FINDING CONSISTENT ANSWERS FROM INCONSISTENT DATA: SYSTEMS, ALGORITHMS, AND COMPLEXITY

by

Xiating Ouyang

A dissertation submitted in partial fulfillment of

the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2023

Date of final oral examination: November 21, 2023

To my family

ACKNOWLEDGMENTS

It is with such a bittersweet feeling that I must write this section at the very last. Countless people have helped me over the past few years, and it seems daunting to properly thank everyone who made this dissertation possible. Nonetheless I make this attempt, so here we go.

First and foremost, this dissertation would not have been possible without the guidance, help, and insights of my advisor, Paris Koutris. I still remember the first day, when Paris explained to me the connection between SQL queries and first-order logics. Little did I know that this would lead to such a pleasant journey of exploration that has just come to an end. Throughout the years, he attentively listened to my ideas, patiently endured my stupidity, insightfully streamlined our solutions, and candidly advised on my decisions. Most importantly, he offered me the supreme luxury that I can ever hope for: freedom. It is extremely fortunate for me to immerse myself in research while working with Paris, and I will never regret spending some prime years of my life freely culminating this epsilon progression in the knowledge of mankind.

My heartfelt gratitude also extends to my coauthor, Jef Wijsen. We have only met remotely since we started collaborating in 2020, yet I have learned so much from him on conducting research, writing papers, and being a great collaborator. Alongside his kindness and humor, I particularly admire his extraordinary patience and persistence in obtaining the dichotomy for CQA over the course of almost two decades—a perfect illustration of scholarship at its finest.

My other defense committee members, Uri Andrews, Jin-Yi Cai, and Jignesh Patel, also invested heavily in this dissertation. I wish to thank Uri for his kind representation of the Wisconsin Logic Group in the committee. Jin-Yi often draws nontrivial connections between seemingly unrelated subjects and offers fruitful insights on various philosophical matters, and I could always learn something unexpected from our discussions. I am very grateful for Jignesh's thought-provoking comments, questions, and insights, which never fail to ground me on the practical aspects of this thesis work.

I have been fortunate to learn from Eric Bach, Shuchi Chawla, AnHai Doan, Jun Le Goh, Dieter van Melkebeek, Steffen Lempp, and Xiangyao Yu in Wisconsin. In particular, I highly appreciate AnHai's unique wisdom over various matters. I also enjoyed my interactions with the engaging database community, and my pleasant conversations with Molham Aref, Marcelo Arenas, Pablo Barceló, Phokion Kolaitis, Wim Martens, Jeff Naughton, Hung Q. Ngo, and Remy Wang vividly stood out. I was also fortunate to be mentored by Ren Mao, Ning Tan, and Fan (Amy) Yang at Meta, Alekh Jindal and Abhishek Roy at Microsoft Gray Systems Lab, and Steve Foote and Lowell Rausch at ThermoFisher Scientific. This also extends to the generous hospitality of the entire Rausch family: Emily, James, Kate, Michael, and Violet—those extremely nice people are my perfect instructors of Midwest 101. I thank Angela Thorp for her kindness and help in all my administrative requests. I also thank my therapists Wei-Chiao Hsu and Jo Hoese (and our group) for offering me support when I needed it the most. I was fortunately supported by an Anthony C. Klug NCR Fellowship for a year and wish to thank the donors for their gracious generosity.

Before hopping on a one-way flight from Hong Kong to Chicago, Yixin Cao at PolyU warmly introduced me to theoretical computer science research and spared no effort in supporting my applications. Among other influence, his lingering image of a critical writer would always emerge whenever I write, and I hope this dissertation does not let him down. At PolyU, Jiannong Cao, Rocky K. C. Chang, Dannis Liu, Qin Lu, and Vincent Ng taught my first classes in computer science and kindly introduced me to multiple aspects in the field, and I thank them for their early inspirations and guidance.

My journey is not complete without the companionship of many dear friends who kindly helped, heard, and perhaps most importantly, putting up with me along the way. I wanted to thank especially Jingcheng Xu, Yue Shi, and Jiang Chang for their "strongly-connected" friendship that has lasted since high school. We have gone through a lot; while we can be continents apart, but as I look back, nothing ever fades. The StudentSay group deserves my dedicated shoutout: Elvis Chang, Kaiyang Chen, Maggie Chen, Yang Guo, Justin LiXie, Holdson Liang, Eric Lin, Jifan Zhang, and

Xingjian Zhen. Our diverse business sectors have added to my life so many colors that had been invisible to me before meeting those amazing people, and I can always find warmth that shields me away from the coldest days in Madison. Paris' group has always been a peaceful and joyful harbor for me in the midst of noise from the real world, and I thank my great peers, officemates, and collaborators Shaleen Deep, Austen Z. Fan, Zhiwei Fan, and Hangdong Zhao for making that possible. My interactions with friends in Xi'an, Hong Kong, and Madison have been memorable, and an incomplete list include Song Bian, Ting Cai, Xufeng Cai, Zhicheng Cai, Bing An Chang, Yiding Chen, Nick Corrado, Yuhang Fan, Kevin Gaffney, Yue Gao, Evangelia Gergatsouli, Jinshan Gu, Xinyu Guan, Ankur Goswami, Yilin He, Hengjing Huang, Shunyi Huang, Aarati Kakaraparthy, Yuping Ke, Fengan Li, Derek Ma, Jeremy McMahan, Yujia Peng, Simiao Ren, Kaile Shen, Kartik Sreenivasan, Xiaoxi Sun, Yuxin Sun, Yifeng Teng, Peihang Wu, Chenyu Xue, Jie You, Peng Yu, Zhe Zeng, Ling Zhang, Jinman Zhao, Shawn Zhong, and Yi Zhou. My apologies for not being able to include everyone, but you are not forgotten; and if you are reading this, thank you!

Every word seems inadequate when it comes to expressing my gratitude towards my family, who have sacrificed so much, but asked so little. My parents Hong Ouyang and Bing Xia and my grandparents Feng-Ying Ma and Zhong-Zhan Ouyang have offered their unwavering love, care, support, kindness, understanding, and encouragement to me unconditionally. Among others, I am grateful for the help and advice from Weisheng Wang and Hao Wu. It is also my honor to become a Badger alumnus with my distant grandfather Felix H. Hsia [Hsi60]. Lastly, I thank my partner Yufei Gao for her companionship, patience, and kindness. She has a magic spell to attract peace and expel negativity in our lives, and I look forward to the next phase of our journey by your side.

TABLE OF CONTENTS

ABSTRACT

Most data analytical pipelines often encounter the problem of querying inconsistent data that could violate pre-determined integrity constraints. Data cleaning is an extensively studied paradigm that singles out a consistent repair of the inconsistent data. Consistent query answering (CQA) is an alternative approach to data cleaning that asks for all tuples guaranteed to be returned by a given query on all (and in most cases, exponentially many) repairs of the inconsistent data.

This dissertation investigates both practical and theoretical perspectives of CQA in the context of relational queries and databases that can possibly violate (and only violate) the primary-key constraints.

- From a practical standpoint, we identify a class of select-project-join (SPJ) queries for which CQA can be solved via a first-order rewriting with linear time guarantees. This is implemented in a system LinCQA, and we show that LinCQA often outperforms the existing CQA systems on both synthetic and real-world workloads, and in some cases, by orders of magnitude.

- From a theoretical perspective, this dissertation provides a complexity classification of CQA on representative classes of conjunctive queries with self-joins: path queries, rooted tree queries, and beyond. The classification criterion is explicit and involves query homomorphisms. This advances the current complexity trichotomy result on self-join-free conjunctive queries established in 2015.

Chapter 1

Introduction

昨夜西風凋碧樹
獨上高樓
望盡天涯路
——晏殊 《蝶戀花》

Data is the new oil. It is produced in an unprecedented high volume, processed by unparalleled computational resources, and utilized by ubiquitous and occasionally critical applications. For example, transportation providers often keep records of the ridership data generated from the business, which can be analyzed to optimize their day-to-day operations such as scheduling and staffing. In medical settings, machine learning models trained on a plethora of medical data are used to assist the diagnosis and treatments.

Yet like oil, raw data is often unusable without an accompanying infrastructure to acquire, store, maintain, process, and query it. Relentless efforts from the data management research community have thus been spent on each aspect. The outcomes have been influential: we have developed elastic storage systems [SKRC10], efficient transaction protocols [LS79, GWYY21], blazingly fast data processing algorithms and tools [ZCF+10, KNR+11, RJG+21, PTH23], and expressive query languages [Cod70, KNP+22a, KNP+23]. While we celebrate these significant progress, it has become apparent to the society that having these pipelines is not sufficient to make great applications: the quality of the data itself also matters.

In reality, data is often, if not always, dirty. The dirtiness of the data can be characterized as the violation of the predetermined *integrity constraints*: the conditions that we believe the data should ideally satisfy. Integrity constraints can be violated for various reasons, including collection of erroneous user input, system flaws when dealing with real-time data, or even careless integration of data from heterogeneous sources, to name a few. These data inconsistencies are the enemy of modern data applications, particularly in critical domains, as they directly impact the trustworthiness of data analytic results and the reliability of the prediction thereof. A natural question thus arises: *how can we manage dirty data effectively?*

Data cleaning [RD00] is perhaps the most widely used approach to manage dirty (or inconsistent) data in practice. It first *repairs* (or clean) the dirty data by modifying the inconsistent portions of the data to improve its quality, and then the users may use the cleaned data (or repair) for their specific tasks. There has been a long line of research on data cleaning. Several frameworks have been proposed [GGM+21, ROA+21, GMPS13, AK09, GGZ03], using techniques such as knowledge bases and machine learning [RCIR17, CIKW16, BKL13, HCG+18, EEI+13, LRB+21, BMNT15, CIP13, TCZ+14, KWW+16]. Data cleaning has also been studied under different contexts [KL21, CCX08, KIJ+15, BFG+07, PSC+15]. Indeed, data cleaning is involved in almost all preprocessing steps in modern data processing tasks.

1.1 Motivations

While data cleaning has been widely adopted, its drawbacks still remain.

1. It is challenging to find the "best" repair. The process of data cleaning is often ad hoc and arbitrary choices are frequently made regarding which data to keep in order to restore data consistency. This comes at the price of losing important information since the number of cleaned versions of the database can be exponential in the size of data. For example, it is possible for two data cleaning methods to yield two different repairs, on which the query answer may differ.

2. The data cleaning process is time-consuming and blocking the data analytics pipeline. There have been efforts to accelerate the data cleaning process [RCIR17, CIP13, CMI+15, ROA+21], but in most cases, users need to wait until the data is clean before it becomes available.

In light of these challenges, *consistent query answering (CQA)* has been proposed as an alternative approach to manage inconsistent data. Informally speaking, given a query to the dirty data, CQA returns the tuples that are *guaranteed* to be returned by the query on all possible repairs of the dirty data, called the *certain answers*. One can think of this definition in a procedural way: we first find *all* possible repairs of the dirty data, and then execute the query on each possible repair, and finally return the certain answers by computing the intersection of all query answers obtained. CQA was initiated by the seminal work by Arenas, Bertossi, and Chomicki [ABC99]. After twenty years, their contribution was acknowledged in a *Gems of PODS session* [Ber19]. An overview of complexity classification results in CQA appeared recently in the *Database Principles* column of SIGMOD Record [Wij19b].

One advantage of CQA is that it considers *all* possible repairs, instead of attempting to find the "best" one. The certain answers are also trustworthy in the sense that they will be returned regardless of which repair or data cleaning method we use, which is ideal for critical applications. If solved efficiently, CQA would also eliminate the blocking time of data cleaning and speed up the entire data analytic pipelines.

Despite its advantages, it remains to tackle the following challenges to materialize these promises:

- **Classification**: *for which queries can CQA be solved efficiently?* This problem is important since by definition of CQA, the naive solution is to enumerate all possible repairs, and the number of repairs can be exponential in the size of data, unfortunately. It is therefore crucial to learn for which queries, enumerating the repairs can be avoided, or otherwise necessary under standard complexity hypotheses.

- **Algorithm**: *how can we solve CQA efficiently?* For the CQA instances that we know can be solved without enumerating the repairs, it is in our best interest to find algorithms that solve it as efficient as possible.

- **System**: *can CQA be integrated into existing systems?* Once we identify an efficient algorithm for CQA, it remains a challenge to implement the algorithm to existing data management systems to realize its full potential. Ideally, the CQA component should be a separate component that can be easily integrated across heterogeneous platforms.

This dissertation focuses on CQA in the context of join queries and relational databases that could possibly violate the primary-key constraints. The primary-key constraint imposes that no two distinct records in the database shall have the same primary-key value for the primary-key attribute, e.g. SSN, or passport number. It is perhaps the most widely imposed integrity constraint over relational systems.

We now elaborate the motivations for solving the aforementioned challenges from both system and theoretical perspectives.

System Motivations. CQA could serve a strong system copilot for data analytic pipelines. Upon receiving a query, the system can first examine whether the CQA problem for that query can be efficiently solved using solutions from the **Classification** challenge. If it can be efficiently solved, the CQA component would then invoke the efficient implementation of a CQA algorithm from the **Algorithm** challenge. This entire component then constitutes a solution to the **System** challenge, which could potentially benefit data analytics pipelines deployed on different platforms.

Theoretical Motivations. Aside from its practical impacts, the theoretical interests of CQA also abound. In database theory, CQA on primary keys for a given Boolean query q (i.e., a first-order sentence) is often defined as the following decision problem $\mathsf{CERTAINTY}(q)$:

Problem: $\mathsf{CERTAINTY}(q)$

Input: A database instance **db**.

Question: Does q evaluate to true on every repair of **db**?

The problem $\mathsf{CERTAINTY}(q)$ is always in the complexity class **coNP** by definition, since a repair that violates q can be verified in **PTIME**.

One of the seven *Millennium Prize Problems* is the **PTIME** vs. **NP** problem. Ladner's [Lad75] theorem asserts that if $\mathbf{PTIME} \subsetneq \mathbf{NP}$, then there are problems that are **NP**-intermediate: decisions problems that are in **NP**, but not in **PTIME** nor **NP**-complete. We remark that Ladner's theorem also implies the existence of **coNP**-intermediate problems.

Many computational problems fall under the framework of *Constraint Satisfaction Problems* (CSP), including the infamous vertex cover, k-coloring of a graph, and so on. In a celebrated result, Bulatov and Zhuk [Bul17, Zhu20] independently showed that CSP problems enjoy a dichotomy between **PTIME** and **NP**-complete, avoiding the **NP**-intermediate class. In the context of CQA, it has been long conjectured that for every Boolean *conjunctive queries* (CQ) q (i.e., a query expressible as a first-order sentence using $\exists$ and $\wedge$), the complexity of $\mathsf{CERTAINTY}(q)$ exhibits a **PTIME**/**coNP**-complete dichotomy, avoiding the **coNP**-intermediate landscape.

Conjecture 1.1. *For each Boolean conjunctive query q, the problem* $\mathsf{CERTAINTY}(q)$ *is in* **PTIME** *or* **coNP***-complete.*

An even stronger conjecture is that the dichotomy of Conjecture 1.1 extends to unions of conjunctive queries. Fontaine [Fon15] showed that this stronger conjecture *implies* the dichotomy theorem for conservative *Constraint Satisfaction Problems* (CSP) [Bul11]. Hence the complexity classification of CQA deepens our understanding of the relationship between **PTIME** and **NP**, and offers an alternative view of the conservative CSPs.

1.2 Current Progress

We now provide an overview of some notable progress in CQA.

