to conjunctive regular path queries: intuitively, taking homomorphic images can be understood as "simplifying" the query (we reduce its number of variables, but this may make the query strictly contained in the original one). This is because CRPQs have an implicit quantification of variables: for instance, the CQ $\gamma(x,y) = x \xrightarrow{a} z \xrightarrow{b} y$ can be rewritten as the CRPQ $\gamma'(x,y) = x \xrightarrow{ab} y$. Coming back to our previous Example 1.2, another way of seeing that $\delta(\bar{x}) \subsetneqq \gamma(\bar{x})$ is by observing that we can obtain $\delta(\bar{x})$ as the result of the following two operations:

- in $\gamma(\bar{x})$, replace the atom $x_1 \xrightarrow{ab(bb)^*} z$ with $x_1 \xrightarrow{a} t \xrightarrow{b(bb)^*} z$, where t is a fresh existentially quantified variable;
- identify variables t with y.[7]

The last operation consists in taking homomorphic images, and the first one amounts to making explicit an implicit quantification. We formalize the first operation by introducing the notion of "refinement" which we will later use, in Section 3, to introduce the notion of maximal under-approximations for CRPQs. We will come back this example once all these notions will have been formally defined (cf. Example 3.4).

Refinements. An *atom m-refinement* of a C2RPQ atom $\gamma(x,y) = x \xrightarrow{L} y$ where L is given by the NFA $\mathcal{A}_L$ is any C2RPQ of the form

$$\rho(x,y) = x \xrightarrow{L_1} t_1 \xrightarrow{L_2} \dots \xrightarrow{L_{n-1}} t_{n-1} \xrightarrow{L_n} y \tag{1}$$

where $1 \leqslant n \leqslant m$, $t_1, \dots, t_{n-1}$ are fresh (existentially quantified) variables, and $L_1, \dots, L_n$ are such that there exists a sequence $(q_0, \dots, q_n)$ of states of $\mathcal{A}_L$ such that q_0 is initial, q_n is final, and for each i, L_i is either of the form (i) $\mathcal{A}_L[q_i, q_{i+1}]$, or (ii) $\{a\}$ if the letter $a \in \mathbb{A}$ belongs to $\mathcal{A}_L[q_i, q_{i+1}]$, or (iii) $\{a^{-1}\}$ if $a^{-1} \in \mathbb{A}^{-}$ belongs to $\mathcal{A}[q_i, q_{i+1}]$. Additionally, if $\varepsilon \in L$, the equality atom "$x = y$" is also an *atom m-refinement* (see the full version for more details on these), Thus, an *atom m-refinement* can be either of the form (1) or "$x = y$". By convention, $t \xrightarrow{a^-} t'$ is a shorthand for $t' \xrightarrow{a} t$. As a consequence, the underlying graph of an atom m-refinement of the form (1) is not necessarily a directed path. By definition, note that $L_1 \cdots L_n \subseteq L$ and hence $\rho \subseteqq \gamma$ for any atom m-refinement ρ of γ. An *atom refinement* is an atom m-refinement for some m.

[5] A formalization of what "maximal under-approximation" means is given in Remark 3.2.

[6] The proof is straightforward: by definition, this union is finite (a finite CQ has only finitely many homomorphic images), and is contained in γ. Moreover, if $\gamma' \in \mathcal{C}$ is contained in γ, then there exists a homomorphism $f\colon \gamma \to \gamma'$. Then $\gamma \xrightarrow{hom}\!\!\!\!\!\rightarrow f(\gamma)$ and $f(\gamma) \in \mathcal{C}$ since it is a subquery of $\gamma' \in \mathcal{C}$ and $\mathcal{C}$ is closed under subqueries. We conclude by noting that γ' is contained in $f(\gamma)$.

[7] We actually obtain two atoms from y to z: one label by b^+ and one by $b(bb)^*$, but since $b(bb)^* \subseteq b^+$ we can discard the b^+ atom preserving the query semantics.

▶ **Definition 2.1.** *Given an atom refinement $\rho = x \xrightarrow{L_1} t_1 \xrightarrow{L_2} \dots \xrightarrow{L_{n-1}} t_{n-1} \xrightarrow{L_n} y$ of $\gamma = x \xrightarrow{L} y$ as in* (1)*, define a* contraction *of ρ between t_i and t_j, where $0 \leqslant i, j \leqslant n$ and $j > i+1$, is any C2RPQ of the form:*

$$\rho' = x \xrightarrow{L_1} t_1 \xrightarrow{L_2} \dots \xrightarrow{L_i} \boldsymbol{t_i} \xrightarrow{\boldsymbol{K}} \boldsymbol{t_j} \xrightarrow{L_{j+1}} \dots \xrightarrow{L_{n-1}} t_{n-1} \xrightarrow{L_n} y$$

such that $K = \mathcal{A}[q_i, q_j]$. Then every contraction ρ' of ρ is a refinement of γ, and $\rho \subseteqq \rho' \subseteqq \gamma$. Informally, we will abuse the notation and write $[L_i \cdots L_j]$ to denote the language K – even if this language does not only depend on $L_i \cdots L_j$.

▶ **Example 2.2.** Let $\gamma(x,y) = x \xrightarrow{(aa^-)^*} y$ be a C2RPQ atom, where $(aa^-)^*$ is implicitly represented by its minimal automaton. Then $\rho(x,y)$ is a refinement of refinement length seven of $\gamma(x,y)$ and $\rho'(x,y)$ is a contraction of $\rho(x,y)$, where:

$$\rho(x,y) = x \xrightarrow{a} t_1 \xrightarrow{(a^-a)^*} t_2 \xrightarrow{(a^-a)^*} t_3 \xleftarrow{a} t_4 \xrightarrow{(aa^-)^*} t_5 \xrightarrow{(aa^-)^*a} t_6 \xleftarrow{a} y,$$
$$\rho'(x,y) = x \xrightarrow{a} t_1 \xrightarrow{(a^-a)^*a^-} t_4 \xrightarrow{(aa^-)^*} y.$$

On the other hand, $\rho''(x,y) = x \xrightarrow{a} t_1 \xleftarrow{a} y$ is not a contraction of $\rho(x,y)$.

An *m-refinement* of a C2RPQ $\gamma(\bar{x}) = \bigwedge_i x_i \xrightarrow{L_i} y_i$ is any query resulting from: 1) replacing every atom by one of its m-refinements, and 2) should some m-refinements be equality atoms, collapsing equal variables and getting rid of equalities, in a rather standard way – more details are given in the full version. A *refinement* is an m-refinement for some m. Note that any atom m-refinements is, by definition, also an atom m'-refinements when $m < m'$: as a consequence, in the refinement of a C2RPQ the atom refinements need not have the same length. For instance, both $\rho(x,x) = x \xrightarrow{c} x$ and $\rho'(x,y) = x \xrightarrow{a} t_1 \xrightarrow{a} y \xleftarrow{c} y$ are refinements of $\gamma(x,y) = x \xrightarrow{a^*} y \xleftarrow{c} x$. For a given C2RPQ γ, let $\mathrm{Ref}^{\leqslant m}(\gamma)$ be the set of all m-refinements of γ, and $\mathrm{Ref}(\gamma)$ be the set of all its refinements. Given a refinement $\rho(\bar{x})$ of $\gamma(\bar{x})$, its *refinement length* is the least integer m such that $\rho(\bar{x}) \in \mathrm{Ref}^{\leqslant m}(\gamma)$. Note that if the automaton representing a language L has more than one final state, for instance the minimal automaton for $L = a^+ + b^+$, then $x \xrightarrow{L} y$ is not a refinement of itself. However, it will always be equivalent to a union of refinements: in this example, $x \xrightarrow{a^+ + b^+} y$ is equivalent to the union of $x \xrightarrow{a^+} y$ and $x \xrightarrow{b^+} y$, which are both refinements of the original C2RPQ.

Expansions. Remember that a C2RPQ whose languages are $\{a\}$ or $\{a^-\}$ for $a \in \mathbb{A}$ is in effect a CQ. The *expansions* of a C2RPQ γ is the set $\mathrm{Exp}(\gamma)$ of all CQs which are refinements of γ. In other words, an expansion of γ is any CQ obtained from γ by replacing each atom $x \xrightarrow{L} y$ by a path $x \xrightarrow{w} y$ for some word $w \in L$. For instance, $\xi(x,y) = x \xrightarrow{a} t_1 \xleftarrow{a} t_2 \xrightarrow{a} t_3 \xleftarrow{a} y$ is an expansion of $\rho(x,y) = x \xrightarrow{(aa^-)^*} y$.

Any C2RPQ is equivalent to the infinitary union of its expansions. In light of this, the semantics for UC2RPQ can be rephrased as follows. Given a UC2RPQ Γ and a graph database G, the evaluation of Γ over G, denoted by $\Gamma(G)$, is the set of tuples $\bar{v}$ of nodes for which there is $\xi \in \mathrm{Exp}(\Gamma)$ such that $\xi \xrightarrow{hom} (G, \bar{v})$. Similarly, containment of UC2RPQs can also be characterized in terms of expansions:

▶ **Proposition 2.3** (Folklore, see e.g. [12, Proposition 3.2] or [8, Theorem 2]). *Let Γ_1 and Γ_2 be UC2RPQs. Then the following are equivalent*

- $\Gamma_1 \subseteq \Gamma_2$*;*
- *for every $\xi_1 \in \mathrm{Exp}(\Gamma_1)$, $\xi_1 \subseteqq \Gamma_2$;*
- *for every $\xi_1 \in \mathrm{Exp}(\Gamma_1)$ there is $\xi_2 \in \mathrm{Exp}(\Gamma_2)$ such that $\xi_2 \xrightarrow{hom} \xi_1$.*

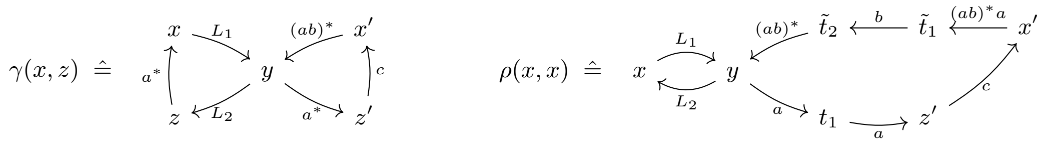


Figure 2 On the left-hand side, a CRPQ γ of tree-width 2. On the right-hand side, a refinement ρ of γ, whose refinement length is three, which is also of tree-width 2.

Hence, every expansion of γ is also a refinement of γ. Moreover, if ρ is a refinement of γ, then $\rho \subseteq \gamma$, and γ is semantically equivalent to the infinitary union of all its refinements, and to the infinitary union of all its expansions.

Our approach to proving Theorems 1.3 and 1.5 and the key lemma heavily rely on refinements. One crucial property that these objects satisfy is that they preserve tree-width k, unless $k = 1$. This is the main reason why our approach cannot capture the case $k = 1$, solved by Barceló, Romero and Vardi [5].

▷ Fact 2.4. Let $k > 1$ and let γ be a C2RPQ of tree-width at most k. Then any refinement of γ has tree-width at most k.

This fact is illustrated on a example in Figure 2.

Proof sketch. The underlying graph of a refinement of γ is obtained from the underlying graph of γ by either contracting some edges (when dealing with equality atoms), or by replacing a single edge by a path of edges (where the non-extremal nodes are new nodes). Both operations preserve the property "having tree-width at most k" when $k > 1$. Details are given in the full version. ◁

For $k = 1$, the property fails: for instance the CRPQ $\gamma(x) = x \xrightarrow{a^*} x$ has tree-width at most 1 (in fact it has tree-width 0), but its refinement $\rho(x) = x \xrightarrow{a^*} t_1 \xrightarrow{a^*} t_2 \xrightarrow{a^*} x$ has tree-width two.

Before introducing maximal under-approximations, we can now show that the statement of Theorem 1.5 is indeed false for $k = 1$.

▶ Remark 2.5. (1) $\not\Rightarrow$ (2) when $k = 1$: consider the CRPQ $\gamma(x,y) = x \xrightarrow{a^*} y \wedge y \xrightarrow{b} x$ of tree-width 1, and hence of semantic tree-width 1, and observe that it is not equivalent to any infinitary union of conjunctive queries of tree-width 1 – this can be proven by considering, for example, the expansion $x \xrightarrow{a} z \xrightarrow{a} y \wedge y \xrightarrow{b} x$ of $\gamma(x,y)$ and applying Proposition 2.3.

(2) $\not\Rightarrow$ (3) when $k = 1$: By [5, Proposition 6.4] the CRPQ of semantic tree-width 1 $\gamma(x) \doteq x \xleftarrow{a} z \xrightarrow{a} y \wedge x \xrightarrow{b} y \equiv x \xrightarrow{ba^-a} x$ is not equivalent to any UCRPQ of tree-width 1. Hence, the implication is false when $\mathcal{L}$ is the class of regular languages over $\mathbb{A}^{\pm}$ that do not use any letter of the form a^-.

3 Maximal under-approximations of bounded tree-width

In this section, we state our key technical result, Lemma 3.8. Essentially, we follow the same structure as Theorem 1.5: given a C2RPQ γ and an integer $k > 1$, we start by consider its maximal under-approximation by infinitary unions of conjunctive queries of tree-width k (Definition 3.1), and then show that this query can in fact be expressed as a UC2RPQ of tree-width k whose atoms are obtained by taking sublanguages from γ (Lemma 3.8).

For the definitions of this section, let us fix any class $\mathcal{C}$ of C2RPQ queries.

▶ **Definition 3.1** (Maximal under-approximation). *Let γ be a C2RPQ. The* maximal under-approximation *of γ by infinitary unions of $\mathcal{C}$-queries, is* $\mathrm{App}_{\mathcal{C}}(\gamma) \mathrel{\hat{=}} \{\alpha \in \mathcal{C} \mid \alpha \subseteqq \gamma\}$.

For intuition, we refer the reader to paragraph "An intuition on maximal under-approximations" at the beginning of Section 2.

▶ Remark 3.2. Observe that $\mathrm{App}_{\mathcal{C}}(\gamma)$ is an infinitary union of $\mathcal{C}$-queries, that $\mathrm{App}_{\mathcal{C}}(\gamma) \subseteqq \gamma$, and that for every infinitary union of $\mathcal{C}$-queries Δ, if $\Delta \subseteqq \gamma$, then $\Delta \subseteqq \mathrm{App}_{\mathcal{C}}(\gamma)$ (i.e., it is the unique maximal under-approximation). Similarly, the maximal under-approximation of a UC2RPQ is simply the union of the maximal under-approximations of the C2RPQs thereof.

Unfortunately, the fact that a query α is part of this union, $\alpha \in \mathrm{App}_{\mathcal{C}}(\gamma)$, does not yield any useful information on the shape of α – we merely know that $\alpha \subseteqq \gamma$. The rest of this subsection is dedicated to introducing another infinitary union of $\mathcal{C}$-queries, namely $\mathrm{App}^{\star}_{\mathcal{C}}(\gamma) \subseteq \mathrm{App}_{\mathcal{C}}(\gamma)$, in which queries $\alpha \in \mathrm{App}^{\star}_{\mathcal{C}}(\gamma)$ come together with a witness – a homomorphism – of their containment in γ.

▶ **Definition 3.3.** *The maximal under-approximation of γ by infinitary unions of homomorphically-smaller $\mathcal{C}$-queries is*

$$\mathrm{App}^{\star}_{\mathcal{C}}(\gamma) \mathrel{\hat{=}} \{\alpha \in \mathcal{C} \mid \exists \rho \in \mathrm{Ref}(\gamma), \exists f\colon \rho \xrightarrow{hom} \!\!\!\!\rightarrow \alpha\}. \tag{2}$$

For instance, let $\mathcal{C}$ be the class of all CRPQs and let $\gamma(x,y) \mathrel{\hat{=}} x \xrightarrow{a^*} y \xrightarrow{a} z \xrightarrow{a^*} x \wedge y \xrightarrow{a^*} x$ be the CRPQ which asks for all pairs of nodes that belong to the same non-empty cycle of a's[8]. Then $\gamma'(x,y) = x \xrightarrow{a^*} y \xrightarrow{a} z \xrightarrow{a^*} x$ is an element of $\mathrm{App}^{\star}_{\mathcal{C}}(\gamma)$ since there is a strong onto homomorphism from the refinement

$$\rho(x,y) = \big(x \xrightarrow{a^*} y \xrightarrow{a} z \xrightarrow{a^*} x \wedge y \xrightarrow{a} z' \xrightarrow{a^*} x\big) \in \mathrm{Ref}(\gamma(x,y))$$

to $\gamma'(x,y)$, mapping z and z' to z.

▶ **Example 3.4** (Example 1.2, cont'd). Both $\delta(\bar{x})$ and $\delta'(\bar{x})$ are semantically equivalent to queries in $\mathrm{App}^{\star}_{\mathcal{T}\mathrm{w}_2}(\gamma(\bar{x}))$. Indeed, starting from $\gamma(\bar{x})$, we can refine $x_0 \xrightarrow{a(bb)^+} z$ into $x_1 \xrightarrow{a} t \xrightarrow{(bb)^+} z$. Denote by $\rho(\bar{x})$ the query obtained. Then merge variables t and y: this new query $\delta'_{\mathrm{app}}(\bar{x})$ is equivalent to $\delta'(\bar{x})$.

$$\rho(\bar{x}) \mathrel{\hat{=}} \begin{tikzcd} x_0 \arrow[rr, "c"] \arrow[d, "a"'] \arrow[dr, "a"] & & x_1 \arrow[dl, "a"'] \arrow[ddl, bend left, "ab(bb)^*"] \\ t \arrow[dr, "(bb)^+"'] & y \arrow[d, "b^+"] & \\ & z & \end{tikzcd} \qquad \delta'_{\mathrm{app}}(\bar{x}) \mathrel{\hat{=}} \begin{tikzcd} x_0 \arrow[rr, "c"] \arrow[dr, "a"] & & x_1 \arrow[dl, "a"'] \arrow[ddl, bend left, "ab(bb)^*"] \\ & y \arrow[d, shift right, "(bb)^+"'] \arrow[d, shift left, "b^+"] & \\ & z & \end{tikzcd} \qquad \delta'(\bar{x}) \mathrel{\hat{=}} \begin{tikzcd} x_0 \arrow[rr, "c"] \arrow[dr, "a"] & & x_1 \arrow[dl, "a"'] \arrow[ddl, bend left, "ab(bb)^*"] \\ & y \arrow[d, "(bb)^+"'] & \\ & z & \end{tikzcd}$$

Clearly, $\mathrm{App}^{\star}_{\mathcal{C}}(\gamma)$ – whose queries are informally called *approximations* – is included, and thus semantically contained, in $\mathrm{App}_{\mathcal{C}}(\gamma)$, since $\rho \subseteqq \gamma$ and $\alpha \subseteqq \rho$ in (2). In fact, under some assumptions on $\mathcal{C}$, the converse containment also holds.

▶ **Observation 3.5.** *If $\mathcal{C}$ is closed under expansions, then for any C2RPQ γ, we have* $\mathrm{App}_{\mathcal{C}}(\gamma) \equiv \mathrm{App}^{\star}_{\mathcal{C}}(\gamma)$.

Proof. This follows immediately from Proposition 2.3 – note that in the definition of $\mathrm{App}^{\star}_{\mathcal{C}}(\gamma)$ we work with strong onto homomorphisms, but we can always restrict homomorphisms to their image to make them strong onto, without changing the expressiveness of the query. ◀

[8] There exists CRPQs with fewer variables that expresses the same property, but this is irrelevant here.

Observe then, by Fact 2.4, that the class $\mathscr{T}\!w_k$ of all C2RPQs of tree-width at most k is closed under refinements and hence under expansions, provided that k is greater or equal to 2. As an immediate consequence, we have:

▶ **Corollary 3.6.** *For $k \geqslant 2$, for all C2RPQ γ, $\mathrm{App}_{\mathscr{T}\!w_k}(\gamma) \equiv \mathrm{App}^{\star}_{\mathscr{T}\!w_k}(\gamma)$.*

▶ **Example 3.7** (Counter-example for $k = 1$)**.** Consider the followings queries:

$$\gamma(x,y) \mathrel{\hat{=}} \begin{tikzcd} z \arrow[r, "(ba)^*"] & z' \arrow[d, "b"] \\ x \arrow[u, "a"] \arrow[r, bend left, "(ab)^+"] & y \arrow[l, bend left, "c"] \end{tikzcd} \qquad \text{and} \qquad \delta(x,y) \mathrel{\hat{=}} \begin{tikzcd} x \arrow[r, bend left, "(ab)^+"] & y. \arrow[l, bend left, "c"] \end{tikzcd}$$

We claim that $\mathrm{App}_{\mathscr{T}\!w_1}(\gamma) \not\subseteq \mathrm{App}^{\star}_{\mathscr{T}\!w_1}(\gamma)$ since $\delta(x,y) \in \mathrm{App}_{\mathscr{T}\!w_1}(\gamma)$ but $\delta(x,y) \not\subseteq \mathrm{App}^{\star}_{\mathscr{T}\!w_1}(\gamma)$. Details are given in the full version.

By definition, $\mathrm{App}^{\star}_{\mathscr{T}\!w_k}(\gamma)$ is an infinitary union of C2RPQs. We show that, in fact, $\mathrm{App}^{\star}_{\mathscr{T}\!w_k}(\gamma)$ is always equivalent to a *finite* union of C2RPQs. This is done by bounding the length of the refinements occurring in the definition of $\mathrm{App}^{\star}_{\mathscr{T}\!w_k}(\gamma)$. For a natural m, let $\mathrm{App}^{\star,\leqslant m}_{\mathcal{C}}(\Gamma) \mathrel{\hat{=}} \{\alpha \in \mathcal{C} \mid \exists \rho \in \mathrm{Ref}^{\leqslant m}(\gamma),\ \exists f\colon \rho \xrightarrow{hom}\!\!\!\!\rightarrow \alpha\}$. Our main technical lemma is then the following:

▶ **Lemma 3.8** (*Key lemma*)**.** *For $k \geqslant 1$ and C2RPQ γ, we have $\mathrm{App}_{\mathscr{T}\!w_k}(\gamma) \equiv \mathrm{App}^{\star,\leqslant \ell}_{\mathscr{T}\!w_k}(\gamma)$, where $\ell = \Theta(\|\gamma\|^2 \cdot (k+1)^{\|\gamma\|+1})$.*

▶ **Corollary 3.9.** *For each $k > 1$ and for each class $\mathcal{L}$ closed under sublanguages, for each query $\Gamma \in \mathrm{UC2RPQ}(\mathcal{L})$, there exists $\Gamma' \in \mathrm{UC2RPQ}(\mathcal{L})$ of tree-width k such that $\Gamma' \subseteqq \Gamma$, and for every $\Delta \in \mathrm{UC2RPQ}$, if Δ has tree-width k and $\Delta \subseteqq \Gamma$, then $\Delta \subseteqq \Gamma'$. Moreover, Γ' is computable from Γ in* ExpSpace.

Using Lemma 3.8 as a black box – which will be proven in Section 5 – , we can now give a proof of Theorems 1.3 and 1.5. The upper bound of Theorem 1.3 follows directly from Corollary 3.9: to test whether a query Γ is of semantic tree-width k, it suffices to test the containment $\Gamma \subseteqq \Gamma'$, where Γ' is the maximal under-approximation given by Corollary 3.9. The containment problem being in ExpSpace [12, 8], we obtain:

▶ **Lemma 3.10.** *For $k \geqslant 1$, the semantic tree-width k problem for UC2RPQ is in* 2ExpSpace.

An ExpSpace lower bound follows by a straightforward adaptation from the ExpSpace lower bound for the case $k = 1$ [5, Proposition 6.2].

▶ **Lemma 3.11.** *The semantic tree-width k problem is* ExpSpace*-hard, even if restricted to Boolean CRPQs.*

We can now rely on the equivalence $\mathrm{App}_{\mathscr{T}\!w_k}(\gamma) \equiv \mathrm{App}^{\star,\leqslant \ell}_{\mathscr{T}\!w_k}(\gamma)$ to prove Theorem 1.5.

Proof of Theorem 1.5. The implications (3) ⇒ (2) ⇒ (1) are straightforward: they follow directly from Fact 2.4. For (1) ⇒ (3), note that (1) implies that $\Gamma \equiv \mathrm{App}_{\mathscr{T}\!w_k}(\Gamma)$, and by Lemma 3.8, $\mathrm{App}_{\mathscr{T}\!w_k}(\Gamma) \equiv \Delta \mathrel{\hat{=}} \bigvee_{\gamma \in \Gamma} \mathrm{App}^{\star,\leqslant \ell_\gamma}_{\mathscr{T}\!w_k}(\gamma)$, so Γ is equivalent to the latter. Since queries of Δ are obtained as homomorphic images of refinements of Γ, all of which are labelled by sublanguages of $\mathcal{L}$, and since $\mathcal{L}$ is closed under sublanguages, it follows that hence Γ is equivalent to a UC2RPQ($\mathcal{L}$) of tree-width k. ◀

We are left with the proof of Lemma 3.8, which will take the next two sections. Since $\mathrm{App}_{\mathcal{T}\mathrm{w}_k}(\gamma) \equiv \mathrm{App}^{\star}_{\mathcal{T}\mathrm{w}_k}(\gamma)$ by Corollary 3.6 and since $\mathrm{App}^{\star,\leqslant\ell}_{\mathcal{T}\mathrm{w}_k}(\gamma)$ is a subset of $\mathrm{App}_{\mathcal{T}\mathrm{w}_k}(\gamma)$, we only need to show that $\mathrm{App}^{\star}_{\mathcal{T}\mathrm{w}_k}(\gamma) \subseteqq \mathrm{App}^{\star,\leqslant\ell}_{\mathcal{T}\mathrm{w}_k}(\gamma)$. Formally, this means that for all $\alpha \in \mathcal{T}\mathrm{w}_k$, if there exists $\rho \in \mathrm{Ref}(\gamma)$ and $f\colon \rho \xrightarrow{hom}\!\!\!\!\!\to \alpha$, then there exists $\alpha' \in \mathcal{T}\mathrm{w}_k$ such that $\alpha \subseteqq \alpha'$ and there exists $\rho' \in \mathrm{Ref}^{\leqslant\ell}(\gamma)$ and $f'\colon \rho' \xrightarrow{hom}\!\!\!\!\!\to \alpha'$. We prove this by "massaging" the homomorphism $f\colon \rho \xrightarrow{hom}\!\!\!\!\!\to \alpha$, by looking where each atom refinement of γ is sent on α relative to a "well-behaved" tree decomposition of α of width k. Next, we introduce the notion of tagged tree decomposition to have a precise handle on this information.

4 Intermezzo: tagged tree decompositions

▶ **Definition 4.1.** *Let $f\colon \rho \to \alpha$ be a homomorphism between two C2RPQs. A* tagged tree decomposition *of f is a triple $(T, \mathbf{v}, \mathbf{t})$ where $(T, \mathbf{v})$ is a tree decomposition of α, and $\mathbf{t}$ is a mapping $\mathbf{t}\colon \mathrm{Atoms}(\gamma) \to V(T)$, called* tagging*, such that $\mathbf{v}(\mathbf{t}(e))$ contains both $f(x)$ and $f(y)$ for each atom $e = x \xrightarrow{\lambda} y \in \mathrm{Atoms}(\gamma)$.*

In other words, $\mathbf{t}$ gives, for each atom of γ, a witnessing bag that contains it, in the sense that it contains the image by f of the atom source and target. By definition, given a tree decomposition $(T, \mathbf{v})$ of α and a homomorphism $f\colon \rho \to \alpha$, there is always one way (usually many) of extending $(T, \mathbf{v})$ into a tagged tree decomposition of f.

▷ Fact 4.2. Let $(T, \mathbf{v}, \mathbf{t})$ be a tagged tree decomposition of some homomorphism $f\colon \rho \to \alpha$. Let T' be the smallest connected subset of T induced by the image of $\mathbf{t}$. Then $(T', \mathbf{v}|_{T'}, \mathbf{t})$ is still a tagged tree decomposition of f, whose width is at most the width of $(T, \mathbf{v}, \mathbf{t})$.

We extend the notion of tagging to paths: a formal definition can be found in the full version, and the notion is illustrated in Figure 3. In the context of a (nice) tagged tree decomposition $(T, \mathbf{v}, \mathbf{t})$ of $f\colon \rho \xrightarrow{hom} \alpha$, given a path π of ρ, say $x_0 \xrightarrow{\lambda_1} x_1 \xrightarrow{\lambda_2} \cdots \xrightarrow{\lambda_n} x_n$ (in blue in Figure 3*a*), the *path induced* by π, denoted $\mathbf{t}[\pi]$, is informally defined as the following "path" in $T \times \alpha$, seen as a sequence of pairs from $V(T) \times \mathit{vars}(\alpha)$:

- it starts with the bag $\mathbf{t}(x_0 \xrightarrow{\lambda_1} x_1)$ of T and the variable $f(x_0)$ of α; and it continues with $(\mathbf{t}(x_0 \xrightarrow{\lambda_1} x_1), f(x_1))$ (corresponding to the first blue edge in b_1 of Figure 3*a*);
- it then follows the shortest path in T (unique, since it is a tree) that goes to the bag $\mathbf{t}(x_1 \xrightarrow{\lambda_2} x_2)$, while staying in $f(x_1)$ in α (in Figure 3*a*, it follows the blue path: $(b_2, x_2), (b_3, x_2)$) and it traverses the atom $x_1 \xrightarrow{\lambda_2} x_2$ (i.e., we go to (b_3, x_3));
- it continues in the same way for all other atoms of the path, ending up with the bag $\mathbf{t}(x_{n-1} \xrightarrow{\lambda_n} x_n)$ and the variable $f(x_n)$ of α.

By construction, note that the constructed sequence $(b_i, z_i)_i$ is such that $z_i \in \mathbf{v}(b_i)$. Moreover, given a bag b of T and a variable z of α, we say that $\mathbf{t}[\pi]$ *leaves* b at z when $z = z_i$ and $b = b_i$ for some i, and either b_{i+1} is undefined, or distinct from b. For example, in Figure 3*a*, $\tau[\pi]$ leaves b_1 at x_1 and leaves b_2 at x_1 and at x_4. We say that an induced path is *cyclic* if it contains three positions $i < j < i'$ such that i and i' contain the same bag but j contains a different bag (the one in Figure 3*a* is cyclic because it passes twice by the bag b_2).

For technical reasons – the proof of Claim 5.2 – , we need to use the classical notion of *nice tree decomposition* (see e.g. [13, Definition 13.1.4, page 149]), which is a tree decomposition $(T, \mathbf{v})$ such that any bag b that is not a leaf either: 1) has exactly two children b_1, b_2 such that $\mathbf{v}(b_1) = \mathbf{v}(b) = \mathbf{v}(b_2)$, or 2) has exactly one child b', and $\mathbf{v}(b')$ is obtained from $\mathbf{v}(b)$ by either adding a single vertex, or by removing a single vertex. A C2RPQ has tree-width k if and only if it has a nice tree decomposition of width at most k [13, Lemma 13.1.2, page 149]. In the context of a nice tree decomposition of width k, a *full bag* is any bag of size

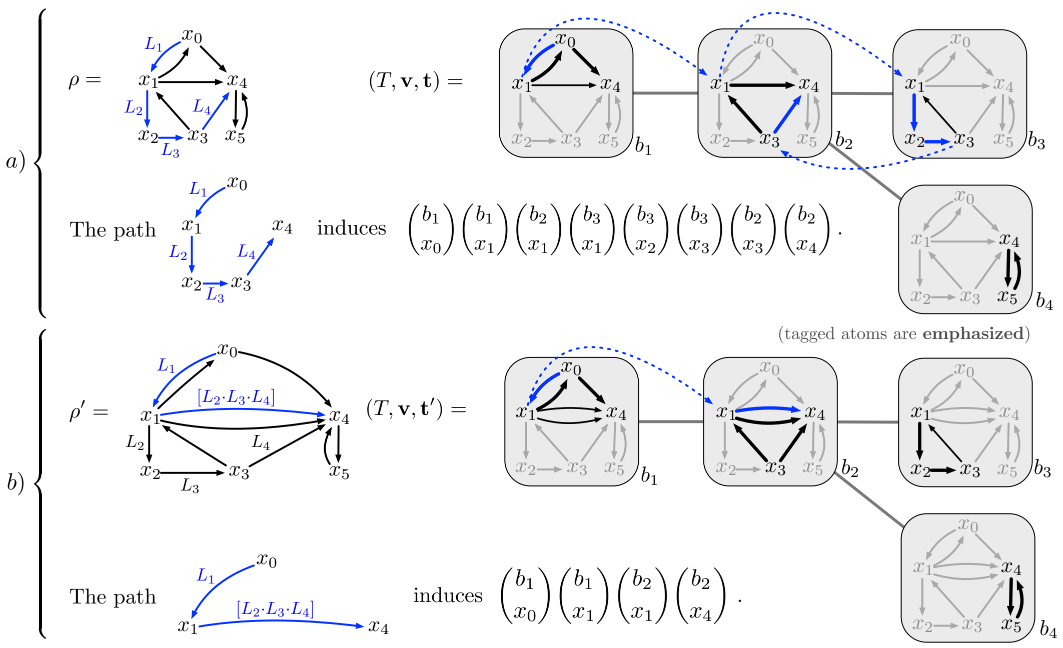

a)
$\rho =$
$(T, \mathbf{v}, \mathbf{t}) =$
The path
induces
$\binom{b_1}{x_0}\binom{b_1}{x_1}\binom{b_2}{x_1}\binom{b_3}{x_1}\binom{b_3}{x_2}\binom{b_3}{x_3}\binom{b_2}{x_3}\binom{b_2}{x_4}$.
(tagged atoms are **emphasized**)
b)
$\rho' =$
$[L_2 \cdot L_3 \cdot L_4]$
$(T, \mathbf{v}, \mathbf{t}') =$
The path
induces
$\binom{b_1}{x_0}\binom{b_1}{x_1}\binom{b_2}{x_1}\binom{b_2}{x_4}$.
b_1
b_2
b_3
b_4

Figure 3 *a*) Non-acyclic path induced by some path in ρ (left-hand side), in a tagged tree decomposition (right-hand side) of $f: \rho \xrightarrow{hom} \alpha$ in the case where $\alpha = \rho$ and f is the identity homomorphism $\mathrm{id}_\rho: \rho \xrightarrow{hom} \rho$.
b) Suppose that the path in ρ is the image of an atom refinement of γ. Then the cycle in the induced path can be avoided by adding an atom $x_1 \xrightarrow{[L_2 \cdot L_3 \cdot L_4]} x_4$ to ρ, obtaining the path of ρ', whose induced path is acyclic.

$k+1$. Note that any *non-branching path* – i.e. a path whose non-extremal bags have degree 2 – in a nice tree decomposition with n bags must have at least $\lfloor n/2 \rfloor$ bags which are not full. Finally, we define a *nice tagged tree decomposition* of $f: \rho \to \alpha$ to be a tagged tree decomposition of f that is also a nice tree decomposition of α.

5 Proof of the key lemma

We can now start to describe the constructions used to prove Lemma 3.8. We call a *trio* to a triple (α, ρ, f) such that $\alpha \in \mathcal{T}w_k$, $\rho \in \mathrm{Ref}(\gamma)$ and f is a strong onto homomorphism from γ to α. For clarity, we will denote such a trio by simply "$f: \rho \xrightarrow{hom} \alpha$". Using this terminology, in order to prove Lemma 3.8, we must show that for every trio $f: \rho \xrightarrow{hom} \alpha$, there exists another trio $f': \rho' \xrightarrow{hom} \alpha'$ such that $\alpha \subsetneqq \alpha'$ and $\rho' \in \mathrm{Ref}^{\leqslant \ell}(\gamma)$. Our first construction, which will ultimately allow us to bound the size of atom refinements, shows that we can assume w.l.o.g. that they induce acyclic paths in a nice tagged tree decomposition of f.

▷ Claim 5.1. For any trio $f: \rho \xrightarrow{hom} \alpha$, there exists a trio $f': \rho' \xrightarrow{hom} \alpha'$ and a nice tagged tree decomposition $(T', \mathbf{v}', \mathbf{t}')$ of width at most k of f' such that $\alpha \subsetneqq \alpha'$, $\|\rho'\| \leqslant \|\rho\|$ and every atom refinement of ρ' induces an acyclic path in the tree T', in which case we say that $(T', \mathbf{v}', \mathbf{t}')$ is *locally acyclic* .

The construction behind Claim 5.1 is quite simple, and is described and proven in Section 5, and illustrated in Figure 3.

Informal proof of Claim 5.1. Start with a trio $f\colon \rho \xrightarrow{hom}\!\!\!\!\rightarrow \alpha$, and let $(T, \mathbf{v}, \mathbf{t})$ be a nice tagged tree decomposition of f. Consider an atom refinement $\pi \mathrel{\hat{=}} z_0 \xrightarrow{L_1} z_1 \xrightarrow{L_2} \cdots \xrightarrow{L_n} z_n$ in ρ of some atom $x \xrightarrow{L} y$ (with $z_0 \mathrel{\hat{=}} x$ and $z_n \mathrel{\hat{=}} y$), and assume that it induces a cyclic path in T, as in Figure 3a. It means that some variables z_i and z_j are mapped by f to the same bag of T, somewhere along the path induced by π. It suffices then to contract ρ by replacing the atoms $z_i \xrightarrow{L_{i+1}} \cdots \xrightarrow{L_j} z_j$ by a single atom $z_i \xrightarrow{[L_{i+1}\cdots L_j]} z_j$ (in Figure 3, $z_i = x_1$ and $z_j = x_4$). We thus obtain a new refinement ρ' of γ. Then define α' be simply adding an atom $f(z_i) \xrightarrow{L_{i+1}\cdots L_j} f(z_j)$, see Figure 3b. The definitions of f' and $(T', \mathbf{v}', \mathbf{t}')$ are then straightforward – potentially, α' should be restricted to the image of $f'\colon \rho' \to \alpha'$ so that f' is still strong onto. Crucially, $\alpha \subsetneq \alpha'$, and α' still has tree-width at most k since we picked $f(z_i)$ and $f(z_j)$ so that they belonged to the same bag of T. $\lhd$

Ultimately, Claim 5.1 will allow us to give a bound on the number of leaves of a nice tagged tree decomposition of a trio. The following claim – which is significantly more technical than the foregoing – will give us a bound on the height of a decomposition.

▷ Claim 5.2. Let $f\colon \rho \xrightarrow{hom}\!\!\!\!\rightarrow \alpha$ be a trio and $(T, \mathbf{v}, \mathbf{t})$ be a locally acyclic nice tagged tree decomposition of width at most k of f. Then there is a trio $f'\colon \rho' \xrightarrow{hom}\!\!\!\!\rightarrow \alpha'$ and a nice tagged tree decomposition $(T', \mathbf{v}', \mathbf{t}')$ of width at most k of f' such that:

- $\alpha \subsetneq \alpha'$,
- $(T', \mathbf{v}', \mathbf{t}')$ is locally acyclic w.r.t. f', and
- the size of the longest non-branching path in T is at most $\Theta(\|\gamma\| \cdot (k+1)^{\|\gamma\|+1})$.

To prove Claim 5.2, we will try to find, in a long non-branching path, some kind of shortcut. The piece of information that is relevant to finding this shortcut is what we call the profile of a bag.

▶ **Definition 5.3.** *Given a trio $f\colon \rho \xrightarrow{hom}\!\!\!\!\rightarrow \alpha$ and a nice tagged tree decomposition $(T, \mathbf{v}, \mathbf{t})$ of f, for each bag b of T, we say that:*

- *b is* “atomic” *if there is at least one atom $e \in \mathbf{t}^{-1}(b)$ and at least one variable x of e such that $x \in vars(\gamma)$, i.e., the atom e is not in the “middle” part of an atom refinement;*
- *otherwise, when b is non-atomic, we assign to each variable $z \in \mathbf{v}(b) \subseteq V(\alpha)$ a* type

$$\mathrm{type}_z \mathrel{\hat{=}} \left\{ x \xrightarrow{L} y \text{ atom of } \gamma \;\middle|\; \begin{array}{l}\text{the path induced by the atom refinement}\\ \text{of } x \xrightarrow{L} y \text{ in } \rho \text{ leaves } b \text{ at } z\end{array} \right\};$$

then the profile *of b is the multiset of the types of z where z ranges over $\mathbf{v}(b)$.*

The rest of the proof consists in two parts: first, we show that if two non-atomic bags b and b' occurring in some non-branching path of T have the same profile, then we can essentially replace the path between b and b' by a path of constant length (Subclaim 5.4): while this construction is quite elementary, it motivates the intricate definition of the profile of a bag; then, we show that in every non-branching path, if it is sufficiently long, then we can find b and b' satisfying the aforementioned property: this part simply relies on a basic combinatorial argument (see the full version for details).

▷ Subclaim 5.4. Suppose there are two bags b and b' such that: (i) they contain at most k nodes (i.e., not full bags), (ii) they have the same profile, (iii) there is a non-branching path in T between these bags, and (iv) no bags of the path between b and b' (both included) are atomic. Then, there exists a trio $f'\colon \rho' \xrightarrow{hom}\!\!\!\!\rightarrow \alpha'$ and a nice tagged tree decomposition of f' of width at most k that can be obtained by replacing the non-branching path between b and b' in the nice tagged tree-decomposition of $f\colon \rho \xrightarrow{hom}\!\!\!\!\rightarrow \alpha$ by another non-branching path of at most $2k+1$ bags, such that $\alpha \subsetneq \alpha'$.

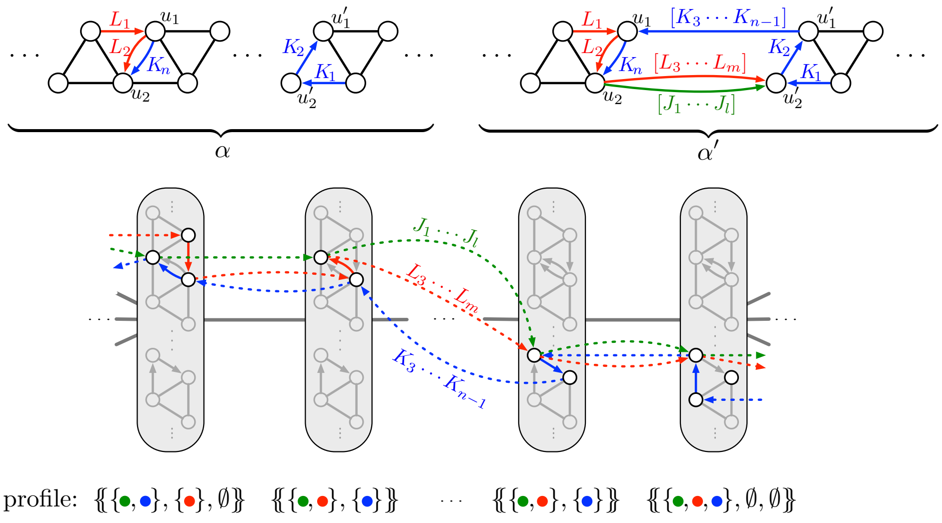


Figure 4 A long non-branching path in the tree decomposition of width 2 of an approximation α. There are two non-full bags in the path with the same profile, and thus the query α can be simplified to α' by applying contractions to the atom refinements involved.

The basic idea behind Subclaim 5.4 is that we the definition of profile was carefully design so that we could contract every refinements between b and b', while preserving every desirable properties on the trio. The construction is illustrated in Figure 4, and both an informal proof and a formal proof can be found in the full version. We can now describe key steps in the proof of Claim 5.2. A formal proof can also be found in the full version.

Proof sketch of Claim 5.2. We claim that, starting from a trio $f\colon \rho \xrightarrow{hom}\!\!\!\!\rightarrow \alpha$ and a locally acyclic nice tagged tree decomposition of f, if we can find a long non-branching path, then an elementary argument (see the full version) yields the existence of two bags b and b' on this path, satisfying the assumptions of Subclaim 5.4, and sufficiently far apart that the construction described in Subclaim 5.4 strictly shortens the path between b and b'. Overall, the iterative application of this construction, which preserves both the niceness and the local acyclicity of the tagged tree decomposition, and only produces bigger approximations (in the sense of containment), yields a trio $f'\colon \rho' \xrightarrow{hom}\!\!\!\!\rightarrow \alpha'$ with a locally acyclic nice tagged tree decomposition of width at most k, whose non-branching paths are all "small", and such that $\alpha \subseteq \alpha'$. $\triangleleft$

Finally, our main lemma follows from Claims 5.1 and 5.2.

Proof of Lemma 3.8. In order to show $\mathrm{App}_{\mathcal{Tw}_k}(\gamma) \subseteq \mathrm{App}^{\star,\leqslant \ell}_{\mathcal{Tw}_k}(\gamma)$ – the other containment being trivial – , pick a trio $f\colon \rho \to \alpha$. Applying Claim 5.1 and then Claim 5.2 yields the existence of a trio $f'\colon \rho' \to \alpha'$ together with a nice tagged tree decomposition $(T', \mathbf{v}', \mathbf{t}')$ of f' such that $\alpha \subseteq \alpha'$ and $(T', \mathbf{v}', \mathbf{t}')$ is locally acyclic, and any non-branching path in T' has length at most $\Theta(\|\gamma\| \cdot (k+1)^{\|\gamma\|})$.

Moreover, we can assume w.l.o.g., by applying Fact 4.2, that every leaf of T' is tagged by at least one atom of ρ'. The local acyclicity of T' implies that if b be a leaf of T', and $\pi \mathrel{\hat{=}} x \xrightarrow{L_1} t_1 \xrightarrow{L_2} \cdots \xrightarrow{L_{n-1}} t_{n-1} \xrightarrow{L_n} y$ is an atom refinement in ρ' of some atom $x \xrightarrow{L} y$ of γ, then if b is tagged by one atom of π this atom must either be $z_0 \xrightarrow{L_1} z_1$ or $z_{n-1} \xrightarrow{L_n} z_n$ by local acyclicity – see e.g. Figure 3 for a visual proof. The number of such atoms in ρ' being bounded by $2\|\gamma\|$, we conclude that T' has at most $2\|\gamma\|$ leaves.

Then, observe that a tree with at most p leaves and whose non-branching paths have length at most q is of height at most[9] $p \cdot q - 1$. We conclude that the height of T' is $\Theta(\|\gamma\|^2 \cdot (k+1)^{\|\gamma\|})$. Using again the local acyclicity of T', observe that the refinement length of ρ' is at most twice the height of T', and hence $\rho' \in \mathrm{Ref}^{\leqslant \ell}(\gamma)$ where $\ell = \Theta(\|\gamma\|^2 \cdot (k+1)^{\|\gamma\|})$. In other words, $\alpha' \in \mathrm{App}^{\star;\leqslant \ell}_{\mathcal{T}\mathfrak{w}_k}(\gamma)$. Hence, we have shown that for all $\alpha \in \mathrm{App}^{\star}_{\mathcal{T}\mathfrak{w}_k}(\gamma)$, there exists $\alpha' \in \mathrm{App}^{\star;\leqslant \ell}_{\mathcal{T}\mathfrak{w}_k}(\gamma)$ such that $\alpha \subseteqq \alpha'$. ◀

6 Queries over simple regular expressions

A *simple regular expression*, or *SRE*, is a regular expression the form a^* for some letter $a \in \mathbb{A}$ or of the form $a_1 + \cdots + a_m$ for some $a_1, \ldots, a_m \in \mathbb{A}$.

Let UCRPQ(SRE) be the set of all UCRPQ whose languages are expressed via SRE expressions. Observe that UCRPQ(SRE) is semantically closed under concatenation, that is, concatenations of SRE expressions can be also expressed in the language. For example, $\gamma(x,y) = x \xrightarrow{a^* \cdot (a+b) \cdot b^*} y$ is equivalent to $\gamma'(x,y) = x \xrightarrow{a^*} z \wedge z \xrightarrow{a+b} z' \wedge z' \xrightarrow{b^*} y$. One interest of UCRPQ(SRE) comes from the fact that it is used widely in practice, as recent studies on SPARQL query logs on Wikidata, DBpedia and other sources show that this kind of regular expressions cover a majority of the queries investigated, e.g., 75% of the "property paths" (C2RPQ atoms) of the corpus of 1.5M queries of [7, Table 15]. An additional interest comes from the fact that the containment problem for UCRPQ(SRE) is much better behaved than for general UCRPQs, since it is in Π_2^p [11, Corollary 5.2], that is, just one level up the polynomial hierarchy compared to the CQ containment problem, which is in NP [9], and in sharp contrast with the costly ExpSpace-complete CRPQ containment problem [8, 12].

We devote this section to showing the following result.

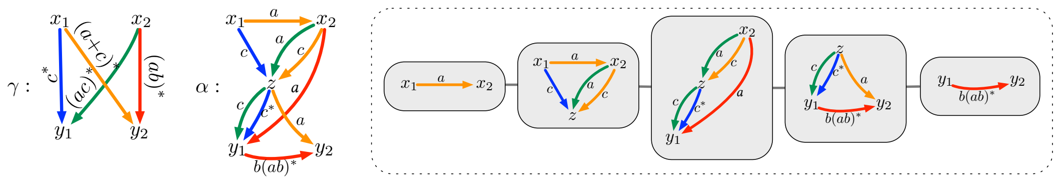
▶ **Theorem 6.1.** *For $k > 1$, the semantic tree-width k problem for* UCRPQ(SRE) *is in Π_2^p.*

Observe that simple regular expressions are closed under sublanguages. Hence, in the light of Theorem 1.5, the maximal under-approximation of a UCRPQ(SRE) query by infinitary unions of CQs of tree-width k is always equivalent to a UCRPQ(SRE) query of tree-width k. We will explain how the construction of the maximal under-approximation of the previous section can be exploited to improve the complexity from 2ExpSpace down to Π_2^p.

6.1 Summary queries

We will first show that the maximal under-approximation of tree-width k of a UC2RPQ can be expressed as a union of polynomial sized "summary" queries. Each summary query represents a union of exponentially-bounded C2RPQs sharing some *common structure*. These are normal UC2RPQ queries extended with some special kind of atoms, called "path-l approximations". Intuitively, a path-l approximation is a maximal under-approximation of tree-width l of queries of the form $\bigwedge_i x_i \xrightarrow{L_i} y_i$ such that $x_i \neq y_j$ for all i,j. Path-l approximations may require an exponential size when represented as UCRPQs. Formally, a *path-l approximation* is a query of the form "$\mathbf{P}_l(X,Y,\gamma)$" where: (i) X, Y, are two disjoint sets of variables of size at most l, (ii) $\gamma(\bar{z})$ is a conjunction of atoms $\bigwedge_{1 \leqslant i \leqslant n} A_i(x_i, y_i)$ where $\bar{z}$ contains all variables of $X \cup Y$, (iii) each A_i is a C2RPQ atom of the form $x_i \xrightarrow{L} y_i$ or $y_i \xrightarrow{L} x_i$ such that x_i is in X and y_i is in Y. We give its semantics in terms of infinitary

[9] The length of a path being its number of nodes, and with the convention that the height of a single node is zero.



■ **Figure 5** The query $\mathbf{P}_l(\{x_1, x_2\}, \{y_1, y_2\}, \gamma)$ (where $l = 2$) on the left contains the approximation α, witnessed by the path decomposition on the right.

unions of CQs. A query like the one before is defined to be equivalent to the (infinitary) union of all queries $\alpha(\bar{z}) \in \mathrm{App}_{\mathcal{T}\mathcal{w}_k}(\gamma)$ that admit a path decomposition of width l having the bag X at the root and Y at the leaf – where a *path decomposition* is defined to be any tree decomposition whose underlying tree is a path. See Figure 5 for an example.

We now simply define a *k-summary query* as a C2RPQ extended with path-l approximation atoms for any $l \leqslant k$, with the expected semantics. The important property of summary queries is that they are exponentially more succinct than the UC2RPQ counterpart for expressing maximal under-approximations, as the next lemma shows.

▶ **Proposition 6.2.** *For every class $\mathcal{L}$ closed under sublanguages, and for every* C2RPQ($\mathcal{L}$) *γ, $\mathrm{App}_{\mathcal{T}\mathcal{w}_k}(\gamma)$ can be expressed as a union of polynomial-sized k-summary queries having only* C2RPQ($\mathcal{L}$) *atoms. Further, one can test in* NP *if a summary query is part of this union. We call $\mathrm{App}^{\mathrm{zip}}_{\mathcal{T}\mathcal{w}_k}(\gamma)$ to any such a union of summary queries.*

Informal proof. As corollary of the proof of Lemma 3.8, we can assume to have $\mathrm{App}_{\mathcal{T}\mathcal{w}_k}(\gamma)$ expressed as a union of C2RPQ($\mathcal{L}$) with a nice tree decomposition of width k with a linear number of leaves, and hence it suffices to replace non-branching paths with path-l approximations. Concretely, for any such C2RPQ α having a witnessing tree decomposition with a long non-branching path, the tree must contain a sub-path whose every bag is non-atomic, and such that it starts and ends in bags of size at most k (by the niceness property). Such a non-atomic non-branching path can be "compressed" by replacing the subquery corresponding to the path with a corresponding path-l approximation query. The resulting summary query will contain α and in turn be contained in γ. Simultaneously applying such replacement to all non-atomic non-branching paths of maximal length then yields a polynomial sized summary query.

Further, given a k-summary query σ, one can test in NP whether there exists an element of $\mathrm{App}^{\star}_{\mathcal{T}\mathcal{w}_k}(\gamma)$ that leads to such summary query by the process just mentioned. This is done by first checking that σ is of the "right shape" (essentially a query of tree-width k when disregarding the path-l approximation atoms), and that it is contained in γ via a polynomial refinement $\rho \in \mathrm{Ref}(\gamma)$ and a strong onto homomorphism to the query resulting from replacing $\mathbf{P}_l(X, Y, \delta)$ atoms with δ in σ. ◀

6.2 Semantic tree-width problem

With the previous results in place, we now show that the semantic tree-width k problem is in Π^p_2 for UCRPQ(SRE), for every $k > 1$.

▶ **Theorem 6.1.** *For $k > 1$, the semantic tree-width k problem for* UCRPQ(SRE) *is in Π^p_2.*

Proof. It suffices to show the statement for any CRPQ(SRE) γ. Remember that γ is of semantic tree-width k if, and only if, $\gamma \subseteqq \mathrm{App}^{\mathrm{zip}}_{\mathcal{T}\mathcal{w}_k}(\gamma)$. The first ingredient to this proof is the fact that this containment has a polynomial counter-example property:

▷ Claim 6.3. If $\gamma \not\subseteq \mathrm{App}^{\mathrm{zip}}_{\mathcal{Tw}_k}(\gamma)$ then there is a polynomial-sized expansion ξ of γ such that $\xi \not\subseteq \mathrm{App}^{\mathrm{zip}}_{\mathcal{Tw}_k}(\gamma)$.

This is because any path $x_0 \xrightarrow{a} \cdots \xrightarrow{a} x_m$ in an expansion ξ of γ such that $m > \|\gamma\|$ and $\xi \subseteq \mathrm{App}^{\mathrm{zip}}_{\mathcal{Tw}_k}(\gamma)$ can be "pumped" to an even longer path of any length greater than m obtaining another expansion ξ' such that $\xi' \subseteq \mathrm{App}^{\mathrm{zip}}_{\mathcal{Tw}_k}(\gamma)$. This implies that a minimal counterexample must be of polynomial size.

The second ingredient is that testing whether CQ is a counterexample is in NP.

▷ Claim 6.4. The problem of testing, given a CQ γ, whether $\gamma \subseteq \mathrm{App}^{\mathrm{zip}}_{\mathcal{Tw}_k}(\gamma)$, is in NP.

Informal proof. We first guess a polynomial-sized k-summary query δ_{zip} and test in NP that it is part of $\mathrm{App}^{\mathrm{zip}}_{\mathcal{Tw}_k}(\gamma)$ by Proposition 6.2. We now guess a valuation $\mu : \mathit{vars}(\delta_{\mathrm{zip}}) \to \mathit{vars}(\gamma)$ and test that it is a homomorphism $\xi \xrightarrow{hom} \gamma$, where ξ is an expansion of the CRPQ resulting from discarding the path-l approximation atoms of δ_{zip}, which can be done in NL. It remains to check that each atom $\mathbf{P}_l(X, Y, \hat{\delta})$ of δ has an expansion $\hat{\xi}$ such that $h : \hat{\xi} \xrightarrow{hom} \gamma$ for a homomorphism such that $h(x) = \mu(x)$ for all $x \in X \cup Y$. This can be done via an NL algorithm using $l + 1$ pointers to traverse the width-l nice path decomposition of $\hat{\xi}$, simultaneously guessing $\hat{\xi}$, the expansion of γ which homomorphically maps to $\hat{\xi}$, and the valuation of h. ◁

As a consequence of the two claims, we obtain a Σ^p_2 algorithm for non-containment of $\gamma \subseteq \mathrm{App}^{\mathrm{zip}}_{\mathcal{Tw}_k}(\gamma)$. We first guess an expansion ξ of γ of polynomial size, and we then test $\xi \not\subseteq \mathrm{App}^{\mathrm{zip}}_{\mathcal{Tw}_k}(\gamma)$ in coNP. This gives a Π^p_2 algorithm for the semantic tree-width k problem. ◀

7 Discussion

We have studied the definability and approximation of UC2RPQ queries by queries of bounded tree-width and shown that the maximal under-approximation in terms of an infinitary union of conjunctive queries of tree-width $k > 1$ can be always effectively expressed as a UC2RPQ of tree-width k (Corollary 3.9). However, while the semantic tree-width 1 problem as shown to be ExpSpace-complete [5, Theorem 6.1, Proposition 6.2], we have left a gap between our lower and upper bounds in Theorem 1.3.

▷ Question 7.1. For $k > 1$, is the semantic tree-width k problem ExpSpace-complete?

We also do not know whether the Π^p_2 bound on the semantic tree-width k problem for UCRPQ(SRE) has a matching lower bound. The known lower bound for the UCRPQ(SRE) containment problem [11, Theorem 5.1] does not seem to be useful to be used in a reduction, since it necessitates queries of arbitrary high tree-width.

It is worth stressing that in [5] two-way navigation plays a crucial part to prove the existence of the maximal under-approximation by a UC2RPQ of tree-width 1 [5, Theorem 5.2], but this feature plays no role whatsoever in our proof – see Theorem 1.5 and Remark 2.5. Moreover, $\mathcal{Tw}_k$ queries enjoy the very nice property of being closed under refinement when $k > 1$ but not when $k = 1$ – see Example 3.7 – which forces Barceló, Romero, and Vardi to introduce the notion of "pseudoacyclicity" [5, §5.2.1], namely the greatest subclass of $\mathcal{Tw}_1$ closed under refinement, while we can directly work with the rather comfortable $\mathrm{App}^{\star}_{\mathcal{Tw}_k}(\gamma)$. On the other hand, graphs of tree-width $k > 1$ being combinatorially less trivial than graphs of tree-width 1, our proof must carefully handle this information, using tagged tree decompositions of Section 4. Similarly to [5, Theorem 6.3] for the case $k = 1$, our results implies that for each $k > 1$ the evaluation problem for UC2RPQs Γ of semantic tree-width k is fixed-parameter tractable (FPT) in the size of the query, i.e. in $O(|G|^{k+1} \cdot f(|\Gamma|))$ for a computable function f, where G is the database given as input. This improves the dependence on the size of the database, namely $O(|G|^{2k+2} \cdot f(|\Gamma|))$, proven by Romero, Barceló and Vardi [15, Corollary IV.12].

▷ Question 7.2 (Also mentionned in [15, §IV-(4))]). Does every r.e. class of CRPQs with FPT evaluation has bounded semantic tree-width?

Finally, as a consequence of the existence of a minimal equivalent CQ [9, Theorem 12], a CQ is equivalent to a CQ of tree-width at most k if, and only if, it is equivalent to a finite union of CQs of tree-width at most k. Example 1.2 suggests that this is false for CRPQs. Both for $k = 1$ (see [5, §6.5]) and $k \geqslant 2$, we do not if the problem of whether a given CRPQ is equivalent to a single CRPQ of tree-width at most k is decidable. More generally, while we know that there exists CRPQs γ_1, γ_2 such that $\gamma_1 \vee \gamma_2$ is not equivalent to a single CRPQ[10], we have a very limited understanding of how much union adds to the expressive power of CRPQs. This begs the following question:

▷ Question 7.3. Is the problem of whether a given UCRPQ (resp. UC2RPQ) is equivalent to a single CRPQ (resp. C2RPQ) decidable?

References

1 Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan Reutter, and Domagoj Vrgoč. Foundations of modern query languages for graph databases. *ACM Comput. Surv.*, 50(5), September 2017. doi:10.1145/3104031.

2 Stefan Arnborg, Derek G Corneil, and Andrzej Proskurowski. Complexity of finding embeddings in a k-tree. *SIAM Journal on Algebraic Discrete Methods*, 8(2):277–284, 1987.

3 Pablo Barceló, Leonid Libkin, and Miguel Romero. Efficient approximations of conjunctive queries. In *Proceedings of the 31st ACM SIGMOD-SIGACT-SIGAI symposium on Principles of Database Systems*, pages 249–260, 2012. doi:10.1145/2213556.2213591.

4 Pablo Barceló and Miguel Romero. The Complexity of Reverse Engineering Problems for Conjunctive Queries. In Michael Benedikt and Giorgio Orsi, editors, *20th International Conference on Database Theory (ICDT 2017)*, volume 68 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 7:1–7:17, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.ICDT.2017.7.

5 Pablo Barceló, Miguel Romero, and Moshe Y. Vardi. Semantic acyclicity on graph databases. *SIAM Journal on computing*, 45(4):1339–1376, 2016. doi:10.1137/15M1034714.

6 Pablo Barceló Baeza. Querying graph databases. In *Proceedings of the 32nd ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, PODS '13, pages 175–188, New York, NY, USA, 2013. Association for Computing Machinery. doi:10.1145/2463664.2465216.

7 Angela Bonifati, Wim Martens, and Thomas Timm. An analytical study of large SPARQL query logs. *VLDB Journal*, 29(2):655–679, 2020. doi:10.1007/s00778-019-00558-9.

8 Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Moshe Y. Vardi. Containment of conjunctive regular path queries with inverse. In *Principles of Knowledge Representation and Reasoning (KR)*, pages 176–185, 2000.

[10] See [4, Figure 1 & Example 21, p. 15]: by denoting $\gamma_1, \gamma_2, \gamma_3$ the CQs in the top-left, below-left and right parts of Figure 1, respectively, their example shows that any CRPQ that contains both γ_1 and γ_2 must also contain γ_3. As γ_3 is contained in neither γ_1 nor γ_2, our claim follows.

9 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.

10 Diego Figueira. Foundations of graph path query languages – course notes for the reasoning web summer school 2021. In *Reasoning Web. Declarative Artificial Intelligence – 17th International Summer School 2021, Leuven, Belgium, September 8-15, 2021, Tutorial Lectures*, volume 13100 of *Lecture Notes in Computer Science*, pages 1–21. Springer, 2021. doi:10.1007/978-3-030-95481-9_1.

11 Diego Figueira, Adwait Godbole, S. Krishna, Wim Martens, Matthias Niewerth, and Tina Trautner. Containment of simple conjunctive regular path queries. In *Principles of Knowledge Representation and Reasoning (KR)*, 2020. URL: https://hal.archives-ouvertes.fr/hal-02505244.

12 Daniela Florescu, Alon Levy, and Dan Suciu. Query containment for conjunctive queries with regular expressions. In *ACM Symposium on Principles of Database Systems (PODS)*, pages 139–148. ACM Press, 1998. doi:10.1145/275487.275503.

13 Ton Kloks. *Treewidth: computations and approximations.* Lecture Notes in Computer Science. Springer, 1994. doi:10.1007/BFb0045375.

14 Jaroslav Nešetřil and Patrice Ossona de Mendez. *Sparsity – Graphs, Structures, and Algorithms*, volume 28 of *Algorithms and combinatorics.* Springer, 2012. doi:10.1007/978-3-642-27875-4.

15 Miguel Romero, Pablo Barceló, and Moshe Y. Vardi. The homomorphism problem for regular graph patterns. In *Annual Symposium on Logic in Computer Science (LICS)*, pages 1–12. IEEE Computer Society Press, 2017. doi:10.1109/LICS.2017.8005106.

Work-Efficient Query Evaluation with PRAMs

Jens Keppeler
TU Dortmund University, Germany

Thomas Schwentick
TU Dortmund University, Germany

Christopher Spinrath
TU Dortmund University, Germany

Abstract

The paper studies query evaluation in parallel constant time in the PRAM model. While it is well-known that all relational algebra queries can be evaluated in constant time on an appropriate CRCW-PRAM, this paper is interested in the efficiency of evaluation algorithms, that is, in the number of processors or, asymptotically equivalent, in the work. Naive evaluation in the parallel setting results in huge (polynomial) bounds on the work of such algorithms and in presentations of the result sets that can be extremely scattered in memory. The paper first discusses some obstacles for constant time PRAM query evaluation. It presents algorithms for relational operators that are considerably more efficient than the naive approaches. Further it explores three settings, in which efficient sequential query evaluation algorithms exist: acyclic queries, semi-join algebra queries, and join queries – the latter in the worst-case optimal framework. Under natural assumptions on the representation of the database, the work of the given algorithms matches the best sequential algorithms in the case of semi-join queries, and it comes close in the other two settings. An important tool is the compaction technique from Hagerup (1992).

2012 ACM Subject Classification Theory of computation → Shared memory algorithms; Theory of computation → Database query processing and optimization (theory)

Keywords and phrases PRAM, query evaluation, work-efficient, parallel, acyclic queries, free-connex queries

Digital Object Identifier 10.4230/LIPIcs.ICDT.2023.16

Related Version The full version of this paper contains some proofs and additional details omitted in this version due to space constraints.
Full Version: `https://arxiv.org/abs/2301.08178` [19]

Acknowledgements We are grateful to Uri Zwick for clarifications regarding results in [13] and to Jonas Schmidt and Jennifer Todtenhoefer for careful proofreading. We thank Martin Dietzfelbinger for helpful discussions. Furthermore, we thank the reviewers of ICDT for many insightful suggestions.

1 Introduction

Parallel query evaluation has been an active research area during the last 10+ years. Parallel evaluation algorithms have been thoroughly investigated, mostly using the Massively Parallel Communication (MPC) model [7]. For surveys we refer to [20, 15].

The MPC model is arguably very well-suited to study parallel query evaluation. However, it is not the only model for parallel query evaluation. Indeed, there is also the Parallel Random Access Machine (PRAM) model, a more “theoretical” model which allows for a more fine-grained analysis of parallel algorithms, particularly in non-distributed settings. It was shown by Immerman [16, 17] that PRAMs with polynomially many processors can evaluate first-order formulas and thus relational algebra queries in time $\mathcal{O}(1)$.

In the study of PRAM algorithms it is considered very important that algorithms perform well compared with sequential algorithms. The overall number of computation steps in a PRAM-computation is called its *work*. An important goal is to design parallel algorithms that are *work-optimal* in the sense that their work asymptotically matches the running time of the best sequential algorithms.

Obviously, for $\mathcal{O}(1)$-time PRAM algorithms the work and the number of processors differ at most by a constant factor. Thus, the result by Immerman shows that relational algebra queries can be evaluated with polynomial work. Surprisingly, to the best of our knowledge, work-efficiency of $\mathcal{O}(1)$-time PRAM algorithms for query evaluation has not been investigated in the literature. This paper is meant to lay some groundwork in this direction.

The proof of the afore-mentioned result that each relational algebra query can be evaluated in constant-time by a PRAM with polynomial work is scattered over several papers. It consists basically of three steps, translating from queries to first-order logic formulas [11], to bounded-depth circuits [6], and then to PRAMs [16]. It was not meant as a "practical translation" of queries and does not yield one. However, it is not hard to see directly that the operators of the relational algebra can, in principle, be evaluated in constant time on a PRAM. It is a bit less obvious, though, how the output of such an operation is represented, and how it can be fed into the next operator.

▶ **Example 1.1.** Let us consider an example to illustrate some issues of constant-time query evaluation on a PRAM. Let q be the following conjunctive query, written in a rule-based fashion, for readability.

$$q(x, y, z) \leftarrow E(x, x_1), E(x_1, x_2), E(y, y_1), E(y_1, y_2), E(z, z_1), E(z_1, z_2), R(x_2, y_2, z_2)$$

A (very) naive evaluation algorithm can assign one processor to each combination of six E-tuples and one R-tuple, resulting in work $\mathcal{O}(|E|^6|R|)$. Since the query is obviously acyclic, it can be evaluated more efficiently in the spirit of Yannakakis' algorithm. For simplicity, we ignore the semi-join step of the Yannakakis algorithm. The join underlying the first two atoms $E(x, x_1), E(x_1, x_2)$ can be computed as a sub-query $q_1(x, x_2) \leftarrow E(x, x_1), E(x_1, x_2)$. This can be evaluated by $|E|^2$ many processors, each getting a pair of tuples $E(a_1, a_2), E(b_1, b_2)$ and producing output tuple (a_1, b_2) in case $a_2 = b_1$. The output can be written into a 2-dimensional table, indexed in both dimensions by the tuples of E. In the next round, the output tuples can be joined with tuples from R to compute $q_2(x, y_2, z_2) \leftarrow E(x, x_1), E(x_1, x_2), R(x_2, y_2, z_2)$. However, since it is not clear in advance, which entries of the two-dimensional table carry q_1-tuples, the required work is about $|E|^2 \cdot |R|$. Output tuples of q_2 can again be written into a 2-dimensional table, this time indexed by one tuple from E (for x) and one tuple from R (for y_2 and z_2). Proceeding in a similar fashion, q can be evaluated with work $\mathcal{O}(|E|^3|R|)$. Other evaluation orders are possible but result in similar work bounds. In terms of the input size of the database, this amounts to $\mathcal{O}(\mathsf{IN}^4)$, whereas Yannakakis' algorithm yields $\mathcal{O}(\mathsf{IN} \cdot \mathsf{OUT})$ in the sequential setting, where IN denotes the number of tuples in the given relations and OUT the size of the query result, respectively.

Let us take a look at the representation of the output of this algorithm. The result tuples reside in a 3-dimensional table, each indexed by a tuple from E. It is thus scattered over a space of size $|E|^3$, no matter the size of the result. Furthermore, the same output tuple might occur several times due to several valuations. To produce a table, in which each output tuples occurs exactly once, a deduplication step is needed, which could yield additional work in the order of $|E|^6$.

The example illustrates two challenges posed by the $\mathcal{O}(1)$-time PRAM setting, which will be discussed in more detail in Section 3.

- It is, in general, not possible to represent the result of a query in a compact form, say, as an array, contiguously filled with result tuples. This obstacle results here in upper bounds in terms of the size of the input database, but not in the size of the query result.
- It might be necessary to deduplicate output (or intermediate) relations, but, unfortunately, this cannot be done by sorting a relation, since sorting is not possible in $\mathcal{O}(1)$-time on a PRAM, either.

We will use compactification techniques for PRAMs from [14] to deal with the first challenge. The second challenge is addressed by a suitable representation of the relations. Besides the setting without any assumptions, we consider the setting, where data items are mapped to an initial segment of the natural numbers by a dictionary, and the setting where the relations are represented by ordered arrays.

We show that, for each $\varepsilon > 0$, there is a Yannakakis-based algorithms for acyclic join queries in the dictionary setting with an upper work bound of $\mathcal{O}(\mathsf{IN} \cdot \mathsf{OUT})^{1+\varepsilon}$. Two other results are work-optimal algorithms for queries of the semijoin algebra and almost worst-case and work-optimal algorithms for join queries.

We emphasize that the stated result does not claim a fixed algorithm that has an upper work bound W such that, for every $\varepsilon > 0$, it holds $W \in \mathcal{O}(\mathsf{IN} \cdot \mathsf{OUT})^{1+\varepsilon}$. It rather means that there is a uniform algorithm that has ε as a parameter and has the stated work bound, for each fixed $\varepsilon > 0$. The linear factor hidden in the $\mathcal{O}$-notation thus depends on ε. This holds analogously for our other upper bounds of this form.

This paper consists roughly of three parts, each of which has some contributions to our knowledge on work-efficient constant-time PRAM query evaluation.

The first part presents some preliminaries in Section 2, discusses the framework including some of our (typical) assumptions and data structures in Section 4, surveys lower bound results that pose challenges for $\mathcal{O}(1)$-time PRAM query evaluation in Section 3, and presents some basic operations that will be used in our query evaluation algorithms in Section 4.

The second part presents algorithms for these basic operations in Section 5 and for relational operators in Section 6.

After this preparation, the third part studies query evaluation in three settings in which (arguably) efficient algorithms exist for sequential query evaluation: the semi-join algebra (Subsection 7.1), acyclic queries (Subsection 7.2), and worst-case optimal join evaluation (Subsection 7.3).

Related work. Due to space limitations, we only mention two other related papers. In a recent paper, query evaluation by circuits has been studied [28]. Although this is in principle closely related, the paper ignores polylogarithmic depth factors and therefore does not study $\mathcal{O}(1)$-time. The work of $\mathcal{O}(1)$-time PRAM algorithms has recently been studied in the context of dynamic complexity, where the database can change and the algorithms need to *maintain* the query result [25].

2 Preliminaries

In this section, we fix some notation and recall some concepts from database theory and PRAMs that are relevant for this paper. For a natural number n, we write $[n]$ for $\{1, \ldots, n\}$.

A *database schema* Σ is a finite set of relation symbols, where each symbol R is equipped with a finite set $\mathtt{attr}(R)$ of attributes. A tuple $t = (a_1, \ldots, a_{|X|})$ over a finite list $X = (A_1, \ldots, A_k)$ of attributes has, for each i, a value a_i for attribute A_i. Unless we are interested in the lexicographic order of a relation, induced by X, we can view X as a set. An R-relation is a finite set of tuples over $\mathtt{attr}(R)$. The arity of R is $|\mathtt{attr}(R)|$. For $Y \subseteq X$, we write $t[Y]$ for the restriction of t to Y. For $Y \subseteq \mathtt{attr}(R)$, $R[Y] = \{t[Y] \mid t \in R\}$. A database D over Σ consists of an R-relation $D(R)$, for each $R \in \Sigma$. We usually write R instead of $D(R)$ if D is understood from the context. That is, we do not distinguish between relations and relation symbols. The size $|R|$ of a relation R is the number of tuples in R. By $|D|$ we denote the number of tuple entries in database D. For details on (the operators of) the relational algebra, we refer to [3]. We always assume a fixed schema and therefore a fixed maximal arity of tuples.

Parallel Random Access Machines (PRAMs). A *parallel random access machine* (PRAM) consists of a number of processors that work in parallel and use a shared memory. The memory is comprised of memory cells which can be accessed by a processor in $\mathcal{O}(1)$ time. Furthermore, we assume that the usual arithmetic and bitwise operations can be done in $\mathcal{O}(1)$ time by a processor. In particular, since the schema is considered fixed, a database tuple can be loaded, compared, etc. in $\mathcal{O}(1)$ time by one processor.

We mostly use the Concurrent-Read Concurrent-Write model (CRCW-PRAM), i.e. processors are allowed to read and write concurrently from and to the same memory location. More precisely, we mainly assume the *arbitrary* PRAM model: if multiple processors concurrently write to the same memory location, one of them, "arbitrarily", succeeds. For some algorithms the *common* model would suffice, where all processors need to write the same value into the same location. We sometimes also use the weaker Exclusive-Read Exclusive-Write model (EREW-PRAM), where concurrent access is forbidden and the CREW-PRAM, where concurrent writing is forbidden. The work w of a PRAM computation is the sum of the number of all computation steps of all processors made during the computation. We define the space s required by a PRAM computation as the maximal index of any memory cell accessed during the computation. We refer to [18] for more details on PRAMs and to [26, Section 2.2.3] for a discussion of alternative space measures.

In principle, we assume that relations are stored as sequences of tuples, i.e., as arrays. Informally an array $\mathcal{A}$ is a sequence $t_1, \ldots, t_N$ and it represents a relation R, if each tuple from R appears once as some t_i. In Section 4, we describe more precisely, how databases are represented for our PRAM algorithms.

3 Obstacles

We next discuss some obstacles that pose challenges for $\mathcal{O}(1)$-time parallel algorithms for query evaluation. They stem from various lower bound results from the literature.

The first obstacle, already mentioned in the introduction, is that we cannot expect that query results can be stored in arrays in a compact fashion, that is, as a sequence $t_1, \ldots, t_m$ of tuples, for a result with m tuples. This follows from the following lower bound on the *linear approximate compaction problem*, where the, somewhat relaxed, goal is to move m scattered tuples into a target array of size $4m$.

▶ **Proposition 3.1** ([22, Theorem 4.1]). *Solving the linear approximate compaction problem on a randomized strong priority CRCW-PRAM requires $\Omega(\log^* n)$ expected time.*

Since the PRAM model of that bound is stronger than the arbitrary CRCW model, it applies to our setting.

The following theorem illustrates, how this lower bound restrains the ability to compute query results in a compact form, even if the input relations are given by compact arrays. Analogous results can be shown for simple projection and selection queries.

▶ **Theorem 3.2.** *Let q be the conjunctive query defined by $q : H(x) \leftarrow R(x), S(x)$. Every algorithm, which, upon input of arrays for relations R and S, computes an array of size $|q(D)|$ for $q(D)$ without empty cells, requires $\omega(1)$ time. This holds even if the arrays for R and S are compact and their entries are lexicographically ordered.*

Proof sketch. Towards a contradiction, we assume that there is an algorithm on a PRAM which computes in constant time, upon input of an input database D, an array of size $|q(D)|$ for the query result $q(D)$. This can be used to solve the linear approximate compaction problem in constant time as follows. Let $\mathcal{A}$ be an instance for the linear approximate compaction problem with m non-empty cells. In the first step create an array $\mathcal{A}_R$ of size n for the unary relation R that consists of even numbers 2 to $2n$. This can be done in parallel in constant time with n processors.

In the second step, store, for every $i \in \{1,\dots,n\}$, the value $2i$ in the array $\mathcal{A}_S$ of size n for relation S if $\mathcal{A}[i]$ has a value (i.e. the i-th cell is not empty), and $2i+1$, otherwise. Hence, the query result of q exactly consists of even numbers $2i$ where i is an index such that $\mathcal{A}[i]$ has a value. From the array for the query result a solution for the linear approximate compaction problem can be obtained by replacing every value $2i$ in the result by $\mathcal{A}[i]$. The size of the compact array is m.

All in all, the algorithm takes constant time to solve the linear approximate compaction problem. This is a contradiction to Proposition 3.1. We note that by construction, the arrays for the relations R and S are ordered and have no empty cells. ◀

As a consequence of Theorem 3.2, our main data structure to represent relations are arrays that might contain *empty cells* that do not correspond to tuples in the relation.

As the example in the introduction illustrated, it is important that intermediate results can be compacted to some extent. Indeed, processors can be assigned to all cells of an array, but not so easily to only the non-empty cells. We extensively use a classical technique by Hagerup [14] that yields some (non-linear) compaction (see Proposition 5.1 for the statement of this result). However, Hagerup's compaction algorithm does not preserve the order of the elements in the array, so ordered arrays with non-empty cells are transformed into more compact but unordered arrays.

This brings us to another notorious obstacle for $\mathcal{O}(1)$-time parallel algorithms: they can not sort with polynomially many processors, even not in a (slightly) non-compact fashion. The *padded sort problem* asks to sort n given items into an array of length $n + o(n)$ with empty cells in spots without an item.

▶ **Proposition 3.3** ([22, Theorem 4.2]). *Solving the padded sort problem on a randomized strong priority CRCW-PRAM requires $\Omega(\log^* n)$ expected time.*

Thus, we cannot rely on sorting as an intermediate operator, either.

Yet another weakness of $\mathcal{O}(1)$-time parallel algorithms is counting. It follows readily from the equivalence with polynomial-size constant-depth circuits that (reasonable) CRCW-PRAMs cannot tell whether the number of ones in a sequence of zeros and ones is even [12, 2], let alone count them. These obstacles apply in particular to the evaluation of aggregate queries and for query evaluation under bag semantics with explicit representation of multiplicities of tuples. However, we do not study any of those in this paper.

Note. It turned out at submission time of this paper that we had missed a paper by Goldberg and Zwick [13], that contains two improvements to the above: it shows that *ordered* compaction and approximate counting, up to a factor of $1 + \frac{1}{(\log n)^a}$, for any $a > 0$, are possible in constant time with work $\mathcal{O}(n^{1+\varepsilon})$. In fact both results rely on the same technique for computing consistent approximate prefix sums. We discuss this issue further in our conclusion.

4 Basics

Query evaluation algorithms often use additional data structures like index structures. We consider two different kinds of such data structures for our $\mathcal{O}(1)$-time parallel algorithms.

The first setting that we consider is that of dictionary-based compressed databases, see, e.g., [10]. In a nutshell, the database has a dictionary that maps data values to natural numbers and internally stores and manipulates tuples over these numbers to improve performance. Such dictionaries are often defined attribute-wise, but for the purpose of this paper this does not matter. Query evaluation does not need to touch the actual dictionaries, it only works with the numbers. In this paper, we write "in the presence of a dictionary" or "in the dictionary setting" to indicate that we assume that such a dictionary exists for the database D at hand, and that it uses numbers of size at most $\mathcal{O}(|D|)$. In particular, the database relations then only contain numbers of this size.

In the other *ordered* setting, we assume that database relations are represented by ordered arrays, for each order of its attributes. In particular, we assume a linear order on the data values.

Arrays. As mentioned before, we assume in this paper that relations are stored in 1-dimensional arrays, whose entries are tuples that might be augmented by additional data.

More formally, an array $\mathcal{A}$ is a sequence of consecutive memory cells. The number of cells is its *size* $|\mathcal{A}|$. By $\mathcal{A}[i]$ for $1 \leq i \leq |\mathcal{A}|$ we refer to the i-th cell of $\mathcal{A}$ and to the (current) content of that cell, and we call i its index. We assume that the size of an array is always available to all processors (for instance, it might be stored in a "hidden" cell with index 0). Given an index i, any processor can access cell $\mathcal{A}[i]$ in $\mathcal{O}(1)$ time with $\mathcal{O}(1)$ work.

In this paper, a cell $\mathcal{A}[i]$ of an array always holds a distinguished database tuple $t = \mathcal{A}[i].t$ (over some schema) and a flag that indicates whether the cell is inhabited.[1] There might be additional data, e.g., further Boolean flags and links to other cells of (possibly) other arrays.

We say that an array *represents* a relation R if some inhabited cell holds tuple t, for each tuple t of R, and no inhabited cells contain other tuples. It represents R *concisely*, if each tuple occurs in exactly one inhabited cell. To indicate that an array represents a relation R we usually denote it as $\mathcal{A}_R$, $\mathcal{A}'_R$, etc.

An array $\mathcal{A}$ that represents a relation R is *ordered* if it is lexicographically ordered with respect to some ordered list X of the attributes from R's schema in the obvious sense. Its order is Y-compatible, for a set Y of attributes, if the attributes of Y form a prefix of X (hence the order induces a partial order with respect to Y).

We often consider the *induced tuple sequence* $t_1, \ldots, t_{|\mathcal{A}|}$ of an array $\mathcal{A}$. Here, $t_i = \mathcal{A}[i].t$ is a *proper* tuple, if $\mathcal{A}[i]$ is inhabited, or otherwise t_i is the *empty tuple* $\perp$.

▶ **Example 4.1.** The tuple sequence $[(1,5), \perp, (3,4), (8,3), (1,5), \perp, \perp, (7,3)]$ from an array $\mathcal{A}$ of size eight has five proper tuples and three empty tuples. It represents the relation $R = \{(1,5), (3,4), (8,3), (7,3)\}$, but *not* concisely. The sequence $[(1,5), (3,4), (7,3), \perp, (8,3)]$ represents R concisely and ordered with respect to the canonical attribute order.

Operations and links. Before we explain the basic operations used by our evaluation algorithms we illustrate some aspects by an example.

[1] If a cell is not inhabited, its data is basically ignored.

▶ **Example 4.2.** We sketch how to evaluate the projection $\pi_B(R)$ given the array $\mathcal{A} = [(1,5),(3,4),(7,3),\perp,(8,3)]$ from Example 4.1.

First, with the operation `Map` the array $\mathcal{A}'[5,4,3,\perp,3]$ is computed and each tuple $\mathcal{A}[i]$ is augmented by a link to $\mathcal{A}'[i]$ and vice versa. To achieve this, the tuples from $\mathcal{A}$ are loaded to processors $1,\dots,5$, each processor applies the necessary changes to its tuple, and then writes the new tuple to the new array $\mathcal{A}'$. We note that it is not known in advance which cells of $\mathcal{A}$ are inhabited and therefore, we need to assign one processor for each cell. Each processor only applies a constant number of steps, so the overall work for the `Map`-operation is $\mathcal{O}(|\mathcal{A}|)$.

To get an array that represents $\pi_B(R)$ concisely, a second operation eliminates duplicates. To this end, it creates a copy $\mathcal{A}''$ of $\mathcal{A}'$ and checks with one processor for each pair (i,j) of indices with $i < j$ in parallel, whether $\mathcal{A}''[i] = \mathcal{A}''[j]$ holds. If $\mathcal{A}''[i] = \mathcal{A}''[j]$ holds, then cell $\mathcal{A}''[i]$ is made uninhabited. Lastly, the algorithm creates links from every cell in $\mathcal{A}'$ to the unique inhabited cell in $\mathcal{A}''$ holding the same tuple. To do this, it checks with one processor for each pair (i,j) of indices with $i < j$ in parallel, whether $\mathcal{A}'[i] = \mathcal{A}''[j]$ holds and $\mathcal{A}''[j]$ is inhabited. If this is the case, the processor for (i,j) augments the cell $\mathcal{A}'[i]$ with a link to $\mathcal{A}''[j]$ and vice versa. Note that multiple processors might attempt to augment a cell $\mathcal{A}''[j]$ with a link to a cell of $\mathcal{A}'$; but since this happens in parallel only one processor will be successful. Overall, we get 2-step-links from $\mathcal{A}$ to $\mathcal{A}''$ and 2-step-links from $\mathcal{A}''$ to "representatives" in $\mathcal{A}$.

The second operation has a work bound of $\mathcal{O}(|\mathcal{A}|^2)$ because it suffices to assign one processor for each pair (i,j) of indices and each processor only applies a constant number of steps. We will show in Section 5 that work bounds $\mathcal{O}(|\mathcal{A}|)$ and $\mathcal{O}(|\mathcal{A}|^{1+\varepsilon})$ can be achieved for eliminating duplicates in the dictionary and ordered setting, respectively (cf. Lemma 5.10).

Of course, one might skip the intermediate writing and reading of tuples of $\mathcal{A}'$. We will often blur the distinction whether tuples reside in an array or within a sequence of processors.

Basic operations. Next, we describe some basic operations which we will use as building blocks in the remainder of this paper to query evaluation algorithms for PRAMs. We will describe algorithms for them in the next section.

Just as in Example 4.2, the operations usually get arrays as input, produce arrays as output, augment tuples and add links between tuples. In fact, each time a tuple of a new array results from some tuple of an input array we silently assume that (possibly mutual) links are added.

`Concatenate`$(\mathcal{A},\mathcal{B})$ computes a new array $\mathcal{C}$ of size $|\mathcal{A}| + |\mathcal{B}|$ consisting of the tuples from $\mathcal{A}$ followed by the tuples from $\mathcal{B}$ in the obvious way. As usual, mutual links are added, in particular, the link from $\mathcal{B}[1]$ indicates where the tuples from $\mathcal{B}$ start in $\mathcal{C}$.

`Map`$(\mathcal{A}, f)$ returns an array $\mathcal{B}$ with $\mathcal{B}[i] = f(\mathcal{A}[i])$, where f is a function that maps proper tuples in $\mathcal{A}$ to empty or proper (possibly augmented) tuples and empty to empty tuples.

`Partition`$(\mathcal{A}, n, g)$ yields n arrays $\mathcal{A}_1,\dots,\mathcal{A}_n$ of size $|\mathcal{A}|$ and adds links. Here, g is a function that maps proper tuples in $\mathcal{A}$ to numbers in $[n]$. For every $i \in [|\mathcal{A}|]$ and every $j \in [n]$, $\mathcal{A}_j[i] = \mathcal{A}[i]$, if $\mathcal{A}[i]$ is inhabited and $g(\mathcal{A}[i]) = j$, otherwise $\mathcal{A}_j[i]$ is uninhabited.

`Compact`$_\varepsilon(\mathcal{A})$ copies the proper tuples in $\mathcal{A}$ into distinct cells of an array $\mathcal{B}$ of size at most $b^{i\varepsilon}$, where $i \le \frac{1}{\varepsilon}(1+\varepsilon)$ is a non-negative integer[2] such that $b^{i\varepsilon} \le n^{1+\varepsilon} b^{\varepsilon}$, and b is $|\mathcal{A}|$ or an upper bound for the (possibly unknown) number n of proper tuples in $\mathcal{A}$, given as an optional parameter. We refer to i as the "compaction parameter" and note that it can be inferred from the size of $\mathcal{B}$. Mutual links are added as usual.

[2] In the special case $n = 0$, $i = 0$ is required.

`SearchRepresentatives`$(\mathcal{A}, \mathcal{B})$ links every inhabited cell $\mathcal{A}[i]$ to an inhabited representative cell $\mathcal{B}[j]$, such that $\mathcal{B}[j].t = \mathcal{A}[i].t$ holds, if such a cell exists. Furthermore, for every i_1, i_2 with $\mathcal{A}[i_1] = \mathcal{A}[i_2] \neq \bot$, both $\mathcal{A}[i_1]$ and $\mathcal{A}[i_2]$ are linked to the same representative. If required a copy of $\mathcal{B}$ might be produced in which representatives are marked. We stress that $\mathcal{B}$ does not have to represent its relation concisely, and, in fact, the operation is used to remove duplicates.

`Deduplicate`$(\mathcal{A})$ chooses one representative tuple for each tuple-value, marks the remaining cells as uninhabited and redirects incoming links from other arrays towards the representatives, if possible.

5 Algorithmic Techniques and Algorithms for Basic Array Operations

In this section, we first describe some important algorithmic techniques and present algorithms for the basic operations established in Section 4, afterwards.

Compaction. To implement the operation $\texttt{Compact}_\varepsilon$, we will utilise the following classical result by Hagerup, whose formulation is slightly adapted to our setting.

▶ **Proposition 5.1** ([14], Unnumbered theorem, p. 340)**.** *For every $\varepsilon > 0$, there is a $\mathcal{O}(1)$-time parallel algorithm that, given an array $\mathcal{A}$ and a number k, copies the proper tuples in $\mathcal{A}$ to distinct cells of an array of size at most $k^{1+\varepsilon}$ or detects that $\mathcal{A}$ contains more than k proper tuples. The algorithm requires $\mathcal{O}(|\mathcal{A}|)$ work and space on an arbitrary CRCW-PRAM.*

The space bound is only implicit in [14]. For the sake of convenience, we give a detailed account of the algorithm in the full version of this paper [19], including an analysis of the space requirements.

Array hash tables. In the presence of dictionaries we use *array hash tables* which associate each inhabited cell in $\mathcal{A}$ with a number from $[|\mathcal{A}|]$, such that $\mathcal{A}[i], \mathcal{A}[j]$ get the same number if and only if $\mathcal{A}[i].t = \mathcal{A}[j].t$ holds. Array hash tables can be efficiently computed.

▶ **Lemma 5.2.** *There is a $\mathcal{O}(1)$-time parallel algorithm that, in the presence of a dictionary, computes an array hash table for a given array $\mathcal{A}$, and requires $\mathcal{O}(|\mathcal{A}|)$ work and $\mathcal{O}(|\mathcal{A}| \cdot |D|)$ space on an arbitrary CRCW-PRAM.*

We note that due to the "arbitrary" resolution of concurrent write, the result of such a computation is not uniquely determined by the relations.

Proof sketch. Let $A_1, \ldots, A_\ell$ be the attributes of the relation R represented by $\mathcal{A}$ in an arbitrary but fixed order and define $X_j = \{A_1, \ldots, A_j\}$ for all $j \in \{1, \ldots, \ell\}$. The algorithm inductively computes hash values for tuples in $R[X_j]$ for increasing j from 1 to ℓ.

The idea is to assign, to each tuple $t \in R[X_j]$, a processor number in the range $\{1, \ldots, |\mathcal{A}|\}$ as hash value and augment each cell of $\mathcal{A}$ containing a proper tuple t' with $t'[X] = t$ by this hash value. Since the same (projected) tuple $t \in R[X_j]$ might occur in multiple, pairwise different, cells of $\mathcal{A}$, it does not suffice to load all tuples in $\mathcal{A}$ to $|\mathcal{A}|$ processors and let each processor augment the tuple loaded to it by its processor number: multiple (different) numbers might get assigned to the same tuple (in different cells of $\mathcal{A}$). To resolve these conflicts, the algorithm utilizes the presence of a dictionary and the hash values for tuples in $R[X_{j-1}]$.

For the base case $j = 1$ the algorithm allocates an auxiliary array of size $\mathcal{O}(|D|)$ and loads the tuples in $\mathcal{A}$ to processors 1 to $|\mathcal{A}|$. To be more precise, for each $i \in \{1, \ldots, |\mathcal{A}|\}$, the tuple t_i in cell $\mathcal{A}[i]$ is loaded to processor i. Recall that, in the dictionary setting, each value in the active domain is a number of size at most $\mathcal{O}(|D|)$. Thus, the projection $t[A_1]$ can be used as index for the auxiliary array. Each processor i with a proper tuple writes its processor number i into cell $t_i[A_1]$ of the auxiliary array and then assigns to t_i the value actually written at position $t_i[A_1]$. Note that, for each value a, all processors i with $t_i[A_1] = a$ will assign the same value to their tuple t_i, since precisely one processor among the processors with $t_i[A_1] = a$ succeeds in writing its number to cell $t_i[A_1]$ on an arbitrary CRCW-PRAM. This can be done with $\mathcal{O}(|\mathcal{A}|)$ work and $\mathcal{O}(|D|)$ space.