From a theoretical perspective, Fuxman and Miller [FM07] identified $\mathcal{C}_{\mathsf{forest}}$, a class of Boolean conjunctive queries q for which $\mathsf{CERTAINTY}(q)$ can be solved via an **FO**-rewriting: we can construct another query such that executing it directly on the inconsistent database will return the consistent answers of the original query. This notion will be made precise in Chapter 2 later. Wijsen [Wij10] then provided an explicit criterion for the **FO**-border of $\mathsf{CERTAINTY}(q)$ for the class of acyclic self-join-free Boolean conjunctive queries. Later, Kolaitis and Pema [KP12] gave a **PTIME**/**coNP**-complete dichotomy for CQA for every self-join-free Boolean conjunctive queries with two distinct atoms. Koutris and Suciu [KS14] then provided a **PTIME**/**coNP**-complete dichotomy for CQA for every self-join-free Boolean conjunctive queries with simple primary keys. Eventually, Koutris and Wijsen [KW15, KW17] settled the complexity landscape of CQA over self-join-free conjunctive queries into **FO**, **L**-hard and **PTIME**, and **coNP**-complete. Koutris and Wijsen [KW19, KW21] subsequently showed that when $\mathsf{CERTAINTY}(q)$ is in **PTIME**, it can also be expressed in symmetric stratified Datalog (which is in **L**). Theorem 1.1 summarizes the complexity classification for self-join-free Boolean conjunctive queries.

Theorem 1.1 ([KW21])**.** *For every self-join-free Boolean conjunctive query q, $\mathsf{CERTAINTY}(q)$ is in* **FO***,* **L***-complete, or* **coNP***-complete, and it is decidable which of the three cases applies.*

From a systems standpoint, most CQA systems fall into two categories: (1) systems that can compute the consistent answers to SPJ queries with arbitrary denial constraints but require solvers for computationally hard problems, or (2) systems that output the **FO**-rewriting but only target a specific class of queries that occurs frequently in practice. For (1), some notable examples include EQUIP [KPT13], which relies on Integer Programming solvers, and CAvSAT [DK21b, DK19], which requires SAT solvers). This is a natural approach since CQA is always in **coNP**, and thus one may attempt to reduce the complement of CQA to more general ones that we have efficient solvers for. For (2), Fuxman and Miller [FFM05] implemented ConQuer, which outputs the **FO**-rewriting for every query in $\mathcal{C}_{\mathsf{forest}}$, the **FO**-rewritable class they identified in [FM07], as a single SQL query. Conquesto [KJL+20] is the most recent system targeting **FO**-rewritable join queries by producing the rewriting in non-recursive Datalog.

We identify several drawbacks with all systems above. Both EQUIP and CAvSAT rely on solvers for **NP**-complete problems, which does not guarantee efficient termination, even if the input query is **FO**-rewritable. Even though $\mathcal{C}_{\mathsf{forest}}$ captures many join queries seen in practice, it excludes queries that involve *(i)* joining with only part of a composite primary key, often appearing in snowflake schemas, and *(ii)* joining two tables on both primary-key and non-primary-key attributes, which commonly occur in settings such as entity matching and cross-comparison scenarios. Conquesto, on the other hand, implements the generic **FO**-rewriting algorithm without strong performance guarantees. Moreover, neither ConQuer nor Conquesto has theoretical guarantees on the running time of their produced rewritings.

1.3 Contributions

From a system perspective, we show that if an α-acyclic self-join-free Boolean conjunctive query q has a *pair-pruning join tree* (PPJT), then $\mathsf{CERTAINTY}(q)$ admits a **FO**-rewriting that can be implemented to run in linear time in the size of the inconsistent database. Such rewritings can be extended to queries with free-variables (or projection attributes), stated in Theorem 1.2.

Theorem 1.2 ([FKOW23]). *Let q be a conjunctive query and* **db** *be a database instance of size N. If q admits a free-connex pair-pruning join tree, then the set of consistent answers OUT_c of q on db can be computed in time $O(N + |\mathsf{OUT}_c|)$.*

Recall that Yannakakis' algorithm [Yan81] can evaluate any free-connex α-acyclic Boolean conjunctive query q in time $O(N + |\mathsf{OUT}|)$, where OUT is the set of query answers. Theorem 1.2 implies that in the context of CQA, if q has a free-connex pair-pruning join tree, then its consistent answers can also be computed in the same running time, exhibiting no asymptotic overhead.

We then implemented a system LinCQA, that can output an **FO**-rewriting of every query in our class as either a SQL query or a non-recursive Datalog program. We show that LinCQA often outperforms existing CQA systems in real-world workloads, and sometimes by orders of magnitudes.

From a theoretical perspective, we push the complexity classification of $\mathsf{CERTAINTY}(q)$ beyond self-join-free conjunctive queries. We first provide in Theorem 1.3 a complete complexity classification for the class of Boolean path queries of the form

$$\exists x_1 \cdots \exists x_{k+1}(R_1(\underline{x_1}, x_2) \wedge R_2(\underline{x_2}, x_3) \wedge \cdots \wedge R_k(\underline{x_k}, x_{k+1})),$$

where it is possible that for some $i \neq j$, $R_i = R_j$.

Theorem 1.3 ([KOW21]). *For every Boolean path query q, $\mathsf{CERTAINTY}(q)$ is in* **FO**, **NL**-*complete,* **PTIME**-*complete, or* **coNP**-*complete, and it is decidable in polynomial time in the size of q which of the four cases applies.*

Comparing Theorem 1.1 and Theorem 1.3, it is striking that there are path queries q for which $\mathsf{CERTAINTY}(q)$ is **NL**-complete or **PTIME**-complete, whereas these complexity classes do not occur for self-join-free queries (under standard complexity assumptions). Moreover, when self-joins are prohibited from a path query q, $\mathsf{CERTAINTY}(q)$ is always in **FO**. So even for the restricted class of path queries, allowing self-joins immediately results in a more varied complexity landscape.

We then extend our complexity classification for path queries to a wider class of Boolean conjunctive queries that allow for arbitrary arities, which we call the *rooted tree queries*, to be defined in Chapter 5.

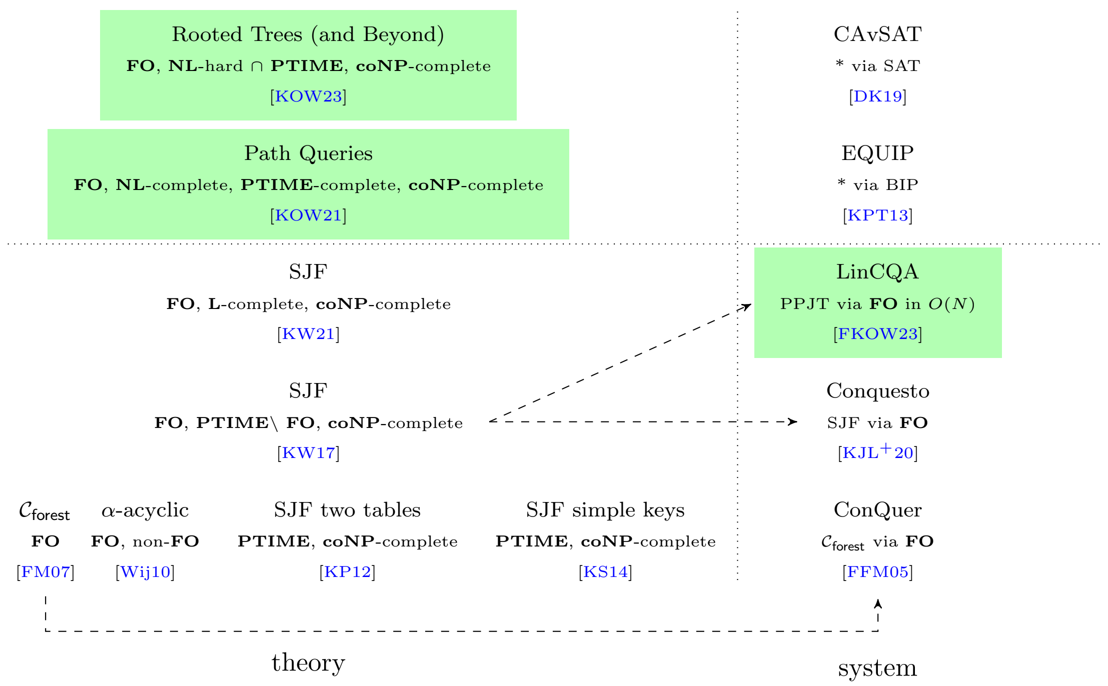

Rooted Trees (and Beyond)
FO, **NL**-hard $\cap$ **PTIME**, **coNP**-complete
[KOW23]
Path Queries
FO, **NL**-complete, **PTIME**-complete, **coNP**-complete
[KOW21]
CAvSAT
* via SAT
[DK19]
EQUIP
* via BIP
[KPT13]
SJF
FO, **L**-complete, **coNP**-complete
[KW21]
LinCQA
PPJT via **FO** in $O(N)$
[FKOW23]
SJF
FO, **PTIME**\ **FO**, **coNP**-complete
[KW17]
Conquesto
SJF via **FO**
[KJL+20]
$\mathcal{C}_{\mathsf{forest}}$
FO
[FM07]
α-acyclic
FO, non-**FO**
[Wij10]
SJF two tables
PTIME, **coNP**-complete
[KP12]
SJF simple keys
PTIME, **coNP**-complete
[KS14]
ConQuer
$\mathcal{C}_{\mathsf{forest}}$ via **FO**
[FFM05]
theory
system

Figure 1.1: A summary of some existing work on CQA and the contributions of this dissertation, highlighted in green. Results for the class of CQs that allow and prohibit self-joins are above and below the horizontal dotted lines, respectively. The theoretical (algorithm and complexity) results and the system results on CQA are to the left and right of the vertical line, respectively. The dashed arrows from A to B indicate that the theoretical work A inspires the system work B.

Theorem 1.4 ([KOW23])**.** *For every rooted tree query q, the problem* $\mathsf{CERTAINTY}(q)$ *is in* **FO**, **NL**-*hard* $\cap$ **LFPL**, *or* **coNP**-*complete, and it is decidable in polynomial time in the size of q which of the three cases applies.*

Figure 1.1 summarizes the contributions of this dissertation.

1.4 Related Work

In this section, we discuss some related work to this dissertation.

Consistent query answering (CQA). Since its inception by Arenas, Bertossi, and Chomicki [ABC99] in 1999, a long line of research has focused on obtaining the complexity classification of CQA, which we have discussed in Section 1.2. The term $\mathsf{CERTAINTY}(q)$ was coined in [Wij10] to refer to CQA for Boolean queries q on databases that violate primary keys, one per relation, which are fixed by q's schema. In particular, Fuxman's dissertation [Fux07] is the first in the field devoted to CQA. The trichotomy complexity classification of $\mathsf{CERTAINTY}(q)$ for the class of self-join-free queries have been settled in Theorem 1.1. A few extensions beyond this trichotomy result are known. The complexity of $\mathsf{CERTAINTY}(q)$ for self-join-free Boolean conjunctive queries with negated atoms was studied in [KW18]. For self-join-free Boolean conjunctive queries with respect to multiple keys, it remains decidable whether or not $\mathsf{CERTAINTY}(q)$ is in **FO** [KW20].

Little is known about $\mathsf{CERTAINTY}(q)$ beyond self-join-free conjunctive queries. Fontaine [Fon15] showed that if we strengthen Conjecture 1.1 from conjunctive queries to unions of conjunctive queries, then it implies Bulatov's dichotomy theorem for conservative CSP [Bul11]. This relationship between CQA and CSP was further explored in [LW15]. In [AKV15], the authors show the **FO** boundary for $\mathsf{CERTAINTY}(q)$ for constant-free Boolean conjunctive queries q using a single binary relation name with a singleton primary key.

The counting variant of the problem $\mathsf{CERTAINTY}(q)$, denoted $\sharp\mathsf{CERTAINTY}(q)$, asks to count the number of repairs that satisfy some Boolean query q. For self-join-free Boolean conjunctive queries, $\sharp\mathsf{CERTAINTY}(q)$ exhibits a dichotomy between **FP** and $\sharp$**PTIME**-complete [MW13]. This dichotomy has been shown to extend to queries with self-joins if primary keys are singletons [MW14], and to functional dependencies [CLPS22a]. Calautti, Console, and Pieris present in [CCP19] a complexity analysis of these counting problems under many-one logspace reductions and conducted an experimental evaluation of randomized approximation schemes for approximating the percentage of repairs that satisfy a given query [CCP21]. CQA is also studied under different notions of repairs like operational repairs [CLP18, CLPS22b] and preferred repairs [SCM12, KLP20]. CQA has also been studied for queries with aggregation, in both theory and practice [DK21a, KW23].

In practice, there are other systems supporting CQA have often used efficient solvers for Disjunctive Logic Programming [GGZ03], Answer Set Programming (ASP) [MRT15, MB10], Binary Integer Programming (BIP) [KPT13], and SAT solvers [DK19].

Certifiable robustness. The notion of certain answers also have traces in the machine learning community, with the idea that given an inconsistent training dataset, an ML classifier may still produce the same prediction to some given test point regardless of which consistent version (also called a possible world) of the inconsistent training dataset it is trained on. Robust learning algorithms have received much attention recently. Robustness of decision tree under adversarial attack has been studied in [CZS+19, VV21] and interests in certifying robust training methods have been observed [SWZ+21]. The notion of certain prediction has been reinvented in [KLW+20] to connect the certifiable robustness of Nearest Neighbor Classifier to the downstream data cleaning task. Most recently, efficient algorithms have been developed for certifying the robustness of k-Nearest Neighbors [FK22], SVM, linear regression [ZCT23], and Naive Bayes Classifiers [BOFK23].

1.5 Organization

This dissertation is organized as follows.

- Chapter 2 introduces the background, definitions, and the problem statements.
- In Chapter 3, we formally define the notion of (free-connex) pair-pruning join trees and show Theorem 1.2. We then describe the system LinCQA and provide experimental results over real-world datasets.
- In Chapter 4, we prove Theorem 1.3, the tetrachotomy theorem for path queries. Here we will introduce the notion of *word rewinding* and explain how it is useful in the complexity classification.
- In Chapter 5, we show Theorem 1.4, the trichotomy classification for tree queries, and extend it to other classes of CQs. The complexity classification would generalize the notion of word rewinding into query homomorphisms. In particular, we present a **PTIME** algorithm to find a *frugal repair* of the inconsistent database, such that all repairs satisfy q if and only if any frugal repair satisfies q whenever $\mathsf{CERTAINTY}(q)$ is in **PTIME**.
- In Chapter 6, we discuss some future directions and provide some preliminary results.
- Chapter 7 concludes this dissertation.

Chapter 2

Background

"All models are wrong, but some are useful."

—George E. P. Box

In this chapter, we define the notations used in this dissertation. While we primarily adopt the standard notations in databases, terminological correspondences to model theory will be added where appropriate. We assume disjoint sets of variables and constants and basic familiarity with first-order logic (**FO**) in Chapter 3 of [ABL$^+$22] and Chapter 2.1 of [Lib04].

We use the example database **Company** shown in Figure 2.1 to illustrate our constructs incrementally.

2.1 Databases and Queries

A *database schema* is a finite set of table names. Each table name is associated with a finite sequence of *attributes*, and the length of that sequence is called the *arity* of that table.

A *relational database instance* (or database for short) **db** associates to each table name a finite set of *tuples* that agree on the arity of the table, called a *relation*. A database instance can also

	Employee	
employee_id	office_city	wfh_city
0011	Boston	Boston
0011	Chicago	New York
0011	Chicago	Chicago
0022	New York	New York
0022	Chicago	Chicago
0034	Boston	New York

	Manager	
office_city	manager_id	start_year
Boston	0011	2020
Boston	0011	2021
Chicago	0022	2020
LA	0034	2020
LA	0037	2020
New York	0022	2020

Contact	
office_city	contact_id
Boston	0011
Boston	0022
Chicago	0022
LA	0034
LA	0037
New York	0022

Figure 2.1: An example database **Company**.

be equivalently defined as a finite structure with no functions in (finite) model theory. We denote $\mathsf{adom}(\mathbf{db})$ as the set of all constants that appear in $\mathbf{db}$, also called the *active domain* of $\mathbf{db}$.

Let $\vec{x}$ be a sequence of variables and constants. We write $\mathsf{vars}(\vec{x})$ for the set of variables that appear in $\vec{x}$. An *atom* with relation name R takes the form $R(\vec{x})$. If $\vec{x}$ does not contain any variable, then $R(\vec{x})$ is also called a *fact* or a *tuple*. It is convenient to think of a database as a finite set of facts. For two sequences of variables and constants $\vec{x}$ and $\vec{y}$, we denote $\vec{x} \cdot \vec{y}$ as the sequence of variable and constants obtained by *concatenating* $\vec{x}$ and $\vec{y}$. An empty sequence is denoted by $\langle\rangle$. When it is clear from context, we often drop the brackets around sequences of length 1. For example, we use x to denote both the variable x and the sequence $\langle x \rangle$.

Example 2.1. The **Company** database contains three (3) tables: Employee, Manager, and Contact, each with arities 3, 2, and 2 respectively. All three tables have 6 facts, including facts $\mathsf{Employee}(0011, \text{Boston}, \text{Boston})$ and $\mathsf{Contact}(\text{LA}, 0037)$ to name a few.

For the **Company** database, we can have atoms $\mathsf{Employee}(x, y, y)$, $\mathsf{Manager}(u, v, \mathsf{2020})$, and $\mathsf{Contact}(\mathtt{LA}, \mathsf{2020})$. Facts $\mathsf{Employee}(0011, \text{Chicago}, \text{New York})$ and $\mathsf{Contact}(\mathtt{LA}, \mathsf{2020})$ are in the **Company** database whereas the fact $\mathsf{Contact}(\mathtt{LA}, \mathsf{0042})$ is not. □

A *query* is a first-order formula $q(\vec{u})$ where $\vec{u}$ is a sequence of *variables* and each predicate in q is an atom. The sequence $\vec{u}$ is called the *free variables* of q. We denote $\mathsf{vars}(q)$ as the set of variables that occur in q and $\mathsf{consts}(q)$ as the set of constants in q. A query q is *Boolean* if q is a first-order sentence (i.e., $\vec{u}$ is empty), and a query q is *full* if $\vec{u} = \mathsf{vars}(q)$. For a query q, let $\vec{x} = \langle x_1, \dots, x_\ell \rangle$ be a sequence of distinct variables that occur in q and $\vec{a} = \langle a_1, \dots, a_\ell \rangle$ be a sequence of constants, then $q_{[\vec{x} \to \vec{a}]}$ denotes the query obtained from q by replacing all free occurrences of x_i with a_i and removing all variables x_i from the sequence $\vec{u}$ for all $1 \leq i \leq \ell$.

A query is a *conjunctive query* (CQ) if it is a first-order formula in $\mathbf{FO}(\exists, \wedge)$. Conjunctive queries coincide with the select-project-join (SPJ) SQL queries in database terminology. We refer to the classical *Cow Book* [RG03] for more details on the SQL syntax and semantics. Each CQ q can be represented as a succinct rule of the following rule form (by omitting the quantifier $\exists$ and replacing $\wedge$ with comma):

$$q(\vec{u}) \text{ :- } R_1(\vec{x}_1), R_2(\vec{x}_2), \dots, R_n(\vec{x}_n), \tag{2.1}$$

where each $R_i(\vec{x_i})$ is an atom for $1 \leq i \leq n$. The atom $q(\vec{u})$ is the *head* of the rule, and the set of atoms $\{R_1(\vec{x}_1), R_2(\vec{x}_2), \dots, R_n(\vec{x}_n)\}$ is called the *body* of the rule, denoted by $\mathsf{body}(q)$.

We say that q has a *self-join* if some relation name occurs more than once in the body of q. A CQ is *self-join-free* if it has no self-joins. If a self-join-free query q is understood, an atom $R(\vec{x})$ in q can be unambiguously denoted by R. If the body of a CQ of the form (2.1) can be partitioned into two nonempty parts that have no variable in common, then we say that the query is *disconnected*; or otherwise it is *connected.*

Example 2.2. Consider the query over the **Company** database:

> *return the id's of all employees who work in some office city with a manager who started in year 2020.*

It can be expressed by the following SPJ SQL query:

```
SELECT E.employee_id
FROM Employee E, Manager M
WHERE E.office_city=M.office_city
     AND M.start_year=2020
```

and the following CQ:

$$q(x) \text{ :- } \mathsf{Employee}(x, y, z), \mathsf{Manager}(y, w, 2020),$$

which essentially represents the following first-order formula

$$q(x) = \exists y, z, w : \mathsf{Employee}(x, y, z) \wedge \mathsf{Manager}(y, w, 2020).$$

The following CQ q' is a BCQ, since it merely asks whether the employee_id 0011 satisfies the conditions in q:

$$q'() \text{ :- } \mathsf{Employee}(0011, y, z), \mathsf{Manager}(y, w, 2020).$$

It is easy to see that q' is equivalent to $q_{[x \to 0011]}$.

It can be verified that the CQ q is connected. The CQ q is self-join-free since it does not contain a relation name that occurs multiple times. The □

Let $p(\vec{u})$ and $q(\vec{v})$ be two CQs. We write $p \leq_{\to} q$ if there exists a homomorphism from p to q, i.e., a mapping $h : \mathsf{vars}(p) \to \mathsf{vars}(q) \cup \mathsf{consts}(q)$ that acts as identity when applied on constants, such that $h(\vec{u}) = \vec{v}$ and for every atom $R(\vec{x})$ in p, $R(h(\vec{x}))$ is an atom of q. For $u \in \mathsf{vars}(p)$ and $v \in \mathsf{vars}(q)$, we write $p \leq_{u \to v} q$ if there exists a homomorphism h from p to q with $h(u) = v$. A CQ $q(\vec{u})$ is *minimal* if every endomorphism of q is an automorphism.

Let $\mathbf{db}$ be a database instance and $q(\vec{u})$ a query. A tuple $\vec{t}$ is an *answer* to q on $\mathbf{db}$ if it has the same length as $\vec{u}$ and $\mathbf{db}$ satisfies the first-order sentence $q_{[\vec{u} \to \vec{t}]}$. We denote $q(\mathbf{db})$ as the *set* of answers to q on $\mathbf{db}$, or formally,

$$q(\mathbf{db}) := \{\vec{t} \mid \mathbf{db} \models q_{[\vec{u} \to \vec{t}]}\}.$$

We remark here that this dissertation assumes the *set semantics*: the answers to queries and the databases are both sets of tuples.[1] In particular, when q is Boolean, $q(\mathbf{db})$ is either $\{\langle\rangle\}$ (interpreted as true), or an $\emptyset$ (interpreted as false). Note that this construct coincides with the set-theoretic definition of $0 := \emptyset$ and $1 := \{\emptyset\}$.

As a special case, if q is a CQ, it can be verified that a tuple $\vec{t}$ is an answer to q on $\mathbf{db}$ if there exists a homomorphism $\mu : \mathsf{vars}(q) \to \mathsf{adom}(\mathbf{db})$ that acts as an identity on constants in q such that $\mu(\vec{u}) = \vec{t}$, and for any atom $R(\vec{x})$ in q, the fact $R(\mu(\vec{x}))$ is in $\mathbf{db}$.

The size of a database is defined to be the total number of facts in $\mathbf{db}$, and the size of a query is defined to be the sum of all predicate arities in the query.

Example 2.3. Consider the query q in Example 2.2 and the **Company** database. We have that

$$q(\mathbf{Company}) = \{0011, 0022, 0034\}.$$

The tuple 0011 is an answer since the $\mathbf{Company} \models q_{[x\to 0011]}$, where

$$q_{[x\to 0011]} = \exists y, z, w : \mathsf{Employee}(0011, y, z) \wedge \mathsf{Manager}(y, w, \mathsf{2020}),$$

witnessed by (among others), $y = \text{Boston}$, $z = \text{Boston}$, and $w = 0011$.

Alternatively, 0011 is an answer since for the following homomorphism h:

$$\begin{cases} x \mapsto 0011 \\ y \mapsto \text{Boston} \\ z \mapsto \text{Boston} \\ w \mapsto 0011 \end{cases}$$

we have $h(x) = 0011$, $\mathsf{Employee}(h(x), h(y), h(z)) = \mathsf{Employee}(0011, \text{Boston}, \text{Boston}) \in \mathbf{Company}$, and $\mathsf{Manager}(h(y), h(w), h(\mathsf{2020})) = \mathsf{Employee}(\text{Boston}, \text{Boston}, \mathsf{2020}) \in \mathbf{Company}$. Note that while there are other homomorphisms witnessing that 0011 is a answer of $q(\mathbf{Company})$, we only list the tuple 0011 once in $q(\mathbf{Company})$ since we focus on the set semantics. In bag semantics, the multiplicity of 0011 in the *bag* $q(\mathbf{Company})$ is equal to the number of such homomorphisms. $\square$

2.2 Integrity Constraints and Primary Keys

An *integrity constraint* Σ is a finite set of first-order sentences. A database $\mathbf{db}$ *satisfies* an integrity constraint Σ if $\mathbf{db} \models \Sigma$. In this case, we say that the database $\mathbf{db}$ is *consistent* with respect to Σ, or otherwise *inconsistent* with respect to Σ.

[1]The commercial databases often adopt the *bag semantics*, in which the tuples in database and the query answer are allowed to repeat.

Let R be a relation name with $n \geq 1$ attributes. A *primary key* for R is a subset of $\{1, 2, \ldots, n\}$ of size k for $1 \leq k \leq n$, essentially specifying a subset of attribute positions in R. Without loss of generality, whenever R has a primary key of size k, we assume its primary key is $\{1, 2, \ldots, k\}$, i.e. the first k attributes. We say that R is *simple-key* if $k = 1$.

For an atom $R(\vec{u})$ where the relation R has a primary key of size k, we often write $R(\underline{\vec{x}}, \vec{y})$ instead of $R(\vec{u})$ by underlining the primary-key positions, for sequences $\vec{x}$ and $\vec{y}$ such that the length of $\vec{x}$ is k and $\vec{u} = \vec{x} \cdot \vec{y}$.

The *primary-key constraint* of a relation R with arity n and primary key $\{1, 2, \ldots, k\}$ is the first-order sentence

$$\forall \vec{x}, \vec{y}, \vec{z} : (R(\underline{\vec{x}}, \vec{y}) \land R(\underline{\vec{x}}, \vec{z}) \rightarrow \vec{y} = \vec{z}),$$

which semantically asserts that no two distinct facts in the relation R shall share the same primary key attributes.

In this dissertation, we assume that that the set Σ of integrity constraints are restricted to contain only primary-key constraints.

Let $\mathbf{db}$ be a database and Σ an integrity constraint. A subset of facts $\mathbf{r}$ of $\mathbf{db}$ is a *repair* of $\mathbf{db}$ with respect to Σ if $\mathbf{r}$ is a maximal subset of $\mathbf{db}$ such that $\mathbf{r} \models \Sigma$. It is easy to see that if $\mathbf{db} \models \Sigma$, then $\mathbf{db}$ itself is its one and only repair. However, if $\mathbf{db} \not\models \Sigma$, $\mathbf{db}$ could contain multiple repairs. We denote $\mathsf{repairs}(\mathbf{db}, \Sigma)$ as the set of all repairs of $\mathbf{db}$ with respect to Σ.

Let $\mathbf{db}$ be a database and Σ a primary-key constraint. Then $\mathbf{db} \not\models \Sigma$ is tantamount to saying that there exist two distinct facts in some relation of $\mathbf{db}$ that share the same primary key attributes. A *block* of a relation is a maximal set of tuples that agree on all primary-key attributes. Whenever a database $\mathbf{db}$ is understood, we denote $R(\underline{\vec{c}}, *)$ as the block in $\mathbf{db}$ containing all tuples with primary-key value $\vec{c}$ in relation R. A repair of $\mathbf{db}$ can therefore be obtained by selecting exactly one tuple from $R(\underline{\vec{c}}, *)$, for each relation name R in $\mathbf{db}$ and primary-key value $\vec{c}$ in R. It is easy to see that all repairs of $\mathbf{db}$ have the same size, i.e., contains the same number of tuples.

Example 2.4. Consider the **Company** database. Suppose that the tables Employee, Manager, and Contact, each with primary keys **employee_id**, **office_city**, and **office_city** respectively. For example, it is thus asserted that in the Employee table, no two facts shall share the same employee_id, or equivalently, the following first-order sentence shall hold:

$$\forall x, y, z, y', z' : (\mathsf{Employee}(\underline{x}, y, z) \land \mathsf{Employee}(\underline{x}, y', z') \rightarrow y = y' \land z = z').$$

Note that **Company** does not satisfy the primary-key constraint, since there are two distinct tuples $\mathsf{Employee}(\underline{0011}, \text{Boston}, \text{Boston})$ and $\mathsf{Employee}(\underline{0011}, \text{Chicago}, \text{New York})$ in **Company** that agree on the primary-key attribute value 0011.

The **Company** database contains (among others) nonempty blocks

$$\begin{aligned}\mathsf{Employee}(\underline{0011},*) = \{&\mathsf{Employee}(\underline{0011}, \text{Boston}, \text{Boston}),\\ &\mathsf{Employee}(\underline{0011}, \text{Chicago}, \text{New York}),\\ &\mathsf{Employee}(\underline{0011}, \text{Chicago}, \text{Chicago})\},\end{aligned}$$

$$\mathsf{Manager}(\underline{\text{Chicago}},*) = \{\mathsf{Manager}(\underline{\text{Chicago}}, 0022, 2020)\},$$

and

$$\mathsf{Contact}(\underline{\text{LA}},*) = \{\mathsf{Contact}(\underline{\text{LA}}, 0034), \mathsf{Contact}(\underline{\text{LA}}, 0037)\}.$$

A repair of **Company** is highlighted in gray. □

2.3 Computational Complexity

For the sake of brevity, we introduce only the concepts in computational complexity used in this dissertation and refer the unfamiliar readers to Chapter 2.3 of [Lib04] for more details.

Let $\mathcal{L}$ be a language over a finite set of alphabet. A *decision problem* P is a function $P : \mathcal{L} \to \{0, 1\}$. Informally, the class $\mathcal{L}$ defines the class of valid inputs to the problem P, and for each valid input $x \in \mathcal{L}$, its output is either 0 (false) or 1 (true). An input $x \in \mathcal{L}$ is a "yes"-instance for P if $P(x) = 1$, or otherwise a "no"-instance.

The class **PTIME** denotes the class of decision problems that can be solved by a deterministic Turing machine in polynomial-time. The class **NP** denotes the class of decision problems that can be solved by a non-deterministic Turing machine in polynomial-time, or equivalently, whose "yes"-certificate can be verified in **PTIME**. A decision problem is in the class **coNP** if its complement is in **NP**.

Example 2.5. Consider the decision problem SAT: Given a CNF formula φ, does φ has a satisfying assignment? The complement of SAT is the UNSAT problem: Given a CNF formula φ, is φ not satisfiable by all assignments?

We have that SAT is in **NP**, since a satisfying assignment (a "yes"-certificate to SAT) can be verified in **PTIME** by evaluating the assignment directly in **PTIME**. UNSAT is the complement of SAT since for every CNF formula φ, φ is a "yes"-instance for SAT if and only if φ is a "no"-instance for UNSAT. This shows that UNSAT is in **coNP**. □

We use **NL** to denote the class of decision problems that are decidable by a non-deterministic Turing machine using logarithm space. The class **L** represents the class of decision problems decidable by a deterministic Turing machine using logarithm space. Immerman–Szelepcsényi theorem states that **NL** is closed under complement [Imm88, Sze88]. A decision problem P is in **FO** if there exists a first-order sentence ψ of constant size such that x is a "yes"-instance for P if and only if $x \models \psi$. The class **FO** coincides with the class $\mathbf{AC}^0$.

Let P be a problem that concerns a database **db** and a query q. The *query complexity* of the problem P is the computational complexity of P in terms of the size of the query while treating the size of the database as a constant; the *data complexity* of the problem P is the computational complexity of P in terms of the size of the database while treating the size of the query as a constant; and the *combined complexity* of the problem P is the computational complexity of P in terms of both the query size and the database size.

A fundamental problem in database theory is the query evaluation problem (or model checking):

Problem: QueryEval

Input: a Boolean query q and a database **db**

Output: does $\mathbf{db} \models q$ hold?