For $j > 1$ the algorithm proceeds similarly but also takes, for a tuple t, the hash value $h_{j-1}(t[X_{j-1}])$ for $t[X_{j-1}]$ into account, in addition to $t[A_j]$. For this purpose, the algorithm first computes the hash values for $R[X_{j-1}]$. It then allocates an auxiliary array of size $\mathcal{O}(|\mathcal{A}| \cdot |D|)$ which is interpreted as two-dimensional array and each processor i writes its number into cell $(h_{j-1}(t_i[X_{j-1}]), t_i[X_j])$ of the auxiliary array (if t_i is a proper tuple). The number in this cell is then the hash value for $t_i[X_j]$.

Writing and reading back the processor numbers requires $\mathcal{O}(|\mathcal{A}|)$ work and the auxiliary array requires $\mathcal{O}(|\mathcal{A}| \cdot |D|)$ space. The same bounds hold for the recursive invocations of `ComputeHashvalues`. Since the recursion depth is ℓ, the procedure requires $\mathcal{O}(\ell \cdot |\mathcal{A}|) = \mathcal{O}(|\mathcal{A}|)$ work and, because the space for the auxiliary arrays can be reused, $\mathcal{O}(|\mathcal{A}| \cdot |D|)$ space in total. ◀

Since most of our query evaluation algorithms rely on these two techniques, we adopt the arbitrary CRCW-PRAM as our standard model and refer to it simply by *CRCW-PRAM*.

Search in ordered arrays. In sequential database processing, indexes implemented by search trees play an important role, in particularly for the test whether a given tuple is in a given relation. We use ordered arrays instead. Our search algorithm for ordered arrays uses links from each cell to the next and previous inhabited cell. We refer to those links as predecessor and successor links, respectively, and say an array is *fully linked* if it has predecessor and successor links.

▶ **Proposition 5.3.** *For every $\varepsilon > 0$, there is a $\mathcal{O}(1)$-time parallel algorithm that computes, for an array $\mathcal{A}$, predecessor and successor links with work $\mathcal{O}(|\mathcal{A}|^{1+\varepsilon})$ on a common CRCW-PRAM.*

Proof sketch. We describe the computation of predecessor links. Successor links can be computed analogously. Let $n = |\mathcal{A}|$ and $\delta = \frac{\varepsilon}{2}$. In the first round, the algorithm considers subintervals of length n^δ and establishes predecessor links within them. To this end, it uses, for each interval a $n^\delta \times n^\delta$-table whose entries (i, j) with $i < j$ are initialised by 1, if $\mathcal{A}[i]$ is inhabited and, otherwise, by 0. Next, for each triple i, j, k of positions in the interval entry (i, j) is set to 0 if $i < k < j$ and $\mathcal{A}[k]$ is inhabited. It is easy to see that afterwards entry (i, j) still carries a 1 if and only if i is the predecessor of j. And for all such pairs a link from $\mathcal{A}[j]$ to $\mathcal{A}[i]$ is added. For every interval, $(n^\delta)^3 = n^{3\delta}$ processors suffice for this computation, i.e. one processor for each triple i, j, k of positions in the interval. Since there are $\frac{n}{n^\delta} = n^{1-\delta}$ intervals, this yields an overall work of $n^{1-\delta} \cdot n^{3\delta} = n^{1+2\delta} = n^{1+\varepsilon}$. In the next round, intervals of length $n^{2\delta}$ are considered and each is viewed as a sequence of n^δ smaller intervals of length n^δ. The goal in the second round is to establish predecessor links for the minimum cells of each of the smaller intervals. This can be done similarly with the same asymptotic work as round 1. After $\lceil \frac{1}{\delta} \rceil$ rounds, this process has established predecessor links for all cells (besides for the minimum cells without a predecessor). ◀

In fully linked ordered arrays, tuples can be searched for efficiently.

▶ **Proposition 5.4.** *For every $\varepsilon > 0$, there is a $\mathcal{O}(1)$-time parallel algorithm that computes, for a given tuple t and an ordered array $\mathcal{A}$ with predecessor and successor links, the largest tuple t' in $\mathcal{A}$ with $t' \leq t$ with work $\mathcal{O}(|\mathcal{A}|^{\varepsilon})$ on an CREW-PRAM.*

Proof sketch. Let $n = |\mathcal{A}|$. In the first round, using n^{ε} processors, the algorithm tests for all cells with positions $k = in^{1-\varepsilon}$ whether $\mathcal{A}[k]$ is inhabited, or the predecessor of $\mathcal{A}[k]$ contains a tuple $t' \leq t$ and whether this does not hold for position $(i+1)n^{1-\varepsilon}$ or its successor. By a suitable process the search continues recursively in the thus identified sub-interval. After $\lceil \frac{1}{\varepsilon} \rceil$ rounds it terminates. Since, in each round, n^{ε} processors are used, the statement follows. ◀

We note that analogously it is possible to search for m tuples in parallel with work $\mathcal{O}(m\,|\mathcal{A}|^{\varepsilon})$.

As an alternative to ordered arrays, bounded-depth search trees could be used. They can be defined in the obvious way with degree about n^{ε}. The work for a search is asymptotically the same as for fully linked ordered arrays.

Algorithms for basic array operations. In the remainder of this section we consider algorithms for basic array operations. For the operations `Concatenate`, `Map`, and `Partition`, neither the algorithm nor the analysis depends on the setting, i.e. they are the same in the dictionary setting and the ordered setting. Furthermore, their implementation is straightforward and only requires EREW-PRAMs, instead of CRCW-PRAMs.

▶ **Lemma 5.5.** *There is a $\mathcal{O}(1)$-time parallel algorithm for* `Concatenate` *that, given arrays $\mathcal{A}$ and $\mathcal{B}$, requires $\mathcal{O}(|\mathcal{A}| + |\mathcal{B}|)$ work and space on an EREW-PRAM.*

Proof sketch. The tuples in $\mathcal{A}$ are loaded to processors 1 to $|\mathcal{A}|$ and the tuples in $\mathcal{B}$ to processors $|\mathcal{A}| + 1$ to $|\mathcal{A}| + |\mathcal{B}|$. The tuples are then stored in the output array $\mathcal{C}$. Each processor can also augment its tuple (between $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$) with mutual links. ◀

▶ **Lemma 5.6.** *There is a $\mathcal{O}(1)$-time parallel algorithm for* `Map` *that, given an array $\mathcal{A}$ and a function f that can be evaluated in $\mathcal{O}(1)$-time with work and space $\mathcal{O}(1)$ on an EREW-PRAM, requires $\mathcal{O}(|\mathcal{A}|)$ work and $\mathcal{O}(|\mathcal{A}|)$ space on an EREW-PRAM. If f is order-preserving and $\mathcal{A}$ is ordered, then the output array is ordered, too.*

Proof sketch. The algorithm loads the tuples in $\mathcal{A}$ to processors 1 to $|\mathcal{A}|$ and each processor computes the image $f(t)$ for its tuple t.

Since $f(t)$ has to be computed for $|\mathcal{A}|$ tuples, the bounds for work and space follow. ◀

▶ **Lemma 5.7.** *There is a $\mathcal{O}(1)$-time parallel algorithm for* `Partition` *that, given an array $\mathcal{A}$, an integer n, and a function g that maps proper tuples in $\mathcal{A}$ into $\{1,\ldots,n\}$ and can be evaluated in $\mathcal{O}(1)$-time with work and space $\mathcal{O}(1)$, requires $\mathcal{O}(n \cdot |\mathcal{A}|)$ work and $\mathcal{O}(n \cdot |\mathcal{A}|)$ space on an EREW-PRAM.*

Proof sketch. The algorithm first augments every proper tuple t with the number $g(t)$ using `Map`. This requires $\mathcal{O}(|\mathcal{A}|)$ work and $\mathcal{O}(|\mathcal{A}|)$ space, cf. Lemma 5.6.

Then the arrays $\mathcal{A}_1, \ldots, \mathcal{A}_n$ of size $|\mathcal{A}|$ are allocated (and initialised). This requires $\mathcal{O}(n \cdot |\mathcal{A}|)$ work and space.

For each $i \in \{1,\ldots, |\mathcal{A}|\}$ in parallel, the tuple t_i in cell $\mathcal{A}[i]$ is then – if it is a proper tuple – copied into cell $\mathcal{A}_j[i]$ where $j = g(t_i)$. This requires $\mathcal{O}(\mathcal{A})$ work. ◀

Let us point out that the upper bound for the work stated in Lemma 5.7 can be reduced to $\mathcal{O}(n + |\mathcal{A}|)$ by adapting the classical *lazy array initialisation technique* for (sequential) RAMs to PRAMs.[3] It turned out, however, that this is not necessary for our results, since n is either a constant or the work is dominated by other operations in our algorithms.

The algorithm for $\texttt{Compact}_\varepsilon$ does not depend on the setting either. It is implicitly proved in the proof of Proposition 5.1 in [14] for a fixed choice of b. The idea is to try several, but constantly many, compaction parameters.

▶ **Lemma 5.8** ([14], implicit in proof). *For every $\varepsilon > 0$, there is a $\mathcal{O}(1)$-time parallel algorithm for $\texttt{Compact}_\varepsilon$ that, given an array $\mathcal{A}$ and an upper bound b for the number of proper tuples in $\mathcal{A}$, requires $\mathcal{O}(|\mathcal{A}|)$ work and space on a CRCW-PRAM.*

Proof sketch. Let n denote the number of proper tuples in the given array $\mathcal{A}$.

We assume $n \geq 1$ in the following[4] and set $k = \lceil \frac{1}{\varepsilon} \rceil$. The algorithm invokes the algorithm guaranteed by Proposition 5.1 with $k_i = b^{\frac{i\varepsilon}{1+\varepsilon}}$ for every $0 \leq i \leq \lceil k(1+\varepsilon) \rceil$ in ascending order until it is successful (or the current k_i is larger than $|\mathcal{A}|$ in which case the procedure can just return $\mathcal{A}$). Each of the (constantly many) invocations requires $\mathcal{O}(|\mathcal{A}|)$ work and space.

If the algorithm is successful for $i = 0$, the requirements are trivially met.

Otherwise, let i be such that the compaction for $i+1$ was successful but the compaction for i was not (note that for $j = \lceil k(1+\varepsilon) \rceil$ the compaction will always succeed, since k_j is an upper bound for the number of proper tuples in $\mathcal{A}$). Then the resulting array has size

$$k_{i+1}^{1+\varepsilon} = \left(b^{\frac{(i+1)\varepsilon}{1+\varepsilon}} \right)^{1+\varepsilon} = b^{(i+1)\varepsilon} = b^{i\varepsilon} \cdot b^{\varepsilon}.$$

Moreover, since the compaction algorithm was not successful for i, we also have that $k_i = b^{\frac{i\varepsilon}{1+\varepsilon}} < n$ and, thus, $b^{i\varepsilon} < n^{1+\varepsilon}$.

All in all, the array $\mathcal{B}$ has size $b^{(i+1)\varepsilon} \leq n^{1+\varepsilon} \cdot b^{\varepsilon}$. ◀

For $\texttt{SearchRepresentatives}(\mathcal{A}, \mathcal{B})$ we present four algorithms, depending on the setting and, in the ordered setting, whether $\mathcal{A}$ or $\mathcal{B}$ is suitably ordered. This operation is crucial for deduplication, semi-join and join and the upper bounds impact the bounds for those operations as well.

▶ **Lemma 5.9.** *For every $\varepsilon > 0$, there are $\mathcal{O}(1)$-time parallel algorithms for* $\texttt{SearchRepre-sentatives}$ *that, given arrays $\mathcal{A}$ and $\mathcal{B}$, have the following bounds on a CRCW-PRAM.*

(a) *Work $\mathcal{O}((|\mathcal{A}| + |\mathcal{B}|) \cdot |\mathcal{B}|)$ and space $\mathcal{O}((|\mathcal{A}| + |\mathcal{B}|))$, without any assumptions;*

(b) *Work $\mathcal{O}(|\mathcal{A}| + |\mathcal{B}|)$ and space $\mathcal{O}((|\mathcal{A}| + |\mathcal{B}|) \cdot |D|)$ in the presence of a dictionary;*

(c) *Work $\mathcal{O}(|\mathcal{A}| \cdot |\mathcal{B}|^{\varepsilon})$ and space $\mathcal{O}((|\mathcal{A}| + |\mathcal{B}|))$, if $\mathcal{B}$ is ordered and fully linked;*

(d) *Work $\mathcal{O}(|\mathcal{B}| \cdot |\mathcal{A}|^{\varepsilon})$ and space $\mathcal{O}((|\mathcal{A}| + |\mathcal{B}|))$, if $\mathcal{A}$ is ordered and fully linked and $\mathcal{B}$ is concise.*

Proof sketch. For (a), the naive algorithm can be used. In a first phase, it uses one processor per pair (i, j) of indices for the cells of $\mathcal{B}$ to mark duplicates in $\mathcal{B}$: if $i < j$ and $\mathcal{B}[i].t = \mathcal{B}[j].t$, then $\mathcal{B}[j]$ is marked as duplicate. On the second phase, it uses one processor per pair (i, j) of indices for the cells of $\mathcal{A}$ and $\mathcal{B}$ and links $\mathcal{A}[i]$ to $\mathcal{B}[j]$ if $\mathcal{B}[i].t = \mathcal{B}[j].t$ and $\mathcal{B}[j]$ is not marked as duplicate.

[3] In a nutshell, this requires replacing a global counter by one counter per processor and maintaining back-references to initialised cells per processor (processors can still read counters and back-references of other processors).

[4] The algorithm yields $i = 0$, if $n = 0$ holds, as required.

For (b), first an array hash table for (the concatenation of) $\mathcal{A}$ and $\mathcal{B}$ is computed with $\mathcal{O}(|\mathcal{A}| + |\mathcal{B}|)$ work and $\mathcal{O}((|\mathcal{A}| + |\mathcal{B}|) \cdot |D|)$ space, thanks to Lemma 5.2 and Lemma 5.5. For a proper tuple t, let $h(t)$ denote the hash value in the range $\{1,\dots, |\mathcal{A}| + |\mathcal{B}|\}$ assigned to t. The algorithm then allocates an auxiliary array of size $|\mathcal{A}| + |\mathcal{B}|$ and, for each proper tuple s_i in $\mathcal{B}$, it writes, in parallel, i into cell $h(s_i)$ of the auxiliary array. Here s_i denotes the i-tuple from $\mathcal{B}$. Other processors might attempt to write an index to cell $h(s_i)$ but only one will succeed. This requires $\mathcal{O}(|\mathcal{B}|)$ work to write the indices and $\mathcal{O}(|\mathcal{A}| + |\mathcal{B}|)$ space for the auxiliary array.

For each proper tuple t in $\mathcal{A}$ it is then checked in parallel, if cell $h(t)$ contains an index i. If yes, then t is marked and augmented with a pointer to cell $\mathcal{B}[i]$, since $\mathcal{B}[i].t = t$. If not, then t has no partner tuple in $\mathcal{B}$, thus t is not augmented by a link.

Towards (c), the algorithm identifies, for each proper tuple t in $\mathcal{A}$, the smallest proper tuple s in $\mathcal{B}$ such that $t \leq s$. If $t = s$ holds, t's cell is marked and a link to the cell of s is added. For each tuple, this can be done with work $|\mathcal{B}|^{\varepsilon}$, thanks to Proposition 5.4 and these searches can be done in parallel by assigning $|\mathcal{B}|^{\varepsilon}$ processors per tuple of $\mathcal{A}$.

For (d), the algorithm searches, for each proper tuple s in $\mathcal{B}$, the smallest tuple t in $\mathcal{A}$ with $t \geq s$. If $t = s$ then the cell of t is marked and a link to the cell of s is added. If $\mathcal{A}$ is guaranteed to be concise, that's all. Otherwise, for each proper tuple t in $\mathcal{A}$ the smallest inhabited cell $\mathcal{A}[i]$ with $\mathcal{A}[i].t = t$ is searched. If it is marked then t is marked as well and a link to the cell in $\mathcal{B}$ to which $\mathcal{A}[i]$ links is added. ◀

▶ **Lemma 5.10.** *For every $\varepsilon > 0$, there are $\mathcal{O}(1)$-time parallel algorithms for* `Deduplicate` *that, given an array $\mathcal{A}$ have the following bounds on an arbitrary CRCW-PRAM.*

(a) *Work $\mathcal{O}(|\mathcal{A}|^2)$ and space $\mathcal{O}(|\mathcal{A}|)$, without any assumptions;*
(b) *Work $\mathcal{O}(|\mathcal{A}|)$ and space $\mathcal{O}(|\mathcal{A}| \cdot |D|)$ in the presence of a dictionary;*
(c) *Work $\mathcal{O}(|\mathcal{A}|^{1+\varepsilon})$ and space $\mathcal{O}(|\mathcal{A}|)$, if $\mathcal{A}$ is ordered.*

Proof sketch. In all three cases, `SearchRepresentatives`$(\mathcal{A}, \mathcal{A})$ is invoked and afterwards all inhabited cells with tuples that received a link to a different tuple are made uninhabited, leaving only the other tuples as representatives. For (c), it might be necessary to compute full links according to Proposition 5.3. The bounds then follow with Lemma 5.9. ◀

6 Algorithms for Database Operations

In this section, we present $\mathcal{O}(1)$-time parallel algorithms for the operators of the relational algebra and analyse their complexity with respect to work and space.

We formulate the results for relations rather than arrays. We always assume that a relation R is represented concisely by an array $\mathcal{A}_R$, but we make no assumptions about the compactness of the representation. All algorithms produce output arrays which represent the result relation concisely.

With the notable exception of the join operator, for most operators the algorithms are simple combinations of the algorithms of Section 5. The respective proofs are given in the full version of this paper [19]. The algorithms for the join operator are more involved and are presented at the end of the section.

We note that the relational algebra has an additional rename operator, which, of course, does not require a parallel algorithm.

▶ **Proposition 6.1.** *There is a $\mathcal{O}(1)$-time parallel algorithm that receives as input a relation R and an attribute X in R, and an element a in the domain or an attribute Y, and computes the selection $\sigma_{X=a}(R)$ (if a is given) or $\sigma_{X=Y}(R)$ (if Y is given). The algorithm requires $\mathcal{O}(|\mathcal{A}_R|)$ work and space on an EREW-PRAM. The output array is of size at most $|\mathcal{A}_R|$. If $\mathcal{A}_R$ is ordered, then the output is ordered, too.*

The algorithm is a simple application of the operation `Map`.

▶ **Proposition 6.2.** *For every $\varepsilon > 0$, there are $\mathcal{O}(1)$-time parallel algorithms for CRCW-PRAMs that compute upon input of two relations R and S the semijoin $R \ltimes S$ with the following bounds. Here, X denotes the joint attributes of R and S.*

(a) *Work $\mathcal{O}((|\mathcal{A}_R| + |\mathcal{A}_S|) \cdot |\mathcal{A}_S|)$ and space $\mathcal{O}((|\mathcal{A}_R| + |\mathcal{A}_S|))$, without any assumptions;*
(b) *Work $\mathcal{O}(|\mathcal{A}_R| + |\mathcal{A}_S|)$ and space $\mathcal{O}((|\mathcal{A}_R| + |\mathcal{A}_S|) \cdot |D|)$ in the presence of a dictionary;*
(c) *Work $\mathcal{O}(|\mathcal{A}_R| \cdot |\mathcal{A}_S|^{\varepsilon})$ and space $\mathcal{O}((|\mathcal{A}_R| + |\mathcal{A}_S|))$, if $\mathcal{A}_S$ is X-compatibly ordered and fully linked;*
(d) *Work $\mathcal{O}(|\mathcal{A}_S| \cdot |\mathcal{A}_R|^{\varepsilon})$ and space $\mathcal{O}((|\mathcal{A}_R| + |\mathcal{A}_S|))$, if $\mathcal{A}_R$ is X-compatibly ordered and fully linked and $\mathcal{A}_S$ is concise.*

The output array is of size $|\mathcal{A}_R|$. If $\mathcal{A}_R$ is ordered, then the output is ordered, too. Moreover, each $t \in R \ltimes S$ in the output of $R \ltimes S$ gets augmented by a link to a corresponding tuple in S.

▶ **Proposition 6.3.** *For every $\varepsilon > 0$, there are CRCW-PRAM $\mathcal{O}(1)$-time parallel algorithms that compute upon input of two relations R and S the difference $R \setminus S$ with the following bounds.*

(a) *Work $\mathcal{O}((|\mathcal{A}_R| + |\mathcal{A}_S|) \cdot |\mathcal{A}_S|)$ and space $\mathcal{O}((|\mathcal{A}_R| + |\mathcal{A}_S|))$, without any assumptions;*
(b) *Work $\mathcal{O}(|\mathcal{A}_R| + |\mathcal{A}_S|)$ and space $\mathcal{O}((|\mathcal{A}_R| + |\mathcal{A}_S|) \cdot |D|)$ in the presence of a dictionary;*
(c) *Work $\mathcal{O}(|\mathcal{A}_R| \cdot |\mathcal{A}_S|^{\varepsilon})$ and space $\mathcal{O}((|\mathcal{A}_R| + |\mathcal{A}_S|))$, if $\mathcal{A}_S$ is ordered and fully linked;*
(d) *Work $\mathcal{O}(|\mathcal{A}_S| \cdot |\mathcal{A}_R|^{\varepsilon})$ and space $\mathcal{O}((|\mathcal{A}_R| + |\mathcal{A}_S|))$, if $\mathcal{A}_R$ is ordered and fully linked and $\mathcal{A}_S$ is concise.*

The output array is of size $|\mathcal{A}_R|$. If $\mathcal{A}_R$ is ordered, then the output is ordered, too.

The algorithms for Proposition 6.2 and Proposition 6.3 combine the appropriate algorithm for `SearchRepresentatives` with suitable applications of `Map`.

▶ **Proposition 6.4.** *For every $\varepsilon > 0$, there are CRCW-PRAM $\mathcal{O}(1)$-time parallel algorithms that receive as input a relation R and a list X of attributes from R, and evaluate the projection $\pi_X(R)$ with the following bounds.*

(a) *Work $\mathcal{O}((|\mathcal{A}_R|^2)$ and space $\mathcal{O}(|\mathcal{A}_R|)$, without any assumptions;*
(b) *Work $\mathcal{O}(|\mathcal{A}_R|)$ and space $\mathcal{O}(|\mathcal{A}_R| \cdot |D|)$ in the presence of a dictionary;*
(c) *Work $\mathcal{O}(|\mathcal{A}_R|^{1+\varepsilon})$ and space $\mathcal{O}(|\mathcal{A}_R|)$, if $\mathcal{A}_R$ is X-compatibly ordered.*

The output array is of size $|\mathcal{A}_R|$. If $\mathcal{A}_R$ is ordered then the output is ordered, too.

The algorithms combine `Deduplicate` with `Map` in a straightforward manner. We note that we do not require in (c) that $\mathcal{A}_R$ is fully linked, since the work bound allows to compute links.

▶ **Proposition 6.5.** *For every $\varepsilon > 0$, there are CRCW-PRAM $\mathcal{O}(1)$-time parallel algorithms that compute upon input of two relations R and S the union $R \cup S$ with the following bounds.*

(a) *Work $\mathcal{O}((|\mathcal{A}_R| + |\mathcal{A}_S|) \cdot |\mathcal{A}_S|)$ and space $\mathcal{O}((|\mathcal{A}_R| + |\mathcal{A}_S|))$, without any assumptions;*
(b) *Work $\mathcal{O}(|\mathcal{A}_R| + |\mathcal{A}_S|)$ and space $\mathcal{O}((|\mathcal{A}_R| + |\mathcal{A}_S|) \cdot |D|)$ in the presence of a dictionary;*
(c) *Work $\mathcal{O}(|\mathcal{A}_R| \cdot |\mathcal{A}_S|^{\varepsilon} + |\mathcal{A}_S|)$ and space $\mathcal{O}((|\mathcal{A}_R| + |\mathcal{A}_S|))$, if $\mathcal{A}_S$ is ordered and fully linked.*

The output array is of size $|\mathcal{A}_R| + |\mathcal{A}_S|$.

The algorithms basically concatenate $R \setminus S$ and S. We note that thanks to the symmetry of union, the algorithm of (c) can also be applied if $\mathcal{A}_R$ is ordered.

▶ **Proposition 6.6.** *For every $\varepsilon > 0$, there are CRCW-PRAM $\mathcal{O}(1)$-time parallel algorithms that compute upon input of two relations R and S the join $R \bowtie S$ with the following bounds. Here, X denotes the joint attributes of R and S.*

(a) *Work* $\mathcal{O}((|\mathcal{A}_S|^2 + |\pi_X(S)|^{1+\varepsilon}|\mathcal{A}_S|^{1+\varepsilon}) + (|\mathcal{A}_R| + |\mathcal{A}_S|)|\mathcal{A}_S| + |R \bowtie S|\,|\mathcal{A}_R|^{2\varepsilon}|\mathcal{A}_S|^{2\varepsilon})$ *and space* $\mathcal{O}(|\pi_X(S)|^{1+\varepsilon}|\mathcal{A}_S|^{1+\varepsilon} + |\mathcal{A}_R| + |\mathcal{A}_S| + |R \bowtie S|\,|\mathcal{A}_R|^{2\varepsilon}|\mathcal{A}_S|^{2\varepsilon})$ *without any assumptions;*

(b) *Work* $\mathcal{O}(|\pi_X(S)|^{1+\varepsilon}|\mathcal{A}_S|^{1+\varepsilon} + (|\mathcal{A}_R| + |\mathcal{A}_S|) + |R \bowtie S|\,|\mathcal{A}_R|^{2\varepsilon}|\mathcal{A}_S|^{2\varepsilon})$ *and space* $\mathcal{O}(|\pi_X(S)|^{1+\varepsilon}|\mathcal{A}_S|^{1+\varepsilon} + (|\mathcal{A}_R| + |\mathcal{A}_S|)|D| + |R \bowtie S|\,|\mathcal{A}_R|^{2\varepsilon}|\mathcal{A}_S|^{2\varepsilon})$ *in the presence of a dictionary;*

(c) *Work* $\mathcal{O}(|\mathcal{A}_S|^{1+\varepsilon} + |\mathcal{A}_R| \cdot |\mathcal{A}_S|^{\varepsilon} + |R \bowtie S|\,|\mathcal{A}_R|^{2\varepsilon}|\mathcal{A}_S|^{2\varepsilon})$ *and space* $\mathcal{O}(|\mathcal{A}_R| + |\mathcal{A}_S| + |R \bowtie S|\,|\mathcal{A}_R|^{2\varepsilon}|\mathcal{A}_S|^{2\varepsilon})$, *if* $\mathcal{A}_S$ *is* X*-compatibly ordered and fully linked.*

The output array is of size $|R \bowtie S|\,|\mathcal{A}_R|^{2\varepsilon}|\mathcal{A}_S|^{2\varepsilon}$*.*

Proof idea. The algorithms proceed in three phases, the grouping phase, the pairing phase and the joining phase. For (a) and (b), the tuples of S are grouped with respect to their X-attributes in the grouping phase. Each group is compacted into an array of some size $|\mathcal{A}_S|^{\ell\varepsilon}$. Likewise the projection $\pi_X(S)$, containing the "index tuples", is compacted. In the pairing phase, a semijoin reduction is performed and the remaining R-tuples are partitioned with respect to the size of their corresponding "X-group" from S. Finally, during the joining phase, output tuples are produced, by combining tuples from R with the tuples from their "X-group" from S. The work bounds for the three phases can be seen as the three main summands in the statement of the proposition.

If S is represented by an array that is X-compatibly ordered and fully linked, the grouping phase can be performed more efficiently. In that case, $\mathcal{A}_S$ itself can be viewed as the concatenation of all "X-groups". Thus, this steps is for free and, furthermore, the compaction of the "X-groups" can be done in-place and therefore only requires work $|\mathcal{A}_S|$ in total. The pairing phase and the joining phase are basically as for (a) and (b), but the work bounds for the pairing phase differ, due to the more efficient semijoin algorithm in the ordered setting. ◀

7 Query Evaluation

After studying algorithms for basic operations and operators of the relational algebra, we are now prepared to investigate the complexity of $\mathcal{O}(1)$-time parallel algorithms for query evaluation.

Although every query of the relational algebra can be evaluated by a $\mathcal{O}(1)$-time parallel algorithms with polynomial work, the polynomials can be arbitrarily bad. In fact, that a graph has a k-clique can be expressed by a conjunctive query with k variables and it follows from Rossman's $\omega(n^{k/4})$ lower bound for the size of bounded-depth circuit families for k-Clique [24] that any $\mathcal{O}(1)$-time parallel algorithm that evaluates this query needs work $\omega(n^{k/4})$.

We therefore concentrate in this section on restricted query evaluation settings. We study two restrictions of query languages which allow efficient sequential algorithms, the semijoin algebra and free-connex and/or acylic conjunctive queries. Furthermore, we present a $\mathcal{O}(1)$-time parallel version of worst-case optimal join algorithms.

In the following, IN always denotes the maximum number of tuples in any relation of the underlying database that is addressed by the given query. Furthermore, we always assume that the database relations are represented concisely by *compact* arrays without any uninhabited cells.

7.1 Semi-Join Algebra

The semijoin algebra is the fragment of the relational algebra that uses only selection, projection, rename, union, set difference and, not least, semijoin. It is well-known that semijoin queries produce only query results of size $\mathcal{O}(|D|)$ and can be evaluated in time $\mathcal{O}(|D|)$ [21, Theorem 7]. From the results of Section 6 we can easily conclude the following.

▶ **Proposition 7.1.** *For each query q of the semijoin algebra and for every $\varepsilon > 0$ there are CRCW-PRAM $\mathcal{O}(1)$-time parallel algorithms that, given a database D, evaluate $q(D)$ with the following bounds.*

(a) *Work $\mathcal{O}(\mathit{IN}^{2+\varepsilon})$ and space $\mathcal{O}(\mathit{IN}^{2+\varepsilon})$, without any assumptions;*

(b) *Work $\mathcal{O}(\mathit{IN})$ and space $\mathcal{O}(\mathit{IN} \cdot |D|)$ in the presence of a dictionary.*

Proof sketch. Towards (a), the operators of the query are evaluated with the naive algorithms from Section 6 (stated as (a)). After each evaluation the result array is compacted by $\texttt{Compact}_{\varepsilon/2}$. Statement (b) follows by using the (b)-algorithms from Section 6. ◀

Altogether, semijoin queries can be evaluated work-optimally by a $\mathcal{O}(1)$-time parallel algorithm. We plan to address ordered setting in a journal version of this paper. We expect that the results of [13] enable almost work-optimal $\mathcal{O}(1)$-time parallel algorithms with a $\mathcal{O}(\mathsf{IN}^{1+\varepsilon})$ work bound, if the relations are represented by suitably ordered arrays. We discuss this further in our conclusion.

7.2 Evaluation of Conjunctive Queries

In this section we give algorithms to evaluate subclasses of conjunctive queries in parallel. More precisely, we consider acyclic join queries, acyclic conjunctive queries, free-connex acyclic conjuntive queries and arbitrary free-connex conjunctive queries.

Conjunctive queries are conjunctions of relation atoms. We write a *conjunctive query* (*CQ* for short) q as a rule of the form $q : \mathsf{A} \leftarrow \mathsf{A}_1, \ldots, \mathsf{A}_m$, where $\mathsf{A}, \mathsf{A}_1, \ldots, \mathsf{A}_m$ are atoms and $m \geq 1$. A conjunctive query q is *acyclic*, if it has a join tree T_q, i.e. an undirected tree $(V(T), E(T))$ where $V(T)$ consists of the atoms in q and for each variable v in T_q the set $\{\alpha \in V(T) | \alpha \text{ contains } v\}$ induces a connected subtree of T_q. It is *free-connex acyclic* if q is acyclic and the Boolean query whose body consists of the body atoms *and* the head atom of q is acyclic as well [5, 9]. A *join query* is a conjunctive query with no quantified variable, i.e. every variable in a join query is free. For more background on (acyclic) conjunctive queries we refer to [1, 3].

Our algorithms rely on the well-known Yannakakis algorithm [29]. Yannakakis' algorithm receives as input an acyclic conjunctive query q, the join tree T_q and a database D. With each node v in T_q a relation S_v is associated. Initially, $S_v = R_v(D)$, where R_v is the relation that is labelled in v. The algorithm is divided into three steps.

(1) **bottom-up semijoin reduction:** All nodes are visited in bottom-up traversal order of T. When a node v is visited, S_v is updated to $S_v \ltimes S_c$ for every child c of v in T.

(2) **top-down semijoin reduction:** All nodes are visited in top-down traversal order of T. When a node v is visited, the relation S_c is updated to $S_c \ltimes S_v$ for every child c of v in T.

(3) All nodes are visited in bottom-up traversal order in T. When a node v is visited, the algorithm updates, for every child c of v, the relation S_v to $\pi_{\mathsf{free}(q) \cup \mathtt{attr}(S_v)}(S_v \bowtie S_c)$, where $\mathsf{free}(q)$ denotes the attributes that are associated with the free variables of q.

Proposition 6.2 immediately yields the following lemma.

▶ **Lemma 7.2.** *There are CRCW-PRAM $\mathcal{O}(1)$-time parallel algorithms for phase (1) and (2) of the Yannakakis algorithm with the following bounds.*

(a) *Work $\mathcal{O}(\mathsf{IN}^2)$ and space $\mathcal{O}(\mathsf{IN})$, without any assumptions;*

(b) *Work $\mathcal{O}(\mathsf{IN})$ and space $\mathcal{O}(\mathsf{IN} \cdot |D|)$ in the presence of a dictionary.*

By combining Yannakakis' algorithm with the algorithms from Section 6 we obtain the following results.

▶ **Proposition 7.3.** *For every $\varepsilon > 0$ and every acyclic join query q, there are CRCW-PRAM $\mathcal{O}(1)$-time parallel algorithms that compute $q(D)$, given a database D, with the following bounds.*

(a) *Work $\mathcal{O}(\mathsf{IN}^2 + \mathsf{OUT}^{2+\varepsilon}\mathsf{IN}^{\varepsilon})$ and space $\mathcal{O}(\mathsf{IN} + \mathsf{OUT}^{2+\varepsilon}\mathsf{IN}^{\varepsilon})$, without any assumptions;*

(b) *Work $\mathcal{O}(\mathsf{IN}^{1+\varepsilon} \cdot \mathsf{OUT}^{1+\varepsilon})$ and space $\mathcal{O}((\mathsf{IN} \cdot \mathsf{OUT})^{1+\varepsilon}\,|D|)$, in the presence of a dictionary.*

To perform phase (3) of the Yannakakis algorithm the parallel algorithms first shrink every array $\mathcal{A}_{R_v}$ to the size $|S_v|^{1+\varepsilon'}\,|R_v(D)|^{\varepsilon'}$ using $\texttt{Compact}_{\varepsilon'}(S_v)$, for some very small ε', depending (only) on the size of the join tree. Likewise, by calling the join algorithm with a suitable parameter, it strongly compacts each intermediate join result. That the stated bounds are met can be established by a straightforward, but tedious calculation, given in the full version of this paper [19].

▶ **Proposition 7.4.** *For every $\varepsilon > 0$, and every acyclic conjunctive query q, there are CRCW-PRAM $\mathcal{O}(1)$-time parallel algorithms that compute $q(D)$, given a database D, with the following bounds.*

(a) *Work $\mathcal{O}(\mathsf{IN}^2 + \mathsf{OUT}^{2+\varepsilon}\mathsf{IN}^{2+\varepsilon})$ and space $\mathcal{O}(\mathsf{IN} + \mathsf{IN}^{2+\varepsilon} \cdot \mathsf{OUT}^{1+\varepsilon})$, without any assumptions;*

(b) *Work $\mathcal{O}(\mathsf{OUT}^{1+\varepsilon}\mathsf{IN}^{2+2\varepsilon})$ and space $\mathcal{O}(\mathsf{OUT}^{1+\varepsilon}\mathsf{IN}^{2+\varepsilon}\,|D|)$, in the presence of a dictionary.*

The algorithms are obtained from the algorithms for Proposition 7.3 by a suitable adaptation of phase (3). A proof sketch for Proposition 7.4 is given in the full version [19].

It turns out that the bounds for acyclic join queries carry over to free-connex acyclic conjunctive queries. We use the reduction from free-connex acyclic queries to join queries given in [8]. We adapt it for $\mathcal{O}(1)$-time parallel algorithms.

▶ **Lemma 7.5.** *For every free-connex acyclic query q and every database D there exists an acyclic join query $\tilde{q}$ and a database $\widetilde{D}$ such that $q(D) = \widetilde{q}(\widetilde{D})$. Here, $\tilde{q}$ only depends on q.*

Furthermore, there are CRCW-PRAM $\mathcal{O}(1)$-time parallel algorithms that compute upon input of a free-connex acyclic query q and a database D the corresponding join query $\widetilde{q}$ and database $\widetilde{D}$ with the following bounds.

(a) *Work $\mathcal{O}(\mathsf{IN}^2)$ and space $\mathcal{O}(\mathsf{IN})$, without any assumptions;*

(b) *Work $\mathcal{O}(\mathsf{IN})$ and space $\mathcal{O}(\mathsf{IN} \cdot |D|)$ in the presence of a dictionary;*

A proof sketch for Lemma 7.5 is given in the full version of this paper [19]. By combining Lemma 7.5 and Proposition 7.3 we obtain the following result.

▶ **Corollary 7.6.** *There are CRCW-PRAM $\mathcal{O}(1)$-time parallel algorithms that receives as input a free-connex acyclic conjunctive query q and a database D and computes the result $q(D)$, with the following bounds.*

(a) *Work $\mathcal{O}(\mathsf{IN}^2 + \mathsf{OUT}^{2+\varepsilon}\mathsf{IN}^{\varepsilon})$ and space $\mathcal{O}(\mathsf{IN} + \mathsf{OUT}^{2+\varepsilon}\mathsf{IN}^{\varepsilon})$, without any assumptions;*

(b) *Work $\mathcal{O}(\mathsf{IN}^{1+\varepsilon} \cdot \mathsf{OUT}^{1+\varepsilon})$ and space $\mathcal{O}((\mathsf{IN} \cdot \mathsf{OUT})^{1+\varepsilon}\,|D|)$, in the presence of a dictionary.*

In [5, Definition 36] and [8, Definition 3.2], a definition of free-connex, not necessarily acyclic, conjunctive queries is given. Corollary 7.6 can be extended to that class of queries along the lines of [8, Lemma 4.4].

We plan to give a more detailed account in a journal version of this paper.

7.3 Weakly Worst-Case Optimal Work for Natural Joins

This section is concerned with the evaluation of *natural join queries* $q = R_1 \bowtie \ldots \bowtie R_m$ over some schema $\Sigma = \{R_1, \ldots, R_m\}$ with attributes $\mathtt{attr}(q) = \bigcup_{i=1}^m \mathtt{attr}(R_i)$. It was shown in [4] that $|q(D)| \leq \prod_{i=1}^m |R_i|^{x_i}$ holds for every database D and that this bound is tight for infinitely many databases D (this is also known as the AGM bound). Here $x_1, \ldots, x_m$ is a fractional edge cover of q defined as a solution of the following linear program.

$$\text{minimize} \sum_{i=1}^m x_i \text{ subject to} \sum_{i: A \in \mathtt{attr}(R_i)} x_i \geq 1 \text{ for all } A \in \mathtt{attr}(q)$$
$$\text{and } x_i \geq 0 \text{ for all } 1 \leq i \leq m$$

We say that a natural join query q has *weakly worst-case optimal* $\mathcal{O}(1)$-time parallel algorithms, if, for every $\varepsilon > 0$, there is a $\mathcal{O}(1)$-time parallel algorithm that evaluates q with work $(\prod_{i=1}^m |R_i|^{x_i} + \mathsf{IN})^{1+\varepsilon}$. For comparison, in the sequential setting, algorithms are considered worst-case optimal if they have a time bound $O(\prod_{i=1}^m |R_i|^{x_i} + \mathsf{IN})$ [23]. In this subsection, we show that natural join queries indeed have weakly worst-case optimal $\mathcal{O}(1)$-time parallel algorithms.

▶ **Theorem 7.7.** *For every $\varepsilon > 0$ and natural join query $q = R_1 \bowtie \ldots \bowtie R_m$ with attributes $X = (A_1, \ldots, A_k)$, there is a $\mathcal{O}(1)$-time parallel algorithm that, given arrays $\mathcal{A}_{R_1}, \ldots, \mathcal{A}_{R_m}$ ordered w.r.t. X, computes $q(D)$ and requires $\mathcal{O}\left(\left((\prod_{i=1}^m |R_i|^{x_i}) + \mathsf{IN}\right) \cdot \mathsf{IN}^{\varepsilon}\right)$ work and space on a CRCW-PRAM where $(x_1, \ldots, x_m)$ is a fractional edge cover of q.*

Proof idea. A $\mathcal{O}(1)$-time parallel algorithm can proceed, from a high-level perspective, similarly to the sequential attribute elimination join algorithm, see e.g. [3, Algorithm 10].

In a nutshell, the algorithm computes iteratively, for increasing j from 1 to k relations L_j defined as follows: $L_1 = \bigcap_{1 \leq i \leq m, A_1 \in \mathtt{attr}(R_i)} \pi_{A_1}(R_i)$ and, for $j > 1$, L_j is the union of all relations $V_t = \{t\} \times \bigcap_{1 \leq i \leq m, A_j \in \mathtt{attr}(R_i)} \pi_{A_j}(R_i \ltimes \{t\})$ for each $t \in L_{j-1}$. L_k is then the query result $q(D)$. Note that each L_j contains tuples over attributes $X_j = (A_1, \ldots, A_j)$.

To achieve the desired running time in the sequential setting, it is essential that each relation V_t for $t \in L_{j-1}$ is computed in time $\tilde{\mathcal{O}}(\min_{1 \leq i \leq m} |R_i \ltimes \{t\}|)$, where $\tilde{\mathcal{O}}$ hides a logarithmic factor; for instance with the Leapfrog algorithm, see e.g. [27], [3, Proposition 27.10].

In the parallel setting each relation V_t is computed with work $\mathcal{O}(\min_{1 \leq i \leq m} |R_i \ltimes \{t\}| \cdot \mathsf{IN}^{\frac{1}{2}\varepsilon})$ – for all tuples $t \in L_{i-1}$ in parallel. Note that the work bound is *not* uniform, i.e. the work bound for a tuple t depends on how many "matching" tuples there are in each of the input relations. This makes assigning processors challenging.

Utilizing that the input relations are ordered w.r.t. X_j, our algorithm groups the tuples in the relations $\pi_{X_j}(R_i)$ w.r.t. X_{j-1} and identifies, for each $t \in L_{j-1}$, the corresponding group in $\pi_{X_j}(R_i)$. These groups are compacted using $\mathtt{Compact}_\delta$ for $\delta = \frac{\varepsilon}{4}$ which allows to approximate the size of $R_i \ltimes \{t\}$ up to a factor of $\mathsf{IN}^{\frac{1}{2}\varepsilon}$ for each i, and, thus, $\min_{1 \leq i \leq m} |R_i \ltimes \{t\}|$ for each tuple t.

The tuples in L_{j-1} are then partitioned w.r.t. (the approximation of) $\min_{1 \leq i \leq m} |R_i \ltimes \{t\}|$ into sets $S_{j,\ell}$. Each tuple in a set $S_{j,\ell}$ can then be assigned the same number of processors, determined by the size of the array for the smallest group, similarly as in the Leapfrog algorithm. This is feasible because the number of sets $S_{j,\ell}$ in the partition is bounded by a constant due to the guarantees of $\mathtt{Compact}_\varepsilon$.

The full proof is given in the full version of this paper [19]. ◀

We plan to address the evaluation of natural join queries in the dictionary setting in a journal version of this paper. We expect that almost the same work bound holds, with an additional summand $\mathcal{O}(\max_{1 \leq i \leq m} |R_i|^2)$, accounting for the grouping of each $\pi_{X_j}(R_i)$ with respect to $\pi_{X_{j-1}}(R_i)$.

8 Conclusion

This paper is meant as a first study on work-efficient $\mathcal{O}(1)$-time parallel algorithms for query evaluation and many questions remain open. The results are very encouraging as they show that quite work-efficient $\mathcal{O}(1)$-time parallel algorithms for query evaluation are possible. In fact, the results give a hint at what could be a good notion of *work-efficiency* in the context of constant-time parallel query evaluation. Our impression is that work-optimality is very hard to achieve in constant time and that query evaluation should be considered as work-efficient for a query language, if there are constant-time parallel algorithms with $\mathcal{O}(T^{1+\varepsilon})$ work, for every $\varepsilon > 0$, where T is the best sequential time of an evaluation algorithm. Of course, it would be nice if this impression could be substantiated by lower bound results, but that seems to be quite challenging.

We have not given results for all combinations of query languages and settings, e.g., Subsection 7.1 and Subsection 7.2 do not yet cover the ordered setting and Subsection 7.3 not the dictionary setting.

As mentioned in Section 3, when finding the results of this paper we were unaware of the fact that [13] provides algorithms for *ordered* compaction with constant time and work $\mathcal{O}(n^{1+\varepsilon})$. Naturally, these algorithms can be useful for the ordered setting and we expect them to yield a $\mathcal{O}(n^{1+\varepsilon})$ work bound for the semi-join algebra (Subsection 7.1). We do not expect them to improve the bounds for natural joins (Subsection 7.3) or for general acyclic queries (Subsection 7.2). We plan to fully explore the consequences in a journal version of this paper, but we decided against incorporating them into the final version of this paper, due to the lack of peer-review. In that journal version we will also address some of the reviewer's suggestions that could not be incorporated yet.

References

1 Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995. URL: `http://webdam.inria.fr/Alice/`.

2 Miklós Ajtai. Σ_1^1 formulae on finite structures. *Ann. of Pure and Applied Logic*, 24:1–48, 1983. `doi:10.1016/0168-0072(83)90038-6`.

3 Marcelo Arenas, Pablo Barceló, Leonid Libkin, Wim Martens, and Andreas Pieris. *Database Theory*. Open source at `https://github.com/pdm-book/community`, 2021. Preliminary Version, August 19, 2022.

4 Albert Atserias, Martin Grohe, and Dániel Marx. Size bounds and query plans for relational joins. *SIAM J. Comput.*, 42(4):1737–1767, 2013. `doi:10.1137/110859440`.

5 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`.

6 David A. Mix Barrington, Neil Immerman, and Howard Straubing. On uniformity within NC^1. *J. Comput. Syst. Sci.*, 41(3):274–306, 1990. `doi:10.1016/0022-0000(90)90022-D`.

7 Paul Beame, Paraschos Koutris, and Dan Suciu. Communication steps for parallel query processing. *J. ACM*, 64(6):40:1–40:58, 2017. `doi:10.1145/3125644`.

8 Christoph Berkholz, Fabian Gerhardt, and Nicole Schweikardt. Constant delay enumeration for conjunctive queries: a tutorial. *ACM SIGLOG News*, 7(1):4–33, 2020. `doi:10.1145/3385634.3385636`.

9 Johann Brault-Baron. *De la pertinence de l'énumération : complexité en logiques propositionnelle et du premier ordre*. Theses, Université de Caen, April 2013. URL: `https://hal.archives-ouvertes.fr/tel-01081392`.

10 Zhiyuan Chen, Johannes Gehrke, and Flip Korn. Query optimization in compressed database systems. In Sharad Mehrotra and Timos K. Sellis, editors, *Proceedings of the 2001 ACM SIGMOD international conference on Management of data, Santa Barbara, CA, USA, May 21-24, 2001*, pages 271–282. ACM, 2001. doi:10.1145/375663.375692.

11 E. F. Codd. Relational completeness of data base sublanguages. In R. Rustin, editor, *Database Systems*, pages 33–64. Prentice-Hall, 1972.

12 Merrick L. Furst, James B. Saxe, and Michael Sipser. Parity, circuits, and the polynomial-time hierarchy. *Mathematical Systems Theory*, 17(1):13–27, 1984. doi:10.1007/BF01744431.

13 Tal Goldberg and Uri Zwick. Optimal deterministic approximate parallel prefix sums and their applications. In *Third Israel Symposium on Theory of Computing and Systems, ISTCS 1995, Tel Aviv, Israel, January 4-6, 1995, Proceedings*, pages 220–228. IEEE Computer Society, 1995. doi:10.1109/ISTCS.1995.377028.

14 Torben Hagerup. On a compaction theorem of Ragde. *Inf. Process. Lett.*, 43(6):335–340, 1992. doi:10.1016/0020-0190(92)90121-B.

15 Xiao Hu and Ke Yi. Massively parallel join algorithms. *SIGMOD Rec.*, 49(3):6–17, 2020. doi:10.1145/3444831.3444833.

16 Neil Immerman. Expressibility and parallel complexity. *SIAM J. Comput.*, 18(3):625–638, 1989. doi:10.1137/0218043.

17 Neil Immerman. *Descriptive Complexity*. Graduate texts in computer science. Springer, 1999. doi:10.1007/978-1-4612-0539-5.

18 Joseph F. JáJá. *An Introduction to Parallel Algorithms*. Addison-Wesley, 1992.

19 Jens Keppeler, Thomas Schwentick, and Christopher Spinrath. Work-efficient query evaluation with PRAMs, 2023. doi:10.48550/ARXIV.2301.08178.

20 Paraschos Koutris, Semih Salihoglu, and Dan Suciu. Algorithmic aspects of parallel data processing. *Found. Trends Databases*, 8(4):239–370, 2018. doi:10.1561/1900000055.

21 Dirk Leinders, Maarten Marx, Jerzy Tyszkiewicz, and Jan Van den Bussche. The semijoin algebra and the guarded fragment. *Journal of Logic, Language and Information*, 14(3):331–343, 2005. doi:10.1007/s10849-005-5789-8.

22 Philip D. MacKenzie. Load balancing requires omega($\log^* n$) expected time. In Greg N. Frederickson, editor, *Proceedings of the Third Annual ACM/SIGACT-SIAM Symposium on Discrete Algorithms, 27-29 January 1992, Orlando, Florida, USA*, pages 94–99. ACM/SIAM, 1992. URL: http://dl.acm.org/citation.cfm?id=139404.139425.

23 Hung Q. Ngo, Ely Porat, Christopher Ré, and Atri Rudra. Worst-case optimal join algorithms. *J. ACM*, 65(3):16:1–16:40, 2018. doi:10.1145/3180143.

24 Benjamin Rossman. On the constant-depth complexity of k-clique. In *Proceedings of the 40th Annual ACM Symposium on Theory of Computing, Victoria, British Columbia, Canada, May 17-20, 2008*, pages 721–730, 2008. doi:10.1145/1374376.1374480.

25 Jonas Schmidt, Thomas Schwentick, Till Tantau, Nils Vortmeier, and Thomas Zeume. Work-sensitive dynamic complexity of formal languages. In Stefan Kiefer and Christine Tasson, editors, *Foundations of Software Science and Computation Structures - 24th International Conference, FOSSACS 2021*, volume 12650 of *Lecture Notes in Computer Science*, pages 490–509. Springer, 2021. doi:10.1007/978-3-030-71995-1_25.

26 Peter van Emde Boas. Machine models and simulation. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity*, pages 1–66. Elsevier and MIT Press, 1990.

27 Todd L. Veldhuizen. Triejoin: A simple, worst-case optimal join algorithm. In Nicole Schweikardt, Vassilis Christophides, and Vincent Leroy, editors, *Proc. 17th International Conference on Database Theory (ICDT), Athens, Greece, March 24-28, 2014*, pages 96–106. OpenProceedings.org, 2014. doi:10.5441/002/icdt.2014.13.

28 Yilei Wang and Ke Yi. Query evaluation by circuits. In Leonid Libkin and Pablo Barceló, editors, *PODS '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, pages 67–78. ACM, 2022. doi:10.1145/3517804.3524142.

29 Mihalis Yannakakis. Algorithms for acyclic database schemes. In *Very Large Data Bases, 7th International Conference, September 9-11, 1981, Cannes, France, Proceedings*, pages 82–94. IEEE Computer Society, 1981.

Conjunctive Queries with Free Access Patterns Under Updates

Ahmet Kara
Universität Zürich, Switzerland

Milos Nikolic
University of Edinburgh, UK

Dan Olteanu
Universität Zürich, Switzerland

Haozhe Zhang
Universität Zürich, Switzerland

Abstract

We study the problem of answering conjunctive queries with free access patterns under updates. A free access pattern is a partition of the free variables of the query into input and output. The query returns tuples over the output variables given a tuple of values over the input variables.

We introduce a fully dynamic evaluation approach for such queries. We also give a syntactic characterisation of those queries that admit constant time per single-tuple update and whose output tuples can be enumerated with constant delay given an input tuple. Finally, we chart the complexity trade-off between the preprocessing time, update time and enumeration delay for such queries. For a class of queries, our approach achieves optimal, albeit non-constant, update time and delay. Their optimality is predicated on the Online Matrix-Vector Multiplication conjecture. Our results recover prior work on the dynamic evaluation of conjunctive queries without access patterns.

2012 ACM Subject Classification Theory of computation → Database query processing and optimization (theory); Information systems → Database views; Information systems → Data streams

Keywords and phrases fully dynamic algorithm, enumeration delay, complexity trade-off, dichotomy

Digital Object Identifier 10.4230/LIPIcs.ICDT.2023.17

Funding This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 682588.

1 Introduction

We consider the problem of dynamic evaluation for conjunctive queries with access restrictions. Restricted access to data is commonplace [28, 29, 27]: For instance, the flight information behind a user-interface query can only be accessed by providing values for specific input fields such as the departure and destination airports in a flight booking database.

We formalise such queries as **c**onjunctive **q**ueries with free **a**ccess **p**atterns (CQAP for short): The free variables of a CQAP are partitioned into *input* and *output*. The query yields tuples of values over the output variables *given* a tuple of values over the input variables. In database systems, CQAPs formalise the notion of parameterized queries (or prepared statements) [1]. In probabilistic graphical models, they correspond to conditional queries [25]: Such inference queries ask for (the probability of) each possible value of a tuple of random variables (corresponding to CQAP output variables) given specific values for a tuple of random variables (corresponding to CQAP input variables). Prior work on queries with access patterns considered a more general setting than CQAP: There, each relation in the query body may have input and output variables such that values for the latter can only be obtained if values for the former are supplied [15, 34, 12, 5, 6]. In this more general setting, and in sharp contrast to our simpler setting, a fundamental question is whether the query can even be answered for a given access pattern to each relation [28, 29, 27].

We introduce a fully dynamic evaluation approach for CQAPs. It is fully dynamic in the sense that it supports both inserts and deletes of tuples to the input database. It computes a data structure that supports the enumeration of the distinct output tuples for any values of the input variables and maintains this data structure under updates to the input database.

Our analysis of the overall computation time is refined into three components. The *preprocessing time* is the time to compute the data structure before receiving any updates. Given a tuple over the input variables, the *enumeration delay* is the time between the start of the enumeration process and the output of the first tuple, the time between outputting any two consecutive tuples, and the time between outputting the last tuple and the end of the enumeration process [13]. The *update time* is the time used to update the data structure[1] for one single-tuple update. The preprocessing step may be replaced by a sequence of inserts to the initially empty database. However, as shown in prior work on conjunctive queries under updates [19, 21], bulk inserts, as performed in the preprocessing step, may take asymptotically less time than a sequence of single-tuple inserts.

There are simple, albeit more expensive alternatives to our approach. For instance, on an update request we may only update the input database, and on an enumeration request we may use an existing enumeration algorithm for the residual query obtained by setting the input variables to constants in the original query. However, such an approach needs time-consuming and independent preparation for each enumeration request, e.g., to remove dangling tuples and possibly create a data structure to support enumeration. In contrast, the data structure constructed by our approach shares this preparation across the enumeration requests and can readily serve enumeration requests for any values of the input variables.

The contributions of this paper are as follows.

Section 3 introduces the CQAP language. Two new notions account for the nature of free access patterns: *access-top variable orders* and *query fractures*.

An access-top variable order is a decomposition of the query into a rooted forest of variables, where: the input variables are above all other variables; and the free (input and output) variables are above the bound variables. This variable order is compiled into a forest of view trees, which is a data structure that represents compactly the query output.

Since access to the query output requires fixing values for the input variables, the query can be fractured by breaking its joins on the input variables and replacing each of their occurrences with fresh variables within each connected component of the query hypergraph. This does not violate the access pattern, since each fresh input variable can be set to the corresponding given input value. Yet this may lead to structurally simpler queries whose dynamic evaluation admits lower complexity.

Section 3 also introduces the *static* and *dynamic* widths that capture the complexities of the preprocessing and respectively update steps. For a given CQAP, these widths are defined over the access-top variable orders of the fracture of the query.

Section 4 introduces our approach for CQAP evaluation. Computing and maintaining each view in the view tree accounts for preprocessing and respectively updates, while the view tree as a whole allows for the enumeration of the output tuples with constant delay.

[1] We do not allow updates during the enumeration; this functionality is orthogonal to our contributions and can be supported using a versioned data structure.

Section 5 gives a syntactic characterisation of those CQAPs that admit linear-time preprocessing and constant-time update and enumeration delay. We call this class CQAP_0. All queries outside CQAP_0 do not admit constant-time update and delay regardless of the preprocessing time, unless the widely held Online Matrix-Vector Multiplication conjecture [17] fails. Our dichotomy generalises a prior dichotomy for q-hierarchical queries *without access patterns* [7]. The q-hierarchical queries are in CQAP_0, yet they have no input variables. The class CQAP_0 further contains cyclic queries with input variables. For instance, the edge triangle detection problem is in CQAP_0: Given an edge (u, v), check whether it participates in a triangle. The smallest query patterns not in CQAP_0 strictly include the non-q-hierarchical ones and also contain others that are sensitive to the interplay of the output and input variables. Proving that they do not admit constant-time update and delay requires different and additional hardness reductions from the Online Matrix-Vector Multiplication problem.

Section 6 charts the preprocessing time - update time - enumeration delay trade-off for the dynamic evaluation of the class of CQAPs whose fractures are hierarchical. It shows that as the preprocessing and update times increase, the enumeration delay decreases. Our trade-off reveals the optimality for a particular class of CQAPs with hierarchical fractures, called CQAP_1, which lies outside CQAP_0: The complexity of CQAP_1 for both the update time and the enumeration delay matches the lower bound $\Omega(N^{\frac{1}{2}})$ for queries outside CQAP_0, where N is the size of the input database. This is weakly Pareto optimal as we cannot lower both the update time and delay complexities (whether one of them can be lowered remains open). Our approach for CQAP_1 exhibits a continuum of trade-offs: $\mathcal{O}(N^{1+\epsilon})$ preprocessing time, $\mathcal{O}(N^{\epsilon})$ amortized update time and $\mathcal{O}(N^{1-\epsilon})$ enumeration delay, for $\epsilon \in [0, 1]$. By tweaking the parameter ϵ, one can optimise the overall time for a sequence of enumeration and update tasks and achieve an asymptotically lower compute time than prior work. A well-studied query in CQAP_1 is the Dynamic Set Intersection problem [26]: We are given sets $S_1, \ldots, S_m$ subject to element insertions and deletions. For each access request (i, j) with $i, j \in [m]$, we need to decide whether the intersection of S_i and S_j is empty. Our approach recovers the complexity given by prior work [26] for this problem using $\epsilon = 0.5$.

2 Preliminaries

Data Model. A schema $\mathcal{X} = (X_1, \ldots, X_n)$ is a tuple of distinct variables. Each variable X_i has a discrete domain $\mathsf{Dom}(X_i)$. We treat schemas and sets of variables interchangeably, assuming a fixed ordering of variables. A tuple $\mathbf{x}$ of values has schema $\mathcal{X} = \mathsf{Sch}(\mathbf{x})$ and is an element from $\mathsf{Dom}(\mathcal{X}) = \mathsf{Dom}(X_1) \times \cdots \times \mathsf{Dom}(X_n)$. A relation R over schema $\mathcal{X}$ is a function $R : \mathsf{Dom}(\mathcal{X}) \to \mathbb{Z}$ such that the multiplicity $R(\mathbf{x})$ is non-zero for finitely many tuples $\mathbf{x}$. A tuple $\mathbf{x}$ is in R, denoted by $\mathbf{x} \in R$, if $R(\mathbf{x}) \neq 0$. The size $|R|$ of R is the size of the set $\{\mathbf{x} \mid \mathbf{x} \in R\}$. A database is a set of relations and has size given by the sum of the sizes of its relations. Given a tuple $\mathbf{x}$ over schema $\mathcal{X}$ and $\mathcal{S} \subseteq \mathcal{X}$, $\mathbf{x}[\mathcal{S}]$ is the restriction of $\mathbf{x}$ onto $\mathcal{S}$. For a relation R over schema $\mathcal{X}$, schema $\mathcal{S} \subseteq \mathcal{X}$, and tuple $\mathbf{t} \in \mathsf{Dom}(\mathcal{S})$: $\sigma_{\mathcal{S}=\mathbf{t}} R = \{\, \mathbf{x} \mid \mathbf{x} \in R \wedge \mathbf{x}[\mathcal{S}] = \mathbf{t} \,\}$ is the set of tuples in R that agree with $\mathbf{t}$ on the variables in $\mathcal{S}$; $\pi_{\mathcal{S}} R = \{\, \mathbf{x}[\mathcal{S}] \mid \mathbf{x} \in R \,\}$ stands for the set of tuples in R projected onto $\mathcal{S}$, i.e., the set of distinct $\mathcal{S}$-values from the tuples in R with non-zero multiplicities. For a relation R over schema $\mathcal{X}$ and $\mathcal{Y} \subseteq \mathcal{X}$, the *indicator projection* $I_{\mathcal{Y}} R$ is a relation over $\mathcal{Y}$ such that [2]:

$$\text{for all } \mathbf{y} \in \mathsf{Dom}(\mathcal{Y}) : I_{\mathcal{Y}} R(\mathbf{y}) = \begin{cases} 1 & \text{if there is } \mathbf{t} \in R \text{ such that } \mathbf{y} = \mathbf{t}[\mathcal{Y}] \\ 0 & \text{otherwise} \end{cases}$$

An update is a relation where tuples with positive multiplicities represent inserts and tuples with negative multiplicities represent deletes. Applying an update to a relation means unioning the update with the relation. A single-tuple update to a relation R is a singleton relation $\delta R = \{\mathbf{x} \to m\}$, where the multiplicity $m = \delta R(t)$ of the tuple t in δR is non-zero.

Computational Model. We consider the RAM model of computation. Each relation or materialised view R over schema $\mathcal{X}$ is implemented by a data structure that stores key-value entries $(\mathbf{x}, R(\mathbf{x}))$ for each tuple $\mathbf{x}$ with $R(\mathbf{x}) \neq 0$ and needs $O(|R|)$ space. This data structure can: (1) look up, insert, and delete entries in (amortised) constant time, (2) enumerate all stored entries in R with constant delay, and (3) report $|R|$ in constant time. For a schema $\mathcal{S} \subset \mathcal{X}$, we use an index data structure that for any $\mathbf{t} \in \mathsf{Dom}(\mathcal{S})$ can: (4) enumerate all tuples in $\sigma_{\mathcal{S}=\mathbf{t}}R$ with constant delay, (5) check $\mathbf{t} \in \pi_{\mathcal{S}}R$ in constant time; (6) return $|\sigma_{\mathcal{S}=\mathbf{t}}R|$ in constant time; and (7) insert and delete index entries in (amortised) constant time.

We next exemplify a data structure that conforms to the above computational model. Consider a relation (materialized view) R over schema $\mathcal{X}$. A hash table with chaining stores key-value entries $(\mathbf{x}, R(\mathbf{x}))$ for each tuple $\mathbf{x}$ over $\mathcal{X}$ with $R(\mathbf{x}) \neq 0$. The entries are doubly linked to support enumeration with constant delay. The hash table can report the number of its entries in constant time and supports lookups, inserts, and deletes in constant time on average, under the assumption of simple uniform hashing.

To support index operations on a schema $\mathcal{F} \subset \mathcal{X}$, we create another hash table with chaining where each table entry stores a tuple $\mathbf{t}$ of $\mathcal{F}$-values as key and a doubly-linked list of pointers to the entries in R having the $\mathcal{F}$-values $\mathbf{t}$ as value. Looking up an index entry given $\mathbf{t}$ takes constant time on average under simple uniform hashing, and its doubly-linked list enables enumeration of the matching entries in R with constant delay. Inserting an index entry into the hash table additionally prepends a new pointer to the doubly-linked list for a given $\mathbf{t}$; overall, this operation takes constant time on average. For efficient deletion of index entries, each entry in R also stores back-pointers to its index entries (one back-pointer per index for R). When an entry is deleted from R, locating and deleting its index entries in doubly-linked lists takes constant time per index.

3 Conjunctive Queries with Free Access Patterns

We introduce the queries investigated in this paper along with several of their properties. A *conjunctive query with free access patterns* (CQAP for short) has the form

$$Q(\mathcal{O}|\mathcal{I}) = R_1(\mathcal{X}_1), \ldots, R_n(\mathcal{X}_n).$$

We denote by: $(R_i)_{i\in[n]}$ the relation symbols; $(R_i(\mathcal{X}_i))_{i\in[n]}$ the atoms; $\mathit{vars}(Q) = \bigcup_{i\in[n]} \mathcal{X}_i$ the set of variables; $\mathit{atoms}(X)$ the set of the atoms containing the variable X; $\mathit{atoms}(Q) = \{R_i(\mathcal{X}_i) \mid i \in [n]\}$ the set of all atoms; and $\mathit{free}(Q) = \mathcal{O} \cup \mathcal{I} \subseteq \mathit{vars}(Q)$ the set of *free* variables, which are partitioned into *input* variables $\mathcal{I}$ and *output* variables $\mathcal{O}$. An empty set of input or output variables is denoted by a dot $(\cdot)$.

Given a database $\mathcal{D}$ and a tuple $\mathbf{i}$ over $\mathcal{I}$, the output of Q for the input tuple $\mathbf{i}$ is denoted by $Q(\mathcal{O}|\mathbf{i})$ and is defined by $\pi_{\mathcal{O}}\sigma_{\mathcal{I}=\mathbf{i}}Q(\mathcal{D})$: This is the set of tuples $\mathbf{o}$ over $\mathcal{O}$ such that the assignment $\mathbf{i} \circ \mathbf{o}$ to the free variables satisfies the body of Q.

The hypergraph of a query Q is $\mathcal{H} = (\mathcal{V} = \mathit{vars}(Q), \mathcal{E} = \{\{\mathcal{X}_i \mid i \in [n]\}\})$, whose vertices are the variables and hyperedges are the schemas of the atoms in Q. The *fracture* of a CQAP Q is a CQAP $Q_\dagger$ constructed as follows. We start with $Q_\dagger$ as a copy of Q. We replace each occurrence of an input variable by a fresh variable. Then, we compute the connected components of the hypergraph of the modified query. Finally, we replace in each connected component of the modified query all new variables originating from the same input variable by one input variable.

We next define the notion of dominance for variables in a CQAP Q. For variables A and B, we say that B *dominates* A if $\mathit{atoms}(A) \subset \mathit{atoms}(B)$. The query Q is *free-dominant* (*input-dominant*) if for any two variables A and B, it holds: if A is free (input) and B dominates A, then B is free (input). The query Q is *almost free-dominant* (*almost input-dominant*) if: (1) For any variable B that is not free (input) and for any atom $R(\mathcal{X}) \in \mathit{atoms}(B)$, there is another atom $S(\mathcal{Y}) \in \mathit{atoms}(B)$ such that $\mathcal{X} \cup \mathcal{Y}$ cover all free (input) variables dominated by B; (2) Q is not already free-dominant (input-dominant). A query Q is *hierarchical* if for any $A, B \in \mathit{vars}(Q)$, either $\mathit{atoms}(A) \subseteq \mathit{atoms}(B)$, $\mathit{atoms}(B) \subseteq \mathit{atoms}(A)$, or $\mathit{atoms}(B) \cap \mathit{atoms}(A) = \emptyset$. A query is q-hierarchical if it is hierarchical and free-dominant.

▶ **Definition 1.** *A query is in $CQAP_0$ if its fracture is hierarchical, free-dominant, and input-dominant. A query is in $CQAP_1$ if its fracture is hierarchical and is almost free-dominant, or almost input-dominant, or both.*

The subset of CQAP_0 without input variables is the class of q-hierarchical queries [7].

▶ **Example 2.** The query $Q_1(A, C \mid B, D) = R(A, B), S(B, C), T(C, D), U(A, D)$ is input-dominant, free-dominant, but not hierarchical. Its fracture $Q_\dagger(A, C \mid B_1, B_2, D_1, D_2) = R(A, B_1), S(B_2, C), T(C, D_1), U(A, D_2)$ is hierarchical but not input-dominant: C dominates both B_2 and D_1 and A dominates both B_1 and D_2, yet A and C are not input. It is however almost input-dominant: A is not input and for any of its atoms $R(A, B_1)$ and $U(A, D_2)$, there is another atom $U(A, D_2)$ and respectively $R(A, B_1)$ such that both $R(A, B_1)$ and $U(A, D_2)$ cover the variables B_1 and D_2 dominated by A; a similar reasoning applies to C. This means that Q_1 is in CQAP_1.

The query $Q_2(A \mid B) = S(A, B), T(B)$ is in CQAP_0, since its fracture $Q_\dagger(A \mid B_1, B_2) = S(A, B_1), T(B_2)$ is hierarchical, free-dominant, and input-dominant.

The query $Q_3(B \mid A) = S(A, B), T(B)$ is in CQAP_1. Its fracture is the query itself. It is hierarchical, yet not input-dominant, since B dominates A and is not input. It is, however, almost input-dominant: for each atom of B, there is one other atom such that together they cover A. Indeed, atom $S(A, B)$ already covers A, and it also does so together with $T(B)$; atom $T(B)$ does not cover A, but it does so together with $S(A, B)$.

The following are the smallest hierarchical queries that are not in CQAP_0 but in CQAP_1: $Q(A \mid \cdot) = R(A, B), S(B)$; $Q(B \mid A) = R(A, B), S(B)$; and $Q(\cdot \mid A) = R(A, B), S(B)$. ⌟

3.1 Variable Orders

Variable orders are used as logical plans for the evaluation of conjunctive queries [31]. We next adapt them to CQAPs. Given a query, two variables *depend* on each other if they occur in the same query atom. A *variable order* (VO) ω for a CQAP Q is a pair $(T_\omega, \mathit{dep}_\omega)$, where:

- T_ω is a (rooted) forest with one node per variable. The variables of each atom in Q lie along the same root-to-leaf path in T_ω.
- The function dep_ω maps each variable X to the subset of its ancestor variables in T_ω on which the variables in the subtree rooted at X depend.

An *extended* VO is a VO where we add as new leaves atoms corresponding to relations and their indicator projections. We add each atom in the query as child of its variable placed lowest in the VO. We explain next how the indicator projections are added to a VO ω. The role of the indicators is to reduce the asymptotic complexity of cyclic queries [2].

```
indicators(CQAP $Q$, VO $\omega$) : extended VO
switch $\omega$:
  $R(\mathcal{Y})$          1  return $R(\mathcal{Y})$

      $X$
     /   \
  $\omega_1 \cdots \omega_k$   2  let $\hat{\omega}_i$ = indicators($\omega_i$)  $\forall i \in [k]$
                 3  let $\mathcal{S} = \{X\} \cup dep_\omega(X)$ and $\mathcal{R}$ be the set of atoms in $\omega$
                 4  let $\mathcal{I} = \{ I_\mathcal{Z} R(\mathcal{Z}) \mid R(\mathcal{Y}) \in (atoms(Q) \setminus \mathcal{R})$ and $\mathcal{Z} = \mathcal{Y} \cap \mathcal{S} \neq \emptyset \}$
                 5  let $\{I_1, ..., I_\ell\} = \text{GYO}^*(\mathcal{I}, \mathcal{R})$
                                    $X$
                             /     /  \     \
                 6  return { $\hat{\omega}_1 \cdots \hat{\omega}_k$   $I_1 \cdots I_\ell$
```

Figure 1 Adding indicator projections to a VO ω of a CQAP Q. The function indicators is defined using pattern matching on the structure of the VO ω, which can be a leaf or an inner node (cf. left column under **switch**). Each variable X in ω gets as new children the indicator projections of relations that do not occur in the subtree rooted at X but form a cycle with those that occur. GYO* (Section 3.1) is based on the GYO reduction [4].

Given a CQAP Q and a VO ω, where the atoms of Q have been already added, the function indicators in Figure 1 extends ω with indicator projections. At each variable X in ω, we compute the set $\mathcal{I}$ of all possible indicator projections (Line 4). Such indicators $I_\mathcal{Z} R$ are for relations R whose atoms are not included in the subtree rooted at X but share a non-empty set $\mathcal{Z}$ of variables with $\{X\} \cup dep_\omega(X)$. We choose from this set those indicators that form a cycle with the atoms in the subtree of ω rooted at X (Line 5). We achieve this using a variant of the GYO reduction [4]. Given the hypergraph formed by the hyperedges representing these indicators $\mathcal{I}$ and the atoms $\mathcal{R}$, GYO repeatedly applies two rules until it reaches a fixpoint: (1) Remove a node that only appears in one hyperedge; (2) Remove a hyperedge that is included in another hyperedge. If the result of GYO is a hypergraph with no nodes and one empty hyperedge, then the input hypergraph is (α-)acyclic. Otherwise, the input hypergraph is cyclic and the GYO's output is a hypergraph with cycles. Our GYO variant, dubbed GYO* in Figure 1, returns the hyperedges that originated from the indicator projections in $\mathcal{I}$ and contribute to this non-empty output hypergraph. The chosen indicator projections become children of X (Line 6).

In the rest of this paper, whenever we refer to a variable order, we always assume an extended VO.

▶ **Example 3.** Consider the triangle CQAP query

$$Q(B, C|A) = R(A, B), S(B, C), T(C, A).$$

The fracture $Q_\dagger$ of Q is the query itself. Figure 2 depicts a VO ω for Q. The input variable A is on top of the output variables B and C. At variable C, the function indicators from Figure 1 creates an indicator projection $I_{A,B}R$ since the relation R is not under C but forms a cycle with the relations S and T. ⌟

We introduce notation for an extended VO ω. Its subtree rooted at X is denoted by ω_X. The sets $vars(\omega)$ and $\mathsf{anc}_\omega(X)$ consist of all variables of ω and respectively the variables on the path from X to the root excluding X. We denote by $atoms(\omega)$ all atoms and indicators at the leaves of ω and by Q_X the join of all atoms $atoms(\omega)$ (all variables are free).

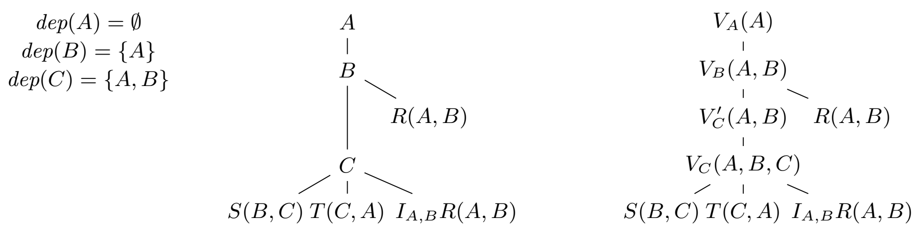


Figure 2 Left: (Access-top extended) VO for the query $Q(B, C|A) = R(A, B), S(B, C), T(C, A)$. Right: The view tree constructed from this VO. Note the indicator $I_{A,B}R(A, B)$ added below the variable C (left) and below the view V_C (right).

We next introduce classes of VOs for CQAP queries. A VO ω is *canonical* if the variables of the leaf atom of each root-to-leaf path are the inner nodes of the path. Hierarchical queries are precisely those conjunctive queries that admit canonical variable orders. A VO ω is *free-top* if no bound variable is an ancestor of a free variable. It is *input-top* if no output variable is an ancestor of an input variable. The sets of free-top and input-top VOs for Q are denoted as $\mathsf{free\text{-}top}(Q)$ and $\mathsf{input\text{-}top}(Q)$, respectively. A VO is called *access-top* if it is free-top and input-top: $\mathsf{acc\text{-}top}(Q) = \mathsf{free\text{-}top}(Q) \cap \mathsf{input\text{-}top}(Q)$.

▶ **Example 4.** The query $Q(B|A) = R(A, B), S(B)$ admits the VO (in term notation; "-" represents the parent-child relationship): $B - \{A - R(A, B), S(B)\}$, where B has the variable A and the atom $S(B)$ as children and A has the atom $R(A, B)$ as child. The dependency sets are $dep(B) = \emptyset$ and $dep(A) = \{B\}$. This VO is free-top, since both variables are free; it is not input-top, since the output variable B is on top of the input variable A. By swapping A and B in the order, it becomes input-top and then also access-top; the dependencies then become: $dep(A) = \emptyset$ and $dep(B) = \{A\}$.

The triangle query $Q(A, B|\cdot) = R(A, B), S(B, C), T(A, C)$ admits the VO $C - A - \{T(A, C), B - \{R(A, B), S(B, C), I_{AC}T(A, C)\}\}$, where one child of B is the indicator projection $I_{AC}T$ of T on $\{A, C\}$. The dependency sets are $dep(C) = \emptyset$, $dep(A) = \{C\}$, and $dep(B) = \{A, C\}$. The VO is input-top, since the query has no input variables; it is not free-top, since the bound variable C is on top of the free variables A and B.

The fracture of the 4-cycle query in Example 2 admits the access-top VO consisting of two disconnected paths: $B_1 - D_2 - A - \{R(A, B_1), U(A, D_2)\}$ and $B_2 - D_1 - C - \{S(B_2, C), T(C, D_1)\}$, where the dependency sets are: $dep(A) = \{B_1, D_2\}$, $dep(D_2) = \{B_1\}$, $dep(B_1) = dep(B_2) = \emptyset$, $dep(C) = \{B_2, D_1\}$, and $dep(D_1) = \{B_2\}$. ⌟

3.2 Width Measures

We next introduce two width measures for a VO ω and CQAP Q. They capture the complexity of computing and maintaining the output of Q.

▶ **Definition 5.** *The static width* $\mathsf{w}(\omega)$ *and dynamic width* $\delta(\omega)$ *of a VO* ω *are:*

$$\mathsf{w}(\omega) = \max_{X \in vars(\omega)} \rho^*_{Q_X}(\{X\} \cup dep_\omega(X))$$

$$\delta(\omega) = \max_{X \in vars(\omega)} \max_{R(\mathcal{Y}) \in atoms(\omega_X)} \rho^*_{Q_X}((\{X\} \cup dep_\omega(X)) \setminus \mathcal{Y})$$

For a query Q_X and a set of variables $\mathcal{X} = \{X\} \cup dep_\omega(X)$, the fractional edge cover number [3] $\rho^*_{Q_X}(\mathcal{X})$ defines a worst-case upper bound on the time needed to compute $Q_X(\mathcal{X})$. Here, Q_X is the join of all atoms under X in the VO ω. The static width w of a VO ω is then defined by the maximum over the fractional edge cover numbers of the queries Q_X for the variables X in ω. The dynamic width is defined similarly, with one simplification: We consider every case of a relation (or indicator projection) R being replaced by a single-tuple update, so its variables $\mathcal{Y}$ are all set to constants and can be ignored in the computation of the fractional edge cover number.

We consider the standard lexicographic ordering $\leq$ on pairs of dynamic and static widths: $(\delta_1, \mathsf{w}_1) \leq (\delta_2, \mathsf{w}_2)$ if $\delta_1 \leq \delta_2$ or $\delta_1 = \delta_2$ and $\mathsf{w}_1 \leq \mathsf{w}_2$. Given a set $\mathcal{S}$ of VOs, we define $\min_{\omega \in \mathcal{S}}(\delta(\omega), \mathsf{w}(\omega)) = (\delta, \mathsf{w})$ such that $\forall \omega \in \mathcal{S} : (\delta, \mathsf{w}) \leq (\delta(\omega), \mathsf{w}(\omega))$.

▶ **Definition 6.** *The dynamic width $\delta(Q)$ and static width $\mathsf{w}(Q)$ of a CQAP Q are:*

$$(\delta(Q), \mathsf{w}(Q)) = \min_{\omega \in \mathsf{acc\text{-}top}(Q_\dagger)} (\delta(\omega), \mathsf{w}(\omega))$$

Since we are interested in dynamic evaluation, Definition 6 first minimises for the dynamic width and then for the static width. To determine the dynamic and the static width of a CQAP Q, we first search for the VOs of the fracture $Q_\dagger$ with minimal dynamic width and choose among them one with the smallest static width. The extended technical report [22] further expands on the width measures with examples and properties.

▶ **Example 7.** Consider the query $Q(\mathcal{O} \mid \mathcal{I}) = R(A, B, C), S(A, B, D), T(A, E)$. The static width w and the dynamic width δ of Q vary depending on the access pattern:
For $Q(\{C, D, E\} \mid \{A, B\})$, $\mathsf{w} = 1$ and $\delta = 0$. For $Q(\{A, C, D, E\} \mid \{B\})$, $\mathsf{w} = 1$ and $\delta = 1$.
For $Q(\{A, C, D\} \mid \{B, E\})$, $\mathsf{w} = 2$ and $\delta = 1$. For $Q(\{A, E\} \mid \{B, C, D\})$, $\mathsf{w} = 2$ and $\delta = 2$.
For $Q(\{A, B\} \mid \{C, D, E\})$, $\mathsf{w} = 3$ and $\delta = 2$. For $Q(\{A, B, C, D, E\}|\cdot)$, $Q(\cdot|\{A, B, C, D, E\})$ and $Q(\{B, C, D, E\}|\{A\})$, $\mathsf{w} = 1$ and $\delta = 0$.

Recall the triangle CQAP query $Q(B, C|A) = R(A, B), S(B, C), T(C, A)$ from Example 3 and its access-top VO in Figure 2. By adding the indicator $I_{A,B}R$ below C, the fractional edge cover number $\rho^*(\{C\} \cup \mathit{dep}(C)) = \rho^*(\{A, B, C\})$ of the query $Q_C(A, B, C) = S(B, C), T(C, A), I_{A,B}R(A, B)$ reduces from 2 to $\frac{3}{2}$. This fractional edge cover number is the largest one among the fractional edge cover numbers of the queries induced by other variables, thus the static width of the VO ω is $\frac{3}{2}$.

The dynamic width of ω is dominated by the fractional edge cover number $\rho^*(\{C\} \cup \mathit{dep}(C)) - \mathcal{S}) = \rho^*(\{A, B, C\} - \mathcal{S})$ of the query Q_C, where $\mathcal{S}$ is the schema of S, T, or $I_{A,B}R$. In each of these three cases, $\{A, B, C\} - \mathcal{S}$ consists of a single variable. Hence, the fractional edge cover number is 1 and then the dynamic width of ω is 1. ⌟

4 CQAP Evaluation

In this section, we introduce a fully dynamic evaluation approach for arbitrary CQAPs whose complexity is stated in the following theorem.

▶ **Theorem 8.** *Given a CQAP with static width w and dynamic width δ and a database of size N, the query can be evaluated with $\mathcal{O}(N^{\mathsf{w}})$ preprocessing time, $\mathcal{O}(N^{\delta})$ update time under single-tuple updates, and $\mathcal{O}(1)$ enumeration delay.*

Our approach has three stages: preprocessing, enumeration, and updates. They are detailed in the following subsections. We consider in the following a fixed CQAP $Q(\mathcal{O}|\mathcal{I})$, its fracture $Q_\dagger(\mathcal{O}|\mathcal{I}_\dagger)$, and a database of size N.

```
$\tau$(VO $\omega$) : view tree
switch $\omega$:
  $R(\mathcal{Y})$
      1  return $R(\mathcal{Y})$
  $X$ with children $\omega_1 \cdots \omega_k$
      2  let $T_i = \tau(\omega_i)\quad \forall i \in [k]$
      3  let $\mathcal{S} = \{X\} \cup dep_\omega(X)$ and $V_X(\mathcal{S})$ = join of roots of $T_1, \ldots, T_k$
      4  if $X$ has no sibling   return $V_X(\mathcal{S})$ with children $T_1 \cdots T_k$
      5  let $V'_X(\mathcal{S} \setminus \{X\}) = V_X(\mathcal{S})$   return $V'_X(\mathcal{S} \setminus \{X\})$ with child $V_X(\mathcal{S})$ with children $T_1 \cdots T_k$
```

Figure 3 Constructing a view tree following a VO ω. The function τ is defined using pattern matching on the structure of the VO ω, which can be a leaf or an inner node (cf. left column under **switch**). At each variable X in ω, the function creates a view V_X whose schema consists of X and the dependency set of X. If X has siblings, it adds a view on top of V_X that marginalises out X.

4.1 Preprocessing

In the preprocessing stage, we construct a set of view trees that represent the result of $Q_\dagger$ over both its input and output variables. A view tree [30] is a (rooted) tree with one view per node. It is a logical project-join plan in the classical database systems literature, but where each intermediate result is materialised. The view at a node is defined as the join of the views at its children, possibly followed by a projection. The view trees are modelled following an access-top VO ω of $Q_\dagger$. In the following, we discuss the case of ω consisting of a single tree; otherwise, we apply the preprocessing stage to each tree in ω.

Given an access-top VO ω, the function $\tau(\omega)$ in Figure 3 returns a view tree constructed from ω. The function traverses ω bottom-up and creates at each variable X, a view V_X defined over the join of the child views of X. The schema of V_X consists of X and the dependency set of X (Line 3). This view allows to efficiently enumerate the X-values given a tuple of values for the variables in the dependency set. If X has siblings, the function creates an additional view V'_X on top of V_X whose purpose is to aggregate away (or marginalise out) X from V_X (Line 5). This view allows to efficiently maintain the ancestor views of V_X under updates to the views created for the siblings of X.

The time to construct the view tree $\tau(\omega)$ is dominated by the time to materialise the view V_X for each variable X. The auxiliary view V'_X above V_X can be materialised by marginalising out X in one scan over V_X. Each view V_X can be materialised in $\mathcal{O}(N^{\mathsf{w}})$ time, where $\mathsf{w} = \rho^*_{Q_X}(\{X \cup dep_\omega(X)\})$. The definition of the static width of ω implies that the view tree $\tau(\omega)$ can be constructed in $\mathcal{O}(N^{\mathsf{w}(\omega)})$ time. By choosing a VO whose static width is $\mathsf{w}(Q)$, the preprocessing time of our approach becomes $\mathcal{O}(N^{\mathsf{w}(Q)})$, as stated in Theorem 8.

The next example demonstrates the construction of a view tree for a CQAP$_0$ query.

▶ **Example 9.** Figure 4 shows the hypergraphs of the query $Q(B,C,D,E|A) = R(A,B,C)$, $S(A,B,D)$, $T(A,E)$ and its fracture $Q_\dagger(B,C,D,E|A_1,A_2) = R(A_1,B,C)$, $S(A_1,B,D)$, $T(A_2,E)$. The fracture has two connected components: $Q_1(B,C,D|A_1) = R(A_1,B,C)$,

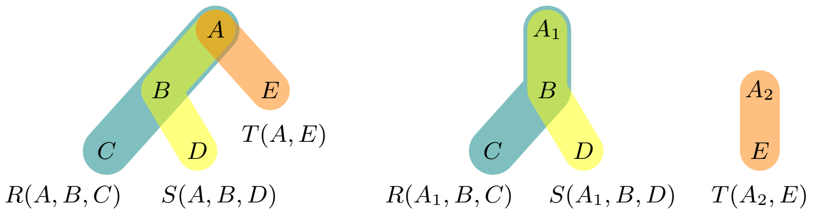


Figure 4 (Left) Hypergraph of the two queries with the same body but different access patterns, as used in Examples 9 and 10; (middle and right) hypergraph of their fractures.

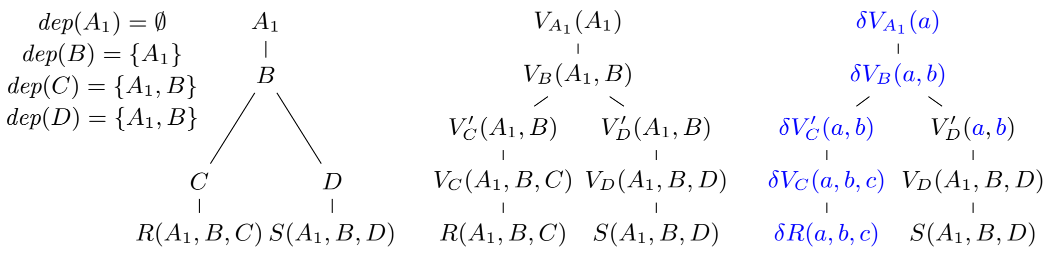


Figure 5 (Left) Access-top VO for $Q_1(B,C,D|A_1) = R(A_1,B,C), S(A_1, B,D)$; (middle) the view tree constructed from the VO; (right) the delta view tree under a single-tuple update to R.

$S(A_1, B,D)$ and $Q_2(E|A_2) = T(A_2,E)$. Figure 5 depicts an access-top VO (left) for Q_1 and its corresponding view tree (middle). The VO has static width 1. Each variable in the VO is mapped to a view in the view tree, e.g., B is mapped to $V_B(A_1,B)$, where $\{B,A_1\} = \{B\} \cup dep(B)$. The views V'_C and V'_D are auxiliary views. The views V'_C, V'_D, and V_{A_1} marginalise out the variables C, D and respectively B from their child views. The view V_B is the intersection of V'_C and V'_D. Hence, all views can be computed in $\mathcal{O}(N)$ time. Since the query fracture is acyclic, the view tree does not contain indicator projections.

The only access-top VO for the connected component Q_2 of $Q_\dagger$ is the top-down path $A_2 - E - T(A_2,E)$. The views mapped to A_2 and E are $V_{A_2}(A_2)$ and respectively $V_E(A_2,E)$. They can obviously be computed in $\mathcal{O}(N)$ time. ⌟

The next example considers a CQAP$_1$ whose preprocessing time is quadratic.

▶ **Example 10.** Consider the CQAP$_1$ $Q(E,D|A,C) = R(A,B,C), S(A,B,D), T(A,E)$ and its fracture $Q_\dagger(E,D|A_1,A_2,C) = R(A_1,B,C), S(A_1,B,D), T(A_2,E)$. The fracture has the two connected components $Q_1(B,D|A_1,C) = R(A_1,B,C), S(A_1,B,D)$ and $Q_2(E|A_2) = T(A_2,E)$. The hypergraphs (Figure 4) of Q and its fracture are the same as for the query in Example 9. Figure 6 depicts an access-top VO (left) for Q_1 and its corresponding view tree (middle). The VO has static width 2. The view V_B joins the relations R and S, which takes $\mathcal{O}(N^2)$ time. The views V_D, V_C, and V_A are constructed from V_B by marginalising out one variable at a time. Hence, the view tree construction takes $\mathcal{O}(N^2)$ time. The view tree for Q_2 is the same as in Example 9 and can be constructed in linear time. ⌟

Finally, we exemplify the construction of a view tree for a cyclic query.

▶ **Example 11.** Figure 2 depicts a VO and the view tree constructed from it for the triangle CQAP query $Q(B,C|A) = R(A,B), S(B,C), T(C,A)$ from Example 3. The view V_C joins the relations R and S and the indicator projection $I_{A,B}R$, which can be computed in $\mathcal{O}(N^{\frac{3}{2}})$ time using a worst-case optimal join algorithm. The view V_B can be computed in linear

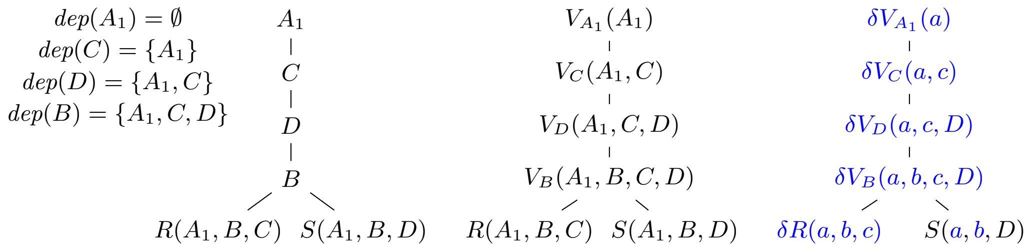


Figure 6 (Left) Access-top VO for $Q_1(B, D|A_1, C) = R(A_1, B, C), S(A_1, B, D)$; (middle) the view tree corresponding to the VO; (right) the delta view tree under a single-tuple update to R.

time by looking up each tuple from V'_C in R. The views V'_C and V_A are constructed by marginalising out one variable at a time in time $\mathcal{O}(N^{\frac{3}{2}})$ and $\mathcal{O}(N)$ time, respectively. Hence, the view tree construction takes $\mathcal{O}(N^{\frac{3}{2}})$ time. ⌟

4.2 Enumeration

The view trees constructed by the function τ for any access-top VO for $Q_\dagger$ allow for constant-delay enumeration of the tuples in $Q(\mathcal{O}|\mathbf{i})$ given any tuple $\mathbf{i}$ over the input variables $\mathcal{I}$.