If q is a BCQ, then QueryEval is **NP**-complete in both combined complexity and query complexity via an easy reduction from 3-COLORABILITY [AHV95]. Counterintuitive as this result is, commercial databases are in general efficient and effective since the queries often asked in our daily lives do not involve a large number of tables and attributes. For any query q, QueryEval is in **FO** (or $\mathbf{AC}^0$) in data complexity, since the size of the query q is considered a constant. A long line of literature has been dedicated to efficient query processing algorithms [HW23, ZDK23, ZFOK23, KNS17, WWS23, NPRR12] and its hardness [FKZ23].

2.4 Consistent Query Answering (CQA)

For every Boolean query q and integrity constraint Σ, the *consistent query answering (CQA)* problem, denoted by $\mathsf{CERTAINTY}(q, \Sigma)$, is the following decision problem:

Problem: $\mathsf{CERTAINTY}(q, \Sigma)$

Input: a database instance **db**

Output: does $\mathbf{r} \models q$ hold for every $\mathbf{r} \in \mathsf{repairs}(\mathbf{db}, \Sigma)$?

If Σ is a primary-key constraint, we often drop Σ from the notation and simply use the notation $\mathsf{CERTAINTY}(q)$, where every atom in q would be of the form $R(\underline{\vec{x}}, \vec{y})$ to encode the primary key constraint on relation R.

If q is non-Boolean, the problem $\mathsf{CERTAINTY}(q, \Sigma)$ essentially asks to compute the following set:

$$\bigcap_{\mathbf{r} \in \mathsf{repairs}(\mathbf{db}, \Sigma)} q(\mathbf{r}).$$

For the study of computational complexity, we often consider the decision version of CQA since it is both a simplification of the problem, and that we may *reduce* the non-Boolean CQA problem to the Boolean one, to be formally stated in Lemma 3.4 of Chapter 3.

In this dissertation, we study the *data complexity* of $\mathsf{CERTAINTY}(q)$, i.e., the size of the query q is assumed to be a fixed constant. It is easy to see that $\mathsf{CERTAINTY}(q)$ is in **coNP**, since a "no"-certificate (i.e., a repair $\mathbf{r}$ of $\mathbf{db}$ such that $\mathbf{r} \not\models q$) can be verified in **PTIME**.

The problem $\mathsf{CERTAINTY}(q)$ has a *first-order rewriting* if there is another query q' such that evaluating q' on the input database $\mathbf{db}$ would return the answers of $\mathsf{CERTAINTY}(q)$. In other words, executing q' directly on the inconsistent database *simulates* computing the original query q over all possible repairs.

Example 2.6. Recall that in Example 2.2, the query q returns $\{0011, 0022, 0034\}$ on the inconsistent database **Company**. For $\mathsf{CERTAINTY}(q)$ however, the only output is 0022: for any repair that contains the tuples $\mathsf{Employee}(\underline{0011}, \mathsf{Boston}, \mathsf{Boston})$ and $\mathsf{Manager}(\underline{\mathsf{Boston}}, 0011, 2021)$, neither 0011 nor 0034 would be returned by q; and in any repair, 0022 is returned by q with the following crucial observation:

Regardless of which tuple in $\mathsf{Employee}(\underline{0022}, *)$ *the repair contains, both offices are present in the* Manager *table and both managers in Chicago and New York offices started in 2020.*

Based on the observation, it is sufficient to solve $\mathsf{CERTAINTY}(q)$ by running the following single SQL query, called an **FO**-rewriting of $\mathsf{CERTAINTY}(q)$.

```
SELECT E.employee_id FROM Employee E EXCEPT
SELECT E.employee_id FROM Employee E
WHERE E.office_city NOT IN (
     SELECT M.office_city FROM Manager EXCEPT
     SELECT M.office_city FROM Manager
     WHERE M.start_year <> 2020)
```

□

2.5 Datalog

A Datalog program P is a finite set of rules of the form (2.1), with the extension that negated atoms can be used in rule bodies. A rule can be interpreted as a logical implication: if the body is true, then so is the head of the rule. We assume that rules are always safe, meaning that every variable occurring in the rule must also occur in a non-negated atom of the rule body. A relation belongs to the intensional database (IDB) if it is defined by rules, i.e., if it appears as the head of some rule; otherwise it belongs to the extensional database (EDB), i.e., it is a stored table.

Datalog naturally allows for recursion, by allowing the head predicate of some rule to appear in the body of other rules (or even in its own body!). Given a Datalog program P, the *dependency graph* of P is the directed graph in which the set of all rules in P forms the vertex set, and there is an edge between two rules r_1 and r_2 if the head predicate of r_1 appears in the body of r_2. The Datalog program is *recursive* if its dependency graph is acyclic, or otherwise *non-recursive*. A Datalog program is *linear* if every rule contains at most one IDB in its body.

Example 2.7. The following Datalog program computes the transitive closure T of a directed graph with an edge set E:

$$T(x,y) \text{ :- } E(x,y). \tag{2.2}$$

$$T(x,z) \text{ :- } T(x,y), E(y,z). \tag{2.3}$$

Here $E(x,y)$ is an EDB and both $T(x,y)$ and $T(x,z)$ are IDBs. This program is recursive since the predicate name T appears both in the head and the body of rule (2.3). □

Datalog can be extended with stratified negation [AG94]. This means that the rules of a Datalog program P can be partitioned into $(P_1, P_2, \ldots, P_n)$ such that the rule body of a rule in P_i uses only IDB predicates defined by rules in some P_j with $j < i$. Here, it is understood that all rules with the same head predicate belong in the same partition. Multiple Datalog engines have been developed and analyzed [FZZ+19, FMK22, KNP+22b, Fan22, SYI+16].

Chapter 3

LinCQA: Linear Time Rewritings for CQA

山重水複疑無路
柳暗花明又一村
—陸游 《遊山西村》

Practical research on CQA systems have paralleled the advancements in the theoretical breakthroughs in CQA. In this Chapter, we present LinCQA, a CQA system that produces first-order rewritings for CQA with linear running time guarantees.

Given the problem definition of CQA, the most straightforward systems would implement the "brute-force" algorithm by inspecting all possible repairs as efficiently as possible. A notable system is EQUIP [KPT13], which requires Integer Programming solvers to efficiently explore all possible repairs. CAvSAT [DK21b, DK19] is a recent system that reduces CQA to SAT solvers, and has extended capabilities to deal with queries with aggregations. The crux in both systems rely on an efficient encoding of the integrity constraints into their respective solvers. Thanks to the vast expressibility of both BIP and SAT solvers, EQUIP and CAvSAT have the potential to handle not only primary key constraints, but in general any denial constraint. These systems are often built on top of an existing database system housing the inconsistent data, reduce the CQA problem to a solver instance, and emit the final consistent answers. While this allows for a wide usecase for these systems, there are some drawbacks. First, even for queries joining two tables, the running time of those systems could be prohibitive since the sizes of the solver instances are often proportional to the sizes of the inconsistent database. More crucially, certain CQA instances could actually admit first-order rewritings, and those systems would blindly reducing certain computationally easy CQA instances to relatively expensive computational problems like BIP and SAT.

In light of those drawbacks, another line of research naturally focuses on producing efficient first-order rewritings for the rewritable CQA instances. Compared to the systems using solvers discussed above, these systems are often a light-weight SQL-to-SQL translator that can be added directly to the existing DBMS seamlessly. These systems are also often data-agnostic and easy to maintain and optimize, since it is sufficient to produce the first-order rewritings based on only the schema information and the query and one can easily verify and optimize the output query for each individual use cases. The earliest such system is ConQuer developed by Fuxman and Miller [FFM05], which can produce a first-order rewriting for every SQL query in their class $\mathcal{C}_{\mathsf{forest}}$. Conquesto [KJL$^+$20] targets *every* **FO**-rewritable SJF SPJ queries by producing the rewriting in non-recursive Datalog. However, it is often challenging task to identify a wide-class of queries that admit first-order rewritings and provide running time guarantees over the produced rewritings.

Contributions. To address the above observed issues, we make the following contributions:

Theory & Algorithms. We identify a subclass of acyclic Boolean join queries that captures a wide range of queries commonly seen in practice for which we can produce **FO**-rewritings with a *linear running time guarantee* (Section 3.2). This class subsumes all acyclic Boolean queries in $\mathcal{C}_{\mathsf{forest}}$. For consistent databases, Yannakakis' algorithm [BFMY83] evaluates acyclic Boolean join queries in linear time in the size of the database. Our algorithm shows that even when inconsistency is introduced with respect to primary key constraints, the consistent answers of many acyclic Boolean join queries can still be computed in linear time, exhibiting no overhead to Yannakakis' algorithm. Our algorithm for the acyclic Boolean join queries can be extended to handle acyclic SPJ queries with projections. Our technical treatment follows Yannakakis' algorithm by considering a rooted join tree with an additional annotation of the **FO**-rewritability property, called a *pair-pruning join tree (PPJT)*. Our algorithm follows the pair-pruning join tree to compute the consistent answers and degenerates to Yannakakis' algorithm if the database has no inconsistencies.

Implementation. We implement our algorithm in LinCQA (<u>Lin</u>ear <u>C</u>onsistent <u>Q</u>uery <u>A</u>nswering) [1], a system prototype that produces an efficient and optimized rewriting in both SQL and non-recursive Datalog with negation (Section 3.3).

Evaluation. We perform an extensive experimental evaluation comparing LinCQA to the other state-of-the-art CQA systems. Our findings show that *(i)* a properly implemented rewriting can significantly outperform a generic CQA system (e.g., CAvSAT); *(ii)* LinCQA achieves the best overall performance throughout all our experiments under different inconsistency scenarios; and *(iii)* the strong theoretical guarantees of LinCQA translate to a significant performance gap for worst-case database instances. LinCQA often outperforms other CQA systems, in several cases by orders of magnitude on both synthetic and real-world workloads. We also demonstrate that CQA can be an effective approach even for real-world datasets of very large scale (∼400GB), which, to the best of our knowledge, have not been tested before.

[1] https://github.com/xiatingouyang/LinCQA/

3.1 Background

In this section, we define some additional notations used in this chapter.

Acyclic queries and join trees. Let q be a CQ. A *join tree* of q is an undirected tree whose nodes are the atoms of q such that for every two distinct atoms R and S, their common variables occur in all atoms on the unique path between R and S in the tree. A CQ q is *acyclic*[2] if it has a join tree. If τ is a subtree of a join tree of a query q, we will denote by q_τ the query whose atoms are the nodes of τ. Whenever R is a node in an undirected tree τ, then (τ, R) denotes the rooted tree obtained by choosing R as the root of the tree.

Example 3.1. The join tree of the query q in Example 2.2 has a single edge between the atoms $\mathsf{Employee}(\underline{x}, y, z)$ and $\mathsf{Manager}(\underline{y}, w, 2020)$. $\square$

Attack graphs. Let q be an acyclic, self-join-free BCQ with join tree τ. We define $\mathcal{K}(q)$ as the set of all functional dependencies of the form $\mathsf{key}(F) \rightarrow \mathsf{vars}(F)$ for every atom F in q.

$$\mathcal{K}(q) := \{\mathsf{key}(F) \rightarrow \mathsf{vars}(F) \mid F \in q\}.$$

Following [KW17], for every atom $F \in q$, we define $F^{+,q}$ as the set of all variables in q that are functionally determined by $\mathsf{key}(F)$ with respect to all functional dependencies of the form $\mathsf{key}(G) \rightarrow \mathsf{vars}(G)$ with $G \in q \setminus \{F\}$. Formally,

$$F^{+,q} := \{x \in \mathsf{vars}(q) \mid \mathcal{K}(q \setminus \{F\}) \models \mathsf{key}(F) \rightarrow x\}.$$

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 ($F \neq G$) if there exists a sequence $F_0, F_1, \ldots, F_n$ of (not necessarily distinct) atoms of q such that

- $F_0 = F$ and $F_n = G$; and
- for all $i \in \{0, \ldots, n-1\}$, $\mathsf{vars}(F_i) \cap \mathsf{vars}(F_{i+1}) \nsubseteq F^{+,q}$.

A directed edge from F to G in the attack graph of q is also called an *attack* from F to G. The foregoing definitions extend to queries with free variables: it is sufficient and correct to treat free variables as if they were constants. An atom without incoming edges in the attack graph is called *unattacked*. The attack graph of q is used to determine the data complexity of $\mathsf{CERTAINTY}(q)$: the attack graph of q is acyclic if and only if $\mathsf{CERTAINTY}(q)$ is in **FO** [Wij12, KW17].

[2]Throughout this section, whenever we say that a CQ is acyclic, we mean acyclicity as defined in [BFMY83], a notion that today is also known as α-acyclicity, to distinguish it from other notions of acyclicity.

Example 3.2. For the query $q(x)$ in Example 2.2, y is the only variable shared by atoms $\mathsf{Employee}$ and $\mathsf{Manager}$. The attack graph contains a directed edge from $\mathsf{Employee}$ to $\mathsf{Manager}$ because $y \notin \mathsf{Employee}^{+,q}$. Conversely, $\mathsf{Manager}$ does not attack $\mathsf{Employee}$ since $y \in \mathsf{Manager}^{+,q}$. Since the attack graph of $q(x)$ is acyclic, it follows that $\mathsf{CERTAINTY}(q)$ is in **FO**, as witnessed by the **FO**-rewriting in Example 2.6. □

3.2 A Linear-Time Rewriting

Before presenting our linear-time rewriting for $\mathsf{CERTAINTY}(q)$, we first provide a motivating example. Consider the following query on the **Company** database shown in Figure 2.1:

Is there an office whose contact person works for the office and, moreover, manages the office since 2020?

This query can be expressed by the following CQ:

$$q^{\mathsf{ex}}() \text{ :- } \mathsf{Employee}(\underline{x}, y, z), \mathsf{Manager}(\underline{y}, x, 2020), \mathsf{Contact}(\underline{y}, x).$$

To the best of our knowledge, the most efficient running time for $\mathsf{CERTAINTY}(q^{\mathsf{ex}})$ guaranteed by existing systems is quadratic in the input database size, denoted N. The problem $\mathsf{CERTAINTY}(q^{\mathsf{ex}})$ admits an **FO**-rewriting by the classification theorem in [KOW21]. However, the non-recursive Datalog rewriting of $\mathsf{CERTAINTY}(q^{\mathsf{ex}})$ produced by Conquesto contains Cartesian products between two tables, which means that it runs in $\Omega(N^2)$ time in the worst case. Also, since q^{ex} is not in $\mathcal{C}_{\mathsf{forest}}$, ConQuer cannot produce an **FO**-rewriting. Both EQUIP and CAvSAT solve the problem through Integer Programming or SAT solvers, which can take exponential time. One key observation is that q^{ex} requires a primary-key to primary-key join and a non-key to non-key join at the same time. As will become apparent in our technical treatment in Section 3.2.2, this property allows us to solve $\mathsf{CERTAINTY}(q^{\mathsf{ex}})$ in $O(N)$ time, while existing CQA systems will run in more than linear time.

The remainder of this section is organized as follows. In Section 3.2.1, we introduce the notion of pair-pruning join tree (PPJT). In Section 3.2.2, we consider every Boolean query q having a PPJT and present a novel linear-time non-recursive Datalog program for $\mathsf{CERTAINTY}(q)$ (Theorem 3.1). Finally, we extend our result to all acyclic self-join-free CQs in Section 3.2.3 (Theorem 3.2) .

3.2.1 Pair-pruning Join Tree

Here we introduce the notion of a *pair-pruning join tree* (PPJT). We first assume that the query q is connected, and then discuss how to handle disconnected queries at the end of the section.

Recall that an atom in a self-join-free query can be uniquely denoted by its relation name. For example, we may use $\mathsf{Employee}$ as a shorthand for the atom $\mathsf{Employee}(\underline{x}, y, z)$ in q^{ex}.

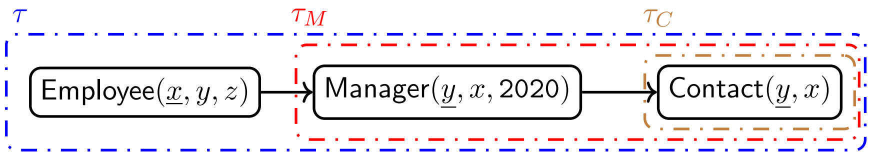

τ
τ_M
τ_C
Employee($\underline{x}, y, z$)
Manager($\underline{y}, x, 2020$)
Contact($\underline{y}, x$)

Figure 3.1: A pair-pruning join tree (PPJT) of the query q^{ex}.