Assume that ω_i is a tree in the forest ω for which $\tau(\omega_i)$ constructs the view tree T_i, for $i \in [n]$. Let $Q_i(\mathcal{O}_i|\mathcal{I}_i)$ with $\mathcal{O}_i = \mathcal{O} \cap vars(\omega_i)$ and $\mathcal{I}_i = \mathcal{I}_\dagger \cap vars(\omega_i)$ be the CQAP that joins the atoms at the leaves of T_i. We first explain how to enumerate the tuples in $Q_i(\mathcal{O}_i \mid \mathbf{i})$ from T_i with constant delay, given an input tuple $\mathbf{i}$ over $\mathcal{I}_i$. We traverse the view tree T_i in preorder and execute at each view V_X the following steps. In case $X \in \mathcal{I}_i$, we check whether the projection of $\mathbf{i}$ onto the schema of V_X is in V_X. If not, the query output is empty and we stop. Otherwise, we continue with the preorder traversal. In case $X \in \mathcal{O}_i$, we retrieve in constant time the first X-value in V_X given that the values over the variables in the root path of X are already fixed to constants. After all views are visited once, we have constructed the first complete output tuple and report it. Then, we iterate with constant delay over the remaining distinct X-values in the last visited view V_X. For each distinct X-value, we obtain a new tuple and report it. After all X-values in V_X are exhausted, we backtrack.

Assume now that we have a procedure that enumerates the tuples in $Q_i(\mathcal{O}_i \mid \mathbf{i}_i)$ for any tuple $\mathbf{i}_i$ over $\mathcal{I}_i$ with constant delay. Consider a tuple $\mathbf{i}$ over the input variables $\mathcal{I}$ of Q. It holds $Q(\mathcal{O}|\mathbf{i}) = \times_{i\in[n]} Q_i(\mathcal{O}_i|\mathbf{i}_i)$ where $\mathbf{i}_i[X'] = \mathbf{i}[X]$ if $X = X'$ or X is replaced by X' when constructing the fracture of Q. We can enumerate the tuples in $Q(\mathcal{O} \mid \mathbf{i})$ with constant delay by nesting the enumeration procedures for $Q_1(\mathcal{O}_1 \mid \mathbf{i}_1), \ldots, Q_n(\mathcal{O}_n \mid \mathbf{i}_n)$.

▶ **Example 12.** Consider the query $Q(B, C, D, E|A)$ from Example 9 and the two connected components $Q_1(B, C, D|A_1)$ and $Q_2(E|A_2)$ of its fracture. Figure 5 (middle) depicts the view tree for Q_1. Given an A_1-value a, we can use this view tree to enumerate the distinct tuples in $Q_1(B, C, D|a)$ with constant delay. We first check if a is included in the view V_{A_1}. If not, $Q_1(B, C, D|a)$ must be empty and we stop. Otherwise, we retrieve the first B-value b paired with a in V_B, the first C-value c paired with (a, b) in V_C, and the first D-value d paired with (a, b) in V_D. Thus, we obtain in constant time the first output tuple (b, c, d) in $Q_1(B, C, D|a)$ and report it. Then, we iterate over the remaining distinct D-values paired with (a, b) in V_D and report for each such D-value d', a new tuple (b, c, d'). After all D-values are exhausted, we retrieve the next distinct C-value paired with (a, b) in V_C and restart the iteration over the distinct D-values paired with (a, b) in V_D, and so on. Overall, we construct each distinct tuple in $Q_1(B, C, D|a)$ in constant time after the previous one is constructed.

Assume now that we have constant-delay enumeration procedures for the tuples in $Q_1(B, C, D|a)$ and the tuples in $Q_2(E|a)$ for any A-value a. We can enumerate with constant delay the tuples in $Q(B, C, D, E|a)$ as follows. We ask for the first tuple (b, c, d) in $Q_1(B, C, D|a)$ and then iterate over the distinct E-values in $Q_2(E|a)$. For each such E-value e, we report the tuple (b, c, d, e). Then, we ask for the next tuple in $Q_1(B, C, D|a)$ and restart the enumeration over the tuples in $Q_2(E|a)$, and so on. ⌟

4.3 Updates

We now explain how to update the view trees constructed by the function τ in Figure 3. Consider a single-tuple update $\delta R = \{\mathbf{x} \to m\}$ to an input relation R; m is positive in case of insertion and negative in case of deletion. We first update each view tree that has an atom $R(\mathcal{X})$ at a leaf: We update each view on the path from that leaf to the root of the view tree using the classical delta rules [9]. The update δR may affect indicator projections $I_{\mathcal{Z}}R$. A new single-tuple update $\delta I_{\mathcal{Z}}R = \{\mathbf{x}[\mathcal{Z}] \to k\}$ to $I_{\mathcal{Z}}R$ is triggered in the following two cases. If δR is an insertion and $\mathbf{x}[\mathcal{Z}]$ is a value not already in $\pi_{\mathcal{Z}}R$, then the new update is triggered with $k = 1$. If δR is a deletion and $\pi_{\mathcal{Z}}R$ does not contain $\mathbf{x}[\mathcal{Z}]$ after applying the update to R, then the new update is triggered with $k = -1$. This update is propagated up to the root of each view tree, like for δR.

Recall that the time to compute a view V_X is $\mathcal{O}(N^{\mathsf{w}})$, where $\mathsf{w} = \rho^*_{Q_X}(\{X\} \cup dep_\omega(X))$. In case of an update to a relation or indicator R over schema $\mathcal{Y}$, the variables in $\mathcal{Y}$ are set to constants. The time to update V_X is then $\mathcal{O}(N^{\delta})$, where $\delta = \rho^*_{Q_X}((\{X\} \cup dep_\omega(X)) \setminus \mathcal{Y})$. Assuming that the dynamic width of ω is $\delta(Q)$, we conclude that the update time of our approach is $\mathcal{O}(N^{\delta(Q)})$, as stated in Theorem 8.

▶ **Example 13.** Figure 5 (right) shows the delta view tree for the view tree to the left under a single-tuple update $\delta R(a, b, c)$ to R. We update the relation $R(A, B, C)$ with $\delta R(a, b, c)$ in constant time. The ancestor views of δR (in blue) are the deltas of the corresponding views, computed by propagating δR from the leaf to the root. They can also be effected in constant time. Overall, maintaining the view tree under a single-tuple update to any relation takes $O(1)$ time.

Consider now the delta view tree in Figure 6 (right) obtained from the view tree to its left under the single-tuple update $\delta R(a, b, c)$. We update $V_B(A_1, B, C, D)$ with $\delta V_B(a, b, c, D) = \delta R(a, b, c), S(a, b, D)$ in $\mathcal{O}(N)$ time, since there are at most N D-values paired with (a, b) in S. We then update the views V_D, V_C, and V_{A_1} in $\mathcal{O}(1)$ time. Updates to S are handled analogously. Overall, maintaining the view tree under a single-tuple update to any relation takes $O(N)$ time. ⌟

4.4 Discussion

So far in this section, we explained how our approach works. We conclude with a high-level discussion on key decisions behind our approach.

1. Variable orders. Our approach can be rephrased to use tree decompositions [16] instead of VOs, since they are different syntaxes for the same query decomposition class [31]. Indeed, the set consisting of a variable and its dependency set in a VO can be interpreted as a bag of a tree decomposition whose edges between bags reflect those between the variables in the VO. Variable orders are more natural for our algorithms for constructing view trees and for enumeration as well as worst-case optimal join algorithms such as the LeapFrog

TrieJoin [33] and their use for constructing factorized representations of query results [31]: These algorithms proceed one variable at a time and not one bag of variables at a time. VO-based algorithms express more naturally computation by variable elimination.

2. Access-top VOs. Access-top VOs can have higher static and dynamic widths than arbitrary VOs. However, they are needed to attain the constant-delay enumeration in Theorem 8, as explained next. The maintenance procedure for view trees ensures that each view is calibrated[2] with respect to all of its descendant views and relations, since the updates are propagated bottom-up from the relations to the top view. Since the views constructed for the input variables are above all other views in a view tree constructed from an access-top VO, these views are calibrated. For a given tuple of values over the input variables, the calibration of these views guarantees that if they do not agree with this tuple, then there is no output tuple associated with the input tuple. For constant-delay enumeration, we follow a top-down traversal of the view tree and use the constant-time lookup of the hash maps implementing the views. Furthermore, since the output variables are above the bound variables in the VO, tuples of values over the output variables can be retrieved from views whose schemas do not contain bound variables. Hence, we can enumerate the *distinct* tuples over the output variables for a given tuple over the input variables.

In case we would have used an arbitrary (and not access-top) VO, then the input variables may be anywhere in the VO; in particular, there may be views above the relations with the input variables that do not have input variables. On an enumeration request, the values given to the input variables act as selection conditions on the relations and may require the calibration of the views on top before the enumeration starts; this calibration may be as expensive as computing the query. Otherwise, we incur a non-constant cost for the enumeration of each output tuple. Either way, the enumeration delay may not be constant.

3. Lazy approach using residual queries. A simple CQAP evaluation approach is the lazy approach. On updates, the lazy approach just updates the input relations. On enumeration, where each input variable is given a value, it computes the residual query obtained by setting the input variables to the given values. The enumeration of the tuples in the output of a residual query cannot guarantee constant delay, since the parts of the input relations, which satisfy the selection conditions on the input variables, are not necessarily calibrated, and the calibration may take as much time as computing the residual query.

4. Replacing each occurrence of an input variable by a fresh variable. Although this query rewriting removes the joins on the input variables, it does not affect the correctness of query evaluation. For enumeration, all fresh variables are fixed to given values. In access-top VOs, these variables are above the other variables and are in views that are calibrated with respect to the relations in their respective connected component of the rewritten query. We can then check whether all view trees satisfy the assignment of values to the input values. If a view tree fails, then the query output is empty for the values given to the input variables.

5. Query fractures. The query rewriting in the previous discussion point is only the first step of query fracturing. The second step merges all fresh variables for an input variable into one variable in case they are in the same connected component. This does not affect correctness

[2] A relation R is calibrated with respect to other relations in a query Q if each tuple in R participates to at least one tuple in the output of Q.

but may affect the complexity, as exemplified next. Consider the triangle query in Example 11: $Q(B, C|A) = R(A, B), S(B, C), T(C, A)$. If we were to replace A by two fresh variables A_1 and A_2, then the rewritten query would be: $Q'(B, C|A_1, A_2) = R(A_1, B), S(B, C), T(C, A_2)$. It still has one connected component. An access-top VO for Q' is $A_1 - A_2 - B - C$ (A_1 and A_2 may be swapped, same for B and C). The static width of Q' is 2. Yet by merging back A_1 and A_2, we obtain Q, which admits the access-top VO $A - B - C$ and static width 3/2 (same width can be obtained if B and C are swapped), as in Example 11.

5 A Dichotomy for CQAPs

The following dichotomy states that the queries in CQAP$_0$ are precisely those CQAPs that can be evaluated with constant update time and enumeration delay.

▶ **Theorem 14.** *Let any CQAP query Q and database of size N.*

- *If Q is in CQAP$_0$, then it admits $\mathcal{O}(N)$ preprocessing time, $\mathcal{O}(1)$ enumeration delay, and $\mathcal{O}(1)$ update time for single-tuple updates.*
- *If Q is not in CQAP$_0$ and has no repeating relation symbols, then there is no algorithm that computes Q with arbitrary preprocessing time, $\mathcal{O}(N^{\frac{1}{2}-\gamma})$ enumeration delay, and $\mathcal{O}(N^{\frac{1}{2}-\gamma})$ amortised update time, for any $\gamma > 0$, unless the OMv conjecture fails.*

The hardness result in Theorem 14 is based on the following OMv problem:

▶ **Definition 15** (Online Matrix-Vector Multiplication (OMv) [17]). *We are given an $n \times n$ Boolean matrix $\mathbf{M}$ and receive n Boolean column vectors $\mathbf{v}_1, \ldots, \mathbf{v}_n$ of size n, one by one; after seeing each vector $\mathbf{v}_i$, we output the product $\mathbf{M}\mathbf{v}_i$ before we see the next vector.*

It is strongly believed that the OMv problem cannot be solved in subcubic time.

▶ **Conjecture 16** (OMv Conjecture, Theorem 2.4 [17]). *For any $\gamma > 0$, there is no algorithm that solves the OMv problem in time $\mathcal{O}(n^{3-\gamma})$.*

Queries in CQAP$_0$ have dynamic width 0 and static width 1 [22]. Our approach from Section 4 achieves linear preprocessing time, constant update time and enumeration delay for such queries (Theorem 8), so it is optimal for CQAP$_0$.

The smallest queries not included in CQAP$_0$ are: $Q_1(\mathcal{O}|\cdot) = R(A), S(A, B), T(B)$ with $\mathcal{O} \subseteq \{A, B\}$; $Q_2(A|\cdot) = R(A, B), S(B)$; $Q_3(\cdot|A) = R(A, B), S(B)$; and $Q_4(B|A) = R(A, B), S(B)$. Each query is equal to its fracture. Query Q_1 is not hierarchical. Q_2 is not free-dominant. Q_3 and Q_4 are not input-dominant. Prior work showed that there is no algorithm that achieves constant update time and enumeration delay for Q_1 and Q_2, unless the OMv conjecture fails [7]. To prove the hardness statement in Theorem 14, we show that this negative result also holds for Q_3 and Q_4. Then, given an arbitrary CQAP Q that is not in CQAP$_0$, we reduce the evaluation of one of the four queries above to the evaluation of Q.

6 Trade-Offs for CQAPs with Hierarchical Fractures

For CQAPs with hierarchical fractures, the complexities in Theorem 8 can be parameterised to uncover trade-offs between preprocessing, update, and enumeration.

▶ **Theorem 17.** *Let any CQAP Q with static width w and dynamic width δ, a database of size N, and $\epsilon \in [0, 1]$. If Q's fracture is hierarchical, then Q admits $\mathcal{O}(N^{1+(\mathsf{w}-1)\epsilon})$ preprocessing time, $\mathcal{O}(N^{1-\epsilon})$ enumeration delay, and $\mathcal{O}(N^{\delta\epsilon})$ amortised update time for single-tuple updates.*

This trade-off continuum can be obtained using one algorithm parameterised by ϵ. This algorithm either recovers or has lower complexity than prior approaches. Using $\epsilon = 1$, we recover the complexities in Theorem 8 and therefore also the constant update time and delay for queries in CQAP_0 in Theorem 14.

Theorem 17 can be refined for CQAP_1, since $\delta = 1$ and $\mathsf{w} \leq 2$ for queries in this class.

▶ **Corollary 18** (Theorem 17). *Let any query in $CQAP_1$, a database of size N, and $\epsilon \in [0, 1]$. Then Q admits $\mathcal{O}(N^{1+\epsilon})$ preprocessing time, $\mathcal{O}(N^{1-\epsilon})$ enumeration delay, and $\mathcal{O}(N^{\epsilon})$ amortised update time for single-tuple updates.*

For $\epsilon = 0.5$, the update time and delay for queries in CQAP_1 match the lower bound in Theorem 14 for all queries outside CQAP_0. This makes our approach weakly Pareto optimal for CQAP_1, as lowering both the update time and delay would violate the OMv conjecture.

Our algorithm has two core ideas. (For lack of space, we defer the details to the extended technical report [22].) First, we partition the input relations into heavy and light parts based on the degrees of the values. This transforms a query over the input relations into a union of queries over heavy and light relation parts. Second, we employ different evaluation strategies for different heavy-light combinations of parts of the input relations. This allows us to confine the worst-case behaviour caused by high-degree values in the database during query evaluation.

We construct a set of VOs for the hierarchical fracture of a given CQAP. Each VO represents a different evaluation strategy over heavy and light relation parts. For VOs over light relation parts, we follow the general approach from Section 4 and construct view trees from access-top VOs. For VOs involving heavy relation parts, we construct view trees from VOs that are not access-top, thus yielding non-constant enumeration delay but better preprocessing and update times. This trade-off is controlled by the parameter ϵ.

Enumerating distinct tuples from the constructed view trees poses two challenges. First, these view trees may encode overlapping subsets of the query result. To enumerate only distinct tuples from these view trees, we use the union algorithm [14] and view tree iterators, as in prior work [23]. Second, for views trees built from VOs that are not access-top, the enumeration approach from Section 4 would report the values of bound variables before the values of free variables or the values of output variables before setting the values of input variables. To resolve this issue, we instantiate a view tree iterator for each value of the variable that violates the free-dominance or input-dominance condition. We then use the union algorithm to report only distinct tuples over the output variables. By partitioning input relations, we ensure that the number of instantiated iterators depends on ϵ. For view trees built from access-top VOs, we use the enumeration approach from Section 4.

6.1 Data Partitioning

We partition relations based on the frequencies of their values. For a database $\mathcal{D}$, relation $R \in \mathcal{D}$ over schema $\mathcal{X}$, schema $\mathcal{S} \subset \mathcal{X}$, and threshold θ, the pair $(R^{\mathcal{S}\to H}, R^{\mathcal{S}\to L})$ is a *partition* of R on $\mathcal{S}$ with threshold θ if it satisfies the conditions:

$$\begin{aligned}
&\text{(union)} && R(\mathbf{x}) = R^{\mathcal{S}\to H}(\mathbf{x}) + R^{\mathcal{S}\to L}(\mathbf{x}) \text{ for } \mathbf{x} \in \mathsf{Dom}(\mathcal{X})\\
&\text{(domain partition)} && \pi_{\mathcal{S}} R^{\mathcal{S}\to H} \cap \pi_{\mathcal{S}} R^{\mathcal{S}\to L} = \emptyset\\
&\text{(heavy part)} && \forall \mathbf{t} \in \pi_{\mathcal{S}} R^{\mathcal{S}\to H},\ \exists K \in \mathcal{D}\text{: } |\sigma_{\mathcal{S}=\mathbf{t}} K| \geq \tfrac{1}{2}\theta\\
&\text{(light part)} && \forall \mathbf{t} \in \pi_{\mathcal{S}} R^{\mathcal{S}\to L} \text{ and } \forall K \in \mathcal{D}\text{: } |\sigma_{\mathcal{S}=\mathbf{t}} K| < \tfrac{3}{2}\theta
\end{aligned}$$

We call $(R^{\mathcal{S}\to H}, R^{\mathcal{S}\to L})$ a *strict* partition of R on $\mathcal{S}$ with threshold θ if it satisfies the union and domain partition conditions and the strict versions of the heavy and light part conditions:

(strict heavy part) $\forall \mathbf{t} \in \pi_{\mathcal{S}} R^{\mathcal{S}\to H}$, $\exists K \in \mathcal{D}$: $|\sigma_{\mathcal{S}=\mathbf{t}} K| \geq \theta$

(strict light part) $\forall \mathbf{t} \in \pi_{\mathcal{S}} R^{\mathcal{S}\to L}$ and $\forall K \in \mathcal{D}$: $|\sigma_{\mathcal{S}=\mathbf{t}} K| < \theta$

The relation $R^{\mathcal{S}\to H}$ is called *heavy*, and the relation $R^{\mathcal{S}\to L}$ is called *light* on the partition key $\mathcal{S}$, as they consist of all $\mathcal{S}$-tuples that are heavy and respectively light in R. Due to the domain partition, the relations $R^{\mathcal{S}\to H}$ and $R^{\mathcal{S}\to L}$ are disjoint. For $|\mathcal{D}| = N$ and a strict partition $(R^{\mathcal{S}\to H}, R^{\mathcal{S}\to L})$ of R on $\mathcal{S}$ with threshold $\theta = N^{\epsilon}$ for $\epsilon \in [0,1]$, we have: (1) $\forall \mathbf{t} \in \pi_{\mathcal{S}} R^{\mathcal{S}\to L} : |\sigma_{\mathcal{S}=\mathbf{t}} R^{\mathcal{S}\to L}| < \theta = N^{\epsilon}$; and (2) $|\pi_{\mathcal{S}} R^{\mathcal{S}\to H}| \leq \frac{N}{\theta} = N^{1-\epsilon}$. The first bound follows from the strict light part condition. In the second bound, $\pi_{\mathcal{S}} R^{\mathcal{S}\to H}$ refers to the tuples over schema $\mathcal{S}$ with high degrees in some relation in the database. The database can contain at most $\frac{N}{\theta}$ such tuples; otherwise, the database size would exceed N.

Disjoint relation parts can be further partitioned independently of each other on different partition keys. We write $R^{\mathcal{S}_1\to s_1,\ldots,\mathcal{S}_n\to s_n}$ to denote the relation part obtained after partitioning $R^{\mathcal{S}_1\to s_1,\ldots,\mathcal{S}_{n-1}\to s_{n-1}}$ on $\mathcal{S}_n$, where $s_i \in \{H, L\}$ for $i \in [n]$. The domain of $R^{\mathcal{S}_1\to s_1,\ldots,\mathcal{S}_n\to s_n}$ is the intersection of the domains of $R^{\mathcal{S}_i\to s_i}$, for $i \in [n]$. We refer to $\mathcal{S}_1 \to s_1, \ldots, \mathcal{S}_n \to s_n$ as a heavy-light signature for R. Consider for instance a relation R with schema (A, B, C). One possible partition of R consists of the relation parts $R^{A\to L}$, $R^{A\to H, AB\to L}$, and $R^{A\to H, AB\to H}$. The union of these relation parts constitutes the relation R.

6.2 Preprocessing

The preprocessing has two steps. First, we construct a set of VOs corresponding to the different evaluation strategies over the heavy and light relation parts. Second, we build a view tree from each such VO using the function τ from the general case (Figure 3).

We next describe the construction of a set of VOs from a canonical VO ω of a hierarchical CQAP $Q(\mathcal{O}|\mathcal{I})$. Without loss of generality, we assume that ω is a tree; in case ω is a forest, the reasoning below applies independently to each tree in the forest. The construction proceeds recursively on the structure of ω and forms the query $Q_X(\mathcal{O}_X|\mathcal{I}_X)$ at each variable X. The query Q_X is the join of the atoms in ω_X, the set $\mathcal{O}_X$ consists of the output variables in ω_X, and the set $\mathcal{I}_X$ consists of the input variables in ω_X and all ancestor variables along the path from X to the root of ω. The next step analyses the query Q_X.

If Q_X is in CQAP$_0$, we turn ω_X into an access-top VO for Q_X by pulling the free variables above the bound variables and the input variables above the output variables. For queries in CQAP$_0$, this restructuring does not increase their static width.

If Q_X is not in CQAP$_0$, then ω_X contains a bound variable that dominates a free variable or an output variable that dominates an input variable. If X does not violate either of these conditions, we recur on each subtree and combine the constructed VOs. Otherwise, we create two sets of VOs, which encode different evaluation strategies for different parts of the result of Q_X. Let *key* be the set of variables on the path from X to the root of the canonical VO for Q, including X. For the first set of VOs, each leaf atom $R^{sig}(\mathcal{X})$ below X is replaced by $R^{sig,key\to H}(\mathcal{X})$ before recurring on each subtree, denoting that the evaluation of Q_X is over relations parts that are heavy on *key*. For the second set of VOs, we turn ω_X into an access-top VO over relations parts that are light on *key*; this restructuring of the VO may increase its static width.

We construct a view tree for each VO formed in the previous step. For each view tree, we strict partition the input relations based on their heavy-light signature and compute the queries defining the views. We refer to this step as view tree materialisation. The

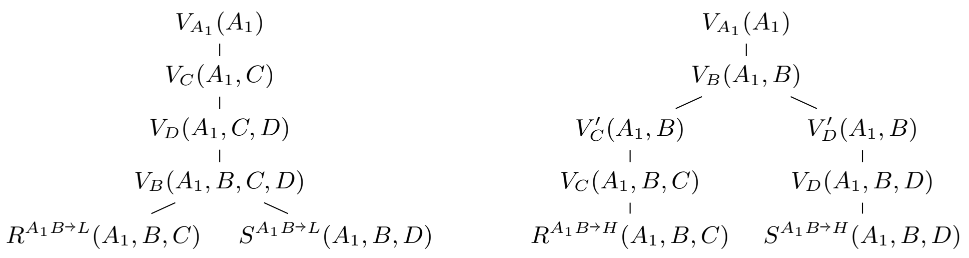


Figure 7 View trees constructed for $Q_1(D|A_1, C) = R(A_1, B, C), S(A_1, B, D)$ from Example 19 using the VOs: (left) $A_1 - C - D - B - \{R^{A_1B\to L}(A_1, B, C), S^{A_1B\to L}(A_1, B, D)\}$ and (right) $A_1 - B - \{C - R^{A_1B\to H}(A_1, B, C), D - S^{A_1B\to H}(A_1, B, D)\}$.

view trees constructed for the evaluation of queries in CQAP$_0$ or over heavy relation parts follow canonical VOs, meaning that they can be materialised in linear time. The view trees constructed for the evaluation of queries over light relation parts follow access-top VOs. Using the degree constraints in the input relations, each such view trees can be materialised in $\mathcal{O}(N^{1+(\mathsf{w}-1)\epsilon})$, where w is the static width of the query.

▶ **Example 19.** We explain the construction of the views tree for the connected component from Figure 4 (middle) corresponding to the query $Q_1(D|A_1, C) = R(A_1, B, C), S(A_1, B, D)$. In the canonical VO of this query, shown in Figure 5 (left), the bound variable B dominates the free variables C and D. We strictly partition the relations R and S on (A_1, B) with threshold N^ϵ, where N is the database size. To evaluate the join over the light relation parts, we turn the subtree in the canonical VO rooted at B into an access-top VO and construct a view tree following this new VO, see Figure 7 (left). We compute the view $V_B(A_1, B, C, D)$ in time $\mathcal{O}(N^{1+\epsilon})$: For each (a, b, c) in the light part $R^{A_1B\to L}(A_1, B, C)$ of R, we fetch the D-values in $S^{A_1B\to L}(A_1, B, D)$ that are paired with (a, b). The iteration in $R^{A_1B\to L}(A_1, B, C)$ takes $\mathcal{O}(N)$ time and for each (a, b), there are at most N^ϵ D-values in $S^{A_1B\to L}(A_1, B, D)$. The views V_D, V_C, and V_A result from V_B by marginalising out one variable at a time. Overall, this takes $\mathcal{O}(N^{1+\epsilon})$ time.

To evaluate the join over the heavy parts of R and S, we construct a view tree following the canonical VO (Figure 7 right). The VO and view tree are the same as in Figure 4, except that the leaves are the heavy parts of R and S. The view tree can be materialised in $\mathcal{O}(N)$ time, cf. Example 9. Overall, the two view trees can be computed in $\mathcal{O}(N^{1+\epsilon})$ time. ⌟

6.3 Updates

A single-tuple update to an input relation may cause changes in several view trees constructed for a given hierarchical CQAP. If the input relation is partitioned, we first identify which part of the relation is affected by the update. We then propagate the update in each view tree containing the affected relation part, as discussed in Section 4.

▶ **Example 20.** We consider the maintenance of the view trees from Figure 7 under a single-tuple update $\delta R(a, b, c)$ to R. The update affects the heavy part $R^{A_1B\to H}$ if $(a, b) \in \pi_{A_1,B} R^{A_1B\to H}$; otherwise, it affects the light part $R^{A_1B\to L}$. For the former, we propagate the update from $R^{A_1B\to H}$ to the root. For each view on this path, we compute its delta query and update the view in constant time for fixed (a, b, c). For the latter, we compute the delta $\delta V_B(a, b, c, D) = \delta R^{A_1B\to L}(a, b, c), S^{A_1B\to L}(a, b, D)$ in $\mathcal{O}(N^\epsilon)$ time because there are at most N^ϵ D-values paired with (a, b) in $S^{A_1B\to L}$. We then update $V_D(a, c, D)$ with

$\delta V_D(a, c, D) = \delta V_B(a, b, c, D)$ in $\mathcal{O}(N^\epsilon)$ time and update the views $V_C(A_1, C)$ and $V_{A_1}(A_1)$ in constant time. The case of single-tuple updates to S is analogous. Overall, maintaining the two view trees under a single-tuple update to any input relation takes $\mathcal{O}(N^\epsilon)$ time. ⌟

An update may change the degree of values over a partition key from light to heavy or vice versa. In such cases, we need to rebalance the partitioning and possibly recompute some views. Although such rebalancing steps may take time more than $\mathcal{O}(N^{\delta\epsilon})$, they happen periodically and their amortised cost remains the same as for a single-tuple update.

7 Related Work

Our work is the first to investigate the dynamic evaluation for queries with access patterns.

Free Access Patterns. Our notion of queries with free access patterns corresponds to parameterized queries [1]. These queries have selection conditions that set variables to parameter values to be supplied at query time. Prior work closest in spirit to ours investigates the space-delay trade-off for the static evaluation of full conjunctive queries with free access patterns [11]. It constructs a succinct representation of the query output, from which the tuples that conform with value bindings of the input variables can be enumerated. It does not support queries with projection nor dynamic evaluation. Follow-up work considers the static evaluation for Boolean conjunctive queries with access patterns [10]. Further works on queries with access patterns [15, 34, 12, 5, 6] consider the setting where *input* relations have input and output variables and there is no restriction on whether they are bound or free; also, a variable may be input in a relation and output in another. This poses the challenge of whether the query can be answered under specific access restrictions [28, 29, 27].

Dynamic evaluation. Our work generalises the dichotomy for q-hierarchical queries under updates [7] and the complexity trade-offs for queries under updates [19, 20, 21]. The IVM approaches Dynamic Yannakakis [18] and F-IVM [30], which is implemented on top of DBToaster [24], achieve (i) linear-time preprocessing, linear-time single-tuple updates, and constant enumeration delay for free-connex acyclic queries; and (ii) linear-time preprocessing, constant-time single-tuple updates, and constant enumeration delay for q-hierarchical queries. Theorem 8 recovers these results by noting that the static and dynamic widths are: 1 and respectively in $\{0, 1\}$ for free-connex acyclic queries and 1 and respectively 0 for q-hierarchical queries. We refer the reader to a comprehensive comparison [23] of dynamic query evaluation techniques and how they are recovered by the trade-off [21] extended in our work.

Our CQAP_0 dichotomy strictly generalises the one for q-hierarchical queries [7]: The set of q-hierarchical queries is a strict subset of CQAP_0, while there are hard patterns of non-CQAP_0 beyond those for non-q-hierarchical queries.

There are key technical differences between the prior framework for dynamic evaluation trade-off [21] and ours: different data partitioning; new modular construction of view trees; access-top variable orders; new iterators for view trees modelled on any variable order. We create a set of variable orders that represent heavy/light evaluation strategies and then map them to view trees. One advantage is a simpler complexity analysis for the views, since the variables orders and their view trees share the same width measures.

Cutset optimisations. Cutset conditioning [32] and cutset sampling [8] are used for efficient exact and approximate inference in Bayesian networks. The idea is to *choose* a cutset, which is a subset of variables, such that conditioning on the variables in the cutset, i.e., instantiating them with possible values, yields a network with a small treewidth that allows exact inference. The set of input variables of a CQAP can be seen as a *given* cutset, while fixing the input variables to given values is conditioning. Query fracturing, as introduced in our work, is a query rewriting technique that does not have a counterpart in cutset optimisations in AI.

8 Conclusion

This paper introduces a fully dynamic evaluation approach for conjunctive queries with free access patterns. It gives a syntactic characterisation of those queries that admit constant-time update and delay and further investigates the trade-off between preprocessing time, update time, and enumeration delay for such queries.

References

1 Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995. URL: `http://webdam.inria.fr/Alice/`.

2 Mahmoud Abo Khamis, Hung Q. Ngo, and Atri Rudra. FAQ: Questions Asked Frequently. In *PODS*, pages 13–28, 2016.

3 Albert Atserias, Martin Grohe, and Dániel Marx. Size Bounds and Query Plans for Relational Joins. *SIAM J. Comput.*, 42(4):1737–1767, 2013.

4 Catriel Beeri, Ronald Fagin, David Maier, and Mihalis Yannakakis. On the Desirability of Acyclic Database Schemes. *J. ACM*, 30(3):479–513, 1983.

5 Michael Benedikt, Julien Leblay, and Efthymia Tsamoura. Querying with Access Patterns and Integrity Constraints. *VLDB*, 8(6):690–701, 2015.

6 Michael Benedikt, Balder Ten Cate, and Efthymia Tsamoura. Generating Low-cost Plans from Proofs. In *PODS*, pages 200–211, 2014.

7 Christoph Berkholz, Jens Keppeler, and Nicole Schweikardt. Answering Conjunctive Queries Under Updates. In *PODS*, pages 303–318, 2017.

8 Bozhena Bidyuk and Rina Dechter. Cutset Sampling for Bayesian Networks. *J. Artif. Intell. Res.*, 28:1–48, 2007.

9 Rada Chirkova and Jun Yang. Materialized Views. *Found. & Trends DB*, 4(4):295–405, 2012.

10 Shaleen Deep, Xiao Hu, and Paraschos Koutris. Space-Time Tradeoffs for Answering Boolean Conjunctive Queries. *CoRR*, abs/2109.10889, 2021. `arXiv:2109.10889`.

11 Shaleen Deep and Paraschos Koutris. Compressed Representations of Conjunctive Query Results. In *PODS*, pages 307–322, 2018.

12 Alin Deutsch, Bertram Ludäscher, and Alan Nash. Rewriting Queries using Views with Access Patterns under Integrity Constraints. *Theor. Comput. Sci.*, 371(3):200–226, 2007.

13 Arnaud Durand and Etienne Grandjean. First-order Queries on Structures of Bounded Degree are Computable with Constant Delay. *TOCL*, 8(4):21, 2007.

14 Arnaud Durand and Yann Strozecki. Enumeration Complexity of Logical Query Problems with Second-order Variables. In *CSL*, pages 189–202, 2011.

15 Daniela Florescu, Alon Levy, Ioana Manolescu, and Dan Suciu. Query Optimization in the Presence of Limited Access Patterns. *SIGMOD Rec.*, 28(2):311–322, 1999.

16 Georg Gottlob, Nicola Leone, and Francesco Scarcello. Hypertree Decompositions and Tractable Queries. In *PODS*, pages 21–32, 1999.

17 Monika Henzinger, Sebastian Krinninger, Danupon Nanongkai, and Thatchaphol Saranurak. Unifying and Strengthening Hardness for Dynamic Problems via the Online Matrix-Vector Multiplication Conjecture. In *STOC*, pages 21–30, 2015.

18 Muhammad Idris, Martín Ugarte, and Stijn Vansummeren. The Dynamic Yannakakis Algorithm: Compact and Efficient Query Processing Under Updates. In *SIGMOD*, pages 1259–1274, 2017.

19 Ahmet Kara, Hung Q. Ngo, Milos Nikolic, Dan Olteanu, and Haozhe Zhang. Counting Triangles under Updates in Worst-Case Optimal Time. In *ICDT*, pages 4:1–4:18, 2019.

20 Ahmet Kara, Hung Q. Ngo, Milos Nikolic, Dan Olteanu, and Haozhe Zhang. Maintaining Triangle Queries under Updates. *TODS*, 45(3):11:1–11:46, 2020.

21 Ahmet Kara, Milos Nikolic, Dan Olteanu, and Haozhe Zhang. Trade-offs in Static and Dynamic Evaluation of Hierarchical Queries. In *PODS*, pages 375–392, 2020.

22 Ahmet Kara, Milos Nikolic, Dan Olteanu, and Haozhe Zhang. Conjunctive Queries with Free Access Patterns under Updates. *CoRR*, abs/2206.09032, 2022. `arXiv:2206.09032`.

23 Ahmet Kara, Milos Nikolic, Dan Olteanu, and Haozhe Zhang. Trade-offs in Static and Dynamic Evaluation of Hierarchical Queries. *To appear in LMCS*, 2023.

24 Christoph Koch et al. DBToaster: Higher-order Delta Processing for Dynamic, Frequently Fresh Views. *VLDB J.*, 23(2):253–278, 2014.

25 Daphne Koller and Nir Friedman. *Probabilistic Graphical Models - Principles and Techniques*. MIT Press, 2009.

26 Tsvi Kopelowitz, Seth Pettie, and Ely Porat. Dynamic Set Intersection. In *WADS*, pages 470–481, 2015.

27 Chen Li and Edward Chang. On Answering Queries in the Presence of Limited Access Patterns. In *ICDT*, pages 219–233, 2001.

28 Alan Nash and Bertram Ludäscher. Processing First-Order Queries under Limited Access Patterns. In *PODS*, pages 307–318, 2004.

29 Alan Nash and Bertram Ludäscher. Processing Unions of Conjunctive Queries with Negation under Limited Access Patterns. In *EDBT*, pages 422–440, 2004.

30 Milos Nikolic and Dan Olteanu. Incremental View Maintenance with Triple Lock Factorization Benefits. In *SIGMOD*, pages 365–380, 2018.

31 Dan Olteanu and Jakub Závodný. Size Bounds for Factorised Representations of Query Results. *TODS*, 40(1):2:1–2:44, 2015.

32 Judea Pearl. *Probabilistic Reasoning in Intelligent Systems - Networks of Plausible Inference*. Morgan Kaufmann series in representation and reasoning. Morgan Kaufmann, 1989.

33 Todd L. Veldhuizen. Triejoin: A Simple, Worst-Case Optimal Join Algorithm. In *ICDT*, pages 96–106, 2014.

34 Ramana Yerneni, Chen Li, Jeffrey Ullman, and Hector Garcia-Molina. Optimizing Large Join Queries in Mediation Systems. In *ICDT*, pages 348–364, 1999.

Finite-Cliquewidth Sets of Existential Rules

Toward a General Criterion for Decidable yet Highly Expressive Querying

Thomas Feller
Technische Universität Dresden, Germany

Tim S. Lyon
Technische Universität Dresden, Germany

Piotr Ostropolski-Nalewaja
Technische Universität Dresden, Germany
University of Wrocław, Poland

Sebastian Rudolph
Technische Universität Dresden, Germany

Abstract

In our pursuit of generic criteria for decidable ontology-based querying, we introduce *finite-cliquewidth sets* (**fcs**) of existential rules, a model-theoretically defined class of rule sets, inspired by the *cliquewidth* measure from graph theory. By a generic argument, we show that **fcs** ensures decidability of entailment for a sizable class of queries (dubbed *DaMSOQs*) subsuming conjunctive queries (CQs). The **fcs** class properly generalizes the class of finite-expansion sets (**fes**), and for signatures of arity ≤ 2, the class of bounded-treewidth sets (**bts**). For higher arities, **bts** is only indirectly subsumed by **fcs** by means of reification. Despite the generality of **fcs**, we provide a rule set with decidable CQ entailment (by virtue of first-order-rewritability) that falls outside **fcs**, thus demonstrating the incomparability of **fcs** and the class of finite-unification sets (**fus**). In spite of this, we show that if we restrict ourselves to single-headed rule sets over signatures of arity ≤ 2, then **fcs** subsumes **fus**.

2012 ACM Subject Classification Theory of computation → Description logics; Theory of computation → Database query languages (principles); Theory of computation → Logic and databases; Mathematics of computing → Graph theory

Keywords and phrases existential rules, TGDs, cliquewidth, treewidth, bounded-treewidth sets, finite-unification sets, first-order rewritability, monadic second-order logic, datalog

Digital Object Identifier 10.4230/LIPIcs.ICDT.2023.18

Related Version *Full Version*: `https://arxiv.org/abs/2209.02464` [15]

Funding Work supported by the European Research Council (ERC) Consolidator Grant 771779 *A Grand Unified Theory of Decidability in Logic-Based Knowledge Representation* (DeciGUT).

1 Introduction

The problem of querying under *existential rules*[1] (henceforth often shortened to *rules*) is a popular topic in the research fields of database theory and knowledge representation. For arbitrary rule sets, query entailment is undecidable [9], motivating research into expressive fragments for which decidability can be regained.

[1] Existential rules are also referred to as *tuple-generating dependencies* (TGDs) [1], *conceptual graph rules* [31], Datalog$^{\pm}$ [20], and $\forall\exists$*-rules* [3] in the literature.

A theoretical tool which has not only proven useful for describing querying methods, but also for identifying classes of rule sets with decidable query entailment, is the *chase* [4]. Given a database $\mathcal{D}$ and a rule set $\mathcal{R}$, the (potentially non-terminating) chase procedure yields a so-called *universal model* [14]. Universal models satisfy exactly those queries entailed by $\mathcal{D}$ and $\mathcal{R}$ and thus allow for the reduction of query entailment to query evaluation.

Rule sets admitting finite universal models (a property equivalent to being a *finite-expansion set* [3] or *core-chase terminating* [14]) are particularly well-behaved. But even in cases where universal models are necessarily infinite, they may still be sufficiently "tame" to allow for decidable query entailment, as is the case with the class of *bounded-treewidth sets* (**bts**) [3]. A rule set $\mathcal{R}$ qualifies as **bts** *iff* for every database $\mathcal{D}$, there exists a universal model of $\mathcal{D}$ and $\mathcal{R}$ whose *treewidth* (a structural measure originating from graph theory) is bounded by some $n \in \mathbb{N}$. The **bts** class subsumes class of finite expansion sets (**fes**), and gives rise to decidable query entailment for a sizable family of concrete, syntactically defined classes of rule sets deploying more or less refined versions of guardedness [3, 7, 23].

While **bts** is a fairly general class, it still fails to contain rather simple rule sets; e.g., the rule set $\mathcal{R}^{\infty}_{\text{tran}}$, consisting of the two rules

$$\mathtt{E}(x,y) \to \exists z \mathtt{E}(y,z) \quad \text{and} \quad \mathtt{E}(x,y) \wedge \mathtt{E}(y,z) \to \mathtt{E}(x,z),$$

falls outside the **bts** class. To give an idea as to why this holds, consider any database $\mathcal{D}$ containing an E-fact (e.g., $\mathtt{E}(\mathtt{a},\mathtt{b})$): the chase will yield a universal model resembling the transitive closure of an infinite E-path, a clique-like structure of infinite treewidth. What is more, not only does $\mathcal{R}^{\infty}_{\text{tran}}$ fail to be **bts**, it also fails to fall under the other generic decidability criteria: it is neither **fus** (described below) nor *finitely controllable* (which would guarantee the existence of a finite "countermodel" for each non-entailed query). In spite of this, results for description logics confirm the decidability of (conjunctive) query entailment over $\mathcal{R}^{\infty}_{\text{tran}}$ (see [18, 24]), incentivizing a generalization of the decidability criteria mentioned above.

On a separate, but related note: the **bts** class is incomparable with the class of *finite-unification sets* (**fus**) [3], another class giving rise to decidable query entailment thanks to *first-order rewritability.* Such rule sets may feature non-guarded rules [8]; for instance, they may include *concept products* [6, 30], which create a biclique linking instances of two unary predicates, as in

$$\mathtt{Elephant}(x) \wedge \mathtt{Mouse}(y) \to \mathtt{BiggerThan}(x,y).$$

The above examples demonstrate a crucial weakness of the **bts** class, namely, its inability to tolerate universal models exhibiting "harmless" unbounded clique-like structures. Opportunely, the graph-theoretic notion of *cliquewidth* [13] overcomes this problem, while retaining most of the desirable properties associated with the notion of treewidth. Inspired by this less mainstream, but more powerful concept, we set out to introduce *finite-cliquewidth sets* (**fcs**) of rules. As the original cliquewidth notion is tailored to finite undirected graphs, some (not entirely straightforward) generalizations are necessary to adapt it to countable instances, while at the same time, preserving its advantageous properties.

Our contributions can be summarized as follows:

- We introduce an abstract framework showing how to utilize specific types of model-theoretic measures to establish the decidability of query entailment for a comprehensive class of queries (dubbed *datalog / monadic second-order queries*, or *DaMSOQs* for short) that significantly extends the class of conjunctive queries.

- Generalizing the eponymous notion from finite graph theory, we introduce the model-theoretic measure of *cliquewidth* for countable instances over arbitrary signatures. Based on our framework, we show that the derived notion of *finite-cliquewidth sets* guarantees decidability of DaMSOQ entailment. In particular, we demonstrate that $\mathcal{R}^{\infty}_{\text{tran}}$ is indeed **fcs**, thus showing that **fcs** incorporates rule sets outside **bts** and **fus**.
- We compare the **fcs** and **bts** classes, obtaining: (good news) for binary signatures, **fcs** subsumes **bts**, which (bad news) does not hold for higher-arity signatures, but (relieving news) **fcs** still "indirectly subsumes" higher-arity **bts** through reification.
- We compare the **fcs** and **fus** classes, obtaining: (good news) for sets of single-headed rules over signatures of arity ≤ 2, **fcs** subsumes **fus**, which (bad news) does not hold for multi-headed rules, but (relieving news) this could not be any different as there are **fus** rule sets for which DaMSOQ entailment is undecidable.

For space reasons, we defer technical details and most proofs to the extended version [15].

2 Preliminaries

Syntax and formulae. We let $\mathfrak{T}$ denote a set of *terms*, defined as the union of three countably infinite, mutually disjoint sets of *constants* $\mathfrak{C}$, *nulls* $\mathfrak{N}$, and *variables* $\mathfrak{V}$. We use $\mathtt{a}$, $\mathtt{b}$, $\mathtt{c}$, ... (occasionally annotated) to denote constants, and use x, y, z, ... (occasionally annotated) to denote both nulls and variables. A *signature* Σ is a finite set of *predicate symbols* (called *predicates*), which are capitalized ($\mathtt{E}$, $\mathtt{R}$, ...). Throughout the paper, we assume a fixed signature Σ, unless stated otherwise. For each predicate $\mathtt{R} \in \Sigma$, we denote its *arity* with $\mathsf{ar}(\mathtt{R})$. We say Σ is *binary* if it contains only predicates of arity ≤ 2. We assume Σ to contain a special, "universal" unary predicate $\top$, assumed to hold of every term.[2] An *atom* over Σ is an expression of the form $\mathtt{R}(\vec{t})$, where $\vec{t}$ is an $\mathsf{ar}(\mathtt{R})$-tuple of terms. If $\vec{t}$ consists only of constants, then $\mathtt{R}(\vec{t})$ is a *ground atom*. An *instance* $\mathcal{I}$ over Σ is a (possibly countably infinite) set of atoms over constants and nulls, whereas a *database* $\mathcal{D}$ is a finite set of ground atoms. The *active domain* $\mathsf{dom}(\mathcal{X})$ of a set of atoms $\mathcal{X}$ is the set of terms appearing in the atoms of $\mathcal{X}$. Moreover, as instances over binary signatures can be viewed as directed edge-labelled graphs, we often use graph-theoretic terminology when discussing such objects.

Homomorphisms. Given sets $\mathcal{X}$, $\mathcal{Y}$ of atoms, a *homomorphism* from $\mathcal{X}$ to $\mathcal{Y}$ is a mapping $h : \mathsf{dom}(\mathcal{X}) \to \mathsf{dom}(\mathcal{Y})$ that satisfies: (i) $\mathtt{R}(h(\vec{t})) \in \mathcal{Y}$, for all $\mathtt{R}(\vec{t}) \in \mathcal{X}$, and (ii) $h(\mathtt{a}) = \mathtt{a}$, for each $\mathtt{a} \in \mathfrak{C}$. $\mathcal{X}$ and $\mathcal{Y}$ are *homomorphically equivalent* (written $\mathcal{X} \equiv \mathcal{Y}$) *iff* homomorphisms exist from $\mathcal{X}$ to $\mathcal{Y}$ and from $\mathcal{Y}$ to $\mathcal{X}$. A homomorphism h is an *isomorphism iff* it is bijective and h^{-1} is also a homomorphism. An instance $\mathcal{I}'$ is an *induced sub-instance* of an instance $\mathcal{I}$ *iff* (i) $\mathcal{I}' \subseteq \mathcal{I}$ and (ii) if $\mathtt{R}(\vec{t}) \in \mathcal{I}$ and $\vec{t} \subseteq \mathsf{dom}(\mathcal{I}')$, then $\mathtt{R}(\vec{t}) \in \mathcal{I}'$.

Existential rules. An *(existential) rule* ρ is a first-order sentence $\forall \vec{x}\vec{y}\, \phi(\vec{x}, \vec{y}) \to \exists \vec{z}\, \psi(\vec{y}, \vec{z})$, where $\vec{x}$, $\vec{y}$, and $\vec{z}$ are mutually disjoint tuples of variables, and both the *body* $\phi(\vec{x}, \vec{y})$ and the *head* $\psi(\vec{y}, \vec{z})$ of ρ (denoted with $\mathsf{body}(\rho)$ and $\mathsf{head}(\rho)$, respectively) are conjunctions (possibly empty, sometimes seen as sets) of atoms over the indicated variables. The *frontier* $fr(\rho)$ of ρ is the set of variables $\vec{y}$ shared between the body and the head. We often omit the universal quantifiers prefixing existential rules. A rule ρ is (i) *n-ary iff* all predicates appearing in ρ

[2] Assuming the presence of such a built-in *domain predicate* $\top$ does not affect our results, but it allows for a simpler and more concise presentation.

are of arity at most n, (ii) *single-headed iff* $\mathsf{head}(\rho)$ contains a single atom, (iii) *datalog iff* ρ does not contain an existential quantifier (otherwise *non-datalog*). We call a finite set of existential rules $\mathcal{R}$ a *rule set.* Satisfaction of a rule ρ (a rule set $\mathcal{R}$) by an instance $\mathcal{I}$ is defined as usual and is written $\mathcal{I} \models \rho$ ($\mathcal{I} \models \mathcal{R}$, respectively). Given a database $\mathcal{D}$ and a rule set $\mathcal{R}$, we define the pair $(\mathcal{D}, \mathcal{R})$ to be a *knowledge base*, and define an instance $\mathcal{I}$ to be a *model* of $(\mathcal{D}, \mathcal{R})$, written $\mathcal{I} \models (\mathcal{D}, \mathcal{R})$, *iff* $\mathcal{D} \subseteq \mathcal{I}$ and $\mathcal{I} \models \mathcal{R}$. A model $\mathcal{I}$ of $(\mathcal{D}, \mathcal{R})$ is called *universal iff* there is a homomorphism from $\mathcal{I}$ into every model of $(\mathcal{D}, \mathcal{R})$.

Rule application and Skolem chase. A rule $\rho = \phi(\vec{x}, \vec{y}) \to \exists \vec{z} \psi(\vec{y}, \vec{z})$ is *applicable* to an instance $\mathcal{I}$ *iff* there is a homomorphism h from $\phi(\vec{x}, \vec{y})$ to $\mathcal{I}$. We then call (ρ, h) a *trigger* of $\mathcal{I}$. The *application* of a trigger (ρ, h) in $\mathcal{I}$ yields the instance $\mathbf{Ch}(\mathcal{I}, \rho, h) = \mathcal{I} \cup \bar{h}(\psi(\vec{y}, \vec{z}))$, where $\bar{h}$ extends h, mapping each variable z from $\vec{z}$ to a null denoted $z_{\rho, h(\vec{y})}$. Note that this entails $\mathbf{Ch}(\mathbf{Ch}(\mathcal{I}, \rho, h), \rho, h') = \mathbf{Ch}(\mathcal{I}, \rho, h)$ whenever $h(\vec{y}) = h'(\vec{y})$. Moreover, applications of different rules or the same rule with different frontier-mappings are independent, so their order is irrelevant. Hence we can define the parallel one-step application of all applicable rules as

$$\mathbf{Ch}_1(\mathcal{I}, \mathcal{R}) = \bigcup_{\rho \in \mathcal{R},\ (\rho, h) \text{ trigger of } \mathcal{I}} \mathbf{Ch}(\mathcal{I}, \rho, h).$$

Then, we define the *(breadth-first) Skolem chase sequence* by letting $\mathbf{Ch}_0(\mathcal{I}, \mathcal{R}) = \mathcal{I}$ and $\mathbf{Ch}_{i+1}(\mathcal{I}, \mathcal{R}) = \mathbf{Ch}_1(\mathbf{Ch}_i(\mathcal{I}, \mathcal{R}), \mathcal{R})$, ultimately obtaining the *Skolem chase* $\mathbf{Ch}_\infty(\mathcal{I}, \mathcal{R}) = \bigcup_{i \in \mathbb{N}} \mathbf{Ch}_i(\mathcal{I}, \mathcal{R})$. We note that the Skolem chase of a countable instance is countable, as is the number of overall rule applications performed to obtain $\mathbf{Ch}_\infty(\mathcal{I}, \mathcal{R})$.

(Unions of) conjunctive queries and their entailment. A *conjunctive query* (CQ) is a formula $q(\vec{y}) = \exists \vec{x}\, \phi(\vec{x}, \vec{y})$ with $\phi(\vec{x}, \vec{y})$ a conjunction (sometimes written as a set) of atoms over the variables from $\vec{x}, \vec{y}$ and constants. The variables from $\vec{y}$ are called *free.* A *Boolean* CQ (or BCQ) is a CQ with no free variables. A *union of conjunctive queries* (UCQ) $\psi(\vec{y})$ is a disjunction of CQs with free variables $\vec{y}$. We will treat UCQs as sets of CQs. A BCQ $q = \exists \vec{x}\, \phi(\vec{x})$ is satisfied in an instance $\mathcal{I}$ if there exists a homomorphism from $\phi(\vec{x})$ to $\mathcal{I}$. $\mathcal{I}$ satisfies a union of BCQs if it satisfies at least one of its disjuncts. An instance $\mathcal{I}$ and rule set $\mathcal{R}$ *entail* a BCQ $q = \exists \vec{x}\, \phi(\vec{x})$, written $(\mathcal{I}, \mathcal{R}) \models q$ *iff* $\phi(\vec{x})$ maps homomorphically into every model of $\mathcal{I}$ and $\mathcal{R}$. This coincides with the existence of a homomorphism from $\phi(\vec{x})$ into any universal model of $\mathcal{I}$ and $\mathcal{R}$ (e.g., the Skolem chase $\mathbf{Ch}_\infty(\mathcal{I}, \mathcal{R})$).

Rewritings and finite-unification sets. Given a rule set $\mathcal{R}$ and a CQ $q(\vec{y})$, we say that a UCQ $\psi(\vec{y})$ is a *rewriting* of $q(\vec{y})$ under the rule set $\mathcal{R}$ *iff* for any database $\mathcal{D}$ and any tuple of its elements $\vec{\mathsf{a}}$ the following holds: $\mathbf{Ch}_\infty(\mathcal{D}, \mathcal{R}) \models q(\vec{\mathsf{a}})$ *iff* $\mathcal{D} \models \psi(\vec{\mathsf{a}})$. A rule set $\mathcal{R}$ is a *finite-unification set* (**fus**) *iff* for every CQ, there exists a UCQ rewriting [3]. This property is also referred to as *first-order rewritability.* If a rule set $\mathcal{R}$ is **fus**, then for any given CQ $q(\vec{y})$, we fix one of its rewritings under $\mathcal{R}$ and denote it with $\mathsf{rew}_{\mathcal{R}}(q(\vec{y}))$.

Treewidth and bounded-treewidth sets. Let $\mathcal{I}$ be an instance. A *tree decomposition* of $\mathcal{I}$ is a (potentially infinite) tree $T = (V, E)$, where:

- $V \subseteq 2^{\mathsf{dom}(\mathcal{I})}$, that is, each node $X \in V$ is a set of terms of $\mathcal{I}$, and $\bigcup_{X \in V} X = \mathsf{dom}(\mathcal{I})$,
- for each $\mathtt{R}(t_1, \dots, t_n) \in \mathcal{I}$, there is an $X \in V$ with $\{t_1, \dots, t_n\} \subseteq X$,
- for each term t in $\mathcal{I}$, the subgraph of T induced by the nodes X with $t \in X$ is connected.

The *width* of a tree decomposition is set to be the maximum over the sizes of all its nodes minus 1, if such a maximum exists; otherwise, it is set to ∞. Last, we define the *treewidth* of an instance $\mathcal{I}$ to be the minimal width among all of its tree decompositions, and denote the treewidth of $\mathcal{I}$ as $\mathsf{tw}(\mathcal{I})$. A set of rules $\mathcal{R}$ is a *bounded-treewidth set* (**bts**) *iff* for any database $\mathcal{D}$, there is a universal model for $(\mathcal{D}, \mathcal{R})$ of finite treewidth.[3]

3 A Generic Decidability Argument

In this section, we provide an abstract framework for establishing decidability of entailment for a wide range of queries, based on certain model-theoretic criteria being met by the considered rule set. We first recall the classical notion of a (Boolean) datalog query, and after, we specify the class of queries considered in our framework.

▶ **Definition 1.** *Given a signature* Σ*, a* (Boolean) datalog query $\mathtt{q}$ *over* Σ *is represented by a finite set* $\mathcal{R}_{\mathtt{q}}$ *of datalog rules with predicates from* $\Sigma_{\mathrm{EDB}} \uplus \Sigma_{\mathrm{IDB}}$ *where* $\Sigma_{\mathrm{EDB}} \subseteq \Sigma$ *and* $\Sigma_{\mathrm{IDB}} \cap \Sigma = \emptyset$ *such that (i)* Σ_{EDB}*-atoms do not occur in rule heads of* $\mathcal{R}_{\mathtt{q}}$*, and (ii)* Σ_{IDB} *contains a distinguished nullary predicate* $\mathtt{Goal}$*. Given an instance* $\mathcal{I}$ *and a datalog query* $\mathtt{q}$*, we say that* $\mathtt{q}$ holds in $\mathcal{I}$ *(*$\mathcal{I}$ satisfies $\mathtt{q}$*), written* $\mathcal{I} \models \mathtt{q}$*, iff* $\mathtt{Goal} \in \mathbf{Ch}_{\infty}(\mathcal{I}, \mathcal{R}_{\mathtt{q}})$*. Query entailment is defined via satisfaction as usual.* ⌟

Datalog queries can be equivalently expressed as sentences in second-order logic (with the Σ_{IDB} predicates quantified over) or in least fixed-point logic (LFP). For our purposes, the given formulation is the most convenient; e.g., it makes clear that the second-order entailment problem $(\mathcal{D}, \mathcal{R}) \models \mathtt{q}$ reduces to the first-order entailment problem $(\mathcal{D}, \mathcal{R} \cup \mathcal{R}_{\mathtt{q}}) \models \mathtt{Goal}$.

▶ **Definition 2.** *A* datalog/MSO query (DaMSOQ) *over a signature* Σ *is a pair* $(\mathtt{q}, \Xi)$*, where* $\mathtt{q}$ *is a datalog query over* Σ *and* Ξ *is a monadic second-order (MSO)*[4] *sentence equivalent to* $\mathtt{q}$*. Satisfaction and entailment of DaMSOQs is defined via any of their constituents.* ⌟

Consequently, DaMSOQs are the (semantic) intersection of datalog and MSO queries. While representing DaMSOQs as a pair $(\mathtt{q}, \Xi)$ is logically redundant, it is purposeful and necessary: the below decision procedure requires both constituents as input and it is not generally possible to compute one from the other. Arguably, the most comprehensive, known DaMSOQ fragment that is well-investigated and has a syntactic definition is that of *nested monadically defined queries*, a very expressive yet computationally manageable formalism [29], subsuming (unions of) Boolean CQs, *monadic datalog queries* [10], *conjunctive 2-way regular path queries* [16], and nested versions thereof (e.g. *regular queries* [27] and others [5]).

▶ **Definition 3.** *Let* Σ *be a finite signature. A* width measure *(for* Σ*) is a function* w *mapping every countable instance over* Σ *to a value from* $\mathbb{N} \cup \{\infty\}$*. We call* w MSO-friendly *iff there exists an algorithm that, taking a number* $n \in \mathbb{N}$ *and an MSO sentence* Ξ *as input,*

- *never terminates if* Ξ *is unsatisfiable, and*
- *always terminates if* Ξ *has a model* $\mathcal{I}$ *with* $\mathsf{w}(\mathcal{I}) \leq n$*.* ⌟

[3] The term "*finite*-treewidth set" would be more fitting and in line with our terminology, but we stick to the established name. Also, the **bts** notion is not used entirely consistently in the literature; it sometimes refers to structural properties of a specific type of chase. The "semantic **bts**" notion adopted here subsumes all the others.

[4] For an introduction to monadic second-order logic, see [13, Section 1.3].

As an unsophisticated example, note that the *expansion* function $\mathsf{expansion} : \mathcal{I} \mapsto |\mathsf{dom}(\mathcal{I})|$, mapping every countable instance to the size of its domain, is an MSO-friendly width measure: there are up to isomorphism only finitely many instances with n elements, which can be computed and checked. As a less trivial example, the notion of treewidth has also been reported to fall into this category [3].

▶ **Definition 4.** *Let* w *be a width measure. A rule set* $\mathcal{R}$ *is called a* finite-w set *iff for every database* $\mathcal{D}$*, there exists a universal model* $\mathcal{I}^*$ *of* $(\mathcal{D}, \mathcal{R})$ *satisfying* $\mathsf{w}(\mathcal{I}^*) \in \mathbb{N}$. ⌟

Note that the finite width required by this definition does not need to be uniformly bounded: it may depend on the database and thus grow beyond any finite bound. This is already the case when using the $\mathsf{expansion}$ measure from above, giving rise to the class of *finite-expansion sets* (**fes**), coinciding with the notion of *core-chase terminating* rule sets [14].

▶ **Theorem 5.** *Let* w *be an MSO-friendly width measure and let* $\mathcal{R}$ *be a* finite-w set*. Then, the entailment problem* $(\mathcal{D}, \mathcal{R}) \models (\mathfrak{q}, \Xi)$ *for any database* $\mathcal{D}$ *and DaMSOQ* $(\mathfrak{q}, \Xi)$ *is decidable.*

Proof. We prove decidability by providing two semi-decision procedures: one terminating whenever $(\mathcal{D}, \mathcal{R}) \models (\mathfrak{q}, \Xi)$, the other terminating whenever $(\mathcal{D}, \mathcal{R}) \not\models (\mathfrak{q}, \Xi)$. Then, these two procedures, run in parallel, constitute a decision procedure.

- Detecting $(\mathcal{D}, \mathcal{R}) \models (\mathfrak{q}, \Xi)$. We note that $(\mathcal{D}, \mathcal{R}) \models (\mathfrak{q}, \Xi)$ *iff* $(\mathcal{D}, \mathcal{R}) \models \mathfrak{q}$ *iff* $(\mathcal{D}, \mathcal{R} \cup \mathcal{R}_\mathfrak{q}) \models$ `Goal`. The latter is a first-order entailment problem. Thanks to the completeness of first-order logic [19], we can recursively enumerate all the consequences of $(\mathcal{D}, \mathcal{R} \cup \mathcal{R}_\mathfrak{q})$ and terminate as soon as we find `Goal` among those, witnessing entailment of the query.
- Detecting $(\mathcal{D}, \mathcal{R}) \not\models (\mathfrak{q}, \Xi)$. In that case, there must exist some model $\mathcal{I}$ of $(\mathcal{D}, \mathcal{R})$ with $\mathcal{I} \not\models (\mathfrak{q}, \Xi)$. Note that such "countermodels" can be characterized by the MSO formula $\bigwedge\mathcal{D} \wedge \bigwedge\mathcal{R} \wedge \neg\Xi$. Moreover, any universal model $\mathcal{I}^*$ of $(\mathcal{D}, \mathcal{R})$ must satisfy $\mathcal{I}^* \not\models (\mathfrak{q}, \Xi)$, which can be shown (by contradiction) as follows: Let $\mathcal{I}^*$ be a universal model of $(\mathcal{D}, \mathcal{R})$ and suppose $\mathcal{I}^* \models (\mathfrak{q}, \Xi)$, i.e., $\mathcal{I}^* \models \mathfrak{q}$. Since satisfaction of datalog queries is preserved under homomorphisms and $\mathcal{I}^*$ is universal, we know there exists a homomorphism from $\mathcal{I}^*$ to $\mathcal{I}$, implying $\mathcal{I} \models \mathfrak{q}$, and thus $\mathcal{I} \models (\mathfrak{q}, \Xi)$, contradicting our assumption. As $\mathcal{R}$ is a finite-w set, there exists a universal model $\mathcal{I}^*$ of $(\mathcal{D}, \mathcal{R})$ for which $\mathsf{w}(\mathcal{I}^*)$ is finite. Hence, the following procedure will terminate, witnessing non-entailment: enumerate all natural numbers in their natural order and for each number n initiate a parallel thread with the algorithm from Definition 3 with input n and $\bigwedge\mathcal{D} \wedge \bigwedge\mathcal{R} \wedge \neg\Xi$ (the algorithm is guaranteed to exist due to MSO-friendliness of w). Terminate as soon as one thread does. ◀

4 Cliquewidth and its Properties

In this section, we will propose a width measure, for which we use the term *cliquewidth*. Our definition of this measure works for arbitrary countable instances and thus properly generalizes Courcelle's earlier eponymous notions for finite directed edge-labelled graphs [13] and countable unlabelled undirected graphs [12], as well as Grohe and Turán's cliquewidth notion for finite instances of arbitrary arity [21].

4.1 Cliquewidth of Countable Instances

Intuitively, the notion of cliquewidth is based on the idea of assembling the considered structure (e.g., instance or graph) from its singleton elements (e.g., terms or nodes). To better distinguish these elements during assembly, each may be assigned an initial color (from some finite set $\mathbb{L}$). The "assembly process" consists of successively applying the following operations to previously assembled node-colored structures:

- take the disjoint union of two structures ($\oplus$),
- uniformly assign the color k' to all hitherto k-colored elements ($\mathsf{Recolor}_{k\to k'}$),
- given a predicate $\mathtt{R}$ and a color sequence $\vec{k}$ of length $\mathsf{ar}(\mathtt{R})$, add $\mathtt{R}$-atoms for all tuples of appropriately colored elements ($\mathsf{Add}_{\mathtt{R},\vec{k}}$).

Then, the cliquewidth of a structure is the minimal number of colors needed to assemble it through successive applications of the above operations. For finite structures (e.g., graphs and instances), this is a straightforward, conceivable notion. In order to generalize it to the countably infinite case, one has to find a way to describe infinite assembly processes. In the finite case, an "assembly plan" can be described by an algebraic expression using the above operators, which in turn can be represented by its corresponding "syntax tree." The more elusive idea of an "infinite assembly plan" is then implemented by allowing for infinite, "unfounded" syntax trees. We formalize this idea of "assembly-plan-encoding syntax trees" by representing them as countably infinite instances of a very particular shape.

▶ **Definition 6.** *We define the* infinite binary tree *to be the instance*

$$\mathcal{T}_{\mathrm{bin}} = \{\mathtt{Succ}_0(s, s0) \mid s \in \{0,1\}^*\} \cup \{\mathtt{Succ}_1(s, s1) \mid s \in \{0,1\}^*\}$$

with binary predicates $\mathtt{Succ}_0$ and $\mathtt{Succ}_1$. That is, the nulls of $\mathcal{T}_{\mathrm{bin}}$ are denoted by finite sequences of 0 *and* 1*. The* root *of $\mathcal{T}_{\mathrm{bin}}$ is the null identified by the empty sequence, denoted ε.*

Given a finite set $\mathbb{L}$ of colors*, a finite set* Cnst $\subseteq \mathfrak{C}$ *of constants, and a finite signature Σ, the set* $\mathrm{Dec}(\mathbb{L}, \mathrm{Cnst}, \Sigma)$ *of* decorators *consists of the following unary predicate symbols:*

- $\mathtt{c}_k$ *for any* $\mathtt{c} \in \mathrm{Cnst} \cup \{*\}$ *and* $k \in \mathbb{L}$,
- $\mathtt{Add}_{\mathtt{R},\vec{k}}$ *for any* $\mathtt{R} \in \Sigma$ *and* $\vec{k} \in \mathbb{L}^{\mathsf{ar}(\mathtt{R})}$,
- $\mathtt{Recolor}_{k\to k'}$ *for* $k, k' \in \mathbb{L}$,
- $\oplus$, *and* $\mathtt{Void}$.

A $(\mathbb{L}, \mathrm{Cnst}, \Sigma)$-decorated infinite binary tree *(or,* decorated tree *for short) is $\mathcal{T}_{\mathrm{bin}}$ extended with facts over* $\mathrm{Dec}(\mathbb{L}, \mathrm{Cnst}, \Sigma)$ *that only use nulls from the original domain of $\mathcal{T}_{\mathrm{bin}}$, i.e., from* $\{0,1\}^*$*. A decorated tree $\mathcal{T}$ is called a* well-decorated tree *iff*

- *for every null* $s \in \{0,1\}^*$, *$\mathcal{T}$ contains exactly one fact* $\mathrm{Dec}(s)$ *with* $\mathrm{Dec} \in \mathrm{Dec}(\mathbb{L}, \mathrm{Cnst}, \Sigma)$,
- *for every* $\mathtt{c} \in \mathrm{Cnst}$, *$\mathcal{T}$ contains at most one fact of the form* $\mathtt{c}_k(s)$,
- *if* $\mathtt{Add}_{\mathtt{R},\vec{k}}(s) \in \mathcal{T}$ *or* $\mathtt{Recolor}_{k\to k'}(s) \in \mathcal{T}$, *then* $\mathtt{Void}(s0) \notin \mathcal{T}$ *and* $\mathtt{Void}(s1) \in \mathcal{T}$,
- *if* $\oplus(s) \in \mathcal{T}$, *then* $\mathtt{Void}(s0), \mathtt{Void}(s1) \notin \mathcal{T}$,
- *if* $\mathtt{Void}(s) \in \mathcal{T}$ *or* $\mathtt{c}_k(s) \in \mathcal{T}$, *then* $\mathtt{Void}(s0), \mathtt{Void}(s1) \in \mathcal{T}$. ⌟

Recall that, due to Rabin's famous Tree Theorem [26], the validity of a given MSO sentence Ξ in $\mathcal{T}_{\mathrm{bin}}$ is decidable. Also, it should be obvious that, given a decorated tree, checking well-decoratedness can be done in first-order logic. More precisely, fixing $\mathbb{L}$, Σ, and Cnst, there is a first-order sentence Φ_{well} such that for any $(\mathbb{L}, \mathrm{Cnst}, \Sigma)$-decorated tree $\mathcal{T}$, $\mathcal{T}$ is well-decorated *iff* $\mathcal{T} \models \Phi_{\mathrm{well}}$.

▶ **Definition 7.** *Let $\mathcal{T}$ be a $(\mathbb{L}, \mathrm{Cnst}, \Sigma)$-well-decorated tree. We define the function* $\mathsf{ent}^{\mathcal{T}} : \{0,1\}^* \to 2^{\mathrm{Cnst}\cup\{0,1\}^*}$ *mapping each null* $s \in \{0,1\}^*$ *to its* entities *(a set of nulls and constants) as follows:*

$$\mathsf{ent}^{\mathcal{T}}(s) = \{ss' \mid *_k(ss') \in \mathcal{T}, s' \in \{0,1\}^*\} \cup \{\mathtt{c} \mid \mathtt{c}_k(ss') \in \mathcal{T}, \mathtt{c} \in \mathrm{Cnst}, s' \in \{0,1\}^*\}.$$

Every tree node s also endows each of its entities with a color *from $\mathbb{L}$ through the function* $\mathsf{col}^{\mathcal{T}}_{s} : \mathsf{ent}^{\mathcal{T}}(s) \to \mathbb{L}$ *in the following way:*

$$\mathsf{col}^{\mathcal{T}}_{s}(e) = \begin{cases} k & \textit{if } e = \mathsf{c} \in \mathrm{Cnst} \textit{ and } \mathsf{c}_k(s) \in \mathcal{T}, \textit{or if } e = s \in \{0,1\}^* \textit{ and } *_k(s) \in \mathcal{T}, \\ k' & \textit{if } \mathtt{Recolor}_{k \to k'}(s) \in \mathcal{T} \textit{ and } \mathsf{col}^{\mathcal{T}}_{s0}(e) = k, \\ \mathsf{col}^{\mathcal{T}}_{s0}(e) & \textit{if } \mathtt{Recolor}_{k \to k'}(s) \in \mathcal{T} \textit{ and } \mathsf{col}^{\mathcal{T}}_{s0}(e) \neq k, \textit{or if } \mathtt{Add}_{\mathtt{R},\vec{k}}(s) \in \mathcal{T}, \\ \mathsf{col}^{\mathcal{T}}_{sb}(e) & \textit{if } \oplus(s) \in \mathcal{T}, e \in \mathsf{ent}^{\mathcal{T}}(sb), \textit{ and } b \in \{0,1\}. \end{cases}$$

Every node s is assigned a set of Σ-atoms over its entities as indicated by the sets Atoms_s*:*

$$\mathrm{Atoms}_s = \begin{cases} \{\top(\mathsf{c})\} & \textit{if } \mathsf{c}_k(s) \in \mathcal{T}, \mathsf{c} \in \mathrm{Cnst}, \\ \{\top(s)\} & \textit{if } *_k(s) \in \mathcal{T}, \\ \{\mathtt{R}(\vec{e}) \mid \mathsf{col}^{\mathcal{T}}_{s}(\vec{e}) = \vec{k}\} & \textit{if } \mathtt{Add}_{\mathtt{R},\vec{k}}(s) \in \mathcal{T}, \\ \emptyset & \textit{otherwise.} \end{cases}$$

Defining a colored instance *as a pair $(\mathcal{I}, \lambda)$ of an instance $\mathcal{I}$ and a function λ mapping elements of* $\mathsf{dom}(\mathcal{I})$ *to colors in a set $\mathbb{L}$, we now associate each node s in $\mathcal{T}$ with the colored Σ-instance $(\mathcal{I}^{\mathcal{T}}_{s}, \lambda^{\mathcal{T}}_{s})$, with $\mathcal{I}^{\mathcal{T}}_{s} = \bigcup_{s' \in \{0,1\}^*} \mathrm{Atoms}_{ss'}$ and $\lambda^{\mathcal{T}}_{s} = \mathsf{col}^{\mathcal{T}}_{s}$. Finally, we define the* colored instance $(\mathcal{I}^{\mathcal{T}}, \lambda^{\mathcal{T}})$ represented by $\mathcal{T}$ *as $(\mathcal{I}^{\mathcal{T}}_{\varepsilon}, \lambda^{\mathcal{T}}_{\varepsilon})$ where ε is the root of $\mathcal{T}$.*

▶ **Definition 8.** *Given a colored instance $(\mathcal{I}, \lambda)$ over a finite set* Cnst *of constants and a countable set of nulls as well as a finite signature Σ, the* cliquewidth *of $(\mathcal{I}, \lambda)$, written* $\mathsf{cw}(\mathcal{I}, \lambda)$*, is defined to be the smallest natural number n such that $(\mathcal{I}, \lambda)$ is isomorphic to a colored instance represented by some $(\mathbb{L}, \mathrm{Cnst}, \Sigma)$-well-decorated tree with $|\mathbb{L}| = n$. If no such number exists, we let $\mathsf{cw}(\mathcal{I}, \lambda) = \infty$. The cliquewidth $\mathsf{cw}(\mathcal{I})$ of an instance $\mathcal{I}$ is defined to be the minimum cliquewidth over all of its colored versions.* ⌟

▶ **Example 9.** The instance $\mathcal{I}_< = \{\mathtt{R}(n, m) \mid n, m \in \mathbf{N}, n < m\}$ has a cliquewidth of 2, witnessed by the well-decorated tree corresponding to the (non-well-founded) expression E implicitly defined by $E = \mathsf{Add}_{\mathtt{R},1,2}(*_1 \oplus \mathsf{Recolor}_{1\to 2}(E))$.

We will later use the following operation on decorated trees.

▶ **Definition 10.** *Let $(\mathcal{I}, \lambda)$ be a colored instance and $\mathcal{T}$ be a well-decorated tree witnessing that $\mathsf{cw}(\mathcal{I}, \lambda) \leq n$. Then for any $\mathtt{R} \in \Sigma$ we let $\mathsf{Add}_{\mathtt{R},\vec{k}}(\mathcal{I}, \lambda)$ denote the instance $\mathcal{I}^{\mathcal{T}'}$ represented by the well-decorated tree $\mathcal{T}'$ defined as follows:*
- *The root ε of $\mathcal{T}'$ is decorated by* $\mathtt{Add}_{\mathtt{R},\vec{k}}$*,*
- *the left sub-tree of ε is isomorphic to $\mathcal{T}$,*
- *the right sub-tree of ε is wholly decorated with* `Void`*.* ⌟

4.2 Finite-Cliquewidth Sets and Decidability

We now identify a new class of rule sets for which DaMSOQ query entailment is decidable, that is, the class of *finite-cliquewidth sets.*

▶ **Theorem 11.** *For a fixed $n \in \mathbb{N}$, determining if a given MSO formula Ξ has a model $\mathcal{I}$ with $\mathsf{cw}(\mathcal{I}) \leq n$ is decidable. Thus, cliquewidth is MSO-friendly.*

Proof (Sketch). We use the classical idea of MSO interpretations. Picking $\mathbb{L} = \{1, \ldots, n\}$, one shows that for every given MSO sentence Ξ, one can compute an MSO sentence Ξ', such that for every $(\mathbb{L}, \mathrm{Cnst}, \Sigma)$-well-decorated tree $\mathcal{T}$, $\mathcal{I}^{\mathcal{T}} \models \Xi$ *iff* $\mathcal{T} \models \Xi'$. Thus, checking if Ξ holds in some Cnst, Σ-instance of cliquewidth $\leq n$ can be done by checking if Ξ' holds in some $(\mathbb{L}, \mathrm{Cnst}, \Sigma)$-well-decorated tree, which in turn is equivalent to the existence of a $(\mathbb{L}, \mathrm{Cnst}, \Sigma)$-decorated tree that is a model of $\Xi' \wedge \Phi_{\mathrm{well}}$. Obtain Ξ'' from $\Xi' \wedge \Phi_{\mathrm{well}}$ by reinterpreting all unary predicates as MSO set variables that are quantified over existentially. Ξ'' is an MSO formula over the signature $\{\mathtt{Succ}_0, \mathtt{Succ}_1\}$ which is valid in $\mathcal{T}_{\mathrm{bin}}$ *iff* some decoration exists that makes $\Xi' \wedge \Phi_{\mathrm{well}}$ true. Thus, we have reduced our problem to checking the validity of a MSO sentence in $\mathcal{T}_{\mathrm{bin}}$, which is decidable by Rabin's Tree Theorem [26]. ◀

With this insight in place and the appropriate rule set notion defined, we can leverage Theorem 5 for our decidability result.

▶ **Definition 12.** *A rule set $\mathcal{R}$ is called a* finite-cliquewidth set *(**fcs**) iff for any database $\mathcal{D}$, there exists a universal model for $(\mathcal{D}, \mathcal{R})$ of finite cliquewidth.* ⌟

▶ **Corollary 13.** *For every **fcs** rule set $\mathcal{R}$, the query entailment problem $(\mathcal{D}, \mathcal{R}) \models (\mathtt{q}, \Xi)$ for databases $\mathcal{D}$, and DaMSOQs $(\mathtt{q}, \Xi)$ is decidable.*

In view of Example 9, it is not hard to verify that the rule set $\mathcal{R}^{\infty}_{\mathrm{tran}}$ from the introduction is **fcs**. Yet, it is neither **bts** (as argued before), nor **fus**, which can be observed from the fact that it does not admit a finite rewriting of the BCQ $\mathtt{E}(\mathtt{a}, \mathtt{b})$. Notably, it also does not exhibit *finite controllability* (**fc**), another generic property that guarantees decidability of query entailment [28]. A rule set $\mathcal{R}$ is **fc** *iff* for every $\mathcal{D}$ and CQ q with $(\mathcal{D}, \mathcal{R}) \not\models q$ there exists a *finite* (possibly non-universal) model $\mathcal{I} \models (\mathcal{D}, \mathcal{R})$ with $\mathcal{I} \not\models q$. Picking $\mathcal{D} = \{\mathtt{E}(\mathtt{a}, \mathtt{b})\}$ and $q = \exists x \mathtt{E}(x, x)$ reveals that $\mathcal{R}^{\infty}_{\mathrm{tran}}$ is not **fc**. Therefore, **fcs** encompasses rule sets not captured by any of the popular general decidability classes (namely, **bts**, **fus**, and **fc**). On another note, is no surprise that, akin to **bts**, **fus**, and **fc**, the membership of a rule set in **fcs** is undecidable, which can be argued exactly in the same way as for **bts** and **fus** [3].

4.3 Cliquewidth and Treewidth

We now show that for binary signatures, the class of instances with finite cliquewidth subsumes the class of instances with finite treewidth, implying **bts** $\subseteq$ **fcs**.

▶ **Theorem 14.** *Let $\mathcal{I}$ be a countable instance over a binary signature. If $\mathcal{I}$ has finite treewidth, then $\mathcal{I}$ has finite cliquewidth.*

Proof (Sketch). We convert a tree decomposition T of $\mathcal{I}$ with a width n into a well-decorated tree: (1) By copying nodes, transform T into an infinite binary tree T'. (2) For each term t from $\mathcal{I}$, let the *pivotal* node of t in T' be the node closest to the root containing t. For any two terms t and t' from $\mathcal{I}$ co-occurring in an atom, their pivotal nodes are in an ancestor relationship. (3) Assign one of $n + 1$ "slots" to every term so that in each node of T', every element has a distinct slot. (4) Extract a well-decorated tree from T' by transforming every node into the following bottom-up sequence of operations: (i) $\oplus$-assemble the input from below, (ii) introduce every element for which the current node is pivotal with a color that encodes "open link requests" to elements (identified by their slots) further up, (iii) satisfy the color-link requests from below via $\mathtt{Add}$, (iv) remove the satisfied requests by $\mathtt{Recolor}$. ◀

We note that the converse of Theorem 14 does not hold: Despite its finite cliquewidth, the treewidth of instance $\mathcal{I}_{<}$ from Example 9 is infinite, as its $\mathtt{R}$-edges form an infinite clique.

To the informed reader, Theorem 14 might not come as a surprise, given that this relationship is known to hold for countable unlabelled undirected graphs[5] [12]. It does, however, cease to hold for infinite structures with predicates of higher arity.

▶ **Example 15.** Let $\mathtt{R}$ be a ternary predicate. The instance $\mathcal{I}_{\text{tern}} = \{\mathtt{R}(-1, n, n{+}1) \mid n \in \mathbb{N}\}$ has a treewidth of 2, however, it does not have finite cliquewidth [15]. Concomitantly, the rule set $\mathcal{R}_{\text{tern}} = \{\mathtt{R}(v, x, y) \to \exists z \mathtt{R}(v, y, z)\}$ is **bts**, but not **fcs**.

While this result may be somewhat discouraging, one can show that its effects can be greatly mitigated by the technique of reification.

▶ **Definition 16.** *Given a finite signature $\Sigma = \Sigma_{\leq 2} \uplus \Sigma_{\geq 3}$ divided into at-most-binary and higher-arity predicates, we define the* reified version *of Σ as the binary signature $\Sigma^{\text{rf}} = \Sigma_{\leq 2} \uplus \Sigma^{\text{rf}}_2$ with $\Sigma^{\text{rf}}_2 = \{\mathtt{R}_i \mid \mathtt{R} \in \Sigma_{\geq 3}, 1 \leq i \leq \mathsf{ar}(\mathtt{R})\}$ a fresh set of binary predicates. The function* reify *maps atoms over $\Sigma_{\leq 2}$ to themselves, while any higher-arity atom $\alpha = \mathtt{R}(t_1, \ldots, t_k)$ with $k \geq 3$ is mapped to the set $\{\mathtt{R}_i(u_\alpha, t_i) \mid 1 \leq i \leq k\}$, where u_α is a fresh null or variable. We lift* reify *to instances, rules, and queries in the natural way.* ⌟

It is best to think of the "reification term" u_α as a locally existentially quantified variable. In particular, in rule heads, u_α will be existentially quantified. Moreover, in datalog queries, reify is only applied to Σ_{EDB}-atoms, while Σ_{IDB}-atoms are left unaltered; this ensures that the result is again a datalog query.

▶ **Example 17.** Consider the instance $\mathcal{I}_{\text{tern}}$ from Example 15. We observe that $\mathsf{reify}(\mathcal{I}_{\text{tern}}) = \{\mathtt{R}_1(u_n, -1), \mathtt{R}_2(u_n, n), \mathtt{R}_3(u_n, n{+}1) \mid n \in \mathbb{N}\}$ has a treewidth of 3 and a cliquewidth of 6, witnessed by the (non-well-founded) expression $\mathsf{Add}_{\mathtt{R}_1,5,1}(*_1 \oplus E)$, where E is implicitly defined via $E = \mathsf{Recolor}_{2\to 3}(\mathsf{Recolor}_{4\to 5}(\mathsf{Recolor}_{3\to 6}(\mathsf{Add}_{\mathtt{R}_3,4,3}(\mathsf{Add}_{\mathtt{R}_2,4,2}(*_2 \oplus (*_4 \oplus E))))))$.

Let us list in all brevity some pleasant and fairly straightforward properties of reification:

(i) If $\mathsf{tw}(\mathcal{I})$ is finite, then so are $\mathsf{tw}(\mathsf{reify}(\mathcal{I}))$ and $\mathsf{cw}(\mathsf{reify}(\mathcal{I}))$.
(ii) $\mathbf{Ch}_\infty(\mathsf{reify}(\mathcal{I}), \mathsf{reify}(\mathcal{R})) \equiv \mathsf{reify}(\mathbf{Ch}_\infty(\mathcal{I}, \mathcal{R}))$.
(iii) If $\mathcal{R}$ is **bts**, then $\mathsf{reify}(\mathcal{R})$ is **bts** and **fcs**.
(iv) $(\mathcal{D}, \mathcal{R}) \models (\mathsf{q}, \Xi)$ *iff* $(\mathsf{reify}(\mathcal{D}), \mathsf{reify}(\mathcal{R})) \models (\mathsf{reify}(\mathsf{q}), \mathsf{reify}(\Xi))$.

▶ **Example 18.** Revisiting Example 15, we can confirm that $\mathsf{reify}(\mathcal{R}_{\text{tern}})$ comprising the rule $\mathtt{R}_1(u, v) \wedge \mathtt{R}_2(u, x) \wedge \mathtt{R}_3(u, y) \to \exists u' z \big(\mathtt{R}_1(u', v) \wedge \mathtt{R}_2(u', y) \wedge \mathtt{R}_3(u', z)\big)$ is **bts** and **fcs**.

The above insights regarding reification allow us to effortlessly reduce any query entailment problem over an arbitrary **bts** rule set to a reasoning problem over a binary **fcs** one. Also, reification is a highly local transformation; it can be performed independently and atom-by-atom on $\mathcal{D}$, $\mathcal{R}$, and (q, Ξ). Therefore, restricting ourselves to **fcs** rule sets does not deprive us of the expressiveness, versatility, and reasoning capabilities offered by arbitrary **bts** rule sets over arbitrary signatures, which includes the numerous classes based on guardedness: guarded and frontier-guarded rules as well as their respective variants weakly, jointly, and glut-(frontier-)guarded rules [3, 7, 23]. It is noteworthy that our line of argument also gives rise to an independent proof of decidability of query entailment for **bts**:[6]

[5] This follows as a direct consequence of a compactness property relating the cliquewidth of countable undirected graphs to that of their finite induced subgraphs.

[6] Note that this result establishes entailment for arbitrary DaMSOQs, while previously reported results [3, 7] only covered CQs. But even when restricting the attention to CQ entailment, the proofs given in these (mutually inspired) prior works do not appear entirely conclusive to us: both invoke a result by Courcelle [11], which, as stated in the title of the article and confirmed by closer inspection, only deals with classes of *finite* structures/graphs with a uniform treewidth bound. Hence, the case of infinite structures is not covered, despite being the prevalent one for universal models. Personal communication with Courcelle confirmed that the case of arbitrary countable structures – although generally believed to hold true – is not an immediate consequence of his result.

▶ **Theorem 19.** *For every* **bts** *rule set* $\mathcal{R}$*, the query entailment problem* $(\mathcal{D},\mathcal{R}) \models (\mathtt{q},\Xi)$ *for databases* $\mathcal{D}$*, and DaMSOQs* $(\mathtt{q},\Xi)$ *is decidable.*

Proof. As argued, $(\mathcal{D},\mathcal{R}) \models (\mathtt{q},\Xi)$ reduces to $(\mathsf{reify}(\mathcal{D}),\mathsf{reify}(\mathcal{R})) \models (\mathsf{reify}(\mathtt{q}),\mathsf{reify}(\Xi))$. As $\mathcal{R}$ is **bts**, so is $\mathsf{reify}(\mathcal{R})$. The latter being binary, we conclude that it is **fcs**. Then, the claim follows from decidability of DaMSOQ entailment for **fcs** rule sets (Corollary 13). ◀

5 Comparing FCS and FUS

We have seen that the well-known **bts** class is subsumed by the **fcs** class (directly for arities $\leq$2, and via reification otherwise). As discussed in the introduction, another prominent class of rule sets (incomparable to **bts**) with decidable CQ entailment is the **fus** class. We dedicate the remainder of the paper to mapping out the relationship between **fcs** and **fus**, obtaining the following two results (established in Section 5.1 and Section 5.2, respectively):

▶ **Theorem 20.** *Any* **fus** *rule set of single-headed rules over a binary signature is* **fcs***.*

▶ **Theorem 21.** *There exists a* **fus** *rule set of multi-headed rules over a binary signature that is not* **fcs***.*

As a consequence of these findings, the necessary restriction to single-headed rules prevents us from wielding the powers of reification in this setting.

5.1 The Case of Single-Headed Rules

In this section, we establish Theorem 20. To this end, let a binary signature Σ, a finite unification set $\mathcal{R}$ of single-headed rules over Σ, and a database $\mathcal{D}$ over Σ be arbitrary but fixed for the remainder of the section. We will abbreviate $\mathbf{Ch}_\infty(\mathcal{D},\mathcal{R})$ by $\mathbf{Ch}_\infty$. Noting that $\mathbf{Ch}_\infty$ is a universal model, Theorem 20 is an immediate consequence of the following lemma, which we are going to establish in this section.

▶ **Lemma 22.** $\mathbf{Ch}_\infty$ *has finite cliquewidth.*

Looking past datalog. Let $\mathbf{Ch}_\exists \subseteq \mathbf{Ch}_\infty$ be the instance containing the *existential atoms* of $\mathbf{Ch}_\infty$, that is, the atoms derived via the non-datalog rules of $\mathcal{R}$. We show that $\mathbf{Ch}_\exists$ forms a *typed polyforest*, meaning that $\mathbf{Ch}_\exists$ can be viewed as a directed graph where (i) edges are typed with binary predicates from Σ and (ii) when disregarding the orientation of the edges, the graph forms a forest. This implies that $\mathbf{Ch}_\exists$ has treewidth 1. A tree decomposition of $\mathbf{Ch}_\exists$ can be extended into a finite-width tree decomposition of $\mathcal{D} \cup \mathbf{Ch}_\exists$ by adding the finite set $\mathsf{dom}(\mathcal{D})$ to every node. In the sequel, we use $\mathcal{D} \cup \mathbf{Ch}_\exists$ as a basis, to which, in a very controlled manner, we then add the "missing" non-existential atoms derived via datalog rules. Thereby, $\mathcal{R}$ being **fus** will be of great help.