Definition 3.1 (PPJT)**.** Let q be an acyclic self-join-free BCQ. Let τ be a join tree of q and R a node in τ. The tree (τ, R) is a *pair-pruning join tree (PPJT)* of q if for any rooted subtree (τ', R') of (τ, R), the atom R' is unattacked in $q_{\tau'}$.

Example 3.3. For the join tree τ in Figure 3.1, the rooted tree $(\tau, \mathsf{Employee})$ is a PPJT for q^{ex}. The atom $\mathsf{Employee}(\underline{x}, y, z)$ is unattacked in q. For the child subtree $(\tau_M, \mathsf{Manager})$ of $(\tau, \mathsf{Employee})$, the atom $\mathsf{Manager}(\underline{y}, x, 2020)$ is also unattacked in the following subquery

$$q^{\mathsf{ex}}_{\tau_M}() \text{ :- } \mathsf{Manager}(\underline{y}, x, 2020), \mathsf{Contact}(\underline{y}, x).$$

Finally, for the subtree $(\tau_C, \mathsf{Contact})$, the atom $\mathsf{Contact}(\underline{y}, x)$ is also unattacked in the corresponding subquery $q^{\mathsf{ex}}_{\tau_C}()$:- $\mathsf{Contact}(\underline{y}, x)$. Hence $(\tau, \mathsf{Employee})$ is a PPJT of q^{ex}. $\square$

Which queries admit a PPJT? As we show next, having a PPJT is a sufficient condition for the existence of an **FO**-rewriting.

Proposition 3.1. *Let q be an acyclic self-join-free BCQ. If q has a PPJT, then* $\mathsf{CERTAINTY}(q)$ *admits an* **FO***-rewriting.*

Proof. Suppose, for the sake of contradiction, that the attack graph is not acyclic. Then there must be two atoms R, S such that $R \overset{q}{\rightsquigarrow} S$ and $S \overset{q}{\rightsquigarrow} R$ by Lemma 3.6 of [KW17]. Let (τ, T) be the PPJT for q, and let (τ', U) be the smallest subtree of (τ, T) that contains both S and R (it may be that $U = R$ or $U = S$). The first observation is that in the subquery $q_{\tau'}$ it also holds that R attacks S and vice versa. Moreover, since (τ', U) is the smallest possible subtree, the unique path that connects R and S must go through the root U. We now distinguish two cases:

- If $U = R$, then S attacks the root of the subtree q', a contradiction to the PPJT definition.
- If $U \neq R$, then the unique path from R to S goes through U. Since R must attack every atom in that path by Lemma 4.9 of [Wij12], it must also attack U, a contradiction as well.

The proof is now complete by the classification theorem of [KOW21]. $\square$

We note that not all acyclic self-join-free BCQs with an acyclic attack graph have a PPJT, as demonstrated in the next example.

Example 3.4. Let $q()$:- $R(\underline{x,w},y), S(\underline{y,w},z), T(\underline{w},z)$. The attack graph of q is acyclic. The only join tree τ of q is the path $R - S - T$. However, neither (τ, R) nor (τ, S) is a PPJT for q since R and S are attacked in q; and (τ, T) is not a PPJT since in its subtree (τ', S), S is attacked in the subquery that contains R and S. $\square$

Fuxman and Miller [FM07] identified a large class of self-join-free CQs, called $\mathcal{C}_{\mathsf{forest}}$, that includes most queries with primary-key-foreign-key joins, path queries, and queries on a star schema, such as found in SSB and TPC-H [OOCR09, PF00]. This class covers most of the SPJ queries seen in practical settings. In view of this, the following proposition is of practical significance.

Proposition 3.2. *Every acyclic BCQ in $\mathcal{C}_{\mathsf{forest}}$ has a PPJT.*

Proof. Let q be a query in $\mathcal{C}_{\mathsf{forest}}$ and let G be the join graph of q as in Definition 6 of [FM07]. In particular, (i) the vertices of G are the atoms of q, and (ii) there is an arc from R to S if $R \neq S$ and there is some variable $w \in \mathsf{vars}(S)$ such that $w \in \mathsf{vars}(R) \setminus \mathsf{key}(R)$. By the definition of $\mathcal{C}_{\mathsf{forest}}$, G is a directed forest with connected components $\tau_1, \tau_2, \ldots, \tau_n$, where the root atoms are $R_1, R_2, \ldots, R_n$ respectively.

Claim 1: each τ_i is a join tree. Suppose for the sake of contradiction that τ_i is not a join tree. Then there exists a variable w and two non-adjacent atoms R and S in τ_i such that $w \in \mathsf{vars}(R)$, $w \in \mathsf{vars}(S)$, and for any atom T_i in the unique path $R - T_1 - \cdots - T_k - S$, we have $w \notin \mathsf{vars}(T_i)$. We must have $w \in \mathsf{key}(R)$ and $w \in \mathsf{key}(S)$, or otherwise there would be an arc between R and S, a contradiction. From the property $\mathcal{C}_{\mathsf{forest}}$, it also holds that no atom in the tree receives arcs from two different nodes. Hence, there is either an arc (T_1, R) or (T_k, S). Without loss of generality, assume there is an arc from T_1 to R. Then, since all nonkey-to-key joins are full, $w \in \mathsf{vars}(T_1)$, a contradiction to our assumption.

Claim 2: the forest $\tau_1 \cup \cdots \cup \tau_n$ can be extended to a join tree τ of q. To show this, we will show that $\tau_1 \cup \cdots \cup \tau_n$ corresponds to a partial join tree as constructed by the GYO ear-removal algorithm. Indeed, suppose that atom T is a child of atom T' in τ_i. Then, T was an ear while constructing τ_i for q_{τ_i}, with T' as its witness. Recall that this means that if a variable x is not exclusive in T, then $x \in T'$. We will show that this is a valid ear removal step for q as well. Indeed, consider an exclusive variable x in T for q_{τ_i} that does not remain exclusive in q. Then, x occurs in some other tree τ_j. We will now use the fact that, by Lemma 2 of [FM07], if τ_i and τ_j share a variable x, then x can only appear in the root atoms R_i and R_j. This implies that x appears at the root of τ_i, and hence at T' as well, a contradiction.

We finally claim that (τ, R_1) is a PPJT for q. By construction, τ is a join tree. Next, consider any two adjacent atoms R and S in τ such that R is a parent of S in (τ, R_1). Let p be any connected subquery of q containing R and S. It suffices to show that S does not attack R in p. If R and S are both root nodes of some τ_i and τ_j, we must have $\mathsf{vars}(R) \cap \mathsf{vars}(S) \subseteq \mathsf{key}(S) \subseteq S^{+,p}$, and thus S does not attack R in p. If R and S are in the same join tree τ_i, since there is no arc from S to R, all nonkeys of S are not present in R, and thus $\mathsf{vars}(R) \cap \mathsf{vars}(S) = \mathsf{vars}(R) \cap \mathsf{key}(S) \subseteq \mathsf{key}(S) \subseteq S^{+,p}$. Hence, there is no attack from S to R as well. $\square$

Furthermore, it is easy to verify that, unlike $\mathcal{C}_{\mathsf{forest}}$, PPJT captures *all* **FO**-rewritable self-join-free SPJ queries on two tables, a.k.a. binary joins. For example, the binary join q_5 in Section 3.4.2 admits a PPJT but is not in $\mathcal{C}_{\mathsf{forest}}$.

How to find a PPJT. For any acyclic self-join-free BCQ q, we can check whether q admits a PPJT via a brute-force search over all possible join trees and roots. If q involves n relations, then there are at most n^{n-1} candidate rooted join trees for PPJT (n^{n-2} join trees and for each join tree, n choices for the root). For the data complexity of $\mathsf{CERTAINTY}(q)$, this exhaustive search runs in constant time since we assume n is a constant. In practice, the search cost is acceptable for most join queries that do not involve too many tables.

Proposition 3.3 shows that the foregoing brute-force search for q can be optimized to run in polynomial time when q has an acyclic attack graph and, when expressed as a rule, does not contain two distinct body atoms $R(\underline{\vec{x}}, \vec{y})$ and $S(\underline{\vec{u}}, \vec{w})$ such that every variable occurring in $\vec{x}$ also occurs in $\vec{u}$. Most queries we observe and used in our experiments fall under this category.

Proposition 3.3. *Let q be an acyclic self-join-free BCQ whose attack graph is acyclic. If for all two distinct atoms $F, G \in q$, neither of $\mathsf{key}(F)$ or $\mathsf{key}(G)$ is included in the other, then q has a PPJT that can be constructed in quadratic time in the number of atoms in q.*

Proof. Let q be a self-join-free Boolean conjunctive query with an acyclic attack graph. Let τ be a join tree for q (thus q is α-acyclic). Assume the following hypothesis:

Hypothesis of Disjoint Keys: for all atoms $G, H \in q$, $G \neq H$, we have that $\mathsf{key}(G)$ and $\mathsf{key}(H)$ are not comparable by set inclusion.

We show, by induction on $|q|$, that $\mathsf{CERTAINTY}(q)$ is in linear time. For the basis of the induction, $|q| = \emptyset$, it is trivial that $\mathsf{CERTAINTY}(q)$ is in linear time. For the induction step, let $|q| \geq 1$. Let F be an unattacked atom of q. Let (τ, F) be a join tree of q with root F. Let $F_1, \ldots, F_n$ be the children of F in (τ, F) with subtrees $\tau_1, \tau_2, \ldots, \tau_n$.

Let $i \in \{1, \ldots, n\}$. We claim that q_{τ_i} has an acyclic attack graph. Assume for the sake of contradiction that the attack graph of q_{τ_i} has a cycle, and therefore has a cycle of size 2. Then there are $G, H \in q_{\tau_i}$ such that $G \overset{q_{\tau_i}}{\rightsquigarrow} H \overset{q_{\tau_i}}{\rightsquigarrow} G$. From the *Hypothesis of Disjoint Keys*, it follows $G \overset{q}{\rightsquigarrow} H \overset{q}{\rightsquigarrow} G$, contradicting the acyclicity of q's attack graph.

We claim the following:

$$\text{for every } G \in q_{\tau_i}, \ \mathbf{vars}(G) \cap \mathbf{vars}(F) \subseteq \mathsf{key}(G). \tag{3.1}$$

This claim follows from the *Hypothesis of Disjoint Keys* and the assumption that F is unattacked in q's attack graph.

It suffices to show that there is an atom $F_i' \in q_{\tau_i}$ (possibly $F_i' = F_i$) such that

1. F_i' is unattacked in the attack graph of q_{τ_i}; and
2. $\mathsf{vars}(q_{\tau_i}) \cap \mathbf{vars}(F) \subseteq \mathsf{key}(F_i')$.

We distinguish two cases:

Case that F_i is unattacked in the attack graph of q_{τ_i}. Then we can pick $F_i' := F_i$.

Case that F_i is attacked in the attack graph of q_{τ_i}. We can assume an atom G such that $G \overset{q_{\tau_i}}{\rightsquigarrow} F_i$. Since $G \overset{q}{\not\rightsquigarrow} F$, by the *Hypothesis of Disjoint Keys*, it must be that $\mathbf{vars}(F_i) \cap \mathbf{vars}(F) \subseteq \mathsf{key}(G)$. Then from $\mathsf{vars}(q_{\tau_i}) \cap \mathbf{vars}(F) \subseteq \mathbf{vars}(F_i)$, it follows $\mathsf{vars}(q_{\tau_i}) \cap \mathbf{vars}(F) \subseteq \mathsf{key}(G)$. If G is unattacked in the attack graph of $q_{\tau_i} i$, then we can pick $F_i' := G$. Otherwise we repeat the same reasoning (with G playing the role previously played by F_i). This repetition cannot go on forever since the attack graph of q_{τ_i} is acyclic.

This concludes the proof. □

Main Result. We previously showed that the existence of a PPJT implies an **FO**-rewriting that computes the consistent answers. Our main result shows that it also leads to an efficient algorithm that runs in linear time.

Theorem 3.1. *Let q be an acyclic self-join-free BCQ that admits a PPJT, and* **db** *be a database instance of size N. Then, there exists an algorithm for* $\mathsf{CERTAINTY}(q)$ *that runs in time $O(N)$.*

It is worth contrasting our result with Yannakakis' algorithm, which computes the result of any acyclic BCQ also in linear time $O(N)$ [Yan81]. Hence, the existence of a PPJT implies that computing $\mathsf{CERTAINTY}(q)$ will have the same asymptotic complexity.

Disconnected CQs. Every disconnected BCQ q can be written as $q = q_1, q_2, \ldots, q_n$ where $\mathsf{vars}(q_i) \cap \mathsf{vars}(q_j) = \emptyset$ for $1 \leq i < j \leq n$ and each q_i is connected. If each q_i has a PPJT, then $\mathsf{CERTAINTY}(q)$ can be solved by checking whether the input database is a "yes"-instance for each $\mathsf{CERTAINTY}(q_i)$, by the following Lemma.

Lemma 3.1. *Let $q = q_1 \cup q_2 \cup \cdots \cup q_k$ be a Boolean conjunctive query such that for all $1 \leq i < j \leq k$, $\mathsf{vars}(q_i) \cap \mathsf{vars}(q_j) = \emptyset$. Then, the following are equivalent for every database instance* $\mathbf{db}$*:*

1. $\mathbf{db}$ *is a "yes"-instance for* $\mathsf{CERTAINTY}(q)$*; and*
2. *for each $1 \leq i \leq k$,* $\mathbf{db}$ *is a "yes"-instance for* $\mathsf{CERTAINTY}(q_i)$*.*

Proof. We give the proof for $k = 2$. The generalization to larger k is straightforward.

$\boxed{1 \Longrightarrow 2}$ Assume that (1) holds true. Then each repair $\mathbf{r}$ of $\mathbf{db}$ satisfies q, and therefore satisfies both q_1 and q_2. Therefore, $\mathbf{db}$ is a "yes"-instance for both $\mathsf{CERTAINTY}(q_1)$ and $\mathsf{CERTAINTY}(q_2)$.

$\boxed{2 \Longrightarrow 1}$ Assume that (2) holds true. Let $\mathbf{r}$ be any repair of $\mathbf{db}$. Then there are valuations μ from $\mathsf{vars}(q_1)$ to $\mathsf{adom}(\mathbf{db})$, and θ from $\mathsf{vars}(q_2)$ to $\mathsf{adom}(\mathbf{db})$ such that $\mu(q_1) \subseteq \mathbf{r}$ and $\theta(q_2) \subseteq \mathbf{r}$. Since $\mathsf{vars}(q_1) \cap \mathsf{vars}(q_2) = \emptyset$ by construction, we can define a valuation σ as follows, for every variable $z \in \mathsf{vars}(q_1) \cup \mathsf{vars}(q_2)$:

$$\sigma(z) = \begin{cases} \mu(z) & \text{if } z \in \mathsf{vars}(q_1) \\ \theta(z) & \text{if } z \in \mathsf{vars}(q_2) \end{cases}$$

From $\sigma(q) = \sigma(q_1) \cup \sigma(q_2) = \mu(q_1) \cup \theta(q_2) \subseteq \mathbf{r}$, it follows that $\mathbf{r}$ satisfies q. Therefore, $\mathbf{db}$ is a "yes"-instance for $\mathsf{CERTAINTY}(q)$. $\square$

3.2.2 The Rewriting Rules

We now show how to produce an efficient rewriting in Datalog and prove Theorem 3.1. In Section 3.3, we will discuss how to translate the Datalog program to SQL. Let q be an acyclic self-join-free BCQ with a PPJT (τ, R) and $\mathbf{db}$ an instance for the problem $\mathsf{CERTAINTY}(q)$: does the query q evaluate to true on every repair of $\mathbf{db}$?