Rewriting datalog rules. We transform the datalog subset of $\mathcal{R}$ into a new rule set $\mathcal{R}^{\mathrm{rew}}_{\mathrm{DL}}$, which we then use to fix a set of colors and establish the finiteness of $\mathsf{cw}(\mathbf{Ch}_\infty)$. Letting $\mathcal{R}_{\mathrm{DL}}$ denote the set of all datalog rules from $\mathcal{R}$, we note the following useful equation: $\mathbf{Ch}_\infty = \mathbf{Ch}_\infty(\mathcal{D} \cup \mathbf{Ch}_\exists, \mathcal{R}_{\mathrm{DL}})$ (†). For any $\mathtt{R} \in \Sigma$, we let $\mathsf{rew}(\mathtt{R})$ denote the datalog rule set $\{\varphi(\vec{x},\vec{y}) \to \mathtt{R}(\vec{y}) \mid \exists\vec{x}\varphi(\vec{x},\vec{y}) \in \mathsf{rew}_{\mathcal{R}}(\mathtt{R}(\vec{y}))\}$ giving rise to the overall datalog rule set $\mathcal{R}^{\mathrm{rew}}_{\mathrm{DL}} = \bigcup_{\mathtt{R}\in\Sigma}\mathsf{rew}(\mathtt{R})$. We now show that the rule set $\mathcal{R}^{\mathrm{rew}}_{\mathrm{DL}}$ admits an important property:

▶ **Lemma 23.** *For any* $\mathtt{R}(\vec{t}) \in \mathbf{Ch}_\infty \setminus (\mathcal{D} \cup \mathbf{Ch}_\exists)$*, there exists a trigger* (ρ, h) *in* $\mathcal{D} \cup \mathbf{Ch}_\exists$ *with* $\rho \in \mathcal{R}^{\mathrm{rew}}_{\mathrm{DL}}$ *such that applying* (ρ, h) *adds* $\mathtt{R}(\vec{t})$ *to* $\mathcal{D} \cup \mathbf{Ch}_\exists$*.*

Proof. Given that $\mathcal{R}$ is **fus**, it follows that for any $\mathsf{ar}(\mathtt{R})$-tuple $\vec{t}$ of terms from $\mathcal{D} \cup \mathbf{Ch}_\exists$, there exists a CQ $\exists \vec{x} \varphi(\vec{x}, \vec{y}) \in \mathsf{rew}_\mathcal{R}(\mathtt{R}(\vec{y}))$ such that the following holds: $\mathbf{Ch}_\infty(\mathcal{D} \cup \mathbf{Ch}_\exists, \mathcal{R}) \models \mathtt{R}(\vec{t})$ *iff* $\mathcal{D} \cup \mathbf{Ch}_\exists \models \exists \vec{x} \varphi(\vec{x}, \vec{t})$. Thus, said trigger exists for some $\rho \in \mathsf{rew}(\mathtt{R})$ in $\mathcal{D} \cup \mathbf{Ch}_\exists$. ◀

From Lemma 23 and (†), we conclude $\mathbf{Ch}_1(\mathcal{D} \cup \mathbf{Ch}_\exists, \mathcal{R}^{\mathrm{rew}}_{\mathrm{DL}}) = \mathbf{Ch}_\infty$ (‡). This tells us that we can apply rules of $\mathcal{R}^{\mathrm{rew}}_{\mathrm{DL}}$ in one step to obtain $\mathbf{Ch}_\infty$. Ultimately, we will leverage this to bound the cliquewidth of $\mathbf{Ch}_\infty$. In view of (‡), we may reformulate Lemma 22 as follows:

▶ **Lemma 24.** $\mathbf{Ch}_1(\mathcal{D} \cup \mathbf{Ch}_\exists, \mathcal{R}^{\mathrm{rew}}_{\mathrm{DL}})$ *has finite cliquewidth.*

Separating connected from disconnected rules. Next, we distinguish between two types of rules in $\mathcal{R}^{\mathrm{rew}}_{\mathrm{DL}}$. We say a datalog rule is *disconnected iff* its head variables belong to distinct connected components in its body; otherwise it is called *connected*. We let $\mathcal{R}^{\mathrm{conn}}_{\mathrm{DL}}$ denote all connected rules of $\mathcal{R}^{\mathrm{rew}}_{\mathrm{DL}}$ and let $\mathcal{R}^{\mathrm{disc}}_{\mathrm{DL}}$ denote all disconnected rules. One can observe that upon applying connected rules, frontier variables can only be mapped to – and hence connect – "nearby terms" of $\mathcal{D} \cup \mathbf{Ch}_\exists$ (using a path-based distance, bounded by the size of rule bodies), which, together with our insights about the structure of $\mathcal{D} \cup \mathbf{Ch}_\exists$, permits the construction of a tree decomposition of finite width, giving rise to the following lemma:

▶ **Lemma 25.** $\mathbf{Ch}_1(\mathcal{D} \cup \mathbf{Ch}_\exists, \mathcal{R}^{\mathrm{conn}}_{\mathrm{DL}})$ *has finite treewidth.*

Note that this lemma does not generalize to all of $\mathcal{R}^{\mathrm{rew}}_{\mathrm{DL}}$, since disconnected rules from $\mathcal{R}^{\mathrm{disc}}_{\mathrm{DL}}$ might realize "concept products" as in $\mathtt{A}(x) \wedge \mathtt{A}(y) \to \mathtt{E}(x, y)$. Clearly, such rules from $\mathcal{R}^{\mathrm{disc}}_{\mathrm{DL}}$ are the reason why **bts** fails to subsume **fus**. Let us define $\mathbf{Ch}_{\exists+} = \mathbf{Ch}_1(\mathcal{D} \cup \mathbf{Ch}_\exists, \mathcal{R}^{\mathrm{conn}}_{\mathrm{DL}})$. As $\mathbf{Ch}_1(\mathbf{Ch}_\exists, \mathcal{R}^{\mathrm{rew}}_{\mathrm{DL}}) = \mathbf{Ch}_1(\mathbf{Ch}_{\exists+}, \mathcal{R}^{\mathrm{disc}}_{\mathrm{DL}})$, all that is left to show is:

▶ **Lemma 26.** $\mathbf{Ch}_1(\mathbf{Ch}_{\exists+}, \mathcal{R}^{\mathrm{disc}}_{\mathrm{DL}})$ *has finite cliquewidth.*

Searching for a suitable coloring. Having established the finite treewidth of $\mathbf{Ch}_{\exists+}$ by Lemma 25, its finite cliquewidth can be inferred by Theorem 14, implying the existence of a well-decorated tree $\mathcal{T}$ with $\mathcal{I}^\mathcal{T} = \mathbf{Ch}_{\exists+}$. Moreover, we know that $\mathbf{Ch}_{\exists+}$ already contains all terms of $\mathbf{Ch}_\infty$. Thus, all that remains is to "add" the missing datalog atoms of $\mathbf{Ch}_\infty \setminus \mathbf{Ch}_{\exists+}$ to $\mathcal{I}^\mathcal{T}$. Certainly, the coloring function $\mathsf{col}^\mathcal{T}_\varepsilon$ provided by $\mathcal{T}$ cannot be expected to be very helpful in this task. To work around this, the following technical lemma ensures that an arbitrary coloring can be installed on top of a given instance of finite cliquewidth.

▶ **Lemma 27** (Recoloring Lemma). *Let $\mathcal{I}$ be an instance satisfying* $\mathsf{cw}(\mathcal{I}) = n$ *and let* $\lambda' : \mathsf{dom}(\mathcal{I}) \to \mathbb{L}'$ *be an arbitrary coloring of $\mathcal{I}$. Then* $\mathsf{cw}(\mathcal{I}, \lambda') \leq (n+1) \cdot |\mathbb{L}'|$.

Proof (Sketch). Let $\mathsf{cw}(\mathcal{I}) = n$ be witnessed by a well-decorated tree $\mathcal{T}$. Let $\mathbb{L}$ with $|\mathbb{L}| = n$ be the set of colors used in $\mathcal{T}$. We construct a well-decorated tree $\mathcal{T}^{\lambda'}$ representing $(\mathcal{I}, \lambda')$ and using the color set $(\mathbb{L} \times \mathbb{L}') \uplus \mathbb{L}'$, thus witnessing $\mathsf{cw}(\mathcal{I}, \lambda') \leq (n+1) \cdot |\mathbb{L}'|$. We obtain $\mathcal{T}^{\lambda'}$ from $\mathcal{T}$ by modifying each node s of $\mathcal{T}$ as follows:

- If s is labeled with $*_k$, we change it to $*_{(k, \lambda'(s))}$.
- If s is labeled with $\mathtt{c}_k$ with $\mathtt{c} \in \mathrm{Cnst}$, we change it to $\mathtt{c}_{(k, \lambda'(\mathtt{c}))}$.
- If s is labeled with $\mathtt{Add}_{\mathtt{R},k}$, we replace s with a sequence of nodes, one for each decorator in $\{\mathtt{Add}_{\mathtt{R},(k,\ell)} \mid \ell \in \mathbb{L}'\}$. We proceed in an analogous fashion for decorators $\mathtt{Add}_{\mathtt{R},k,k'}$ and $\mathtt{Recolor}_{k \to k'}$.
- If v is labeled with other decorators, we keep it as is.

Last, on top of the obtained tree, we apply a "color projection" to $\mathbb{L}'$ by adding recoloring statements of the form $\mathtt{Recolor}_{(k,\ell) \to \ell}$ for all $(k, \ell) \in \mathbb{L} \times \mathbb{L}'$. To complete the construction, we add the missing nodes to $\mathcal{T}^{\lambda'}$, each decorated with $\mathtt{Void}$. ◀

We proceed by defining types for elements in $\mathbf{Ch}_{\exists+}$, giving rise to the desired coloring. For the following, note that by definition, any rule from $\mathcal{R}_{\mathrm{DL}}^{\mathrm{disc}}$ must have two frontier variables.

▶ **Definition 28.** *Let $\rho \in \mathcal{R}_{\mathrm{DL}}^{\mathrm{disc}}$ with* $\mathsf{body}(\rho) = \varphi(x_1, x_2, \vec{y})$*, with x_1, x_2 frontier variables. Let us define $\rho_1(x) = \exists \vec{y} x_2\ \varphi(x, x_2, \vec{y})$ and $\rho_2(x) = \exists \vec{y} x_1\ \varphi(x_1, x, \vec{y})$. Then, for a term $t \in \mathsf{dom}(\mathbf{Ch}_{\exists+})$, we define its* type *$\tau(t)$ as $\{\rho_i(x) \mid \rho \in \mathcal{R}_{\mathrm{DL}}^{\mathrm{disc}},\ \mathbf{Ch}_{\exists+} \models \rho_i(t), i \in \{1, 2\}\}$.* ⌟

Now, we define our new coloring function $\lambda_\exists$. Given a term t of $\mathbf{Ch}_{\exists+}$, we let $\lambda_\exists(t) = \tau(t)$.

▶ **Corollary 29.** *There exists an $n_\exists \in \mathbb{N}$ with* $\mathsf{cw}(\mathbf{Ch}_{\exists+}, \lambda_\exists) = n_\exists$.

Proof. From Lemma 25 and Theorem 14, we know that $\mathsf{cw}(\mathbf{Ch}_{\exists+})$ is finite. Hence, there exists a natural number n and a coloring λ such that $\mathsf{cw}(\mathbf{Ch}_{\exists+}, \lambda) = n$. Moreover, the codomain of $\lambda_\exists$ is finite, say n'. Thus, we get $n_\exists = (n + 1) \cdot n'$ by Lemma 27. ◀

Coping with disconnected rules. We now conclude the proof of Lemma 26, i.e., we show that $\mathsf{cw}(\mathbf{Ch}_1(\mathbf{Ch}_{\exists+}, \mathcal{R}_{\mathrm{DL}}^{\mathrm{disc}}), \lambda_\exists) \leq n_\exists$, thereby proving Theorem 20. To this end, consider some rule $\rho \in \mathcal{R}_{\mathrm{DL}}^{\mathrm{disc}}$. As stated earlier, we can assume that ρ is of the form $\varphi(x_1, x_2, \vec{y}) \to \mathtt{R}(x_1, x_2)$. From this, we obtain the following useful correspondence:

▶ **Lemma 30.** *For any $t, t' \in \mathsf{dom}(\mathbf{Ch}_{\exists+})$ and $\rho \in \mathcal{R}_{\mathrm{DL}}^{\mathrm{disc}}$,*

$$\mathbf{Ch}_{\exists+} \models \exists \vec{y} \varphi(t, t', \vec{y}) \quad \textit{iff} \quad \rho_1(x) \in \tau(t) \textit{ and } \rho_2(x) \in \tau(t').$$

Proof. ($\Rightarrow$) follows from the definition of each set. For ($\Leftarrow$), we exploit disconnectedness and split $\varphi(x_1, x_2, \vec{y})$ into two distinct parts. Let $\varphi_1(x_1, \vec{y}_1)$ be the connected component of $\varphi(x_1, x_2, \vec{y})$ that contains x_1, and $\varphi_2(x_2, \vec{y}_2)$ denote the remainder (which includes the connected component of x_2). By assumption, $\vec{y}_1$ and $\vec{y}_2$ are disjoint, whence $\exists \vec{y} \varphi(t, t', \vec{y})$ is equivalent to $\exists \vec{y}_1 \varphi_1(t, \vec{y}_1) \wedge \exists \vec{y}_2 \varphi_2(t', \vec{y}_2)$. Note that $\rho_1(x) \in \tau(t)$ implies $\mathbf{Ch}_{\exists+} \models \exists \vec{y}_1 \varphi_1(t, \vec{y}_1)$, while $\rho_2(x) \in \tau(t')$ implies $\mathbf{Ch}_{\exists+} \models \exists \vec{y}_2 \varphi_2(t', \vec{y}_2)$. Therefore, $\mathbf{Ch}_{\exists+} \models \exists \vec{y} \varphi(t, t', \vec{y})$. ◀

▶ **Lemma 31.** *Let $\rho \in \mathcal{R}_{\mathrm{DL}}^{\mathrm{disc}}$ be of the form $\varphi(x_1, x_2, \vec{y}) \to \mathtt{R}(x_1, x_2)$. Then,*

$$\bigcup_{\rho_1(x) \in \ell,\ \rho_2(x) \in \ell'} \mathsf{Add}_{\mathtt{R},\ell,\ell'}(\mathbf{Ch}_{\exists+}, \lambda_\exists) = \mathbf{Ch}_{\exists+} \cup \{\mathtt{R}(t, t') \mid \mathbf{Ch}_{\exists+} \models \exists \vec{y} \varphi(t, t', \vec{y})\}.$$

Proof. All $\mathbf{Ch}_{\exists+}$ atoms are contained in both sides of the equation. Considering any $\mathtt{R}(t, t') \notin \mathbf{Ch}_{\exists+}$ we find it contained in the left-hand side *iff* $\rho_1(x) \in \lambda_\exists(t) = \tau(t)$ and $\rho_2(x) \in \lambda_\exists(t') = \tau(t')$ *iff* $\mathbf{Ch}_{\exists+} \models \exists \vec{y} \varphi(t, t' \vec{y})$ (by Lemma 30) *iff* $\mathtt{R}(t, t')$ is contained in the right-hand side of the equation. ◀

Of course, the right hand side of the equation in Lemma 31 coincides with $\mathbf{Ch}_1(\mathbf{Ch}_{\exists+}, \{\rho\})$. We observe – given that the coloring $\lambda_\exists$ remains unaltered – that the finitely many distinct applications of $\mathsf{Add}_{\mathtt{R},\ell,\ell'}$ in Lemma 31 are independent and can be chained without changing the result. Further, no such application increases the cliquewidth of the instance it is applied to. Since these arguments can be lifted to the application of *all* rules from $\mathcal{R}_{\mathrm{DL}}^{\mathrm{disc}}$, we get

$$\mathsf{cw}\Big(\mathbf{Ch}_{\exists+} \cup \bigcup_{\varphi(x_1, x_2, \vec{y}) \to \mathtt{R}(x_1, x_2) \in \mathcal{R}_{\mathrm{DL}}^{\mathrm{disc}}} \{\mathtt{R}(t, t') \mid \mathbf{Ch}_{\exists+} \models \exists \vec{y} \varphi(t, t', \vec{y})\}\Big) \leq n_\exists.$$

Observe that the considered instance is equal to $\mathbf{Ch}_1(\mathbf{Ch}_{\exists+}, \mathcal{R}_{\mathrm{DL}}^{\mathrm{disc}})$. Hence, we have established Lemma 26, concluding Lemma 24, which finishes the proof of Lemma 22, thus entailing the desired Theorem 20.

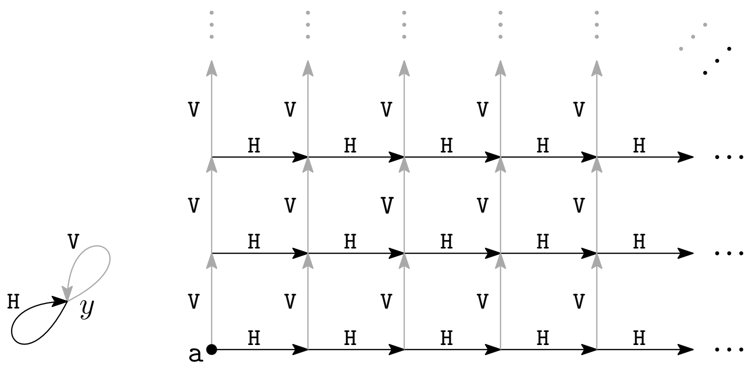


Figure 1 The instance $\mathcal{G}_\infty$.

5.2 The Case of Multi-Headed Rules

We will now prove Theorem 21, which implies that for multi-headed rules **fus** $\not\subseteq$ **fcs**. To this end, we exhibit a **fus** rule set that yields universal models of infinite cliquewidth. Let $\Sigma_{\text{grid}} = \{\mathtt{H}, \mathtt{V}\}$ with $\mathtt{H}, \mathtt{V}$ binary predicates. Let $\mathcal{R}_{\text{grid}}$ denote the following rule set over Σ_{grid}:

$$\begin{array}{lrl}
\text{(loop)} & \to & \exists x \;\big(\mathtt{H}(x,x) \wedge \mathtt{V}(x,x)\big) \\
\text{(grow)} & \to & \exists yy' \big(\mathtt{H}(x,y) \wedge \mathtt{V}(x,y')\big) \\
\text{(grid)} & \mathtt{H}(x,y) \wedge \mathtt{V}(x,x') \to & \exists y' \;\big(\mathtt{H}(x',y') \wedge \mathtt{V}(y,y')\big)
\end{array}$$

We make use of $\mathcal{R}_{\text{grid}}$ to establish Theorem 21, the proof of which consists of two parts. First, we provide a database $\mathcal{D}_{\text{grid}}$ to form a knowledge base $(\mathcal{D}_{\text{grid}}, \mathcal{R}_{\text{grid}})$ for which no universal model of finite cliquewidth exists (Lemma 33). Second, we show that $\mathcal{R}_{\text{grid}}$ is a finite unification set (Lemma 34).

$\mathcal{R}_{\text{grid}}$ is not a finite-cliquewidth set. Toward establishing this result, let $\mathcal{D}_{\text{grid}} = \{\top(\mathtt{a})\}$ and define the instance $\mathcal{G}_\infty$, where $\mathtt{a}$ is a constant and y as well as $x_{i,j}$ with $i, j \in \mathbb{N}$ are nulls:

$$\begin{aligned}
\mathcal{G}_\infty \quad = \quad & \big\{\mathtt{H}(\mathtt{a}, x_{1,0}), \mathtt{V}(\mathtt{a}, x_{0,1}), \mathtt{H}(y,y), \mathtt{V}(y,y)\big\} \\
& \cup \big\{\mathtt{H}(x_{i,j}, x_{i+1,j}), \mathtt{V}(x_{i,j}, x_{i,j+1}) \mid (i,j) \in (\mathbb{N} \times \mathbb{N}) \setminus \{(0,0)\}\big\}.
\end{aligned}$$

Figure 1 depicts $\mathcal{G}_\infty$ graphically. The following lemma summarizes consecutively established properties of $\mathcal{G}_\infty$ and its relationship with $(\mathcal{D}_{\text{grid}}, \mathcal{R}_{\text{grid}})$.

▶ **Lemma 32.** *With $\mathcal{G}_\infty$ and $(\mathcal{D}_{\text{grid}}, \mathcal{R}_{\text{grid}})$ as given above, we obtain:*

- *$\mathcal{G}_\infty$ has infinite cliquewidth,*
- *$\mathcal{G}_\infty$ is a universal model of $(\mathcal{D}_{\text{grid}}, \mathcal{R}_{\text{grid}})$,*
- *the only homomorphism $h : \mathcal{G}_\infty \to \mathcal{G}_\infty$ is the identity,*
- *any universal model of $(\mathcal{D}_{\text{grid}}, \mathcal{R}_{\text{grid}})$ contains an induced sub-instance isomorphic to $\mathcal{G}_\infty$,*
- *$(\mathcal{D}_{\text{grid}}, \mathcal{R}_{\text{grid}})$ has no universal model of finite cliquewidth.*

From the last point (established via the insight that taking induced subinstances never increases cliquewidth), the announced result immediately follows.

▶ **Lemma 33.** *$\mathcal{R}_{\text{grid}}$ is not* **fcs**.

$\mathcal{R}_{\text{grid}}$ is a finite-unification set. Unfortunately, $\mathcal{R}_{\text{grid}}$ does not fall into any of the known syntactic **fus** subclasses and showing that a provided rule set is **fus** is a notoriously difficult task in general.[7] At least, thanks to the rule (loop), every BCQ that is free of constants is always entailed and can thus be trivially re-written. For queries involving constants, we provide a hand-tailored query rewriting algorithm, along the lines of prior work exploring the multifariousness of **fus** [25]. The required argument is quite elaborate and we just sketch the main ideas here due to space restrictions.

We make use of special queries, referred to as *marked queries*: queries with some of their terms "marked" with the purpose of indicating terms that need to be mapped to database constants in the course of rewriting. The notion of query satisfaction is lifted to such marked CQs, which enables us to identify the subclass of *properly marked queries* as those who actually have a match into some instance $\mathbf{Ch}_\infty(\mathcal{D}, \mathcal{R}_{\text{grid}})$, where $\mathcal{D}$ is an arbitrary database.

With these notions at hand, we can now define a principled rewriting procedure consisting of the exhaustive application of three different types of transformation rules. We show that this given set of transformations is in fact sound and complete, i.e., it produces correct first-order rewritings upon termination. Last, we provide a termination argument by showing that the operations reduce certain features of the rewritten query. Thus, we arrive at the second announced result, completing the overall proof.

▶ **Lemma 34.** $\mathcal{R}_{\text{grid}}$ *is* **fus***.*

5.3 Fus and Expressive Queries

Let us put the negative result of Section 5.2 into perspective. Thanks to Corollary 13, we know that **fcs** ensures decidability of arbitrary DaMSOQ entailment, a quite powerful class of queries. By contrast, **fus** is a notion tailored to CQs and unions thereof. As it so happens, searching for a method to establish decidability of DaMSOQ entailment for arbitrary **fus** rule sets turns out to be futile. This even holds for a fixed database and a fixed rule set.

▶ **Lemma 35.** *DaMSOQ entailment is undecidable for* $(\mathcal{D}_{\text{grid}}, \mathcal{R}_{\text{grid}})$*.*

The corresponding proof is rather standard: One can, given a deterministic Turing machine TM, create a DaMSOQ $(\mathfrak{q}_{TM}, \Xi_{TM})$ that is even expressible in monadic datalog and, when evaluated over $\mathcal{G}_\infty$, uses the infinite grid to simulate a run of that Turing machine, resulting in a query match *iff* TM halts on the empty tape. As $\mathcal{G}_\infty$ is a universal model of $(\mathcal{D}_{\text{grid}}, \mathcal{R}_{\text{grid}})$, and DaMSOQ satisfaction is preserved under homomorphisms, the entailment $(\mathcal{D}_{\text{grid}}, \mathcal{R}_{\text{grid}}) \models (\mathfrak{q}_{TM}, \Xi_{TM})$ coincides with the termination of TM, concluding the argument.

6 Conclusions and Future Work

In this paper, we have introduced a generic framework, by means of which model-theoretic properties of a class of existential rule sets can be harnessed to establish that the entailment of *datalog/MSO queries* – a very expressive query formalism subsuming numerous popular query languages – is decidable for that class. We have put this framework to use by introducing *finite-cliquewidth sets* and clarified this class's relationship with notable others, resulting in the insight that a plethora of known as well as hitherto unknown decidability results can be uniformly obtained via the decidability of DaMSOQ entailment over finite-cliquewidth sets of rules. Our results entail various appealing directions for follow-up investigations:

[7] This should not be too surprising however, as being **fus** is an undecidable property [3].

- Certainly, the class of single-headed binary **fus** rule sets is not the most general fragment of **fus** subsumed by **fcs**. We strongly conjecture that many of the known, syntactically defined, "concrete subclasses" of **fus** are actually contained in **fcs**, and we are confident that corresponding results can be established.
- Conversely, we are striving to put the notion of **fcs** to good use by identifying comprehensive, syntactically defined subclasses of **fcs**, enabling decidable, highly expressive querying beyond the realms of **bts**, **fus**, or **fc**. As a case in point, the popular modeling feature of *transitivity* of a relation has been difficult to accommodate in existing frameworks [2, 17, 22, 32], whereas the observations presented in this paper seem to indicate that **fcs** can tolerate transitivity rules well and natively.
- Finally, we are searching for even more general MSO-friendly width notions that give rise to classes of rule sets subsuming **fcs**, and which also encompass **bts** natively, without arity restrictions or "reification detours."

Aside from these major avenues for future research, there are also interesting side roads worth exploring:

- Are there other, more general model-theoretic criteria of "structural well-behavedness" that, when ensured for universal models, guarantee decidability of query entailment "just" for (U)CQs? Clearly, if we wanted **fus** to be subsumed by such a criterion, it would have to accept structures like $\mathcal{G}_\infty$ (i.e., infinite grids) and we would have to relinquish decidability of DaMSOQ entailment.
- More generally: Are there other "decidability sweet spots" between expressibility of query classes and structural restrictions on universal models?

References

1 Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995. URL: `http://webdam.inria.fr/Alice/`.

2 Jean-François Baget, Meghyn Bienvenu, Marie-Laure Mugnier, and Swan Rocher. Combining existential rules and transitivity: Next steps. In Qiang Yang and Michael J. Wooldridge, editors, *Proceedings of the 24th International Joint Conference on Artificial Intelligence (IJCAI'15)*, pages 2720–2726. AAAI Press, 2015. URL: `http://ijcai.org/Abstract/15/385`.

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):1620–1654, 2011. `doi:10.1016/j.artint.2011.03.002`.

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

5 Pierre Bourhis, Markus Krötzsch, and Sebastian Rudolph. How to best nest regular path queries. In Meghyn Bienvenu, Magdalena Ortiz, Riccardo Rosati, and Mantas Simkus, editors, *Informal Proceedings of the 27th International Workshop on Description Logics, Vienna, Austria, July 17-20, 2014*, volume 1193 of *CEUR Workshop Proceedings*, pages 404–415. CEUR-WS.org, 2014. URL: `http://ceur-ws.org/Vol-1193/paper_80.pdf`.

6 Pierre Bourhis, Michael Morak, and Andreas Pieris. Making cross products and guarded ontology languages compatible. In Carles Sierra, editor, *Proceedings of the 26th International Joint Conference on Artificial Intelligence (IJCAI'17)*, pages 880–886. ijcai.org, 2017. `doi:10.24963/ijcai.2017/122`.

7 Andrea Calì, Georg Gottlob, and Michael Kifer. Taming the infinite chase: Query answering under expressive relational constraints. *Journal of Artificial Intelligence Research*, 48:115–174, 2013. `doi:10.1613/jair.3873`.

8 Andrea Calì, Georg Gottlob, and Andreas Pieris. Towards more expressive ontology languages: The query answering problem. *Artificial Intelligence*, 193:87–128, 2012. `doi:10.1016/j.artint.2012.08.002`.

9 Ashok K. Chandra, Harry R. Lewis, and Johann A. Makowsky. Embedded implicational dependencies and their inference problem. In *Proceedings of the 13th Annual ACM Symposium on Theory of Computing (STOC'81)*, pages 342–354. ACM, 1981. doi:10.1145/800076.802488.

10 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, *Proceedings of the 20th Annual ACM Symposium on Theory of Computing (STOC'88)*, pages 477–490. ACM, 1988. doi:10.1145/62212.62259.

11 Bruno Courcelle. The monadic second-order logic of graphs. I. Recognizable sets of finite graphs. *Information and Computation*, 85(1):12–75, 1990. doi:10.1016/0890-5401(90)90043-H.

12 Bruno Courcelle. Clique-width of countable graphs: A compactness property. *Discrete Mathematics*, 276(1-3):127–148, 2004. doi:10.1016/S0012-365X(03)00303-0.

13 Bruno Courcelle and Joost Engelfriet. *Graph Structure and Monadic Second-Order Logic – A Language-Theoretic Approach*, volume 138 of *Encyclopedia of mathematics and its applications*. Cambridge University Press, 2012. URL: http://www.cambridge.org/fr/knowledge/isbn/item5758776/?site_locale=fr_FR.

14 Alin Deutsch, Alan Nash, and Jeffrey B. Remmel. The chase revisited. In Maurizio Lenzerini and Domenico Lembo, editors, *Proceedings of the 27th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS'08)*, pages 149–158. ACM, 2008. doi:10.1145/1376916.1376938.

15 Thomas Feller, Tim S. Lyon, Piotr Ostropolski-Nalewaja, and Sebastian Rudolph. Finite-cliquewidth sets of existential rules: Toward a general criterion for decidable yet highly expressive querying. *CoRR*, abs/2209.02464, 2022. doi:10.48550/arXiv.2209.02464.

16 Daniela Florescu, Alon Y. Levy, and Dan Suciu. Query containment for conjunctive queries with regular expressions. In Alberto O. Mendelzon and Jan Paredaens, editors, *Proceedings of the 17th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS'98)*, pages 139–148. ACM, 1998. doi:10.1145/275487.275503.

17 Harald Ganzinger, Christoph Meyer, and Margus Veanes. The two-variable guarded fragment with transitive relations. In *Proceedings of the 14th Annual IEEE Symposium on Logic in Computer Science (LICS'99)*, pages 24–34. IEEE Computer Society, 1999. doi:10.1109/LICS.1999.782582.

18 Birte Glimm, Carsten Lutz, Ian Horrocks, and Ulrike Sattler. Conjunctive query answering for the description logic SHIQ. *Journal of Artificial Intelligence Research*, 31:157–204, 2008. doi:10.1613/jair.2372.

19 Kurt Gödel. *Über die Vollständigkeit des Logikkalküls*. PhD thesis, Universität Wien, 1929.

20 Georg Gottlob. Datalog+/-: A unified approach to ontologies and integrity constraints. In Valeria De Antonellis, Silvana Castano, Barbara Catania, and Giovanna Guerrini, editors, *Proceedings of the 17th Italian Symposium on Advanced Database Systems, (SEBD'09)*, pages 5–6. Edizioni Seneca, 2009.

21 Martin Grohe and György Turán. Learnability and definability in trees and similar structures. *Theory of Computing Systems*, 37(1):193–220, 2004. doi:10.1007/s00224-003-1112-8.

22 Emanuel Kieronski and Sebastian Rudolph. Finite model theory of the triguarded fragment and related logics. In *Proceedings of the 36th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS'21)*, pages 1–13. IEEE, 2021. doi:10.1109/LICS52264.2021.9470734.

23 Markus Krötzsch and Sebastian Rudolph. Extending decidable existential rules by joining acyclicity and guardedness. In Toby Walsh, editor, *Proceedings of the 22nd International Joint Conference on Artificial Intelligence (IJCAI'11)*, pages 963–968. IJCAI/AAAI, 2011. doi:10.5591/978-1-57735-516-8/IJCAI11-166.

24 Magdalena Ortiz, Sebastian Rudolph, and Mantas Simkus. Query answering in the horn fragments of the description logics SHOIQ and SROIQ. In Toby Walsh, editor, *Proceedings of the 22nd International Joint Conference on Artificial Intelligence (IJCAI'11)*, pages 1039–1044. IJCAI/AAAI, 2011. doi:10.5591/978-1-57735-516-8/IJCAI11-178.

25 Piotr Ostropolski-Nalewaja, Jerzy Marcinkowski, David Carral, and Sebastian Rudolph. A journey to the frontiers of query rewritability. *CoRR*, abs/2012.11269, 2020. `arXiv:2012.11269`.

26 Michael O. Rabin. Decidability of second-order theories and automata on infinite trees. *Transactions of the American Mathematical Society*, 141:1–35, 1969. URL: `http://www.jstor.org/stable/1995086`.

27 Juan L. Reutter, Miguel Romero, and Moshe Y. Vardi. Regular queries on graph databases. *Theory of Computing Systems*, 61(1):31–83, 2017. `doi:10.1007/s00224-016-9676-2`.

28 Riccardo Rosati. On the decidability and finite controllability of query processing in databases with incomplete information. In Stijn Vansummeren, editor, *Proceedings of the 25th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS'06)*, pages 356–365. ACM, 2006. `doi:10.1145/1142351.1142404`.

29 Sebastian Rudolph and Markus Krötzsch. Flag & check: Data access with monadically defined queries. In Richard Hull and Wenfei Fan, editors, *Proceedings of the 32nd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS'13)*, pages 151–162. ACM, 2013. `doi:10.1145/2463664.2465227`.

30 Sebastian Rudolph, Markus Krötzsch, and Pascal Hitzler. All elephants are bigger than all mice. In Franz Baader, Carsten Lutz, and Boris Motik, editors, *Proceedings of the 21st International Workshop on Description Logics (DL2008)*, volume 353 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2008. URL: `http://ceur-ws.org/Vol-353/RudolphKraetzschHitzler.pdf`.

31 Eric Salvat and Marie-Laure Mugnier. Sound and complete forward and backward chainings of graph rules. In Peter W. Eklund, Gerard Ellis, and Graham Mann, editors, *Proceedings of the 4th International Conference on Conceptual Structures (ICCS'96)*, volume 1115 of *LNCS*, pages 248–262. Springer, 1996. `doi:10.1007/3-540-61534-2_16`.

32 Wieslaw Szwast and Lidia Tendera. The guarded fragment with transitive guards. *Annals of Pure and Applied Logic*, 128(1-3):227–276, 2004. `doi:10.1016/j.apal.2004.01.003`.

Generalizing Greenwald-Khanna Streaming Quantile Summaries for Weighted Inputs*

Sepehr Assadi
Department of Computer Science, Rutgers University, Piscataway, NJ, USA

Nirmit Joshi
Department of Computer Science, Northwestern University, Evanston, IL, USA

Milind Prabhu
Department of Computer Science and Engineering, University of Michigan, Ann Arbor, MI, USA

Vihan Shah
Department of Computer Science, Rutgers University, Piscataway, NJ, USA

Abstract

Estimating quantiles, like the median or percentiles, is a fundamental task in data mining and data science. A (streaming) quantile summary is a data structure that can process a set S of n elements in a streaming fashion and at the end, for any $\phi \in (0,1]$, return a ϕ-quantile of S up to an ε error, i.e., return a ϕ'-quantile with $\phi' = \phi \pm \varepsilon$. We are particularly interested in comparison-based summaries that only compare elements of the universe under a total ordering and are otherwise completely oblivious of the universe. The best known deterministic quantile summary is the 20-year old Greenwald-Khanna (GK) summary that uses $O((1/\varepsilon)\log(\varepsilon n))$ space [SIGMOD'01]. This bound was recently proved to be optimal for all deterministic comparison-based summaries by Cormode and Vesleý [PODS'20].

In this paper, we study weighted quantiles, a generalization of the quantiles problem, where each element arrives with a positive integer weight which denotes the number of copies of that element being inserted. The only known method of handling weighted inputs via GK summaries is the naive approach of breaking each weighted element into multiple unweighted items, and feeding them one by one to the summary, which results in a prohibitively large update time (proportional to the maximum weight of input elements).

We give the first non-trivial extension of GK summaries for weighted inputs and show that it takes $O((1/\varepsilon)\log(\varepsilon n))$ space and $O(\log(1/\varepsilon) + \log\log(\varepsilon n))$ update time per element to process a stream of length n (under some quite mild assumptions on the range of weights and ε). En route to this, we also simplify the original GK summaries for unweighted quantiles.

2012 ACM Subject Classification Theory of computation → Streaming, sublinear and near linear time algorithms; Theory of computation → Approximation algorithms analysis; Theory of computation → Data structures design and analysis

Keywords and phrases Streaming algorithms, Quantile summaries, Rank estimation

Digital Object Identifier 10.4230/LIPIcs.ICDT.2023.19

Funding *Sepehr Assadi*: Research supported in part by the NSF CAREER Grant CCF-2047061, and gift from Google Research.
Vihan Shah: Research supported in part by the NSF CAREER Grant CCF-2047061. Part of this work was done when the author was an undergraduate student at Rutgers University-Camden and was supported in part by the NSF grant CCF-1910565.

Acknowledgements We would like to thank Rajiv Gandhi for making the collaboration between the authors possible and for his support throughout this project.

* A full version of the paper with the same title appears on arXiv.

1 Introduction

Given a set S of elements $x_1, \ldots, x_n$ from a totally ordered universe, the rank of an element x in this universe, denoted by $\text{rank}(x)$, is the number of elements x_j in S with $x_j \leq x$. Similarly, the ϕ-quantile of S, for any $\phi \in (0, 1]$, is the element $x_i \in S$ with $\text{rank}(x_i) = \lceil \phi \cdot n \rceil$. Computing quantiles is a fundamental problem with a wide range of applications considering they provide a concise representation of the distribution of the input elements. Throughout this paper, we solely focus on comparison-based algorithms for this problem that can only compare two elements of the universe according to their ordering and are otherwise completely oblivious of the universe.

We are interested in the quantile estimation problem in the *streaming* model, introduced in the seminal work of Alon, Matias, and Szegedy [3]. In this model, the elements of S are arriving one by one in an arbitrary order and the streaming algorithm can make just one pass over this data and use a limited memory and thus cannot simply store S entirely. Already more than four decades ago, Munro and Paterson proved that one cannot solve this problem *exactly* in the streaming model [17] and thus the focus has been on finding *approximation* algorithms: Given $\varepsilon > 0$, the algorithm is allowed to return an ε-approximate ϕ-quantile, i.e., a $(\phi \pm \varepsilon)$-quantile. More formally, we are interested in the following data structure:

▶ **Definition 1** (Quantile Summary). *An* ε-approximate quantile summary *processes any set of elements in a streaming fashion and at the end finds an ε-approximate ϕ-quantile for any given quantile $\phi \in (0, 1]$, defined as any ϕ'-quantile for $\phi' \in [\phi - \varepsilon, \phi + \varepsilon]$.*

In the absence of the streaming aspect of the problem, one can always compute an ε-approximate quantile summary in $O(1/\varepsilon)$ space; simply store the ε-quantile, 3ε-quantile, 5ε-quantile and so on from S. It is easy to see that given any ϕ, returning the closest stored quantile results in an ε-approximate ϕ-quantile. It is also easy to see that this space is information-theoretically optimal for the problem. However, this approach cannot be directly implemented in the streaming model as a-priori it is not clear how to compute the needed quantiles of S in the first place.

The first (streaming) ε-approximate quantile summary was proposed by Manku, Rajagopalan, and Lindsay [14]. The MRL summary uses $O((1/\varepsilon) \log^2(\varepsilon n))$ space and requires the prior knowledge of the length of the stream. This summary was soon after improved by Greenwald and Khanna [9] who proposed the GK summary that uses $O((1/\varepsilon) \log(\varepsilon n))$ space and no longer requires knowing the length of the stream. This is the state-of-the-art for deterministic comparison-based summaries. By allowing for randomization one can further improve upon the space requirement of these algorithms and achieve bounds with no dependence on the length of the stream. The state-of-the-art result for randomized summaries is an algorithm due to Karnin, Lang and Liberty [12] which uses $O((1/\varepsilon) \log \log (1/\varepsilon\delta))$ space to construct an ε-approximate quantile summary with probability at least $(1-\delta)$. We provide a more detailed discussion of the literature on randomized summaries and non-comparison based summaries in the full version of the paper. While using randomization gives streaming algorithms which are more space-efficient, a major drawback of most of these algorithms is that their analysis crucially depends on the assumption that the input stream is independent of the randomness used by the algorithm. This assumption is unrealistic in several settings; for instance, when the future input to the algorithm depends on its previous outputs. Recently, this has invoked an interest in *adverserially robust* algorithms that work even when an adversary is allowed to choose the stream adaptively [16, 7, 10, 2, 8, 18]. Deterministic algorithms are inherently adverserially robust and therefore understanding them is an interesting goal in itself.

In this paper, we focus on deterministic summaries; specifically on furthering our understanding of GK summaries. Over the years, two important questions have been raised about them: Is it possible to *improve* the space of GK summaries, perhaps even all the way down to the information-theoretic optimal bound of $O(1/\varepsilon)$? And, is it possible to *simplify* GK summaries and their intricate analysis in a way that allow for generalizations of these summaries to be more easily proposed and studied? (see Problem 2 of "List of Open Problems in Sublinear Algorithms" [19] posed by Cormode or [12, 6, 13, 1] for similar variations of this question, for example, when the input items are weighted).

The first question was addressed initially by Hung and Ting [11] who proved an $\Omega((1/\varepsilon)\log(1/\varepsilon))$ space lower bound for ε-approximate quantile summaries, improving over the information-theoretic bound. Very recently, this question was fully settled by Cormode and Vesleý [6] who proved that in fact GK summaries are asymptotically optimal: $\Omega((1/\varepsilon)\log(\varepsilon n))$ space is needed for any deterministic (comparison-based) summary. The second question above however is still left without a satisfying resolution. In this paper, we make progress towards answering this question by showing that the GK summary can be generalized to handle weighted inputs. Formally, we present algorithms to construct the following data-structure.

▶ **Definition 2** (Weighted Quantile Summary)**.** *Consider a weighted stream S_w of n updates $(x_i, w(x_i))$ for $1 \leq i \leq n$. The i-th update denotes the insertion of $w(x_i)$ copies of the element x_i (the weight $w(x_i)$ is guaranteed to be a positive integer). We define $W_k = \sum_{i=1}^{k} w(x_i)$ to be the sum of the weights of the first k elements of S_w. An* ε-approximate weighted quantile summary *is a data-structure that makes a single pass over S_w and at the end, for any $\phi \in [0,1)$, finds an x_j such that,*

$$\left(\sum_{x_i<x_j} w(x_i),\ w(x_j) + \sum_{x_i<x_j} w(x_i)\right] \cap \left[(\phi-\varepsilon)W_n, (\phi+\varepsilon)W_n\right] \neq \emptyset. \tag{1}$$

A notable application of the weighted quantiles problem is in the very popular XGBoost library [5] which contains an efficient implementation of the gradient-boosted trees algorithm. To solve the weighted quantiles problem, XGBoost uses a merge and prune summary [4] via an extension of the ideas in [15]. However, they do not give an upper bound on the space achieved by this summary. Our result addresses this issue by proposing a new and efficient weighted quantile summary with formal space guarantees.

Our Contributions

One approach to construct a weighted quantile summary is to break each weighted item into multiple unweighted items and feed them to an unweighted summary such as the GK summary. However, such algorithms are slow since the time required to process an element is proportional to its weight. As such, it has been asked if faster algorithms exist. We answer this in the affirmative by proposing a fast algorithm for this problem in Section 3. In particular, this algorithm uses $O((1/\varepsilon)\log(\varepsilon n))$ space and $O(\log(1/\varepsilon) + \log\log(\varepsilon n))$ update time per element to process a stream of length n, when the weights are poly(n) and $\varepsilon \geq \frac{1}{n^{1-\delta}}$ for any $\delta \in (0,1)$ (Theorem 11). This matches the space and time complexity of the GK summary when it is used to summarize a stream of n unweighted items [9, 13]. To our knowledge, this constitutes the first (non-trivial) extension of the GK algorithm for weighted streams.

En route to this, we also present a new description of the GK summaries by simplifying or entirely bypassing several of their more intricate components in [9] such as their so-called "tree representation" and their complex "compress" operations in Section 4.2. As a warm-up to this, we also present a simple and greedy algorithm for unweighted quantiles which uses $O((1/\varepsilon)\log^2(\varepsilon n))$ space in Section 4.1. This algorithm, although has a suboptimal space bound, will be useful in motivating and providing intuition for GK summaries. Interestingly, this summary is quite similar (albeit *not* identical) to the so-called GKAdaptive summary [13] that was already proposed by [9] as a more practical variant of their GK summaries (Luo *et al.* [13] further confirmed this by showing that GKAdaptive outperforms GK summaries experimentally). While no theoretical guarantees are known for GKAdaptive, we prove that this slight modification of this algorithm submits to a simple analysis of an $O((1/\varepsilon)\log^2(\varepsilon n))$ space upper bound (Theorem 25).

We also emphasize that, similar to the original GK summaries, our weighted extension does not need foreknowledge of the stream length. This guarantee implies that we can track the quantiles throughout the stream, with error proportional to the current weight of the stream, and not only at the end.

2 Preliminaries

We now present the basic setup of our quantile summary and preliminary definitions. We start with an alternate equivalent formulation of the problem defined in Definition 2 in terms of the unweighted quantiles problem for which we first define the notion of *unfolding* streams.

Unfolding Streams

For the weighted stream S_w, we define its corresponding unfolded stream $\mathsf{Unfold}(S_w)$ to be the stream which contains $w(x_i)$ copies of x_i for $1 \leq i \leq n$. More explicitly,

$$\mathsf{Unfold}(S_w) := \langle x_1^{(1)}, x_1^{(2)}, \ldots, x_1^{(w(x_1))}, \ldots x_n^{(1)}, x_n^{(2)}, \ldots, x_n^{(w(x_n))} \rangle$$

where $x_i^{(j)}$ is the j-th copy of element x_i.

It is easy to verify that the goal of the problem, as stated in Definition 2, is equivalent to creating an ε-approximate quantile summary of $\mathsf{Unfold}(S_w)$. Note that to break ties while assigning ranks to equal elements, we will assume that elements that appeared earlier in $\mathsf{Unfold}(S_w)$ have lower ranks. As a side note we would like to point out here that although the algorithm we present does not "unfold" the stream, we will continue working with $\mathsf{Unfold}(S_w)$ to present the analysis of the algorithm.

We use `WQS` to denote the summary of S_w that our algorithm creates. `WQS` will consist of a subset of the elements of the stream along with some auxiliary metadata about the stored elements. We use e_i to denote the i-th largest element of the stream stored in `WQS`. We use $e_i^{(j)}$ to refer to the the j-th copy of e_i in $\mathsf{Unfold}(S_w)$, for $1 \leq j \leq w(e_i)$. We also use e to refer to an arbitrary element of the summary (when the rank is not relevant). The number of elements of the stream stored in `WQS` shall be denoted by s. For each element e, `WQS` stores $w(e)$. The other main information we store for each element e are its r-min and r-max values, which we now define:

- r-min(e) and r-max(e): are lower and upper bounds maintained by `WQS` on the rank of $e^{(1)}$ (the first copy of e to appear in $\mathsf{Unfold}(S_w)$). Since we are not storing all elements, we cannot determine the exact rank of a stored element, and thus focus on maintaining proper lower and upper bounds.

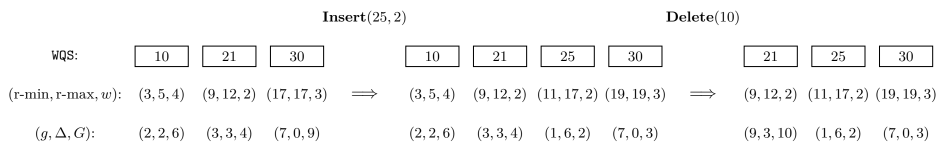


Figure 1 An illustration of the update operations in the summary starting from some arbitrary state (the parameters (g, Δ, G) in this figure are defined in Section 2.2).

To handle corner cases that arise later, we assume that `WQS` contains a sentinel element e_0 and define r-min(e_0) = r-max$(e_0) = 0$ and $w(e_0) = 1$. Also, we insert a $+\infty$ element at the start of the stream which is considered larger than any other element and store it in `WQS` as e_s. The r-min and r-max of this element is also always equal to the weight of inserted elements (including itself). Since $+\infty$ is the largest element, inserting it in S_w does not affect the rank of any other element.

▶ **Observation 3.** (r-min$(e) + j - 1$) *and* (r-max$(e) + j - 1$) *are lower and upper bounds on the rank of* $e^{(j)}$.

During the stream, we insert and delete elements from the summary. This changes the rank of the elements so we have to update `WQS` to reflect the changes. The procedure used to update the r-min and r-max values of elements is describe below:

Insert$(x, w(x))$. Inserts a given element x with weight $w(x)$ into `WQS`.

(i) Store the element x along with its weight $w(x)$ in `WQS`.
(ii) Find the smallest element e_i in `WQS` such that $e_i > x$;
(iii) Set r-min(x) = r-min$(e_{i-1})+w(e_{i-1})$ and r-max(x) = r-max(e_i); moreover, increase r-min(e_j) and r-max(e_j) by $w(x)$ for all $j \geq i$.

Delete(e_i). Deletes the element e_i from `WQS`.
(i) Remove element e_i from `WQS`; keep all remaining r-min, r-max values unchanged.

We now justify that after the above operations are performed, for each element e in the summary, its r-min and r-max values are valid lower and upper bounds on the rank of $e^{(1)}$. Suppose that a new element x satisfying $e_{i-1} < x < e_i$ is inserted into `WQS`. The rank of x is at least one more than the rank of the last copy of e_{i-1}. Therefore, r-min(x), which is set to (r-min$(e_{i-1}) + w(e_{i-1}) - 1) + 1$ = r-min$(e_{i-1}) + w(e_{i-1})$, is a valid lower bound on the rank of $e^{(1)}$. The rank of x is at most equal to the rank of the first copy of e_i. Therefore, setting r-max(x) equal to r-max(e_i) makes it a valid upper bound. After the insertion of x, the ranks of all elements in the summary larger than x increase by $w(x)$ and hence their r-min and r-max values need to be updated. The ranks of elements smaller than x do not change. Also, deleting an element from the summary does not change the bounds on the ranks of other elements in the summary.

The following claim shows that if a certain condition on r-min and r-max values of the elements in `WQS` is maintained, we can guarantee that `WQS` will be an ε-approximate summary of Unfold(S_w).

▷ Claim 4. Suppose in `WQS` over a length n stream, r-max$(e_i) -$ (r-min$(e_{i-1}) + w(e_{i-1}) - 1) \leq \lfloor \varepsilon W_n \rfloor$; then `WQS` is an ε-approximate quantile summary of Unfold(S_w).

The proof of this claim is presented in the full version.

2.1 Time Steps and Bands

Another important notion is that of *time steps* and *bands*. For simplicity, we define this for the unweighted setting, and then we build upon that for to define them for the weighted setting.

Unweighted Setting. We measure the time as the number of elements appeared in the stream so far in multiples of $\Theta(1/\varepsilon)$. Formally,

▶ **Definition 5** (Time Steps). *Let $\ell := \frac{1}{\varepsilon}$ which we assume is an integer. We partition the stream into consecutive **chunks** of size ℓ; the **time step** t then refers to the t-th chunk of elements denoted by $(x_1^{(t)}, \dots, x_\ell^{(t)})$ (we assume that the length of the stream is a multiple of ℓ)*[1]. *We define $t_0(x)$ as the time step in which x appears in the stream.*

The next important definitions are *band-values* and *bands* borrowed from [9]. Roughly speaking, we would like to be able to partition elements of the stream into a "small" number of groups (bands) so that elements within a group have "almost the same" time of insertion (as a proxy on how accurate our estimate of their r-min, r-max is). Formally,

▶ **Definition 6** (Band Values and Bands). *For any element x of the unweighted stream S, we assign an integer called a **band-value**, denoted by* $\text{b-value}(x)$*, as follows:*
- **(i)** *At the time step $t = t_0(x)$, we set $\text{b-value}(x) = 0$;*
- **(ii)** *At any time step $t > t_0(x)$, if t is a multiple of $2^{\text{b-value}(x)}$, then we increase $\text{b-value}(x)$ by one.*

Weighted Setting. We define an equivalent notion of time steps for weighted streams. We say that $t_k = \lfloor \varepsilon W_k \rfloor$ time steps have elapsed after the arrival of k elements in S_w. Intuitively, a chunk of total weight $\ell := \frac{1}{\varepsilon}$ arrives in the stream in a single time step. For each element x_k in S_w, we define its insertion time step $t_0(x_k) = \lfloor \varepsilon(W_{k-1} + 1) \rfloor$. The band value of an element x of S_w is the band value assigned to the first copy of x in $\mathsf{Unfold}(S_w)$ by Definition 6.

We would formally also have an equivalent definition of bands for the weighted setting that will allow us to compute them in $O(1)$ time. Their equivalence is proved in the full version of the paper.

▶ **Definition 7** (Band-Values and Bands). *When k elements of S_w have been inserted, for any element x of the stream,* $\text{b-value}(x)$ *is α if and only if the following inequality is satisfied,*

$$2^{\alpha-1} + (t_k \bmod 2^{\alpha-1}) \le t_k - t_0(x) < 2^{\alpha} + (t_k \bmod 2^{\alpha}).$$

*For any integer $\alpha \ge 0$, we refer to the set of all elements x with $\text{b-value}(x) = \alpha$ as the **band** α, denoted by Band_α; we also use $\text{Band}_{\le\alpha}$ to denote the union of bands 0 to α.*

A corollary of Definition 7 is that *number of band-values* after seeing k elements is $B^{(k)} = O(\log t_k) = O(\log(\varepsilon W_k))$. We also note that at any point, the sum of weights of all the elements belonging to bands 0 to α is at most $O(\ell \cdot 2^{\alpha+1})$ because all the copies of all these elements belong to $\text{Band}{\le}\ \alpha$ for $\mathsf{Unfold}(S_w)$. We note these facts below:

$$\#\text{ of b-values } B^{(k)} = O(\log \varepsilon W_k) \quad \text{and} \quad \sum_{x \in \text{Band}\le\alpha} w(x) \le O(\ell \cdot 2^{\alpha+1}) \text{ for all } \alpha \ge 0. \quad (2)$$

[1] Both assumptions in this definition are without loss of generality: we can change the value of ε by an $O(1)$ factor to guarantee the first one and add $O(1/\varepsilon)$ dummy elements at the end to guarantee the second one.

We now make the following observation:

▶ **Observation 8.** *At any point in time, if* b-value$(x) \le$ b-value(y) *for elements x and y, then at any point after this,* b-value$(x) \le$ b-value(y)*.*

This is simply because, in the unweighted setting, band-values of elements are updated *simultaneously* based on the value of the current time step (Definition 6). Thus, the b-value of the first copies of each element in a band is also updated simultaneously in $\mathsf{Unfold}(S_w)$. Thus, Observation 8 is true.

2.2 Indirect handling of r-min and r-max values

To describe our algorithm, it is better to store the r-min and r-max values indirectly as g and Δ values which we define for the weighted algorithm as follows. For any element e_i in `WQS`,

$$g_i = \text{r-min}(e_i) - (\text{r-min}(e_{i-1}) + w(e_{i-1}) - 1), \qquad \Delta_i = \text{r-max}(e_i) - \text{r-min}(e_i); \tag{3}$$

The g value can be interpreted to be the difference between the minimum possible rank of e_i and the minimum possible rank of the last copy of e_{i-1}. The Δ value is the difference between the r-max and r-min values of the first copy of e_i. The r-min and r-max values can be recovered given the g-values, Δ-values and the weights of all elements in `WQS` as follows:

$$\text{r-min}(e_i) = g_i + \sum_{j=1}^{i-1}(g_j + w(e_j) - 1), \qquad \text{r-max}(e_i) = \Delta_i + g_i + \sum_{j=1}^{i-1}(g_j + w(e_j) - 1).$$

This motivates the definition of the quantity G_i, for each element e_i in `WQS`:

$$G_i = g_i + w(e_i) - 1. \tag{4}$$

We will soon see that the G-value has a nice property that will prove useful in the analysis of the algorithms that we propose. We now use the g and Δ values defined to state the invariant that we maintain to ensure that `WQS` is an ε-approximate quantile summary.

▶ **Invariant 1.** *After seeing k elements of S_w, each element $e_i \in$ `WQS` satisfies $g_i + \Delta_i \le t_k$.*

From Equation (3) we note that $g_i + \Delta_i = \text{r-max}(e_i) - (\text{r-min}(e_{i-1}) + w(e_{i-1}) - 1)$. Also, $t_k = \lfloor \varepsilon W_k \rfloor$ by definition. Therefore, if `WQS` maintains Invariant 1, Claim 4 implies that it is an ε-approximate quantile summary of S_w.

The following observation now describes how g and Δ values of elements are updated during **Insert** and **Delete** operations (see Section 2).

▶ **Observation 9.** *In the summary* `WQS`*:*

- ***Insert$(x, w(x))$:*** *Sets $g(x) = 1$ and $\Delta(x) = g_i + \Delta_i - 1$ and keeps the remaining (g, Δ) values unchanged.*
- ***Delete(e_i):*** *Sets g_{i+1} to equal $g_{i+1} + G_i = g_{i+1} + (g_i + w(e_i) - 1)$, and keeps the remaining (g, Δ) values unchanged.*

The correctness of Observation 9 follows from Equation (3) and the way in which r-min and r-max values change when these operations are performed. As promised, we present useful properties of G and Δ values.

***G*-value.** To understand this, we define the notion of *coverage* of any element in WQS. We say that e_i *covers* e_{i-1} whenever e_{i-1} is deleted from the summary, in which case, e_i also covers all elements that e_{i-1} was covering so far (every element only covers itself upon insertion). We define:

- $C(e_i)$: the set of elements covered by e_i. By definition, at any point of time,

$$G_i = \sum_{x \in C(e_i)} w(x) \text{ and } C(e_i) \cap C(e_j) = \emptyset \tag{5}$$

  for any e_i, e_j currently stored in WQS.

We claim that G_i equals the sum of weights of the elements in the coverage of e_i. This is easy to verify by induction. When we insert an element, we set its g value to be 1 and the element only covers itself, thus its G value is equal to its weight by Equation (4). In the way G-value is updated upon deletions, according to Observation 9, this continues to be the case throughout the algorithm.

Δ-value. The Δ-value of an element is a measure of the error with which we know its rank. We can use Invariant 1 to deduce the following upper bound on the Δ value of an element x in terms of its insertion time $t_0(x)$.

$$\Delta(x) \leq t_0(x). \tag{6}$$

The intuition here is that after seeing k elements of the stream, the maximum possible difference in possible ranks is bounded by t_k if Invariant 1 is maintained. Hence, the error in the rank of a newly inserted element is also upper bounded by t_k. Formally, suppose that x is the j-th element of the stream and satisfies $e_{i-1} < x < e_i$ at the time of insertion into WQS. When x is inserted into WQS, we set $\Delta(x) = g_i + \Delta_i - 1$. Invariant 1 implies that $\Delta(x) \leq \lfloor \varepsilon W_{j-1} \rfloor - 1 \leq \lfloor \varepsilon (W_{j-1} + 1) \rfloor = t_0(x)$.

3 A non-trivial extension of GK algorithm for weighted streams

In this section, we present our extension of the GK algorithm for weighted streams. As a warm-up to our main algorithm, we explicitly analyze the special case of the algorithm when all weights are 1 (the unweighted setting) in Section 4.2. Along the way, we also present a very simple and greedy way of maintaining unweighted quantile summaries in $O(\frac{1}{\varepsilon} \cdot \log^2(\varepsilon n))$ space in Section 4.1. There, we shed further light on some counter-intuitive choices in GK summaries which turn out to be a basis for their tighter $O(\frac{1}{\varepsilon} \log(\varepsilon n))$ space. In particular, we motivate the following definition:

▶ **Definition 10** (Segment). *The **segment** of an element e_i in* WQS*, denoted by* seg(e_i)*, is defined as the maximal set of consecutive elements $e_j, e_{j+1}, \cdots, e_{i-1}$ in* WQS *with* b-value *strictly less than* b-value(e_i)*. We let G_i^* be the sum of the G-values of e_i and its segment, i.e., $G_i^* = G_i + \sum_{e_k \in \text{seg}(e_i)} G_k$.*

See Figure 2 below for an illustration.

At any step, the algorithm first inserts the arriving element into WQS; we call this the *insertion step.* It then only deletes an element from WQS if it can be deleted *together* with its entire segment without violating Invariant 1. While there is any such element whose deletion

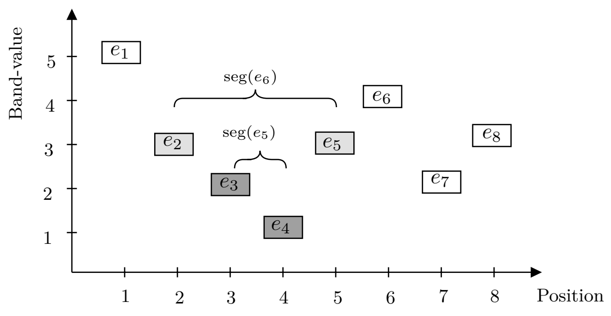


Figure 2 An illustration of Definition 10. The ranks of elements increase along the horizontal axis. The segment of the element e_5 contains e_3 and e_4. The segment of e_6 contains e_2, e_3, e_4 and e_5.

(along with its segment) does not violate Invariant 1 (and another simple but important condition on b-value), the algorithm deletes it from `WQS`; we call this the *deletion step.* We now give a formal description of the algorithm.

Algorithm 1. A generalization of the GK algorithm for weighted streams:

```
For each arriving item $(x_j, w(x_j))$:
    (i) Run Insert$(x_j, w(x_j))$:
    (ii) While there exists an element $e_i$ in WQS satisfying:

        (1) b-value$(e_i) \leq$ b-value$(e_{i+1})$    and    (2) $G_i^* + g_{i+1} + \Delta_{i+1} \leq t_j$

      Run Delete$(e_k)$ for $e_k$ in $\{e_i\} \cup \text{seg}(e_i)$.
```

Theorem 11. *For any $\varepsilon > 0$ and a weighted stream of length n with total weight W_n, Algorithm 1 maintains an ε-approximate quantile summary in $O(\frac{1}{\varepsilon} \cdot \log(\varepsilon W_n))$ space. Also, there is an implementation of Algorithm 1 that takes $O\big(\log(1/\varepsilon) + \log\log(\varepsilon W_n) + \frac{\log^2(\varepsilon W_n)}{\varepsilon n}\big)$ worst-case update time per element.*

We remark here that as long as the weights are poly(n) bounded and $\varepsilon \geq 1/n^{1-\delta}$ for any fixed $\delta \in (0,1)$, the space used by the algorithm will be $O((1/\varepsilon)\log(\varepsilon n))$ and its update time will be $O(\log(1/\varepsilon) + \log\log(\varepsilon n))$. This matches the space and time complexities of the implementation of the GK summary described in [13]. Note that the interesting regime for ε is at least a small constant, because when $\varepsilon < 1/n^{1-\delta}$, the information-theoretic lower bound of $(1/2\varepsilon)$ on the summary size already implies that we need to store $\Omega(n^{1-\delta})$ elements even for original GK summaries on unweighted inputs, which is prohibitive for most applications.

Algorithm 1 maintains a valid ε-approximate summary since it may only delete an element e_i along with its segment if the condition (ii): $G_i^* + g_{i+1} + \Delta_{i+1} \leq t_k$ is satisfied. Thus, Invariant 1 is satisfied for the element e_{i+1} after the deletion of e_i (other g and Δ values are unaffected by this). We now focus on bounding the space used by the algorithm in the following. Then, in Section 3.2, we give an efficient implementation to finalize the proof of Theorem 11.

3.1 Space Analysis

In this subsection, we prove a bound on the space used by Algorithm 1. Formally, we have the following:

▶ **Lemma 12.** *For any $\varepsilon > 0$ and a stream of length n with the total weight W_n, Algorithm 1 maintains an ε-approximate quantile summary in $O(\frac{1}{\varepsilon} \cdot \log(\varepsilon W_n))$ space.*

We first make a critical observation.

▶ **Observation 13.** *Elements from* $\text{Band}_{\leq\alpha}$ *in* `WQS` *only cover elements of* $\text{Band}_{\leq\alpha}$ *at any time.*

This is because when e_i and $\text{seg}(e_i)$ get covered by e_{i+1}, Algorithm 1 ensures that $\text{b-value}(e_i) \leq \text{b-value}(e_{i+1})$. From Definition 10, $\text{seg}(e_i)$ contains elements with b-value less than $\text{b-value}(e_i)$. Thus, $C(e_{i+1})$ contains elements with b-value at most $\text{b-value}(e_{i+1})$ and this continues to be the case at a later time by Observation 8.

Another important observation is that, after executing a deletion step after k insertions, an element e_i present in `WQS` either satisfies $\text{b-value}(e_i) > \text{b-value}(e_{i+1})$ or $G^*_i + g_{i+1} + \Delta_{i+1} > t_k$; otherwise Algorithm 1 would have deleted this element. We refer to the elements in `WQS` satisfying the former condition as *type-1* elements and the ones satisfying only the latter condition as *type-2* elements. Thus, each element is exactly one of the two types (except only $e_s = +\infty$ which we can ignore). It will therefore suffice to obtain a bound on the number of type-1 and type-2 elements to bound the space complexity of `WQS`. Let us first bound the number of type-1 elements in the following lemma.

▶ **Lemma 14.** *After the deletion step when k elements have been seen, the number of type-1 elements stored in* `WQS` *is $O(\ell \cdot \log t_k)$.*

Proof. We first partition the type-1 elements into $B^{(k)}$ sets $Y_0, \dots, Y_{B^{(k)}}$ where for any band-value α:

$$Y_\alpha := \{e_i \in \texttt{WQS} \mid e_i \text{ is type-1 and b-value}(e_{i+1}) = \alpha\}\,;$$

(notice that elements in Y_α are such that the band-value of their next element is α, not themselves[2]) We will show that the size of any set Y_α is at most $O(\ell)$. We map each element of e_i to the smallest element e_j with b-value greater than α; see Figure 3a for an illustration. Let T_α be the set of all such elements e_j. Also, it is easy to see that the mapping from Y_α to T_α is one to one; giving us $|Y_\alpha| = |T_\alpha|$. Note that e_{j-1} must be a type-2 element. Hence,

$$G^*_{j-1} + g_j + \Delta_j > t_k. \tag{7}$$

Since $\text{b-value}(e_j)$ is greater than $\text{b-value}(e_{i+1}) = \alpha$, by Observation 8, one can argue that e_j is inserted in `WQS` before e_{i+1}. Let g'_j be the g-value of e_j when e_{i+1} got inserted. By Invariant 1,

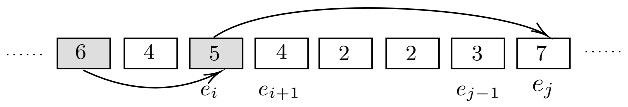
$$g'_j + \Delta_j \leq t_0(e_{i+1}) \quad (\Delta \text{ value does not change over time}). \tag{8}$$

Subtracting Equation (8) from Equation (7) and using the bounds from Definition 7 we conclude that,

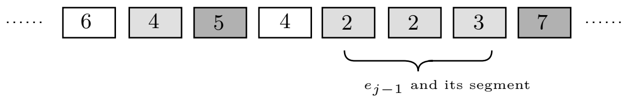
$$G^*_{j-1} + (g_j - g'_j) > t_k - t_0(e_{i+1}) \geq 2^{\alpha-1} - 2. \tag{9}$$

In the above equation:

[2] While this may sound counter-intuitive at first glance, recall that the criterion for defining the type of an element is a function of both this element and the next one; this definition allows us to take this into account.



(a) The shaded blocks are elements of Y_4. The arrows indicate the mapping from elements in Y_4 to elements in T_4. Each element e_i in Y_4 is mapped to the first larger element e_j with a band-value higher than 4.



(b) Each dark gray block represents an element e_j in T_4. All elements which are either e_{j-1} or are in the $\text{seg}(e_{j-1})$ are shaded light gray.

Figure 3 The two figures represent a section of the summary with each block representing an element. The number inside the block is the element's band-value.

(i) The term $(g_j - g'_j)$ counts the sum of weights of the elements covered by e_j after e_{i+1} is inserted. Claim 15 will show that these elements are in $\text{Band}_{\leq\alpha}$.

(ii) The term G^*_{j-1} counts the sum of the weights of the elements covered by e_{j-1} and $\text{seg}(e_{j-1})$. By Definition 10, e_{j-1} and all elements in $\text{seg}(e_{j-1})$ have b-value $\leq \alpha$. Observation 13 allows us to conclude that the sum of weights of elements counted by G^*_{j-1} are in $\text{Band}_{\leq\alpha}$ as well.

(iii) Additionally, it is easy to see that for distinct e_{j_1} and e_{j_2} in T_α, the segments of e_{j_1-1} and e_{j_2-1} do not overlap (as can be observed in Figure 3). Thus, by Equation (5), the elements covered by e_{j_1} and its segment are distinct from the elements covered by and e_{j_2} and its segment.

Finally, from the above discussion, we conclude that the LHS of Equation (9), summed over all T_α, is proportional to the total weight of all the elements in $\text{Band}_{\leq\alpha}$. Formally,

$$|T_\alpha| \cdot (2^{\alpha-1} - 2) \leq \sum_{e_j \in T_\alpha} G^*_{j-1} + \sum_{e_j \in T_\alpha} (g_j - g'_j) \leq \sum_{x_k \in \text{Band}_{\leq\alpha}} w(x_k) + \sum_{x_k \in \text{Band}_{\leq\alpha}} w(x_k) \leq O(\ell \cdot 2^{\alpha+2}),$$

where the last inequality follows from Equation (2). Hence, $|T_\alpha| = |Y_\alpha| = O(\ell)$ for $\alpha \geq 3$. We have $O(\ell)$ elements for $\alpha = 0, 1, 2$ anyway. Since there are $B^{(k)} = \log t_k$ possible values of α, the number of type-1 elements is $O(\ell \log t_k)$.

▷ Claim 15. All the elements covered by e_j after e_{i+1} was inserted have b-value at most α currently.

Proof. Let us assume that there exists an element which currently has b-value $> \alpha$ but gets covered by e_j after e_{i+1} was inserted. All such elements are less than e_j and greater than e_{i+1} since they get covered by e_j. Now consider the smallest such element e. Clearly, e_{i+1} belongs to the segment of e just after the insertion of e_{i+1}. Since e does not belong to the summary right now, it must have been deleted. This implies that its segment, which contained e_{i+1}, got deleted. This means e_{i+1} is also deleted, which is a contradiction. ◁

This finalizes the proof of Lemma 14. ◀

It now remains to bound the number of type-2 elements in `WQS` which we do in the following lemma.

▶ **Lemma 16.** *After the deletion step when k elements of the stream have been seen, the number of type-2 elements is $O(\ell \cdot \log t_k)$.*

Proof. Any type-2 element e_i in WQS, has the property that $G_i^* + g_{i+1} + \Delta_i > t_k$. This will give a lower bound on $G_i^* + g_{i+1}$ in terms of the b-value(e_{i+1}).

▷ Claim 17. After seeing k elements, for any type-2 element e_i, $G_i^* + g_{i+1} \geq 2^{\text{b-value}(e_{i+1})-1} - 2$.

Proof. As e_i is a type-2 element, $G_i^* + g_{i+1} + \Delta_{i+1} > t_k$. By Equation (6), $\Delta_{i+1} \leq t_0(e_{i+1})$ and therefore,

$$G_i^* + g_{i+1} > t_k - t_0(e_{i+1}) \geq 2^{\text{b-value}(e_{i+1})-1} - 2,$$

where the second inequality is by Definition 7. ◁

Claim 17 gives us a lower bound on the G^*-value of each type-2 element e_i as a function of the g-value and band-value of the *next* element e_{i+1}. Therefore, we partition the type-2 elements into sets $X_0, \ldots, X_{B^{(k)}}$ such that for any band-value α,

$$X_\alpha := \{e_i \in \texttt{WQS} \mid e_i \text{ is type-2 and b-value}(e_{i+1}) = \alpha\}.$$

Moreover, for any $e_i \in X_\alpha$, since e_i is a type-2 element, b-value$(e_i) \leq$ b-value$(e_{i+1}) = \alpha$. Summing over the inequality of Claim 17 for each element in X_α, we obtain:

$$|X_\alpha| \cdot (2^{\alpha-1} - 2) \leq \sum_{e_i \in X_\alpha} G_i^* + g_{i+1}. \tag{10}$$

We next show an upper bound on the right-hand side of Equation (10) which will imply the necessary bound on $|X_\alpha|$.

▷ Claim 18. After seeing k elements, for any $\alpha \geq 0$, $\sum_{e_i \in X_\alpha} G_i^* \leq 2 \sum_{e_j \in \texttt{WQS} \cap \text{Band}_{\leq \alpha}} G_j$.

Proof. We partition X_α into two disjoint sets $X_\alpha \cap \text{Band}_\alpha$ and $X_\alpha \cap \text{Band}_{\leq \alpha-1}$ and observe that two elements from one of these two sets must have disjoint segments. Also, the elements in their segments must all be in $\text{Band}_{\leq \alpha}$. Therefore,

$$\begin{aligned}\sum_{e_i \in X_\alpha} G_i^* &= \sum_{e_i \in X_\alpha \cap \text{Band}_\alpha} G_i^* + \sum_{e_i \in X_\alpha \cap \text{Band}_{\leq \alpha-1}} G_i^* \leq \sum_{e_j \in \texttt{WQS} \cap \text{Band}_{\leq \alpha}} G_j + \sum_{e_j \in \texttt{WQS} \cap \text{Band}_{\leq \alpha}} G_j. \\ &= 2 \sum_{e_j \in \texttt{WQS} \cap \text{Band}_{\leq \alpha}} G_j.\end{aligned}$$ ◁

The next claim bounds the sum of G-values of the elements in WQS from $\text{Band}_{\leq \alpha}$.

▷ Claim 19. After seeing k elements, for any $\alpha \geq 0$, $\sum_{e_i \in \texttt{WQS} \cap \text{Band}_{\leq \alpha}} G_i \leq O(\ell \cdot 2^{\alpha+1})$.

Proof. An element is only deleted by Algorithm 1 if condition (1) is satisfied. By Observation 8, this continues to be the case at any later point in the algorithm. Therefore, $C(e_i)$ only contains elements whose b-value is at most b-value(e_i). Therefore,

$$\begin{aligned}\sum_{e_i \in \texttt{WQS} \cap \text{Band}_{\leq \alpha}} G_i &= \sum_{e_i \in \texttt{WQS} \cap \text{Band}_{\leq \alpha}} \sum_{x_j \in C(e_i)} w(x_j) \\ &\qquad \text{(as } G_i = \textstyle\sum_{x_j \in C(e_i)} w(x_j) \text{ by Equation (5))} \\ &\leq \sum_{x_j \in \text{Band}_{\leq \alpha}} w(x_j) \\ &\qquad \text{(as } C(e_i)\text{'s are disjoint and their elements belong to Band} \leq \alpha) \\ &= O(\ell \cdot 2^{\alpha+1}), \qquad \text{(by the bound in Equation (2))}\end{aligned}$$

completing the argument. ◁

By plugging the bounds of Claim 18 and Claim 19 in Equation (10) and using the fact that a G value of an element is at least its g value, we have that,

$$|X_\alpha| \cdot (2^{\alpha-1} - 2) \leq \sum_{e_i \in X_\alpha} G_i^* + \sum_{e_j \in \texttt{WQS} \cap \text{Band}_{\leq \alpha}} g_j \leq 2 \sum_{e_j \in \texttt{WQS} \cap \text{Band}_{\leq \alpha}} G_j + \sum_{e_j \in \texttt{WQS} \cap \text{Band}_{\leq \alpha}} G_j \leq 3 \cdot O(\ell \cdot 2^{\alpha+1}),$$

which implies $|X_\alpha| = O(\ell)$ for $3 \leq \alpha \leq B^{(k)}$. There can be $O(\ell)$ elements each in X_0, X_1 and X_2 since there are at most $O(\ell)$ elements in $\text{Band}_{\leq 2}$. By Equation (2) we have $B^{(k)} = O(\log t_k)$ and therefore that the number of type-2 elements is $O(\ell \cdot \log t_k)$. ◀

We have now shown that, after performing the deletion step after k elements have been seen, the number of type-1 elements in `WQS` is $O(\ell \cdot \log t_k)$ by Lemma 14 and the number of type-2 elements in `WQS` is $O(\ell \cdot \log t_k)$ by Lemma 16. Since each element in `WQS` (other than $+\infty$) is either type-1 or type-2, the total number of elements in `WQS` is $O(\ell \cdot \log t_k)$.

This finalizes the proof of Lemma 12 since $t_n = O(\varepsilon W_n)$ and $\ell = O(\frac{1}{\varepsilon})$. We conclude the discussion of the space complexity with the following remark;

▶ Remark 20 (Delaying Deletions). Suppose in Algorithm 1, instead of running the **deletion step** in Line (ii) after each element, we run it only after inserting c elements $c > 1$; then, the space complexity of the algorithm only increases by an additive term $O(c)$.

Performing the deletion step after k elements, the number of elements reduces to $O(\ell \cdot \log t_k)$ as proved earlier as long as we have been satisfying both the conditions of the deletions of Algorithm 1 while performing every deletion. Thus, the extra space is only due to storing the additional $O(c)$ elements that are inserted in `WQS`.

The above remark will be useful in proposing an implementation of Algorithm 1 which has an asymptotically faster update time per element, which we do in the following.

3.2 An Efficient Implementation of Algorithm 1

In this section, we present an efficient implementation of Algorithm 1. This is similar to the implementation of the GK summary proposed in [13]. The key idea is that the deletion step is slow and therefore performing it after every time step is rather time inefficient. However, not performing the deletion step for too long blows up the space. The fast implementation we present deals with this trade-off and chooses the delay between consecutive deletion steps so that both the time and space complexity of the algorithm are optimized. Formally, we show the following:

▶ **Lemma 21.** *There is an implementation of Algorithm 1 that takes* $O\big(\log(1/\varepsilon) + \log\log(\varepsilon W_n) + \frac{\log^2(\varepsilon W_n)}{\varepsilon n}\big)$ *worst case processing time per element.*

Part I: Storing `WQS`

We store our summary `WQS` as a balanced binary search tree (BST), where each node contains an element of `WQS` along with its metadata. For each element e we store $w(e), g(e), \Delta(e)$ and $t_0(e)$. The sorting key of the BST is the value of elements. The **Insert** and **Delete** operations insert elements into and delete elements from the BST respectively.

Part II: Performing a Deletion Step

The deletion step involves the deletion of elements in the summary that satisfy the two conditions of Algorithm 1. Checking condition (ii) requires that we know the G^* values corresponding to each element of the summary, which we show how to do efficiently in the following.

Computing G^* values. First, we perform an inorder traversal of WQS and store the elements e_i in sorted order as a temporary linked list. The G^* value computation will use a stack and will make one pass over the list from the smallest to the largest element. We describe the computation when the traversal reaches the element e_i in the list. To obtain G_i^*, we sum up the G^* values of all elements on the top of the stack with b-value less than b-value(e_i) and add the sum to G_i. All these elements are popped from the stack and then e_i along with its computed G^* value is pushed onto the stack. We claim that at this point G_i^* has been correctly computed. Since each element is pushed and popped from the stack at most once, the G^* values of all elements can be computed in time linear in the size of WQS.

We now describe how each deletion step is performed.

▶ **Algorithm.** Performing a deletion step efficiently:

1. Perform an inorder traversal of WQS (which is a BST) to obtain a temporary (doubly-linked) list of elements sorted by value.
2. Compute b-values of all elements of WQS using Definition 7.
3. Compute the G^* value of all elements using the algorithm described above.
4. Traverse the list from larger elements to smaller ones. For each element e_i, delete it from BST (as well as the list), if it satisfies both the deletion conditions mentioned in Algorithm 1.

Having described an insertion step and a deletion step, below is an implementation of Algorithm 1 with fast amortized update time. We also describe how to modify this implementation to also get the same bound on the worst-case update time.

▶ **Implementation 1.** Efficient Implementation of Algorithm 1.

```
- Initialize WQS to be an empty balanced binary search tree.
- DeleteTime ← 2.
- For each arriving item $(x_k, w(x_k))$:
  (i)  Run **Insert**$(x_k, w(x_k))$.
  (ii) If $(k = \text{DeleteTime})$:
       - Execute the deletion step and update DeleteTime ← DeleteTime + $\lceil \ell \log t_k \rceil$.
```

Space Analysis

The space complexity of the above implementation is still $O(\frac{1}{\varepsilon}\log(\varepsilon W_n))$. This follows from the fact that, after performing a deletion when k elements have been seen, we wait for another $O(\ell \cdot \log t_k)$ elements only, which increases the space complexity by only a constant factor due to Remark 20. Thus, the space complexity , after n insertions, remains $O(\ell \cdot \log t_n) = O(\frac{1}{\varepsilon}\log(\varepsilon W_n))$, as $\ell = 1/\varepsilon$ and $t_n = O(\varepsilon W_n)$.

Time Analysis

The main purpose behind storing WQS as a BST was to decrease the time required to perform an **Insert** and **Delete** operation on WQS. This takes only $O(\log s)$, where s is the summary size which is at most $O(\frac{1}{\varepsilon}\log \varepsilon W_n)$. Thus, we now have the following observation, which is directly implied by the fact that we perform **Insert** and **Delete** at most once per element.

▶ **Observation 22.** *Over a stream of length n, the total time taken by the fast implementation of Implementation 1 to perform all* ***Insert*** *and* ***Delete*** *operations is $O(n \cdot (\log(1/\varepsilon) + \log\log(\varepsilon W_n)))$.*

Note that the only time taken by Implementation 1 **not** taken into account in Observation 22 is the part that determines *which elements* to delete, which we bound in the following.

▶ **Lemma 23.** *Over a stream of length n, the total time taken by Implementation 1 to decide which elements need to be deleted over all the executed deletion steps is $O(n + \frac{1}{\varepsilon}\log^2(\varepsilon W_n))$.*

Proof. The time taken to decide which elements need to be deleted inside one deletion step (when k elements have been seen) step is $O(s) = O(\ell \cdot \log t_k)$. This is because creating a linked list, followed by computation of b-value and G^*-value of all elements can be performed in $O(s)$ time, as discussed before. Finally, making a linear pass over the list from the largest to the smallest element (to check if the deletion conditions hold) requires $O(s)$ time.

Next, we obtain a bound on the number of deletion steps performed by the algorithm. Consider the deletion steps performed when t_k is the intervals $[2^i, 2^{i+1})$, for $1 \leq i \leq \lceil \log(\varepsilon W_n) \rceil$. Let $d(i)$ be the number of such deletion steps and $n(i)$ denote the number of elements x_k of the stream for which t_k is in the range $[2^i, 2^{i+1})$. After the deletion step when k elements have been seen, we wait for $\lceil \ell \log t_k \rceil$ insertions. Therefore, there are at least $\ell \cdot i$ elements inserted between two consecutive deletion steps that happen in the considered interval. Therefore, we get the following bound on the number of deletion steps that are performed during the interval.

$$d(i) \leq \frac{n(i)}{\ell \cdot i} + 1. \tag{11}$$

The time spent deciding which element to delete in a deletion step (after seeing k elements) is at most $O(\ell \log t_k) = O(\ell \cdot i)$, when t_k is in the interval $[2^i, 2^{i+1})$. This and Equation (11), give the following bound on the total time spent to decide which elements to delete over all deletions steps.

$$\begin{aligned} O\left(\sum_{i=1}^{\lceil \log(\varepsilon W_n) \rceil} d(i) \cdot \ell i\right) &= O\left(\sum_{i=1}^{\lceil \log(\varepsilon W_n) \rceil} (n(i) + \ell i)\right) \\ &= O\left(n + \frac{1}{\varepsilon}\log^2(\varepsilon W_n)\right) \end{aligned}$$

This finalizes the proof of the lemma. ◀

Observation 22 and Lemma 23 together clearly imply that the total time taken by Implementation 1 over a stream of length n is $O\big(\, n \cdot (\log(1/\varepsilon) + \log\log(\varepsilon W_n)) + \frac{1}{\varepsilon}\log^2(\varepsilon W_n) \,\big)$. Thus, the amortized update time per element is $O\big(\log(1/\varepsilon) + \log\log(\varepsilon W_n) + \frac{\log^2(\varepsilon W_n)}{\varepsilon n}\big)$.

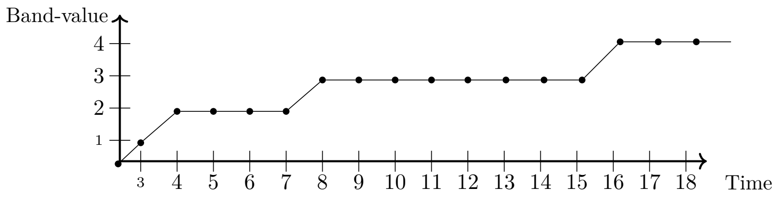
We can obtain the same bound on the worst-case update time per element using standard ideas of distributing time of inefficient operations over multiple time steps. The idea is to process the deletion step over all the following time steps before executing the next deletion step. Formally, we have the following:

▷ Claim 24. There is an implementation of Algorithm 1 with worst-case update time $O\big(\log(1/\varepsilon) + \log\log(\varepsilon W_n) + \frac{\log^2(\varepsilon W_n)}{\varepsilon n}\big)$.

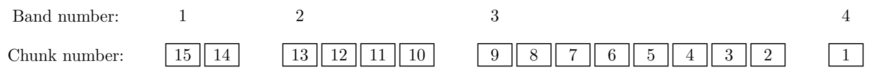
4 Unweighted Quantiles

For a definition of an unweighted quantile summary, see Definition 1. This is a special case of the weighted quantiles problem where each element of the stream arrives with a weight of 1. The GK-algorithm [9] solves this problem optimally [6] by proposing a summary of size $O\left(\frac{1}{\varepsilon}\log(\varepsilon n)\right)$. In the following sections, we attempt to simplify the GK algorithm while still being able to prove similar space guarantees.

Towards this end, we describe two algorithms. These algorithms also use the notion of time steps and bands which are formally defined in Section 2.1 (see Definition 5 and Definition 6). The n elements of the stream S are processed in $O(\varepsilon n)$ chunks each of size $\ell = O(1/\varepsilon)$. We refer to each such chunk of ℓ elements as a time step. (Contrast this with Algorithm 1 in the context of which a time step was defined to be a chunk of $O(1/\varepsilon)$ *weight*). When an element x of the stream arrives in the i-th time step, we set $t_0(x) = i$. Elements are further grouped geometrically based on what time step they appear in into $O(\log t)$ bands, where t is the current time-step (see Figure 4).



(a) Progression of the band-values of elements inserted at time step 2.



(b) Distribution of band values at $t = 15$.

Figure 4 An illustration of band-values and bands.

We use `QS` to represent the unweighted summary and e_i to be the i-th largest element stored in `QS`. For each element e_i, we wish to maintain lower and upper bounds on rank(e_i) denoted by r-min(e_i) and r-max(e_i) respectively. This is done implicitly by storing g_i and Δ_i values as described in Equation (3) in Section 2.2. Unlike the weighted setting, we do not need G_i values. (To see why, note that in Equation (4) when the weight of each element is 1, $G_i = g_i$.) Any summary which maintains the following invariant is guaranteed to be an ε-approximate quantile summary.

▶ **Invariant 2.** *After t times steps of the stream S, each element $e_i \in$ `QS` satisfies $g_i + \Delta_i \leq t$.*

Note that this is just a special case of Invariant 1 when all weights are one i.e. $W_n = n$. The goal of our algorithms is to maintain this invariant using limited space.

4.1 A Greedy $O\left(\frac{1}{\varepsilon}\log^2(\varepsilon n)\right)$ Size Summary

As a warm-up to our main algorithm, we first present a very simple and greedy way of updating the quantile summary `QS` to maintain Invariant 2 in $O(\frac{1}{\varepsilon} \cdot \log^2(\varepsilon n))$ space.

▶ **Algorithm 2.** A greedy algorithm for updating the quantile summary.

```
For each time step $t$ with arriving items $(x_1^{(t)}, \ldots, x_\ell^{(t)})$:
    (i) Run Insert$(x_j^{(t)})$ for each element of the chunk.
    (ii) Repeatedly run Delete$(e_i)$ for any (arbitrarily chosen) element $e_i$ in QS satisfying:
            (1) b-value$(e_i) \le$ b-value$(e_{i+1})$    and    (2) $g_i + g_{i+1} + \Delta_{i+1} \le t$
```

▶ **Theorem 25.** *For any $\varepsilon > 0$ and a stream of length n, Algorithm 2 maintains an ε-approximate quantile summary in $O(\frac{1}{\varepsilon} \cdot \log^2(\varepsilon n))$ space. Also, there is an implementation of Algorithm 2 that takes $O\big(\log(1/\varepsilon) + \log\log(\varepsilon n)\big)$ worst-case processing time per element. Finally, quantile queries can be answered in $O\big(\log(1/\varepsilon) + \log\log(\varepsilon n)\big)$ worst-case time per query.*

Algorithm 2 maintains Invariant 2 since it may only delete an element e_i if $g_i + g_{i+1} + \Delta_{i+1} \le t$, which then implies that $g_{i+1} + \Delta_{i+1} \le t$ *after* the deletion. The other (g, Δ)-values remain unchanged. As argued, maintaining Invariant 2 directly implies that QS is an ε-approximate quantile summary throughout the stream. Below we discuss some important insights on showing the space complexity of the algorithm. For formal proofs and its efficient implementation, we refer the reader to the full version of the paper.

After performing the deletion step at time t, we classify the elements into either *type-1* or *type-2*, and bound each one of them separately as we did previously. An element e_i present in QS satisfies b-value$(e_i) >$ b-value(e_{i+1}) then it is of type-1, or it must satisfy $g_i + g_{i+1} + \Delta_{i+1} > t$ and it is of type-2. We first bound the number of type-2 elements using a similar counting argument as in Section 3.1.

▶ **Lemma 26.** *After the deletion step at time step t, the number of type-2 elements stored in* QS *is $O(\ell \log t)$.*

A more interesting is the bound on the number of type-2 elements. Observe that we do not delete an element with its segment (unlike Algorithm 1), which was crucial in proving a bound on type-1 element. However, as we show, the number of type-1 elements cannot be much larger than the type-2 ones even for this deletion strategy. This is simply because the band-values of consecutive type-1 elements strictly decrease from one element to the next and thus we cannot have many type-1 elements next to each other.

▶ **Lemma 27.** *After the deletion step at time step t, the number of type-1 elements stored in* QS *is $O(\log t)$ times larger than the type-2 elements.*

▶ Remark 28. *In the space analysis, we bounded the number of type-2 elements in the summary after the deletion step by $O(\ell \cdot \log t) = O((1/\varepsilon) \cdot \log(\varepsilon n))$, which is quite efficient on is own. However, in the worst case, there can be $O(\log t)$ type-1 elements for every type-2 element as shown in Figure 5. Thus, Algorithm 2 may end up storing as many as $O(\ell \cdot \log^2 t) = O((1/\varepsilon) \cdot \log^2(\varepsilon n))$ type-1 elements in the summary, leading to its sub-optimal space requirement.*

4.2 The Simplified GK $O(\frac{1}{\varepsilon} \cdot \log(\varepsilon n))$ Size Summary

We give our description of GK summaries. As we say in Remark 28, one source of sub-optimality of Algorithm 2 was a large number of type-1 elements stored in the summary compared to the type-2 ones. A way to improve this is to *actively* try to decrease the number

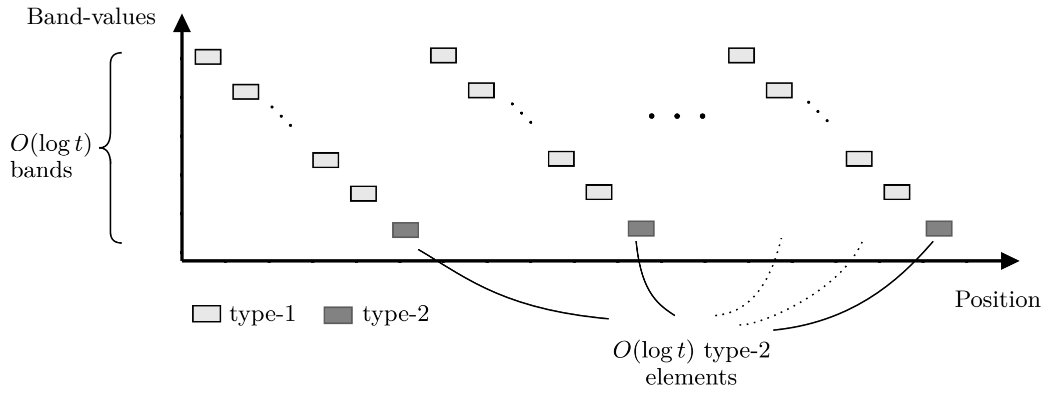


Figure 5 Each block in the figure represents an element stored in QS. The ranks of elements increase along the horizontal axis. The figure illustrates why Algorithm 2 might end up storing $O(\ell \log^2 t)$ elements in QS. By Lemma 26, there could be as many as $O(\ell \cdot \log t)$ type-2 elements in QS. Each of these type-2 elements could be preceded by a sequence of $O(\log t)$ type-1 elements (since there are $O(\log t)$ bands).

of stored type-1 elements. Roughly speaking, this is done by deleting type-2 elements from the summary only if it does not contribute to creating a long sequence of type-1 elements (e.g., as in Figure 5). Roughly speaking, while there is an element whose deletion *together* with its entire segment (Definition 10) doesn't violate Invariant 2 (and the same condition on b-values), we delete the element and its entire segment. Let g_i^* denote the sum of g-values of the elements in $\text{seg}(e_i)$. Below is a formal description of the algorithm and the theorem, whose proof we include in the full version.

▶ **Algorithm 3.** An improved algorithm for updating the quantile summary.

For each time step t with arriving items $(x_1^{(t)}, \ldots, x_\ell^{(t)})$:

(i) Run **Insert**$(x_j^{(t)})$ for each element of the chunk.

(ii) While there exists an element e_i in QS satisfying:

$$(1)\ \text{b-value}(e_i) \le \text{b-value}(e_{i+1}) \quad \underline{\text{and}} \quad (2)\ g_i^* + g_{i+1} + \Delta_{i+1} \le t$$

Run **Delete**(e_k) for e_k in $\{e_i\} \cup \text{seg}(e_i)$.

▶ **Theorem 29.** *For any $\varepsilon > 0$ and a stream of length n, Algorithm 3 maintains an ε-approximate quantile summary in $O(\frac{1}{\varepsilon} \cdot \log(\varepsilon n))$ space. Also, there is an implementation of Algorithm 3 that takes $O(\log(\frac{1}{\varepsilon}) + \log\log(\varepsilon n))$ worst case update time per element.*

We shall note that even though our description of Algorithm 3 varies from the presentation of GK summaries in [9], the two algorithms behave in an almost identical way.

References

1 Pankaj K. Agarwal, Graham Cormode, Zengfeng Huang, Jeff M. Phillips, Zhewei Wei, and Ke Yi. Mergeable summaries. In *Proceedings of the 31st ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2012, Scottsdale, AZ, USA, May 20-24, 2012*, 2012.

2 Noga Alon, Omri Ben-Eliezer, Yuval Dagan, Shay Moran, Moni Naor, and Eylon Yogev. Adversarial laws of large numbers and optimal regret in online classification. In *Proceedings of the 53rd Annual ACM SIGACT Symposium on Theory of Computing*, pages 447–455, 2021.

3 Noga Alon, Yossi Matias, and Mario Szegedy. The space complexity of approximating the frequency moments. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing, Philadelphia, Pennsylvania, USA, May 22-24, 1996*, pages 20–29, 1996.

4 Tianqi Chen and Carlos Guestrin. XGBoost. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, August 2016. doi:10.1145/2939672.2939785.

5 Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, pages 785–794, 2016.

6 Graham Cormode and Pavel Veselý. A tight lower bound for comparison-based quantile summaries. In *Proceedings of the 39th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2020, Portland, OR, USA, June 14-19, 2020*, pages 81–93, 2020.

7 Anna C Gilbert, Brett Hemenway, Atri Rudra, Martin J Strauss, and Mary Wootters. Recovering simple signals. In *2012 Information Theory and Applications Workshop*, pages 382–391. IEEE, 2012.

8 Anna C Gilbert, Brett Hemenway, Martin J Strauss, David P Woodruff, and Mary Wootters. Reusable low-error compressive sampling schemes through privacy. In *2012 IEEE Statistical Signal Processing Workshop (SSP)*, pages 536–539. IEEE, 2012.

9 Michael Greenwald and Sanjeev Khanna. Space-efficient online computation of quantile summaries. In *Proceedings of the 2001 ACM SIGMOD international conference on Management of data, Santa Barbara, CA, USA, May 21-24, 2001*, pages 58–66, 2001.