Let us first revisit Yannakakis' algorithm for evaluating q on a database $\mathbf{db}$ in linear time. Given a rooted join tree (τ, R) of q, Yannakakis' algorithm visits all nodes in a bottom-up fashion. For every internal node S of (τ, R), it keeps the tuples in table S that join with every child of S in (τ, R), where each such child has been visited recursively. In the end, the algorithm returns whether the root table R is empty or not. Equivalently, Yannakakis' algorithm evaluates q on $\mathbf{db}$ by removing tuples from each table that cannot contribute to an answer in $\mathbf{db}$ at each recursive step.

Our algorithm for CQA proceeds like Yannakakis' algorithm in a bottom-up fashion. At each step, we remove tuples from each table that cannot contribute to an answer to q in at least one repair of $\mathbf{db}$. Informally, if a tuple cannot contribute to an answer in at least one repair of $\mathbf{db}$ containing it, then it cannot contribute to a consistent answer to q on $\mathbf{db}$. Specifically, given a

PPJT (τ, R) of q, to compute all tuples of each internal node S of (τ, R) that may contribute to a consistent answer, we need to "prune" the blocks of S in which there is some tuple that violates either the local selection condition on table S, or the joining condition with some child table of S in (τ, R). The term "pair-pruning" is motivated by the latter process, where we consider only one pair of tables at a time. This idea is formalized in Algorithm 1, where the procedures SELF-PRUNING and PAIR-PRUNING prune, respectively, the blocks that violate the local selection condition and the joining condition.

To ease the exposition of the rewriting, we now present both procedures in Datalog syntax. We will use two predicates for every atom S in the tree (let T be the unique parent of S in τ):

- the predicate S_{fkey} has arity equal to $|\mathsf{key}(S)|$ and collects the primary-key values of the S-table that cannot contribute to a consistent answer for q [3]; and
- the predicate S_{join} has arity equal to $|\mathsf{vars}(S) \cap \mathsf{vars}(T)|$ and collects the values for these variables in the S-table that may contribute to a consistent answer.

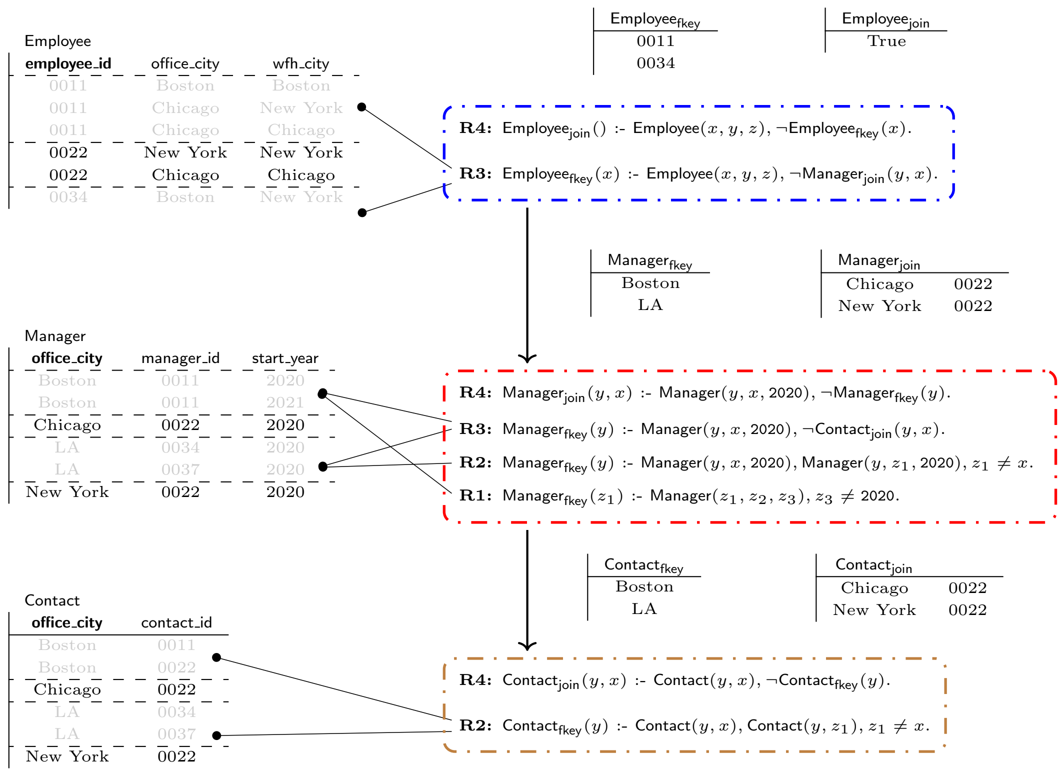
Algorithm 1: PPJT-REWRITING(τ, R)

```
   Input: PPJT $(\tau, R)$ of $q$
   Output: a Datalog program $P$ deciding $\mathsf{CERTAINTY}(q)$
1  $P := \emptyset$
2  $P := P \cup$ SELF-PRUNING$(R)$
3  foreach child node $S$ of $R$ in $\tau$ do
4      $P := P \cup$ PPJT-REWRITING$(\tau, S)$
5      $P := P \cup$ PAIR-PRUNING$(R, S)$
6  $P := P \cup$ EXIT-RULE$(R)$
7  return $P$
```

Figure 3.2 depicts how each step generates the rewriting rules for q^{ex}. We now describe how each step is implemented in detail.

SELF-PRUNING(R): Let $R(\underline{x_1, \dots x_k}, x_{k+1}, \dots, x_n)$, where x_i can be a variable or a constant. The first rule finds the primary-key values of the R-table that can be pruned because some tuple with that primary-key violates the local selection conditions imposed on R.

[3]The f in fkey is for "false key".



Employee

employee_id	office_city	wfh_city
0011	Boston	Boston
0011	Chicago	New York
0011	Chicago	Chicago
0022	New York	New York
0022	Chicago	Chicago
0034	Boston	New York

Employee$_{\mathsf{fkey}}$
0011
0034

Employee$_{\mathsf{join}}$
True

R4: $\mathsf{Employee_{join}}() \text{ :- } \mathsf{Employee}(x, y, z), \neg\mathsf{Employee_{fkey}}(x).$

R3: $\mathsf{Employee_{fkey}}(x) \text{ :- } \mathsf{Employee}(x, y, z), \neg\mathsf{Manager_{join}}(y, x).$

Manager$_{\mathsf{fkey}}$
Boston
LA

Manager$_{\mathsf{join}}$	
Chicago	0022
New York	0022

Manager

office_city	manager_id	start_year
Boston	0011	2020
Boston	0011	2021
Chicago	0022	2020
LA	0034	2020
LA	0037	2020
New York	0022	2020

R4: $\mathsf{Manager_{join}}(y, x) \text{ :- } \mathsf{Manager}(y, x, 2020), \neg\mathsf{Manager_{fkey}}(y).$

R3: $\mathsf{Manager_{fkey}}(y) \text{ :- } \mathsf{Manager}(y, x, 2020), \neg\mathsf{Contact_{join}}(y, x).$

R2: $\mathsf{Manager_{fkey}}(y) \text{ :- } \mathsf{Manager}(y, x, 2020), \mathsf{Manager}(y, z_1, 2020), z_1 \neq x.$

R1: $\mathsf{Manager_{fkey}}(z_1) \text{ :- } \mathsf{Manager}(z_1, z_2, z_3), z_3 \neq 2020.$

Contact$_{\mathsf{fkey}}$
Boston
LA

Contact$_{\mathsf{join}}$	
Chicago	0022
New York	0022

Contact

office_city	contact_id
Boston	0011
Boston	0022
Chicago	0022
LA	0034
LA	0037
New York	0022

R4: $\mathsf{Contact_{join}}(y, x) \text{ :- } \mathsf{Contact}(y, x), \neg\mathsf{Contact_{fkey}}(y).$

R2: $\mathsf{Contact_{fkey}}(y) \text{ :- } \mathsf{Contact}(y, x), \mathsf{Contact}(y, z_1), z_1 \neq x.$

Figure 3.2: The non-recursive Datalog program for evaluating $\mathsf{CERTAINTY}(q^{\mathsf{ex}})$ together with an example execution on the **Company** database in Figure 2.1. The faded-out rows denote blocks that are removed since they do not contribute to any consistent answer. The arrows denote which rules remove which blocks (some blocks can be removed by multiple rules).

Rule 3.1. If $x_i = c$ for some constant c, we add the rule

$$R_{\mathsf{fkey}}(z_1, \dots, z_k) \text{ :- } R(z_1, \dots, z_n), z_i \neq c.$$

If for some variable x_i there exists $j < i$ with $x_i = x_j$, we add the rule

$$R_{\mathsf{fkey}}(z_1, \dots, z_k) \text{ :- } R(z_1, \dots, z_n), z_i \neq z_j.$$

Here, $z_1, \dots, z_n$ are fresh distinct variables.

The second rule finds the primary-key values of the R-table that can be pruned because R joins with its parent T in the tree. The underlying intuition is that if some R-block of the input database contains two tuples that disagree on a non-key position that is used in an equality-join with T, then for every given T-tuple t, we can pick an R-tuple in that block that does not join with t. Therefore, that R-block cannot contribute to a consistent answer.

Rule 3.2. For each variable x_i with $i > k$ (so in a non-key position) such that $x_i \in \mathsf{vars}(T)$, we produce a rule

$$\begin{aligned} R_{\mathsf{fkey}}(x_1, \dots, x_k) \text{ :- } & R(x_1, \dots, x_k, x_{k+1}, \dots, x_n), \\ & R(x_1, \dots, x_k, z_{k+1}, \dots, z_k), z_i \neq x_i. \end{aligned}$$

where $z_{k+1}, \dots, z_n$ are fresh variables.

Example 3.5. The self-pruning phase on(τ_M, $\mathsf{Manager}$) produces one rule using Rule 3.1. When executed on the **Company** database, the key Boston is added to $\mathsf{Manager}_{\mathsf{fkey}}$, since the tuple (Boston, 0011, 2021) has $\mathsf{start\_year} \neq 2020$. The self-pruning phase on the PPJT (τ_C, $\mathsf{Contact}$) produces one rule using Rule 3.2 (here x is the non-key join variable). Hence, the keys Boston and LA will be added to $\mathsf{Contact}_{\mathsf{fkey}}$. $\square$

<u>Pair-Pruning(R, S)</u>: Suppose that q contains the atoms $R(\underline{\vec{x}}, \vec{y})$ and $S(\underline{\vec{u}}, \vec{v})$, where the S-atom is a child of the R-atom in the PPJT. Let $\vec{w}$ be a sequence of distinct variables containing all (and only) variables in $\mathsf{vars}(R) \cap \mathsf{vars}(S)$. The third rule prunes all R-blocks containing some tuple that cannot join with any S-tuple to contribute to a consistent answer.

Rule 3.3. Add the rule

$$R_{\mathsf{fkey}}(\vec{x}) \text{ :- } R(\vec{x}, \vec{y}), \neg S_{\mathsf{join}}(\vec{w}),$$

where the rules for S_{join} will be defined in Rule 3.4.

The rule is safe because every variable in $\vec{w}$ occurs in $R(\vec{x}, \vec{y})$.

Example 3.6. Figure 3.2 shows the two pair-pruning rules generated (in general, there will be one pair-pruning rule for each parent-child edge in the PPJT). In both cases, the join variables are $\{y, x\}$. For the table $\mathsf{Employee}$, the rule prunes the two blocks with keys $0011, 0034$ and adds them to $\mathsf{Employee}_{\mathsf{fkey}}$. □

<u>EXIT-RULE(R)</u>: Suppose that q contains $R(\underline{\vec{x}}, \vec{y})$. If R is an internal node, let $\vec{w}$ be a sequence of distinct variables containing all (and only) the join variables of R and its parent node in τ. If R is the root node, let $\vec{w}$ be the empty vector. The exit rule removes the pruned blocks of R and projects on the variables in $\vec{w}$. If R is an internal node, the resulting tuples in the projection could contribute to a consistent answer, and will be later used for pair pruning; if R is the root, the projection returns the final result.

Rule 3.4. If R_{fkey} exists in the head of a rule, we produce the rule

$$R_{\mathsf{join}}(\vec{w}) \text{ :- } R(\vec{x}, \vec{y}), \neg R_{\mathsf{fkey}}(\vec{x}).$$

Otherwise, we produce the rule

$$R_{\mathsf{join}}(\vec{w}) \text{ :- } R(\vec{x}, \vec{y}).$$

Example 3.7. Figure 3.2 shows the three exit rules for q^{ex}—one rule for each node in the PPJT. The boolean predicate $\mathsf{Employee}_{\mathsf{join}}$ determines whether `True` is the consistent answer to the query. □

Runtime Analysis. It is easy to see that **Rule 1, 3, and 4** can be evaluated in linear time. We now argue how to evaluate **Rule 2** in linear time as well. Indeed, instead of performing the self-join on the key, it suffices to create a hash table using the primary key as the hash key (which can be constructed in linear time). Then, for every value of the key, we can easily check whether all tuples in the block have the same value at the i-th attribute.

Proof of Theorem 3.1 We now provide the proof of Theorem 3.1.

Definition 3.2. Let $\mathbf{db}$ be a database instance for $\mathsf{CERTAINTY}(q)$ and $R(\underline{\vec{x}}, \vec{y})$ an atom in q. We define the good keys of R with respect to query q and $\mathbf{db}$, denoted by $R_{\mathsf{gkey}}(q, \mathbf{db})$, as follows:

$$R_{\mathsf{gkey}}(q, \mathbf{db}) := \{\vec{c} \mid \mathbf{db} \text{ is a "yes"-instance for } \mathsf{CERTAINTY}(q_{\vec{x} \to \vec{c}})\}.$$

Let q be a self-join-free acyclic BCQ with a PPJT (τ, R). Lemma 3.2 implies that in order to solve $\mathsf{CERTAINTY}(q)$, it suffices to check whether $R_{\mathsf{gkey}}(q, \mathbf{db}) \neq \emptyset$.

Lemma 3.2. *Let q be a self-join-free acyclic BCQ with a PPJT (τ, R). Let $\mathbf{db}$ be a database instance for $\mathsf{CERTAINTY}(q)$. Then the following statements are equivalent:*

1. *$\mathbf{db}$ is a "yes"-instance for $\mathsf{CERTAINTY}(q)$; and*
2. *$R_{\mathsf{gkey}}(q, \mathbf{db}) \neq \emptyset$.*

Proof. By Lemma 4.4 in [KW17], we have that $\mathbf{db}$ is a "yes"-instance for $\mathsf{CERTAINTY}(q)$ if and only if there exists a sequence of constant $\vec{c}$ such that $\mathbf{db}$ is a "yes"-instance for $\mathsf{CERTAINTY}(q_{[\vec{x} \to \vec{c}]})$, and the latter is equivalent to $R_{\mathsf{gkey}}(q, \mathbf{db}) \neq \emptyset$ by Definition 3.2. □

Example 3.8. The atom $\mathsf{Employee}(\underline{x}, y, z)$ is unattacked in q^{ex}. Observe that for employee_id = 0022, no matter whether we choose the tuple Employee(0022, New York, New York) or the tuple Employee(0022, Chicago, Chicago) in a repair, the chosen tuple will join with some corresponding tuple in the Manager and Contact table. The query $q^{\mathsf{ex}}_{[x \to 0022]}$ will then return `True` for all repairs of database **Company**, and $0022 \in \mathsf{Employee}_{\mathsf{gkey}}(q^{\mathsf{ex}}, \mathbf{Company}) \neq \emptyset$. The **Company** database is then concluded to be a "yes"-instance for $\mathsf{CERTAINTY}(q^{\mathsf{ex}})$ by Lemma 3.2.