10 Moritz Hardt and David P Woodruff. How robust are linear sketches to adaptive inputs? In *Proceedings of the forty-fifth annual ACM symposium on Theory of computing*, pages 121–130, 2013.

11 Regant Y. S. Hung and Hing-Fung Ting. An $\Omega\left(\frac{1}{\varepsilon}\log\frac{1}{\varepsilon}\right)$ space lower bound for finding ε-approximate quantiles in a data stream. In *Frontiers in Algorithmics, 4th International Workshop, FAW 2010, Wuhan, China, August 11-13, 2010. Proceedings*, pages 89–100, 2010.

12 Zohar S. Karnin, Kevin J. Lang, and Edo Liberty. Optimal quantile approximation in streams. In *IEEE 57th Annual Symposium on Foundations of Computer Science, FOCS 2016, 9-11 October 2016, Hyatt Regency, New Brunswick, New Jersey, USA*, pages 71–78, 2016.

13 Ge Luo, Lu Wang, Ke Yi, and Graham Cormode. Quantiles over data streams: experimental comparisons, new analyses, and further improvements. *VLDB J.*, 25(4):449–472, 2016.

14 Gurmeet Singh Manku, Sridhar Rajagopalan, and Bruce G. Lindsay. Approximate medians and other quantiles in one pass and with limited memory. In *SIGMOD 1998, Proceedings ACM SIGMOD International Conference on Management of Data, June 2-4, 1998, Seattle, Washington, USA*, pages 426–435, 1998.

15 Gurmeet Singh Manku, Sridhar Rajagopalan, and Bruce G. Lindsay. Random sampling techniques for space efficient online computation of order statistics of large datasets. In *SIGMOD 1999, Proceedings ACM SIGMOD International Conference on Management of Data, June 1-3, 1999, Philadelphia, Pennsylvania, USA*, pages 251–262, 1999.

16 Ilya Mironov, Moni Naor, and Gil Segev. Sketching in adversarial environments. *SIAM Journal on Computing*, 40(6):1845–1870, 2011.

17 J. Ian Munro and Mike Paterson. Selection and sorting with limited storage. In *19th Annual Symposium on Foundations of Computer Science, Ann Arbor, Michigan, USA, 16-18 October 1978*, pages 253–258, 1978.

18 Moni Naor and Eylon Yogev. Bloom filters in adversarial environments. In *Annual Cryptology Conference*, pages 565–584. Springer, 2015.

19 List of open problems in sublinear algorithms – problem 2: Quantiles. https://sublinear.info/2.

Probabilistic Query Evaluation with Bag Semantics

Martin Grohe
RWTH Aachen University, Germany

Peter Lindner
École Polytechnique Fédérale de Lausanne (EPFL), Switzerland

Christoph Standke
RWTH Aachen University, Germany

Abstract

We initiate the study of probabilistic query evaluation under *bag semantics* where tuples are allowed to be present with duplicates. We focus on self-join free conjunctive queries, and probabilistic databases where occurrences of different facts are independent, which is the natural generalization of tuple-independent probabilistic databases to the bag semantics setting. For set semantics, the data complexity of this problem is well understood, even for the more general class of unions of conjunctive queries: it is either in polynomial time, or ♯P-hard, depending on the query (Dalvi & Suciu, JACM 2012).

Due to potentially unbounded multiplicities, the bag probabilistic databases we discuss are no longer finite objects, which requires a treatment of representation mechanisms. Moreover, the answer to a Boolean query is a probability distribution over non-negative integers, rather than a probability distribution over $\{\mathsf{true}, \mathsf{false}\}$. Therefore, we discuss two flavors of probabilistic query evaluation: computing expectations of answer tuple multiplicities, and computing the probability that a tuple is contained in the answer at most k times for some parameter k. Subject to mild technical assumptions on the representation systems, it turns out that expectations are easy to compute, even for unions of conjunctive queries. For query answer probabilities, we obtain a dichotomy between solvability in polynomial time and ♯P-hardness for self-join free conjunctive queries.

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

Keywords and phrases Probabilistic Query Evaluation, Probabilistic Databases, Bag Semantics

Digital Object Identifier 10.4230/LIPIcs.ICDT.2023.20

Related Version *Extended Version*: `https://arxiv.org/abs/2201.11524` [20]

Funding The work of Martin Grohe and Christoph Standke has been funded by the German Research Foundation (DFG) under grants GR 1492/16-1 and GRK 2236 (UnRAVeL).

1 Introduction

Probabilistic databases (PDBs) provide a framework for managing uncertain data. In database theory, they have been intensely studied since the late 1990s [30, 31]. Most efforts have been directed towards tuple-independent relational databases *under a set semantics*. Many relational database systems, however, use a bag semantics, where identical tuples may appear several times in the same relation. Despite receiving little attention so far, bag semantics are also a natural setting for probabilistic databases. For example, they naturally enter the picture when aggregation is performed, or when statistics are computed (e.g., by random sampling, say, without replacement). Either case might involve computing projections without duplicate elimination first. Even when starting from a tuple-independent probabilistic database with set semantics, this typically gives rise to (proper) bags. A bag semantics for PDBs has only been considered recently [19, 18], in the context of infinite

PDBs. Even in the traditional setting of a PDB where only finitely many facts appear with non-zero probability, under a bag semantics we have to consider infinite probability spaces, simply because there is no a priori bound on the number of times a fact may appear in a bag. In general, while the complexity landscape of query answering is well understood for simple models of PDBs under set semantics, the picture for *bag semantics* is still unexplored. In this work, we take first steps to address this.

Formally, probabilistic databases are probability distributions over conventional database instances. In a database instance, the answer to a Boolean query under *set semantics* is either true (1) or false (0). In a probabilistic database, the answer to such a query becomes a $\{0, 1\}$-valued random variable. The problem of interest is *probabilistic query evaluation*, that is, computing the probability that a Boolean query returns true, when given a probabilistic database. The restriction to *Boolean* queries comes with no loss of generality: To compute the probability of any tuple in the result of a non-Boolean query, all we have to do is replace the free variables of the query according to the target tuple, and solve the problem for the resulting Boolean query.

Under a bag semantics, a Boolean query is still just a query without free variables, but the answer to Boolean query can be any non-negative integer, which can be interpreted as the multiplicity of the empty tuple in the query answer, or more intuitively as the number of different ways in which the query is satisfied. In probabilistic query evaluation, we then get $\mathbb{N}$-valued answer random variables. Still, the reduction from the non-Boolean to the Boolean case works as described above. *Therefore, without loss of generality, we only discuss Boolean queries in this paper.*

As most of the database theory literature, we study the *data complexity* of query evaluation [32], that is, the complexity of the problem, when the query Q is fixed, and the PDB is the input. The standard model for complexity theoretic investigations is that of *tuple-independent* PDBs, where the distinct facts constitute independent events. Probabilistic query evaluation is well-understood for the class of *unions of conjunctive queries* (UCQs) on PDBs that are tuple-independent (see the related works section below). However, all existing results discuss the problem under set semantics. Here, on the contrary, we discuss the probabilistic query evaluation under *bag semantics.*

For tuple-independent (set) PDBs, a variety of representation systems have been proposed (cf. [17, 30]), although for complexity theoretic discussions, it is usually assumed that the input is just given as a table of facts, together with their marginal probabilities [31]. In the bag version of tuple-independent PDBs [18], different facts are still independent. Yet, the individual facts (or, rather, their multiplicities) are $\mathbb{N}$-valued, instead of Boolean, random variables. As this, in general, rules out the naive representation through a list of facts, multiplicities, and probabilities, it is necessary to first define suitable representation systems before the complexity of computational problems can be discussed.

Once we have settled on a suitable class of representations, we investigate the problem of probabilistic query evaluation again, subject to representation system $\mathcal{R}$. Under bag semantics, there are now *two* natural computational problems regarding query evaluation: $\mathsf{EXPECTATION}_{\mathcal{R}}(Q)$, which is computing the expected outcome, and $\mathsf{PQE}_{\mathcal{R}}(Q, k)$ which is computing the probability that the outcome is at most k. Notably, these two problems coincide for set semantics, because the expected value of a $\{0, 1\}$-valued random variables coincides with the probability that the outcome is 1. Under a bag semantics, however, the two versions exhibit quite different properties.

Recall that using a set semantics, unions of conjunctive queries can either be answered in polynomial time, or are $\sharp\mathsf{P}$-hard [12]. Interestingly, computing expectations using a bag semantics is extraordinarily easy in comparison: With only mild assumptions on the representation, the expectation of any UCQ can be computed in polynomial time. Furthermore, the variance of the random variable can also be computed in polynomial time, which via Chebyshev's inequality gives us a way to estimate the probability that the query answer is close to its expectation. These results contrast the usual landscape of computational problems in uncertain data management, which are rarely solvable efficiently.

The computation of probabilities of concrete answer multiplicities, however, appears to be less accessible, and in fact, in its properties is more similar to the set semantics version of probabilistic query evaluation. Our main result states that for Boolean conjunctive queries without self-joins, we have a dichotomy between polynomial time and $\sharp$P-hardness of the query. This holds whenever efficient access to fact probabilities is guaranteed by the representation system and is independent of k. Although the proof builds upon ideas and notions introduced for the set semantics dichotomy [9, 10, 12], we are confronted with a number of completely new and intricate technical challenges due to the change of semantics. On the one hand, the bag semantics turns disjunctions and existential quantification into sums. This facilitates the computation of expected values, because it allows us to exploit linearity. On the other hand, the new semantics keep us from directly applying some of the central ideas from [12] when analyzing $\mathsf{PQE}_{\mathcal{R}}(Q,k)$, thus necessitating novel techniques. The bag semantics dichotomy for answer count probabilities is, hence, far from being a simple corollary from the set semantics dichotomy. From the technical perspective, the most interesting result is the transfer of hardness from $\mathsf{PQE}_{\mathcal{R}}(Q,0)$ to $\mathsf{PQE}_{\mathcal{R}}(Q,k)$. In essence, we need to find a way to compute the probability that Q has 0 answers, with only having access to the probability that Q has at most k answers for any single fixed k. This reduction uses new non-trivial techniques: By manipulating the input table, we can construct multiple instances of the $\mathsf{PQE}_{\mathcal{R}}(Q,k)$ problem. We then transform the solutions to these problems, which are obtained through oracle calls, into function values of a polynomial (with a priori unknown coefficients) in such a way, that the solution to $\mathsf{PQE}_{\mathcal{R}}(Q,0)$ on the original input is hidden in the leading coefficient of this polynomial. Using a technique from polynomial interpolation, we can find these leading coefficients, and hence, solve $\mathsf{PQE}_{\mathcal{R}}(Q,0)$.

Related Work

The most prominent result regarding probabilistic query evaluation is the Dichotomy theorem by Dalvi and Suciu [12] that provides a separation between unions of conjunctive queries for which probabilistic evaluation is possible in polynomial time, and such where the problem becomes $\sharp$P-hard. They started their investigations with self-join free conjunctive queries [10] and later extended their results to general CQs [11] and then UCQs [12]. Beyond the queries they investigate, there are a few similar results for fragments with negations or inequalities [14, 26, 27], for homomorphism-closed queries [4] and others [28], and on restricted classes of PDBs [1]. Good overviews over related results are given in [31, 29]. In recent developments, the original dichotomies for self-join free CQs, and for general UCQs have been shown to hold even under severe restrictions to the fact probabilities that are allowed to appear [2, 23].

The bag semantics for CQs we use here is introduced in [7]. A detailed analysis of the interplay of bag and set semantics is presented in [8]. Considering multiplicities as semi-ring annotations [16, 21], embeds bag semantics into a broader mathematical framework.

In recent work (independent of ours), Feng et al. [13] analyze the fine-grained complexity of computing expectations of queries in probabilistic bag databases, albeit assuming finite multiplicity supports and hence still in the realm of finite probabilistic databases.

2 Preliminaries

We denote by $\mathbb{N}$ and $\mathbb{N}_+$ the sets of non-negative, and of positive integers, respectively. We denote open, closed and half open intervals of real numbers by (a,b), $[a,b]$, $[a,b)$ and $(a,b]$, respectively, where $a \leq b$. By $\binom{n}{k}$ we denote the binomial coefficient and by $\binom{n}{n_1,\dots,n_k}$ the multinomial coefficient.

Let Ω be a non-empty finite or countably infinite set and let $P\colon \Omega \to [0,1]$ be a function satisfying $\sum_{\omega\in\Omega} P(\omega) = 1$. Then (Ω, P) is a *(discrete) probability space.* Subsets $A \subseteq \Omega$ are called *events.* We write $\Pr_{\omega\sim\Omega}(\omega \in A)$ for the probability of a randomly drawn $\omega \in \Omega$ (distributed according to P) to be in A. More generally, we may write $\Pr_{\omega\in\Omega}(\omega \text{ has property } \varphi)$ for the probability of a randomly drawn element to satisfy some property φ. *All probability spaces appearing in this paper are discrete.*

Functions $X\colon \Omega \to \mathbb{R}$ on a probability space are called *random variables.* The expected value and variance of X are denoted by $\mathbb{E}(X)$ and $\operatorname{Var}(X)$, respectively. The values $\mathbb{E}(X^k)$ for integers $k \geq 2$ are called the higher-order *moments* of X.

2.1 Probabilistic Bag Databases

We fix a countable, non-empty set dom (the *domain*). A *database schema* τ is a finite, non-empty set of relation symbols. Every relation symbol R has an arity $\operatorname{ar}(R) \in \mathbb{N}_+$.

A *fact* over τ and dom is an expression $R(\mathbf{a})$ where $\mathbf{a} \in \mathrm{dom}^{\operatorname{ar}(R)}$. A *(bag) database instance* D is a bag (i.e. multiset) of facts. Formally, a bag (instance) is specified by a function $\sharp_D$ that maps every fact f to its multiplicity $\sharp_D(f)$ in D. The *active domain* $\operatorname{adom}(D)$ is the set of domain elements a from dom for which there exists a fact f containing a such that $\sharp_D(f) > 0$.

A *probabilistic (bag) database* (or, *(bag) PDB*) $\mathcal{D}$ is a pair $(\mathbb{D}, P)$ where $\mathbb{D}$ is a set of bag instances and $P\colon 2^{\mathbb{D}} \to [0,1]$ is a probability distribution over $\mathbb{D}$. Note that, even when the total number of different facts is finite, $\mathbb{D}$ may be infinite, as facts may have arbitrarily large multiplicities. We let $\sharp_{\mathcal{D}}(f)$ denote the random variable $D \mapsto \sharp_D(f)$ for all facts f. If $\mathcal{D} = (\mathbb{D}, P)$ is a PDB, then $\operatorname{adom}(\mathcal{D}) := \bigcup_{D\in\mathbb{D}} \operatorname{adom}(D)$. We call a PDB *fact-finite* if the set $\{f\colon \sharp_D(f) > 0 \text{ for some } D \in \mathbb{D}\}$ is finite. In this case, $\operatorname{adom}(\mathcal{D})$ is finite, too.

A bag PDB $\mathcal{D}$ is called *tuple-independent* if for all $k \in \mathbb{N}$, all pairwise distinct facts $f_1,\dots,f_k$, and all $n_1,\dots,n_k \in \mathbb{N}$, the events $\sharp_D(f_i) = n_i$ are independent, i. e.,

$$\Pr_{D\sim\mathcal{D}}\bigl(\sharp_D(f_i) = n_i \text{ for all } i = 1,\dots,k\bigr) = \prod_{i=1}^{k} \Pr_{D\sim\mathcal{D}}\bigl(\sharp_D(f_i) = n_i\bigr).$$

Unless it is stated otherwise, all probabilistic databases we treat in this paper are assumed to be fact-finite and tuple-independent.

2.2 UCQs with Bag Semantics

Let $\mathcal{V}$ be a countably infinite set of variables. An *atom* is an expression of the shape $R(\mathbf{t})$ where $R \in \tau$ and $\mathbf{t} \in (\mathrm{dom} \cup \mathcal{V})^{\operatorname{ar}(R)}$. A *conjunctive query (CQ)* is a formula Q of first-order logic (over τ and dom) of the shape

$$Q = \exists x_1 \dots \exists x_m\colon R_1(\mathbf{t}_1) \wedge \dots \wedge R_n(\mathbf{t}_n),$$

in which we always assume that the x_i are pairwise different, and that x_i appears in at least one of $\mathbf{t}_1,\dots,\mathbf{t}_n$ for all $i = 1,\dots,m$. A CQ Q is *self-join free*, if every relation symbol occurs at most once within Q. In general, the *self-join width* of a CQ Q is the maximum number of repetitions of the same relation symbol in Q. If Q is a CQ of the above shape, we let Q^* denote the quantifier-free part $R_1(\mathbf{t}_1) \wedge \dots R_n(\mathbf{t}_n)$ of Q, and we call $R_i(\mathbf{t}_i)$ an *atom of* Q for all $i = 1, \dots, n$. A *union of conjunctive queries (UCQ)* is a formula of the shape $Q = Q_1 \vee \dots \vee Q_N$ where $Q_1, \dots, Q_N$ are CQs. A query is called *Boolean*, if it contains no free variables (that is, there are no occurrences of variables that are not bound by a quantifier). *From now on, and throughout the remainder of the paper, we only discuss Boolean (U)CQs.*

Recall that $\sharp_D$ is the multiplicity function of the instance D. The bag semantics of (U)CQs extends $\sharp_D$ to queries. For Boolean CQs $Q = \exists x_1 \dots \exists x_m \colon R_1(\mathbf{t}_1) \wedge \dots \wedge R_n(\mathbf{t}_n)$ we define

$$\sharp_D(Q) \coloneqq \sum_{\mathbf{a} \in \operatorname{adom}(D)^m} \prod_{i=1}^{n} \sharp_D\big(R_i(\mathbf{t}_i[\mathbf{x}/\mathbf{a}])\big), \tag{1}$$

where $\mathbf{x} = (x_1, \dots, x_m)$ and $\mathbf{a} = (a_1, \dots, a_m)$, and $R_i(\mathbf{t}_i[\mathbf{x}/\mathbf{a}])$ denotes the fact obtained from $R_i(\mathbf{t}_i)$ by replacing, for all $j = 1, \dots, m$, every occurrence of x_j by a_j. If $Q = Q_1 \vee \dots \vee Q_N$ is a Boolean UCQ, then each of the Q_i is a Boolean CQ. We define

$$\sharp_D(Q) \coloneqq \sharp_D(Q_1) + \dots + \sharp_D(Q_N). \tag{2}$$

Whenever convenient, we write $\sharp_D Q$ instead of $\sharp_D(Q)$. We emphasize once more, that the query being Boolean *does not* mean that its answer is 0 or 1 under bag semantics, but could be any non-negative integer.

▶ Remark 2.1. We point out that in (1), conjunctions should intuitively be understood as joins rather than intersections. Our definition (1) for the bag semantics of CQs matches the one that was given in [7]. This, and the extension (2) for UCQs, are essentially special cases of how semiring annotations of formulae are introduced in the provenance semiring framework [16, 21], the only difference being that we use the active domain semantics. For UCQs however, this is equivalent since the value of (1) stays the same when the quantifiers range over arbitrary supersets of adom(D). ⌟

Note that the result $\sharp_D Q$ of a Boolean UCQ on a bag instance D is a non-negative integer. Thus, evaluated over a PDB $\mathcal{D} = (\mathbb{D}, P)$, this yields a $\mathbb{N}$-valued random variable $\sharp_{\mathcal{D}} Q$ with

$$\Pr\big(\sharp_{\mathcal{D}} Q = k\big) = \Pr_{D \sim \mathcal{D}}\big(\sharp_D Q = k\big).$$

▶ **Example 2.2.** Consider the tuple-independent bag PDB over facts $R(a)$ and $S(a)$, where $R(a)$ has multiplicity 2 or 3, both with probability $\frac{1}{2}$, and $S(a)$ has multiplicity 1, 2 or 3, with probability $\frac{1}{3}$ each. Then, the probability of the event $\sharp_{\mathcal{D}}(R(a) \wedge S(a)) = 6$ is given by $\Pr\big(\sharp_{\mathcal{D}}(R(a)) = 2\big) \Pr\big(\sharp_{\mathcal{D}}(S(a)) = 3\big) + \Pr\big(\sharp_{\mathcal{D}}(R(a)) = 3\big) \Pr\big(\sharp_{\mathcal{D}}(S(a)) = 2\big) = \frac{1}{3}$. ⌟

There are now two straight-forward ways to formulate the problem of answering a Boolean UCQ over a probabilistic database. We could either ask for the expectation $\mathbb{E}\big(\sharp_{\mathcal{D}} Q\big)$, or compute the probability that $\sharp_{\mathcal{D}} Q$ is at most / at least / equal to k. These two options coincide for set semantics, as $\sharp_{\mathcal{D}} Q$ is $\{0, 1\}$-valued in this setting.[1] For bag PDBs, these are two separate problems to explore. Complexity-wise, we focus on *data complexity* [32]. That is, the query (and for the second option, additionally the number k) is a parameter of the problem, so that the input is only the PDB. Before we can start working on these problems, we first need to discuss how bag PDBs are presented as an input to an algorithm. This is the purpose of the next section.

[1] In fact, in the literature both approaches have been used to introduce the problem of probabilistic query evaluation [30, 31].

Relation R	Parameter
$R(1,1)$	$(\texttt{Bernoulli}, 1/2)$
$R(1,2)$	$(\texttt{Binomial}, 10, 1/3)$
$R(2,2)$	$(0 \mapsto 1/4; 1 \mapsto 1/4; 5 \mapsto 1/2)$

Relation S	Parameter
$S(1)$	$(\texttt{Geometric}, 1/3)$
$S(2)$	$(\texttt{Poisson}, 3)$

Figure 1 Example of a parameterized TI representation.

3 Representation Systems

For the set version of probabilistic query evaluation, the default representation system represents tuple-independent PDBs by specifying all facts together with their marginal probability. The distinction between a PDB and its representation is then usually blurred in the literature. This does not easily extend to bag PDBs, as the distributions of $\sharp_D(f)$ for facts f may have infinite support.

▶ **Example 3.1.** Let $\mathcal{D} = (\mathbb{D}, P)$ be a bag PDB over a single fact f with multiplicity distribution $\sharp_\mathcal{D}(f) \sim \text{Geometric}\left(\frac{1}{2}\right)$, i.e., $\Pr_{D\sim\mathcal{D}}\left(\sharp_D(f) = k\right) = 2^{-k}$. Then the instances of $\mathcal{D}$ with positive probability are $\{\!\{\}\!\}, \{\!\{f\}\!\}, \{\!\{f,f\}\!\}, \dots$, so $\mathcal{D}$ is an infinite PDB. ⌟

To use such PDBs as inputs for algorithms, we introduce a suitable class of representation systems (RS) [17]. All computational problems are then stated with respect to an RS.

▶ **Definition 3.2** (cf. [17]). A *representation system (RS)* for bag PDBs is a pair $(\mathbf{T}, [\![\,\cdot\,]\!])$ where $\mathbf{T}$ is a non-empty set (the elements of which we call *tables*), and $[\![\,\cdot\,]\!]$ is a function that maps every $T \in \mathbf{T}$ to a probabilistic database $[\![T]\!]$. ⌟

Given an RS, we abuse notation and also use T to refer to the PDB $[\![T]\!]$. Note that Definition 3.2 is not tailored to tuple-independence yet and requires no independence assumptions. For representing tuple-independent bag PDBs, we introduce a particular subclass of RS's where facts are labeled with the parameters of parameterized distributions over multiplicities. For example, a fact f whose multiplicity is geometrically distributed with parameter $\frac{1}{2}$ could be annotated with $(\texttt{Geometric}, 1/2)$, representing $\frac{1}{2}$ using two integers.

▶ **Definition 3.3.** A *parameterized TI representation system* (in short: TIRS) is a tuple $\mathcal{R} = (\Lambda, \mathbf{P}, \Sigma, \mathbf{T}, \langle\,\cdot\,\rangle, [\![\,\cdot\,]\!])$ where $\Lambda \neq \emptyset$ is a set (the *parameter set*); $\mathbf{P}$ is a family $\left(P_\lambda\right)_{\lambda\in\Lambda}$ of probability distributions P_λ over $\mathbb{N}$; $\Sigma \neq \emptyset$ is a finite set of symbols (the *encoding alphabet*); $\langle\,\cdot\,\rangle\colon \Lambda \to \Sigma^*$ is an injective function (the *encoding function*); and $(\mathbf{T}, [\![\,\cdot\,]\!])$ is an RS where

- $\mathbf{T}$ is the family of all finite sets T of pairs $\left(f, \langle\lambda_f\rangle\right)$ with pairwise different facts f of a given schema and $\lambda_f \in \Lambda$ for all f; and
- $[\![\,\cdot\,]\!]$ maps every $T \in \mathbf{T}$ to the tuple-independent bag PDB $\mathcal{D}$ with multiplicity probabilities $\Pr\left(\sharp_\mathcal{D} f = k\right) = P_{\lambda_f}(k)$ for all $\left(f, \langle\lambda_f\rangle\right) \in T$. ⌟

Whenever a TIRS $\mathcal{R}$ is given, we assume $\mathcal{R} = (\Lambda_\mathcal{R}, \mathbf{P}_\mathcal{R}, \Sigma_\mathcal{R}, \mathbf{T}_\mathcal{R}, \langle\,\cdot\,\rangle_\mathcal{R}, [\![\,\cdot\,]\!]_\mathcal{R})$ by default.

▶ **Example 3.4.** Figure 1 shows a table T from a TIRS $\mathcal{R}$, illustrating how the parameters can be used to encode several multiplicity distributions. Four of the distributions are standard parameterized distributions, presented using their symbolic name together with their parameters. The multiplicity distribution for $R(2,2)$ is a generic distribution with finite support $\{0,1,5\}$. The annotation $(\texttt{Binomial}, 10, 1/3)$ of $R(1,2)$ in the table specifies that $\sharp_T R(1,2) \sim \text{Binomial}\left(10, \frac{1}{3}\right)$. That is,

$$\Pr\left(\sharp_T R(1,2) = k\right) = \begin{cases} \binom{10}{k}\left(\frac{1}{3}\right)^k\left(\frac{2}{3}\right)^{10-k} & \text{if } 0 \leq k \leq 10 \text{ and} \\ 0 & \text{if } k > 10. \end{cases}$$

The multiplicity probabilities of the other facts are given analogously in terms of the Bernoulli, geometric, and Poisson distributions, respectively. The supports of the multiplicity distributions are $\{0,1\}$ for the Bernoulli, $\{0,\dots,n\}$ for the Binomial, and $\mathbb{N}$ for both the geometric and Poisson distributions (and finite sets for explicitly encoded distributions). For the first three parameterized distributions, multiplicity probabilities always stay rational if the parameters are rational. This is not the case for the Poisson distribution. ⌟

While Definition 3.2 seems abstract, this level of detail in the encoding of probability distributions allows us to rigorously discuss computational complexity without resorting to a very narrow framework that only supports some predefined distributions. Our model also comprises tuple-independent set PDBs: The traditional representation system can be recovered from Definition 3.3 using only the Bernoulli distribution. Moreover, we remark that we can always represent facts that are present with probability 0, by just omitting them from the tables (for example, fact $R(2,1)$ in Figure 1).

▶ Remark 3.5. In this work, we focus on TIRS's where the values needed for computation (moments in Section 4 and probabilities in Section 5) are rational. An extension to support irrational values is possible through models of real complexity [25, 5]. A principled treatment requires a substantial amount of introductory overhead that would go beyond the scope of this paper, and which we therefore leave for future work. ⌟

4 Expectations and Variances

Before computing the probabilities of answer counts, we discuss the computation of the expectation and the variance of the answer count. Recall that in PDBs without multiplicities, the answer to a Boolean query (under set semantics) is either 0 (i. e., false) or 1 (i. e., true). That is, the answer count is a $\{0,1\}$-valued random variable there, meaning that its expectation coincides with the probability of the answer count being 1. Because of this correspondence, the semantics of Boolean queries on (set) PDBs are sometimes also defined in terms of the expected value [31]. For bag PDBs, the situation is different, and this equivalence no longer holds. Thus, computing expectations, and computing answer count probabilities have to receive a separate treatment. Formally, we discuss the following problems in this section:

Problem	EXPECTATION$_{\mathcal{R}}(Q)$
Parameter:	A Boolean UCQ Q.
Input:	A table $T \in \mathbf{T}_{\mathcal{R}}$.
Output:	The expectation $\mathbb{E}\left(\sharp_T Q\right)$.

Problem	VARIANCE$_{\mathcal{R}}(Q)$
Parameter:	A Boolean UCQ Q.
Input:	A table $T \in \mathbf{T}_{\mathcal{R}}$.
Output:	The variance $\operatorname{Var}\left(\sharp_T Q\right)$.

4.1 Expected Answer Count

We have pointed out above that computing expected answer counts for set PDBs and set semantics is equivalent to computing the probability that the query returns true. There are conjunctive queries, for example, $Q = \exists x \exists y\colon R(x) \wedge S(x,y) \wedge T(y)$, for which the latter problem is $\sharp$P-hard [15, 9]. Under a set semantics, disjunctions and existential quantifiers semantically correspond to taking maximums instead of adding multiplicities. Under a bag semantics, we are now able to exploit the linearity of expectation to easily compute expected values, which was not possible under a set semantics.

▶ **Lemma 4.1.** *Let $\mathcal{D}$ be a tuple-independent PDB and let Q be a Boolean CQ, $Q = \exists x_1 \dots \exists x_m \colon R_1(\mathbf{t}_1) \wedge \dots \wedge R_n(\mathbf{t}_n)$. For every $\mathbf{a} \in \operatorname{adom}(\mathcal{D})^m$, we let $F(\mathbf{a})$ denote the set of facts appearing in $Q^*[\mathbf{x}/\mathbf{a}]$, and for every $f \in F(\mathbf{a})$, we let $\nu(f, \mathbf{a})$ denote the number of times f appears in $Q^*[\mathbf{x}/\mathbf{a}]$. Then*

$$\mathbb{E}\left(\sharp_{\mathcal{D}} Q\right) = \sum_{\mathbf{a} \in \operatorname{adom}(\mathcal{D})^m} \prod_{f \in F(\mathbf{a})} \mathbb{E}\left(\left(\sharp_{\mathcal{D}} f\right)^{\nu(f,\mathbf{a})}\right). \tag{3}$$

Proof. By definition, we have

$$\sharp_D Q = \sum_{\mathbf{a} \in \operatorname{adom}(D)} \sharp_D(Q^*[\mathbf{x}/\mathbf{a}]) = \sum_{\mathbf{a} \in \operatorname{adom}(\mathcal{D})} \sharp_D(Q^*[\mathbf{x}/\mathbf{a}])$$

for every individual instance D of $\mathcal{D}$. The last equation above holds because, as Q^* is assumed to contain every quantified variable, $\sharp_D(Q^*[\mathbf{x}/\mathbf{a}]) = 0$ whenever the tuple $\mathbf{a}$ contains an element that is not in the active domain of D. By linearity of expectation, we have

$$\mathbb{E}\left(\sharp_{\mathcal{D}} Q\right) = \sum_{\mathbf{a} \in \operatorname{adom}(\mathcal{D})} \mathbb{E}\left(\sharp_{\mathcal{D}}(Q^*[\mathbf{x}/\mathbf{a}])\right).$$

Recall, that $Q^*[\mathbf{x}/\mathbf{a}]$ is a conjunction of facts $R_i(\mathbf{t}_i[\mathbf{x}/\mathbf{a}])$. Thus, $\sharp_{\mathcal{D}}\big(\bigwedge_{i=1}^n R_i\big(\mathbf{t}_i[\mathbf{x}/\mathbf{a}]\big)\big) = \prod_{i=1}^n \sharp_{\mathcal{D}}\big(R_i\big(\mathbf{t}_i[\mathbf{x}/\mathbf{a}]\big)\big)$. Because $\mathcal{D}$ is tuple-independent, any two facts in $F(\mathbf{a})$ are either equal, or independent. Therefore,

$$\mathbb{E}\left(\prod_{i=1}^n \sharp_{\mathcal{D}} R_i(\mathbf{t}_i[\mathbf{x}/\mathbf{a}])\right) = \prod_{f \in F(\mathbf{a})} \mathbb{E}\left(\left(\sharp_{\mathcal{D}} f\right)^{\nu(f,\mathbf{a})}\right),$$

as the expectation of a product of independent random variables is the product of their expectations. Together, this yields the expression from (3). ◀

By linearity, the expectation of a UCQ is the sum of the expectations of its CQs.

▶ **Lemma 4.2.** *Let $\mathcal{D}$ be a PDB and let $Q = \bigvee_{i=1}^N Q_i$ be a Boolean UCQ. Then we have $\mathbb{E}\left(\sharp_{\mathcal{D}} Q\right) = \sum_{i=1}^N \mathbb{E}\left(\sharp_{\mathcal{D}} Q_i\right)$.*

Given that we can compute the necessary moments of fact multiplicities efficiently, Lemmas 4.1 and 4.2 yield a polynomial time procedure to compute the expectation of a UCQ. The order of moments we need is governed by the self-join width of the individual CQs.

▶ **Definition 4.3.** A TIRS $\mathcal{R}$ has *polynomially computable moments* up to order k, if for all $\lambda \in \Lambda_{\mathcal{R}}$, we have $\sum_{n=0}^\infty n^k \cdot P_\lambda(n) < \infty$ and the function $\langle\lambda\rangle \mapsto \sum_{n=0}^\infty n^\ell \cdot P_\lambda(n)$ can be computed in polynomial time in $|\langle\lambda\rangle|$ for all $\ell \leq k$. ⌟

Before giving the main statement, let us revisit Example 3.4 for illustration.

▶ **Example 4.4.** Let $\mathcal{R}$ be the TIRS from Example 3.4. The moments of $X \sim \text{Bernoulli}(p)$ are $\mathbb{E}(X^k) = p$ for all $k \geq 1$. Direct calculation shows that for $X \sim \text{Binomial}(n, p)$, the moment $\mathbb{E}(X^k)$ is given by a polynomial in n and p. In general, for most of the common distributions, one of the following cases applies. Either, as above, a closed form expression for $\mathbb{E}(X^k)$ is known, or, the moments of X are characterized in terms of the moment generating function (mgf) $\mathbb{E}(e^{tX})$ of X, where t is a real-valued variable. In the latter case, $\mathbb{E}(X^k)$ is obtained by taking the kth derivative of the mgf and evaluating it at $t = 0$ [6, p. 62]. An inspection of the mgfs of the geometric, and the Poisson distributions [6, p. 621f] reveals that their kth moments are polynomials in their respective parameters as well. Together, $\mathcal{R}$ has polynomially computable moments up to order k for all $k \in \mathbb{N}_+$. ⌟

▶ **Proposition 4.5.** *Let $Q = \bigvee_{i=1}^{N} Q_i$ be a Boolean UCQ, and let $\mathcal{R}$ be a TIRS with polynomially computable moments up to order k, where k is the maximum self-join width among the Q_i. Then* $\mathsf{EXPECTATION}_{\mathcal{R}}(\sharp_T Q)$ *is computable in polynomial time.*

Proof. We plug (3) into the formula from Lemma 4.2. This yields at most $\leq N \cdot |\mathrm{adom}(T)|^m \cdot m$ terms (where m is the maximal number of atoms among the CQs $Q_1, \dots, Q_N$). These terms only contain moments of fact multiplicities of order at most k. ◀

We emphasize that the number k from Proposition 4.5, that dictates which moments we need to be able to compute efficiently, comes from the fixed query Q and is therefore constant. More precisely, it is given through the number of self-joins in the query. In particular, if all CQs in Q are self-join free, it suffices to have efficient access to the expectations of the multiplicities.

4.2 Variance of the Answer Count

With the ideas from the previous section, we can also compute the variance of query answers in polynomial time. Naturally, to be able to calculate the variance efficiently, we need moments of up to the double order in comparison to the computation of the expected value.

▶ **Proposition 4.6.** *Let $Q = \bigvee_{i=1}^{N} Q_i$ be a Boolean UCQ, and let $\mathcal{R}$ be a TIRS with polynomially computable moments up to order $2k$, where k is the maximum self-join width among the Q_i. Then* $\mathsf{VARIANCE}_{\mathcal{R}}(\sharp_T Q)$ *is computable in polynomial time.*

As before, the main idea is to rewrite the variance in terms of the moments of fact multiplicities. This can be achieved by exploiting tuple-independence and linearity of expectation. The full proof is contained in the extended version of this paper [20].

Despite the fact that the variance of query answers may be of independent interest, it can be also used to obtain bounds for the probability that the true value of $\sharp_T Q$ is close to its expectation, using the Chebyshev inequality [24, Theorem 5.11]. This can be used to derive bounds on $\Pr(\sharp_T Q \leq k)$, when the exact value is hard to compute.

▶ Remark 4.7. Proposition 4.6 extends naturally to higher-order moments: If $\mathcal{R}$ is a TIRS with polynomially computable moments up to order $\ell \cdot k$ and $Q = \bigvee_{i=1}^{N} Q_i$ a Boolean UCQ where the maximum self-join width among the Q_i is k, then all centralized and all raw moments of order up to ℓ of $\sharp_T Q$ are computable in polynomial time. ⌟

5 Answer Count Probabilities

In this section, we treat the alternative version of probabilistic query evaluation in bag PDBs using answer count probabilities rather than expected values. Formally, we discuss the following problem.

Problem $\mathsf{PQE}_{\mathcal{R}}(Q, k)$

PARAMETER:	A Boolean (U)CQ Q, and $k \in \mathbb{N}$.
INPUT:	A table $T \in \mathbf{T}_{\mathcal{R}}$.
OUTPUT:	The probability $\Pr(\sharp_T Q \leq k)$.

This problem amounts to evaluating the cumulative distribution function of the random variable $\sharp_T Q$ at k. The properties of this problem bear a close resemblance to the set version of probabilistic query evaluation, and we hence name this problem “PQE”.

▶ Remark 5.1. Instead of asking for $\Pr(\sharp_T Q \leq k)$, we could similarly define the problem of evaluating the probability that $\sharp_T Q$ is *at least*, or *exactly* equal to k. This has no impact on complexity discussions, though, as these variants are polynomial time equivalent. ⌟

Throughout this section, calculations involve the probabilities for the multiplicities of individual facts. However, we want to discuss the complexity of $\mathsf{PQE}_{\mathcal{R}}(Q, k)$ independently of the complexity, in k, of evaluating the multiplicity distributions. This motivates the following definition, together with taking k to be a parameter, instead of it being part of the input.

▶ **Definition 5.2.** A TIRS $\mathcal{R}$ is called a *p-TIRS*, if for all $k \in \mathbb{N}$ there exists a polynomial p_k such that for all $\lambda \in \Lambda_{\mathcal{R}}$, the function $\langle \lambda \rangle \mapsto P_\lambda(k)$ can be evaluated in time $\mathcal{O}\big(p_k(|\langle \lambda \rangle|)\big)$. ⌟

Definition 5.2 captures reasonable assumptions for "efficient" TIRS's with respect to the evaluation of probabilities: If the requirement from the definition is not given, then $\mathsf{PQE}_{\mathcal{R}}(\exists x \colon R(x), k)$ can not be solved in polynomial time, even on the class of tables that only contain a single annotated fact $R(a)$. This effect only arises due to the presence of unwieldy probability distributions in $\mathcal{R}$.

As it turns out, solving $\mathsf{PQE}_{\mathcal{R}}(Q, k)$ proves to be far more intricate compared to the problems of the previous section. For our investigation, we concentrate on self-join free conjunctive queries. While some simple results follow easily from the set semantics version of the problem, the complexity theoretic discussions quickly become quite involved and require the application of a set of interesting non-trivial techniques.

Our main result in this section is a dichotomy for Boolean CQs without self-joins. From now on, we employ nomenclature (like *hierarchical*) that was introduced in [11, 12]. If Q is a Boolean self-join free CQ, then for every variable x, we let $\mathrm{at}(x)$ denote the set of relation symbols R such that Q contains an R-atom that contains x. We call Q *hierarchical* if for all distinct x and y, whenever $\mathrm{at}(x) \cap \mathrm{at}(y) \neq \emptyset$, then $\mathrm{at}(x) \subseteq \mathrm{at}(y)$ or $\mathrm{at}(y) \subseteq \mathrm{at}(x)$. This definition essentially provides the separation between easy and hard Boolean CQs without self-joins. In the bag semantics setting, however, there exists an edge case where the problem gets easy just due to the limited expressive power of the representation system. This edge case is governed only by the probabilities for multiplicity zero that appear in the representations. We denote this set by $\mathsf{zeroPr}(\mathcal{R})$, i.e.,

$$\mathsf{zeroPr}(\mathcal{R}) = \{p \in [0,1] \colon P_\lambda(0) = p \text{ for some } \lambda \in \Lambda(\mathcal{R})\}.$$

If a p-TIRS satisfies $\mathsf{zeroPr}(\mathcal{R}) \subseteq \{0,1\}$, then it can only represent bag PDBs whose deduplication is deterministic. In this case, as we will show in the next subsection, the problem becomes easy even for arbitrary UCQs.[2]

▶ **Theorem 5.3.** *Let Q be a Boolean CQ without self-joins and let $\mathcal{R}$ be a p-TIRS.*
1. *If Q is hierarchical or $\mathsf{zeroPr}(\mathcal{R}) \subseteq \{0,1\}$, then $\mathsf{PQE}_{\mathcal{R}}(Q, k)$ is solvable in polynomial time for all $k \in \mathbb{N}$.*
2. *Otherwise, $\mathsf{PQE}(Q, k)$ is $\sharp\mathsf{P}$-hard for all $k \in \mathbb{N}$.*

▶ Remark 5.4. It is natural to ask what happens, if k is treated as part of the input. With a reduction similar to the proof of Proposition 5 in [28], it is easy to identify situations in which the corresponding problem is $\sharp\mathsf{P}$-hard. For example, this is already the case for the

[2] Using the same definition in the set semantics version of the problem would come down to restricting the input tuple-independent PDB to only use 0 and 1 as marginal probabilities, so the problem would collapse to traditional (non-probabilistic) query evaluation. Under a bag semantics, there still exist interesting examples in this class, as the probability distribution over non-zero multiplicities is not restricted in any way.

simple query $\exists x\colon R(x)$, provided that k is encoded in binary and that the p-TIRS supports fair coin flips whose outcomes are either a positive integer, or zero. A full proof is shown in the extended version [20]. ⌟

The remainder of this section is dedicated to establishing Theorem 5.3.

5.1 Tractable Cases

Let us first discuss the case of p-TIRS's $\mathcal{R}$ with $\mathsf{zeroPr}(\mathcal{R}) \subseteq \{0,1\}$. Here, $\mathsf{PQE}_{\mathcal{R}}(Q,0)$ is trivial, because the problem essentially reduces to deterministic query evaluation. The following lemma generalizes this to all values of k.

▶ **Lemma 5.5.** *If $\mathcal{R}$ is a p-TIRS with $\mathsf{zeroPr}(\mathcal{R}) \subseteq \{0,1\}$, then $\mathsf{PQE}_{\mathcal{R}}(Q,k)$ is solvable in polynomial time for all Boolean UCQs Q, and all $k \in \mathbb{N}$.*

Proof. Let $\mathcal{R}$ be any p-TIRS with $\mathsf{zeroPr}(\mathcal{R}) \subseteq \{0,1\}$ and let Q be any Boolean UCQ. If $P_\lambda(0) = 1$ for all $\lambda \in \Lambda_{\mathcal{R}}$, then $\mathcal{R}$ can only represent the PDB where the empty instance has probability 1. In this case, $\sharp_T Q = 0$ almost surely, so $\Pr(\sharp_T Q \leq k) = 1$ for all $k \in \mathbb{N}$.

In the general case, suppose $Q = \bigvee_{i=1}^{N} Q_i$ such that $Q_1, \dots, Q_N$ are CQs. Let A be the set of functions α that map the variables of Q into the active domain of the input T. We call α *good*, if there exists $i \in \{1, \dots, N\}$ such that all the facts emerging from the atoms of Q_i by replacing every variable x with $\alpha(x)$ have positive multiplicity in T (almost surely). If there are at least $k+1$ good α in A, then $\sharp_T Q > k$ with probability 1 and, hence, we return 0 in this case. Otherwise, when there are at most k good α, we restrict T to the set of all facts that can be obtained from atoms of Q by replacing all variables x with $\alpha(x)$ (and retaining the parameters λ). The resulting table T' contains at most k times the number of atoms in Q many facts, which is independent of the number of facts in T. Hence, we can compute $\Pr(\sharp_T Q \leq k) = \Pr(\sharp_{T'} Q \leq k)$ in time polynomial in T by using brute-force. ◀

From now on, we focus on the structure of queries again. The polynomial time procedure for Boolean CQs without self-joins is reminiscent of the original algorithm for set semantics as described in [11]. Therefore, we need to introduce some more vocabulary from their work. A variable x is called *maximal*, if $\mathrm{at}(y) \subseteq \mathrm{at}(x)$ for all y with $\mathrm{at}(x) \cap \mathrm{at}(y) \neq \emptyset$. With every CQ Q we associate an undirected graph G_Q whose vertices are the variables appearing in Q, and where two variables x and y are adjacent if they appear in a common atom. Let $V_1, \dots, V_m$ be the vertex sets of the connected components of G_Q. We can then write the quantifier-free part Q^* of Q as $Q^* = Q_0^* \wedge \bigwedge_{i=1}^{m} Q_i^*$ where Q_0^* is the conjunction of the constant atoms of Q and $Q_1^*, \dots, Q_m^*$ are the conjunctions of atoms corresponding to the connected components $V_1, \dots, V_m$. We call $Q_1^*, \dots, Q_m^*$ the *connected components* (short: *components*) of Q.

▶ Remark 5.6. If Q is hierarchical, then every component of Q contains a maximal variable.[3] Moreover, if x is maximal in a component Q_i^*, then x appears in all atoms of Q_i^*. ⌟

▶ Remark 5.7. For every CQ Q with components $Q_1^*, \dots, Q_m^*$, and constant atoms Q_0^*, the answer on every instance D is given by the product of the answers of the queries $Q_0, \dots, Q_m$, where $Q_i = \exists \mathbf{x}_i\colon Q_i^*$ (and $Q_0 = Q_0^*$), and $\mathbf{x}_i$ are exactly the variables appearing in the component Q_i^*. That is, $\sharp_D Q = \sharp_D Q_0^* \cdot \prod_{i=1}^{m} \sharp_D \big(\exists \mathbf{x}_i\colon Q_i^*\big)$. This is shown in the extended version of this paper [20]. If convenient, we therefore use $Q_0 \wedge Q_1 \wedge \dots \wedge Q_m$ as an alternative representation of Q. ⌟

[3] This is true, since the sets $\mathrm{at}(x)$ for the variables of any component have a pairwise non-empty intersection, meaning that they are pairwise comparable with respect to $\subseteq$.

The main result of this subsection is the following.

▶ **Theorem 5.8.** *Let $\mathcal{R}$ be a p-TIRS, and let Q be a hierarchical Boolean CQ without self-joins. Then $\mathsf{PQE}_{\mathcal{R}}(Q,k)$ is solvable in polynomial time for each $k \in \mathbb{N}$.*

Proof Sketch. The theorem is established by giving a polynomial time algorithm that computes, and adds up the probabilities $\Pr\big(\sharp_T Q = k'\big)$ for all $k' \leq k$. The important observation is that (as under set semantics) the components Q_i of the query (and the conjunction Q_0 of the constant atom) yield independent events, which follows since Q is self-join free. In order to compute the probability of $\sharp_T Q = k'$, we can thus sum over all decompositions of k' into a product $k' = k_0 \cdot k_1 \cdot \dots \cdot k_m$, and reduce the problem to the computations of $\Pr\big(\sharp_T Q_i = k_i\big)$. Although the cases $k = 0$, and the conjunction Q_0 have to be treated slightly different for technical reasons, we can proceed recursively: Every component contains a maximal variable, and setting this variable to any constant, the component potentially breaks up into a smaller hierarchical, self-join free CQ. Investigating the expressions shows that the total number of operations on the probabilities of fact probabilities is polynomial in the size of T. ◀

▶ Remark 5.9. The full proof of Theorem 5.8 can be found in the extended version [20]. As pointed out, the proof borrows main ideas from the algorithm for the probabilistic evaluation of hierarchical Boolean self-join free CQs on tuple-independent PDBs with set semantics, as presented in [12, p. 30:15] (originating in [9, 10]). The novel component is the treatment of multiplicities using bag semantics. In comparison to the algorithm of Dalvi and Suciu, existential quantifiers behave quite differently here, and we additionally need to argue about the possible ways to distribute a given multiplicity over subformulae or facts. ⌟

5.2 Intractable Cases

We now show that in the remaining case (non-hierarchical queries and p-TIRS's with $\mathsf{zeroPr}(\mathcal{R}) \cap (0,1) \neq \emptyset$), the problems $\mathsf{PQE}_{\mathcal{R}}(Q,k)$ are all hard to solve.

Let Q be a fixed query and let $\mathsf{PQE}^{\mathsf{set}}(Q)$ denote the traditional set version of the probabilistic query evaluation problem. That is, $\mathsf{PQE}^{\mathsf{set}}(Q)$ is the problem to compute the probability that Q evaluates to true under *set* semantics, on input a tuple-independent *set* PDB. We recall that the bag version $\mathsf{PQE}_{\mathcal{R}}(Q,k)$ of the problem (introduced at the beginning of the section) takes the additional parameter k, and depends on the representation system $\mathcal{R}$. Let us first discuss $\mathsf{PQE}_{\mathcal{R}}(Q,k)$ for $k = 0$. In this case, subject to very mild requirements on $\mathcal{R}$, we can lift $\sharp\mathsf{P}$-hardness from the set version [12], even for the full class of UCQs.

▶ **Proposition 5.10.** *Let $S \subseteq [0,1]$ be finite and let $\mathcal{R}$ be a p-TIRS such that $1-p \in \mathsf{zeroPr}(\mathcal{R})$ for all $p \in S$. Let Q be a Boolean UCQ. If $\mathsf{PQE}^{\mathsf{set}}(Q)$ is $\sharp\mathsf{P}$-hard on tuple-independent (set) PDBs with marginal probabilities in S, then $\mathsf{PQE}_{\mathcal{R}}(Q,0)$ is $\sharp\mathsf{P}$-hard.*

Proof. Let $\mathcal{D}$ be an input to $\mathsf{PQE}^{\mathsf{set}}(Q)$ with fact set F where all marginal probabilities are in S, given by the list of all facts f with their marginal probability p_f. For all $p \in S$, pick $\lambda_p \in \Lambda$ such that $P_\lambda(0) = 1 - p$. Let $T = \bigcup_{f\in F}\big\{(f, \langle\lambda_{p_f}\rangle)\big\}$, and let δ be the function that maps every instance D of T to its deduplication D' (which is an instance of $\mathcal{D}$). Then, by the choice of the parameters, we have $\Pr_{D\sim[\![T]\!]}\big(\delta(D) = D'\big) = \Pr_{\mathcal{D}}\big(\{D'\}\big)$ for all D'. Moreover, $\sharp_D Q > 0$ if and only if $\delta(D) \models Q$. Thus,

$$\Pr_{D\sim[\![T]\!]}\big(\sharp_D Q > 0\big) = \Pr_{D\sim[\![T]\!]}\big(\delta(D) \models Q\big) = \Pr_{D'\sim\mathcal{D}}\big(D' \models Q\big).$$

Therefore, $\mathsf{PQE}^{\mathsf{set}}(Q)$ over tuple-independent PDBs with marginal probabilities from S can be solved by solving $\mathsf{PQE}_{\mathcal{R}}(Q,0)$. ◀

Remarkably, [23, Theorem 2.2] shows that Boolean UCQs for which $\mathsf{PQE}^{\mathsf{set}}(Q)$ is $\sharp$P-hard are already hard when the marginal probabilities are restricted to $S = \{c, 1\}$, for any rational $c \in (0, 1)$. Hence, $\mathsf{PQE}_{\mathcal{R}}(Q, 0)$ is also $\sharp$P-hard on these queries, as soon as $\{0, 1-c\} \subseteq \mathsf{zeroPr}(\mathcal{R})$.

Our goal is now to show that if $\mathcal{R}$ is a p-TIRS, then for any Boolean CQ Q without self-joins, $\sharp$P-hardness of $\mathsf{PQE}_{\mathcal{R}}(Q, 0)$ transfers to $\mathsf{PQE}_{\mathcal{R}}(Q, k)$ for all $k > 0$.

▶ **Theorem 5.11.** *Let $\mathcal{R}$ be a p-TIRS and let Q be a Boolean self-join free CQ. Then, if $\mathsf{PQE}_{\mathcal{R}}(Q, 0)$ is $\sharp$P-hard, $\mathsf{PQE}_{\mathcal{R}}(Q, k)$ is $\sharp$P-hard for each $k \in \mathbb{N}$.*

Proving Theorem 5.11 is quite involved, and is split over various lemmas in the remainder of this subsection. Let $\mathcal{R}$ be any fixed p-TIRS and let Q be a Boolean CQ without self-joins. We demonstrate the theorem by presenting an algorithm that solves $\mathsf{PQE}_{\mathcal{R}}(Q, 0)$ in polynomial time, when given an oracle for $\mathsf{PQE}_{\mathcal{R}}(Q, k)$ for *any* positive k.

Clearly, we cannot simply infer $\Pr(\sharp_T Q = 0)$ from $\Pr(\sharp_T Q \leq k)$. Naively, we would want to shift the answer count of Q by k, so that the problem could be answered immediately. However, this is not possible in general. Our way out is to use the oracle several times, on manipulated inputs. Since the algorithms we describe are confined to the p-TIRS $\mathcal{R}$, we are severely restricted in the flexibility of manipulating the probabilities of fact multiplicities: Unless further assumptions are made, we can only work with the annotations that are already present in the input T to the problem. We may, however, also drop entries from T or introduce copies of facts using new domain elements.

For a given table T and a fixed single-component query Q, we exploit this idea in Algorithm 1 in order to construct a new table $T^{(m)}$, called the *inflation* of T of order m. It has the property that $\sharp_{T^{(m)}} Q$ is the sum of answer counts of Q on m independent copies of T. A small example for the result of running Algorithm 1 for $m = 2$ is shown in Figure 3. We will later use oracle answers on several inflations in order to interpolate $\Pr(\sharp_T Q_i = 0)$ per component Q_i individually, and then combine the results together.

Algorithm 1 $\mathsf{inflate}_Q(T, m)$.

Parameter: Boolean self-join free CQ Q with a single component and no constant atoms
Input: $T \in \mathbf{T}_{\mathcal{R}}$, $m \in \mathbb{N}$
Output: Inflation of order m of T: $T^{(m)} = \bigcup_{i=1}^{m} T_{m,i} \in \mathbf{T}_{\mathcal{R}}$ such that
(O1) for all $i \neq j$ we have $T_{m,i} \cap T_{m,j} = \emptyset$,
(O2) for all $i = 1, \dots, m$ we have $\sharp_{T_{m,i}} Q \sim \sharp_T Q$ i.i.d., and
(O3) $\sharp_{T^{(m)}} Q = \sum_{i=1}^{m} \sharp_{T_{m,i}} Q$.

```
1:  Initialize $T_{m,1}, \dots, T_{m,m}$ to be empty.
2:  For each domain element $a$, introduce new pairwise distinct elements $a^{(1)}, \dots, a^{(m)}$.
3:  for all relation symbols $R$ appearing in $Q$ do
4:      Let $R(t_1, \dots, t_r)$ be the unique atom in $Q$ with relation symbol $R$.
5:      for all pairs of the form $\big(R(a_1, \dots, a_r), \langle\lambda\rangle\big) \in T$ do
6:          for all $i = 1, \dots, m$ do
7:              Add $\big(R(a_{i,1}, \dots, a_{i,k}), \langle\lambda\rangle\big)$ to $T_{m,i}$ where $a_{i,j} = \begin{cases} a_j^{(i)}, & \text{if } t_j \text{ is a variable;} \\ a_j, & \text{if } t_j \text{ is a constant.} \end{cases}$
8:          end for
9:      end for
10: end for
11: return $T^{(m)} := \bigcup_{i=1}^{m} T_{m,i}$
```

Table T

Relation R	Parameter
$R(a,a,a)$	(Binomial, 10, 1/3)
$R(a,b,c)$	(Geometric, 1/2)

Table $T^{(2)}$

Relation R	Parameter
$R(a^{(1)},a^{(1)},a)$	(Binomial, 10, 1/3)
$R(a^{(1)},b^{(1)},c)$	(Geometric, 1/2)
$R(a^{(2)},a^{(2)},a)$	(Binomial, 10, 1/3)
$R(a^{(2)},b^{(2)},c)$	(Geometric, 1/2)

Figure 3 Example of a table T and its inflation $T^{(2)}$ for the query $\exists x, y\colon R(x,y,a)$.

▶ **Lemma 5.12.** *For every fixed Boolean self-join free CQ Q with a single component and no constant atoms, Algorithm 1 runs in time $\mathcal{O}(|T| \cdot m)$, and satisfies the output conditions* (O1)*,* (O2) *and* (O3)*.*

The proof of Lemma 5.12 is contained in the extended version [20]. The assumption that the input to Algorithm 1 is self-join free with just a single connected component and no constant atoms is essential to establish (O1) and (O3), because it eliminates any potential co-dependencies among the individual tables $T_{m,1},\dots,T_{m,m}$ we create. The following example shows that this assumption is inevitable, as the conditions of Lemma 5.12 can not be established in general.

▶ **Example 5.13.** Let $\mathcal{R}$ be a TIRS with $\Lambda_{\mathcal{R}} = \{\lambda\}$ such that $P_\lambda(2) = P_\lambda(3) = \frac{1}{2}$ (and $P_\lambda(k) = 0$ for all $k \notin \{2,3\}$). Consider $Q = \exists x \exists y\colon R(x) \wedge S(y)$, and $T = \big((R(1),\langle\lambda\rangle),(S(1),\langle\lambda\rangle)\big) \in \mathbf{T}_{\mathcal{R}}$. Note that Q has two components and, hence, does not satisfy the assumptions of Lemma 5.12. Then $\sharp_T Q$ takes the values 4, 6 and 9, with probabilities $\frac{1}{4},\frac{1}{2},\frac{1}{4}$. Thus, if $X,Y \sim \sharp_T Q$ i.i.d., then $X+Y$ is 13 with probability $\frac{1}{16}+\frac{1}{16}=\frac{1}{8}$. However, for every $T' \in \mathbf{T}_{\mathcal{R}}$, the random variable $\sharp_{T'}Q$ almost surely takes composite numbers, as it is equal to the sum of all multiplicities of R-facts, times the sum of all multiplicities of S-facts, both of these numbers being either 0 or at least 2. Thus, there exists no $T' \in \mathbf{T}_{\mathcal{R}}$ such that $\sharp_{T'}Q = X+Y$. ⌟

For this reason, our main algorithm will call Algorithm 1 independently, for each connected component Q_i of Q. Then, Algorithm 1 does not inflate the whole table T, but only the part T_i corresponding to Q_i. If $Q = Q' \wedge Q_i$ and we denote $\sharp_{T_i}Q_i$ by X and $\sharp_{T\setminus T_i}Q'$ by Y, then replacing T_i in T with its inflation of order n yields a new table with answer count $(\sharp_{T\setminus T_i}Q')\cdot(\sharp_{T_i^{(n)}}Q_i) = Y\cdot\sum_{i=1}^n X_i$, where $X_1,\dots,X_n \sim X$ i.i.d. Before further describing the reduction, we first explore some algebraic properties of the above situation in general.

▶ **Lemma 5.14.** *Let X and Y be independent random variables with values in $\mathbb{N}$ and let $k \in \mathbb{N}$. Suppose $X_1, X_2, \dots$ are i.i.d. random variables with $X_1 \sim X$. Let $p_0 := \Pr(X=0)$ and $q_0 := \Pr(Y=0)$. Then, there exist $z_1,\dots,z_k \geq 0$ such that for all $n \in \mathbb{N}$ we have*

$$\Pr\left(Y\cdot\sum_{i=1}^n X_i \leq k\right) = q_0 + (1-q_0)\cdot p_0^n + \sum_{j=1}^k \binom{n}{j}\cdot p_0^{n-j}\cdot z_j.$$

This is demonstrated in the extended version [20]. We now describe how $p_0 = \Pr(X=0)$ can be recovered from the values of $\Pr\left(Y\cdot\sum_{i=1}^n X_i \leq k\right)$ and $q_0 = \Pr(Y=0)$ whenever $q_0 < 1$ and $p_0 > 0$. With the values $z_1,\dots,z_n$ from Lemma 5.14, and $z_0 := 1-q_0$, we define a function

$$g(n) := \Pr\left(Y\cdot\sum_{i=1}^n X_i \leq k\right) - q_0 = \sum_{j=0}^k \binom{n}{j}\cdot p_0^{n-j}\cdot z_j. \tag{4}$$

Now, for $m \in \mathbb{N}$ and $x = 0, 1 \dots, m$, we define

$$h_m(x) := g(m+x) \cdot g(m-x) = \sum_{j_1,j_2=0}^{k} \binom{m+x}{j_1} \cdot \binom{m-x}{j_2} \cdot p_0^{2m-j_1-j_2} \cdot z_{j_1} \cdot z_{j_2}. \qquad (5)$$

Then, for every fixed m, h_m is a polynomial in x with domain $\{0, \dots, m\}$. As it will turn out, the leading coefficient $\mathrm{lc}(h_m)$ of h_m can be used to recover the value of p_0 as follows: Let $j_{\max}$ be the maximum j such that $z_j \neq 0$. Since for fixed m, both $\binom{m+x}{j}$ and $\binom{m-x}{j}$ are polynomials of degree j in x, the degree of h_m is $2j_{\max}$ and its leading coefficient is

$$\mathrm{lc}(h_m) = (-1)^{j_{\max}} \cdot (j_{\max}!)^{-2} \cdot p_0^{2m-2j_{\max}} \cdot z_{j_{\max}}^2,$$

which yields

$$p_0 = \sqrt{\frac{\mathrm{lc}(h_{m+1})}{\mathrm{lc}(h_m)}}. \qquad (6)$$

Thus, it suffices to determine $\mathrm{lc}(h_m)$ and $\mathrm{lc}(h_{m+1})$. However, we neither know $j_{\max}$, nor $z_{j_{\max}}$, and we only have access to the values of h_m and h_{m+1}. To find the leading coefficients anyway, we employ the method of finite differences, a standard tool from polynomial interpolation [22, chapter 4]. For this, we use the difference operator Δ that is defined as $\Delta f(x) := f(x+1) - f(x)$ for all functions f. When f is a (non-zero) polynomial of degree n, the difference operator reduces its degree by one and its leading coefficient is multiplied by n. Therefore, after taking differences n times, starting from subsequent values of a polynomial f, we are left with the constant function $\Delta^n f = n! \cdot \mathrm{lc}(f) \neq 0$. In particular, taking differences more than n times yields the zero function. Hence, we can determine $\mathrm{lc}(f)$ by finding the largest ℓ for which $\Delta^\ell f(0) \neq 0$.

Algorithm 2 $\mathsf{solveComponent}_Q(T, i)$.

Parameter: Boolean self-join free CQ $Q = Q_0 \wedge \bigwedge_{i=1}^{r} Q_i$ with connected components $Q_1 \dots, Q_r$.
Oracle Access: Oracle for $\mathsf{PQE}_{\mathcal{R}}(Q, k)$ that, on input $\widetilde{T}$, returns $\Pr(\sharp_{\widetilde{T}} Q \leq k)$
Input: $T \in \mathbf{T}_{\mathcal{R}}$, $i \in \{1, \dots, r\}$
Output: $\Pr(\sharp_{T_i} Q_i = 0)$

```
1: if $P_\lambda(0) = 1$ for all $\lambda \in \Lambda$ then return $0$ end if
2: Fix $\lambda$ with $P_\lambda(0) < 1$ and suppose $Q = Q' \wedge Q_i$ (cf. Remark 5.7).
3: if $Q'$ is empty then
4:     Set $q_0 := 0$ and $g(0) := 1$.
5: else
6:     Let $T' \in \mathbf{T}_{\mathcal{R}}$ be the canonical database for $Q'$, with $\lambda_f := \lambda$ for all facts $f$ in $T'$.
7:     Calculate $q_0 := \Pr\big(\sharp_{T'} Q' = 0\big)$ and set $g(0) := 1 - q_0$.
8: end if
9: for $n = 1, 2, \dots, 4k+1$ do
10:    Set $T_i^{(n)} := \mathsf{inflate}_{Q_i}(T_i, n)$.
11:    Set $g(n) := \Pr\big(\sharp_{T' \cup T_i^{(n)}} Q \leq k\big) - q_0$, using the oracle.
12: end for
13: if $g(k+1) = 0$ then return $0$ end if
14: for $x = 0, 1, \dots, 2k$ and $m = 2k, 2k+1$ do $h_m(x) := g(m+x) \cdot g(m-x)$ end for
15: Initialize $\ell := 2k$.
16: while $\Delta^\ell h_{2k}(0) = 0$ do $\ell := \ell - 1$ end while
17: return $\sqrt{\Delta^\ell h_{2k+1}(0) / \Delta^\ell h_{2k}(0)}$
```

The full procedure that uses the above steps to calculate p_0 yields Algorithm 2. Recall that it focuses on a single connected component. To ensure easy access to the value of q_0, we utilize a table that encodes the canonical database of the remainder of the query.[4] Note that k is always treated as a fixed constant, and our goal is to reduce $\mathsf{PQE}_{\mathcal{R}}(Q, 0)$ to $\mathsf{PQE}_{\mathcal{R}}(Q, k)$.

▶ **Lemma 5.15.** *Algorithm 2 runs in polynomial time and yields the correct result.*

Proof. With the notation introduced in the algorithm, we let $Y = \sharp_{T'} Q'$ (or $Y = 1$ if Q' is empty) and $X = \sharp_T Q_i = \sharp_{T_i} Q_i$. Then, $q_0 = \Pr(Y = 0)$ as in Lemma 5.14 and the aim of the algorithm is to return p_0.

First, line 1 covers the edge case that $\mathcal{R}$ can only represent the empty database instance. In all other cases, we fix λ with $P_\lambda(0) > 0$. As $q_0 = 1 - (1 - P_\lambda(0))^t$ where t is the number of atoms of Q', we have $q_0 < 1$. From Lemma 5.12, we see that $\sharp_{T' \cup T_i^{(n)}} Q = Y \cdot \sum_{i=1}^n X_i$, so we are in the situation of Lemma 5.14. Hence, g and h_m are as in (4) and (5). Now, as $g(k+1) = p_0 \cdot \sum_{j=0}^k \binom{n}{j} \cdot p_0^{k-j} \cdot z_j$ with $z_0 = 1 - q_0 > 0$, we find that p_0 is zero if and only if $g(k+1)$ is zero. This is checked in line 13. Finally, the paragraphs following Lemma 5.14 apply, and we determine the degree of h_{2k} using the method of finite differences by setting ℓ to the maximum possible degree and decreasing it step-by-step as long as $\Delta^\ell h_{2k}(0) = 0$ in lines 15 and 16. Then, we have $\ell = 2j_{\max}$ and return

$$\sqrt{\frac{\Delta^\ell h_{2k+1}(0)}{\Delta^\ell h_{2k}(0)}} = \sqrt{\frac{\ell! \operatorname{lc}(h_{m+1})}{\ell! \operatorname{lc}(h_m)}} \overset{(6)}{=} p_0.$$

Concerning the runtime, since $\mathcal{R}$ is a p-TIRS, all answers of the oracle calls are of polynomial size in the input. Since k is fixed, the algorithm performs a constant number of computation steps and each term in the calculations is either independent of the input or of polynomial size, yielding a polynomial runtime. ◀

Proof of Theorem 5.11. Let $k > 0$ and suppose that we have an oracle for $\mathsf{PQE}_{\mathcal{R}}(Q, k)$. Let $Q = Q_0 \wedge \bigwedge_{i=1}^m Q_i$ be the partition of Q into components, with Q_0 being the conjunction of the constant atoms. Then the $\sharp_T Q_i$ are independent and $\sharp_T Q = \sharp_T Q_0 \cdot \prod_{i=1}^m \sharp_T Q_i$. Therefore,

$$\Pr\big(\sharp_T Q = 0\big) = 1 - \Pr(\sharp_T Q_0 \neq 0) \cdot \prod_{i=1}^m \big(1 - \Pr\big(\sharp_T Q_i = 0\big)\big).$$

As $\Pr(\sharp_T Q_0 \neq 0)$ is easy to compute and Algorithm 2 computes $\Pr\big(\sharp_T Q_i = 0\big)$ for $i = 1, \dots, k$ with oracle calls for $\mathsf{PQE}_{\mathcal{R}}(Q, k)$, this yields a polynomial time Turing-reduction from $\mathsf{PQE}_{\mathcal{R}}(Q, 0)$ to $\mathsf{PQE}_{\mathcal{R}}(Q, k)$. ◀

With the results from the previous subsections, this completes the proof of Theorem 5.3.

Proof of Theorem 5.3. For p-TIRS's with $\mathsf{zeroPr}(\mathcal{R}) \subseteq \{0, 1\}$, the statement is given by Lemma 5.5. By Theorem 5.8, $\mathsf{PQE}_{\mathcal{R}}(Q, k)$ is solvable in polynomial time for hierarchical Boolean CQs without self-joins. For the case of Q being non-hierarchical (and $\mathsf{zeroPr}(\mathcal{R}) \cap (0, 1) \neq \emptyset$), let $p \in (0, 1)$ such that $p \in \mathsf{zeroPr}(\mathcal{R})$. By [2, Theorem 3.4], the set version $\mathsf{PQE}^{\mathsf{set}}(Q)$ is already hard on the class of tuple-independent set PDBs where all probabilities are equal to $1 - p$. It follows from Proposition 5.10 that $\mathsf{PQE}_{\mathcal{R}}(Q, 0)$ is $\sharp\mathsf{P}$-hard. By Theorem 5.11, so is $\mathsf{PQE}_{\mathcal{R}}(Q, k)$ for all $k \in \mathbb{N}_+$. ◀

[4] The canonical database belonging to a self-join free CQ is the instance containing the atoms appearing in the query, with all variables being treated as constants.

6 Conclusion

The results of our paper extend the understanding of probabilistic query evaluation into a new direction by discussing bag semantics. We investigated two principal computational problems: computing expectations, and computing the probability of answer counts. Interestingly, even though these problems are equivalent for set semantics, they behave quite differently under bag semantics. Our findings show that generally, computing expectations is the easier problem. For computing answer count probabilities, in the case of self-join free CQs, we obtain a polynomial time vs. ♯P-hard dichotomy, depending on whether the query is hierarchical. This transfers the corresponding results of [9, 10] from set to bag semantics.

While our results for the expectation problem concern UCQs, the complexity of computing answer count probabilities remains open beyond self-join free CQs. It is also unclear, how the problem behaves on bag versions of other well-representable classes of set PDBs. A more detailed analysis of the complexity of $\mathsf{PQE}_{\mathcal{R}}(Q, k)$ in terms of k remains open as well.

To formally argue about the complexity of some natural distributions such as the Poisson distribution, irrational probabilities or parameters have to be supported. This yields non-trivial complexity theoretic questions that we leave for future work.

References

1 Antoine Amarilli, Pierre Bourhis, and Pierre Senellart. Tractable Lineages on Treelike Instances: Limits and Extensions. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems (PODS 2016)*, pages 355–370. Association for Computing Machinery, 2016. `doi:10.1145/2902251.2902301`.

2 Antoine Amarilli and Benny Kimelfeld. Uniform Reliability of Self-Join-Free Conjunctive Queries, 2021. Extended version of [3]. `arXiv:1908.07093v6`.

3 Antoine Amarilli and Benny Kimelfeld. Uniform Reliability of Self-Join-Free Conjunctive Queries. In *24th International Conference on Database Theory (ICDT 2021)*, volume 186 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 17:1–17:17. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. `doi:10.4230/LIPIcs.ICDT.2021.17`.

4 Antoine Amarilli and İsmail İlkan Ceylan. A Dichotomy for Homomorphism-Closed Queries on Probabilistic Graphs. In *23rd International Conference on Database Theory (ICDT 2020)*, volume 155 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 5:1–5:20, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. `doi:10.4230/LIPIcs.ICDT.2020.5`.

5 Mark Braverman and Stephen Cook. Computing over the Reals: Foundations for Scientific Computing. *Notices of the AMS*, 53(3):318–329, 2006.

6 George Casella and Roger L. Berger. *Statistical Inference*. Thomson Learning, Pacific Grove, CA, USA, 2nd edition, 2002.

7 Surajit Chaudhuri and Moshe Y. Vardi. Optimization of *Real* Conjunctive Queries. In *Proceedings of the Twelfth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS 1993)*, pages 59–70. ACM Press, 1993. `doi:10.1145/153850.153856`.

8 Sara Cohen. Equivalence of Queries That Are Sensitive to Multiplicities. *The VLDB Journal*, 18(3):765–785, 2009. `doi:10.1007/s00778-008-0122-1`.

9 Nilesh Dalvi and Dan Suciu. Efficient Query Evaluation on Probabilistic Databases. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases (VLDB 2004)*, volume 30, pages 864–875. VLDB Endowment, 2004.

10 Nilesh Dalvi and Dan Suciu. Efficient Query Evaluation on Probabilistic Databases. *The VLDB Journal*, 16(4):523–544, 2007. `doi:10.1007/s00778-006-0004-3`.

11 Nilesh Dalvi and Dan Suciu. The Dichotomy of Conjunctive Queries on Probabilistic Structures. In *Proceedings of the Twenty-Sixth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS 2007)*, pages 293–302, New York, NY, USA, 2007. Association for Computing Machinery. doi:10.1145/1265530.1265571.

12 Nilesh Dalvi and Dan Suciu. The Dichotomy of Probabilistic Inference for Unions of Conjunctive Queries. *Journal of the ACM*, 59(6):1–87, 2012. doi:10.1145/2395116.2395119.

13 Su Feng, Boris Glavic, Aaron Huber, Oliver Kennedy, and Atri Rudra. Computing Expected Multiplicities for Bag-TIDBs with Bounded Multiplicities, 2022. arXiv:2204.02758v3.

14 Robert Fink and Dan Olteanu. Dichotomies for Queries with Negation in Probabilistic Databases. *ACM Transactions on Database Systems*, 41(1):4:1–4:47, 2016. doi:10.1145/2877203.

15 Erich Grädel, Yuri Gurevich, and Colin Hirsch. The Complexity of Query Reliability. In *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS 1998)*, pages 227–234. ACM Press, 1998. doi:10.1145/275487.295124.

16 Todd J. Green, Grigoris Karvounarakis, and Val Tannen. Provenance Semirings. In *Proceedings of the Twenty-Sixth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS 2007)*, pages 31–40. Association for Computing Machinery, 2007. doi:10.1145/1265530.1265535.

17 Todd J. Green and Val Tannen. Models for Incomplete and Probabilistic Information. In *Current Trends in Database Technology – EDBT 2006*, pages 278–296, Berlin, Germany, 2006. Springer-Verlag Berlin Heidelbeg. doi:10.1007/11896548_24.

18 Martin Grohe and Peter Lindner. Independence in Infinite Probabilistic Databases. *J. ACM*, 69(5):37:1–37:42, 2022. doi:10.1145/3549525.

19 Martin Grohe and Peter Lindner. Infinite Probabilistic Databases. *Logical Methods in Computer Science*, Volume 18, Issue 1, 2022. doi:10.46298/lmcs-18(1:34)2022.

20 Martin Grohe, Peter Lindner, and Christoph Standke. Probabilistic Query Evaluation with Bag Semantics, 2022. arXiv:2201.11524.

21 Erich Grädel and Val Tannen. Semiring Provenance for First-Order Model Checking, 2017. arXiv:1712.01980v1.

22 Francis Begnaud Hildebrand. *Introduction to Numerical Analysis.* Courier Corporation, 1987.

23 Batya Kenig and Dan Suciu. A Dichotomy for the Generalized Model Counting Problem for Unions of Conjunctive Queries. In *Proceedings of the 40th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems (PODS 2021)*, pages 312–324, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3452021.3458313.

24 Achim Klenke. *Probability Theory: A Comprehensive Course.* Universitext. Springer-Verlag London, London, UK, 2nd edition, 2014. Translation from the German language edition. doi:10.1007/978-1-4471-5361-0.

25 Ker-I Ko. *Complexity Theory of Real Functions.* Progress in Theoretical Computer Science. Birkhäuser Boston, 1991. doi:10.1007/978-1-4684-6802-1.

26 Dan Olteanu and Jiewen Huang. Using OBDDs for Efficient Query Evaluation on Probabilistic Databases. In *SUM 2008: Scalable Uncertainty Management*, Lecture Notes in Computer Science, pages 326–340, Berlin, Heidelberg, 2008. Springer. doi:10.1007/978-3-540-87993-0_26.

27 Dan Olteanu and Jiewen Huang. Secondary-Storage Confidence Computation for Conjunctive Queries with Inequalities. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pages 389–402. ACM, 2009. doi:10.1145/1559845.1559887.

28 Christopher Ré and Dan Suciu. The Trichotomy Of HAVING Queries On A Probabilistic Database. *The VLDB Journal*, 18(5):1091–1116, July 2009. doi:10.1007/s00778-009-0151-4.

29 Dan Suciu. Probabilistic Databases for All. In *Proceedings of the 39th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 19–31. ACM, 2020. doi:10.1145/3375395.3389129.

30 Dan Suciu, Dan Olteanu, Christopher Ré, and Christoph Koch. *Probabilistic Databases*, volume 3.2 of *Synthesis Lectures on Data Management*. Morgan & Claypool, San Rafael, CA, USA, 2011. Lecture ♯16. `doi:10.2200/S00362ED1V01Y201105DTM016`.

31 Guy Van den Broeck and Dan Suciu. Query Processing on Probabilistic Data: A Survey. *Foundations and Trends® in Databases*, 7(3–4):197–341, 2017. `doi:10.1561/1900000052`.

32 Moshe Y. Vardi. The Complexity of Relational Query Languages. In *Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing (STOC 1982)*, pages 137–146, New York, NY, USA, 1982. ACM Press. `doi:10.1145/800070.802186`.

On Efficient Range-Summability of IID Random Variables in Two or Higher Dimensions

Jingfan Meng
School of Computer Science, Georgia Institute of Technology, Atlanta, GA, USA

Huayi Wang
School of Computer Science, Georgia Institute of Technology, Atlanta, GA, USA

Jun Xu
School of Computer Science, Georgia Institute of Technology, Atlanta, GA, USA

Mitsunori Ogihara
Department of Computer Science, University of Miami, Coral Gables, MI, USA

Abstract

d-dimensional (for $d > 1$) efficient range-summability (dD-ERS) of random variables (RVs) is a fundamental algorithmic problem that has applications to two important families of database problems, namely, fast approximate wavelet tracking (FAWT) on data streams and approximately answering range-sum queries over a data cube. Whether there are efficient solutions to the dD-ERS problem, or to the latter database problem, have been two long-standing open problems. Both are solved in this work. Specifically, we propose a novel solution framework to dD-ERS on RVs that have Gaussian or Poisson distribution. Our dD-ERS solutions are the first ones that have polylogarithmic time complexities. Furthermore, we develop a novel k-wise independence theory that allows our dD-ERS solutions to have both high computational efficiencies and strong provable independence guarantees. Finally, we show that under a sufficient and likely necessary condition, certain existing solutions for 1D-ERS can be generalized to higher dimensions.

2012 ACM Subject Classification Theory of computation → Streaming, sublinear and near linear time algorithms

Keywords and phrases fast range-summation, multidimensional data streams, Haar wavelet transform

Digital Object Identifier 10.4230/LIPIcs.ICDT.2023.21

Related Version *Previous Version*: `https://arxiv.org/abs/2110.07753`

Funding This material is based upon work supported by the National Science Foundation under Grant No. CNS-1909048, CNS-2007006, CNS-2051800, and by Keysight Technologies under Grant No. BG005054.

1 Introduction

Efficient range-summability (ERS) of random variables (RVs) is a fundamental algorithmic problem that has been studied for nearly two decades [5, 21, 6, 15]. This problem has so far been defined only in one dimension (1D) as follows. Let $X_0, X_1, \cdots, X_{\Delta-1}$ be a list of *underlying RVs* each of which has the same *target distribution* X. Here, the (index) universe size Δ is typically a large number (say $\Delta = 2^{64}$). A 1D-ERS problem calls for the following oracle for answering range-sum queries over (realizations of) these underlying RVs. At initialization, the oracle chooses a *random outcome* ω from the *sample space* Ω, which *mathematically determines* the (values of the) realizations $X_0(\omega), X_1(\omega), \cdots, X_{\Delta-1}(\omega)$; here the phrase "mathematically determines" emphasizes that (an implementation of) the oracle does not actually realize these RVs (and pay the $O(\Delta)$ time cost) at initialization. Thereafter, given any query range $[l, u) \triangleq \{l, l+1, \cdots, u-1\}$ that lies in the universe $[0, \Delta)$, the oracle is required to return $S[l,u) \triangleq \sum_{i=l}^{u-1} X_i(\omega)$, the sum of the realizations of all underlying RVs in the range. This requirement is called the *consistency* requirement, which is one of the two essential requirements for the ERS oracle. We will show that such an ERS oracle can be efficiently implemented using hash functions. With such an implementation, the outcome ω corresponds to the seeds of these hash functions.

The other essential requirement is *correct distribution*, which has two aspects. The first aspect is that the underlying RVs $X_0, X_1, \cdots, X_{\Delta-1}$ each has the same target (marginal) distribution X. The second aspect is that these RVs should satisfy certain independence guarantees. Ideally, it is desired for these RVs to be mutually independent, but this comes at a high storage cost as we will elaborate shortly. In practice, another type of independence guarantee, namely k-wise independence (in the sense that any subset of k underlying RVs are independent), is good enough for most applications when $k \geq 4$. We will show that our solution for ERS in $d > 1$ dimensions can provide k-wise independence guarantee at a small storage cost of $O(\log^d \Delta)$ for an arbitrarily large k.

A straightforward but naive way to answer a range-sum query, say over $[l,u)$, is simply to sum up the realization of every underlying RV $X_l(\omega), X_{l+1}(\omega), \cdots, X_{u-1}(\omega)$ in the query range. This solution, however, has a time complexity of $O(\Delta)$ when $u-l$ is $O(\Delta)$. In contrast, an efficient solution should be able to do so with only $O(\text{polylog}(\Delta))$ time complexity. Indeed, all existing ERS solutions [2, 5, 21, 6, 15] have $O(\log \Delta)$ time complexity.

1.1 Related Work on 1D-ERS

There are in general two families of solutions to the ERS problem in 1D, following two different approaches. The first approach is based on error correction codes (ECC). Solutions taking this approach include BCH3 [21], EH3 [5], and RM7 [2]. This approach has two drawbacks. First, it works only when the target distribution X is Rademacher ($\Pr[X=1] = \Pr[X=0] = 0.5$, aka. single-step random walk). Second, although it guarantees 3-wise (in the case of BCH3 and EH3) or 7-wise (in the case of RM7) independence among the underlying RVs, almost all empirical independence beyond that is destroyed. In addition, RM7 is very slow in practice [21].

The second approach is based on a data structure called dyadic simulation tree (DST), which we will describe in Subsection 3.1. The DST-based approach was first briefly mentioned in [6] and later fully developed in [15]. The DST-based approach is better than the ECC-based approach in two aspects. First, it supports a wider range of target distributions including Gaussian, Cauchy, Rademacher [15], and Poisson (see Appendix C of [14]). Second, it provides stronger independence guarantees at a low computational cost. For example, when implemented using the tabulation hashing scheme [24], it guarantees 5-wise independence at a much lower computational cost than RM7 [15]. We will describe a nontrivial generalization of this result to 2D in Section 4.

1.2 ERS in Higher Dimensions

In this work, we formulate the ERS problems in $d > 1$ dimensions (dD), which we denote as dD-ERS, and propose the first-ever solutions to dD-ERS. A dD-ERS problem is similarly defined on a d-dimensional universe $[0,\Delta)^d$ that contains Δ^d integral points. Each dD point $\vec{\mathbf{i}} \in [0,\Delta)^d$ is associated with an RV $X_{\vec{\mathbf{i}}}$, and every such RV has the same target (marginal) distribution X. Here, for ease of presentation, we assume Δ is the same on each dimension and is a power of 2, but our solutions can work without these two assumptions. Let $\vec{\mathbf{l}} = (l_1, l_2, \cdots, l_d)^T$ and $\vec{\mathbf{u}} = (u_1, u_2, \cdots, u_d)^T$ be two dD points in $[0,\Delta)^d$ such that

$l_j < u_j$ for each dimension $j = 1, 2, \cdots, d$. We define $[\vec{\mathbf{l}}, \vec{\mathbf{u}})$ as the dD rectangular range "cornered" by these two points in the sense $[\vec{\mathbf{l}}, \vec{\mathbf{u}}) \triangleq [l_1, u_1) \times [l_2, u_2) \times \cdots \times [l_d, u_d)$, where $\times$ is the Cartesian product.

A dD-ERS problem calls for the following oracle. At initialization, the oracle chooses an outcome ω that *mathematically determines* the realization $X_{\vec{\mathbf{i}}}(\omega)$ for each $\vec{\mathbf{i}} \in [0, \Delta)^d$. Thereafter, given any dD range $[\vec{\mathbf{l}}, \vec{\mathbf{u}})$, the oracle needs to return in $O(\text{polylog}(\Delta))$ time $S[\vec{\mathbf{l}}, \vec{\mathbf{u}}) \triangleq \sum_{\vec{\mathbf{i}} \in [\vec{\mathbf{l}}, \vec{\mathbf{u}})} X_{\vec{\mathbf{i}}}(\omega)$, the sum of the realizations of all underlying RVs in this dD range. Unless otherwise stated, the vectors that appear in the sequel are assumed to be column vectors. We write them in boldface and with a rightward arrow on the top like in "$\vec{\mathbf{x}}$".

Several 1D-ERS solutions have been proposed as an essential building block for efficient solutions to several database problems. In two such database problems that we will describe in Section 2, their 1D solutions, both proposed in [7], can be readily generalized to dD if their underlying 1D-ERS oracles can be generalized to dD. In fact, in [16], authors stated explicitly that the only missing component for their solutions of the 1D database problems to be generalized to 2D was an efficient 2D-ERS oracle where X is the Rademacher distribution. However, until this paper, *no solution* to any dD-ERS problem for $d > 1$ has been proposed.

1.3 Our dD-ERS Solutions

In this paper, we propose novel solutions to the two dD-ERS problems wherein the target distributions are Gaussian and Poisson respectively. We refer to these two problems as dD Gaussian-ERS and dD Poisson-ERS, respectively. Both solutions generalize the corresponding DST-based 1D-ERS solutions to higher dimensions and have a low time complexity of $O(\log^d \Delta)$ per range-sum query. Our dD Gaussian-ERS solution, in particular, is based on the Haar wavelet transform (HWT), since DST is equivalent to HWT when (and only when) the target distribution X is Gaussian, as will be shown in Subsection 3.2.

Furthermore, we identify a sufficient condition that, if satisfied by the target distribution X, guarantees that the corresponding DST-based 1D-ERS solution can be generalized to a dD-ERS solution. We prove (see Appendix A of [14]) that Gaussian and Poisson are two "nice" distributions that satisfy this sufficient condition. We will also show that, for all such "nice" distributions (including those we might discover in the future), this generalization process (from 1D to dD) follows a universal algorithmic framework that can be characterized as the Cartesian product of d DSTs. We will also provide strong evidence that X "being nice" is likely necessary for this DST generalization (from 1D to dD) to be feasible (see Section 5).

Unfortunately, so far we have not found any "nice" distribution other than Gaussian and Poisson. Hence dD-ERS for other target distributions remains an open problem, and is likely not solvable by the (generalized) DST approach. We emphasize this is not a shortcoming of the DST approach: That we have obtained computationally efficient solutions in the cases of Gaussian and Poisson is already a pleasant surprise, as the dD-ERS problem has been open for nearly two decades. Furthermore, we will show that our dD Gaussian-ERS solution leads to computationally efficient solutions to both aforementioned database problems (to be described in Section 2), by answering their calls for a dD Gaussian-ERS or equivalent oracle.

Our dD Gaussian-ERS and Poisson-ERS solutions both support two different types of independence guarantees, at different storage costs. The first type is the ideal case in which the Δ^d underlying RVs are mutually independent. As will be shown in Section 3, we can achieve this ideal case by paying $O(T \log^d \Delta)$ storage cost, where T is the total number of range-sum queries to be answered (i.e., $O(\log^d \Delta)$ storage cost per range query). The second type is also quite strong: The Δ^d underlying RVs are k-wise independent, where the constant k can be arbitrarily large. In Section 4, we propose a k-wise independence scheme that can provide the second type of guarantees by employing $O(\log^d \Delta)$ k-wise independent hash functions. Its storage cost is quite small: only $O(\log^d \Delta)$ for storing the seeds of these hash functions. We emphasize that the issue of how strong this independence guarantee (among the underlying RVs) needs to be affects only the storage cost of our Gaussian-ERS and Poisson-ERS solutions, and is orthogonal to all other issues described in earlier paragraphs such as the $O(\log^d \Delta)$ time complexity of both solutions and the sufficient and likely necessary condition for a DST-based dD-ERS solution to exist.

This k-wise independence scheme makes our dD-ERS solutions very practically useful for two reasons. First, such a k-wise independent hash function in practice requires a very short seed (not longer than a few kilobytes), and each hash operation can be computed in nanoseconds [3, 19]. Second, most applications of ERS only require the underlying RVs to be 4-wise independent [7, 16].

The contributions of this work can be summarized as follows. First, we provide the first set of answers to the long-standing open question whether there is an efficient solution to *any* dD-ERS problem for $d > 1$. Second, our Gaussian-ERS solution solves a long-standing open problem in data streaming that we will describe next. Third, our k-wise independence theory and hashing scheme make our dD ERS solutions very practically useful.

The rest of the paper is organized as follows. In Section 2, we describe two applications of our dD Gaussian-ERS solutions. In Section 3, we first describe our HWT-based Gaussian-ERS scheme in 1D, and then generalize it to 2D and dD. In Section 4, we describe our k-wise independence theory and scheme. In Section 5, we propose a sufficient and likely necessary condition on the target distribution for the DST approach to be generalized to dD. Finally, we conclude the paper in Section 6.

2 Applications of dD Gaussian-ERS

In this section, we introduce two important applications of our dD Gaussian-ERS solution.

2.1 Fast Approximate Wavelet Tracking

The first application is to the problem of fast approximate wavelet tracking (FAWT) on data streams [7, 4]. We first introduce the FAWT problem in 1D [7], or 1D-FAWT for short. In this problem, the precise system state is comprised of a Δ-dimensional vector $\vec{\mathbf{s}}$, each scalar of which is a counter. The precise system state at any moment of time is determined by a data stream, in which each data item is an update to one such counter (called a *point update*) or all counters in a 1D range (called a *range update*). In 1D-FAWT, $\vec{\mathbf{s}}$ is considered a Δ-dimensional signal vector that is constantly "on the move" caused by the updates in the data stream. Let $\vec{\mathbf{r}}$ be the (Δ-dimensional) vector of HWT coefficients of $\vec{\mathbf{s}}$. Clearly, $\vec{\mathbf{r}}$ is also a "moving target". We denote as $\vec{\mathbf{r}}_t$ the snapshot of $\vec{\mathbf{r}}$ at a time t. In 1D-FAWT, the goal is to closely track (the precise value of) $\vec{\mathbf{r}}$ over time using a *sketch*, in the sense that at moment t, we can recover from the sketch an estimate $\vec{\mathbf{r}}'_t$ of $\vec{\mathbf{r}}_t$, such that $\|\vec{\mathbf{r}}_t - \vec{\mathbf{r}}'_t\|_2$ is small. An acceptable solution should use a sketch whose size (space complexity) is only $O(\text{polylog}(\Delta))$, and be able to maintain the sketch with a computation time cost of $O(\text{polylog}(\Delta))$ per point or range update.

The first solution to 1D-FAWT was proposed in [7]. It requires the efficient computation of an arbitrary scalar in $H\vec{\mathbf{x}}$, where H is the $\Delta \times \Delta$ Haar matrix (to be defined in Subsubsection 3.2.1) and $\vec{\mathbf{x}}$ is a Δ-dimensional vector of 4-wise independent Rademacher RVs. A key step of this computation is to compute a range-sum of 4-wise independent Rademacher RVs (in 1D), that is used therein as a Tug-of-War (ToW) sketch [1] for "sketching" the L_2

difference (approximation error) between the signal vector and its FAWT approximation. An aforementioned ECC-based ERS solution is used therein to tackle this Rademacher-ERS problem. Authors of [16] stated that if they could find a solution to this Rademacher-ERS problem in dD, then the 1D-FAWT solution in [7] would become a dD-FAWT solution. The first solution to dD-FAWT, proposed in [4], explicitly bypassed this ERS problem.

We note that the 1D-FAWT solution above continues to work, and its time and space complexities remain the same, if we replace the $\vec{\mathbf{x}}$ with a Δ-dimensional vector of 4-wise independent standard Gaussian RVs. This is because, with this replacement, the aforementioned ToW sketch becomes a Gaussian Tug-of-War (GToW) sketch (which maps a data item to a Gaussian RV instead of a Rademacher RV) [10], and ToW and GToW are known to have the same (ϵ, δ) accuracy bound [1, 10] for sketching the L_2 norm of a data stream (used here for sketching the aforementioned L_2 difference). Based on this insight, our dD Gaussian-ERS solution can be used to construct a dD-FAWT solution as follows. We simply change, in the *contingent* dD-FAWT solution proposed in [7], the distribution of all Δ^d underlying 4-wise independent RVs from Rademacher to Gaussian. With this replacement, this *contingent* solution will finally work, *provided* we can solve the resulting dD Gaussian-ERS problem. The latter problem is solved by our k-wise (with $k = 4$ here) independence scheme, to be described in Section 4. The resulting dD-FAWT solution has the same time and space complexity of $O(\log^d \Delta)$ as that proposed in [4] for achieving the same accuracy guarantee.

2.2 Range-Sum Queries over Data Cube

Our second application is to the problem of approximately answering range-sum queries over a data cube [8] that is similarly "on the move" propelled by the (point or range) updates that arrive in a stream. This problem can be formulated as follows. The precise system state is comprised of Δ^d counters, namely $\sigma_{\vec{\mathbf{i}}}$ for $\mathbf{i} \in [0, \Delta)^d$, that are "on the move". Given a range $[\vec{\mathbf{l}}, \vec{\mathbf{u}})$ at moment t, the goal is to approximately compute the sum of counter values in this range $C[\vec{\mathbf{l}}, \vec{\mathbf{u}}) \triangleq \sum_{\vec{\mathbf{i}} \in [\vec{\mathbf{l}}, \vec{\mathbf{u}})} \sigma_{\vec{\mathbf{i}}}(t)$, where $\sigma_{\vec{\mathbf{i}}}(t)$ is the value of the counter $\sigma_{\vec{\mathbf{i}}}$ at moment t. A desirable solution to this problem in dD should satisfy three requirements (in which multiplicative terms related to the desired (ϵ, δ) accuracy bound are ignored). First, any range-sum query is answered in $O(\text{polylog}(\Delta))$ time. Second, its space complexity is $O(\text{polylog}(\Delta))$. Third, every point or range update to the system state is processed in $O(\text{polylog}(\Delta))$ time. It has been a long-standing open question whether there is a solution to this problem that satisfies all three requirements when $d > 1$. For example, solutions producing exact answers (to the range queries) [9, 22, 11] all require $O(\Delta^d \log \Delta)$ space and hence do not satisfy the second requirement; and Haar+ tree [12] works only on static data, and hence does not satisfy the third requirement.

In 1D, a solution that satisfies all three requirements (with $d = 1$) was proposed in [7, 6]. It involves 1D-ERS computations on 4-wise independent underlying RVs where the target distribution is either Gaussian or Rademacher, which are tackled using a DST-based (in [6]) or a ECC-based (in [7]) 1D-ERS solution, respectively. As shown in [7, 6], this range-sum query solution can be readily generalized to dD if the ERS computations above can be performed in dD. This gap is again filled by our k-wise ($k = 4$) independence scheme for dD Gaussian-ERS, resulting in the first dD solution that satisfies all three requirements, all with $O(\log^d \Delta)$ (time or space) complexity (ignoring ϵ and δ terms).

In the resulting dD solution, we maintain $O(\log(1/\delta)/\epsilon^2)$ (independent instances of) sketches that each "sketches" the content (counter values) of the data cube. Here we describe only one such sketch, which we denote as A, since these sketches are statistically and functionally identical. At any time t, $A(t)$ should track the current system state, namely

$(\sigma_{\vec{\mathbf{i}}}(t))$'s, as follows: $A(t) \triangleq \sum_{\vec{\mathbf{i}}\in[0,\Delta)^d} \sigma_{\vec{\mathbf{i}}}(t)X_{\vec{\mathbf{i}}}$. Here $X_{\vec{\mathbf{i}}}$ for $\mathbf{i} \in [0,\Delta)^d$ are (realizations of) a set of Δ^d 4-wise independent standard Gaussian underlying RVs that have one-to-one correspondences with the set of Δ^d counters as follows: Each $X_{\vec{\mathbf{i}}}$ is associated with a counter $\sigma_{\vec{\mathbf{i}}}$. If we implement these Δ^d RVs using (an instance of) our dD Gaussian-ERS solution, then we can keep the value of $A(t)$ up-to-date, with a time complexity of $O(\log^d \Delta)$ per point or range update (to the system state). Then, given a query range $[\vec{\mathbf{l}}, \vec{\mathbf{u}})$ at time t, we estimate the range-sum of counters $C[\vec{\mathbf{l}}, \vec{\mathbf{u}})$ from the sketch $A(t)$ using $A(t) \cdot S[\vec{\mathbf{l}}, \vec{\mathbf{u}})$ as the estimator. These $O(\log(1/\delta)/\epsilon^2)$ estimators, one obtained from each sketch, are then combined to produce a final estimation that has the following accuracy guarantee (that is the same as in the 1D case). With probability at least $1-\delta$, the final estimation deviates from the actual value of $C[\vec{\mathbf{l}}, \vec{\mathbf{u}})$ by at most $\epsilon\sqrt{V[\vec{\mathbf{l}}, \vec{\mathbf{u}})}\|\boldsymbol{\sigma}\|_2$, where $V[\vec{\mathbf{l}}, \vec{\mathbf{u}}) \triangleq \prod_{j=1}^{d}(u_j - l_j)$ is the number of counters in the query range, and $\|\boldsymbol{\sigma}(t)\|_2 \triangleq \left(\sum_{\vec{\mathbf{i}}\in[0,\Delta)^d} \sigma_{\vec{\mathbf{i}}}^2(t)\right)^{1/2}$ is the L_2 norm of the system state. Since each sketch uses an independent dD Gaussian-ERS scheme instance, our dD solution satisfies all three aforementioned requirements, all with $O(\log^d \Delta \log(1/\delta)/\epsilon^2)$ time and space complexity.

2.3 A Closer Comparison with Related Work

In this section, per referees' requests, we provide an in-depth comparison of this work with prior works on 1D-FAWT [7, 6], on dD-FAWT [4], and on 1D data cube [6].

We start with explaining how the dD-FAWT solution proposed in [4] manages to avoid confronting the dD-ERS problem. The dD-FAWT solution [4] maintains ToW sketches for groups of wavelet coefficients in the wavelet domain. As explained earlier, each ToW sketch "measures" the L_2 norm (and hence the total energy by squaring) of such a group. By the property of HWT, each point or range update to the system state in the time domain translates into $O(\log^d \Delta)$ updates to the sketches the wavelet domain; we also use this property in our solution to keep its time complexity below $O(\log^d \Delta)$ as shown in Subsection 3.4. To solve the dD-FAWT using these sketches in the wavelet domain, we need only to identify the groups that are (hierarchical) "L_2 heavy hitters" [4]. In [4], a binary search tree built on these sketches is used to search for such "L_2 heavy hitters" in $O(\log \Delta \cdot \log\log \Delta)$ time. Since this dD-FAWT solution [4] does not involve computing the range sums of the Rademacher RVs underlying the ToW sketches, it does not need to formulate or solve any ERS problem.

As we will elaborate in Section 3, our dD Gaussian-ERS solution works in the same way as the dD-FAWT solution proposed in [4], by shifting the (representations of) input streams and the range queries from the time domain to the wavelet domain. Hence, arguably had dD-FAWT solution proposed in [4] used the Gaussian ToW (GToW) instead of the ToW sketch, this shift would have resulted in a dD-FAWT solution containing the bulk of our dD Gaussian-ERS solution as an embedded module. However, such an embedded module is still "two hops away" from our dD Gaussian-ERS solution as follows. First, since the objective of and the intuition behind this shift in [4] were to avoid rather than to solve the ERS problem, it would not be easy for the authors of [4] to realize that the embedded module can be extended to a standalone dD Gaussian-ERS solution. Second, without our aforementioned k-wise independence theory and construction, the embedded module does not yet guarantee 4-wise independence among underlying Gaussian RVs that is needed for dD-FAWT.

On a related note, should we try to extend the 1D-FAWT solution proposed in [7], which maintains the ToW sketches in the time domain, to dD without the aforementioned Rademacher-by-Gaussian replacement, the underlying Rademacher RVs would have to be efficiently range-summable to keep the time complexity of each point or range update to the sketches low. However, this appears to be a tall order for now: For $d > 1$, no ECC-based Rademacher-ERS solution has ever been found as explained earlier, and a DST-based Rademacher-ERS solution is unlikely to exist, as we will show in Section 5.

A referee asked whether the 1D data cube solution proposed in [7, 6] can be extended to dD using the same aforementioned ERS avoidance strategy of maintaining the sketches in the wavelet domain as used in [4]. In retrospect, this solution approach would work, but unlikely to be taken since it is counterintuitive and still "two hops away" (from the right solution) as explained above. Indeed, authors of [7, 6] unsurprisingly took the much more intuitive approach of maintaining sketches in the time domain and as a result had to confront the dD Gaussian or Rademacher-ERS problem as explained in Subsection 2.2.

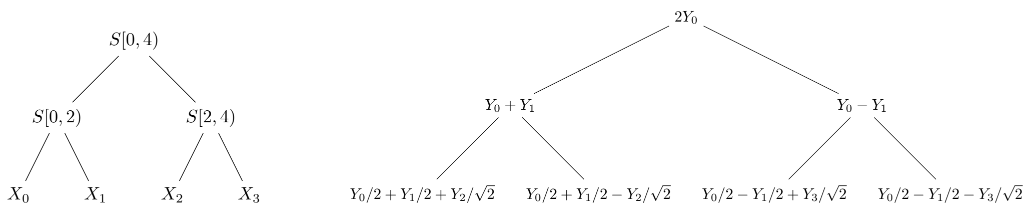
Now we highlight a key difficulty that we believe has prevented authors of [7, 6, 16] from solving the dD-ERS problem and extending their FAWT and data cube solutions from 1D to dD: The Rademacher or Gaussian RVs underlying the sketches need to be both 4-wise independent and efficiently range-summable, and conventional wisdom (until our work) has it that a magic hash function family is needed to achieve both. Authors of [7, 16] tried to extend a magic hash function family, that induces such Rademacher RVs in 1D, to dD. However, as explained earlier, a dD Rademacher-ERS solution is unlikely to exist. Authors of [6] proposed the 1D-DST that laid the foundation of this work and our prior work [15]. A key innovation of [6] is that the 1D-ERS is achieved via a 1D-DST instead of a magic hash function. However, their DST-based 1D Gaussian-ERS solution still relies on a magic hash function, called Nisan's PRG (Pseudorandom Generator) [18], to provide 4-wise independence among the underlying Gaussian RVs. The use of Nisan's PRG [18] however restricts the applicability and the extensibility of the 1D-DST approach, since Nisan's PRG provides independence guarantees only for memory-constrained applications such as data streaming [10]. It is also not clear whether the 1D-DST approach powered by Nisan's PRG can be extended to dD. In comparison, in our dD-ERS solutions, both dD-ERS and 4-wise independence are provided by the specially engineered dD-DST. As a result, a magic hash function family is no longer needed, since the hash values produced by a hash function are no longer required to be efficiently range summable.

Finally, we state a key difference between this work and [7, 6, 16, 4] with respect to wavelets. In this work, ERS is the end and wavelets is the means, whereas in [7, 6, 16] it is the other way around. In [4], wavelets is the end, but [4] cleverly avoids using ERS as the means as just explained.

3 Our Solution to dD Gaussian-ERS

In this section, we describe our dD Gaussian-ERS solution that answers a range-sum query in $O(\log^d \Delta)$ time. To explain this solution with best clarity, for now we require it to provide the aforementioned ideal guarantee that the Δ^d underlying RVs are *mutually independent*, with the understanding that this requirement affects only the space complexity of our solution. In the next section, this requirement will be relaxed to these RVs being k-wise independent, and as a result, the space complexity of our solution is reduced to $O(\log^d \Delta)$.

Our solution can be summarized as follows. Let $\vec{\mathbf{x}}$ denote the Δ^d underlying standard Gaussian RVs, namely $X_{\vec{\mathbf{i}}}$ for $\vec{\mathbf{i}} \in [0, \Delta)^d$, arranged (in the dictionary order of $\vec{\mathbf{i}}$) into a Δ^d-dimensional vector. Then, after the dD Haar wavelet transform (HWT) is performed on $\vec{\mathbf{x}}$, we obtain another Δ^d-dimensional vector $\vec{\mathbf{w}}$ whose scalars are the HWT coefficients of $\vec{\mathbf{x}}$. Our solution builds on the following two observations. The first observation is that scalars in $\vec{\mathbf{x}}$ are i.i.d. standard Gaussian RVs if and only if scalars in $\vec{\mathbf{w}}$ are (see Theorem 2).


$S[0,4)$
$S[0,2)$
$S[2,4)$
X_0
X_1
X_2
X_3
$2Y_0$
$Y_0 + Y_1$
$Y_0 - Y_1$
$Y_0/2 + Y_1/2 + Y_2/\sqrt{2}$
$Y_0/2 + Y_1/2 - Y_2/\sqrt{2}$
$Y_0/2 - Y_1/2 + Y_3/\sqrt{2}$
$Y_0/2 - Y_1/2 - Y_3/\sqrt{2}$

(a) A general DST. **(b)** A Gaussian-DST.

Figure 1 Illustrations of a general DST and a Gaussian-DST with $\Delta = 4$.

The second observation is that the answer to any dD range-sum query can be expressed as a weighted sum of $O(\log^d \Delta)$ scalars (HWT coefficients) in $\vec{\mathbf{w}}$ (see Lemma 10). Our algorithm is simply to *generate and remember* only these $O(\log^d \Delta)$ HWT coefficients (that participate in this range-sum query). Our solution satisfies the correct distribution requirement (with mutual independence guarantee) by the first observation. Since the first observation is true only when the target distribution is Gaussian, this HWT-based solution does not work for any other target distribution.

In the following, we first introduce the concept of the dyadic simulation tree (DST) in 1D in Subsection 3.1. Then, we show that 1D DST is equivalent to 1D HWT in the Gaussian case and present our HWT-based Gaussian-ERS algorithm for 1D, in Subsection 3.2. Finally, we describe our HWT-based Gaussian-ERS algorithms for 2D and dD in Subsection 3.3 and Subsection 3.4, respectively.

3.1 A Brief Introduction to DST

In this section, we briefly introduce the concept of the DST, which as mentioned earlier was proposed in [15] as a general solution approach to the one-dimensional (1D) ERS problems for arbitrary target distributions.

We say that $[l, u)$ is a 1D *dyadic range* if there exist integers $j \geq 0$ and $m \geq 0$ such that $l = j \cdot 2^m$ and $u = (j+1) \cdot 2^m$. We call the sum on a dyadic range a *dyadic range-sum*. Note that any underlying RV X_i is also a dyadic range-sum (on the dyadic range $[i, i+1)$). Let each underlying RV X_i have standard Gaussian distribution $\mathcal{N}(0,1)$. In the following, we focus on how to compute a dyadic range-sum, since any (general) 1D range can be "pieced together" using at most $2\log_2 \Delta$ dyadic ranges [21]. We illustrate the process of computing dyadic range-sums using a "small universe" example (with $\Delta = 4$) shown in Figure 1a. To begin with, the total sum of the universe $S[0,4)$ sitting at the root of the tree is generated directly from its distribution $\mathcal{N}(0,4)$. Then, $S[0,4)$ is split into two children, the half-range-sums $S[0,2)$ and $S[2,4)$, such that RVs $S[0,2)$ and $S[2,4)$ sum up to $S[0,4)$, are (mutually) independent, and each has distribution $\mathcal{N}(0,2)$. This is done by generating the RVs $S[0,2)$ and $S[2,4)$ from a conditional (upon $S[0,4)$) distribution that will be specified shortly. Afterwards, $S[0,2)$ is split in a similar way into two i.i.d. underlying RVs X_0 and X_1, and so is $S[2,4)$ (into X_2 and X_3). As shown in Figure 1a, the four underlying RVs are the leaves of the DST.

We now specify the aforementioned conditional distribution used for each split. Suppose the range-sum to split consists of $2n$ underlying RVs, and that its value is equal to z. The lower half-range-sum S_l (the left child in Figure 1a) is generated from the following conditional pdf (or pmf):

$$f(x \mid z) = \phi_n(x)\phi_n(z-x)/\phi_{2n}(z), \tag{1}$$

$$\begin{array}{c} W_0^{-1} \\ W_0^0 \\ W_0^1 \\ W_1^1 \end{array} \begin{pmatrix} Y_0 \\ Y_1 \\ Y_2 \\ Y_3 \end{pmatrix} = \begin{pmatrix} 1/2 & 1/2 & 1/2 & 1/2 \\ 1/2 & 1/2 & -1/2 & -1/2 \\ 1/\sqrt{2} & -1/\sqrt{2} & 0 & 0 \\ 0 & 0 & 1/\sqrt{2} & -1/\sqrt{2} \end{pmatrix} \cdot \begin{pmatrix} X_0 \\ X_1 \\ X_2 \\ X_3 \end{pmatrix}$$

Figure 2 An illustration of the HWT formula $\vec{\mathbf{y}} = H_4\vec{\mathbf{x}}$.

where $\phi_n(\cdot)$ is the pdf (or pmf) of X^{*n}, the n^{th} convolution power of the target distribution, and $\phi_{2n}(\cdot)$ is the pdf (or pmf) of X^{*2n}. Then, the upper half-range-sum (the right child) is defined as $S_u \triangleq z - S_l$. It was shown in [15] that splitting a (parent) RV using this conditional distribution guarantees that the two resulting RVs S_l and S_u are i.i.d. This guarantee holds regardless of the target distribution. However, *computationally efficient* procedures for generating an RV S_l with distribution $f(x \mid z)$ are found only when the target distribution is one of the few "nice" distributions: Gaussian, Cauchy, and Rademacher as shown in [15], and Poisson as shown in Appendix C of [14].

Among them, Gaussian distribution has a nice property that an RV S_l with distribution $f(x \mid z)$ can be generated as a linear combination of z and a "fresh" standard Gaussian RV Y as $S_l \triangleq z/2 + \sqrt{n/2} \cdot Y$, since if we plug Gaussian pdfs $\phi_n(\cdot)$ and $\phi_{2n}(\cdot)$ into (1), $f(x \mid z)$ is precisely the pdf of $\mathcal{N}(z/2, n/2)$. Here, Y being "fresh" means it is independent of all other RVs.

This linearly decomposable property has a pleasant consequence that *every* dyadic range-sum generated by this 1D Gaussian-DST can be recursively decomposed to a linear combination of some i.i.d. standard Gaussian RVs, as illustrated in Figure 1b. In this example, let Y_0, Y_1, Y_2 and Y_3 be four i.i.d. standard Gaussian RVs. The total sum of the universe $S[0,4)$ is written as $2Y_0$, because they have the same distribution $\mathcal{N}(0,4)$. Then, it is split into two half-range-sums $S[0,2) \triangleq Y_0 + Y_1$ and $S[2,4) \triangleq Y_0 - Y_1$ using the linear decomposition above with $z = 2Y_0$ and a fresh RV Y_1. Finally, $S[0,2)$ and $S[2,4)$ are similarly split into the four underlying RVs using fresh RVs Y_2 and Y_3, respectively.

3.2 HWT Representation of 1D Gaussian-DST

In this section, we show that when the target distribution is Gaussian, a DST is mathematically equivalent to a Haar wavelet transform (HWT) in the 1D case. We will also show that this equivalence carries over to higher dimensions. Note that this equivalence does not apply to any target distribution other than Gaussian, and hence the HWT representation cannot replace the role of DST in general. In the following, we describe in Subsubsection 3.2.2 our HWT-based 1D Gaussian-ERS solution that has $O(\log \Delta)$ time complexity, after making some mathematical preparations in Subsubsection 3.2.1.

3.2.1 Mathematical Preliminaries

It is not hard to verify that, if we apply HWT (to be specified soon) to the four underlying RVs shown in Figure 1b, namely $X_0 = Y_0/2 + Y_1/2 + Y_2/\sqrt{2}$, $X_1 = Y_0/2 + Y_1/2 - Y_2/\sqrt{2}$, $X_2 = Y_0/2 - Y_1/2 + Y_3/\sqrt{2}$, and $X_3 = Y_0/2 - Y_1/2 - Y_3/\sqrt{2}$, then the four HWT coefficients we obtain are precisely Y_0, Y_1, Y_2, Y_3, respectively. In other words, we have $\vec{\mathbf{y}} = H_4\vec{\mathbf{x}}$, where $\vec{\mathbf{x}} \triangleq (X_0, X_1, X_2, X_3)^T$, $\vec{\mathbf{y}} \triangleq (Y_0, Y_1, Y_2, Y_3)^T$, and H_4 is the 4×4 Haar matrix H_4. This example is illustrated as a matrix-vector multiplication in Figure 2.

The above example in which $\Delta = 4$ can be generalized to an arbitrary universe size Δ (that is a power of 2) as follows. In general, HWT is defined as $\vec{\mathbf{w}} = H_\Delta\vec{\mathbf{x}}$, where $\vec{\mathbf{w}}$ and $\vec{\mathbf{x}}$ are both Δ-dimensional vectors, and H_Δ is a $\Delta \times \Delta$ Haar matrix. To simplify notations, we drop the subscript Δ in the sequel. In wavelet terms, $\vec{\mathbf{x}}$ is called a discrete signal vector and $\vec{\mathbf{w}}$ is called the *HWT coefficient* vector. Clearly, the i^{th} HWT coefficient is the inner product between $\vec{\mathbf{x}}$ and the i^{th} row of H, for $i = 0, 1, \cdots, \Delta - 1$. In the wavelet theory, we index each HWT coefficient as W_j^m (instead of W_i) for $m = -1, 0, 1, \cdots, \log_2 \Delta - 1$ and $j = 0, 1, \cdots, 2^{m^+} - 1$ (where $m^+ \triangleq \max\{0, m\}$) in the dictionary order of (m, j), and refer to the corresponding row (transposed into a column vector) in H that computes W_j^m as the *HWT vector* $\vec{\boldsymbol{\psi}}_j^m$. Hence we have $W_j^m \triangleq \langle \vec{\mathbf{x}}, \vec{\boldsymbol{\psi}}_j^m \rangle$ by definition. In wavelet terms, parameter m is called scale and parameter j is called location. In Figure 2, the 4 HWT coefficients and 4 HWT vectors from top to bottom are on 3 different scales ($-1, 0$, and 1) and are "assigned" 3 different colors accordingly.

We define the *indicator vector* of a 1D range R, denoted as $\mathbb{1}_R$, as a Δ-dimensional 0-1 vector, the i^{th} scalar of which takes value 1 if $i \in R$ and 0 otherwise, for $i = 0, 1, \cdots, \Delta - 1$. Throughout this paper, the indicator vectors are the only vectors that are not written in boldface with a rightward arrow on the top. We now specify the HWT vectors. Every HWT vector $\vec{\boldsymbol{\psi}}_j^m$ is normalized such that $\|\vec{\boldsymbol{\psi}}_j^m\|_2 = 1$. The first HWT vector $\vec{\boldsymbol{\psi}}_0^{-1} \triangleq \Delta^{-1/2} \cdot \mathbb{1}_{[0,\Delta)}$ is special: Its corresponding coefficient W_0^{-1} reflects the scaled (by $\Delta^{-1/2}$) range-sum of the entire universe, whereas every other HWT coefficient is the (scaled) difference of two range-sums. Every other HWT vector $\vec{\boldsymbol{\psi}}_j^m$, for $m = 0, 1, \cdots, \log_2 \Delta - 1$ and $j = 0, 1, \cdots, 2^m - 1$, corresponds to the dyadic range $I_j^m \triangleq [j\Delta/2^m, (j+1)\Delta/2^m)$ in the sense the latter serves as the support of the former: $\vec{\boldsymbol{\psi}}_j^m$ is defined by setting the first half of I_j^m to the value $\sqrt{2^m/\Delta}$, the second half of I_j^m to the value $-\sqrt{2^m/\Delta}$, and the rest of the universe $[0, \Delta) \setminus I_j^m$ to the value 0. Note that $\vec{\boldsymbol{\psi}}_j^m$ has the same number of scalars with value $\sqrt{2^m/\Delta}$ as those with value $-\sqrt{2^m/\Delta}$, so $\langle \vec{\boldsymbol{\psi}}_j^m, \mathbb{1}_{I_j^m} \rangle = 0$. From the definition above, H is known to be orthonormal [17], so the following theorem can be applied to it.

▶ **Theorem 1** ([13]). *Let M be an $n \times n$ matrix. If M is orthonormal, then it has the following two properties:*
1. *$M^T = M^{-1}$, and M^T is also orthonormal.*
2. *Given any two n-dimensional vectors $\vec{\mathbf{x}}, \vec{\mathbf{y}}$, we have $\langle \vec{\mathbf{x}}, \vec{\mathbf{y}} \rangle = \langle M\vec{\mathbf{x}}, M\vec{\mathbf{y}} \rangle$.*

Let $\vec{\mathbf{w}}$ be a Δ-dimensional vector of i.i.d. standard Gaussian RVs. We mathematically define the vector of underlying RVs $\vec{\mathbf{x}} = (X_0, X_1, \cdots, X_{\Delta-1})^T$ as $\vec{\mathbf{x}} \triangleq H^T \vec{\mathbf{w}}$. Hence, we have $\vec{\mathbf{w}} = H\vec{\mathbf{x}}$ by the first property in Theorem 1. The underlying RVs defined this way are i.i.d. standard Gaussian, by the following theorem.

▶ **Theorem 2** (Proposition 3.3.2 in [25]). *Let $\vec{\mathbf{x}} = M\vec{\mathbf{w}}$ where M is an orthonormal matrix. Then $\vec{\mathbf{x}}$ is a vector of i.i.d. standard Gaussian RVs if and only if $\vec{\mathbf{w}}$ is.*

3.2.2 Our HWT-based Algorithm for 1D-ERS

Given any range $[l, u)$, we compute its range-sum $S[l, u)$ as $\langle \vec{\mathbf{w}}, H\mathbb{1}_{[l,u)} \rangle$, which is the sum of the HWT coefficients in $\vec{\mathbf{w}}$ weighted by the scalars in $H\mathbb{1}_{[l,u)}$. This weighted sum can be computed in $O(\log \Delta)$ time, because, by Theorem 3, the Δ-dimensional vector $H\mathbb{1}_{[l,u)}$ contains only $O(\log \Delta)$ nonzero scalars (weights), and by Remark 5, for each such scalar, its index can be located and its value computed in $O(1)$ time. We refer to the $O(\log \Delta)$ corresponding scalars in $\vec{\mathbf{w}}$ whose weights are nonzero as *participating HWT coefficients* in the sequel.

To provide the aforementioned ideal guarantee of *mutual independence* (among the Δ underlying RVs), for each such participating HWT coefficient (which is a standard Gaussian RV), we generate the RV and remember its realization (in memory) if it has never been generated before (say for answering an earlier range-sum query), or retrieve its realization from memory otherwise. The space complexity of this algorithm is $O(\min\{T \log \Delta, \Delta\})$, since each of the T range-sum queries involves $O(\log \Delta)$ participating HWT coefficients. This algorithm satisfies the aforementioned consistency requirement, because $\langle \vec{\mathbf{w}}, H\mathbb{1}_{[l,u)}\rangle = \langle H\vec{\mathbf{x}}, H\mathbb{1}_{[l,u)}\rangle = \langle \vec{\mathbf{x}}, \mathbb{1}_{[l,u)}\rangle = X_l + X_{l+1} + \cdots + X_{u-1}$. The second equation above is by the second property in Theorem 1.

▶ **Theorem 3.** *Given any range $[l,u) \subseteq [0,\Delta)$, $H\mathbb{1}_{[l,u)}$ contains at most $2\log_2 \Delta + 2$ nonzero scalars.*

Theorem 3 is a straightforward corollary of Lemma 4, since H has only $\log_2 \Delta + 1$ scales.

▶ **Lemma 4.** *Given any range $[l,u) \subseteq [0,\Delta)$, $H\mathbb{1}_{[l,u)}$ contains at most 2 nonzero scalars on each scale.*

Proof. On scale $m = -1$, there is only one HWT coefficient anyway, so the claim trivially holds. We next prove the claim for any fixed $m \geq 0$. For each HWT vector $\vec{\boldsymbol{\psi}}_j^m$, $j = 0, 1, \cdots, 2^m - 1$, we denote the corresponding HWT coefficient as $r_j^m \triangleq \langle \vec{\boldsymbol{\psi}}_j^m, \mathbb{1}_{[l,u)}\rangle$. It is not hard to verify that the relationship between the range $[l,u)$ and the dyadic range I_j^m must be one of the following three cases.

1. I_j^m and $[l,u)$ are disjoint. In this case, $r_j^m = 0$.
2. $I_j^m \subseteq [l,u)$. In this case, $r_j^m = \langle \vec{\boldsymbol{\psi}}_j^m, \mathbb{1}_{I_j^m}\rangle = 0$ as explained in the second last sentence above Theorem 1.
3. Otherwise, I_j^m partially intersects $[l,u)$. This case may happen only to at most two (I_j^m)'s: the one that covers l and the one that covers $u-1$. In this case, r_j^m can be nonzero. ◀

▶ Remark 5. Each scalar r_j^m (in $H\mathbb{1}_{[l,u)}$) that may be nonzero can be identified and computed in $O(1)$ time as follows. Note r_j^m may be nonzero only in the case (3) above, in which j is equal to either $\lfloor l2^m/\Delta \rfloor$ or $\lfloor (u-1)2^m/\Delta \rfloor$. As a result, if $r_j^m \neq 0$, its value can be computed in two steps [22]. First, intersect $[l,u)$ with the first half and the second half of I_j^m, respectively. Second, scale the size of the first intersection minus the size of the second by $\sqrt{2^m/\Delta}$, as was explained by the third last sentence above Theorem 1.

The following lemma is a special case of Lemma 4 where $l = u-1$. This lemma holds, because in case (3) above, for $m = -1, 0, 1, \cdots, \log_2 \Delta - 1$, there exists a unique dyadic interval I_j^m that covers l (namely, the one with $j = \lfloor l2^m/\Delta \rfloor$).

▶ **Lemma 6.** *Given any $l \in [0,\Delta)$, $H\mathbb{1}_{\{l\}}$ has exactly one nonzero scalar on each scale.*

3.3 Range-Summable Gaussian RVs in 2D

In the following, we describe in Subsubsection 3.3.2 our 2D Gaussian-ERS solution that has $O(\log^2 \Delta)$ time complexity, after making some mathematical preparations in Subsubsection 3.3.1.

3.3.1 Mathematical Preliminaries

Like in the 1D case, our 2D Gaussian-ERS solution builds on the 2D-HWT $\vec{\mathbf{w}} = H^{\otimes 2}\vec{\mathbf{x}}$. Here the vector $\vec{\mathbf{x}}$ is comprised of the Δ^2 underlying RVs $X_{\vec{\mathbf{i}}}$ for $\vec{\mathbf{i}} \in [0,\Delta)^2$, listed in the dictionary order; and the vector $\vec{\mathbf{w}}$ is comprised of the resulting Δ^2 2D-HWT coefficients. The $\Delta^2 \times \Delta^2$ 2D-HWT matrix $H^{\otimes 2}$ is the self *Kronecker product* (defined next) of the $\Delta \times \Delta$ 1D-HWT matrix H.

▶ **Definition 7.** *Let A be a $p \times q$ matrix and B be a $t \times v$ matrix. Then their Kronecker product $A \otimes B$ is the following $pt \times qv$ matrix.*

$$A \otimes B \triangleq \begin{pmatrix} a_{11}B & \cdots & a_{1q}B \\ \vdots & \ddots & \vdots \\ a_{p1}B & \cdots & a_{pq}B \end{pmatrix}.$$

We now state two theorems concerning the Kronecker product.

▶ **Theorem 8** (Theorem 13.3 in [13])**.** *Let P, Q, R, T be four matrices such that the matrix products $P \cdot R$ and $Q \cdot T$ are well-defined. Then $(P \otimes Q) \cdot (R \otimes T) = (P \cdot R) \otimes (Q \cdot T)$.*

▶ **Theorem 9** (Corollary 13.8 in [13])**.** *The Kronecker product of two orthonormal matrices is also orthonormal.*

Now we describe the Δ^2 2D HWT coefficients and the order in which they are listed in $\vec{\mathbf{w}}$. Recall that in the 1D case, each HWT coefficient takes the form W_j^m, where m is the scale, and the j is the location. In the 2D case, each dimension has its own pair of scale and location parameters that is independent of the other dimension. For convenience of presentation, we refer to these two dimensions as vertical (the first) and horizontal (the second), respectively. We denote the vertical scale and location pair as m_1 and j_1, and the horizontal pair as m_2 and j_2. Each HWT coefficient takes the form $W_{j_1,j_2}^{m_1,m_2}$. In the 2D case, there are $(\log_2 \Delta + 1)^2$ scales, namely (m_1, m_2) for $m_1, m_2 = -1, 0, 1, \cdots, \log_2 \Delta - 1$. At each scale (m_1, m_2), there are $n_{m_1,m_2} \triangleq 2^{m_1^+ + m_2^+}$ locations, namely (j_1, j_2) for $j_1 = 0, 1, \cdots, 2^{m_1^+} - 1$ and $j_2 = 0, 1, \cdots, 2^{m_2^+} - 1$.

We now give a 2D example in which $\Delta = 4$. In this 2D example, there are $\Delta^2 = 16$ HWT coefficients. To facilitate the "color coding" of different scales, we arrange the 16 HWT coefficients into a 4×4 matrix $\mathbf{W}$ shown in Figure 3. $\mathbf{W}$ is the only matrix that we write in boldface in order to better distinguish it from its scalars ($W_{j_1,j_2}^{m_1,m_2}$)'s. Figure 3 contains three differently colored rows of heights 1, 1, and 2 respectively, that correspond to vertical scales $m_1 = -1, 0, 1$ respectively, and contains three differently colored columns that correspond to the three horizontal scales. Their "Cartesian product" contains 9 "color cells" that correspond to the 9 different scales (values of (m_1, m_2)). For example, the cell colored in pink corresponds to scale $(1, 1)$ and contains 4 HWT coefficients $W_{0,0}^{1,1}$, $W_{0,1}^{1,1}$, $W_{1,0}^{1,1}$, $W_{1,1}^{1,1}$. The vector $\vec{\mathbf{w}}$ is defined from $\mathbf{W}$ by flattening its 16 scalars in the row-major order, as shown at the bottom of Figure 3.

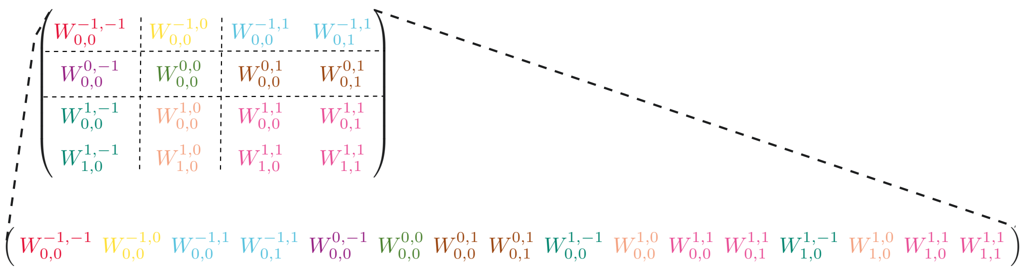

$W_{0,0}^{-1,-1}$ $W_{0,0}^{-1,0}$ $W_{0,0}^{-1,1}$ $W_{0,1}^{-1,1}$
$W_{0,0}^{0,-1}$ $W_{0,0}^{0,0}$ $W_{0,0}^{0,1}$ $W_{0,1}^{0,1}$
$W_{0,0}^{1,-1}$ $W_{0,0}^{1,0}$ $W_{0,0}^{1,1}$ $W_{0,1}^{1,1}$
$W_{1,0}^{1,-1}$ $W_{1,0}^{1,0}$ $W_{1,0}^{1,1}$ $W_{1,1}^{1,1}$
$W_{0,0}^{-1,-1}$ $W_{0,0}^{-1,0}$ $W_{0,0}^{-1,1}$ $W_{0,1}^{-1,1}$ $W_{0,0}^{0,-1}$ $W_{0,0}^{0,0}$ $W_{0,0}^{0,1}$ $W_{0,1}^{0,1}$ $W_{0,0}^{1,-1}$ $W_{0,0}^{1,0}$ $W_{0,0}^{1,1}$ $W_{0,1}^{1,1}$ $W_{1,0}^{1,-1}$ $W_{1,0}^{1,0}$ $W_{1,0}^{1,1}$ $W_{1,1}^{1,1}$

Figure 3 The 2D-HWT coefficients, arranged both as a matrix $\mathbf{W}$ and as a flattened vector $\vec{\mathbf{w}}^T$.

Like in the 1D case, let $\vec{\mathbf{w}}$ be a vector of Δ^2 i.i.d. standard Gaussian RVs. As explained earlier, the vector $\vec{\mathbf{x}}$ of Δ^2 underlying RVs are *mathematically* defined as $\vec{\mathbf{x}} \triangleq (H^{\otimes 2})^T \vec{\mathbf{w}}$. The RVs in $\vec{\mathbf{x}}$ are i.i.d. standard Gaussian by Theorem 2, because $H^{\otimes 2}$ is an orthonormal matrix by Theorem 9.

3.3.2 Our HWT-Based Algorithm for 2D-ERS

Our 2D-ERS algorithm (that guarantees mutual independence among the underlying RVs) is similar to the 1D-ERS algorithm described earlier. Given any 2D range $[\vec{\mathbf{l}}, \vec{\mathbf{u}}) \triangleq [l_1, u_1) \times [l_2, u_2)$, where $\vec{\mathbf{l}} = (l_1, l_2)^T$ and $\vec{\mathbf{u}} = (u_1, u_2)^T$, we compute its range-sum $S[\vec{\mathbf{l}}, \vec{\mathbf{u}})$ as $\langle \vec{\mathbf{w}}, H^{\otimes 2} \mathbb{1}_{[\vec{\mathbf{l}}, \vec{\mathbf{u}})} \rangle$. Here the 2D indicator vector $\mathbb{1}_{[\vec{\mathbf{l}}, \vec{\mathbf{u}})}$ is defined as the result of flattening the following $\Delta \times \Delta$ matrix in row-major order: For $\vec{\mathbf{i}} \in [0, \Delta)^2$, the $\vec{\mathbf{i}}^{th}$ scalar in the matrix takes value 1 if $\vec{\mathbf{i}} \in [\vec{\mathbf{l}}, \vec{\mathbf{u}})$ and takes value 0 otherwise. This return value $\langle \vec{\mathbf{w}}, H^{\otimes 2} \mathbb{1}_{[\vec{\mathbf{l}}, \vec{\mathbf{u}})} \rangle$ can be computed in $O(\log^2 \Delta)$ time, since it involves generating, and computing the weighted sum of, $O(\log^2 \Delta)$ participating HWT coefficients according to Lemma 10. The space complexity is $O(\min\{T \log^2 \Delta, \Delta^2\})$ for remembering the realizations of the $O(\log^2 \Delta)$ participating HWT coefficients (per query) like that explained earlier in the 1D case. Our 2D-ERS algorithm meets the consistency requirement, because $\langle \vec{\mathbf{w}}, H^{\otimes 2} \mathbb{1}_{[\vec{\mathbf{l}}, \vec{\mathbf{u}})} \rangle = \langle H^{\otimes 2} \vec{\mathbf{x}}, H^{\otimes 2} \mathbb{1}_{[\vec{\mathbf{l}}, \vec{\mathbf{u}})} \rangle = \langle \vec{\mathbf{x}}, \mathbb{1}_{[\vec{\mathbf{l}}, \vec{\mathbf{u}})} \rangle = \sum_{(i_1, i_2) \in [\vec{\mathbf{l}}, \vec{\mathbf{u}})} X_{i_1, i_2}$.

▶ **Lemma 10.** *For any 2D range $[\vec{\mathbf{l}}, \vec{\mathbf{u}}) \subseteq [0, \Delta)^2$, $H^{\otimes 2} \mathbb{1}_{[\vec{\mathbf{l}}, \vec{\mathbf{u}})}$ has $O(\log^2 \Delta)$ nonzero scalars.*

Proof. It is not hard to verify $\mathbb{1}_{[\vec{\mathbf{l}}, \vec{\mathbf{u}})} = \mathbb{1}_{[l_1, u_1)} \otimes \mathbb{1}_{[l_2, u_2)}$. By Theorem 8, $H^{\otimes 2} \mathbb{1}_{[\vec{\mathbf{l}}, \vec{\mathbf{u}})} = (H \otimes H) \cdot (\mathbb{1}_{[l_1, u_1)} \otimes \mathbb{1}_{[l_2, u_2)}) = (H \mathbb{1}_{[l_1, u_1)}) \otimes (H \mathbb{1}_{[l_2, u_2)})$. By Theorem 3, both $H \mathbb{1}_{[l_1, u_1)}$ and $H \mathbb{1}_{[l_2, u_2)}$ have $O(\log \Delta)$ nonzero scalars, so their Kronecker product has $O(\log^2 \Delta)$ nonzero scalars. ◀

3.4 Generalization to Higher Dimensions

Our HWT-based Gaussian-ERS solution, just like HWT itself, can be naturally generalized to higher dimensions as follows. In dimension $d > 2$, we continue to have the inverse HWT formula $\vec{\mathbf{x}} \triangleq M^T \vec{\mathbf{w}}$, where $\vec{\mathbf{x}}$ is the vector of Δ^d underlying RVs (arranged in dictionary order of $\vec{\mathbf{i}}$), $\vec{\mathbf{w}}$ is the vector of their HWT coefficients (that are i.i.d. standard Gaussian RVs), and M is the $\Delta^d \times \Delta^d$ HWT matrix in dD. Here $M \triangleq \underbrace{H \otimes \cdots \otimes H}_{d}$, where H is the 1D Haar matrix described above. Since M is orthonormal by Theorem 9, the RVs in $\vec{\mathbf{x}}$ are i.i.d. standard Gaussian by Theorem 2.

In our dD-ERS algorithm (that guarantees mutual independence among the underlying RVs), given a dD range $[\vec{\mathbf{l}}, \vec{\mathbf{u}}) \triangleq [l_1, u_1) \times [l_2, u_2) \times \cdots \times [l_d, u_d)$, its range-sum $S[\vec{\mathbf{l}}, \vec{\mathbf{u}})$ can be computed as $\langle \vec{\mathbf{w}}, M \mathbb{1}_{[\vec{\mathbf{l}}, \vec{\mathbf{u}})} \rangle$, because $\langle \vec{\mathbf{w}}, M \mathbb{1}_{[\vec{\mathbf{l}}, \vec{\mathbf{u}})} \rangle = \langle M \vec{\mathbf{x}}, M \mathbb{1}_{[\vec{\mathbf{l}}, \vec{\mathbf{u}})} \rangle = \langle \vec{\mathbf{x}}, \mathbb{1}_{[\vec{\mathbf{l}}, \vec{\mathbf{u}})} \rangle = \sum_{\vec{\mathbf{i}} \in [\vec{\mathbf{l}}, \vec{\mathbf{u}})} X_{\vec{\mathbf{i}}}$. The weighted sum $\langle \vec{\mathbf{w}}, M \mathbb{1}_{[\vec{\mathbf{l}}, \vec{\mathbf{u}})} \rangle$ can be computed in $O(\log^d \Delta)$ time, because the weight vector $M \mathbb{1}_{[\vec{\mathbf{l}}, \vec{\mathbf{u}})} = M \cdot (\mathbb{1}_{[l_1, u_1)} \otimes \mathbb{1}_{[l_2, u_2)} \otimes \cdots \otimes \mathbb{1}_{[l_d, u_d)}) = (H \mathbb{1}_{[l_1, u_1)}) \otimes (H \mathbb{1}_{[l_2, u_2)}) \otimes \cdots \otimes (H \mathbb{1}_{[l_d, u_d)})$ has only $O(\log^d \Delta)$ nonzero scalars (weights) by Theorem 3 and the property of Kronecker product. Hence, we need to generate and remember only $O(\log^d \Delta)$ corresponding participating HWT coefficients. As a result, our dD-ERS algorithm has $O(\min\{T \log^d \Delta, \Delta^d\})$ space complexity.

4 k-wise Independence Theory

In this section, in all subsequent paragraphs, we assume $d = 2$ (2D) for notational simplicity. All our statements and proofs can be readily generalized to higher dimensions. Recall that, for guaranteeing mutual independence among the Δ^d underlying RVs, our HWT-based dD Gaussian-ERS needs to *remember* (the realization of) every participating HWT coefficient that was generated for answering a past range-sum query, which can lead to high storage overhead when the number of queries T is large. In this section we propose a k-wise independence theory and scheme that guarantees that the Δ^d underlying Gaussian RVs are k-wise independent. It does so by using $O(\log^d \Delta)$ k-wise independent hash functions (described next) instead. This scheme has the same time complexity of $O(\log^d \Delta)$ as the idealized Gaussian-ERS solution, and a much smaller space complexity of $O(\log^d \Delta)$, for storing the seeds of $O(\log^d \Delta)$ k-wise independent hash functions. This scheme significantly extends its 1D version proposed in [15]. Finally, we note this scheme works also for our Poisson-ERS solution. We however will not explain how it works in this paper, since doing so would involve drilling down to the messy and lengthy detail of the Cartesian product of $d > 1$ DSTs (since we cannot use the relatively clean and simple dD HWT in the Poisson case).

A k-wise independent hash function $h(\cdot)$ has the following property: Given an arbitrary set of k distinct keys $i_1, i_2, \cdots, i_k$, their hash values $h(i_1), h(i_2), \cdots, h(i_k)$ are independent. Such hash functions are very computationally efficient when k is a small number such as $k = 2$ (roughly 2 nanoseconds per hash) and $k = 4$ (several nanoseconds per hash) [3, 23, 19]. Typically, the hash values are (uniform random) integers. We can map them to Gaussian RVs using a deterministic function $g(\cdot)$ such as the Box-Muller transform [20].

Recall (from Figure 3) that the Δ^2 HWT coefficients in the vector $\vec{\mathbf{w}}$ are on $(\log_2 \Delta + 1)^2$ different scale pairs, namely (m_1, m_2) for $m_1, m_2 = -1, 0, 1, \cdots, \log_2 \Delta - 1$. Our scheme uses $(\log_2 \Delta + 1)^2$ independent k-wise independent hash functions that we denote as $h_{m_1,m_2}(\cdot)$, for $m_1, m_2 = -1, 0, 1, \cdots, \log_2 \Delta - 1$. During the initialization phase, we uniformly randomly seed these $(\log_2 \Delta + 1)^2$ hash functions; once seeded, they are fixed thereafter as usual. As mentioned earlier, these seeds correspond to the outcome ω that fixes (mathematically defines) the HWT coefficient vector $\vec{\mathbf{w}}$.

Our scheme can be stated *literally in one sentence:* Each such (seeded and fixed) $h_{m_1,m_2}(\cdot)$ is solely responsible for hash-generating any HWT coefficient on scale (m_1, m_2) that is *participating* (as defined earlier) in answering a range-sum query. In other words, for any scale $m_1, m_2 = -1, 0, 1, \cdots, \log_2 \Delta - 1$, and location $j_1 = 0, 1, \cdots, 2^{m_1^+} - 1$, $j_2 = 0, 1, \cdots, 2^{m_2^+} - 1$, the value of the HWT coefficient $W_{j_1,j_2}^{m_1,m_2}$ is *mathematically defined* as $g(h_{m_1,m_2}(j_1, j_2))$, where $g(\cdot)$ is the aforementioned deterministic function (that maps an integer to a Gaussian RV). Hence our scheme has a much lower space complexity of $O(\log^2 \Delta)$, for remembering the seeds of the $O(\log^2 \Delta)$ hash functions.

The following theorem states that our scheme achieves its intended objective of ensuring that the Δ^2 underlying RVs *mathematically defined* by it are k-wise independent. In this theorem and proof, we denote the vector of Δ^2 HWT coefficients and the vector of Δ^2 underlying RVs both mathematically defined by our k-wise scheme as $\vec{\mathbf{v}}$ and $\vec{\mathbf{z}}$, respectively. We do so to distinguish this vector pair from the original vector pair $\vec{\mathbf{w}}$ and $\vec{\mathbf{x}}$ that are mathematically defined by the idealized scheme (that guarantees mutual independence). Recall that $\vec{\mathbf{z}} = M^T \vec{\mathbf{v}}$ and $\vec{\mathbf{x}} = M^T \vec{\mathbf{w}}$, where $M = H^{\otimes 2}$ is the 2D HWT matrix, and that $\vec{\mathbf{x}}$ is comprised of i.i.d. standard Gaussian RVs.

▶ **Theorem 11.** *The vector $\vec{\mathbf{z}}$ is comprised of k-wise independent standard Gaussian RVs.*

Proof. It suffices to prove that *any* k distinct scalars in $\vec{\mathbf{z}}$ – say the $(i_1)^{th}$, $(i_2)^{th}$, $\cdots$, $(i_k)^{th}$ scalars – are i.i.d. standard Gaussian. Let $\vec{\mathbf{z}}'$ be the k-dimensional vector comprised of these k scalars. Let $(M^T)'$ be the $k \times \Delta^2$ matrix formed by the $(i_1)^{th}, (i_2)^{th}, \cdots, (i_k)^{th}$ rows in M^T. Then, we have $\vec{\mathbf{z}}' = (M^T)'\vec{\mathbf{v}}$. Now let the random vector $\vec{\mathbf{x}}'$ be defined as $(M^T)'\vec{\mathbf{w}}$. Then $\vec{\mathbf{x}}'$ is comprised of k i.i.d. standard Gaussian RVs, as its scalars are a subset of those of $\vec{\mathbf{x}}$. Hence, to prove that the scalars in $\vec{\mathbf{z}}'$ are i.i.d. standard Gaussian RVs, it suffices to prove the claim that $\vec{\mathbf{z}}'$ has the same distribution as $\vec{\mathbf{x}}'$.

We prove this claim using Proposition 12. To this end, we first write $\vec{\mathbf{z}}'$ and $\vec{\mathbf{x}}'$ each as the sum of $N = (\log_2 \Delta + 1)^2$ independent random vectors. Recall that in Subsection 3.3, we have classified the HWT coefficients in $\vec{\mathbf{w}}$ and $\vec{\mathbf{v}}$, and the columns of M^T (called HWT vectors there) into N different (m_1, m_2) scales (colors in Figure 3). Recall that n_{m_1,m_2} scalars in $\vec{\mathbf{w}}$ and $\vec{\mathbf{v}}$, and accordingly n_{m_1,m_2} columns of M^T, have scale (m_1, m_2). Let $\vec{\mathbf{w}}_{m_1,m_2}$ and $\vec{\mathbf{v}}_{m_1,m_2}$ be the n_{m_1,m_2}-dimensional vectors comprised of the coefficients classified to scale (m_1, m_2) in $\vec{\mathbf{w}}$ and $\vec{\mathbf{v}}$, respectively. Let $(M^T)'_{m_1,m_2}$ be the $k \times n_{m_1,m_2}$ matrix comprised of the columns of $(M^T)'$ classified to scale (m_1, m_2). Then, we have $\vec{\mathbf{z}}' = \sum_{(m_1,m_2)} (M^T)'_{m_1,m_2} \vec{\mathbf{v}}_{m_1,m_2}$ and $\vec{\mathbf{x}}' = \sum_{(m_1,m_2)} (M^T)'_{m_1,m_2} \vec{\mathbf{w}}_{m_1,m_2}$, where both summations are over all N scales. The N summands in the RHS of the first equation are independent random vectors, because for each scale $(m_1, m_2) \in [-1, \log_2 \Delta)^2$, all scalars in $\vec{\mathbf{v}}_{m_1,m_2}$ are generated by the same per-scale hash function $h_{m_1,m_2}(\cdot)$, which is independent of all $N-1$ other per-scale hash functions. The same can be said about the N summands in the RHS of the second equation, since $\vec{\mathbf{w}}$ is comprised of i.i.d. RVs by design. To prove this claim using Proposition 12, it remains to prove the fact that for each scale $(m_1, m_2) \in [-1, \log_2 \Delta)^2$, the pair of random vectors $(M^T)'_{m_1,m_2} \vec{\mathbf{v}}_{m_1,m_2}$ and $(M^T)'_{m_1,m_2} \vec{\mathbf{w}}_{m_1,m_2}$ have the same distribution.

This fact can be proved as follows. Note that for each scale $(m_1, m_2) \in [-1, \log_2 \Delta)^2$, each row in $(M^T)'_{m_1,m_2}$ has exactly one nonzero scalar, since the corresponding row in M^T, or equivalently the corresponding column in M, has exactly one nonzero scalar at each scale (m_1, m_2), due to Lemma 13. Therefore, although the number of columns in $(M^T)'_{m_1,m_2}$ can be as many as $O(\Delta^2)$, at most k of them (one for each row), say the $(\alpha_1)^{th}, (\alpha_2)^{th}, \cdots, (\alpha_k)^{th}$ columns, contain nonzero scalars. Then, $(M^T)'_{m_1,m_2} \vec{\mathbf{v}}_{m_1,m_2}$ is a function of only the $(\alpha_1)^{th}, (\alpha_2)^{th}, \cdots, (\alpha_k)^{th}$ scalars in $\vec{\mathbf{v}}_{m_1,m_2}$, and these k scalars are i.i.d. Gaussian RVs since they are all generated by the same k-wise independent hash function $h_{m_1,m_2}(\cdot)$. Note that $(M^T)'_{m_1,m_2} \vec{\mathbf{w}}_{m_1,m_2}$ is the same function of the $(\alpha_1)^{th}, (\alpha_2)^{th}, \cdots, (\alpha_k)^{th}$ scalars in $\vec{\mathbf{w}}_{m_1,m_2}$, which are i.i.d. Gaussian RVs by design. Hence, $(M^T)'_{m_1,m_2} \vec{\mathbf{v}}_{m_1,m_2}$ has the same distribution as $(M^T)'_{m_1,m_2} \vec{\mathbf{w}}_{m_1,m_2}$. ◀

▶ **Proposition 12.** *Suppose random vectors $\vec{\mathbf{x}}$ and $\vec{\mathbf{z}}$ each is the sum of N* independent *random vectors as follows: $\vec{\mathbf{x}} = \vec{\mathbf{x}}_1 + \vec{\mathbf{x}}_2 + \cdots + \vec{\mathbf{x}}_N$ and $\vec{\mathbf{z}} = \vec{\mathbf{z}}_1 + \vec{\mathbf{z}}_2 + \cdots + \vec{\mathbf{z}}_N$. Then, $\vec{\mathbf{x}}$ and $\vec{\mathbf{z}}$ have the same distribution if each pair of components $\vec{\mathbf{x}}_i$ and $\vec{\mathbf{z}}_i$ have the same distribution, for $i = 1, 2, \cdots, N$.*

▶ **Lemma 13.** *Any column of $M = H^{\otimes 2}$, which is equal to $H^{\otimes 2} \mathbb{1}_{\{\vec{\mathbf{i}}\}}$ for some $\vec{\mathbf{i}} = (i_1, i_2)^T$, has exactly one nonzero scalar on each 2D scale (m_1, m_2).*

Proof. The 2D indicator vector can be decomposed to the Kronecker product of two 1D indicator vectors as $\mathbb{1}_{\{\vec{\mathbf{i}}\}} = \mathbb{1}_{\{i_1\}} \otimes \mathbb{1}_{\{i_2\}}$, so $H^{\otimes 2} \mathbb{1}_{\{\vec{\mathbf{i}}\}} = (H \mathbb{1}_{\{i_1\}}) \otimes (H \mathbb{1}_{\{i_2\}})$ by Theorem 8. The claim above follows from Lemma 6, which implies that $H \mathbb{1}_{\{i_1\}}$ and $H \mathbb{1}_{\{i_2\}}$ each has exactly one nonzero scalar on each 1D scale. ◀

5 Multidimensional Dyadic Simulation

As explained in Subsection 3.1, in one dimension (1D), any dyadic range-sum $S[l, u)$, *no matter what* the target distribution is, can be computed by performing $O(\log \Delta)$ *binary splits* along the path from the root $S[0, \Delta)$ to the node $S[l, u)$ along the dyadic simulation tree (DST). Since we have just *computationally efficiently* generalized the Gaussian-DST approach (equivalent to the HWT-based approach in the 1D Gaussian case) to any dimension $d \geq 2$, we wonder whether we can do the same for all target distributions. By "computationally efficiently", we mean that a generalized solution should be able to compute any dD range-sum in $O(\log^d \Delta)$ time like in the Gaussian case.

Unfortunately, it appears hard, if not impossible, to generalize the DST approach to dD for arbitrary target distributions. We have identified a sufficient condition on the target distribution for such an efficient generalization to exist. We prove the sufficiency by proposing a DST-based universal algorithmic framework (described in Appendix C of [14]) that solves the dD-ERS problem for any target distribution satisfying this condition. Unfortunately, so far only two distributions, namely Gaussian and Poisson, are known to satisfy this condition, as is elaborated in Appendix A of [14]. We also describe in Appendix B of [14] two example distributions that do not satisfy this sufficient condition, namely Cauchy and Rademacher. In the following, we specify this condition and explain why it is "almost necessary".

For ease of presentation, in the following, we fix the number of dimensions d at 2. We assume all underlying RVs, X_{i_1,i_2} for (i_1, i_2) in the 2D universe $[0, \Delta)^2$, are i.i.d. with a certain target distribution X. This assumption is appropriate for our reasoning below about the time complexity of a 2D ERS solution, since as shown earlier this time complexity is not affected by the strength of the independence guarantee provided, in the cases of Gaussian and Poisson. In a 2D universe, any 2D range can be considered the Cartesian product of its horizontal and vertical 1D ranges. We say a 2D range is *dyadic* if and only if its horizontal and vertical 1D ranges are both dyadic. Since any general (not necessarily dyadic) 1D range can be "pieced together" using $O(\log \Delta)$ 1D dyadic ranges [21], it is not hard to show, using the Cartesian product argument, that any general 2D range can be "pieced together" using $O(\log^2 \Delta)$ 2D dyadic ranges. Hence in the following, we focus on the generation of only 2D dyadic range-sums. We assume all underlying RVs, X_{i_1,i_2} for (i_1, i_2) in the 2D universe $[0, \Delta)^2$, are i.i.d. with a certain target distribution X.

We need to introduce some additional notations. We define each horizontal strip-sum $S^H_i \triangleq X_{i,0} + X_{i,1} + \cdots + X_{i,\Delta-1}$ for $i \in [0, \Delta)$ as the sum of range $[i, i + 1) \times [0, \Delta)$, and each vertical strip-sum $S^V_i \triangleq X_{0,i} + X_{1,i} + \cdots + X_{\Delta-1,i}$ for $i \in [0, \Delta)$ as the sum of range $[0, \Delta) \times [i, i + 1)$. We denote as S the total sum of all underlying RVs in the universe, *i.e.,* $S \triangleq \sum_{i_1=0}^{\Delta-1} \sum_{i_2=0}^{\Delta-1} X_{i_1,i_2} = \sum_{i=0}^{\Delta-1} S^H_i = \sum_{i=0}^{\Delta-1} S^V_i$.

Now we are ready to state this sufficient condition. For ease of presentation, we break it down into two parts. The first part, stated in the following formula, states that the vector of vertical strip-sums and the vector of horizontal strip-sums in $[0, \Delta)^2$ are conditionally independent given the total sum S.

$$(S^V_0, S^V_1, \cdots, S^V_{\Delta-1}) \perp\!\!\!\perp (S^H_0, S^H_1, \cdots, S^H_{\Delta-1}) \mid S. \tag{2}$$

The second part is that this conditional independence relation holds for the two corresponding vectors in any 2D dyadic range (that is not necessarily a square). Intuitively, this condition says that how a 2D dyadic range-sum is split horizontally is conditionally independent (upon this 2D range-sum) of how it is split vertically. Roughly speaking, this condition implies that the 1D-DST governing the horizontal splits is conditionally independent of the other 1D-DST governing the vertical splits. Hence, our DST-based universal algorithmic framework for 2D can be viewed as the Cartesian product of the two 1D-DSTs, as will be elaborated in Appendix C of [14].

In the following, we offer some intuitive evidence why this condition is likely necessary. Without loss of generality, we consider the generation of an arbitrary horizontal strip-sum $S^H_{i_1}$ conditional on the vector of vertical strip-sums $(S^V_0, S^V_1, \cdots, S^V_{\Delta-1})$. Suppose (2) does not hold, which means $(S^H_0, S^H_1, \cdots, S^H_{\Delta-1})$ is not conditionally independent of $(S^V_0, S^V_1, \cdots, S^V_{\Delta-1})$ given S. Then the distribution of $S^H_{i_1}$ is arguably parameterized by the values (realizations) of all Δ vertical strip-sums $S^V_0, S^V_1, \cdots, S^V_{\Delta-1}$, since $S^H_{i_1}$ and any vertical strip-sum $S^V_{i_2}$ for $i_2 \in [0, \Delta)$ are in general dependent RVs by Theorem 14 (See Appendix D of [14] for its nontrivial proof). Hence, unless some magic happens (which we cannot rule out rigorously), to generate (realize) the RV $S^H_{i_1}$, conceivably we need to first realize all Δ RVs $(S^V_0, S^V_1, \cdots, S^V_{\Delta-1})$, the time complexity of which is $\Omega(\Delta)$.

▶ **Theorem 14.** *For any (i_1, i_2) in $[0, \Delta)^2$, $S^H_{i_1}$ and $S^V_{i_2}$ are dependent RVs unless the target distribution X satisfies* $\Pr[X = c] = 1$ *for some constant c.*

6 Conclusion

In this work, we propose novel solutions to dD-ERS for RVs that have Gaussian or Poisson distribution. Our solutions are the first ones that compute any multi-dimensional range-sum in polylogarithmic time. Our dD Gaussian-ERS scheme solves the long-standing open problem of efficiently answering approximate range-sum queries over a multidimensional data cube. We develop a novel k-wise independence theory that provides both high computational efficiencies and strong provable independence guarantees. Finally, we show that when the underlying distribution satisfies a sufficient and likely necessary condition, its DST-based 1D-ERS solution can be generalized to higher dimensions.

References

1 Noga Alon, Yossi Matias, and Mario Szegedy. The space complexity of approximating the frequency moments. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing*, STOC '96, pages 20–29, New York, NY, USA, 1996. Association for Computing Machinery. `doi:10.1145/237814.237823`.

2 A. R. Calderbank, A. Gilbert, K. Levchenko, S. Muthukrishnan, and M. Strauss. Improved range-summable random variable construction algorithms. In *Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '05, pages 840–849, USA, 2005. Society for Industrial and Applied Mathematics. URL: `http://dl.acm.org/citation.cfm?id=1070432.1070550`.

3 J. Lawrence Carter and Mark N. Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences*, 18(2):143–154, 1979. `doi:10.1016/0022-0000(79)90044-8`.

4 Graham Cormode, Minos Garofalakis, and Dimitris Sacharidis. Fast approximate wavelet tracking on streams. In Yannis Ioannidis, Marc H. Scholl, Joachim W. Schmidt, Florian Matthes, Mike Hatzopoulos, Klemens Boehm, Alfons Kemper, Torsten Grust, and Christian Boehm, editors, *Advances in Database Technology – EDBT 2006*, pages 4–22, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. `doi:10.1007/11687238_4`.

5 Joan Feigenbaum, Sampath Kannan, Martin J. Strauss, and Mahesh Viswanathan. An approximate L1-difference algorithm for massive data streams. *SIAM Journal on Computing*, 32(1):131–151, 2002. `doi:10.1137/S0097539799361701`.

6 Anna C. Gilbert, Sudipto Guha, Piotr Indyk, Yannis Kotidis, S. Muthukrishnan, and Martin J. Strauss. Fast, small-space algorithms for approximate histogram maintenance. In *Proceedings of the Thiry-Fourth Annual ACM Symposium on Theory of Computing*, STOC '02, pages 389–398, New York, NY, USA, 2002. Association for Computing Machinery. `doi:10.1145/509907.509966`.

7 Anna C. Gilbert, Yannis Kotidis, S. Muthukrishnan, and Martin J. Strauss. One-pass wavelet decompositions of data streams. *IEEE Trans. on Knowl. and Data Eng.*, 15(3):541–554, March 2003. `doi:10.1109/TKDE.2003.1198389`.

8 J. Gray, A. Bosworth, A. Lyaman, and H. Pirahesh. Data cube: a relational aggregation operator generalizing GROUP-BY, CROSS-TAB, and SUB-TOTALS. In *Proceedings of the Twelfth International Conference on Data Engineering*, pages 152–159, 1996. `doi:10.1109/ICDE.1996.492099`.

9 Nabil Ibtehaz, M. Kaykobad, and M. Sohel Rahman. Multidimensional segment trees can do range updates in poly-logarithmic time. *Theoretical Computer Science*, 854:30–43, 2021. `doi:10.1016/j.tcs.2020.11.034`.

10 Piotr Indyk. Stable distributions, pseudorandom generators, embeddings, and data stream computation. *J. ACM*, 53(3):307–323, May 2006. `doi:10.1145/1147954.1147955`.

11 Mehrdad Jahangiri, Dimitris Sacharidis, and Cyrus Shahabi. SHIFT-SPLIT: I/O efficient maintenance of wavelet-transformed multidimensional data. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, SIGMOD '05, pages 275–286, New York, NY, USA, 2005. Association for Computing Machinery. `doi:10.1145/1066157.1066189`.

12 Panagiotis Karras and Nikos Mamoulis. The Haar+ tree: A refined synopsis data structure. In *2007 IEEE 23rd International Conference on Data Engineering*, pages 436–445, 2007. doi:10.1109/ICDE.2007.367889.

13 Alan J. Laub. *Matrix analysis – for scientists and engineers*. SIAM, 2005. URL: http://bookstore.siam.org/ot91/.

14 Jingfan Meng, Huayi Wang, Jun Xu, and Mitsunori Ogihara. On efficient range-summability of IID random variables in two or higher dimensions (extended version). *CoRR*, abs/2110.07753, 2021. arXiv:2110.07753.

15 Jingfan Meng, Huayi Wang, Jun Xu, and Mitsunori Ogihara. A Dyadic Simulation Approach to Efficient Range-Summability. In Dan Olteanu and Nils Vortmeier, editors, *25th International Conference on Database Theory (ICDT 2022)*, volume 220 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 17:1–17:18, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ICDT.2022.17.

16 S. Muthukrishnan and Martin Strauss. Maintenance of multidimensional histograms. In Paritosh K. Pandya and Jaikumar Radhakrishnan, editors, *FST TCS 2003: Foundations of Software Technology and Theoretical Computer Science*, pages 352–362, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg. doi:10.1007/978-3-540-24597-1_30.

17 Yves Nievergelt. *Multidimensional Wavelets and Applications*, pages 36–72. Birkhäuser Boston, Boston, MA, 1999. doi:10.1007/978-1-4612-0573-9_2.

18 Noam Nisan. Pseudorandom generators for space-bounded computation. *Comb.*, 12(4):449–461, 1992. doi:10.1007/BF01305237.

19 Mihai Pundefinedtraşcu and Mikkel Thorup. The power of simple tabulation hashing. *J. ACM*, 59(3), June 2012. doi:10.1145/2220357.2220361.

20 Christian P. Robert and George Casella. *Monte Carlo Statistical Methods*, page 43. Springer New York, 2004. doi:10.1007/978-1-4757-4145-2_2.

21 Florin Rusu and Alin Dobra. Pseudo-random number generation for sketch-based estimations. *ACM Trans. Database Syst.*, 32(2):11–es, June 2007. doi:10.1145/1242524.1242528.

22 Rolfe R. Schmidt and Cyrus Shahabi. Propolyne: A fast wavelet-based algorithm for progressive evaluation of polynomial range-sum queries. In Christian S. Jensen, Simonas Šaltenis, Keith G. Jeffery, Jaroslav Pokorny, Elisa Bertino, Klemens Böhn, and Matthias Jarke, editors, *Advances in Database Technology – EDBT 2002*, pages 664–681, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg. doi:10.1007/3-540-45876-X_41.

23 Mikkel Thorup and Yin Zhang. Tabulation based 4-universal hashing with applications to second moment estimation. In *Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '04, pages 615–624, USA, 2004. Society for Industrial and Applied Mathematics. URL: http://dl.acm.org/citation.cfm?id=982792.982884.

24 Mikkel Thorup and Yin Zhang. Tabulation based 5-universal hashing and linear probing. In *Proceedings of the Meeting on Algorithm Engineering and Expermiments*, ALENEX '10, pages 62–76, USA, 2010. Society for Industrial and Applied Mathematics. doi:10.1137/1.9781611972900.7.

25 Roman Vershynin. *Random Vectors in High Dimensions*, pages 38–69. Cambridge Series in Statistical and Probabilistic Mathematics. Cambridge University Press, 2018. doi:10.1017/9781108231596.006.

The Consistency of Probabilistic Databases with Independent Cells

Amir Gilad
Duke University, Durham, NC, USA

Aviram Imber
Technion – Israel Institute of Technology, Haifa, Israel

Benny Kimelfeld
Technion – Israel Institute of Technology, Haifa, Israel

Abstract

A probabilistic database with attribute-level uncertainty consists of relations where cells of some attributes may hold probability distributions rather than deterministic content. Such databases arise, implicitly or explicitly, in the context of noisy operations such as missing data imputation, where we automatically fill in missing values, column prediction, where we predict unknown attributes, and database cleaning (and repairing), where we replace the original values due to detected errors or violation of integrity constraints. We study the computational complexity of problems that regard the selection of cell values in the presence of integrity constraints. More precisely, we focus on functional dependencies and study three problems: (1) deciding whether the constraints can be satisfied by any choice of values, (2) finding a most probable such choice, and (3) calculating the probability of satisfying the constraints. The data complexity of these problems is determined by the combination of the set of functional dependencies and the collection of uncertain attributes. We give full classifications into tractable and intractable complexities for several classes of constraints, including a single dependency, matching constraints, and unary functional dependencies.

2012 ACM Subject Classification Information systems → Data management systems

Keywords and phrases Probabilistic databases, attribute-level uncertainty, functional dependencies, most probable database

Digital Object Identifier 10.4230/LIPIcs.ICDT.2023.22

Related Version *Full Version*: `https://arxiv.org/abs/2212.12104`

Funding The work of Amir Gilad was supported by NSF awards IIS-1703431, IIS-1552538, and IIS-2008107. The work of Aviram Imber and Benny Kimelfeld was supported by the Israel Science Foundation (ISF), Grant 768/19, and the German Research Foundation (DFG) Project 412400621 (DIP program).

1 Introduction

Various database tasks amount to reasoning about relations where attribute values are uncertain. To name a few, systems for *data cleaning* may detect errors and suggest alternative fixes with different confidence scores [15, 27, 28], approaches to *data repair* may suggest alternative values due to the violation of integrity constraints (e.g., key constraints and more general functional dependencies) [2, 34], and algorithms for *missing-data imputation* may suggest a probability distribution over possible completions of missing values [3, 24]. Such uncertainty is captured as a probabilistic database in the so called *attribute-level uncertainty* [29] (as opposed to the commonly studied *tuple-level uncertainty* [8]).

We refer to a relation of a probabilistic database in the attribute-level uncertainty as a Cell-Independent Relation (CIR). A CIR is a probabilistic database with a single relation, where the content of a cell is a distribution over possible values, and different cells are probabilistically

tid	room	?specialist	time
1	41	Bart(0.5) | Lisa(0.5)	5 PM
2	163	Bart(0.7) | Lisa(0.3)	5 PM
3	41	Bart(0.2) | Maggie(0.8)	5 PM

(a) CIR U_1 with uncertain specialist.

$F_1 :=$ {?specialist time → room}
$F_2 :=$ {?specialist time → room ,
room time → ?specialist}

(b) Sets F_1 and F_2 of functional dependencies.

tid	room	?specialist	time
1	41	Lisa	5 PM
2	163	Bart	5 PM
3	41	Maggie	5 PM

tid	room	?specialist	time
1	41	Bart	5 PM
2	163	Lisa	5 PM
3	41	Bart	5 PM

(c) Samples r (left) and r' (right) of U_1.

Figure 1 Running example: CIR, FDs, and samples.

independent. The CIR is the correspondent of a relation in the *Tuple-Independent Database* (TID) under the *tuple-level uncertainty*, where the existence of each tuple is uncertain (while its content is certain), and different tuples are probabilistically independent [29]. In contrast, the tuples of a CIR always exist, but their content is uncertain. For illustration, Figure 1a depicts a CIR with uncertain information about specialists attending rooms (e.g., since their attendance is determined by noisy sensors). Some attributes (here room and business) are certain and have deterministic values. The uncertain attributes (e.g., ?specialist) are marked by a question mark and their cells have several options for values. We later explain how this distinction has a crucial impact on the complexity of CIRs.

A natural scenario, studied by previous work for the TID model [12, 21], considers a probabilistic database in the presence of a given set of integrity constraints, and specifically, Functional Dependencies (FDs). Such a scenario gives rise to several interesting computational challenges, and we focus here on three basic ones. In the problem of *possible consistency*, the goal is to test for the existence of a possible world (with a nonzero probability) that satisfies the FDs. The problem of the *most probable database* ("MPD" [12]) is that of finding a possible world that satisfies the FDs and has the highest probability. In the problem of computing the *probability of consistency*, the goal is to calculate the above probability exactly (beyond just deciding whether it is nonzero), that is, the probability that (a random sample of) the given CIR satisfies the underlying FDs. We investigate the computational complexity of these three problems for the CIR model. Our results provide classifications of tractability for different classes of FDs. Importantly, we show that, for the studied classes, the complexity of these problems is determined by two factors: *(1)* the location of the uncertain attributes in the FDs (left or right side), *and (2)* the combination of the FDs in the given set of constraints.

The three problems relate to each other in the following manner. To solve MPD, we need to be able to solve the possible consistency problem. The analysis of the probability of consistency sheds light on the possible consistency problem (is it fundamentally harder to compute the probability than to just determine whether it is nonzero?), but its importance goes beyond that. As we explain in Section 4, computing this probability is useful to any type of constraint over CIRs, as the tractability of this probability implies that we can efficiently sample correctly from the conditional space of consistent samples.

Our study adopts the standard yardstick of *data complexity* [33], where we fix the relational schema and the set of functional dependencies. The schema mentions not only what attributes are in the header of the relation, but also *which attribute is certain and* *which attribute is uncertain*. The complexity of the problems can be different for different combinations of schema and constraints, and we aim for a detailed understanding of which combinations are tractable and which are not.

▶ **Example 1.** Consider again the CIR U_1 in Figure 1a along with the FD set F_1 of Figure 1b, consisting of a single FD. The FD says that, at a specific time, a specialist can be found in only one location. Figure 1c shows a consistent sample r of U_1 whose probability is $\Pr(r) = 0.5 \cdot 0.7 \cdot 0.8 = 0.28$. In particular, this probability is nonzero and, so, U_1 is possibly consistent. This sample has a maximal probability among the consistent samples; therefore, r is a most probable database for U_1. Now, consider the FD set F_2 shown in Figure 1b, where the first FD is the one of F_1 and the second states that no two specialists should be in the same room at the same time. The sample r in Figure 1c is no longer consistent, but r' (in the same figure) is a consistent sample and also a most probable database. In fact, r' is the only consistent sample in this case, so the probability of consistency for F_2 turns out to be that of r'. ⌟

In contrast to the state of affairs for the attribute-level uncertainty, much is known about the complexity of MPD in the case of *tuple-level uncertainty* (i.e., finding the most likely instance of a tuple-independent probabilistic database conditioned on conformance to a set of FDs). As we explain later in the discussion on related work, past research has established a full classification of the complexity of the sets of FDs into tractable and intractable instances of MPD. In this work, we aim to bring our understanding of attribute-level uncertainty closer to tuple-level uncertainty.

Results. We would like to understand the complexity of every scenario defined by a schema and set of FDs, and in particular, establish a dichotomy that charts the exact conditions that cast each problem tractable. This, however, remains open for future investigation. Yet, we make considerable progress towards that. We establish classification results on several classes of functional dependencies:
- Singleton FDs (i.e., FD sets with a single FD);
- Matching constraints (i.e., FD sets of the form $\{X \to Y, Y \to X\}$ for arbitrary X and Y);
- Arbitrary sets of *unary* FDs (i.e., FDs with a single attribute on the left side).

Each classification consists of three internal classifications – one for each of the three problems we study (possible consistency, most probable database, and the probability of consistency). In every case, finding a most probable database turns out to be tractable whenever possible consistency is tractable. There are cases where the probability of consistency is intractable in contrast to the tractability of the most probable database, but we did not find any case where the other direction holds (and we will be surprised if such a case exists). We also establish some general conclusions beyond these classes. For example, in Theorem 14 (of Section 5) we claim that if we make no assumption that some attributes are certain, then possible consistency is hard for *every nontrivial set of FDs.*

▶ **Example 2.** Reconsider the CIR U_1 in Figure 1a along with the FD set F_1 of Figure 1b, consisting of a single FD. Our classification shows that, in general, finding a solution to the possible consistency problem for such an FD, with uncertain attributes on the left side, is NP-complete. Now, reconsider the FD set F_2 shown in Figure 1b, where the first FD is the one of F_1. Thus, $F_1 \subset F_2$, however, interestingly, our results show that for sets with the structure of F_2, finding an MPD (and, hence, also solving possible consistency) is in polynomial time. Intuitively, the additional FD in F_2 constrains the uncertain attribute on the left side of the first FD, making the problem tractable. Finally, computing the probability of consistency for sets with the structure of F_1 and F_2 is #P-hard (or more precisely $\text{FP}^{\#\text{P}}$-complete). ⌟

Related work. In the case of tuple-independent databases, Gribkoff, Van den Broeck, and Suciu [12] established a dichotomy in the complexity of MPD for sets of unary FDs. This dichotomy has been generalized by Livshits, Kimelfeld and Roy [21] to a full classification over all sets of FDs, where they also established that the problem is equivalent to finding a *cardinality repair* of an inconsistent database. Carmeli et al. [5] showed that two tractable cases, namely a *single FD* and a *matching constraint*, remain tractable even if the FDs are treated as *soft constraints* (where every violation incurs a cost).

A most probable database is the same as the "Most Likely Intention" (MLI) in the framework of Probabilistic Unclean Databases (PUD) of De Sa et al. [28], in the special case where the *intention model* demands hard constraints and the *realization model* applies random changes to cells independently in what they refer to as *parfactor/update* PUD. They showed that finding an MLI of a parfactor/update PUD generalizes the problem of finding an *update repair* of an inconsistent database with a minimum number of value changes. In turn, finding a minimal update repair has been studied in the literature and several complexity results are known for special cases of FDs, such as hardness (e.g., for the FD set $\{A \to B, B \to C\}$ due to Kolahi and Lakshmanan [18]) and tractability (e.g., for lhs-chains such as $\{A \to B, AD \to C\}$ due to Livshits et al. [21]). There are, though, substantial differences between finding a most probable consistent sample of a CIR and finding an optimal update repair of an inconsistent database, at least in the variations where complexity results are known. First, they allow to select *any* value (from an infinite domain) for a cell, in contrast to the distributions of the CIR that can limit the space of allowed values; indeed, this plays a major role in past repair algorithms (e.g., Proposition 5.6 of [21] and Algorithm FindVRepair of [18]). Second, they allow to change the value of *any* attribute and do not distinguish between uncertain attributes (where changes are allowed) and certain ones, as we do here; this is critical since, again, without such assumptions the problem is intractable for every nontrivial set of FDs (Theorem 14).

The problem of possible consistency does not have a nontrivial correspondence in the tuple-independent database model since, there, if there is any consistent sample then the subset that consists of all deterministic tuples (i.e., ones with probability one) is such a sample. The probability of consistency might be reminiscent of the problem of *repair counting* that was studied for subset repairs [4, 22]. Besides the fact that subset repairs are about tuple-level uncertainty (and no probabilities are involved), here we do not have any notion of *maximality* (while a repair is required to be a maximal consistent subset).

A CIR can be easily translated into a relation of a *block-independent-disjoint* (BID) probabilistic database [26]. In a BID, every relation is partitioned into independent blocks of mutually exclusive tuples, each associated with a probability. This model has also been studied under the terms *dirty database* [2] and x-tuples [6, 23, 25]. This translation implies that every upper bound for BIDs applies to CIRs, and the contrapositive: every hardness result that we establish (e.g., for the most probable database) extends immediately to BIDs; yet, it does not imply the other direction. Moreover, we are not aware of any positive results on inference over BIDs regarding integrity constraints. In addition, the translation from a CIR to a BID loses the information of which attributes are certain and which are uncertain, and as aforesaid, if we allow every attribute to be uncertain then the problem is hard for every nontrivial set of FDs (Theorem 14).

Organization. The remainder of the paper is organized as follows. We begin with preliminary definitions and notation (Section 2). We then define the CIR data model (Section 3) and the computational problems that we study (Section 4). Next, we describe our analysis for the case of singleton and matching dependencies (Section 5), and then the case of unary functional dependencies (Section 6). Lastly, we give concluding remarks (Section 7). Missing proofs can be found in the full version of the paper [10].

2 Preliminaries

We begin with preliminary definitions and notation.

Relations. We assume countably infinite sets **Val** of values and **Att** of attributes. A *relation schema* is a finite set $R = \{A_1, \dots, A_k\}$ of attributes. An *R-tuple* is a function $t : R \to \mathbf{Val}$ that maps each attribute $A \in R$ to a value that we denote by $t[A]$. A *relation* r is associated with a relation schema, denoted $\mathbf{Att}(r)$, a finite set of tuple identifiers, denoted $\mathsf{tids}(r)$, and a mapping from $\mathsf{tids}(r)$ to $\mathbf{Att}(r)$-tuples. (Note we allow for duplicate tuples, as we do not assume that the tuples of different identifiers are necessarily different.) We say that r is a relation *over* the relation schema $\mathbf{Att}(r)$. We denote by $r[i]$ the tuple that r maps to the identifier i. Hence, $r[i][A]$ is the value that tuple i has for the attribute A. As an example, Figure 1c (left) depicts a relation r with $\mathbf{Att}(r) = \{\mathsf{room}, ?\mathsf{specialist}, \mathsf{time}\}$ (for now, the question mark in ?specialist should be ignored.) Here, $\mathsf{tids}(r) = \{1, 2, 3\}$ and $r[1][\mathsf{room}] = \mathtt{41}$.

Suppose that X is a set of attributes. We denote by $\pi_X r$ the projection of r onto X. More precisely, $\pi_X r$ is the relation r' such that $\mathbf{Att}(r') = X$, $\mathsf{tids}(r') = \mathsf{tids}(r)$, and $r'[i][A] = r[i][A]$ for every $A \in \mathbf{Att}(r) \cap X$. Observe that in our notation, $(\pi_X r)[i]$ is the projection of tuple i to X. As a shorthand notation, we write $r[i][X]$ instead of $(\pi_X r)[i]$. For example, in Figure 1c we have $r[2][\mathsf{room}\ ?\mathsf{specialist}] = (\mathtt{163}, \mathtt{Bart})$.

Functional dependencies. A *functional dependency*, or FD for short, is an expression of the form $X \to Y$ where X and Y are finite sets of attributes. We say that $X \to Y$ is *over* a relation schema R if R contains all mentioned attributes, that is, $X \cup Y \subseteq R$. A relation r satisfies the FD $X \to Y$ over $\mathbf{Att}(r)$ if every two tuples that agree on X also agree on Y. In our notation, we say that r satisfies $X \to Y$ if for every two tuple identifiers i and i' in $\mathsf{tids}(r)$ it holds that $r[i][Y] = r[i'][Y]$ whenever $r[i][X] = r[i'][X]$. A relation r satisfies a set F of FDs over $\mathbf{Att}(r)$, denoted $r \models F$, if r satisfies every FD in F.

We use the standard convention that in instances of X and Y we may remove curly braces and commas. To compactly denote a set of FDs, we may also intuitively combine multiple FD expressions and change the direction of the arrows. For example, the notation $A \leftrightarrow B \leftarrow CD$ is a shorthand notation of $\{A \to B, B \to A, CD \to B\}$.

An FD $X \to Y$ is *unary* if X consists of a single attribute, and it is *trivial* if $Y \subseteq X$ (i.e., it is satisfied by every relation). A *matching constraint* (as termed in past work [5]) is a constraint of the form $X \leftrightarrow Y$, that is, the set $\{X \to Y, Y \to X\}$.

The *closure* F^+ of a set F of FDs is the set of all FDs that are implied by F (or, equivalently, can be inferred by repeatedly applying the axioms of Armstrong). For example, F^+ includes all of the trivial FDs. The closure X_F^+ of a finite set X of attributes is the set of all attributes A such that $X \to A$ is in F^+. Two finite attribute sets X and Y are *equivalent* (w.r.t. F) if $X_F^+ = Y_F^+$, or in other words, $X \to Y$ and $Y \to X$ are both in F^+. By a slight abuse of notation, we say that two attributes A and B are *equivalent* if $\{A\}$ and $\{B\}$ are

tid	business	?spokesperson	?location
1	`S. Propane`	`Mangione`(0.6) | `Strickland`(0.4)	`Arlen`(0.6) | `McMaynerberry`(0.4)
2	`Mega Lo Mart`	`Mangione`(0.45) | `Thatherton`(0.55)	`Arlen`(0.5) | `McMaynerberry`(0.5)
3	`Mega Lo Mart`	`Mangione`(0.4) | `Buckley`(0.6)	`Arlen`(0.55) | `McMaynerberry`(0.45)
4	`Get In Get Out`	`Peggy`(1.0)	`Arlen`(0.35) | `McMaynerberry`(0.3) | `Dallas`(0.35)

Figure 2 CIR U_2 with spokesperson and location as the uncertain attributes.

equivalent. Finally, if F is a set of FDs, then we denote by $\mathbf{Att}(F)$ the set of all attributes that occur in either the left or right sides of rules in F.

Probability distributions. We restrict our study in this paper to finite probability spaces (Ω, π) where Ω is a nonempty finite set of *samples* and $\pi : \Omega \to [0,1]$ is a probability function satisfying $\sum_{o\in\Omega} \pi(o) = 1$. The *support* of $\delta = (\Omega, \pi)$, denoted $\mathsf{supp}(\delta)$, is the set of samples $o \in \Omega$ such that $\pi(o) > 0$. We denote by $\Pr_\delta(o)$ the probability $\pi(o)$. We may write just $\Pr(o)$ when δ is clear from the context.

3 Cell-Independent Relations

A *Cell-Independent Relation*, or *CIR* for short, is similar to an ordinary relation, except that in certain attributes the values may be probabilistic; that is, instead of an ordinary value, each of them contains a probability distribution over values. One could claim that the model should allow every attribute to have uncertain values. However, knowing which attributes are certain has a major impact on the complexity of operations over CIRs. Formally, a CIR U is defined similarly to a relation, with the following differences:

- The schema of U, namely $\mathbf{Att}(U)$, has *marked attributes* where uncertain values are allowed. We denote a marked attribute using a leading question mark, as in $?A$, and the set of marked attributes by $?\mathbf{Att}(U)$. (Note that $?\mathbf{Att}(U)$ is a subset of $\mathbf{Att}(U)$.)
- For every $i \in \mathsf{tids}(U)$ and marked attribute $?A \in ?\mathbf{Att}(U)$, the cell $U[i][?A]$ is a probability distribution over $\mathbf{Val}$.

By interpreting cells as probabilistically independent, a CIR U represents a probability distribution over ordinary relations. Specifically, a sample of U is a relation that is obtained from U by sampling a value for each uncertain cell. More formally, a sample of U is a relation r such that $\mathbf{Att}(r) = \mathbf{Att}(U)$, $\mathsf{tids}(r) = \mathsf{tids}(U)$, and for every $i \in \mathsf{tids}(r)$ and unmarked attribute A we have that $r[i][A] = U[i][A]$.

The probability $\Pr_U(r)$ of a sample r of U is the product of the probabilities of the values chosen for r:

$$\Pr_U(r) = \prod_{i\in\mathsf{tids}(U)} \prod_{?A\in ?\mathbf{Att}(U)} \Pr_{U[i][?A]}(r[i][?A])$$

Note that $\Pr_{U[i][?A]}(r[i][?A])$ is the probability of the value $r[i][?A]$ (i.e., the value that tuple i of r has for the attribute $?A$) according to the distribution $U[i][?A]$ (i.e., the distribution that tuple i of U has for the attribute $?A$).

▶ **Example 3.** Figures 1a and 2 depict examples U_1 and U_2, respectively, of CIRs. U_1 has been discussed in Example 1 and U_2 describes a CIR that stores businesses along with their spokespeople and headquarters locations. Some information in U_2 is noisy (e.g., since the rows are scraped from Web pages), and particularly the identity of the spokesperson and the business location. U_1 has a single uncertain attribute, namely ?specialist, and U_2 has two uncertain attributes, namely ?spokesperson and ?location. In particular, we have:

$$\mathbf{Att}(U_1) = \{\mathsf{room}, ?\mathsf{specialist}, \mathsf{time}\} \qquad ?\mathbf{Att}(U_1) = \{?\mathsf{specialist}\}$$

Distributions over values are written straightforwardly in the examples. For example, the distribution $U_1[2][?\mathsf{specialist}]$ is the uniform distribution that consists of `Bart` and `Lisa`, each with probability 0.5.

The relations r and r' of Figure 1c are samples of U_1. By the choices made in r, the probability $\Pr_U(r)$ is $0.5 \cdot 0.7 \cdot 0.8$. Note that the probability of r is smaller than the probability of the sample where the specialists are `Lisa`, `Bart` and `Maggie`, for instance, respectively. ⌟

Simplified notation. In the analyses that we conduct in later sections, we may simplify the notation when defining a CIR U. When $\mathbf{Att}(U) = \{A_1, \dots, A_k\}$, we may introduce a new tuple $t[i]$ with $t[i][A_\ell] = a_\ell$ simply as $(a_1, \dots, a_k)$, assuming that the attributes are naturally ordered alphabetically by their symbols. For example, if $\mathbf{Att}(U) = \{A, B, C\}$, then (a, b, c) corresponds to the tuple that maps A, B and C to a, b and c, respectively. We can also use a distribution δ instead of a value a_ℓ. In particular, we write $b_1| \dots |b_t$ to denote a uniform distribution among the values $b_1, \dots, b_t$.

▶ **Example 4.** Continuing Example 3, in the simplified notation the tuple $U_1[2]$ can be written as $(\mathtt{163}, \mathtt{Bart}|\mathtt{Lisa}, \mathtt{5\ pm})$ since the attributes are ordered lexicographically and, again, the distribution happens to be uniform. ⌟

Consistency of CIRs

Let F be a set of FDs and let U be a CIR, both over the same schema. A *consistent sample* of U is a relation $r \in \mathsf{supp}(U)$ such that $r \models F$. We say that U is *possibly consistent* if at least one consistent sample exists. By *the probability of consistency*, we refer to the probability $\Pr_{r \sim U}(r \models F)$ that a random sample of U satisfies F. As a shorthand notation, we denote this probability by $\Pr_U(F)$. Note that U is possibly consistent if and only if $\Pr_U(F) > 0$. A consistent sample r is a *most probable database* (using the terminology of Gribkoff, Van den Broeck and Suciu [12]) if $\Pr(r) \geq \Pr(r')$ for every other consistent sample r'.

▶ **Example 5.** Consider the CIR U_1 of Figure 1a. Let F_1 be that of Figure 1b, saying that at a specific time, a specialist can be found in only one location. Figure 1c (left) shows a consistent sample r of U_1. Then $\Pr(r) = 0.5 \cdot 0.7 \cdot 0.8 = 0.28$. In particular, this probability is nonzero, hence U_1 is possibly consistent. The reader can verify that r has a maximal probability among the consistent samples (and, in fact, among all samples); therefore, r is a most probable database for U_1. To calculate the probability of consistency, we will take the complement of the probability of *inconsistency*. An inconsistent sample can be obtained in two ways: *(1)* selecting `Lisa` in both the first and second tuples, *or (2)* selecting `Bart` in the second tuple and in at most one of the first and the third (which we can compute as the complement of the product of the probabilities of selecting the others). Therefore,

$$\Pr_{U_1}(F_1) = 1 - \big(0.3 \cdot 0.5 + 0.7 \cdot (1 - 0.5 \cdot 0.8)\big)\,.$$

Now suppose that we use F_2 of Figure 1b saying that, in addition to F_1, a room can host only one specialist at a specific time. In this case, r is no longer a consistent sample since Room `41` hosts different specialists at `5 PM`, namely `Lisa` and `Maggie`. The reader can verify that the only consistent sample now is r' of Figure 1c. In particular, U_1 remains possible consistent, the sample r' is the most probable database, and the probability of consistency is the probability of r', namely $0.5 \cdot 0.3 \cdot 0.2$. ⌟

4 Consistency Problems

We study three computational problems in the paper, as in the following definition.

▶ **Definition 6.** *Fix a schema R and a set F of FDs over R. In each of the following problems, we are given as input a CIR U over R:*

1. **Possible consistency:** *determine whether* $\Pr_U(F) > 0$*.*
2. **Most probable database:** *find a consistent sample with a maximum probability.*
3. **Probability of consistency:** *calculate* $\Pr_U(F)$*.*

Observe that these problems include the basics of probabilistic inference: *maximum likelihood* computation and *marginal probability* calculation. An MPD can be viewed as an optimal completion of missing values, or an optimal correction of values suspected of being erroneous, assuming the independence of cells (as a *prior* distribution) and conditioned on satisfying the constraints (as a *posterior* distribution). A necessary condition for the tractability of the most probable database is possible consistency, where we decide whether at least one consistent sample exists. The problem of computing the probability of consistency can be thought of as a basic problem that sheds light on possible consistency. For example, if possible consistency is decidable in polynomial time in some case, is it because we can, generally, compute the probability of consistency or because there is something fundamentally easier with feasibility? We will see cases that feature both phenomena.

A more technical reason for computing the probability of consistency is that it provides the ability to *sample* soundly from the conditional probability distribution (the posterior). More precisely, an efficient algorithm for computing the probability of consistency can be used to devise an efficient randomized algorithm that produces a consistent sample r with the probability $\Pr_U(r \mid r \models F)$. The idea is quite simple and applies to every condition F over databases, regardless of being FDs (and was used in different settings, e.g., [7]).