We remark that the converse direction also holds: if **Company** is known to be a "yes"-instance for $\mathsf{CERTAINTY}(q^{\mathsf{ex}})$, then by Lemma 3.2, the set $\mathsf{Employee}_{\mathsf{gkey}}(q^{\mathsf{ex}}, \mathbf{Company})$ must also be nonempty. □

Lemma 3.3. *Let $R(\underline{\vec{x}}, \vec{y})$ be an atom in an acyclic self-join-free BCQ q with a PPJT (τ, R). Let $\mathbf{db}$ be an instance for $\mathsf{CERTAINTY}(q)$. For every sequence $\vec{c}$ of constants, of the same length as $\vec{x}$, the following are equivalent:*

1. *$\vec{c} \in R_{\mathsf{gkey}}(q, \mathbf{db})$; and*
2. *the block $R(\underline{\vec{c}}, *)$ of $\mathbf{db}$ is non-empty and for every fact $R(\underline{\vec{c}}, \vec{d})$ in $\mathbf{db}$, the following hold:*
   - *(a) $\{R(\underline{\vec{c}}, d)\}$ satisfies the BCQ () :- $R(\underline{\vec{x}}, \vec{y})$; and*
   - *(b) for every child subtree (τ_S, S) of (τ, R), there exists $\vec{s} \in S_{\mathsf{gkey}}(q_{\tau_S}, \mathbf{db})$ such that (i) all facts $S(\underline{\vec{s}}, \vec{t})$ agree on the joining variables in $\mathsf{vars}(R) \cap \mathsf{vars}(S)$ and (ii) for every fact $S(\underline{\vec{s}}, \vec{t})$ in $\mathbf{db}$, the pair $\{R(\underline{\vec{c}}, \vec{d}),\ S(\underline{\vec{s}}, \vec{t})\}$ satisfies the BCQ () :- $R(\underline{\vec{x}}, \vec{y}), S(\underline{\vec{u}}, \vec{v})$, where $S(\underline{\vec{u}}, \vec{v})$ is the S-atom of q.*

Proof. We consider two directions.

$\boxed{\mathbf{2} \Longrightarrow \mathbf{1}}$ Here we must have $S_{\mathsf{gkey}}(q_S, \mathbf{db}) \neq \emptyset$ for all child node S of R in τ. Let r be any repair of $\mathbf{db}$ and let $R(\vec{c}, \vec{d}) \in r$. Since 2 holds, for every child node S of R, there exists a fact $S(\vec{s}, \vec{d}) \in r$ with $\vec{s} \in S_{\mathsf{gkey}}(q_S, \mathbf{db})$ and a valuation μ_S such that $R(\mu_S(\vec{x}), \mu_S(\vec{y})) = R(\vec{c}, \vec{d})$ and $S(\mu_S(\vec{u}), \mu_S(\vec{v})) = S(\vec{s}, \vec{t})$. Since r is a repair of $\mathbf{db}$ and $\vec{s} \in S_{\mathsf{gkey}}(q_S, \mathbf{db})$, there exists a valuation ξ_S such that $\xi_S(q_S) \subseteq r$ with $\xi_S(\vec{u}) = \vec{s} = \mu_S(\vec{u})$. Note that all μ_S agree on the valuation of $\mathsf{vars}(\vec{x}) \cup \mathsf{vars}(\vec{y})$, let μ be the valuation such that $R(\mu(\vec{x}), \mu(\vec{y})) = R(\mu(\vec{x_S}), \mu(\vec{y_S}))$ for all child node S of R.

Next we show that for all q_S and any variable $z \in \mathsf{vars}(R) \cap \mathsf{vars}(q_S)$, $\mu(z) = \xi_S(z)$. Since r is consistent, we must have $S(\mu_S(\vec{u}), \mu_S(\vec{v})) = S(\xi_S(\vec{u_S}), \xi_S(\vec{v_S})) \in r$. Since T is a join tree, we must have $z \in \mathsf{vars}(R) \cap \mathsf{vars}(S)$, and it follows that $\xi_S(z) = \mu_S(z) = \mu(z)$, as desired.

Then, the following valuation

$$\mu(z) = \begin{cases} \mu(z) & z \in \mathsf{vars}(R) \\ \xi_i(z) & z \in \mathsf{vars}(q_S) \setminus \mathsf{vars}(R) \\ d & z = d \text{ is constant} \end{cases}$$

is well-defined and satisfies that $\mu(q_{\vec{x} \to \vec{c}}) \subseteq r$, as desired.

$\boxed{\mathbf{1} \Longrightarrow \mathbf{2}}$ By contraposition. Assume that 2 does not hold, and we show that there exists a repair r of $\mathbf{db}$ that does not satisfy $q_{[\vec{x} \to \vec{c}]}$.

If 2a does not hold, then there exists some fact $f = R(\vec{c}, \vec{d})$ that does not satisfy $R(\underline{\vec{x}}, \vec{y})$, and any repair containing the fact f does not satisfy $q_{[\vec{x} \to \vec{c}]}$. Next we assume that 2a holds but 2b does not.

If $S_{\mathsf{gkey}}(q_S, \mathbf{db}) = \emptyset$ for some child node S of R in τ, then by monotonicity of conjunctive queries and Lemma 3.2, $\mathbf{db}$ is a "no"-instance for $\mathsf{CERTAINTY}(q_S)$, $\mathsf{CERTAINTY}(q)$ and thus $\mathsf{CERTAINTY}(q_{[\vec{x} \to \vec{c}]})$. In what follows we assume that $S_{\mathsf{gkey}}(q_S, \mathbf{db}) \neq \emptyset$ for all child node S of R.

Since 2b does not hold, there exist a fact $R(\vec{c}, \vec{d})$ and some child node S of R in τ and query q_S such that for any block $S(\vec{s}, *)$ with $\vec{s} \in S_{\mathsf{gkey}}(q_S, \mathbf{db})$, there exists a fact $S(\vec{s}, \vec{t})$ that does not join with $R(\vec{c}, \vec{d})$.

Let $\mathbf{db}' = \mathbf{db} \setminus R \setminus \{S(\vec{s}, *) \mid \vec{s} \in S_{\mathsf{gkey}}(q_S, \mathbf{db})\}$. We show that $\mathbf{db}'$ is a "no"-instance for $\mathsf{CERTAINTY}(q_S)$. Indeed, suppose otherwise that $\mathbf{db}'$ is a "yes"-instance for $\mathsf{CERTAINTY}(q_S)$, then there exists some $\vec{s}$ such that $\mathbf{db}'$ is a "yes"-instance for $\mathsf{CERTAINTY}(q_{S,[\vec{u} \to \vec{s}]})$. Note that by construction, $\vec{s} \notin S_{\mathsf{gkey}}(q_S, \mathbf{db})$. Since $\mathbf{db}' \subseteq \mathbf{db}$, we have $\mathbf{db}$ is a "yes"-instance for $\mathsf{CERTAINTY}(q_{S,[\vec{u} \to \vec{s}]})$, implying that $\vec{s} \in S_{\mathsf{gkey}}(q_S, \mathbf{db})$, a contradiction.

Consider the following repair r of $\mathbf{db}$ that contains

- $R(\vec{c}, \vec{d})$ and an arbitrary fact from all blocks $R(\vec{b}, *)$ with $\vec{b} \neq \vec{c}$;
- for each $\vec{s} \in S_{\mathsf{gkey}}(q_S, \mathbf{db})$, any fact $S(\vec{s}, \vec{t})$ that does not join with $R(\vec{c}, \vec{d})$; and
- any falsifying repair r' of $\mathbf{db}'$ for $\mathsf{CERTAINTY}(q_S)$.

We show that r does not satisfy $q_{[\vec{x} \to \vec{c}]}$. Suppose for contradiction that there exists a valuation μ with $\mu(q_{[\vec{x} \to \vec{c}]}) \subseteq r$ and $R(\mu(\vec{x}), \mu(\vec{y})) = R(\vec{c}, \vec{d}) \in r$. Let $S(\vec{s^*}, \vec{t^*}) = S(\mu(\vec{u}), \mu(\vec{v}))$, then we must have $\vec{s^*} \notin S_{\mathsf{gkey}}(q_S, \mathbf{db})$, since otherwise we would have $S(\vec{s^*}, \vec{t^*})$ joining with $R(\vec{c}, \vec{d})$ where we have $\vec{s^*} \in S_{\mathsf{gkey}}(q_S, \mathbf{db})$, a contradiction to the construction of r. Since $\vec{s^*} \notin S_{\mathsf{gkey}}(q_S, \mathbf{db})$, we would then have $\mu(q_S) \subseteq r'$, a contradiction to that r' is a falsifying repair of $\mathbf{db}'$ for $\mathsf{CERTAINTY}(q_S)$. Finally, if 2b holds, then all facts $S(\vec{s}, \vec{t})$ must agree on $\vec{w}$ since they all join with the same fact $R(\vec{c}, \vec{d})$.

The proof is now complete. □

Proof of Theorem 3.1. It suffices to present rewriting rules to compute each $R_{\mathsf{gkey}}(q, \mathbf{db})$ for each atom R in q by Lemma 3.3 , and show that these rewriting rules are equivalent to those presented in Section 3.2.2, which are shown to run in linear time. We denote R_{gk} as the Datalog predicate for $R_{\mathsf{gkey}}(q, \mathbf{db})$. It is easy to see that **Rule 1** computes all blocks of R violating item 2a of Lemma 3.3.

To compute the blocks of R violating item 2b, we denote R_{gki} as the predicate for the subset of R_{gk} that satisfies condition (i) of item 2b. For each child $S(\underline{\vec{u}}, \vec{v})$ of R in a PPJT, let $\vec{w}$ be a sequence of all variables in $\mathsf{vars}(R) \cap \mathsf{vars}(S)$. The following rules find all blocks in R that violate condition (ii) of item 2b.

$$S_{\mathsf{join}}(\vec{w}) \text{ :- } S(\underline{\vec{u}}, \vec{v}), S_{\mathsf{gki}}(\vec{u}). \tag{3.2}$$

$$R_{\mathsf{fkey}}(\vec{x}) \text{ :- } R(\vec{x}, \vec{y}) \neg S_{\mathsf{join}}(\vec{w}). \tag{3.3}$$

We then compute the predicate R_{gk} denoting $R_{\mathsf{gkey}}(q, \mathbf{db})$ with

$$R_{\mathsf{gk}}(\vec{x}) \text{ :- } R(\underline{x}, \vec{y}), \neg R_{\mathsf{fkey}}(\vec{x}). \tag{3.4}$$

If R has a parent, then we may compute all blocks in R violating condition (i) of item 2b using the following rules for every variable at the i-th position of $\vec{y}$,

$$R_{\mathsf{gk}^\neg}(\vec{x}) \text{ :- } R(\vec{x}, \vec{y}), R(\vec{x}, \vec{y'}), y_i \neq y'_i. \tag{3.5}$$

$$R_{\mathsf{gki}}(\vec{x}) \text{ :- } R_{\mathsf{gk}}(\vec{x}), \neg R_{\mathsf{gk}^\neg}(\vec{x}). \tag{3.6}$$

Now we explain why Rules (3.2) through (3.6) are equivalent to **Rule 3.2, 3.3, 3.4**. First, the head $R_{\mathsf{gk}^\neg}$ of Rule (3.5) can be safely renamed to R_{fkey} and yield **Rule 3.2**. Rule (3.3) is equivalent to **Rule 3.3**. Finally, Rules (3.2), (3.4) and (3.6) can be merged to **Rule 3.4** since to compute each S_{join}, we only need $S(\vec{u}, \vec{v})$ and S_{fkey}. Note that the Rule (3.2) for the atom R will have 0 arity if R is the root of the PPJT. □

3.2.3 Extension to Non-Boolean Queries

Let $q(\vec{u})$ be an acyclic self-join-free CQ with free variables $\vec{u}$, and $\mathbf{db}$ be a database instance. If $\vec{c}$ is a sequence of constants of the same length as $\vec{u}$, we say that $\vec{c}$ is a *consistent answer* to q on $\mathbf{db}$ if $\vec{c} \in q(\mathbf{r})$ for every repair $\mathbf{r}$ of $\mathbf{db}$. Furthermore, we say that $\vec{c}$ is a *possible answer* to q on $\mathbf{db}$ if $\vec{c} \in q(\mathbf{db})$. It can be easily seen that for CQs every consistent answer is a possible answer.

Lemma 3.4 reduces computing the consistent answers of non-Boolean queries to that of Boolean queries.

Lemma 3.4. *Let q be a CQ with free variables $\vec{u}$, and let $\vec{c}$ be a sequence of constants of the same length as $\vec{u}$. Let $\mathbf{db}$ be an database instance. Then $\vec{c}$ is a consistent answer to q on $\mathbf{db}$ if and only if $\mathbf{db}$ is a "yes"-instance for $\mathsf{CERTAINTY}(q_{[\vec{u}\to\vec{c}]})$.*

Proof. Consider both directions. First we assume that $\vec{c}$ is a consistent answer of q on $\mathbf{db}$. Let r be any repair of $\mathbf{db}$. Then there exists a valuation μ with $\mu(q) \subseteq r$ with $\mu(\vec{u}) = \vec{c}$, and hence $\mu(q_{[\vec{u}\to\vec{c}]}) = \mu(q) \subseteq r$. That is, $q_{[\vec{u}\to\vec{c}]}(r)$ is true. Hence $\mathbf{db}$ is a "yes"-instance for $\mathsf{CERTAINTY}(q_{[\vec{u}\to\vec{c}]})$. For the other direction, we assume that $\mathbf{db}$ is "yes"-instance for $\mathsf{CERTAINTY}(q_{[\vec{u}\to\vec{c}]})$. Let r be any repair of $\mathbf{db}$. Then there is a valuation μ with $\mu(q_{[\vec{u}\to\vec{c}]}) \subseteq r$. Let θ be the valuation with $\theta(\vec{u}) = \vec{c}$. Consider the valuation

$$\mu^+(z) = \begin{cases} \theta(z) & z \in \mathsf{vars}(\vec{u}) \\ \mu(z) & \text{otherwise,} \end{cases}$$

and we have $\mu^+(q) = \mu(q_{[\vec{u}\to\vec{c}]}) \subseteq r$ with $\mu^+(\vec{u}) = \theta(\vec{u}) = \vec{c}$, as desired. □

If q has free variables $\vec{u} = (u_1, u_2, \dots, u_n)$, we say that q admits a PPJT if the Boolean query $q_{[\vec{u}\to\vec{c}]}$ admits a PPJT, where $\vec{c} = (c_1, c_2, \dots, c_n)$ is a sequence of distinct constants.

We can now state our main result for non-Boolean CQs.

Theorem 3.2. *Let q be an acyclic self-join-free Conjunctive Query that admits a PPJT, and $\mathbf{db}$ be a database instance of size N. Let OUT_p be the set of possible answers to q on $\mathbf{db}$, and OUT_c the set of consistent answers to q on $\mathbf{db}$. Then:*

1. *the set of consistent answers can be computed in time $O(N \cdot |\mathsf{OUT}_p|)$; and*
2. *moreover, if q is full, the set of consistent answers can be computed in time $O(N + |\mathsf{OUT}_c|)$.*

Theorem 3.2 exhibit a resemblance with Yannakakis' algorithm, which has a running time of $O(N \cdot |\mathsf{OUT}|)$ for acyclic CQs, and a running time of $O(N + |\mathsf{OUT}|)$ for full acyclic CQs.

Proof. Our algorithm first evaluates q on $\mathbf{db}$ to yield a set S of size $|\mathsf{OUT}_p|$ in time $O(N \cdot |\mathsf{OUT}_p|)$. Here the set S must contain all the consistent answers of q on $\mathbf{db}$. By Lemma 3.4, we then return all answers $\vec{c} \in S$ such that $\mathbf{db}$ is a "yes"-instance for $\mathsf{CERTAINTY}(q_{[\vec{u} \to \vec{c}]})$, which runs in $O(N)$ by Theorem 3.1. This approach gives an algorithm with running time $O(N \cdot |\mathsf{OUT}_p|)$.