As aforesaid, the second and third problems are at least as hard as the first one: finding a most probable database of U requires knowing whether U is possibly consistent, and calculating the exact probability is at least as hard as determining whether it is nonzero. There is no reason to believe a-priori that the complexities of the second and third problems are comparable. Yet, our analysis will show that the third has the same or higher complexity in the situations that we study.

4.1 Complexity Assumptions

In our complexity analysis, we will restrict the discussion to uncertain cells that are finite distributions represented explicitly by giving a probability for each value in the support. Note that if all uncertain cells of U have a finite distribution, then U has a finite set of samples. Yet, its size can be exponential in the number of rows of U (and also in the number of columns of U, though we will treat this number as fixed as we explain next), even if each cell distribution is binary (i.e., has only two nonzero options). Every probability is assumed to be a rational number that is represented using the numerator and the denominator.

We will focus on the *data complexity* of problems, which means that we will make the assumption that the schema R of the CIR and the set F of FDs are both fixed. Hence, every combination (R, F) defines a separate computational problem, and different pairs (R, F) can potentially have different complexities.

Table 1 Complexity of the consistency problems for a binary schema. "Possibility" refers to possible consistency, "MPD" refers to the most probable database problem, and "Probability" refers to the probability of consistency.

FDs	Possibility	MPD	Probability	Results
$A \to ?B$	PTime	PTime	PTime	Proposition 7
$?A \to B$	NP-complete	NP-hard	FP$^{\#P}$-complete	Lemma 10
$A \leftrightarrow ?B$	PTime	PTime	FP$^{\#P}$-complete	Prop. 8 (PTime), 9 (FP$^{\#P}$-c.)
$?A \leftrightarrow ?B$	NP-complete	NP-hard	FP$^{\#P}$-complete	Lemma 11

4.2 Preliminary Observations

In the following sections, we study the complexity of the three consistency problems that we defined in Definition 6. Before we move on to the actual results, let us state some obvious general observations.

- Possible consistency is in NP, since we can verify a "yes" instance U in polynomial time by verifying that a relation r is a consistent sample.
- If possible consistency is NP-complete for some schema R and set F of FDs, then it is NP-hard to find a most probable database, and it is NP-hard to compute the probability of consistency.
- We will show that the probability of consistency can be #P-hard, or more precisely FP$^{\#P}$-complete.[1] Membership in FP$^{\#P}$ of the probability of consistency is based on our assumption that probabilities are represented as rational numbers, and it can be shown using standard techniques (e.g., [1, 11]) that we do not repeat here.

We will take the above for granted and avoid repeating the statements throughout the paper.

5 Singleton and Matching Constraints

In this section, we investigate the complexity of the three problems we study in two special cases: a singleton constraint $\{X \to Y\}$ and a *matching constraint* $X \leftrightarrow Y$ (as it has been termed in past work [5]). We give full classifications of when such constraints are tractable and intractable for the three problems. We note that we leave open the classification of the entire class of FD sets, but we provide it for the general case of unary FDs in Section 6.

We begin with the case of a binary schema, where every set of FDs is equivalent to either a singleton or a matching constraint.

5.1 The Case of a Binary Schema

Throughout this section, we assume that the schema is $\{A, B\}$. The complexity of the different cases of FDs is shown in Table 1. To explain the entries of the table, let us begin with the tractable cases.

[1] Recall that FP$^{\#P}$ is the class of functions that are computable in polynomial time with an oracle to a problem in #P (e.g., counting the number of satisfying assignments of a propositional formula). This class is considered intractable, and above the polynomial hierarchy [30].

5.1.1 Algorithms

In this section, we show algorithms for $A \to ?B$ and for $A \leftrightarrow ?B$.

For $A \to ?B$, we need to determine a value b for each value a of the attribute A. The idea is that we do so independently for each a. Let V_A be the active domain of the attribute A of U, and V_B be the set of all values in the supports of the distributions of B. Formally:

$$V_A := \{U[i][A] \mid i \in \mathsf{tids}(U)\} \qquad V_B := \bigcup \{\mathsf{supp}(U[i][B]) \mid i \in \mathsf{tids}(U)\}$$

A consistent sample r selects a value $b_a \in V_B$ for each $a \in V_A$, and then $\Pr_U(r) = \prod_{a \in V_A} p(a, b_a)$ where $p(a,b)$ is given by:

$$p(a,b) := \prod_{i:U[i][A]=a} \Pr_{U[i][B]}(b)$$

Therefore, to find a most probable database, we consider each $a \in V_A$ independently, and find a $b \in V_B$ that maximizes $p(a,b)$. This b will be used for the tuples with the value a in A. In addition, we have the following formula that gives us immediately a polynomial-time algorithm (via a direct computation) for the probability of consistency:

$$\Pr_U(\{A \to B\}) = \prod_{a \in V_A} \sum_{b \in V_B} p(a,b)$$

Where $\sum_{b \in V_B} p(a,b)$ is the probability that the tuples with the value a for A agree on their B attribute. In summary, we have established the following.

▶ **Proposition 7.** *All three problems in Definition 6 are solvable in polynomial time for $A \to ?B$.*

Next, we discuss $A \leftrightarrow ?B$. Let U be a CIR. A consistent sample r of U entails the matching of each A value a to each B value b, so that no two a values occur with the same b, and no two bs occur with the same a. Therefore, we can solve this problem using an algorithm for minimum-cost perfect matching, as follows. Let V_A, V_B and $p(a,b)$ be as defined earlier in this section for $A \to ?B$. We construct a complete bipartite graph G as follows.

- The left-side vertex set is V_A and the right-side vertex set is V_B.
- The cost of every edge (a,b) is $(-\log p(a,b))$; we use this weight as as our goal is to translate a maximum product into a minimum sum.[2]

Note that $|V_A|$ and $|V_B|$ are not necessarily of the same cardinally. If $|V_A| > |V_B|$, then U has no consistent sample at all. If $|V_A| < |V_B|$, then we add to the left side of the graph dummy vertices a' that are connected to all V_B vertices using the same cost, say 1. With this adjustment, we can now find a most probable database by finding a minimum-cost perfect matching in G (e.g., with the Hungarian method [19]). In summary, we have established the following.

▶ **Proposition 8.** *For $A \leftrightarrow ?B$, a most probable database can be found in polynomial time.*

It turns out that the third problem, the probability of consistency, is intractable. We show it in the next section.

[2] We assume that the computational model for finding a minimum-cost perfect matching can handle the representation of logarithms, including $\log 0 = -\infty$. As an alternative, we could use directly an algorithm for maximizing the product of the edges in the perfect matching [31].

5.1.2 Hardness

We now discuss the hardness results of Table 1. We begin with $A \leftrightarrow ?B$. Recall that possible consistency and the most probable database are solvable in polynomial time (Proposition 8). The probability of consistency, however, is hard.

▶ **Proposition 9.** *For $A \leftrightarrow ?B$, it is* $\mathrm{FP}^{\#\mathrm{P}}$*-complete to compute the probability of consistency.*

Proof. We show a reduction from the problem of counting the perfect matchings of a bipartite graph (which is the same as calculating the permanent of a 0/1-matrix). This problem is known to be #P-complete [32]. We are given a bipartite graph $G = (V_L, V_R, E)$ such that $|V_L| = |V_R|$ and the goal is to compute the number of perfect matchings that G has. We construct a CIR U as follows. For each vertex $v \in V_L$ we collect the set $N_v \subseteq V_R$ of neighbors of v. Let $N_v = \{u_1, \dots, u_\ell\}$. We add to U the tuple $(v, u_1|\dots|u_\ell)$.

Observe that every consistent sample induces a perfect matching (due to $A \leftrightarrow ?B$), and vice versa. Hence, the number of consistent samples of U is the same as the number of perfect matchings of G. Since we used only uniform probabilities, every sample of U has the same probability, namely $1/(\prod_{v\in V_L} |N_v|)$. Therefore, the number of perfect matchings is $\Pr_U(A \leftrightarrow ?B) \cdot \prod_{v\in V_L} |N_v|$. ◀

The next two lemmas address the case of $?A \to B$ and the case of $?A \leftrightarrow ?B$, respectively. We begin with $?A \to B$.

▶ **Lemma 10.** *For $?A \to B$:*
1. *Possible consistency is NP-complete.*
2. *It is* $\mathrm{FP}^{\#\mathrm{P}}$*-complete to compute the probability of consistency.*

Proof. We prove each part separately.

Part 1. We show a reduction from *non-mixed satisfiability* (NM-SAT), where each clause contains either only positive literals ("positive clause") or only negative literals ("negative clause"). This problem is known to be NP-complete [13].

We are given a formula $c_1 \wedge \dots \wedge c_m$ over $x_1, \dots, x_n$. We construct an uncertain table as follows. For each positive $c_i = y_1 \vee \dots \vee y_\ell$ we have in the table the tuple

$$(y_1|\dots|y_\ell, \mathbf{true}),$$

that is, a tuple with a distinct identifier i such that $U[i][A]$ is a uniform distribution over $\{y_1, \dots, y_\ell\}$ and $U[i][B]$ is the value **true**. Similarly, for each negative clause $c_i = \neg y_1 \vee \dots \vee \neg y_\ell$ we have in the table the tuple

$$(y_1|\dots|y_\ell, \mathbf{false}).$$

Hence, for each positive clause we need to select one satisfying variable, for each negative clause we need to select one satisfying variable, and we cannot select the same variable to satisfy both a positive and a negative clause. This immediately implies the correctness of the reduction.

Part 2. To prove Part 2, we use a reduction from counting the perfect matchings, similarly to the proof of Proposition 9, except that now we reverse the order of the attributes: Instead of adding the tuple $(v, u_1|\dots|u_\ell)$, we add the tuple $(u_1|\dots|u_\ell, v)$. The reader can easily verify that each consistent sample again encodes a unique perfect matching, and vice versa. ◀

We now move on to $?A \leftrightarrow ?B$.

▶ **Lemma 11.** *For $?A \leftrightarrow ?B$:*
1. *Possible consistency is NP-hard.*
2. *It is $\mathrm{FP}^{\#\mathrm{P}}$-complete to compute the probability of consistency.*

Proof. We prove each part separately.

Part 1. We need to show the NP-hardness of possible consistency. We show a reduction from standard SAT, where we are given a formula $\varphi = c_1 \wedge \cdots \wedge c_m$ over $x_1, \dots, x_n$, and we construct a CIR U over $\{A, B\}$ as follows. For each clause $c = d_1 \vee \cdots \vee d_\ell$ we add to U the tuple

$$(c, \langle c, d_1\rangle | \dots | \langle c, d_\ell\rangle)\,.$$

Note that the values of U are clauses c and pairs $\langle c, d\rangle$ where d is a literal. In addition to these tuples, we collect every two pairs $\langle c, d\rangle$ and $\langle c', d'\rangle$ such that d and d' are in conflict, that is, if $d = x$ then $d' = \neg x$ and if $d = \neg x$ then $d' = x$. For each such pair, we add to U the tuple

$$(\langle c, d\rangle | \langle c', d'\rangle, \langle c, d\rangle | \langle c', d'\rangle)\,.$$

This completes the reduction. Next, we prove the correctness of the reduction, that is, φ is satisfiable if and only if U is possibly consistent.

For the "only if" direction, suppose that τ is a satisfying truth assignment for φ. We construct a consistent sample r as follows. For every tuples of the form $(c, \langle c, d_1\rangle | \dots | \langle c, d_\ell\rangle)$, we choose for B a value $\langle c, d_i\rangle$ such that $\tau(d_i) = \textbf{true}$. In the case of tuples of the form $(\langle c, d\rangle | \langle c', d'\rangle, \langle c, d\rangle | \langle c', d'\rangle)$, we choose the pair $\langle c', d'\rangle$ such that $\tau(d') = \textbf{false}$ for both attributes A and B. We need to show that r satisfies $?A \leftrightarrow ?B$. It is easy to see why the left attribute determines the right attribute, and so, $?A \to ?B$ holds. Regarding $?B \to ?A$, we need to verify that we do not have any conflicting tuples $(c, \langle c, d\rangle)$ and $(\langle c', d'\rangle, \langle c', d'\rangle)$ where $c = c'$ and $d = d'$. This is due to the fact that $\tau(d) = \textbf{true}$ and $\tau(d') = \textbf{false}$.

For the "if" direction, suppose that r is a consistent sample. We define a satisfying truth assignment τ as follows. Suppose that r contains $(c, \langle c, d\rangle)$. Then r necessarily contains $(\langle c', d'\rangle, \langle c', d'\rangle)$ for every c' that contains the negation d' of d. Therefore, r does not contain any $(c', \langle c', d'\rangle)$ where d' contradicts d. So, we choose τ such that $\tau(d) = \textbf{true}$. If needed, we complete τ to the remaining variables arbitrarily. From the construction of τ it holds that every clause c is satisfied. This completes the proof of Part 1.

Part 2. Note that this part follows immediately from Proposition 9, since every instance of $A \leftrightarrow ?B$ can be viewed as an instance of $?A \leftrightarrow ?B$ where all A values are known. ◀

We have now completed all results of Table 1. We will use these results for the extension to singleton, matching, and unary constraints.

5.2 Beyond Binary Schemas

We generalize the results for the binary case to the more general case where the FD set is either a singleton or a matching constraint and the schema can have more than two attributes.

▶ **Theorem 12.** *Let X and Y be sets of attributes such that $X \not\subseteq Y$ and $Y \not\subseteq X$, and at least one attribute in $X \cup Y$ is uncertain.*

1. *In the case of $X \to Y$: If X consists of only certain attributes, then all three problems are solvable in polynomial time; otherwise, possible consistency is NP-complete and the probability of consistency is* $\mathrm{FP}^{\#\mathrm{P}}$*-complete.*
2. *In the case of $X \leftrightarrow Y$: If either X or Y consists of only certain attributes, then a most probable database can be found in polynomial time; otherwise, possible consistency is NP-hard. In any case, the probability of consistency is* $\mathrm{FP}^{\#\mathrm{P}}$*-complete.*

Proof sketch. For the first part, the tractability side is via a reduction to the case of $A \to ?B$, which is tractable due to Proposition 7. The hardness side is due to a straightforward reduction from $?A \to B$, where hardness is stated in Lemma 10. For the second part, the tractability side is via a reduction to the case of $A \leftrightarrow ?B$, which is tractable due to Proposition 8. The hardness of possible consistency relies on the cases of $?A \to B$ and $?A \leftrightarrow ?B$ from Lemma 10 and Lemma 11, respectively. ◀

▶ **Example 13.** Consider again the CIR U_1 of Figure 1a, and the following two constraints: $F_1 := \{$?specialist time $\to$ room$\}$ and $F_2 := F_1 \cup \{$room time $\to$?specialist$\}$. For F_1, all three problems are hard, since the left hand side of the FD contains the uncertain attribute ?specialist. For F_2, a most probable database can be found in polynomial time, since F_2 is equivalent to room time $\leftrightarrow$?specialist time, where one side (the left side) consists of only certain attributes. However, the probability of consistency remains $\mathrm{FP}^{\#\mathrm{P}}$-hard. ⌟

Note that in Theorem 12, the assumption that $X \not\subseteq Y$ and $Y \not\subseteq X$ does not lose generality, for the following reason. If $X \subseteq Y$, then the FD $X \to Y$ is equivalent to $X \to Y \setminus X$, the FD $Y \to X$ is trivial, and the matching constraint $X \leftrightarrow Y$ is equivalent to the singleton $\{X \to Y\}$ (which is covered in Part 1).

From Theorem 12 we can conclude that when all attributes are uncertain, possible consistency is hard, unless the FDs are all trivial (and then all three problems are clearly solvable in polynomial time); this is under the reasonable (and necessary) assumption that F has no *consensus FDs*, that is, the left hand side of every FD is nonempty [21]. We later discuss this assumption. This emphasizes the importance of having a data model that distinguishes between certain and uncertain attributes.

▶ **Theorem 14.** *Let F be a nontrivial set of FDs over a relation schema R where all attributes are uncertain, none being a consensus FD. Then possible consistency is NP-complete.*

The proof selects between a reduction from MPD with the FD $?A \to B$ (Lemma 10) and a reduction from MPD with the matching constraint $?A \leftrightarrow ?B$ (Lemma 11), depending on the structure of F.

We note that the assumption that F has no consensus FDs is necessary. For example, for $F = \{\emptyset \to ?A\}$, which is nontrivial, we can find a most probable database by considering every possible value a for $?A$, computing the probability of selecting a in all distributions, and finally using the value with the maximal probability.

From Theorem 14 we immediately conclude the hardness of the three problems on *every* nontrivial set of FDs in the block-independent-disjoint (BID) model of probabilistic databases [26], due to the translation mentioned in the Introduction.

6 General Sets of Unary Functional Dependencies

In Section 5.1, we studied the complexity of the three problems in the case of a binary schema, and we gave a full classification of the different possible sets of FDs. In this section, we extend these results to a general classification (dichotomy) for every set of *unary* FDs, that is, FDs with a single attribute on the left side. Our result uses a decomposition technique that we devise next.

6.1 Reduction by Decomposition

In this section, we devise a decomposition technique that allows us to reduce our computational problems from one set of FDs into multiple smaller subsets of the set. This technique is stated in the next theorem. After the theorem, we show several consequences that illustrate the use of the technique. Later, we will use these consequences to establish a full classification of complexity for the sets of unary FDs.

▶ **Theorem 15.** *Let F be a set of FDs over a relation schema R. Suppose that $F = F_1 \cup F_2$ and that all attributes in $\mathbf{Att}(F_1) \cap \mathbf{Att}(F_2)$ are certain (unmarked). Each of the three problems (in Definition 6) can be solved in polynomial time if its version with F_j and $\mathbf{Att}(F_j)$ is solvable in polynomial time for both $j = 1$ and $j = 2$.*

Proof sketch. Let $U_j = \pi_{\mathbf{Att}(F_j)}U$ for $j = 1, 2$. We show the following:

1. U is possibly consistent w.r.t. F if and only if U_1 and U_2 are possibly consistent w.r.t. F_1 and F_2, respectively.
2. MPDs of U_1 and U_2 can be easily combined to produce an MPD of U.
3. $\Pr_U(F) = \Pr_{U_1}(F_1) \cdot \Pr_{U_2}(F_2)$.

The full details are provided in [10]. ◀

An immediate conclusion from Theorem 15 is that we can eliminate the FDs that involve only certain attributes if we know how to deal with the remaining FDs.

▶ **Corollary 16.** *Let F be a set of FDs over a relation schema R. Let $X \to Y$ be an FD in F, and suppose that all attributes in X and Y are certain. Then each of the three problems (in Definition 6) is polynomial-time reducible to its version with R and $F \setminus \{X \to Y\}$.*

▶ Remark 17. Eliminating the FDs over the certain attributes is not always beneficial, since these FDs might be needed for applying a polynomial-time algorithm. As an example, consider the following set of FDs: $\{?A \to B\,,\, B \to C\,,\, C \to ?A\}$. As we will show later, for this set of FDs we can find a most probable database in polynomial time. However, we will also show that possible consistency is NP-hard for the subset $\{?A \to B\,,\, C \to ?A\}$. Hence, $B \to C$ is needed for the polynomial-time algorithm. ⌟

The following consequence of Theorem 15 identifies a general tractable case: the problems are solvable in polynomial time if uncertain attributes do not appear in the left side of the FDs (but they can appear in the right side or outside of the FDs).

▶ **Theorem 18.** *Let F be a set of FDs. If the left side of every FD includes only certain attributes, then each of the three problems (in Definition 6) is solvable in polynomial time.*

Proof sketch. Assume, without loss of generality, that each FD in F contains a single attribute on the right side. For every $A \in \mathbf{Att}(F)$, let F_A be the subset of F that contains all FDs with A being the right side (i.e., all FDs of the form $X \to A$). Then $F = \cup_{A \in \mathbf{Att}(F)} F_A$.

Note that sets F_A and F_B, where $A \neq B$, share only certain attributes. This is true since our assumption implies that an uncertain attribute $?A$ can appear only in $F_{?A}$. Hence, we can apply Theorem 15 repeatedly and conclude that we need a polynomial-time solution for each $F_{?A}$. In the full version [10], we show that we can obtain that using a similar concept to the algorithm for $A \to ?B$ from Section 5.1.1. ◀

6.2 Classification

We now state the precise classification of the complexity of the problems in the case of unary FDs. The statement uses the following terminology. Let F be a set of unary FDs. Recall that two attributes A and B and are *equivalent* if they have the same closure, that is, $\{A\}^+_F = \{B\}^+_F$. An attribute A is called a *sink* if $\{A\}^+_F = \{A\}$, that is, A does not appear in the left hand side of any nontrivial FD. In this section, we will prove the following classification (trichotomy) result, which is also illustrated in Figure 3.

▶ **Theorem 19.** *Let F be a set of unary FDs over a relation schema R. Then following hold.*
1. *If every uncertain attribute is either a sink or equivalent to a certain attribute, then a most probable database can be found in polynomial time; otherwise, possible consistency is NP-complete.*
2. *If every uncertain attribute is a sink, then the probability of consistency can be calculated in polynomial time; otherwise, it is* $\mathrm{FP}^{\#\mathrm{P}}$*-complete.*

The following examples illustrate the instantiation of the theorem to specific scenarios.

▶ **Example 20.** We give several examples for the case of a ternary schema $\{A, B, C\}$. Consider the following sets of FDs:

$$F_1 := \{A \to B \to ?C\} \qquad F_2 := \{A \to ?B \to C\} \qquad F_3 := \{A \leftrightarrow ?B \to C\}$$

Theorem 19 tells us the following. All three problems are solvable in polynomial time in the case of F_1, since the uncertain attribute $?C$ is a sink. In the case of F_2, we can see that $?B$ is neither a sink nor equivalent to any certain attribute; hence, all three problems are intractable for F_2. In the case of F_3, the attribute $?B$ is not a sink but is equivalent to the certain attribute A. Hence, the probability of consistency for F_3 is $\mathrm{FP}^{\#\mathrm{P}}$-complete, but we can find a most probable database in polynomial time. ⌟

Next, we illustrate Theorem 19 on our running example.

▶ **Example 21.** Consider again the CIR U_2 of Figure 2. Consider the following constraints.
1. business $\to$?spokesperson?location
2. ?spokesperson $\to$?location
3. business $\leftrightarrow$?spokesperson $\to$?location

For the first constraint, all three problems are tractable since both ?spokesperson and ?location are sinks. For the second constraint, all three problems are intractable since ?spokesperson is neither a sink nor equivalent to any certain attribute. For the third constraint, a most probable database can be found in polynomial time since ?spokesperson is equivalent to the certain business and ?location is a sink, but the probability of consistency is $\mathrm{FP}^{\#\mathrm{P}}$-complete since ?spokesperson is not a sink. ⌟

In the remainder of this section, we prove each of the two parts of Theorem 19 separately.

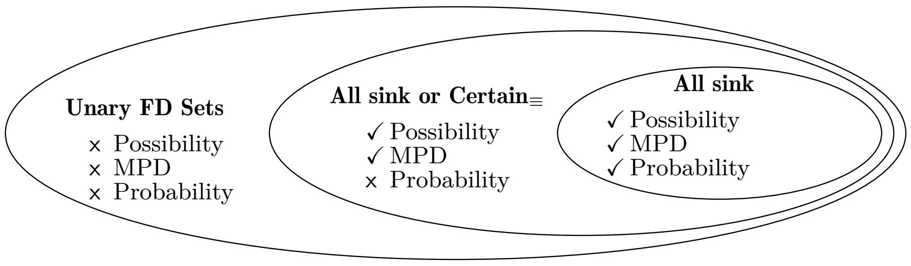


Figure 3 Classification of the complexity of consistency problems for sets of unary FDs. (See Table 1 for the naming of the problems.) "All sink" refers to the case where every uncertain attribute is sink, and "All sink or Certain$_\equiv$" refers to the case where every uncertain attribute is either a sink or equivalent to a certain attribute.

6.2.1 Part 1 of Theorem 19 (Possible Consistency and MPD)

We first prove the tractability side of Part 1 of the theorem.

▶ **Lemma 22.** *Let F be a set of unary FDs over a schema R. If every uncertain attribute is either a sink or equivalent to a certain attribute, then a most probable database can be found in polynomial time.*

Proof sketch. The idea is to define a set $F_{?A}$ of FDs for every uncertain attribute $?A \in ?\mathbf{Att}(U)$, and a set F' of FDs where all left-side attributes are certain, such that F is equivalent to $F' \cup \bigcup_{?A \in ?\mathbf{Att}(U)} F_{?A}$. Then, we repeatedly apply Theorem 15 to reduce the original problem to instances that are solvable in polynomial time by Proposition 8 and Theorem 18. ◀

For the hardness side of Part 1 of Theorem 19, we will need the following lemma, which generalizes the case of $?A \leftrightarrow ?B$ from Lemma 11.

▶ **Lemma 23.** *Let $R = \{?A_1, \dots, ?A_k\}$ consist of $k > 1$ uncertain attributes, and suppose that F is a set of FDs stating that all attributes in R are equivalent. Then possible consistency is NP-complete.*

The next lemma states the hardness side of Part 1 of Theorem 19.

▶ **Lemma 24.** *Let F be a set of unary FDs over a schema R. If there is an uncertain attribute that is neither a sink nor equivalent to a certain attribute, then possible consistency is NP-complete.*

Proof sketch. Let $?A$ be an attribute that is neither a sink nor equivalent to a certain attribute. Let X be the closure of $?A$ and X' be $X \setminus \{?A\}$. Observe the following. First, X' must be nonempty since $?A$ is not a sink. Second, if any attribute in X' implies $?A$ then it is equivalent to $?A$, and then it is necessarily uncertain. We consider two cases:

1. No attribute in X' implies $?A$.
2. Some attribute in X' implies $?A$.

For the first case, we show a reduction from $?A \to B$, where possible consistency is NP-complete due to Lemma 10. For the second case, let $?B_1, \dots, ?B_\ell$ be the set of all attributes in X' that imply $?A$. As said above, each $?B_j$ must be uncertain. Then all of $?B_1, \dots, ?B_\ell, ?A$ are equivalent. We show a reduction from the problem of Lemma 23 where $k = \ell + 1$. ◀

6.2.2 Part 2 of Theorem 19 (Probability of Consistency)

We now move on to Part 2. The tractability side follows immediately from Theorem 18, since if all uncertain attributes are sinks, then all left-side attributes are certain (up to trivial FDs $?A \rightarrow ?A$ that can be ignored). Hence, it remains to prove the hardness side of Part 2 of Theorem 19. We start with the following lemma, where we use a reduction from the case of $A \leftrightarrow ?B$, where probability of consistency is $\mathrm{FP}^{\#\mathrm{P}}$-complete by Proposition 9, to establish hardness for a more general case.

▶ **Lemma 25.** *Let F be a set of unary FDs over a schema R. If at least one uncertain attribute is equivalent to a certain attribute, then the probability of consistency is $\mathrm{FP}^{\#\mathrm{P}}$-complete.*

We can now complete the proof of the hardness side of Part 2.

▶ **Lemma 26.** *Let F be a set of unary FDs over a schema R. If there is at least one uncertain attribute that is not a sink, then the probability of consistency is $\mathrm{FP}^{\#\mathrm{P}}$-complete.*

Proof sketch. Let $?A$ be an uncertain attribute that is not a sink. Let $Y = (?A)^+_F \setminus \{?A\}$. Note that Y is nonempty, since $?A$ is not a sink. If any attribute B in Y functionally determines $?A$, then we can use this attribute as a certain attribute (even if it is uncertain) and use Lemma 25, since $?A$ is equivalent to B. Otherwise, suppose that no attribute in Y determines $?A$. For this case, we show (in [10]) a reduction from $?A \rightarrow B$, where the probability of consistency is $\mathrm{FP}^{\#\mathrm{P}}$-hard according to Lemma 10. ◀

6.2.3 Recap

We can now complete the proof of Theorem 19. For Part 1, the tractability side is given by Lemma 22, and the hardness is given by Lemma 24. As for Part 2, the tractability side follows immediately from Theorem 18, and the hardness side is stated in Lemma 26.

7 Conclusions

We defined the concept of a CIR and studied the complexity of three problems that relate to consistency under FDs: possible consistency, finding a most probable database, and the probability of consistency. A seemingly minor feature of the definition of a CIR is the distinction between certain and uncertain attributes; yet, this distinction turns out to be crucial for detecting tractable cases. We gave classification results for several classes of FD sets, including a single FD, a matching constraint, and arbitrary sets of unary FDs. We also showed that if all attributes are allowed to be uncertain, then the first two problems are intractable for every nontrivial set of FDs.

This work leaves many problems for future investigation. Within the model, we have not yet completed the classification for the whole class of FD sets, where the problem remains open. Recall that a full classification is known for the most probable database for tuple-independent databases [21]. Moreover, as we hit hardness already for simple cases (e.g., $?A \rightarrow B$), it is important to identify realistic properties of the CIR that reduce the complexity of the problems and allow for efficient algorithms.

Going beyond the framework of this paper, we plan to study additional types of constraints that are relevant to data cleaning [9], such as conditional FDs, denial constraints, and foreign-key constraints (where significant progress has been recently made in the problem of consistent query answering [14]). Another useful direction is to consider *soft* or *approximate* versions of the constraints, where it suffices to be consistent to some quantitative extent [5,16,20].

Finally, we have made the assumption of probabilistic independence among the cells as this is the most basic setting to initiate this research. To capture realistic correlations in the database noise, it is important to extend this work to data models that allow for (learnable) probabilistic dependencies, such as Markov Logic [17].

References

1 Serge Abiteboul, T.-H. Hubert Chan, Evgeny Kharlamov, Werner Nutt, and Pierre Senellart. Aggregate queries for discrete and continuous probabilistic XML. In *ICDT*, ACM International Conference Proceeding Series, pages 50–61. ACM, 2010.

2 Periklis Andritsos, Ariel Fuxman, and Renée J. Miller. Clean answers over dirty databases: A probabilistic approach. In *ICDE*, page 30, 2006.

3 Felix Bießmann, Tammo Rukat, Philipp Schmidt, Prathik Naidu, Sebastian Schelter, Andrey Taptunov, Dustin Lange, and David Salinas. Datawig: Missing value imputation for tables. *J. Mach. Learn. Res.*, 20:175:1–175:6, 2019.

4 Marco Calautti, Ester Livshits, Andreas Pieris, and Markus Schneider. Counting database repairs entailing a query: The case of functional dependencies. In *PODS*, pages 403–412. ACM, 2022.

5 Nofar Carmeli, Martin Grohe, Benny Kimelfeld, Ester Livshits, and Muhammad Tibi. Database repairing with soft functional dependencies. In *ICDT*, volume 186 of *LIPIcs*, pages 16:1–16:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

6 Reynold Cheng, Jinchuan Chen, and Xike Xie. Cleaning uncertain data with quality guarantees. *Proc. VLDB Endow.*, 1(1):722–735, 2008.

7 Sara Cohen, Benny Kimelfeld, and Yehoshua Sagiv. Incorporating constraints in probabilistic XML. *ACM Trans. Database Syst.*, 34(3):18:1–18:45, 2009.

8 Nilesh N. Dalvi and Dan Suciu. Efficient query evaluation on probabilistic databases. In *VLDB*, pages 864–875. Morgan Kaufmann, 2004.

9 Wenfei Fan and Floris Geerts. *Foundations of Data Quality Management.* Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2012.

10 Amir Gilad, Aviram Imber, and Benny Kimelfeld. The consistency of probabilistic databases with independent cells, 2022. `doi:10.48550/arXiv.2212.12104`.

11 Erich Grädel, Yuri Gurevich, and Colin Hirsch. The complexity of query reliability. In *PODS*, pages 227–234. ACM Press, 1998.

12 Eric Gribkoff, Guy Van den Broeck, and Dan Suciu. The most probable database problem. In *BUDA*, 2014. URL: `http://www.sigmod2014.org/buda`.

13 Venkatesan Guruswami. Inapproximability results for set splitting and satisfiability problems with no mixed clauses. In *APPROX*, volume 1913 of *LNCS*, pages 155–166. Springer, 2000.

14 Miika Hannula and Jef Wijsen. A dichotomy in consistent query answering for primary keys and unary foreign keys. In *PODS*, pages 437–449. ACM, 2022.

15 Alireza Heidari, Joshua McGrath, Ihab F. Ilyas, and Theodoros Rekatsinas. HoloDetect: Few-shot learning for error detection. In *SIGMOD Conference*, pages 829–846. ACM, 2019.

16 Abhay Kumar Jha, Vibhor Rastogi, and Dan Suciu. Query evaluation with soft-key constraints. In *PODS*, pages 119–128. ACM, 2008.

17 Abhay Kumar Jha and Dan Suciu. Probabilistic databases with markoviews. *Proc. VLDB Endow.*, 5(11):1160–1171, 2012.

18 Solmaz Kolahi and Laks V. S. Lakshmanan. On approximating optimum repairs for functional dependency violations. In *ICDT*, volume 361 of *ACM*, pages 53–62. ACM, 2009.

19 Harold W Kuhn. The hungarian method for the assignment problem. *Naval research logistics quarterly*, 2(1-2):83–97, 1955.

20 Ester Livshits, Alireza Heidari, Ihab F. Ilyas, and Benny Kimelfeld. Approximate denial constraints. *Proc. VLDB Endow.*, 13(10):1682–1695, 2020.

21 Ester Livshits, Benny Kimelfeld, and Sudeepa Roy. Computing optimal repairs for functional dependencies. *ACM Trans. Database Syst.*, 45(1):4:1–4:46, 2020. `doi:10.1145/3360904`.

22 Ester Livshits, Benny Kimelfeld, and Jef Wijsen. Counting subset repairs with functional dependencies. *J. Comput. Syst. Sci.*, 117:154–164, 2021.

23 Dany Maslowski and Jef Wijsen. A dichotomy in the complexity of counting database repairs. *J. Comput. Syst. Sci.*, 79(6):958–983, 2013.

24 Chris Mayfield, Jennifer Neville, and Sunil Prabhakar. ERACER: a database approach for statistical inference and data cleaning. In *SIGMOD Conference*, pages 75–86. ACM, 2010.

25 Luyi Mo, Reynold Cheng, Xiang Li, David W. Cheung, and Xuan S. Yang. Cleaning uncertain data for top-k queries. In *ICDE*, pages 134–145, 2013.

26 Christopher Ré and Dan Suciu. Materialized views in probabilistic databases for information exchange and query optimization. In *Proc. VLDB Endow.*, pages 51–62, 2007.

27 Theodoros Rekatsinas, Xu Chu, Ihab F. Ilyas, and Christopher Ré. HoloClean: Holistic data repairs with probabilistic inference. *PVLDB*, 10(11):1190–1201, 2017.

28 Christopher De Sa, Ihab F. Ilyas, Benny Kimelfeld, Christopher Ré, and Theodoros Rekatsinas. A formal framework for probabilistic unclean databases. In *ICDT*, volume 127 of *LIPIcs*, pages 6:1–6:18, 2019.

29 Dan Suciu, Dan Olteanu, Christopher Ré, and Christoph Koch. *Probabilistic Databases*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2011.

30 Seinosuke Toda. PP is as hard as the polynomial-time hierarchy. *SIAM J. Comput.*, 20(5):865–877, 1991.

31 Frank S.C. Tseng, Wei-Pang Yang, and Arbee L.P. Chen. Finding a complete matching with the maximum product on weighted bipartite graphs. *Computers & Mathematics with Applications*, 25(5):65–71, 1993.

32 Leslie G. Valiant. The complexity of computing the permanent. *Theor. Comput. Sci.*, 8:189–201, 1979.

33 Moshe Y. Vardi. The complexity of relational query languages (extended abstract). In *STOC*, pages 137–146. ACM, 1982.

34 Jef Wijsen. Database repairing using updates. *ACM Trans. Database Syst.*, 30(3):722–768, 2005.

Consistent Query Answering for Primary Keys and Conjunctive Queries with Counting

Aziz Amezian El Khalfioui
Research fellow FNRS
University of Mons, Mons, Belgium

Jef Wijsen
University of Mons, Mons, Belgium

Abstract

The problem of consistent query answering for primary keys and self-join-free conjunctive queries has been intensively studied in recent years and is by now well understood. In this paper, we study an extension of this problem with counting. The queries we consider count how many times each value occurs in a designated (possibly composite) column of an answer to a full conjunctive query. In a setting of database repairs, we adopt the semantics of [Arenas et al., ICDT 2001] which computes tight lower and upper bounds on these counts, where the bounds are taken over all repairs. Ariel Fuxman defined in his PhD thesis a syntactic class of queries, called Cforest, for which this computation can be done by executing two first-order queries (one for lower bounds, and one for upper bounds) followed by simple counting steps. We use the term "parsimonious counting" for this computation. A natural question is whether Cforest contains all self-join-free conjunctive queries that admit parsimonious counting. We answer this question negatively. We define a new syntactic class of queries, called Cparsimony, and prove that it contains all (and only) self-join-free conjunctive queries that admit parsimonious counting.

2012 ACM Subject Classification Information systems → Relational database query languages; Theory of computation → Incomplete, inconsistent, and uncertain databases; Theory of computation → Logic and databases

Keywords and phrases Consistent query answering, primary key, conjunctive query, aggregation, counting

Digital Object Identifier 10.4230/LIPIcs.ICDT.2023.23

Related Version *Extended Version*: `https://doi.org/10.48550/arXiv.2211.04134` [25]

1 Introduction

The problem of consistent query answering (CQA) [2, 4, 5, 37] with respect to primary keys is by now well understood for self-join-free conjunctive queries: a dichotomy between tractable and intractable queries has been established, and it is known which queries have a consistent first-order rewriting [29, 32]. It remains a largely open question to extend these complexity results to queries with aggregation. In this paper, we look at a simple form of aggregation: counting the number of times each (possibly composite) value occurs in the answer to a conjunctive query. Although this problem has been studied since the early years of CQA [18], a fine-grained characterization of its complexity remains open.

Formally, let q be a full (i.e., quantifier-free) self-join-free conjunctive query. We define a counting query as follows. We designate a tuple $\vec{z}$ of distinct variables of q, called the *grouping variables*, and let $\vec{w}$ be a tuple of the variables in q that are not in $\vec{z}$. The variables of q, which are all free, are made explicit by denoting q as $q(\vec{z}, \vec{w})$. We are interested in a query that, on a given database instance **db**, returns all tuples $(\vec{c}, i)$ with $\vec{c}$ a tuple of constants, of the same arity as $\vec{z}$, and with i a positive integer that is the number of distinct tuples $\vec{d}$, of the same arity as $\vec{w}$, satisfying $(\vec{c}, \vec{d}) \in q(\mathbf{db})$. This counting query will be denoted $\mathbf{cnt}(q, \vec{z})$.

For example, consider the database schema of Fig. 1, which is intended to store the unique gender and department of each employee, and the unique building of each department. Ignore for now that the database instance of Fig. 1 is inconsistent (as it stores two departments for Anny, and two buildings for IT). Let $q_0(x, y, z) = E(x, \text{'F'}, y) \wedge D(y, z)$, where x, y, z are variables and 'F' denotes a constant. Then, on a consistent database instance, $\mathbf{cnt}(q_0, z)$ would return the number of female employees working in each building. In SQL, $\mathbf{cnt}(q_0, z)$ can be encoded as follows:

```
SELECT   Building, COUNT(*) AS CNT
FROM     E, D
WHERE    E.Dept = D.Dept AND Gender = 'F'
GROUP BY Building
```

On the database instance of Fig. 1, this query will return $(A, 4)$ and $(B, 3)$. These answers are however not meaningful because they suffer from double-counting due to inconsistencies. We describe next a more meaningful semantics that was introduced in [2].

First, following [1], we define a *repair* of a database instance as a maximal subinstance that satisfies all primary-key constraints. In this paper, we consider no other constraints than primary keys. Then, following the approach of [2], more meaningful answers are obtained by returning, for every value $\vec{c}$ for the grouping variables $\vec{z}$, tight lower and upper bounds on the corresponding counts over all repairs. This new query is denoted by $\mathbf{cqacnt}(q, \vec{z})$. Thus, an answer $(\vec{c}, [m, n])$ to this new query means that on every repair, our original query $\mathbf{cnt}(q, \vec{z})$ returns a tuple $(\vec{c}, i)$ with $m \leq i \leq n$, and, moreover, these bounds m and n are tight. By tight, we mean that for every $j \in \{m, n\}$, there is a repair on which $\mathbf{cnt}(q, \vec{z})$ returns $(\vec{c}, j)$.

For example, the database instance of Fig. 1 has four repairs, because there are two choices for Anny's department, and two choices for the building of the IT department. Note that in Fig. 1, *blocks* of conflicting tuples are separated by dashed lines. The query $\mathbf{cnt}(q_0, z)$ returns different answers on each repair: there are two repairs where the answer is $\{(A, 3), (B, 1)\}$; there is one repair where the answer is $\{(A, 1), (B, 3)\}$; and there is one repair where the answer is $\{(A, 2), (B, 2)\}$. The latter set of answers, for example, is obtained in the repair that assigns Anny to department HR, and IT to building B. The query $\mathbf{cqacnt}(q_0, z)$ would thus return $\{(A, [1, 3]), (B, [1, 3])\}$.

In this paper, we are concerned about the complexity of computing $\mathbf{cqacnt}(q, \vec{z})$. In general, there exist self-join-free conjunctive queries q such that, for some choice of the grouping variables $\vec{z}$, $\mathbf{cqacnt}(q, \vec{z})$ cannot be solved in polynomial time (under standard complexity assumptions). This follows from earlier research showing that there are self-join-free conjunctive queries $q'(\vec{z})$ for which the following problem is coNP-complete: given $\vec{c}$ and $\mathbf{db}$, determine whether $q'(\vec{c})$ is true in every repair of $\mathbf{db}$. The latter problem obviously reduces to counting: $q'(\vec{c})$ is true in every repair of $\mathbf{db}$ if and only if $\mathbf{cqacnt}(q, \vec{z})$ returns $(\vec{c}, [m, n])$ on $\mathbf{db}$ for some $m \geq 1$, where q is the full query obtained from q' by dropping quantification.

In his PhD thesis [18], Fuxman showed that for some q and $\vec{z}$, the answer to $\mathbf{cqacnt}(q, \vec{z})$ can be computed by executing first-order queries followed by simple counting steps. To illustrate his approach, consider the following query in SQL:

```
SELECT   Building, COUNT(DISTINCT Emp) AS CNT
FROM     E, D
WHERE    E.Dept = D.Dept AND Gender = 'F'
GROUP BY Building
```

E

Emp	*Gender*	*Dept*
Suzy	F	HR
Anny	F	HR
Anny	F	IT
Grety	F	IT
Lucy	F	MIS

D

Dept	*Building*
HR	A
IT	A
IT	B
MIS	B

Figure 1 Example database. Primary keys are underlined.

On our example database of Fig. 1, this query returns $\{(A, 3), (B, 3)\}$. We observe that the returned counts match the upper bounds previously found for $\mathbf{cqacnt}(q_0, z)$. Importantly, it can be shown that this is not by accident: on *every* database instance, the latter SQL query will return the correct upper bounds for $\mathbf{cqacnt}(q_0, z)$. Note that the latter query uses `COUNT(DISTINCT Emp)`, which means that duplicates are removed, which is a standard practice in relational algebra.

We now explain how to obtain the lower bounds for our example query. To this end, consider the following query:

```
SELECT Building, Emp
FROM   E, D
WHERE  E.Dept = D.Dept AND Gender = 'F'
```

Following [1], we define the *consistent answer* to such a query as the intersection of the query answers on all repairs. For our example database, the consistent answer is the following table, which we call C:

C

Emp	*Building*
Suzy	A
Lucy	B

Note that Anny does not occur in the consistent answer, because (Anny, A) is false in some repair, and so is (Anny, B). From [29], it follows that computing the consistent answers to the latter SQL query is in FO (i.e., the class of problems that can be solved by a first-order query), using a technique known as *consistent first-order rewriting.* The lower bounds $\{(A, 1), (B, 1)\}$ are now found by executing the following query on C (and, again, this is not by accident):

```
SELECT   Building, COUNT(DISTINCT Emp) AS CNT
FROM     C
GROUP BY Building
```

Since C can be expressed in SQL, we can actually construct a single SQL query that computes the lower bounds in $\mathbf{cqacnt}(q_0, z)$.

In general, if $q(\vec{z}, \vec{w})$ is a full self-join-free conjunctive query for which $\mathbf{cqacnt}(q, \vec{z})$ can be computed as previously described, then we will say that the query obtained from q by existentially binding the variables in $\vec{w}$ (i.e., by binding the variables that are not grouping variables) admits *parsimonious counting.* Thus, our example showed that $\exists x \exists y\, E(x, \text{`F'}, y) \wedge D(y, z)$ admits parsimonious counting. A formal definition of parsimonious counting will be given later on (Definition 8). In this introduction, we content ourselves by saying that parsimonious counting, if possible, computes $\mathbf{cqacnt}(q, \vec{z})$ by executing two first-order queries (one for lower bounds, and one for upper bounds), followed by simple counting steps.

The main contribution of our paper can now be described. In his doctoral dissertation [18], Fuxman defined a class of self-join-free conjunctive queries, called Cforest, and showed the following.

▶ **Theorem 1** ([18]). *Every query in* Cforest *admits parsimonious counting.*

The class Cforest has been used in several studies on consistent query answering. It was an open question whether Cforest contains *all* self-join-free conjunctive queries that admit parsimonious counting. We will answer this question negatively in Section 8. More fundamentally, we introduce a new syntactic class, called Cparsimony, which includes Cforest and contains *all* (and only) self-join-free conjunctive queries that admit parsimonious counting. That is, we prove the following theorem.

▶ **Theorem 2** (Main theorem). *For every self-join-free conjunctive query q, it holds that q admits parsimonious counting if and only if q is in* Cparsimony.

Moreover, a new and simpler proof for Theorem 1 will follow in Section 8.

The remainder of this paper is organized as follows. Section 2 discusses related work. Section 3 introduces preliminary constructs and notations. Section 4 introduces the semantic notion of parsimonious counting. Section 5 introduces our new syntactic class of queries, called Cparsimony, which restricts self-join-free conjunctive queries. Section 6 shows that every query in Cparsimony admits parsimonious counting, and Section 7 shows that Cparsimony contains every self-join-free conjunctive query that admits parsimonious counting. Section 8 shows that Cforest is strictly included in Cparsimony, and provides a new proof for Theorem 1. Section 9 concludes the paper. Several helping lemmas and proofs are available in [25].

2 Related Work

Consistent query answering (CQA) started by a seminal paper in 1999 co-authored by Arenas, Bertossi, and Chomicki [1], who introduced the notions of repair and consistent answer. Two years later, the same authors introduced the *range semantics* (with lower and upper bounds) for queries with aggregation [2, 3][4, Chapter 5], which has been commonly adopted ever since. In particular, it was adopted in the PhD thesis [18] of Fuxman, who provided Theorem 1 (albeit using different terminology) and its proof, and used this result in the implementation of the ConQuer system [19]. ConQuer aims at computations in first-order logic with counting (coined "parsimonious counting" in the current paper), which can be encoded in SQL. This is different from AggCAvSAT [15], a recent system by Dixit and Kolaitis, which uses powerful SAT solvers for computing range semantics, and thus can solve queries that are beyond the computational power of ConQuer. Aggregation queries were also studied in the context of CQA in [6].

Consistent query answering for self-join-free conjunctive queries q and primary keys has been intensively studied. Its decision variant, which was coined CERTAINTY(q) in 2010 [36], asks whether a Boolean query q is true in every repair of a given database instance. A systematic study of its complexity for self-join-free conjunctive queries had started already in 2005 [21], and was eventually solved in two journal articles by Koutris and Wijsen [29, 32], as follows: for every self-join-free Boolean conjunctive query q, CERTAINTY(q) is either in FO, L-complete, or coNP-complete, and it is decidable, given q, which case applies. This complexity classification extends to non-Boolean queries by treating free variables as constants. Other extensions beyond this trichotomy deal with foreign keys [23], more than one key per relation [31], negated atoms [30], or restricted self-joins [28]. For unions of conjunctive queries q, Fontaine [17] established interesting relationships between CERTAINTY(q) and Bulatov's dichotomy theorem for conservative CSP [7].

The counting variant of CERTAINTY(q), denoted ♯CERTAINTY(q), asks to count the number of repairs that satisfy some Boolean query q. This counting problem is fundamentally different from the range semantics in the current paper. For self-join-free conjunctive queries,

$\sharp$CERTAINTY(q) exhibits a dichotomy between FP and $\sharp$P-complete under polynomial-time Turing reductions [33]. This dichotomy has been shown to extend to queries with self-joins if primary keys are singletons [34], and to functional dependencies [11]. Calautti, Console, and Pieris present in [8] a complexity analysis of these counting problems under weaker reductions, in particular, under many-one logspace reductions. The same authors have conducted an experimental evaluation of randomized approximation schemes for approximating the percentage of repairs that satisfy a given query [9]. Other approaches to making CQA more meaningful and/or tractable include operational repairs [10, 12] and preferred repairs [26, 35].

Recent overviews of two decades of theoretical research in CQA are [5, 37]. It is worthwhile to note that theoretical research in CERTAINTY(q) has stimulated implementations and experiments in prototype systems [14, 16, 19, 20, 24, 27].

3 Preliminaries

We assume that every relation name R is associated with an *arity*, which is a positive integer. We assume that all *primary-key positions* precede all *non-primary-key positions*. We say that R has *signature* $[n, k]$ if R has arity n and primary-key positions $1, \dots, k$.

If R has signature $[n, k]$ and $s_1, \dots, s_n$ are variables or constants, then $R(s_1, \dots, s_n)$ is an *R-atom* (or simply *atom*), which will often be denoted as $R(\underline{s_1, \dots, s_k}, s_{k+1}, \dots, s_n)$ to distinguish between primary-key and non-primary-key positions. Two atoms $R_1(\underline{\vec{s}_1}, \vec{t}_1)$ and $R_2(\underline{\vec{s}_2}, \vec{t}_2)$ are said to be *key-equal* if $R_1 = R_2$ and $\vec{s}_1 = \vec{s}_2$. A *fact* is an atom in which no variable occurs. A *database instance* (or simply *database*) is a finite set of facts. A database instance $\mathbf{db}$ is *consistent* if it does not contain two distinct key-equal facts. A *repair* of $\mathbf{db}$ is a $\subseteq$-maximal consistent subset of $\mathbf{db}$.

If $\vec{s}$ is a tuple of variables or constants, then $|\vec{s}|$ denotes the arity of $\vec{s}$, and $\mathsf{vars}(\vec{s})$ denotes the set of variables occurring in $\vec{s}$. By an abuse of notation, if we use a tuple $\vec{z}$ of variables at places where a set of variables is expected, we mean $\mathsf{vars}(\vec{z})$. For an atom $F = R(\underline{\vec{s}}, \vec{t})$, we define $\mathsf{Key}(F) := \mathsf{vars}(\vec{s})$, $\mathsf{notKey}(F) := \mathsf{vars}(\vec{t}) \setminus \mathsf{vars}(\vec{s})$, and $\mathsf{vars}(F) := \mathsf{vars}(\vec{s}) \cup \mathsf{vars}(\vec{t})$. For example, if $F = R(\underline{c, x, x, y}, y, z, c)$, then $\mathsf{Key}(F) = \{x, y\}$ and $\mathsf{notKey}(F) = \{z\}$, where c is a constant.

Conjunctive Queries. A conjunctive query q is a first-order formula of the form:

$$\exists \vec{w} \left(R_1(\underline{\vec{x}_1}, \vec{y}_1) \wedge \cdots \wedge R_n(\underline{\vec{x}_n}, \vec{y}_n)\right), \tag{1}$$

where the variables of $\vec{w}$ are *bound*, and the other variables are *free*. Such a query is also denoted by $q(\vec{z})$ with $\vec{z}$ a tuple composed of the free variables. We write $\mathsf{vars}(q)$ for the set of variables that occur in q, and can assume $\mathsf{vars}(q) = \mathsf{vars}(\vec{w}) \cup \mathsf{vars}(\vec{z})$ without loss of generality. We say that q is *full* if all variables of $\mathsf{vars}(q)$ are free. We say that q is *self-join-free* if $i \neq j$ implies $R_i \neq R_j$. The quantifier-free part $R_1(\underline{\vec{x}_1}, \vec{y}_1) \wedge \cdots \wedge R_n(\underline{\vec{x}_n}, \vec{y}_n)$ of q is denoted $\mathsf{body}(q)$. By slightly overloading notation, we also use $\mathsf{body}(q)$ for the set $\{R_1(\underline{\vec{x}_1}, \vec{y}_1), \dots, R_n(\underline{\vec{x}_n}, \vec{y}_n)\}$. We write $\mathsf{free}(q)$ for the set of free variables in q.

If a self-join-free conjunctive query q is understood, and we use a relation name R at places where an atom is expected, then we mean the unique R-atom of q. If $\vec{c}$ is a tuple of constants of arity $|\vec{z}|$ and $\mathbf{db}$ a database instance, then $\mathbf{db} \models q(\vec{c})$ denotes that $q(\vec{c})$ is true in $\mathbf{db}$ using standard first-order semantics. If $\mathbf{db} \models q(\vec{c})$, we also write $\vec{c} \in q(\mathbf{db})$, and we say that $\vec{c}$ is an *answer* to q on $\mathbf{db}$.

We now introduce operators for turning bound variables into free variables, or vice versa, and for instantiating free variables.

Making bound variables free. Let q be a conjunctive query with $\mathsf{free}(q) = \vec{z}$. Let $\vec{x}$ be a tuple of (not necessarily all) bound variables in q (hence $\vec{x} \cap \vec{z} = \emptyset$). We write $\nexists\vec{x}\,[q]$ for the conjunctive query q' such that $\mathsf{free}(q') = \vec{z} \cup \vec{x}$ and $\mathsf{body}(q') = \mathsf{body}(q)$. Informally, $\nexists\vec{x}\,[q]$ is obtained from q by omitting the quantification $\exists\vec{x}$. For example, if $q(z) = \exists x \exists y\, R(x, y) \wedge R(y, z)$, then $\nexists x\,[q] = \exists y\, R(x, y) \wedge R(y, z)$.

Binding free variables. Let q be a conjunctive query, and $\vec{x}$ a tuple of (not necessarily all) free variables of q. Then $\exists\vec{x}\,[q]$ denotes the query with the same body as q, but whose set of free variables is $\mathsf{free}(q) \setminus \vec{x}$.

Instantiating free variables. Let q be a conjunctive query, and $\vec{z}$ a tuple of distinct free variables of q. Let $\vec{c}$ be a tuple of constants of arity $|\vec{z}|$. Then $q_{[\vec{z}\to\vec{c}]}$ is the query obtained from q by replacing, for every $i \in \{1, 2, \dots, |\vec{z}|\}$, each occurrence of the ith variable in $\vec{z}$ by the ith constant in $\vec{c}$.

Consistent Query Answering. Let $q(\vec{z})$ be a conjunctive query. We write $\mathbf{db} \models_{\mathsf{cqa}} q(\vec{c})$ if for every repair $\mathbf{r}$ of $\mathbf{db}$, we have $\mathbf{r} \models q(\vec{c})$. If $\mathbf{db} \models_{\mathsf{cqa}} q(\vec{c})$, we also say that $\vec{c}$ is a *consistent answer* to q on $\mathbf{db}$. A *consistent first-order rewriting* of $q(\vec{z})$ is a first-order formula $\varphi(\vec{z})$ such that for every database instance $\mathbf{db}$ and every tuple $\vec{c}$ of constants of arity $|\vec{z}|$, we have $\mathbf{db} \models_{\mathsf{cqa}} q(\vec{c})$ if and only if $\mathbf{db} \models \varphi(\vec{c})$. Note incidentally that the set of integrity constraints is always implicitly understood to be the primary keys associated with the relation names that occur in the query.

Query Graph. The *query graph* of a conjunctive query $q(\vec{z})$ is an undirected graph whose vertices are the bound variables of q. There is an edge between x and y if $x \neq y$ and x, y occur together in some atom of $\mathsf{body}(q)$.

Attack Graph. The following is a straightforward extension of attack graphs [29] to deal with free variables.

Let $q(\vec{z})$ be a self-join-free conjunctive query. If S is a subset of $\mathsf{body}(q)$, then $q \setminus S$ denotes the query obtained from q by removing from q all atoms in S. Every variable of $q \setminus S$ that is free in q remains free in $q \setminus S$; and every variable of $q \setminus S$ that is bound in q remains bound in $q \setminus S$.

We define $\mathcal{K}(q)$ as the set of functional dependencies that contains $\emptyset \to \mathsf{free}(q)$ and contains, for every atom F in q, the functional dependency $\mathsf{Key}(F) \to \mathsf{vars}(F)$. Note that since $\mathcal{K}(q)$ contains $\emptyset \to \mathsf{free}(q)$, we have that $\mathcal{K}(q) \models \emptyset \to y$ if and only if $\mathcal{K}(q) \models \mathsf{free}(q) \to y$, for each $y \in \mathsf{vars}(q)$. If F is an atom of q, then $F^{+,q}$ is the set that contains every variable $y \in \mathsf{vars}(q)$ such that either $y \in \mathsf{free}(q)$ or $\mathcal{K}(q \setminus \{F\}) \models \mathsf{Key}(F) \to y$ (or both).

It is known that in the study of consistent query answering for self-join-free conjunctive queries, we often do not need a special treatment of free variables, because computing consistent answers to $q(\vec{z})$ has the same time complexity as the decision problem $\mathsf{CERTAINTY}(q(\vec{z})_{[\vec{z}\to\vec{c}]})$ with $\vec{c}$ a sequence of pairwise distinct fresh constants. The addition of functional dependencies $\emptyset \to \mathsf{free}(q)$ has the same effect as treating variables in $\mathsf{free}(q)$ as constants. In the following example, we omit curly braces and commas when denoting sets of variables. For example, $\{z_1, z_2\}$ is denoted $z_1 z_2$.

▶ **Example 3.** Let $q = \exists u \exists v \exists x \exists y\, R(\underline{u}, x) \wedge S(\underline{x, z_1}, y) \wedge T(\underline{y}, v, z_2) \wedge U(\underline{y}, u)$. We have $\mathsf{free}(q) = z_1 z_2$. Then, $q \setminus \{T\}$[1] is the query $\exists u \exists v \exists x \exists y\, R(\underline{u}, x) \wedge S(\underline{x, z_1}, y) \wedge U(\underline{y}, u)$, whose only free variable is z_1. Note incidentally that since v does not occur in the latter query, the

[1] Recall that we use T as a shorthand for the T-atom of q.

quantification $\exists v$ can be dropped. We have $\mathcal{K}(q \setminus \{T\}) = \{\emptyset \rightarrow z_1, u \rightarrow x, xz_1 \rightarrow y, y \rightarrow u\}$. Note that $\emptyset \rightarrow z_1$ belongs to the latter set because z_1 is free in $q \setminus \{T\}$. The set of variables that are functionally dependent on $\mathsf{Key}(T)$ relative to $\mathcal{K}(q \setminus \{T\})$ is $uxyz_1$. Finally, we obtain $T^{+,q} = uxyz_1z_2$. Note that the latter set contains the variable z_2 that is free in q. ⌟

We say that an atom F of q *attacks* a variable x occurring in q, denoted $F \overset{q}{\rightsquigarrow} x$, if there exists a sequence $\langle x_1, x_2, \ldots, x_n \rangle$ of bound variables of q ($n \geq 1$) such that:

1. if two variables are adjacent in the sequence, then they occur together in some atom of q;
2. $x_1 \in \mathsf{notKey}(F)$ and $x_n = x$; and
3. for every $\ell \in \{1, \ldots, n\}$, $x_\ell \notin F^{+,q}$.

Such a sequence will be called a *witness* of $F \overset{q}{\rightsquigarrow} x$. We say that an atom F of q *attacks* another atom G of q, denoted $F \overset{q}{\rightsquigarrow} G$, if $F \neq G$ and F attacks some variable of $\mathsf{vars}(G)$. It is now easily verified that if F attacks G, then F also attacks a variable in $\mathsf{Key}(G)$. A variable or atom that is not attacked, is called *unattacked* (where q is understood from the context). The *attack graph* of q is a directed graph whose vertices are the atoms of q; there is a directed edge from F to G if $F \overset{q}{\rightsquigarrow} G$. A directed edge in the attack graph is called an *attack*. Koutris and Wijsen [29] showed the following.

▶ **Theorem 4** ([29]). *A self-join-free conjunctive query $q(\vec{z})$ has a consistent first-order rewriting if and only if its attack graph is acyclic.*

An attack from F to G is *weak* if $\mathcal{K}(q) \models \mathsf{Key}(F) \rightarrow \mathsf{Key}(G)$; otherwise it is strong. By a *component* of an attack graph, we always mean a maximal weakly connected component.

Let q be a self-join-free conjunctive query. Whenever the relationship $\mathcal{K}(q) \models Z \rightarrow w$ holds true, then there exists a sequential proof of it, as defined next.

Sequential Proof. Let $q(\vec{z})$ be a self-join-free conjunctive query, and $Z \subseteq \mathsf{vars}(q)$. Let $\langle F_1, F_2, \ldots, F_n \rangle$ be a (possibly empty) sequence of atoms in $\mathsf{body}(q)$ such that for every $i \in \{1, \ldots, n\}$, $\mathsf{Key}(F_i) \subseteq \mathsf{free}(q) \cup Z \cup \left(\bigcup_{j=1}^{i-1} \mathsf{vars}(F_j)\right)$. Such a sequence is called a *sequential proof* of $\mathcal{K}(q) \models Z \rightarrow w$, for every $w \in \mathsf{free}(q) \cup Z \cup \left(\bigcup_{j=1}^{n} \mathsf{vars}(F_j)\right)$. A sequential proof of $\mathcal{K}(q) \models Z \rightarrow w$ is called *minimal* if $\langle F_1, \ldots, F_{n-1} \rangle$ is not a sequential proof of $\mathcal{K}(q) \models Z \rightarrow w$.

4 Parsimonious Counting

Consider a conjunctive query $q(\vec{z}) = \exists \vec{w}\, B$, with B a quantifier-free conjunction of atoms (called the *body*). We introduce a query that takes a database instance $\mathbf{db}$ as input and returns, for every tuple $\vec{c} \in q(\mathbf{db})$, the number of valuations for $\vec{w}$ that make the query true.

▶ **Definition 5** ($\mathbf{cnt}(q, \vec{z})$). *Let $q(\vec{z}, \vec{w})$ be a full conjunctive query, in which notation it is understood that $\vec{z}$ and $\vec{w}$ are disjoint, duplicate-free tuples of variables. $\mathbf{cnt}(q, \vec{z})$ is the query that takes as input a database instance $\mathbf{db}$ and returns every tuple $(\vec{c}, i)$ for which the following hold:*

1. *$\vec{c}$ a tuple of constants of arity $|\vec{z}|$; and*
2. *i is a positive integer such that i is the number of distinct tuples $\vec{d}$, of arity $|\vec{w}|$, satisfying $\mathbf{db} \models q(\vec{c}, \vec{d})$.*

*A maximal set of answers to $q(\mathbf{db})$ that agree on $\vec{z}$ will also be called a $\vec{z}$-*group *(where q and $\mathbf{db}$ are implicitly understood). Thus, $\mathbf{cnt}(q, \vec{z})$ counts the number of tuples in each $\vec{z}$-group.*

The following definition introduces range consistent query answers as introduced in [2].

▶ **Definition 6** ($\mathbf{cqacnt}(q, \vec{z})$). *Let $q(\vec{z}, \vec{w})$ be a full conjunctive query, in which notation it is understood that $\vec{z}$ and $\vec{w}$ are disjoint, duplicate-free tuples of variables. $\mathbf{cqacnt}(q, \vec{z})$ is the query that takes as input a database instance $\mathbf{db}$ and returns every tuple $(\vec{c}, [m, n])$ for which the following hold:*

1. *for every repair $\mathbf{r}$ of $\mathbf{db}$, there exists $\vec{d}$ such that $\mathbf{r} \models q(\vec{c}, \vec{d})$;*
2. *there is a repair of $\mathbf{db}$ on which $\mathbf{cnt}(q, \vec{z})$ returns $(\vec{c}, m)$;*
3. *there is a repair of $\mathbf{db}$ on which $\mathbf{cnt}(q, \vec{z})$ returns $(\vec{c}, n)$; and*
4. *if $\mathbf{cnt}(q, \vec{z})$ returns $(\vec{c}, i)$ on some repair of $\mathbf{db}$, then $m \leq i \leq n$.*

If $(\vec{c}, [m, n])$ is an answer to $\mathbf{cqacnt}(q, \vec{z})$ on $\mathbf{db}$, then we will say that it is a range-consistent answer*. Note that if $(\vec{c}, [m, n])$ is a range-consistent answer, then, by definition, $\vec{c}$ is a consistent answer to $q(\vec{z})$, hence $m \geq 1$.*

The following proposition states that computing $\mathbf{cqacnt}(q(\vec{z}, \vec{w}), \vec{z})$ can be NP-hard, even if the query $\exists \vec{w}\, [q]$ has a consistent first-order rewriting.

▶ **Proposition 7.** *There exists a self-join-free conjunctive query $q(\vec{z})$ that has a consistent first-order rewriting such that $\mathbf{cqacnt}(\mathsf{body}(q), \vec{z})$ is NP-hard to compute.*

Proof sketch. In 3-DIMENSIONAL MATCHING (3DM), we are given a set $M \subseteq A_1 \times A_2 \times A_3$, where A_1, A_2, A_3 are disjoint sets having the same number n of elements. We are asked whether M contains a matching, that is, a subset $M' \subseteq M$ such that $|M'| = n$ and no two elements of M' agree in any coordinate. The problem 3DM is NP-complete [22].

Consider the query $q(z) = \exists x_1 \exists x_2 \exists x_3 \exists y\, Z(\underline{z}) \wedge \bigwedge_{i=1}^{3} \big(R_i(\underline{x_i}, y) \wedge S_i(\underline{x_i}, y)\big)$. The edge-set of q's attack graph is empty. Therefore, q's attack graph is acyclic. By Theorem 4, $q(z)$ has a consistent first-order rewriting. Let $M \subseteq A_1 \times A_2 \times A_3$ be an instance of 3DM. Let $\mathbf{db}_M$ be the database instance that contains $Z(\underline{c})$ and includes, for every $a_1a_2a_3$ in M, $\bigcup_{i=1}^{3}\{R_i(\underline{a_i}, a_1a_2a_3), S_i(\underline{a_i}, a_1a_2a_3)\}$. Moreover, $\mathbf{db}_M$ includes $\bigcup_{i=1}^{3}\{R_i(\underline{\perp_i}, \top), S_i(\underline{\perp_i}, \top)\}$, where $\perp_1, \perp_2, \perp_3, \top$ are fresh constants not in $A_1 \cup A_2 \cup A_3$. Clearly, $\mathbf{db}_M$ is first-order computable from M. It can now be verified that M has a matching if and only if for some ℓ, $\mathbf{cqacnt}(\mathsf{body}(q), z)$ returns $(c, [\ell, n + 1])$ on $\mathbf{db}_M$. ◀

Note that the foregoing proof can be easily adapted from 3DM to 2DM. That is, the query $q(z) = \exists x_1 \exists x_2 \exists y\, Z(\underline{z}) \wedge \bigwedge_{i=1}^{2} \big(R_i(\underline{x_i}, y) \wedge S_i(\underline{x_i}, y)\big)$ has a consistent first-order rewriting, but computing $\mathbf{cqacnt}(\mathsf{body}(q), z)$ is as hard as 2DM.

We now introduce the semantic notion of *parsimonious counting*, which was illustrated by the running example in Section 1. Informally, for a query $q(\vec{z})$ that admits parsimonious counting, it will be the case that on every database instance $\mathbf{db}$, the answers to $\mathbf{cqacnt}(\mathsf{body}(q), \vec{z})$ can be computed by a first-order query followed by a simple counting step.

▶ **Definition 8** (Parsimonious counting). *Let q be a conjunctive query with $\mathsf{free}(q) = \vec{z}$.*[2] *Let $\vec{x}$ be a (possibly empty) sequence of distinct bound variables of $q(\vec{z})$. We say that q* admits parsimonious counting *on $\vec{x}$ if the following hold (let $q'(\vec{z}, \vec{x}) = \nexists \vec{x}\, [q]$):*

(A) *$q(\vec{z})$ has a consistent first-order rewriting;*
(B) *$q'(\vec{z}, \vec{x})$ has a consistent first-order rewriting (call it $\varphi(\vec{z}, \vec{x})$); and*
(C) *for every database instance $\mathbf{db}$, the following conditions* (Ca) *and* (Cb) *are equivalent:*

[2] We will commonly write $q(\vec{z})$ to make explicit that $\mathsf{free}(q) = \vec{z}$.

(a) $(\vec{c}, [m, n])$ *is an answer to* $\mathbf{cqacnt}(\mathsf{body}(q), \vec{z})$ *on* $\mathbf{db}$*;*
(b) $m \geq 1$ *and both the following hold:*
  (i) m *is the number of distinct tuples* $\vec{d}$*, of arity* $\vec{x}$*, such that* $\mathbf{db} \models \varphi(\vec{c}, \vec{d})$*; and*
  (ii) n *is the number of distinct tuples* $\vec{d}$ *such that* $\mathbf{db} \models q'(\vec{c}, \vec{d})$.

We say that q admits parsimonious counting *if it admits parsimonious counting on some sequence* $\vec{x}$ *of bound variables.*

Significantly, since Definition 8 contains a condition that must hold for every database instance $\mathbf{db}$, it does not give us an efficient procedure for deciding whether a given self-join-free query $q(\vec{z})$ admits parsimonious counting.

We now give some examples. From the proof of Proposition 7 and the paragraph after that proof, it follows that under standard complexity assumptions, for $k \geq 2$,

$$q_k(z) := \exists x_1 \cdots \exists x_k \exists y\, Z(\underline{z}) \wedge \bigwedge_{i=1}^{k} \left(R_i(\underline{x_i}, y) \wedge S_i(\underline{x_i}, y)\right)$$

does not admit parsimonious counting, even though $q_k(z)$ has a consistent first-order rewriting. The following example shows a query $q(z)$ that does not admit parsimonious counting, but for which $\mathbf{cqacnt}(\mathsf{body}(q), z)$ can be computed in first-order logic with a counting step that is slightly more involved than what is allowed in parsimonious counting.

▶ **Example 9.** Let $q(z) = \exists x \exists y\, R(\underline{z}, x) \wedge S(\underline{x, y})$ and $q^*(z, x, y) = R(\underline{z}, x) \wedge S(\underline{x, y})$. We first argue that $q(z)$ does not admit parsimonious counting. Let $\mathbf{db}$ be the following database instance:

R	$\underline{z}$	x
	c_1	a
	c_2	a
	c_2	b

S	$\underline{x}$	$\underline{y}$
	a	d
	a	e
	b	f

It can be verified that on this database instance, $\mathbf{cqacnt}(q^*, z)$ must return $(c_1, [2, 2])$ and $(c_2, [1, 2])$. We next show the answer to q^* on $\mathbf{db}$:

$q^*(\mathbf{db})$	z	x	y
	c_1	a	d
	c_1	a	e
	c_2	a	d
	c_2	a	e
	c_2	b	f

The correct upper bound of 2 in $(c_2, [1, 2])$ could only be obtained by counting, within the c_2-group, the number of distinct $\langle x \rangle$-values. However, such a counting would conclude an incorrect upper bound of 1 for the c_1-group. It is now correct to conclude that $q(z)$ does not admit parsimonious counting.

The lower and upper bounds can be obtained from $q^*(\mathbf{db})$ by a counting step that is only slightly more involved than what is allowed in parsimonious counting. First, construct the following relation where $\tilde{R}(\underline{c_j}, v \mid n)$ means that $\mathbf{cnt}(q^*, z)$ returns (c_j, n) on a repair that contains $R(\underline{c_j}, v)$.

$\tilde{R}$	$\underline{z}$	x	
	c_1	a	2
	c_2	a	2
	c_2	b	1

These counts can be obtained from $q^*(\mathbf{db})$ by counting the number of distinct y-values within each zx-group. Next, the lower and upper bounds are obtained as the minimal and maximal counts within each z-group.

Note incidentally that for $q_0(z) := \exists x \exists y\, R(\underline{z}, x) \wedge T(\underline{z}, x) \wedge S(\underline{x, y})$, which is obtained from $q(z)$ by adding $T(\underline{z}, x)$, we have that q_0 admits parsimonious counting. The change occurs because if $\mathbf{db} \models_{\mathsf{cqa}} q_0(c)$, then there exists a unique value a such that $\mathbf{db} \models \forall x\, (R(\underline{c}, x) \rightarrow x = a)$ and $\mathbf{db} \models \forall x\, (T(\underline{c}, x) \rightarrow x = a)$. That is, the only blocks that can contribute to $\mathbf{cqacnt}(\mathsf{body}(q_0), z)$ have cardinality 1. This means that range semantics reduces to counting on a consistent database instance. ⌟

5 The Class Cparsimony

The notion of parsimonious counting is a semantic property defined for conjunctive queries. A natural question is to syntactically characterize the class of conjunctive queries that admit parsimonious counting. In this paper, we will answer this question under the restriction that queries are self-join-free. This is the best we can currently hope for, because consistent query answering for primary keys and conjunctive queries with self-joins is a notorious open problem for which no tools are known (e.g., attack graphs are not helpful in the presence of self-joins). We now define our new syntactic class Cparsimony, which uses the following notion of *frozen variable.*

▶ **Definition 10** (Frozen variable). *Let $q(\vec{z})$ be a self-join-free conjunctive query. We say that a bound variable y of $q(\vec{z})$ is* frozen in q *if there exists a sequential proof of $\mathcal{K}(q) \models \emptyset \rightarrow y$ such that $F \overset{q}{\not\rightsquigarrow} y$ for every atom F that occurs in the sequential proof. We write* $\mathsf{frozen}(q)$ *for the set of all bound variables of* $\mathsf{vars}(q)$ *that are frozen in q. A bound variable that is not frozen in q is called* nonfrozen in q.

▶ **Example 11.** Let $q(z) = \exists x\, R(\underline{z}, x) \wedge S(\underline{z}, x)$. We have $R \overset{q}{\not\rightsquigarrow} x$. Therefore, $\langle R(\underline{z}, x)\rangle$ is a sequential proof of $\mathcal{K}(q) \models \emptyset \rightarrow x$ that uses no atom attacking x. Hence, x is frozen. Note here that z is free, hence $\mathcal{K}(q) \models \emptyset \rightarrow z$ by definition. ⌟

▶ **Definition 12** (The class Cparsimony). *We define* Cparsimony *as the set of self-join-free conjunctive queries $q(\vec{z})$ satisfying the following conditions:*

(I) *the attack graph of $q(\vec{z})$ is acyclic and contains no strong attacks; and*

(II) *there is a tuple $\vec{x}$ of bound variables of $q(\vec{z})$ such that:*

  (1) *every component[3] of $q(\vec{z})$'s attack graph contains an unattacked atom R such that $\mathcal{K}(q) \models \vec{x} \rightarrow \mathsf{Key}(R)$; and*

  (2) *for every atom R in $\mathsf{body}(q(\vec{z}))$, every (possibly empty) path in the query graph of $q(\vec{z})$ between a variable of $\mathsf{notKey}(R)$ and a variable of $\vec{x}$ uses a variable in $\mathsf{Key}(R) \cup \mathsf{frozen}(q)$.*

We will say that such an $\vec{x}$ is an id-set *for $q(\vec{z})$. We will say that an id-set $\vec{x}$ is* minimal *if any sequence obtained from $\vec{x}$ by omitting one or more variables is no longer an id-set.*

Informally, id-sets $\vec{x}$ will play the role of $\vec{x}$ in Definition 8: they identify the values that have to be counted within each $\vec{z}$-group to obtain range-consistent answers.

[3] Whenever we use the term component, we mean a maximal weakly connected component.

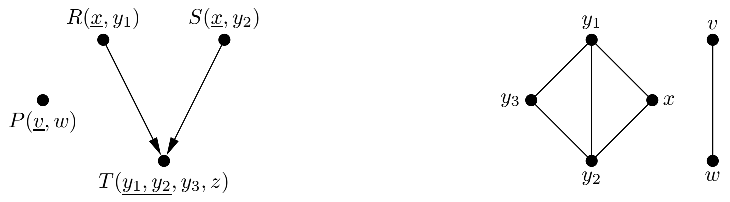


Figure 2 Attack graph *(left)* and query graph *(right)* of $q(z) = \exists x \exists y_1 \exists y_2 \exists y_3 \exists v \exists w\, R(\underline{x}, y_1) \wedge S(\underline{x}, y_2) \wedge T(\underline{y_1, y_2}, y_3, z) \wedge P(\underline{v}, w)$.

We now illustrate Definition 12 by some examples. Then Proposition 17 implies that every query $q(\vec{z})$ in Cparsimony has a unique minimal id-set that can be easily constructed from $q(\vec{z})$'s attack graph. Finally, Proposition 18 settles the complexity of checking membership in Cparsimony.

▶ **Example 13.** In the paragraph following the proof of Proposition 7, we introduced the query $q(z) = \exists x_1 \exists x_2 \exists y\, Z(\underline{z}) \wedge \bigwedge_{i=1}^{2} \left(R_i(\underline{x_i}, y) \wedge S_i(\underline{x_i}, y)\right)$. The edge-set of $q(z)$'s attack graph is empty. No variable is frozen. According to condition II1 in Definition 12, every id-set (if any) must contain x_1. However, no id-set can contain x_1, because for the atom $R_2(\underline{x_2}, y)$, the edge $\{y, x_1\}$ in the query graph is a path between a variable of notKey(R_2) and x_1 that uses no variable of Key(R_2). We conclude that $q(z)$ is not in Cparsimony. ⌟

▶ **Example 14.** The query $q(z) = \exists x \exists y \exists v\, R(\underline{x}, y) \wedge S(\underline{y}, v) \wedge T(\underline{v}, y) \wedge P_1(\underline{z}, y) \wedge P_2(\underline{z}, y)$ belongs to Cparsimony. The attack graph of $q(z)$ has a single attack from S to T. The query graph of $q(z)$ has two undirected edges: $\{x, y\}$ and $\{y, v\}$. The variable y is frozen, because $\langle P_1(\underline{z}, y)\rangle$ is a sequential proof of $\mathcal{K}(q) \models \emptyset \rightarrow y$ (note here that z is free), and $P_1 \overset{q}{\not\rightsquigarrow} y$.

It can be verified that $\langle x\rangle$ is an id-set. Note that $\langle y, x\rangle$ is a path in the query graph between $y \in$ notKey(T) and x that uses no variable of Key$(T) = \{v\}$. However, that path uses the frozen variable y. ⌟

▶ **Example 15.** Let $q(z) = \exists x \exists y_1 \exists y_2 \exists y_3 \exists v \exists w\, R(\underline{x}, y_1) \wedge S(\underline{x}, y_2) \wedge T(\underline{y_1, y_2}, y_3, z) \wedge P(\underline{v}, w)$. The attack graph and the query graph of $q(z)$ are shown in Fig. 2. We now argue that $q(z)$ is in Cparsimony. First, the attack graph of q is acyclic and contains no strong attacks. We next argue that xv is an id-set for q. The attack graph of $q(z)$ has two components. Condition II1 in Definition 12 is obviously satisfied for $\vec{x} = xv$ since $\mathcal{K}(q) \models xv \rightarrow v$ and $\mathcal{K}(q) \models xv \rightarrow x$. It is easily verified that condition II2 is also verified. In particular, for the atom $T(\underline{y_1, y_2}, y_3, z)$, every path between y_3 and x uses either y_1 or y_2. ⌟

▶ **Example 16.** Let $q(z) = \exists x \exists y\, R_1(\underline{x}, y, z) \wedge R_2(\underline{x}, y) \wedge S_1(\underline{y}, x) \wedge S_2(\underline{y}, x)$. The attack graph of $q(z)$ contains no edges and, thus, is acyclic and has four components. It can be verified that no variable is frozen. We claim that $q(z)$ is not in Cparsimony, because it has no id-set. Indeed, from condition II1 in Definition 12, it follows that every id-set must contain either x or y (or both). For the atom $S_1(\underline{y}, x)$, the empty path is a path between a variable in notKey(S_1) to x that uses no variable in Key(S_1). It follows by condition II2 that no id-set can contain x. From $R_2(\underline{x}, y)$, by similar reasoning, we conclude that no id-set can contain y. It follows that $q(z)$ has no id-set. ⌟

▶ **Proposition 17.** *Let $q(\vec{z})$ be a query in* Cparsimony*, and let $\vec{x}$ be a minimal id-set for it. Let $N = \bigcup\{\mathsf{notKey}(R) \mid R \in q\}$. Let V be a $\subseteq$-minimal subset of* $\mathsf{vars}(q)$ *that includes, for every unattacked atom R of q, every bound variable of* $\mathsf{Key}(R) \setminus N$*. Then,*

1. $V = \mathsf{vars}(\vec{x})$*; and*
2. *whenever R, S are unattacked atoms that are weakly connected in $q(\vec{z})$'s attack graph,* $\mathsf{Key}(R) \cap \vec{x} = \mathsf{Key}(S) \cap \vec{x}$*.*

▶ **Proposition 18.** *The following decision problem is in quadratic time: Given a self-join-free conjunctive query $q(\vec{z})$, decide whether or not $q(\vec{z})$ belongs to* Cparsimony*.*

6 The Class Cparsimony Admits Parsimonious Counting

In this section, we show the if-direction of Theorem 2, which is the following theorem.

▶ **Theorem 19.** *Every self-join-free conjunctive query in* Cparsimony *admits parsimonious counting.*

We use a number of helping lemmas and constructs. The following lemma says that if $\vec{x}$ is an id-set of a query $q(\vec{z})$ in Cparsimony, then for a consistent database **db**, the answers to $\mathbf{cnt}(\mathsf{body}(q), \vec{z})$ can be obtained by counting the number of distinct $\vec{x}$-values within each $\vec{z}$-group, while variables not in $\vec{x} \cdot \vec{z}$ can be ignored.

▶ **Lemma 20.** *Let $q(\vec{z}, \vec{x}, \vec{w})$ be a full self-join free conjunctive query, in which notation it is understood that $\vec{z}$, $\vec{x}$ and $\vec{w}$ are disjoint, duplicate-free tuples of variables. Assume that the query $\exists\vec{x}\vec{w}\,[q]$ belongs to* Cparsimony *and that $\vec{x}$ is an id-set for it. Let* **db** *be a consistent database instance. For all tuples $\vec{a}$ and $\vec{b}$ of constants, of arities $|\vec{z}|$ and $|\vec{x}|$ respectively, for all tuples $\vec{c_1}$ and $\vec{c_2}$ of arity $|\vec{w}|$, if* $\mathbf{db} \models q(\vec{a}, \vec{b}, \vec{c_1})$ *and* $\mathbf{db} \models q(\vec{a}, \vec{b}, \vec{c_2})$*, then $\vec{c_1} = \vec{c_2}$.*

We now present the notion of *optimistic repair*, which was originally introduced by Fuxman [18]. Informally, a repair **r** of a database **db** is an optimistic repair with respect to a conjunctive query $q(\vec{z})$ if every tuple that is an answer to $q(\vec{z})$ on **db** is also an answer to $q(\vec{z})$ on **r**. The converse obviously holds true because conjunctive queries are monotone and repairs are subsets of the original database instance.

▶ **Definition 21** (Optimistic repair). *Let $q(\vec{x})$ be a conjunctive query. Let* **db** *be a database instance. We say that a repair* **r** *of* **db** *is an* optimistic repair *with respect to $q(\vec{x})$ if for every tuple $\vec{a}$ of constants, of arity $|\vec{x}|$,* $\mathbf{db} \models q(\vec{a})$ *implies* $\mathbf{r} \models q(\vec{a})$ *(the converse implication is obviously true).*

The following lemma gives a sufficient condition for the existence of optimistic repairs.

▶ **Lemma 22.** *Let $q(\vec{z})$ be a self-join free conjunctive query in* Cparsimony*, and let $\vec{x}$ be a minimal id-set for it. Let $q'(\vec{z}, \vec{x})$ be the query $\nexists\vec{x}\,[q]$. Let* **db** *be a database instance, and $\vec{c}$ a tuple of constants, of arity $|\vec{z}|$, such that* $\mathbf{db} \models_{\mathsf{cqa}} q(\vec{c})$*. Then,* **db** *has an optimistic repair with respect to $q'_{[\vec{z} \to \vec{c}]}$.*

We now present the notion of *pessimistic repair*, also borrowed from [18]. Informally, a repair of a database **db** is a pessimistic repair with respect to a conjunctive query $q(\vec{z})$ if every answer to $q(\vec{z})$ on **r** is a consistent answer to $q(\vec{z})$ on **db**. The converse trivially holds true.

▶ **Definition 23** (Pessimistic repair). *Let $q(\vec{x})$ be a conjunctive query. Let* **db** *be a database instance. We say that a repair* **r** *of* **db** *is a pessimistic repair with respect to $q(\vec{x})$ if for every tuple $\vec{a}$ of constants, of arity $|\vec{x}|$, if* $\mathbf{r} \models q(\vec{a})$*, then* $\mathbf{db} \models_{\mathsf{cqa}} q(\vec{a})$*.*

The following lemma gives a sufficient condition for the existence of pessimistic repairs.

▶ **Lemma 24.** *Let $q(\vec{z})$ be a self-join free conjunctive query in* $\mathsf{Cparsimony}$*, and let $\vec{x}$ be a minimal id-set for it. Let* $\mathbf{db}$ *be a database instance, and $\vec{c}$ a tuple of constants, of arity $|\vec{z}|$, such that $\mathbf{db} \models_{\mathsf{cqa}} q(\vec{c})$. Then,* $\mathbf{db}$ *has a pessimistic repair with respect to $q'_{[\vec{z}\rightarrow\vec{c}]}$.*

The following example illustrates the preceding constructs and lemmas.

▶ **Example 25.** Let $q(z) = \exists x \exists y \exists v\, R(\underline{x}, y) \wedge S(\underline{y, v}, z) \wedge T(\underline{y}, v)$. Let $\mathbf{db}$ be the following database instance:

R	$\underline{x}$	y
	a_1	b_1
	a_2	b_2
	a_3	b_2
	a_4	b_3

S	$\underline{y}$	$\underline{v}$	z
	b_1	c_1	g_1
	b_2	c_2	g_1
	b_2	c_2	g_2
	b_3	c_3	g_2

T	$\underline{y}$	v
	b_1	c_1
	b_2	c_2
	b_3	c_3

Clearly, $\mathbf{db}$ has two repairs, which are $\mathbf{r}_1 := \mathbf{db} \setminus \{S(\underline{b_2, b_2}, g_2)\}$ and $\mathbf{r}_2 := \mathbf{db} \setminus \{S(\underline{b_2, b_2}, g_1)\}$.

We first determine the answers to $\mathbf{cqacnt}(\mathsf{body}(q), z)$ on $\mathbf{db}$ in a naive way without using parsimonious counting, but by enumerating repairs. To this end, let $q^*(z, x, y, v) = R(\underline{x}, y) \wedge S(\underline{y, v}, z) \wedge T(\underline{y}, v)$. We have:

$$q^*(\mathbf{r}_1) = \{(g_1, a_1, b_1, c_1), (g_1, a_2, b_2, c_2), (g_1, a_3, b_2, c_2), (g_2, a_4, b_3, c_3)\}$$
$$q^*(\mathbf{r}_2) = \{(g_1, a_1, b_1, c_1), (g_2, a_2, b_2, c_2), (g_2, a_3, b_2, c_2), (g_2, a_4, b_3, c_3)\}$$

The value g_1 occurs in 3 tuples of $q^*(\mathbf{r}_1)$, and in one tuple of $q^*(\mathbf{r}_2)$. On the other hand, g_2 occurs in one tuple of $q^*(\mathbf{r}_1)$, and in 3 tuples of $q^*(\mathbf{r}_2)$. It follows that $(g_1, [1, 3])$ and $(g_2, [1, 3])$ are the answers to $\mathbf{cqacnt}(\mathsf{body}(q), z)$ on $\mathbf{db}$.

It can be verified that $q(z) \in \mathsf{Cparsimony}$ with an id-set $\vec{x} = \langle x \rangle$. We next compute $\mathbf{cqacnt}(\mathsf{body}(q), z)$ on $\mathbf{db}$ by means of parsimonious counting. To this end, let $q'(z, x) = \nexists x\, [q]$, and let $\varphi(z, x)$ be a consistent first-order rewriting for $q'(z, x)$. If we execute these queries on $\mathbf{db}$, we obtain:[4]

$$q'(\mathbf{db}) = \{(g_1, a_1), (g_1, a_2), (g_1, a_3), (g_2, a_2), (g_2, a_3), (g_2, a_4)\}$$
$$\varphi(\mathbf{db}) = \{(g_1, a_1), (g_2, a_4)\}$$

As stated in Theorem 19, the set $q'(\mathbf{db})$ yields the upper bound 3 for g_1 and g_2, and the set $\varphi(\mathbf{db})$ yields the lower bound 1 for g_1 and g_2. It is important to understand that parsimonious counting obtains these bounds directly on $\mathbf{db}$, without computing any repair.

We elaborate this example further to illustrate the constructs of optimistic and pessimistic repairs. We have:

$$q'(\mathbf{r}_1) = \{(g_1, a_1), (g_1, a_2), (g_1, a_3), (g_2, a_4)\}$$
$$q'(\mathbf{r}_2) = \{(g_1, a_1), (g_2, a_2), (g_2, a_3), (g_2, a_4)\}$$

Note that the consistent answer to $q'(z, x)$ on $\mathbf{db}$ (i.e., the set $\varphi(\mathbf{db})$ used previously) is equal to $q'(\mathbf{r}_1) \cap q'(\mathbf{r}_2) = \{(g_1, a_1), (g_2, a_4)\}$. We see that $\mathbf{r}_1$ is an optimistic repair with respect to $q'(z, x)_{[z\rightarrow g_1]}$, and a pessimistic repair with respect to $q'(z, x)_{[z\rightarrow g_2]}$. On the other hand, $\mathbf{r}_2$ is an optimistic repair with respect to $q'(z, x)_{[z\rightarrow g_2]}$, and a pessimistic repair with respect to $q'(z, x)_{[z\rightarrow g_1]}$. ⌟

[4] $\varphi(\mathbf{db})$ is a shorthand for the set of all tuples (c, d) such that $\mathbf{db} \models \varphi(c, d)$.

Proof of Theorem 19. Let $q(\vec{z}) \in \mathsf{Cparsimony}$. We have to prove that $q(\vec{z})$ admits parsimonious counting. Since $q(\vec{z}) \in \mathsf{Cparsimony}$, we can assume an id-set $\vec{x}$ for $q(\vec{z})$. It suffices to show that conditions A, B, and C in Definition 8 are satisfied for this choice of $\vec{x}$. As in Definition 8, let $q'(\vec{z}, \vec{x}) = \nexists\vec{x}\,[q]$.

Since $q(\vec{z})$ is in $\mathsf{Cparsimony}$, it has an acyclic attack graph. It follows from Theorem 4 that $q(\vec{z})$ has a consistent first-order rewriting. Thus, condition A in Definition 8 is satisfied. It is known [29] that the attack graph of $q'(\vec{z}, \vec{x})$ is a subgraph of the attack graph of $q(\vec{z})$. Informally, no new attacks are introduced when bound variables are made free. It follows that $q'(\vec{z}, \vec{x})$ has an acyclic attack graph, and therefore, by Theorem 4, a consistent first-order rewriting. Thus, condition B in Definition 8 is satisfied. In the remainder of the proof, we show that condition C in Definition 8 is satisfied. To this end, let $\mathbf{db}$ be an arbitrary database instance.

Let $\vec{c}$ be a tuple of constants such that $\mathbf{db} \models_{\mathsf{cqa}} q(\vec{c})$. Let D be the active domain of $\mathbf{db}$. Let f be a function that maps every subset $\mathbf{s}$ of $\mathbf{db}$ to the cardinality of the set $\{\vec{a} \in D^{|\vec{x}|} \mid \mathbf{s} \models q'(\vec{c}, \vec{a})\}$. Clearly, for every repair $\mathbf{r}$ of $\mathbf{db}$, we have $\mathbf{r} \subseteq \mathbf{db}$ and hence, since conjunctive queries are monotone, $f(\mathbf{r}) \leq f(\mathbf{db})$. Moreover, since repairs are consistent, it follows by Lemma 20 that for every repair $\mathbf{r}$ of $\mathbf{db}$, if $(\vec{c}, i)$ is an answer to the query $\mathbf{cnt}(\mathsf{body}(q), \vec{z})$ on $\mathbf{r}$, then $i = f(\mathbf{r})$.

By Lemma 22, we can assume an optimistic repair $\mathbf{o}$ of $\mathbf{db}$ with respect to $q'(\vec{z}, \vec{x})_{[\vec{z}\to\vec{c}]}$. By Definition 21 of optimistic repair, for every tuple $\vec{a}$ of constants, of arity $|\vec{x}|$, we have $\mathbf{o} \models q'(\vec{c}, \vec{a})$ if and only if $\mathbf{db} \models q'(\vec{c}, \vec{a})$. It follows $f(\mathbf{o}) = f(\mathbf{db})$. Consequently, for every repair $\mathbf{r}$ of $\mathbf{db}$, $f(\mathbf{r}) \leq f(\mathbf{o})$. It follows that for some lower bound m, we have that $(\vec{c}, [m, f(\mathbf{db})]$ is an answer to $\mathbf{cqacnt}(\mathsf{body}(q), \vec{z})$ on $\mathbf{db}$.

By Lemma 24, we can assume a pessimistic repair $\mathbf{p}$ of $\mathbf{db}$ with respect to $q'(\vec{z}, \vec{x})_{[\vec{z}\to\vec{c}]}$. Let $\varphi(\vec{z}, \vec{x})$ be a consistent first-order rewriting of $q'(\vec{z}, \vec{x})$. By Definition 23 of pessimistic repair, the following hold:

- $\mathbf{p} \models q(\vec{c}, \vec{a})$ if and only if $\mathbf{db} \models \varphi(\vec{c}, \vec{a})$. Therefore, $f(\mathbf{p})$ is the cardinality of the set $S := \{\vec{a} \in D^{|\vec{x}|} \mid \mathbf{db} \models \varphi(\vec{c}, \vec{a})\}$.
- for every repair $\mathbf{r}$ of $\mathbf{db}$, $f(\mathbf{p}) \leq f(\mathbf{r})$.

It follows that there is an upper bound n such that that $(\vec{c}, [|S|, n])$ is an answer to $\mathbf{cqacnt}(\mathsf{body}(q), \vec{z})$ on $\mathbf{db}$. Putting everything together, we obtain that $(\vec{c}, [|S|, f(\mathbf{db})])$ is an answer to $\mathbf{cqacnt}(\mathsf{body}(q), \vec{z})$ on $\mathbf{db}$. From this, it is correct to conclude that condition C in Definition 8 is satisfied. This concludes the proof. ◀

7 Completeness of the Class Cparsimony

In this section, we show the only-if-direction of Theorem 2, which is the following theorem.

▶ **Theorem 26.** *Every self-join-free conjunctive query that admits parsimonious counting belongs to* $\mathsf{Cparsimony}$.

The following three lemmas state some properties of queries $q(\vec{z})$ that admit parsimonious counting on some $\vec{x}$.

▶ **Lemma 27.** *Let $q(\vec{z})$ be a self-join-free conjunctive query. If $q(\vec{z})$ admits parsimonious counting, then the attack graph of $q(\vec{z})$ is acyclic.*

▶ **Lemma 28.** *Let $q(\vec{z})$ be a self-join-free conjunctive query. Let $\vec{x}$ be a (possibly empty) sequence of bound variables of $q(\vec{z})$. If $q(\vec{z})$ admits parsimonious counting on $\vec{x}$, then the attack graph of $q'(\vec{z}, \vec{x})$ has no strong attack.*

▶ **Lemma 29.** *Let $q(\vec{z})$ be self-join-free conjunctive query whose attack graph is acyclic. Let $\vec{x}$ be a (possibly empty) sequence of bound variables of $q(\vec{z})$. If $q(\vec{z})$ admits parsimonious counting on $\vec{x}$, then $\vec{x}$ satisfies condition II1 in Definition 12.*

The following two lemmas, and their corollary, concern condition II2 in Definition 12.

▶ **Lemma 30.** *Let $q(\vec{z})$ be a self-join-free conjunctive query. Let $\vec{x}$ be a (possibly empty) sequence of bound variables of $q(\vec{z})$, and let $q'(\vec{z},\vec{x}) = \nexists\vec{x}\,[q]$. Let $\vec{c}$ a tuple of constants of arity $|\vec{z}|$. If $q(\vec{z})$ admits parsimonious counting on $\vec{x}$, then for every database instance* **db**, *if* $\mathbf{db} \models_{\mathsf{cqa}} q(\vec{c})$, *then* **db** *has an optimistic repair with respect to $q'_{[\vec{z}\rightarrow\vec{c}]}$.*

Proof. Assume that $q(\vec{z})$ admits parsimonious counting on $\vec{x}$. Let **db** be a database instance such that $\mathbf{db} \models_{\mathsf{cqa}} q(\vec{c})$. Let $(\vec{c},[m,n])$ be an answer to $\mathbf{cqacnt}(\mathsf{body}(q),\vec{z})$ on **db**. Define

$$\mathcal{D} := \{\vec{d} \in D^{|\vec{x}|} \mid \mathbf{db} \models q'(\vec{c},\vec{d})\}, \tag{2}$$

where D be the active domain of **db**. By our hypothesis that $q(\vec{z})$ admits parsimonious counting on $\vec{x}$, it follows by condition C in Definition 8 that

$$n = |\mathcal{D}|. \tag{3}$$

By Definition 6, we can assume a repair $\mathbf{r}$ of **db** such that $(\vec{c},n)$ is an answer to $\mathbf{cnt}(\mathsf{body}(q),\vec{z})$ on $\mathbf{r}$. Since $\mathbf{r}$ is consistent, we have that $(\vec{c},[n,n])$ is an answer to $\mathbf{cqacnt}(\mathsf{body}(q),\vec{z})$ on $\mathbf{r}$. Define

$$\mathcal{R} := \{\vec{d} \in D^{|\vec{x}|} \mid \mathbf{r} \models q'(\vec{c},\vec{d})\}. \tag{4}$$

By our hypothesis that $q(\vec{z})$ admits parsimonious counting on $\vec{x}$, it follows by condition C in Definition 8 that

$$n = |\mathcal{R}|. \tag{5}$$

Since conjunctive queries are monotone and $\mathbf{r} \subseteq \mathbf{db}$, it follows $\mathcal{R} \subseteq \mathcal{D}$. Since $|\mathcal{R}| = |\mathcal{D}|$ by (3) and (5), it follows $\mathcal{R} = \mathcal{D}$. From $\mathcal{D} \subseteq \mathcal{R}$, it follows that $\mathbf{r}$ is an optimistic repair with respect to $q'(\vec{z},\vec{x})_{[\vec{z}\rightarrow\vec{c}]}$. ◀

▶ **Lemma 31.** *Let $q(\vec{z})$, $\vec{x}$, $q'(\vec{z},\vec{x})$, and $\vec{c}$ be as in the statement of Lemma 30. Assume that $\vec{x}$ violates condition II2 in Definition 12. Then, there exists a database* **db** *such that* $\mathbf{db} \models_{\mathsf{cqa}} q(\vec{c})$, *but* **db** *has no optimistic repair with respect to $q'_{[\vec{z}\rightarrow\vec{c}]}$.*

▶ **Corollary 32.** *Let $q(\vec{z})$ be a self-join-free conjunctive query. Let $\vec{x}$ be a sequence of distinct bound variables of $q(\vec{z})$, and let $q'(\vec{z},\vec{x}) = \nexists\vec{x}\,[q]$. If $q(\vec{z})$ admits parsimonious counting on $\vec{x}$, then $\vec{x}$ satisfies condition II2 in Definition 12.*

Proof. Immediately from Lemmas 30 and 31. ◀

Finally, we need the following result.

▶ **Lemma 33.** *Let $q(\vec{z})$ be a self-join-free conjunctive query. Let $\vec{x}$ be a sequence of distinct bound variables of $q(\vec{z})$. Let $q'(\vec{z},\vec{x}) = \nexists\vec{x}\,[q]$. Assume that $\vec{x}$ satisfies condition II2 in Definition 12. If the attack graph of $q(\vec{z})$ has a strong attack from an atom R to an atom S, then the attack graph of $q'(\vec{z},\vec{x})$ has a strong attack from R to S.*

Before giving a proof of Theorem 26, we illustrate the preceding results with an example.

▶ **Example 34.** Let $q(z) = \exists x \exists y\, R(\underline{x}, z, y) \wedge S(\underline{y}, x) \wedge T(\underline{y}, x)$. We will argue that $q(z)$ is not in Cparsimony, and then illustrate that it does not admit parsimonious counting.

The only edges in the attack graph of q are (R, S) and (R, T). Assume for the sake of contradiction that $q \in$ Cparsimony. Then, following Proposition 17, the minimal id-set of $q(\vec{z})$ is $\langle\rangle$. However, since $\mathcal{K}(q(z)) \equiv \{x \to y, y \to x, \emptyset \to z\}$, condition II1 in Definition 12 is violated for $\vec{x} = \langle\rangle$. We conclude by contradiction that $q \notin$ Cparsimony.

We now argue, without using Theorem 26, that $q(z)$ does not admit parsimonious counting. Conditions A and B in Definition 8 of parsimonious counting are satisfied for every choice of $\vec{x}$ in $\{\langle\rangle, \langle x\rangle, \langle y\rangle, \langle x, y\rangle\}$. However, we will show that condition C is not satisfied. To this end, let $\vec{x}$ be a sequence of bound variables of $q(z)$. Let $q'(z, \vec{x}) = \nexists \vec{x}\, [q]$. First, suppose that $\vec{x} \in \{\langle x\rangle, \langle x, y\rangle\}$. Consider the following database instance **db**:

R	$\underline{x}$	z	y
	a	d	e
	b	d	e
	c	d	f

S	$\underline{y}$	x
	e	a
	e	b
	f	c

T	$\underline{y}$	x
	e	a
	e	b
	f	c

We have that $(d, [1, 2])$ is an answer to $\mathbf{cqacnt}(\mathsf{body}(q), z)$, but it can be easily verified that $|q'(\mathbf{db})| = 3$, which is distinct from the upper bound 2.

Assume next that $\vec{x} \in \{\langle y\rangle, \langle x, y\rangle\}$. Consider the following database instance **db**:

R	$\underline{x}$	z	y
	a	d	e
	a	d	f
	b	d	g

S	$\underline{y}$	x
	e	a
	f	a
	g	b

T	$\underline{y}$	x
	e	a
	f	a
	g	b

Now we have that $(d, [2, 2])$ is an answer to $\mathbf{cqacnt}(\mathsf{body}(q), z)$, but $|q'(\mathbf{db})| = 3$.

The only remaining case to be considered is $\vec{x} = \langle\rangle$. In that case $q' = q$. Consider the following database instance **db**:

R	$\underline{x}$	z	y
	a	d	e
	b	d	f

S	$\underline{y}$	x
	e	a
	f	b

T	$\underline{y}$	x
	e	a
	f	b

Since **db** is a consistent database instance, the only repair of **db** is **db** itself. We have that $(d, [2, 2])$ is an answer to $\mathbf{cqacnt}(\mathsf{body}(q), z)$ on **db**. It can be easily verified that $|q'(\mathbf{db})| = 1$, which is distinct from the upper bound 2.

Finally, we claim (without proof) that 2-DIMENSIONAL MATCHING (2DM) can be first-order reduced to computing $\mathbf{cqacnt}(\mathsf{body}(q), z)$. Therefore, since 2DM is NL-hard [13], $q(z)$ cannot admit parsimonious counting under standard complexity assumptions. ⌟

Proof of Theorem 26. Assume that $q(\vec{z})$ admits parsimonious counting. Then, $q(\vec{z})$ has a tuple $\vec{x}$ of bound variables such that for the query $q'(\vec{z}, \vec{x}) := \nexists \vec{x}\, [q]$, the conditions A, B, and C in Definition 8 are satisfied. From conditions A and B, it follows by Theorem 4 that $q(\vec{z})$ and $q'(\vec{z}, \vec{x})$ have acyclic attack graphs. By Lemma 29, condition II1 in Definition 12 is satisfied for $\vec{x}$. By Corollary 32, condition II2 in Definition 12 is satisfied by $\vec{x}$. By Lemma 28, the attack graph of $q'(\vec{z}, \vec{x})$ has no strong attack. By Lemma 33, it is now correct to conclude that the attack graph of $q(\vec{z})$ has no strong attack either, and thus condition I in Definition 12 is satisfied. Since we have shown that $q(\vec{z})$ satisfies all conditions in Definition 12, we conclude $q(\vec{z}) \in$ Cparsimony. ◀

8 Comparison with the Class Cforest

In this section, we introduce Cforest and show Cforest $\subsetneq$ Cparsimony without making use of Theorem 1. Theorem 1 then follows by Theorem 19.

▶ **Definition 35** (Cforest). *Let $q(\vec{z})$ be a self-join-free conjunctive query. The* Fuxman graph *of q is a directed graph whose vertices are the atoms of q. There is a directed edge from an atom R to an atom S if $R \neq S$ and* notKey(R) *contains a bound variable that also occurs in S. The class* Cforest *contains all (and only) self-join free conjunctive queries $q(\vec{z})$ whose Fuxman graph is a directed forest satisfying, for every directed edge from R to S,* Key$(S) \setminus$ free$(q) \subseteq$ notKey(R).

▶ **Theorem 36.** Cforest *is a strict subset of* Cparsimony*.*

9 Conclusion and Future Work

In his PhD thesis, Fuxman [18] defined a syntactically restricted class of self-join-free conjunctive queries, called Cforest, and showed that for every query in Cforest, consistent answers are first-order computable, and range-consistent answers are computable in first-order logic followed by a simple aggregation step. Our notion of "parsimonious counting" captures the latter computation for counting. Later, Koutris and Wijsen [29] syntactically characterized the class of *all* self-join-free conjunctive queries with a consistent first-order rewriting, which strictly includes Cforest. However, it remained an open problem to syntactically characterize the class of *all* self-join-free conjunctive queries that admit parsimonious counting. In this paper, we determined the latter class, named it Cparsimony, and showed that it strictly includes Cforest.

We now list some open problems for future research. In Definition 8 of parsimonious counting, we required that $q(\vec{z})$ has a consistent first-order rewriting. It is known [32] that there are self-join-free conjunctive queries, without consistent first-order rewriting, that have a consistent rewriting in Datalog. We could relax Definition 8 by requiring the existence of a consistent rewriting in Datalog, rather than in first-order logic. It is an open question to syntactically characterize the self-join-free conjunctive queries that admit parsimonious counting under such a relaxed definition.

Another open question is to characterize the complexity of $\mathbf{cqacnt}(q(\vec{z},\vec{w}),\vec{z})$ for every full self-join-free conjunctive query q and choice of free variables $\vec{z}$. It is easily verified that the complexity of computing the answers to $\mathbf{cqacnt}(q(\vec{z},\vec{w}),\vec{z})$ is higher than computing the consistent answers to $q'(\vec{z}) := \exists\vec{w}\,[q]$ (because of the lower bound in range semantics). It remains an open question to characterize this complexity if $q'(\vec{z})$ is not in Cparsimony, even if it has a consistent first-order rewriting.

The notion of parsimonious counting does not require conjunctive queries to be self-join-free. An ambitious open problem is to syntactically characterize the class of all (i.e., not necessarily self-join-free) conjunctive queries that admit parsimonious counting. This problem is largely open, because it is already a notorious open problem to syntactically characterize the class of conjunctive queries that have a consistent first-order rewriting.

Another open question is to extend the results in the current paper to other aggregation operators than COUNT, including MAX, MIN, SUM, and AVG.

References

1 Marcelo Arenas, Leopoldo E. Bertossi, and Jan Chomicki. Consistent query answers in inconsistent databases. In *PODS*, pages 68–79. ACM Press, 1999.

2 Marcelo Arenas, Leopoldo E. Bertossi, and Jan Chomicki. Scalar aggregation in FD-inconsistent databases. In *ICDT*, volume 1973 of *Lecture Notes in Computer Science*, pages 39–53. Springer, 2001.

3 Marcelo Arenas, Leopoldo E. Bertossi, Jan Chomicki, Xin He, Vijay Raghavan, and Jeremy P. Spinrad. Scalar aggregation in inconsistent databases. *Theor. Comput. Sci.*, 296(3):405–434, 2003.

4 Leopoldo E. Bertossi. *Database Repairing and Consistent Query Answering.* Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2011.

5 Leopoldo E. Bertossi. Database repairs and consistent query answering: Origins and further developments. In *PODS*, pages 48–58. ACM, 2019.

6 Leopoldo E. Bertossi, Loreto Bravo, Enrico Franconi, and Andrei Lopatenko. The complexity and approximation of fixing numerical attributes in databases under integrity constraints. *Inf. Syst.*, 33(4-5):407–434, 2008.

7 Andrei A. Bulatov. Complexity of conservative constraint satisfaction problems. *ACM Trans. Comput. Log.*, 12(4):24:1–24:66, 2011.

8 Marco Calautti, Marco Console, and Andreas Pieris. Counting database repairs under primary keys revisited. In *PODS*, pages 104–118. ACM, 2019.

9 Marco Calautti, Marco Console, and Andreas Pieris. Benchmarking approximate consistent query answering. In *PODS*, pages 233–246. ACM, 2021.

10 Marco Calautti, Leonid Libkin, and Andreas Pieris. An operational approach to consistent query answering. In *PODS*, pages 239–251. ACM, 2018.

11 Marco Calautti, Ester Livshits, Andreas Pieris, and Markus Schneider. Counting database repairs entailing a query: The case of functional dependencies. In *PODS*, pages 403–412. ACM, 2022.

12 Marco Calautti, Ester Livshits, Andreas Pieris, and Markus Schneider. Uniform operational consistent query answering. In *PODS*, pages 393–402. ACM, 2022.

13 Ashok K. Chandra, Larry J. Stockmeyer, and Uzi Vishkin. Constant depth reducibility. *SIAM J. Comput.*, 13(2):423–439, 1984.

14 Akhil A. Dixit and Phokion G. Kolaitis. A SAT-based system for consistent query answering. In *SAT*, volume 11628 of *Lecture Notes in Computer Science*, pages 117–135. Springer, 2019.

15 Akhil A. Dixit and Phokion G. Kolaitis. Consistent answers of aggregation queries via SAT. In *ICDE*, pages 924–937. IEEE, 2022.

16 Zhiwei Fan, Paraschos Koutris, Xiating Ouyang, and Jef Wijsen. LinCQA: Faster consistent query answering with linear time guarantees. *CoRR*, abs/2208.12339, 2022. arXiv:2208.12339.

17 Gaëlle Fontaine. Why is it hard to obtain a dichotomy for consistent query answering? *ACM Trans. Comput. Log.*, 16(1):7:1–7:24, 2015.

18 Ariel Fuxman. *Efficient query processing over inconsistent databases.* PhD thesis, University of Toronto, 2007.

19 Ariel Fuxman, Elham Fazli, and Renée J. Miller. ConQuer: Efficient management of inconsistent databases. In *SIGMOD Conference*, pages 155–166. ACM, 2005.

20 Ariel Fuxman, Diego Fuxman, and Renée J. Miller. ConQuer: A system for efficient querying over inconsistent databases. In *VLDB*, pages 1354–1357. ACM, 2005.

21 Ariel Fuxman and Renée J. Miller. First-order query rewriting for inconsistent databases. In *ICDT*, volume 3363 of *Lecture Notes in Computer Science*, pages 337–351. Springer, 2005.

22 M. R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman, 1979.

23 Miika Hannula and Jef Wijsen. A dichotomy in consistent query answering for primary keys and unary foreign keys. In *PODS*, pages 437–449. ACM, 2022.

24 Aziz Amezian El Khalfioui, Jonathan Joertz, Dorian Labeeuw, Gaëtan Staquet, and Jef Wijsen. Optimization of answer set programs for consistent query answering by means of first-order rewriting. In *CIKM*, pages 25–34. ACM, 2020.

25 Aziz Amezian El Khalfioui and Jef Wijsen. Consistent query answering for primary keys and conjunctive queries with counting. *CoRR*, abs/2211.04134, 2022. `arXiv:2211.04134`.

26 Benny Kimelfeld, Ester Livshits, and Liat Peterfreund. Counting and enumerating preferred database repairs. *Theor. Comput. Sci.*, 837:115–157, 2020.

27 Phokion G. Kolaitis, Enela Pema, and Wang-Chiew Tan. Efficient querying of inconsistent databases with binary integer programming. *Proc. VLDB Endow.*, 6(6):397–408, 2013.

28 Paraschos Koutris, Xiating Ouyang, and Jef Wijsen. Consistent query answering for primary keys on path queries. In *PODS*, pages 215–232. ACM, 2021.

29 Paraschos Koutris and Jef Wijsen. Consistent query answering for self-join-free conjunctive queries under primary key constraints. *ACM Trans. Database Syst.*, 42(2):9:1–9:45, 2017.

30 Paraschos Koutris and Jef Wijsen. Consistent query answering for primary keys and conjunctive queries with negated atoms. In *PODS*, pages 209–224. ACM, 2018.

31 Paraschos Koutris and Jef Wijsen. First-order rewritability in consistent query answering with respect to multiple keys. In *PODS*, pages 113–129. ACM, 2020.

32 Paraschos Koutris and Jef Wijsen. Consistent query answering for primary keys in datalog. *Theory Comput. Syst.*, 65(1):122–178, 2021.

33 Dany Maslowski and Jef Wijsen. A dichotomy in the complexity of counting database repairs. *J. Comput. Syst. Sci.*, 79(6):958–983, 2013.

34 Dany Maslowski and Jef Wijsen. Counting database repairs that satisfy conjunctive queries with self-joins. In *ICDT*, pages 155–164. OpenProceedings.org, 2014.

35 Slawek Staworko, Jan Chomicki, and Jerzy Marcinkowski. Prioritized repairing and consistent query answering in relational databases. *Ann. Math. Artif. Intell.*, 64(2-3):209–246, 2012.

36 Jef Wijsen. On the first-order expressibility of computing certain answers to conjunctive queries over uncertain databases. In *PODS*, pages 179–190. ACM, 2010.

37 Jef Wijsen. Foundations of query answering on inconsistent databases. *SIGMOD Rec.*, 48(3):6–16, 2019.

A Simple Algorithm for Consistent Query Answering Under Primary Keys

Diego Figueira
Univ. Bordeaux, CNRS, Bordeaux INP, LaBRI, UMR 5800, Talence, France

Anantha Padmanabha
DI ENS, ENS, CNRS, PSL University, Paris, France
Inria, Paris, France

Luc Segoufin
INRIA, ENS-Paris, PSL University, France

Cristina Sirangelo
Université Paris Cité, CNRS, Inria, IRIF, F-75013, Paris, France

Abstract

We consider the dichotomy conjecture for consistent query answering under primary key constraints. It states that, for every fixed Boolean conjunctive query q, testing whether q is certain (i.e. whether it evaluates to true over all repairs of a given inconsistent database) is either polynomial time or coNP-complete. This conjecture has been verified for self-join-free and path queries.

We propose a simple inflationary fixpoint algorithm for consistent query answering which, for a given database, naively computes a set Δ of subsets of database repairs with at most k facts, where k is the size of the query q. The algorithm runs in polynomial time and can be formally defined as:

1. Initialize Δ with all sets S of at most k facts such that $S \models q$.
2. Add any set S of at most k facts to Δ if there exists a block B (i.e., a maximal set of facts sharing the same key) such that for every fact $a \in B$ there is a set $S' \in \Delta$ contained in $S \cup \{a\}$.

The algorithm answers "q is certain" iff Δ eventually contains the empty set. The algorithm correctly computes certainty when the query q falls in the polynomial time cases of the known dichotomies for self-join-free queries and path queries. For arbitrary Boolean conjunctive queries, the algorithm is an under-approximation: the query is guaranteed to be certain if the algorithm claims so. However, there are polynomial time certain queries (with self-joins) which are not identified as such by the algorithm.

2012 ACM Subject Classification Theory of computation → Database query languages (principles)

Keywords and phrases consistent query answering, primary keys, conjunctive queries

Digital Object Identifier 10.4230/LIPIcs.ICDT.2023.24

Related Version *Full Version*: `https://arxiv.org/abs/2301.08482`

Funding Work supported by ANR QUID, grant ANR-18-CE40-0031.
Anantha Padmanabha: ANR-19-P3IA-0001 (PRAIRIE 3IA Institute).

1 Introduction

A database often comes with integrity constraints. The constraints are helpful in many ways, for instance in order to help optimizing query evaluation. When the database violates its integrity constraints we are faced with several possibilities. A first possibility is to clean the data until all integrity constraints are satisfied. This task is not easy as it is inherently non-deterministic: there could be many equally good ways to "repair" a database. A repair can be understood as a minimal way to change the database in order to satisfy the constraints.

Another possibility is to keep the database in its inconsistent state, postponing the problem until a query is asked to the database. In order to evaluate the query, the classical solution is to consider all possible repairs of the database and to output all the answers on the database D which are "certain", i.e., those answers that are in the output of the query when evaluated on *every* repair of D [2]. However, this method usually has an impact on the complexity of the query evaluation problem. The impact will of course depend on the type of integrity constraints and on the definition of a repair, but most often the worst case complexity increases by a factor at least exponential in the size of the database, as there could be exponentially many ways to repair a database.

Depending on the integrity constraints, what is a "good" notion of repair may be controversial. In this paper we consider primary key constraints, which are arguably the most common kind of integrity constraints in databases. For primary keys, there is a unanimously accepted notion of repair. Primary key constraints identify, for each relation, a set of attributes which are considered to be the relation *key*. An inconsistent database is therefore a database that has distinct tuples within a relation sharing the same key. For such constraints, the standard notion of a repair is any maximal subset of the database satisfying all the primary key constraints. This amounts to keeping exactly one among all tuples having the same key in each relation. A simple analysis shows that there can be exponentially many repairs for a given database, and therefore a naive evaluation algorithm would have to evaluate the query on each of these exponentially many repairs.

We consider Boolean conjunctive queries which can be evaluated efficiently over all databases, in polynomial time in data complexity. With the certain answer semantics described above, a query is certain on an inconsistent database if it is true on all its repairs. The data complexity of certain answers for conjunctive queries over inconsistent databases in the presence of primary key constraints is therefore in coNP since, in order to test whether the query is not certain, it is enough to guess a subset of the database which is a repair and which makes the query false. Further, it has been observed that for some conjunctive queries the certain answering problem is coNP-hard [4] while, for other queries, the certain answering problem can be solved in polynomial time. The main conjecture for inconsistent databases in the presence of primary keys is that there are no intermediate cases: for any conjunctive query, the certain answering problem is either solvable in polynomial time or is complete for coNP.

The conjecture has been proved for self-join-free Boolean conjunctive queries [8]. These are queries on which there are no two atoms using the same relation. It has been also proved for path queries [7]. However, the conjecture remains open for arbitrary conjunctive queries (with self-joins).

In this paper we revisit the two cases above where the conjecture is known to hold: self-join-free queries and path queries.

Contributions. Our main contribution is the design of a simple fixpoint algorithm for computing certain answers of queries over inconsistent databases in the presence of primary key constraints. For every $k \geq 1$, we describe a fixpoint algorithm parameterized by k. The algorithm is always an under-approximation of the certain answers: on Boolean queries, when it outputs "yes" then the query is certain, i.e., it is true on all repairs of the database. But there could false negatives: queries which are certain on which the algorithm outputs "no".

Our main result shows that for all self-join-free queries and path queries whose certain answering problem is computable in polynomial time there is a number k (namely, the number of atoms of the query) such that this simple algorithm correctly computes the certain answer. In other words, if for all k our algorithm gives a false negative answer for a self-join-free or path query it is because the query has a coNP-complete certain answering problem.

A natural question is then to wonder whether our algorithm always correctly computes the certain answering problem on all queries for which this problem is polynomial time computable. We answer negatively to this question, by exhibiting a conjunctive query (with self-joins) whose certain answers can be solved in polynomial time, but for all k the algorithm fails to give a correct answer.

Though our greedy fixpoint computation algorithm is simple, the proof of correctness is not. In the case of self-join-free queries, we provide a semantic condition and show that when the condition holds, the fixpoint gives always the correct answer, by setting the parameter k to be the number of atoms of the query. The proof is by contradiction: if the algorithm fails to give the correct answer, we use the fixpoint definition of the algorithm in order to produce (chase) an infinite sequence of distinct facts of the database, contradicting its finiteness. When the semantic condition does not hold, we show that it implies the condition of [8] characterizing those queries having a coNP-complete certain answers problem.

The situation is simpler for the case of path queries, as we show that for a suitable k (again the number of atoms of the query), our fixpoint algorithm can simulate the polynomial time algorithm of [7] for computing certain answers for q, assuming that certain answering for q is polynomial time solvable.

Related work. Our work is very much inspired by the results of Koutris and Wijsen [9, 8]. For self-join-free queries, the authors prove the polynomial-time case via a long sequence of reductions eventually producing a simple query whose certain answers can be solved efficiently. When unfolding the sequence of reductions this gives a complicated polynomial time algorithm with a complex proof of correctness. We have basically simplified the algorithm and pushed all the difficulty into the proof of correctness. Our algorithm is simple, but the proof of correctness is arguably as complex as theirs. Further, our algorithm does not give, a priori, the optimal LogSpace complexity result of [9] as we know that some of the path queries that can be solved with our algorithm are PTime complete [7]. The semantic condition that we provide for characterizing the polynomial case in the self-join-free case can be effectively tested, but not efficiently, unlike the simple syntactic characterization of [8] based on the so-called “attack graph” of the query.

In the case of path queries, [7] also provides a simple fixpoint algorithm for solving the polynomial cases. Though it seems that their algorithm is different in spirit from ours, the two algorithms have some similarities that we use in order to “simulate” their fixpoint computation using ours.

All missing details can be found in the appendix of the long version of this paper [3].

2 Preliminaries

A **database** is a finite relational structure. A **relational signature** is a finite set of relation symbols associated with an arity. A **finite relational structure** D over a relational signature σ is composed of: a finite set, the domain of D, and a function associating to each symbol R of σ a relation $R(D)$ of the appropriate arity over the domain of D.

An R**-fact** of a database D over a signature σ is a term of the form $R(\bar{a})$ where R is a symbol of σ and $\bar{a}$ a tuple in $R(D)$. A **fact** is an R-fact for some R, R is then the symbol associated to the fact and $\bar{a}$ the tuple associated to the fact. A database can then be viewed as a finite collection of facts. By the **size of a database** we mean the number of facts it contains. Assuming σ is fixed, which we will implicitly do in this paper, this is equivalent to the usual notion of size for a database, up to some polynomial function.

A **primary key constraint** over a signature σ is a special case of a functional dependency designating for a relation symbol R of σ a certain set of indices (columns) of R as a primary key. A database satisfies the primary key constraint if for every relation R over σ, whenever two R-facts agree on the key indices they must be equal. In a set of primary key constraints, each relation of σ has a unique primary key constraint. As all the sets of constraints we consider are primary key constraints we will henceforth omit the "primary" prefix. We use the letter Γ to denote the corresponding set of key constraints.

Given two facts u and v and a set Γ of key constraints, we say that u and v are Γ-equivalent, denoted by $u \sim_\Gamma v$, if u and v have the same associated symbol R and agree on the key of R as specified by Γ. Γ-equivalence is an equivalence relation and the equivalence classes are called **Γ-blocks**. We will omit Γ in our notations whenever it is clear from the context. A database is then a finite collection of blocks, each block being a finite collection of equivalent facts. When writing a query q we will always underline in an atom $R(\bar{x})$ the positions that are part of the key of R as specified by Γ. This will avoid describing explicitly Γ. For instance $R(\underline{x}\ y)$ says that the position of the variable x (i.e., the first position) is the key for the binary relational symbol R; and $R'(\underline{yz}\ x)$ says that the positions of the variables y and z form the key for the ternary relational symbol R'.

If a database D satisfies the key constraints Γ, denoted by $D \models \Gamma$, then each block of D has size one. If not, then a **repair** of D is a subset of the facts of D such that each block of D has exactly one representative in the repair. In particular a repair always satisfies the key constraints. Notice that there could be exponentially many repairs of a given database D.

In this paper a query is a Boolean conjunctive query. We view a query over a relational signature σ as a collection of atoms where an atom is a term $R(\bar{x})$ where R is a relation symbol and $\bar{x}$ is a tuple of variables of the appropriate arity. The query being Boolean, all variables are implicitly existentially quantified. We will consider atoms of a conjunctive query to be ordered in an arbitrary but fixed order. A database D satisfies a query q having atoms $A_1, \ldots, A_k$, denoted by $D \models q$, if there is a mapping μ from the variables of q to the elements of the domain of D such that the fact $\mu(A_i) \in D$ for all i. In this case the sequence $(\mu(A_1), \ldots, \mu(A_k))$ of (not necessarily distinct) facts of D is called a **solution** to q in D. Different mappings yield different solutions. The set of solutions to q in D is denoted by $q(D)$. We will also write $D \models q(\bar{u})$ to denote that the sequence of facts $\bar{u}$ is a solution to q in D. If $\bar{u} = (u_1, \ldots, u_k)$ is a solution to q we also say that u_i **matches** A_i in this solution, and that any subsequence $u_{i_1}, \ldots, u_{i_l}$ matches $A_{i_1}, \ldots, A_{i_l}$.

We say that a query q is **certain** for a database D if all repairs of D satisfy q. We study the complexity of determining whether a query is certain for a database D. We adopt the *data complexity* point of view. For each query q and set of key constraints Γ, we denote by $\mathbf{certain}_\Gamma(q)$ (or simply $\textsc{certain}(q)$ when Γ is understood from the context) the problem of determining, given a database D, whether q is certain for D. Clearly the problem is in coNP as one can guess a (polynomial sized) repair and test whether it does not satisfy q. It is known that for some queries q the problem $\textsc{certain}(q)$ is coNP-complete [4]. However, there are queries q for which $\textsc{certain}(q)$ is in PTime or even expressible in first-order logic (denoted by FO in the sequel) [6, 10].

The following dichotomy has been conjectured (cf [4, 1]):

▶ **Conjecture 1** (Dichotomy conjecture). *For each query q, the problem* $\textsc{certain}(q)$ *is either in* PTime *or* coNP*-complete.*

The conjecture has been proved in the case of self-join-free queries [8] and of path queries [7]; however, it remains open in the general case. A Boolean conjunctive query is **self-join-free** if all its atoms involve different relational symbols. A **path query** is a

Boolean conjunctive query with $n+1$ distinct variables $x_0, x_1, \cdots x_n$ and n atoms $A_1 \cdots A_n$ such that each $A_i = R_i(\underline{x_{i-1}}, x_i)$ for some symbol R_i of σ of arity two. The query may contain self-joins, i.e. $R_i = R_j$ for some $i \neq j$.

▶ **Example 2.** Consider the following example queries taken from [6, 7]. For the self-join-free query $q_1 : R_1(\underline{x}\ y) \wedge R_2(\underline{y}\ z)$ (recall that all variables are implicitly existentially quantified), it is easy to see that the problem CERTAIN(q_1) can be solved in polynomial time [6]. Actually, it can be expressed by the first-order formula $\exists xyz\ R_1(xy) \wedge R_2(yz) \wedge \forall y'(R_1(xy') \rightarrow \exists z' R_2(y'z'))$.

For the self-join-free query $q_2 : R_1(\underline{x}\ y) \wedge R_2(\underline{y}\ x)$ and the path query $q_2' : R(\underline{x_1}\ x_2) \wedge X(\underline{x_2}\ x_3) \wedge R(\underline{x_3}\ x_4) \wedge Y(\underline{x_4}\ x_5) \wedge R(\underline{x_5}\ x_6) \wedge Y(\underline{x_6}\ x_7)$, it has been shown, in [10] and [7] respectively, that CERTAIN(q_2) and CERTAIN(q_2') can be solved in polynomial time but cannot be expressed in first-order logic. Our algorithm works for q_1, q_2 and q_2'.

Finally, for the self-join-free query $q_3 : R_1(\underline{x}\ y) \wedge R_2(\underline{z}\ y)$ and the path query q_3' : $R(\underline{x_1}\ x_2) \wedge X(\underline{x_2}\ x_3) \wedge R(\underline{x_3}\ x_4) \wedge X(\underline{x_4}\ x_5) \wedge R(\underline{x_5}\ x_6) \wedge Y(\underline{x_6}\ x_7) \wedge R(\underline{x_7}\ x_8) \wedge Y(\underline{x_8}\ x_9)$, both CERTAIN$(q_3)$ and CERTAIN(q_3') are known to be coNP-complete [4, 7].

3 Polynomial-time algorithm

To solve CERTAIN(q), we describe a family of algorithms Cert$_k(q)$, where $k \geq 1$ is a parameter. For a fixed k and query q, Cert$_k(q)$ takes a database as input and runs in time polynomial in the size of the database, in such a way that Cert$_k(q)$ is always an under-approximation of CERTAIN(q), i.e., whenever Cert$_k(q)$ says "yes" then q is certain for the input database. However, Cert$_k(q)$ could give false negative answers.

In Section 4 we will show that for self-join-free queries either Cert$_k(q)$ computes CERTAIN(q) (where k is the number of atoms occurring in q) or CERTAIN(q) is complete for coNP. In Section 5 we show an analogous result for path queries.

The algorithm inductively computes sets of facts maintaining the invariant that every repair containing one of these sets makes the query true. The algorithm returns "yes" if the empty set is eventually derived (since all repairs contain the empty set).

We now describe the algorithm. Assume q, Γ and k are fixed. Let D be a database. A k-set over D is a set S of facts of D of size at most k such that no two elements of S are Γ-equivalent. In other words a k-set is a subset of a repair of D of size at most k. We denote by Cert$_k(q)$ the following algorithm. On a database input D, the algorithm Cert$_k(q)$ computes inductively a set $\Delta_k(q, D)$ of k-sets over D while maintaining the invariant:

If $S \in \Delta_k(q, D)$ and r is a repair of D containing S, then $r \models q$. (INV)

Initially $\Delta_k(q, D)$ contains all k-sets S such that $S \models q$. In other words, we start with all solutions to q in all repairs of D. Clearly, this satisfies the invariant (INV). Now we iteratively add a k-set S to $\Delta_k(q, D)$ if there exists a block B of D such that for every fact $u \in B$ there exists $S' \subseteq S \cup \{u\}$ such that $S' \in \Delta_k(q, D)$. Again, it is immediate to verify that the invariant (INV) is maintained.

This is an inflationary fixpoint algorithm and notice that the initial and inductive steps can be expressed in FO. If n is the number of facts of D, the fixpoint is reached in at most n^k steps. In the end, Cert$_k(q)$ returns "yes" iff the empty set belongs to $\Delta_k(q, D)$. Equivalently, Cert$_k(q)$ returns "yes" if there is a block B of D such that for all facts u of B the set $\{u\}$ belongs to $\Delta_k(q, D)$. We write $D \models$ **Cert**$_k(q)$ to denote the fact that Cert$_k(q)$ returns "true" upon input D. Altogether we have shown:

▶ **Proposition 3.** *For all q, Γ, k, Cert$_k(q)$ runs in time polynomial in the size of its input database D and, whenever $D \models$ Cert$_k(q)$ then q is certain for D.*

In order to simplify the notations, as we will mostly consider this case, we write $\Delta(q, D)$ and $\mathrm{Cert}(q)$ to denote $\Delta_k(q, D)$ and $\mathrm{Cert}_k(q)$ respectively, where k is the number of atoms of q. Also, for a fact u, we write $u \in \Delta_k(q, D)$ instead of $\{u\} \in \Delta_k(q, D)$.

▶ **Example 4.** Consider again the query $q_2 : R_1(\underline{x}\ y) \wedge R_2(\underline{y}\ x)$ from Example 2. Let $k = 2$ and consider the execution of $\mathrm{Cert}_2(q_2)$. Initially, $\Delta_2(q_2, D)$ contains all pairs of facts $\{R_1(ab), R_2(ba)\}$ such that both $R_1(ab)$ and $R_2(ba)$ are in D. The first iterative step adds to $\Delta_2(q_2, D)$ (i) all singletons $\{R_1(ab)\}$ such that $R_2(ba)$ is a fact of D whose block contains only $R_2(ba)$, and (ii) analogously all $\{R_2(ab)\}$ such that the block of $R_1(ba)$ is a singleton. And so it continues.

We show that $\mathrm{Cert}_2(q_2)$ computes CERTAIN(q_2). In other words, for q_2, $\mathrm{Cert}_2(q_2)$ is not an *under* approximation but an *exact* computation of certainty.

Observe that, for every repair r and fact α therein, there is at most one other fact α' in r such that $\{\alpha, \alpha'\} \models q_2$. This is because in any repair the first atom of q_2 determines the second atom and vice-versa. This "mutual determinacy" is, in fact, what makes $\mathrm{Cert}_2(q_2)$ a complete procedure, as we shall see next.

In view of Proposition 3, it remains to show that if q_2 is certain for D then $\Delta_2(q_2, D)$ contains the empty set.

Let r be a repair of D. By $|q(r)|$ we denote the number of solutions to q in r, i.e., the cardinality of $q(r)$. We say that a repair r is **minimal** if there is no repair s such that $|q(s)| < |q(r)|$. We prove the following claim.

▷ Claim. If r is a minimal repair and $R_1(ab)$, $R_2(ba)$ are facts of r then $R_1(ab) \in \Delta_2(q_2, D)$.

Towards a contradiction, assume that $R_1(ab) \notin \Delta_2(q_2, D)$. We shall construct an infinite sequence $u_1, u_2, \ldots$ of distinct facts of D, contradicting the finiteness of D. We construct, at the same time, an infinite sequence $v_1, v_2, \ldots$ of facts of D and an infinite sequence of minimal repairs $r_1, r_2, \ldots$ maintaining the following invariant:

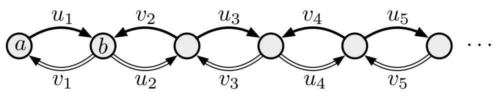
1. the u_i's are pairwise distinct;
2. $u_i \notin \Delta_2(q_2, D)$;
3. if $u_i = R_1(cd)$ then $v_i = R_2(dc)$ and if $u_i = R_2(cd)$ then $v_i = R_1(dc)$;
4. $u_{i+1} \sim v_i$ and $u_{i+1} \neq v_i$;
5. r_i is minimal and contain each u_j, $j \leq i$ together with v_i.

Initially $r_1 = r$, $u_1 = R_1(ab)$ and $v_1 = R_2(ba)$ and the invariant conditions are met, the second item being our initial hypothesis.

Consider step i. Consider the block B_i of v_i. As $u_i \notin \Delta_2(q_2, D)$ we know that B_i must contain an element u_{i+1} such that both $u_{i+1} \notin \Delta_2(q_2, D)$ and $\{u_i, u_{i+1}\} \notin \Delta_2(q_2, D)$. In particular $u_{i+1} \sim v_i$ but $u_{i+1} \neq v_i$ and items 2 and 4 of our inductive hypothesis are met. Towards the first item of our inductive hypothesis, if $u_{i+1} = u_j$ then by item 5 the repair r_i would contain two equivalent facts, v_i and $u_{i+1} = u_j$, which is not possible since we have already established that $u_{i+1} \neq v_i$.

Consider the repair r_{i+1} resulting from replacing v_i with u_{i+1}. Let v_{i+1} be the dual fact of u_{i+1} as required by the third item of the invariant. As $u_i v_i$ forms a solution to q in r_i and r_i is minimal, we must have $v_{i+1} \in r_{i+1}$. Finally notice that r_{i+1} is minimal as its solutions to q are exactly the same as for r_i except for $u_i v_i$ that has been removed and $u_{i+1} v_{i+1}$ that has been added (by the mutual determinacy of the atoms of q_2).

Here is a depiction of how the u_i's and v_i's are defined, where the full and hollow arrows correspond to R_1 and R_2 respectively.



This concludes the construction of the infinite sequence, showing that $R_1(ab) \in \Delta_2(q_2, D)$ for any minimal repair containing both $R_1(ab)$ and $R_2(ba)$.

To conclude that $\mathrm{Cert}_2(q_2)$ returns the correct answer, consider a minimal repair r of D. As q_2 is certain for D, we must have $r \models q$ and this is witnessed by two facts $R_1(ab)$ and $R_2(ba)$ of r. Let B be the block of $R_1(ab)$. Let us show that all facts of B are in $\Delta_2(q_2, D)$ and hence $\emptyset \in \Delta_2(q_2, D)$. Let $R_1(ab')$ be such a fact and consider the repair r' obtained by replacing $R_1(ab)$ with $R_1(ab')$. As r is minimal it follows immediately that r' is minimal and must contain $R_2(b'a)$ (again, this is ensured by the mutual determinacy of q_2). From the claim it follows that $R_1(ab') \in \Delta_2(q_2, D)$, as desired.

Observe that Cert_k does not always compute the certain answers. For instance, the query q_3 from Example 2 is so that $\textsc{certain}(q_3)$ is coNP-complete, and hence $\mathrm{Cert}_k(q_3)$ must have false negatives for all k, assuming coNP $\neq$ PTime (proving this without relying on complexity theoretic assumptions remains plausible, and would not impact our results).

4 Self-join-free queries

In this section we consider the case of self-join-free queries. We exhibit a condition named PCond (for Polynomial-time Condition) and show that any self-join-free query q satisfying PCond is such that $\mathrm{Cert}(q)$ computes $\textsc{certain}(q)$. When PCond fails, we show that $\textsc{certain}(q)$ is coNP-hard.

We start by defining PCond, which will require some extra definitions. Fix, for the rest of this section, a set Γ of primary key constraints. Let D be a database and r a repair of D. For a fact u of r, and for an equivalent fact $v \sim u$ from D, we denote by $r[u \to v]$ the repair obtained from r by replacing the fact u with v.

Consider a self-join-free query q with k atoms. Recall that we write $D \models q(\bar{u})$ when $\bar{u}$ is a solution to q in D. As q is self-join-free, for each fact a in a solution $\bar{u}$ there is a unique atom of q that a can match, namely the only fact of q having the same relation symbol as a. Hence, the order on $\bar{u}$ and on the atoms of q is not relevant. With some abuse of notation we will therefore often treat a solution $\bar{u}$, or the sequence of atoms of q, as a *set* rather than a *sequence*; we will often use different orders among the facts of a same solution, placing up front the most relevant facts. In particular we shall write, for a tuple $\bar{u}$ of facts, $\bar{u} \in \Delta(q, D)$ to denote that the k-set formed by the facts of $\bar{u}$ belongs to $\Delta(q, D)$.

Let A be an atom of q whose associated symbol is R. We denote by ***vars***(A) the set of variables of A and by ***key***(A) the set of variables of A occurring in a position belonging to the primary key of R. For instance $key(R(\underline{x}\, y))$ is $\{x\}$, $key(R'(\underline{yz}\, x))$ is $\{y, z\}$ and $key(R''(\underline{xzx}\, y))$ is $\{x, z\}$.

Given a set X of variables of q and a sequence $A_1 \dots A_n$ of atoms of q, we say that $X\ A_1 \dots A_n$ is a **Γ-derivation** from X to A_n in q if for each $1 \leq i \leq n$ we have that

$$key(A_i) \subseteq X \cup \bigcup_{1 \leq j < i} vars(A_j).$$

If $X = vars(A_0)$, for some atom A_0 of q, we also say that the Γ-derivation is from A_0 to A_n, and we also write it as $A_0 A_1 \dots A_n$. We say that an atom A' is **Γ-determined** by the atom A if there exists a Γ-derivation from A to A'. Moreover, A and A' are **mutually Γ-determined** if A' is Γ-determined by A and A is Γ-determined by A'. This is an equivalence relation among atoms. A set S of atoms of q is said ***stable*** if each pair of facts of S are mutually Γ-determined. Note that a stable set is not necessarily an equivalence class of Γ-determinacy, it may also be a subset of it. As usual, we will omit Γ when it is clear from the context. The key property relating Γ-determinacy and query solutions is given by the lemma:

▶ **Lemma 5.** *Let q be a self-join-free query. Let D be a database instance and r, r' be two repairs of D. Let $A_1 \dots A_n$ be a Γ-derivation in q from atom A_1 to A_n. Let $r \models q(\bar{\alpha} a_1 \dots a_n \bar{\beta})$, and $r' \models q(\bar{\alpha}' a'_1 \dots a'_n \bar{\beta}')$ where for each i, a_i and a'_i match A_i and $\bar{\alpha}\bar{\beta}$ and $\bar{\alpha}'\bar{\beta}'$ match the rest of the atoms of q. If $a_1 = a'_1$ and $a_i \in r'$ for each $i < n$, then 1) $a_i = a'_i$ for each $i < n$ and 2) $a_n \sim a'_n$ (and therefore $a_n = a'_n$ if moreover $a_n \in r'$).*

We are now ready to define PCond. A **Γ-sequence** τ of q is a sequence $\tau = S_1 S_2 \cdots S_n$ where each S_i is a stable set of atoms of q, and the S_i's form a partition of q. In this context, we denote by $S_{\leq i}$ the set $\bigcup_{j \leq i} S_j$.

Let $\tau = S_1 S_2 \cdots S_n$ be a Γ-sequence of q. Let $1 \leq i < n$ and let A be an atom of S_{i+1}. We say that the query q satisfies $\mathbf{PCond}_\tau(A)$ and write $q \models \mathrm{PCond}_\tau(A)$ if the following is true for all databases D, all repairs r of D and all solutions $\bar{\alpha} u \bar{\beta}$ and $\bar{\alpha}' u' \bar{\beta}'$ to q in D such that $\bar{\alpha}$ and $\bar{\alpha}'$ match $S_{\leq i}$ and u and u' match A:

$$\text{If } \left\{\begin{array}{l} r \models q(\bar{\alpha} u \bar{\beta}), \\ u \sim u', \text{ and} \\ r[u \to u'] \models q(\bar{\alpha}' u' \bar{\beta}') \end{array}\right\} \text{ then } \left\{\begin{array}{l} r \models q(\bar{\alpha}' u \bar{\delta}) \text{ and} \\ r[u \to u'] \models q(\bar{\alpha} u' \bar{\delta}') \end{array}\right\} \text{ for some sequences } \bar{\delta} \text{ and } \bar{\delta}'.$$

We write $q \models \mathbf{PCond}_\tau(i)$ if q satisfies $\mathrm{PCond}_\tau(A)$ for all A of S_{i+1}, and we write $q \models \mathbf{PCond}_\tau$ if q satisfies $\mathrm{PCond}_\tau(i)$ for all $1 \leq i < n$. Since the condition is restricted to indices $i < n$, PCond_τ trivially holds for any τ having only one stable set. Finally, we write $q \models \mathbf{PCond}$ if there is a Γ-sequence τ of q such that $q \models \mathrm{PCond}_\tau$. Again, if q has only one Γ-determinacy class (for instance the query q_2 of Example 4) then $q \models \mathrm{PCond}$ in a trivial way.

Our goal is to show that $q \models \mathrm{PCond}$ implies that $\mathrm{Cert}(q)$ computes $\textsc{certain}(q)$. This is the main technical result of this section and is proved by Theorem 7. In Theorem 13 we will conclude the self-join-free case by showing that when PCond fails, then the certainty of the query is hard. Before we prove those results, some examples are in order.

▶ **Example 6.** We recall the three queries from Example 2. The query $q_2 = R_1(\underline{x}\ y) \wedge R_2(\underline{y}\ x)$ satisfies PCond since it has only one maximal stable set.

The query $q_1 = R_1(\underline{x}\ y) \wedge R_2(\underline{y}\ z)$ has two stable sets: $R_1(\underline{x}\ y)$ determines $R_2(\underline{y}\ z)$ but the converse is false. For $\tau = \{R_2(\underline{y}\ z)\}\{R_1(\underline{x}\ y)\}$ we have $q \not\models \mathrm{PCond}_\tau$ because we have $q_1(R_2(bc)R_1(ab))$ and $q_1(R_2(b'c)R_1(ab'))$ but not $q_1(R_2(bc)R_1(ab'))$. However for $\tau = \{R_1(\underline{x}\ y)\}\{R_2(\underline{y}\ z)\}$ it is easy to verify that $q_1 \models \mathrm{PCond}_\tau$. Hence, $q_1 \models \mathrm{PCond}$.

The query $q_3 = R_1(\underline{x}\ y) \wedge R_2(\underline{z}\ y)$ has also two stable sets, but no possible sequence τ makes PCond_τ true. This is because (i) $q_3(R_1(ab)\ R_2(cb))$ and $q_3(R_1(a'b')\ R_2(cb'))$ hold, but not $q_3(R_1(ab)\ R_2(cb'))$, and (ii) $q_3(R_2(ab)\ R_1(cb))$ and $q_3(R_2(a'b')\ R_1(cb'))$ hold, but not $q_3(R_2(ab)\ R_1(cb'))$. Therefore, $q_3 \not\models \mathrm{PCond}$.

▶ **Theorem 7.** *Let q be a self-join-free query. If $q \models \mathrm{PCond}$, then $\mathrm{Cert}(q)$ computes $\textsc{certain}(q)$.*

Suppose q has k atoms. Let $\tau = S_1 \cdots S_n$ be a Γ-sequence of q such that $q \models \mathrm{PCond}_\tau$. We need to show that $\mathrm{Cert}(q) = \mathrm{Cert}_k(q)$ computes precisely $\textsc{certain}(q)$.

We start with some extra notations. Recall that $q(r)$ denotes the set of solutions to q in a repair r; we additionally denote by $q_{\leq i}(r)$ the projection of $q(r)$ on the first i sets of τ. More precisely

$$q_{\leq i}(r) = \{\bar{v} \mid \exists \bar{u} \in q(r) \text{ s.t. } \bar{v} \text{ is the subset of } \bar{u} \text{ matching } S_{\leq i}\},$$

and if $\bar{v} \in q_{\leq i}(r)$ we write equivalently $r \models q_{\leq i}(\bar{v})$. Let D be a database and r a repair of D. We say that r is ***i*-minimal** if there is no repair r' such that $q_{\leq i}(r') \subsetneq q_{\leq i}(r)$. We say that a fact u of a database D is ***i*-compatible**, if it matches some atom of S_i. We will need the limit case when $i = 0$. In that case S_0 is the empty set, as well as $S_{\leq 0}$ (and hence $\text{PCond}_\tau(0)$ is always true), $q_{\leq 0}(r)$ contains only the empty sequence ε for all r, and therefore all repairs are 0-minimal. The proof of the theorem will make use of an induction based on the following invariant property of the database, for each $0 \leq i \leq n$:

$$\text{IND}_i = \text{For all } i\text{-minimal repair } s \text{ and facts } \bar{u} \text{ s.t. } s \models q_{\leq i}(\bar{u}), \text{ we have } \bar{u} \in \Delta(q, D).$$

▶ **Lemma 8.** *Given q, D and a Γ-sequence τ for q, for every $0 \leq i < n$, if IND_{i+1} and $\text{PCond}_\tau(i)$, then IND_i.*

We first show how this statement already implies Theorem 7.

Proof of Theorem 7. From Proposition 3, we know that if D is a database such that $D \models \text{Cert}(q)$ then all repairs of D satisfy q. It remains to show the converse.

Assume all repairs satisfy q and that $q \models \text{PCond}_\tau$ for some sequence τ of length n, which means that $\text{PCond}_\tau(i)$ holds for all i. Observe that IND_n holds true by the base case definition of $\Delta(q, D)$. Hence by n repeated applications of Lemma 8 we obtain that IND_0 holds true. Now take any repair r. By definition r is 0-minimal and by hypothesis it satisfies the query q. By IND_0 it follows that the empty set (denoted by the empty tuple) is in $\Delta(q, D)$, and hence $D \models \text{Cert}(q)$. ◀

We are now left with the proof of Lemma 8, which is the main technical content of the section. Towards this, we define a stronger version of i-minimality. For $1 \leq i < n$, we say that an i-minimal repair s is **strong** if there exists no repair s' such that $q_{\leq i}(s') = q_{\leq i}(s)$ and $|q_{\leq i+1}(s')| < |q_{\leq i+1}(s)|$. Note that a strong i-minimal repair is in particular $(i + 1)$-minimal.

▷ Claim 9. If there exists an i-minimal repair s such that $s \models q_{\leq i}(\bar{u})$, then there exists a strong i-minimal repair s' such that $s' \models q_{\leq i}(\bar{u})$.

For a given database D, for a repair r of D, we denote by $r_{|i+1}$ the set of facts of r which are not $(i + 1)$-compatible. A sequence $\bar{p}$ of facts of the database is **connected** with respect to $D' \subseteq D$ if for every repair r containing $\bar{p}$ and D', and for every two consecutive facts $a\, b$ of $\bar{p}$, if $r \models q(\bar{\alpha}, a, \bar{\beta})$ for some $\bar{\alpha}, \bar{\beta}$, then $b \in \bar{\alpha}\bar{\beta}$. Note that if $\bar{p}$ is the empty tuple (or a tuple of size 1), then $\bar{p}$ is trivially connected with respect to every $D' \subseteq D$.

Proof of Lemma 8. By contradiction, suppose the statement of the lemma is false. Then, there is some i such that IND_{i+1} and $\text{PCond}_\tau(i)$ holds, but for some i-minimal repair s and tuple $\bar{u}$ we have

$$s \models q_{\leq i}(\bar{u}) \text{ but } \bar{u} \notin \Delta(q, D). \tag{h1}$$

From Claim 9, we can assume that s is strong i-minimal. We will build an infinite sequence of pairwise distinct facts $p_1, p_2, \ldots$ from D, contradicting the finiteness of D. We also maintain another sequence of repairs $r_1, r_2, \ldots$. We set $r_0 = s$ and $\bar{p}_0$ to be the empty fact sequence. And for all $l > 0$, we define $\bar{p}_l = p_1, \ldots p_l$.

The sequence is constructed by induction with the following invariant for every $l \geq 0$, assuming $\bar{p} = \bar{p}_l$ and $r = r_l$:

(a) $\bar{p}$ contains only $(i+1)$-compatible facts of D;
(b) the elements of $\bar{p}$ are pairwise distinct;
(c) $\bar{p}$ is connected with respect to $r_{|i+1}$;
(d) r is strong i-minimal, $r \models q_{\leq i}(\bar{u})$ and, if $\bar{p}$ is not empty and v is the last fact of $\bar{p}$, then $r \models q(\bar{u}v\bar{\delta})$, for some $\bar{\delta}$;
(e) $\bar{u}\bar{c} \notin \Delta(q, D)$, where $\bar{c}$ is the maximal suffix of $\bar{p}$ satisfying $r' \models q(\bar{u}\bar{c}\bar{\beta})$ for some $\bar{\beta}$ and strong i-minimal repair r' containing $\bar{p}$ and $r_{|i+1}$.

Note that the invariant (b) leads to a contradiction when l is larger than the size of D.

Base case. When $l = 0$, we have $r_0 = s$ and $\bar{p}$ is the empty sequence. Hence (a), (b) and (c) are trivially true by emptiness of $\bar{p}$; (d) holds since $s \models q_{\leq i}(\bar{u})$ (note that we have assumed s to be strong i-minimal); finally (e) holds with empty $\bar{c}$ since $\bar{u} \notin \Delta(q, D)$ by (h1).

Inductive step. Assume we have $r = r_{l-1}$ and $\bar{p} = p_1, \ldots p_{l-1}$ (possibly empty) satisfying the five properties above. Consider the maximal suffix $\bar{c}$ concerned by property (e). That is, for some $\bar{\beta}$ and strong i-minimal repair r' containing $\bar{p}$ and $r_{|i+1}$ we have:

$$\bar{u}\bar{c} \notin \Delta(q, D) \text{ and } r' \models q(\bar{u}\bar{c}\bar{\beta}) \tag{h2}$$

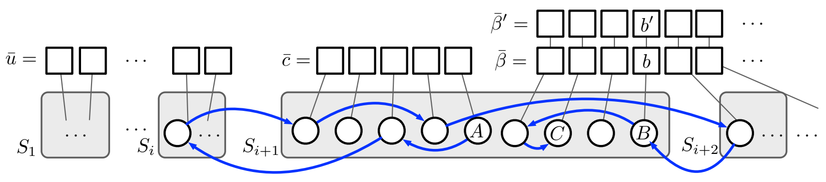
First let $\bar{\beta} = d_1 d_2, \ldots d_t$. Since $\bar{u}\bar{c} \notin \Delta(q, D)$, by definition of $\Delta(q, D)$ there exists some $d_1' \sim d_1$ such that $\bar{u}\bar{c}d_1' \notin \Delta(q, D)$. This again implies that there exists some $d_2' \sim d_2$ such that $\bar{u}\bar{c}d_1'd_2' \notin \Delta(q, D)$. Since k is the number of atoms in q, we can continue this to obtain $\bar{\beta}' = d_1'd_2', \ldots d_t'$ where $d_i' \sim d_i$ but $\bar{u}\bar{c}\beta'$ contains no k-set in $\Delta(q, D)$. Also note that $\bar{\beta}'$ matches all atoms of q not already matched by $\bar{u}\bar{c}$.

Further, we show that $\bar{c}$ cannot match the entire set S_{i+1}. Suppose, by means of contradiction, that $r' \models q_{\leq i+1}(\bar{u}\bar{c})$. As r' is strong i-minimal, it is $(i+1)$-minimal. Hence, since $\textsc{Ind}_{i+1}$ holds by hypothesis, $\bar{u}\bar{c} \in \Delta(q, D)$, which is in contradiction with (h2). Then,

$$r' \not\models q_{\leq i+1}(\bar{u}\bar{c}). \tag{h3}$$

This means that, since $r' \models q(\bar{u}\bar{c}\bar{\beta})$ by (h2), and $\bar{c}$ matches a subset of S_{i+1}, there must be an atom C of S_{i+1} that is not matched by any fact of $\bar{c}$. Consider the atom A of S_{i+1} matching the last element of $\bar{c}$. If instead $\bar{c}$ is empty, choose A as an arbitrary atom of S_{i+1}.

Since A and C are both in S_{i+1}, which is stable, there exists a Γ-derivation σ from A to C. (Notice that σ may contain atoms outside S_{i+1}.) Consider the first atom B of σ which is in S_{i+1} and which is not matched by any fact of $\bar{c}$. The following depiction may help to see the situation:



In the picture directed edges connecting atoms of the query represent the successor relation in the Γ-derivation from A to C.

Let b be the fact of $\bar{\beta}$ matching B and $b' \sim b$ be the fact matching B in $\bar{\beta}'$. We show that

$$b \notin \bar{p}. \tag{h4}$$

Suppose b is in $\bar{p}$, by connectedness of $\bar{p}$ with respect to $r_{|i+1}$, this implies that the suffix $\bar{b}$ of $\bar{p}$ starting with b is part of the solution $\bar{u}\bar{c}\bar{\beta}$, that is, $r' \models q(\bar{u}\bar{b}\bar{\gamma})$ for some $\bar{\gamma}$. By construction, b is not in $\bar{c}$, thus it must occur before $\bar{c}$ in $\bar{p}$ and hence $\bar{b}$ strictly contains $\bar{c}$. This contradicts the maximality of $\bar{c}$ imposed by (e), thus proving that (h4) holds. Note that this also implies $b' \notin \bar{p}$, otherwise if $b' \in \bar{p}$, we have that $b' \sim b$ are both in r', thus $b = b' \in \bar{p}$, contradicting (h4).

Assign $p_l = b'$, so we have $\bar{p}' = \bar{p} \cdot p_l$ and let $r_l = r'[b \to b']$. (To avoid many subscripts, let $r_l = s'$). Observe that

$$s' \text{ contains } \bar{p}' \text{ and } r_{|i+1}. \tag{h5}$$

In fact s' contains $\bar{p}$, as observed earlier, and s' contains b' by construction; moreover s' contains $r'_{|i+1}$ which contains $r_{|i+1}$ by (e). We now show that $\bar{p}'$ and s' have all the desired properties.

(a) By construction b' is $(i+1)$-compatible.

(b) The elements of $\bar{p}'$ are pairwise distinct, as $b' \notin \bar{p}$.

(c) By our choice of b we show that $\bar{p}'$ is connected with respect to $s'_{|i+1}$. Without loss of generality assume that $\bar{p}'$ has at least size 2 (otherwise it is trivially connected). Therefore, $\bar{p}$ is not empty. Since $s'_{|i+1}$ contains $r_{|i+1}$ by (h5), the connectedness property of $\bar{p}$ with respect to $r_{|i+1}$ implies that for every repair containing $\bar{p}'$ and $s'_{|i+1}$ and for every pair of consecutive facts $a\,b$ in $\bar{p}$, every solution in s' containing a also contains b.

It remains to show the same property for the last fact a of $\bar{p}$. Consider a repair t containing $\bar{p}'$ and $s'_{|i+1}$ and suppose $t \models q(\bar{\gamma}a\bar{\delta})$ for some $\bar{\gamma}$ and $\bar{\delta}$. We have to show $b' \in \bar{\gamma}\bar{\delta}$. Let σ_{AB} be the prefix of the Γ-derivation σ going from A to B. (Notice that, since p is not empty $A \neq B$.) By property (d) of $\bar{p}$, since $\bar{p}$ is not empty, a is the last fact in $\bar{c}$. Recall that by (h2) $r' \models q(\bar{u}\bar{c}\bar{\beta})$; thus in this solution the atom A is matched by a. So we can apply Lemma 5 to r' and t with solutions $(\bar{u}\bar{c}\bar{\beta})$ and $(\bar{\gamma}a\bar{\delta})$ respectively, and Γ-derivation σ_{AB}. The hypotheses of Lemma 5 are satisfied since:

- in both solutions A is matched by a;
- by construction of B, for each atom D strictly preceding B in σ_{AB}, the fact matching D in $(\bar{u}\bar{c}\bar{\beta})$ is either in $\bar{c}$ or in $r'_{|i+1}$, both contained in t (in fact $t \supseteq \bar{p}' \supseteq \bar{p} \supseteq \bar{c}$ and $t \supseteq s'_{|i+1} \supseteq r'_{|i+1}$).

We conclude, by Lemma 5, that the facts matching B in the two solutions are equivalent, i.e., the fact matching B in $(\bar{\gamma}a\bar{\delta})$ is equivalent to b (which is the fact matching B in $\bar{u}\bar{c}\bar{\beta}$).

In t the unique fact equivalent to b is b' (since $b' \in \bar{p}' \subseteq t$), thus the fact matching B in $(\bar{\gamma}a\bar{\delta})$ is b'. We have thus proved that any solution in t containing the last fact a of p also contains b'.

(d) The following claim together with strong i-minimality of r' and $r' \models q(\bar{u}b\bar{\gamma})$ for some $\bar{\gamma}$, shows that

(I) s' is also strong i-minimal,
(II) $s' \models q_{\leq i}(\bar{u})$, and
(III) $s' \models q(\bar{u}b'\bar{\delta})$ for some $\bar{\delta}$.

▷ Claim 10. Assume $\mathrm{PCond}_\tau(i)$. Let s be a strong i-minimal repair such that $s \models q(\bar{\alpha}a\bar{\beta})$ where $\bar{\alpha}$ matches $S_{\leq i}$ and a is $(i+1)$-compatible. Then for any $a' \sim a$ we have that $s' = s[a \mapsto a']$ is strong i-minimal and $s' \models q(\bar{\alpha}a'\bar{\delta})$ for some $\bar{\delta}$.

(e) Let $\bar{e}$ be the maximal suffix of $\bar{p}'$ such that, for a strong i-minimal repair t containing $\bar{p}'$ and $s'_{|i+1}$ we have $t \models q(\bar{u}\bar{e}\bar{\delta})$ for some $\bar{\delta}$. Since $s' \models q(\bar{u}\bar{b}'\bar{\delta}')$ for some $\bar{\delta}'$ by item (III) above, $\bar{e}$ cannot be empty. Then let $\bar{e} = \bar{d}\bar{b}'$, where $\bar{d}$ is a suffix of $\bar{p}$.

Since $s'_{|i+1}$ contains $r_{|i+1}$ by (h5), in particular t is a strong i-minimal repair containing $\bar{p}$ and $r_{|i+1}$. Then, by maximality of $\bar{c}$, $\bar{d}$ must be a suffix of $\bar{c}$, implying that $\bar{u}\bar{d}\bar{b}'$ is a subset of $\bar{u}\bar{c}\bar{\beta}'$. Since by definition $\bar{u}\bar{c}\bar{\beta}'$ does not contain any k-set in $\Delta(q, D)$, we have $\bar{u}\bar{d}\bar{b}' \notin \Delta(q, D)$ as needed.

This completes the proof of Lemma 8. ◀

▶ Remark 11. One can verify from the proof that if $q \models \mathrm{PCond}_\tau$ where in the sequence τ each set S_i contains exactly one atom of q, then the fixpoint computing $\mathrm{Cert}(q)$ is bounded, i.e., $\mathrm{Cert}(q)$ can be expressed in FO.

It remains to show that when a self-join-free query q does not satisfy PCond, computing $\textsc{certain}(q)$ is coNP-hard. Towards this, we build on the dichotomy result of [8] based on the notion of attack graph. First we recall this notion using our notation.

Let q be a query, let Γ be a set of primary key constraints. Given an atom A of q let

$$\begin{aligned} A^+ = \{B \text{ atom of } q \mid & \text{ there exist a } \Gamma\text{-derivation } X\ B_1 \ldots B_n \\ & \text{where } X = \mathit{key}(A),\ B_n = B, \text{ and for all } i,\ B_i \neq A\} \end{aligned}$$

Let $\mathit{vars}(A^+) = \bigcup_{B \in A^+} \mathit{vars}(B)$. Given two atoms A and B of q we say that A *attacks* B if there exists a sequence $F_0, F_1, \ldots, F_n$ of atoms of q and $x_1, x_2, \ldots, x_n$ of variables not in $\mathit{vars}(A^+)$ such that $A = F_0, B = F_n$ and for all $i > 0$, x_i is a variable occurring both in F_{i-1} and F_i. The attack from A to B is said to be *weak* if B is Γ-determined by A. The *attack graph* of q and Γ is the graph whose vertices are the atoms of q and whose edges are the attacks. A cycle in this graph is weak if all the attacks involved are weak.

The dichotomy result of [8] can be stated as:

▶ **Theorem 12** ([8], Theorem 3.2). *Let q be a self-join-free query and Γ a set of primary key constraints. If every cycle in the attack graph of q and Γ is weak, then* $\textsc{certain}(q)$ *can be computed in polynomial time; otherwise* $\textsc{certain}(q)$ *is* coNP*-complete.*

To show that our polynomial time algorithm covers all polynomial-time cases, we prove that if the attack graph of q and Γ contains only weak cycles, then PCond holds.

▶ **Theorem 13.** *Assume q is a self-join-free query and Γ a set of primary key constraints. If the attack graph of q and Γ contains only weak cycles then $q \models$ PCond.*

5 Path queries

The dichotomy conjecture has also been shown to hold for *path queries* [7]. Recall that a path query of length n is a Boolean conjunctive query with $n+1$ distinct variables $x_0, x_1, \cdots, x_n$ and n atoms $A_1, \cdots, A_n$ such that $A_i = R_i(\underline{x_{i-1}}\ x_i)$ for some symbol R_i of σ. The query may contain self-joins, i.e., there could be $R_i = R_j$ for $i \neq j$.

For this section, assume that the relational signature σ contains only symbols of arity two and that the set Γ of constraints assigns to each symbol R of σ its first component as a primary key. Note that a path query can be described by a word over σ (e.g., the word describing q is $R_1 \cdots R_n \in \sigma^*$). For simplicity, we will henceforth blur the distinction between path queries and words over σ.

Following [7] we define the language $\mathcal{L}^{\looparrowright}(q)$ as the regular language defined by the following finite state automaton $\mathcal{A}^q$ with epsilon-transitions (we use $s, t, \ldots$ to denote words over σ). The set of states of $\mathcal{A}^q$ is the set of all prefixes of q, including the empty prefix ε, which is the initial state. There is only one accepting state, which is q. There is a transition reading R from state s to the state sR. Moreover, there is an ε-transition in $\mathcal{A}^q$ from any state sR to any state tR such that tR is a prefix of s.

The dichotomy result of [7] can be formulated as follows[1]:

▶ **Theorem 14** ([7], Theorem 3.2)**.** *Let q be a path query. If q is a factor of all the words in the language $\mathcal{L}^{\looparrowright}(q)$, then* CERTAIN$(q)$ *can be evaluated in* PTIME*; otherwise,* CERTAIN(q) *is* coNP*-complete.*

We will show that, also in this case, $\text{Cert}(q)$ captures CERTAIN(q) for all polynomial-time path queries q (recall that $\text{Cert}(q)$ denotes $\text{Cert}_k(q)$ with $k = |q|$).

▶ **Theorem 15.** *Let q be a path query of length n. If q is a factor of all the words in the language $\mathcal{L}^{\looparrowright}(q)$, then* CERTAIN$(q) = \text{Cert}(q)$.

The rest of this section is devoted to the proof of Theorem 15. We will make use of the following fixpoint computation introduced by [7, Fig. 5]. For a fixed path query q and database instance D, let $N(q, D)$ be the set of pairs $\langle c, s\rangle$, where $c \in \mathit{adom}(D)$ and s is a prefix of q, computed via the following fixpoint algorithm.

Initialization Step: $N(q, D) \leftarrow \{\langle c, q\rangle \mid c \in \mathit{adom}(D)\}$
Iterative Step: If s is a prefix of q, add $\langle c, s\rangle$ to $N(q, D)$ if one of the following holds:

1. sR is a prefix of q and there is a fact $R(\underline{c}\ a)$ of D such that for every fact $R(\underline{c}\ b)$ of D we have $\langle b, sR\rangle \in N(q, D)$;
2. There is an ε transition from s to t in $\mathcal{A}^q$ and there is a fact $R(\underline{c}\ a)$ of D such that for every fact $R(\underline{c}\ b)$ of D we have $\langle b, tR\rangle \in N(q, D)$.

Let $N(q)$ be the set of all databases D such that there exists $c \in \mathit{adom}(D)$ with $\langle c, \varepsilon\rangle \in N(q, D)$.

▶ **Lemma 16** ([7], (proof of) Lemma 6.4)**.** *For every path query q, if q is a factor of every word in the language $\mathcal{L}^{\looparrowright}(q)$, then* CERTAIN$(q) = N(q)$.

In view of Lemma 16, Theorem 15 is now a direct consequence of the following proposition.

▶ **Proposition 17.** *For every path query q of length n, and assuming every word of $\mathcal{L}^{\looparrowright}(q)$ contains q as factor, we have $N(q) = \text{Cert}(q)$.*

Note that $\text{Cert}(q) \subseteq N(q)$ follows from $\text{Cert}(q) \subseteq$ CERTAIN(q) (Proposition 3) combined with CERTAIN$(q) = N(q)$ (Lemma 16). So we are left with proving $N(q) \subseteq \text{Cert}(q)$. Let $D \in N(q)$. We will prove that $D \in \text{Cert}(q)$.

For all $l \geq 0$ let s_l be the prefix of q of length l (i.e., $s_0 = \varepsilon$ and $s_n = q$). For every database D and fact $u = R(\underline{a}\ b)$ in D, let us define $\mathit{trace}(u) = R$, $\mathit{key}(u) = a$, and $\mathit{last}(u) = b$. For a sequence of (possibly repeating) facts $\Pi = u_1, \ldots, u_k$ of a database D, we define $\mathit{last}(\Pi) = \mathit{last}(u_k)$ and $\mathit{trace}(\Pi) = \mathit{trace}(u_1) \cdots \mathit{trace}(u_k) \in \Sigma_q^*$. Also, let $S_\Pi = \{u_1, \ldots, u_k\}$ be the *set* of facts in the sequence. Further, Π is called a **valid path** if (i) S_Π is a partial

[1] Actually, [7] provides a much finer tetrachotomy between FO, NL-complete, PTIME-complete and coNP-complete. Here we restrict our attention to the dichotomy between PTIME and coNP-complete.

repair of D, (ii) for all $i < k$ we have $\mathit{last}(u_i) = \mathit{key}(u_{i+1})$, and (iii) $\mathit{trace}(\Pi)$ is a prefix of q. In particular, for any valid path Π of length k, we have $\mathit{trace}(\Pi) = s_k$. Further, for any prefix s_l of q we write $\boldsymbol{\mathit{trace}}(\Pi) \sim s_l$ if there exists a run of the automaton $\mathcal{A}^q$ on $\mathit{trace}(\Pi)$ ending in state s_l.

Let $D \in N(q)$. Let $N^i(q, D)$ and $\Delta^i(q, D)$ be the fix-point computations of $N(q, D)$ and $\Delta(q, D)$ at their i^{th} steps, respectively. To prove that $D \in \mathrm{Cert}(q)$, we will use the following claim:

▷ Claim 18. For all $i \geq 0$, For $c \in \mathit{adom}(D)$ and for all prefix s_l of q if $\langle c, s_l\rangle \in N^i(q, D)$ then for all valid path Π where $\mathit{last}(\Pi) = c$ and $\mathit{trace}(\Pi) \sim s_l$ we have $S_\Pi \in \Delta^i(q, D)$.

Let us show that the claim implies $D \in \mathrm{Cert}(q)$. As $D \in N(q)$, there exists $c \in \mathit{adom}(D)$ such that $\langle c, \varepsilon\rangle \in N^m(q, D)$ for some step m. But note that $\langle c, \varepsilon\rangle \in N^m(q, D)$ can only be produced by application of Rule 1 in the *Iteration step* (Rule 2 is not possible since ε transitions do not start at the state ε). This implies that if R is the first relation occurring in q then there exists a fact of the form $R(\underline{c}\ a)$ and for all facts of the form $R(\underline{c}\ b)$ in D we have $\langle b, R\rangle \in N^{m-1}(q, D)$. For each such b we can apply the claim with the valid path $\Pi = R(\underline{c}\ b)$, obtaining $\{R(\underline{c}\ b)\} \in \Delta^{m-1}(q, D)$ for every $R(\underline{c}\ b)$. Hence, $\emptyset \in \Delta^m(q, D)$ which implies $D \in \mathrm{Cert}(q)$.

6 Cert$_k$ does not capture all polynomial-time queries

We have shown that for all self-join free and path queries whose certainty is solvable in polynomial time, the algorithm Cert$_k$ computes $\textsc{certain}(q)$ with $k = |q|$. A natural question is whether this extends to all queries for which certainty is solvable in polynomial time. In this section, we show that this is not the case. There exists query q whose certain answers can be computed in polynomial time but the algorithm $\mathrm{Cert}_k(q)$ will always have a false negative for any choice of k.

▶ **Theorem 19.** *Let q_4 be the query $q_4 = R(\underline{x}\, yz) \wedge R(\underline{z}\, xy)$. Then $\textsc{certain}(q_4)$ is in polynomial time but cannot be computed by $\mathrm{Cert}_k(q_4)$, for any choice of k.*

Before proving the theorem we discuss some special properties of the query q_4. Note that q_4 contains only two atoms but has self join. For any database D and facts $a, b, c \in D$, we have that if $D \models q_4(ab) \wedge q_4(ac)$ then $b = c$, and if $D \models q_4(ba) \wedge q_4(ca)$ then $b = c$. Also, if $D \models q_4(ab) \wedge q_4(bc)$ then $D \models q_4(ca)$.

Since q_4 contains only two atoms, for a database D, it is convenient to describe the set of solutions to q_4 as a graph. More precisely we say that the facts $a, b \in D$ are *q_4-related* if $D \models q_4(ab)$ or $D \models q_4(ba)$. We define the solution graph of D, denoted by G_D to be an undirected graph whose vertices are the facts of D and pairs of q_4-related facts form the edges. From the properties of q_4 it follows that every connected component in G_D is always a clique of size less than or equal to 3. If a clique has exactly three vertices we call it a *triangle*.

First we show that $\textsc{certain}(q_4)$ is in polynomial time by reducing it to bipartite matching.

▶ **Lemma 20.** *$\textsc{certain}(q_4)$ is in* PTime.

Proof. Fix an input database D. Without loss of generality, assume that there are no facts $a \in D$ such that $D \models q_4(aa)$[2]. We reduce $\textsc{certain}(q_4, D)$ to a bipartite graph matching problem. Consider the bipartite graph $G = (V_1 \cup V_2, E)$ where V_1 is the set of all blocks and V_2 is the set of maximal cliques in G_D. Let $(v_1, v_2) \in E$ if the block v_1 contains a fact which is in the clique v_2.

Suppose that there is a V_1-saturating matching, that is, an injective function $f : V_1 \to V_2$ such that $(v_1, f(v_1)) \in E$ for every $v_1 \in V_1$. We construct a repair r where for every block B of D, we pick the fact (or one of the facts, if there are more than one) which is in $f(B)$. In this way, no two chosen facts will be in the same clique, and also since there is no solution of the form $q_4(aa)$ in D, no two chosen facts will form a solution to q_4. Thus, $r \not\models q_4$.

Conversely, if q_4 is not certain in D, let r be a repair such that $r \not\models q_4$. For each block B of D let $r(B)$ be the fact of B belonging to r. Note that, since V_2 is a partition of D, each $r(B)$ belongs to a unique clique in V_2. Define $f : V_1 \to V_2$ such that each block $B \in V_1$ is mapped to the clique in V_2 where $r(B)$ lies. To verify that f is a witness function of a V_1-saturating matching for G, note that for every $B \in V_1$ we have $(B, f(B)) \in E$, as B and $f(B)$ both contain $r(B)$. Moreover f is injective, otherwise if f maps two distinct blocks to the same clique, this clique contains at least two elements of r; these two elements are therefore q_4-related, and thus $r \models q_4$.

Thus, to check if $D \in \textsc{certain}(q_4)$, it is sufficient to check if there is a bipartite matching of G that saturates V_1. This is known to be in PTime [5]. ◀

We now prove that for all k, $\mathrm{Cert}_k(q_4)$ does not compute $\textsc{certain}(q_4)$. To this end, for every $n \geq 4$ we exhibit a database D_n such that

- $D_n \models \textsc{certain}(q_4)$ (Proposition 21);
- $D_n \not\models \mathrm{Cert}_{n-2}(q_4)$ (Proposition 22).

Fix some $n \geq 4$. The database D_n has n blocks of the form $B_1, \cdots, B_n$ where each B_i consists of $n-1$ facts denoted $b_i^1, \cdots, b_i^{n-1}$. Further, D_n also has $(n-1)(n-3)$ blocks of the form E_l^j for every $1 \leq j \leq n-1$ and $1 \leq l \leq n-3$, where each E_l^j has two facts denoted by u_l^j and v_l^j. The solution graph of D_n is depicted in Figure 1.

D_n is defined in such a way that every fact of D_n is part of a "triangle". More precisely, we have the following triangles in the solution graph of D_n: For every $1 \leq j \leq n-1$, the triples $\{b_1^j, b_2^j, u_1^j\}$ and $\{b_{n-1}^j, b_n^j, v_{n-3}^j\}$ form triangles and for every $1 \leq l < n-3$ we have a triangle $\{v_l^j, u_{l+1}^j, b_{l+2}^j\}$.

▶ **Proposition 21.** *For every $n \geq 2$, $D_n \models \textsc{certain}(q_4)$.*

To see this, note that there are $n + (n-1)(n-3)$ blocks in D_n and there are $(n-1)(n-2)$ cliques (all triangles) in the solution graph of D_n. Thus, in the corresponding bipartite graph (cf. Lemma 20) size of V_1 is strictly smaller than the size of V_2. Hence, there cannot be a V_1-saturating matching which implies $D_n \models \textsc{certain}(q_4)$.

▶ **Proposition 22.** *Let $k \geq 2$. $D_{k+2} \not\models \mathrm{Cert}_k(q_4)$.*

Proof sketch. To prove the proposition, for a set of blocks $\mathbb{X} = \{X_1, \ldots X_k\}$ of D_{k+2}, if $W = \{a_1, \ldots a_k\}$ is a set of facts where each $a_i \in X_i$, then we call W a partial repair of $\mathbb{X}$. Further, W is called an **obstruction set** of $\mathbb{X}$ if W satisfies some "desired properties". In

[2] If there is a fact $a \in D$ such that $D \models q(aa)$ then suppose the block containing a is a singleton set then clearly $D \models \textsc{certain}(q_4)$. Otherwise, $D \models \textsc{certain}(q_4)$ iff $D \setminus \{a\} \models \textsc{certain}(q_4)$, so we can consider the smaller database instance.

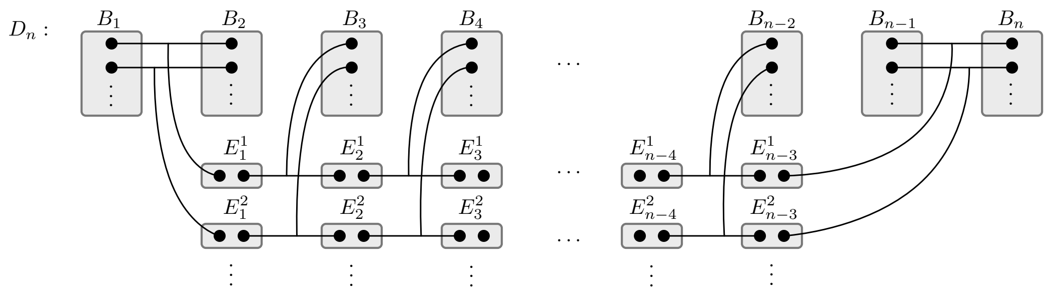


Figure 1 Solution graph for database D_n. Black dots denote facts, rectangles denote blocs, and three-pointed edges denote "triangles" in the solution graph of D_n. There are $n-1$ facts in each B_i and two facts in every E_k^j.

particular we will show that if W is an obstruction set then $W \not\models q_4$. Next we will prove that for any set of blocks $\mathbb{X} = \{X_1, \ldots X_k\}$ of D_{k+2}, we can always pick a partial repair W of $\mathbb{X}$ such that W is an obstruction set.

Now suppose $D_{k+2} \in \text{Cert}_k(q_4)$ then it follows that there has to exist at least one obstruction set $W \in \Delta_k(q_4, D_{k+2})$. We pick the minimum i such that there is some obstruction set in i^{th} step of $\Delta_k(q_4, D_{k+2})$ computation. Note that $i = 0$ is not possible since obstruction sets do not contain a solution to q_4. Thus, to obtain a contradiction, we will show that if there is an obstruction set in $\Delta_k(q_4, D_{k+2})$ at i^{th} step, then there has to exist an obstruction set in $\Delta_k(q_4, D_{k+2})$ in $(i-1)^{th}$ step. ◀

▶ **Theorem 23.** *Let $q_4 = R(\underline{x}\ yz) \wedge R(\underline{z}\ xy)$. Then $\neg\text{CERTAIN}(q_4)$ is complete for bipartite matching under* LogSpace*-reductions.*

Proof. From the proof of Lemma 20 it follows that we can reduce $\neg\text{CERTAIN}(q_4)$ to bipartite matching. The reader can verify that the reduction is in LogSpace. We now prove the other direction.

Given a bipartite graph $G = (V_1 \cup V_2, E)$, let $V_1 = \{s_1, \ldots s_n\}$ and $V_2 = \{t_1, \ldots t_m\}$. Consider the problem of determining whether there exists a matching that saturates V_1. We will reduce this problem to $\neg\text{CERTAIN}(q_4)$.

For all $s_j \in V_1$ let $N(s_j) \subseteq V_2$ denote the neighbours of s_j and similarly for all $t_i \in V_2$ let $N(t_i) \subseteq V_1$ denote the neighbours of t_j.

First note that if there is some $s_j \in V_1$ such that $N(s_j) = \emptyset$, then clearly there cannot be a matching that saturates V_1. So assume that for every $s_j \in V_1$, $N(s_j) \neq \emptyset$. Similarly, if there is some $t_i \in V_2$ such that $N(t_i) = \emptyset$, then t_i does not contribute to any matching and hence can be removed from the input. So we can also assume that for every $t_i \in V_2$, $N(t_i) \neq \emptyset$. Further, suppose there is some t_i such that $|N(t_i)| = 1$, let s_j be the single neighbour of t_i. In this case, if there exists a matching that saturates V_1 then there is a matching that saturates V_1 where s_j is matched with t_i. So we can remove the pair (s_j, t_i) from the input. This means that we can assume that for every $t_i \in V_2$, $|N(t_i)| \geq 2$.

Altogether, we have $|N(s_j)| \geq 1$ for all $s_j \in V_1$ and $|N(t_i)| \geq 2$ for all $t_i \in V_2$. Note that these properties can be checked in LogSpace.

Now we define the database D_G. Note that this construction is very similar to the construction of D_n that we used to prove Theorem 19.

- For every vertex in $s_j \in V_1$ create a block B_j in D_G.
- For every $s_j \in V_1$ and $t_i \in V_2$, if $t_i \in N(s_j)$ then there is a fact denoted by b_j^i in the block B_j. By assumption $N(s_j) \geq 1$ and hence every block B_j is non-empty.

- For every $t_i \in V_2$ if $|N(t_i)| = l$ then let $s_{i_1}, \ldots s_{i_l} \in V_1$ be the neighbours of t. By the above construction, for every $j \leq l$, there is a fact of the form $b^i_{i_j}$ in B_{i_j} that corresponds to the vertex t_i.

  Now if $l = 2$ then define $b^i_{i_1}$ and $b^i_{i_2}$ such that they form a solution to q_4. Otherwise, if $l = 3$ then define $b^i_{i_1}, b^i_{i_2}$ and $b^i_{i_3}$ such that they pair-wise form a solution to q_4 (the three facts form a triangle).

  If $l \geq 4$ then create $l - 3$ new blocks denoted by $E^i_1, \ldots E^i_{l-3}$ where each E^i_j contains exactly two facts u^i_j and v^i_j. Moreover, in the same way as described in the definition of D_n earlier, define the facts appropriately such that $\{b^i_{i_1}, b^i_{i_2}, u^i_1\}$ and $\{b^i_{i_{l-1}}, b^i_{i_l}, v^i_{l-3}\}$ form triangles and for every $1 \leq j < l - 3$ we have a triangle $\{v^i_j, u^i_{j+1}, b^i_{j+2}\}$.

The reader can verify that this is exactly the construction used to define D_n. Again, this construction is in LOGSPACE. For each such j and l define $U(i, l) = \{u^i_k \mid 1 \leq k \leq l - 2\}$ and $V(i, l) = \{v^i_k \mid l - 1 \leq k \leq l - 3\}$.

Now suppose there is a matching that saturates V_1 and let us show that the query is not certain. Consider the repair r where for each block B_j we pick b^i_j if s_j is matched with t_i. Further, pick $U(i, l) \cup V(i, l)$ which gives a partial repair over $E^i_1 \ldots E^i_l$.

If some $t_i \in V_2$ is not matched with any vertex in V_1 then pick $U(i, 1) \cup V(i, 1)$ which gives a partial repair over $E^i_1 \ldots E^i_l$. It can be verified that the obtained repair does not contain any solution.

Finally, suppose there is a repair over D_G that falsifies the query then note that if b^i_j is picked in block B_j then for all other blocks $B_{j'}$, the fact $b^i_{j'}$ cannot be in the repair since that will make the query true. Also $b^i_j \in B_j$ only if there is an edge between s_j and t_i. Hence we can define the matching that maps every $s_j \in V_i$ to $t_i \in V_2$ where b^i_j is the fact in the falsifying repair from the block B_j. ◀

7 Conclusion

We have presented a simple polynomial time algorithm for certain query answering over inconsistent databases under primary key constraints. The query is always certain when the algorithm outputs "yes", but it may produce false negative answers. We showed that for any self-join-free or path query which is not coNP-hard, the algorithm correctly computes all certain answers. A similar fixpoint algorithm can be obtained for other kinds of constraints. It needs a few hypothesis such that being able to check in PTIME whether a set of facts belongs to a repair. However, the analysis of this algorithm under other kinds of constraints is yet to be studied.

It is clear that when the fixpoint of the algorithm is bounded (i.e., it converges after a number of steps which is independent of the input database) the certainty of the query can be expressed in first-order logic, see Remark 11. It turns out that the converse is also true. Using the characterizations of [8] for self-join-free queries and of [7] for path queries, we can show that when the certainty can be expressed in first-order logic, then our algorithm is bounded. This will appear in the journal version of this paper.

As we have shown, our algorithm does not solve all the known cases where certainty can be solved in polynomial time. Hence, it would be interesting to have a (decidable) characterization of the queries whose certainty can be solved using our algorithm; we leave this for future work.

References

1 Foto N. Afrati and Phokion G. Kolaitis. Repair checking in inconsistent databases: algorithms and complexity. In Ronald Fagin, editor, *Database Theory - ICDT 2009, 12th International Conference, St. Petersburg, Russia, March 23-25, 2009, Proceedings*, volume 361 of *ACM International Conference Proceeding Series*, pages 31–41. ACM, 2009. doi:10.1145/1514894.1514899.

2 Marcelo Arenas, Leopoldo E. Bertossi, and Jan Chomicki. Consistent query answers in inconsistent databases. In Victor Vianu and Christos H. Papadimitriou, editors, *Proceedings of the Eighteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 31 - June 2, 1999, Philadelphia, Pennsylvania, USA*, pages 68–79. ACM Press, 1999. doi:10.1145/303976.303983.

3 Diego Figueira, Anantha Padmanabha, Luc Segoufin, and Cristina Sirangelo. A simple algorithm for consistent query answering under primary keys. *arXiv*, 2023. arXiv:2301.08482.

4 Ariel Fuxman and Renée J. Miller. First-order query rewriting for inconsistent databases. *J. Comput. Syst. Sci.*, 73(4):610–635, 2007. doi:10.1016/j.jcss.2006.10.013.

5 John E. Hopcroft and Richard M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM J. Comput.*, 2(4):225–231, 1973. doi:10.1137/0202019.

6 Phokion G. Kolaitis and Enela Pema. A dichotomy in the complexity of consistent query answering for queries with two atoms. *Inf. Process. Lett.*, 112(3):77–85, 2012. doi:10.1016/j.ipl.2011.10.018.

7 Paraschos Koutris, Xiating Ouyang, and Jef Wijsen. Consistent query answering for primary keys on path queries. In Leonid Libkin, Reinhard Pichler, and Paolo Guagliardo, editors, *PODS'21: Proceedings of the 40th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, Virtual Event, China, June 20-25, 2021*, pages 215–232. ACM, 2021. doi:10.1145/3452021.3458334.

8 Paraschos Koutris and Jef Wijsen. Consistent query answering for self-join-free conjunctive queries under primary key constraints. *ACM Trans. Database Syst.*, 42(2):9:1–9:45, 2017. doi:10.1145/3068334.

9 Paraschos Koutris and Jef Wijsen. Consistent query answering for primary keys in datalog. *Theory Comput. Syst.*, 65(1):122–178, 2021. doi:10.1007/s00224-020-09985-6.

10 Jef Wijsen. A remark on the complexity of consistent conjunctive query answering under primary key violations. *Inf. Process. Lett.*, 110(21):950–955, 2010. doi:10.1016/j.ipl.2010.07.021.