If q is full, there is an algorithm that computes the set of consistent answers even faster. The algorithm proceeds by (i) removing all blocks with at least two tuples from $\mathbf{db}$ to yield $\mathbf{db}^c$ and (ii) evaluating q on $\mathbf{db}^c$. It suffices to show that every consistent answer to q on $\mathbf{db}$ is an answer to q on $\mathbf{db}^c$. Assume that $\vec{c}$ is a consistent answer to q on $\mathbf{db}$. Consider $q_{[\vec{u} \to \vec{c}]}$, a disconnected CQ where $\vec{u}$ is a sequence of all variables in q. Its **FO**-rewriting effectively contains **Rule 1** for each atom in $q_{[\vec{u} \to \vec{c}]}$, which is equivalent to step (i), and then checks whether $\mathbf{db}^c$ satisfies $q_{[\vec{u} \to \vec{c}]}$ by Lemma B.1 of [KOW21]. By Lemma 3.4, $\mathbf{db}$ is a "yes"-instance for $\mathsf{CERTAINTY}(q_{[\vec{u} \to \vec{c}]})$, and thus the **FO**-rewriting concludes that $\mathbf{db}^c$ satisfies $q_{[\vec{u} \to \vec{c}]}$, i.e. $\vec{c}$ is an answer to q on $\mathbf{db}^c$. In our algorithm, step (i) runs in $O(N)$ and since q is full, step (ii) runs in time $O(N + |\mathsf{OUT}_c|)$. □

In particular, we show that the consistent answers of certain non-Boolean queries can be computed in time $O(N + |\mathsf{OUT}_c|)$, borrowing ideas from the free-connex acyclic queries [BDG07].

Proposition 3.4. *Let $q(\vec{u})$ be an acyclic self-join-free Conjunctive Query that admits a connected PPJT (τ, R), and $\mathbf{db}$ be a database instance of size N. If $\mathsf{vars}(\vec{u}) \subseteq \mathsf{vars}(R)$, then the consistent answers to q on $\mathbf{db}$ can be computed in time $O(N)$.*

Proof. Consider the **FO**-rewriting of $q_{[\vec{u} \to \vec{c}]}$, for an arbitrary sequence of distinct constants $\vec{c} = (c_1, c_2, \cdots, c_n)$ of the same length of $\vec{u} = (u_1, u_2, \dots, u_n)$. Let S be any arbitrary atom in q and let $q_S(\vec{v})$ be the subquery rooted at S, where $\vec{v}$ contains all free-variables in $\vec{u}$ that occur in q_S. We must have that $\mathsf{vars}(\vec{v}) \subseteq \mathsf{vars}(S)$, since $\mathsf{vars}(\vec{v}) \subseteq \mathsf{vars}(\vec{u}) \subseteq \mathsf{vars}(R)$. Therefore, it suffices to replace each c_i with u_i (a variable) in the **FO**-rewriting of $q_{[\vec{u} \to \vec{c}]}$, and replace the head of the final return rule $R_{\mathsf{join}}(\cdot)$ (an empty head) with $R_{\mathsf{join}}(\vec{u})$. Hence the rewriting still runs in linear time, and the size of the consistent answer is of size at most the size of R. □

We may thus generalize the notion of PPJT of BCQs to free-connex PPJT of CQs by considering the free variables.

Definition 3.3. Let $q(\vec{u})$ be a connected acyclic self-join-free Conjunctive Query. Suppose that $q_{[\vec{u} \to \vec{c}]}$ has n connected components $q^{(1)}_{[\vec{u_1} \to \vec{c_1}]}, \cdots, q^{(n)}_{[\vec{u_n} \to \vec{c_n}]}$. If every BCQ $q^{(i)}_{[\vec{u_i} \to \vec{c_i}]}$ has a PPJT (τ_i, R_i) with $\mathsf{vars}(R_i) \subseteq \mathsf{vars}(\vec{u})$, then $q(\vec{u})$ has a free-connex pair-pruning join tree (τ, F), where

1. $F(\vec{u})$ is a fresh atom; and
2. every rooted tree (τ_i, R_i) is a child of F in τ.

Corollary 3.1. *Let $q(\vec{u})$ be a connected acyclic self-join-free Conjunctive Query with a free-connex pair-pruning join tree. Then the consistent answers to q on $\mathbf{db}$ can be computed in time $O(N + |\mathsf{OUT}_c|)$, where OUT_c is the set of consistent answers of q in $\mathbf{db}$.*

Proof. By Proposition 3.4, the set of consistent answers A_i of each $q^{(i)}(\vec{u_i})$ can be computed in time $O(N)$ and $|A_i| = O(N)$. By Lemma 4.22 and 3.4, $\vec{c}$ is a consistent answer of $q(\vec{u})$ in $\mathbf{db}$ if and only if every $\vec{c_i}$ is a consistent answer of $q^{(i)}(\vec{u_i})$ in $\mathbf{db}$. Hence OUT_c can be computed as the full-join among all A_i, which runs in time $O(N + \mathsf{OUT}_c)$ by the Yannakakis' algorithm. □

The proof of Theorem 1.2 follows immediately from Theorem 3.2 and Corollary 3.1.

Rewriting for non-Boolean Queries. Let $\vec{c} = (c_1, c_2, \dots, c_n)$ be a sequence of fresh, distinct constants. If $q_{[\vec{u} \to \vec{c}]}$ has a PPJT, the Datalog rewriting for $\mathsf{CERTAINTY}(q)$ can be obtained as follows:

1. Produce the program P for $\mathsf{CERTAINTY}(q_{[\vec{u} \to \vec{c}]})$ using the rewriting algorithm for Boolean queries (Subsection 3.2.2).
2. Replace each occurrence of the constant c_i in P with the free variable u_i.
3. Add the rule: $\mathsf{ground}(\vec{u})$:- $\mathsf{body}(q)$.
4. For a relation T, let $\vec{u}_T$ be a sequence of all free variables that occur in the subtree rooted at T. Then, append $\vec{u}_T$ to every occurrence of T_{join} and T_{fkey}.
5. For any rule of P that has a free variable u_i that is unsafe, add the atom $\mathsf{ground}(\vec{u})$ to the rule.

Example 3.9. Consider the non-Boolean query

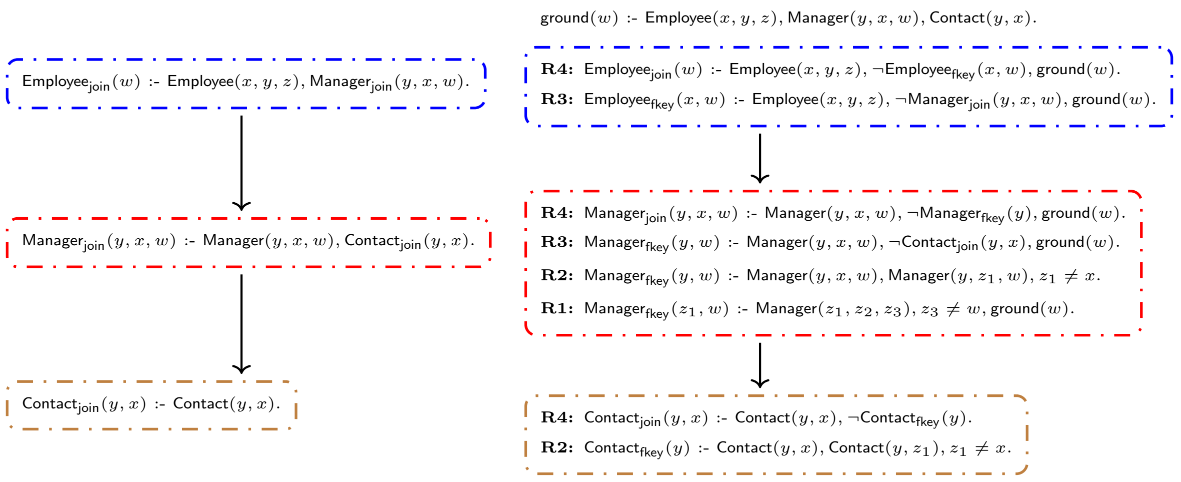
$$q^{\mathsf{nex}}(w) \text{ :- } \mathsf{Employee}(\underline{x}, y, z), \mathsf{Manager}(\underline{y}, x, w), \mathsf{Contact}(\underline{y}, x).$$

Note that the constant 2020 in q^{ex} is replaced by the free variable w in q^{nex}. Hence, the program P for $\mathsf{CERTAINTY}(q^{\mathsf{nex}}_{[w \to c]})$ is the same as Figure 3.2, with the only difference that 2020 is replaced by the constant c. The ground rule produced is:

$$\mathsf{ground}(w) \text{ :- } \mathsf{Employee}(x, y, z), \mathsf{Manager}(y, x, w), \mathsf{Contact}(y, x).$$

Figure 3.3a shows how Yannakakis' algorithm evaluates q^{nex}.

To see how the rules of P would change for the non-Boolean case, consider the self-pruning rule for Contact. This rule would remain as is, because it contains no free variable and the predicate



$\mathsf{Employee}_{\mathsf{join}}(w)$:- $\mathsf{Employee}(x, y, z), \mathsf{Manager}_{\mathsf{join}}(y, x, w)$.

$\mathsf{Manager}_{\mathsf{join}}(y, x, w)$:- $\mathsf{Manager}(y, x, w), \mathsf{Contact}_{\mathsf{join}}(y, x)$.

$\mathsf{Contact}_{\mathsf{join}}(y, x)$:- $\mathsf{Contact}(y, x)$.

$\mathsf{ground}(w)$:- $\mathsf{Employee}(x, y, z), \mathsf{Manager}(y, x, w), \mathsf{Contact}(y, x)$.

R4: $\mathsf{Employee}_{\mathsf{join}}(w)$:- $\mathsf{Employee}(x, y, z), \neg\mathsf{Employee}_{\mathsf{fkey}}(x, w), \mathsf{ground}(w)$.

R3: $\mathsf{Employee}_{\mathsf{fkey}}(x, w)$:- $\mathsf{Employee}(x, y, z), \neg\mathsf{Manager}_{\mathsf{join}}(y, x, w), \mathsf{ground}(w)$.

R4: $\mathsf{Manager}_{\mathsf{join}}(y, x, w)$:- $\mathsf{Manager}(y, x, w), \neg\mathsf{Manager}_{\mathsf{fkey}}(y), \mathsf{ground}(w)$.

R3: $\mathsf{Manager}_{\mathsf{fkey}}(y, w)$:- $\mathsf{Manager}(y, x, w), \neg\mathsf{Contact}_{\mathsf{join}}(y, x), \mathsf{ground}(w)$.

R2: $\mathsf{Manager}_{\mathsf{fkey}}(y, w)$:- $\mathsf{Manager}(y, x, w), \mathsf{Manager}(y, z_1, w), z_1 \neq x$.

R1: $\mathsf{Manager}_{\mathsf{fkey}}(z_1, w)$:- $\mathsf{Manager}(z_1, z_2, z_3), z_3 \neq w, \mathsf{ground}(w)$.

R4: $\mathsf{Contact}_{\mathsf{join}}(y, x)$:- $\mathsf{Contact}(y, x), \neg\mathsf{Contact}_{\mathsf{fkey}}(y)$.

R2: $\mathsf{Contact}_{\mathsf{fkey}}(y)$:- $\mathsf{Contact}(y, x), \mathsf{Contact}(y, z_1), z_1 \neq x$.

(a) Yannakakis' Algorithm (b) PPJT extended to non-Boolean queries

Figure 3.3: The non-recursive Datalog program for q^{nex} and $\mathsf{CERTAINTY}(q^{\mathsf{nex}})$.

$\mathsf{Contact}_{\mathsf{fkey}}$ remains unchanged. In contrast, consider the first self-pruning rule for $\mathsf{Manager}$, which in P would be:

$$\mathsf{Manager}_{\mathsf{fkey}}(y_1) \text{ :- } \mathsf{Manager}(y_1, y_2, y_3), y_3 \neq w.$$

Here, w is unsafe, so we need to add the atom $\mathsf{ground}(w)$. Additionally, w is now a free variable in the subtree rooted at $\mathsf{Manager}$, so the predicate $\mathsf{Manager}_{\mathsf{fkey}}(y_1)$ becomes $\mathsf{Manager}_{\mathsf{fkey}}(y_1, w)$. The transformed rule will be:

$$\mathsf{Manager}_{\mathsf{fkey}}(y_1, w) \text{ :- } \mathsf{Manager}(y_1, y_2, y_3), y_3 \neq w, \mathsf{ground}(w).$$

The full rewriting for q^{nex} can be seen in Figure 3.3b. □

The above rewriting process may introduce Cartesian products in the rules. In the next section, we will see how we can tweak the rules in order to avoid this inefficiency.

3.3 Implementation

In this section, we first present LinCQA, a system that produces the consistent **FO**-rewriting of a query q in both Datalog and SQL formats if q has a PPJT. Having a rewriting in both formats allows us to use both Datalog and SQL engines as a backend. We then briefly discuss how we address the flaws of Conquer and Conquesto that impair their actual runtime performance.

3.3.1 LinCQA: Rewriting in Datalog/SQL

Our implementation takes as input a self-join-free CQ q written in either Datalog or SQL. LinCQA first checks whether or not the query q admits a PPJT; if q admits a PPJT, LinCQA produces a consistent **FO**-rewriting of q in Datalog or SQL.

3.3.1.1 Datalog rewriting

LinCQA implements all rules introduced in Subsection 3.2.2, with one modification to the ground rule atom. Let the input query be

$$q(\vec{u}) \text{ :- } R_1(\underline{\vec{x_1}}, \vec{y_1}), R_2(\underline{\vec{x_2}}, \vec{y_2}), \ldots, R_k(\underline{\vec{x_k}}, \vec{y_k}).$$

In Subsection 3.2.3, the head of the ground rule is $\mathsf{ground}(\vec{u})$. In the implementation, we replace that rule with

$$\mathsf{ground}^*(\vec{x}_1, \vec{x}_2, \ldots, \vec{x}_k, \vec{u}) \text{ :- } \mathsf{body}(q).$$

The rule keeps the key variables of all atoms. For each unsafe rule with head $R_{i,\mathsf{label}}$ where $\mathsf{label} \in \{\mathsf{fkey}, \mathsf{join}\}$, let $\vec{v}$ be the key in the occurrence of R_i in the body of the rule (if the unsafe rule is produced by **Rule 3.2**, both occurrences of R_i share the same key). Then, we add to the rule body the atom

$$\mathsf{ground}^*(\vec{z}_1, \ldots, \vec{z}_{i-1}, \vec{v}, \vec{z}_{i+1}, \ldots, \vec{z}_k, \vec{u})$$

where each $\vec{z}_i$ is a sequence of fresh variables of the same length as $\vec{x}_i$.

The rationale is that appending $\mathsf{ground}(\vec{u})$ to all unsafe rules could potentially introduce a Cartesian product between $\mathsf{ground}(\vec{u})$ and some existing atom $R(\vec{v}, \vec{w})$ in the rule. The Cartesian product has size $O(N \cdot |\mathsf{OUT}_p|)$ and would take $\Omega(N \cdot |\mathsf{OUT}_p|)$ time to compute, often resulting in inefficient evaluations or even out-of-memory errors. On the other hand, adding ground^* guarantees a join with an existing atom in the rule. Hence the revised rules would take $O(N + |\mathsf{ground}^*|)$ time to compute. Note that the size of ground^* can be as large as $N^k \cdot |\mathsf{OUT}_p|$ in the worst case; but as we observe in the experiments, the size of ground^* is small in practice.

3.3.1.2 SQL rewriting

We now describe how to translate the Datalog rules in Subsection 3.2.2 to SQL queries. Given a query q, we first denote the following:

1. **KeyAttri**(R): the primary key attributes of relation R;
2. **JoinAttri**(R, T): the attributes of R that join with T;

3. **Comp**(R): the conjunction of comparison predicates imposed entirely on R, excluding all join predicates (e.g., $\mathsf{R}.A = 42$ and $\mathsf{R}.A = \mathsf{R}.B$); and

4. **NegComp**(R): the negation of **Comp**(R) (e.g., $\mathsf{R}.A \neq 42$ or $\mathsf{R}.A \neq \mathsf{R}.B$).

Translation of Rule 3.1. We translate **Rule 3.1** of Subsection 3.2.2 into the following SQL query computing the keys of R.

```
SELECT KeyAttri(R) FROM R WHERE NegComp(R)
```

Translation of Rule 3.2. We first produce the projection on all key attributes and the joining attributes of R with its parent T (if it exists), and then compute all blocks containing at least two facts that disagree on the joining attributes. This can be effectively implemented in SQL with `GROUP BY` and `HAVING`.

```
SELECT KeyAttri(R)
FROM (SELECT DISTINCT KeyAttri(R), JoinAttri(R, T) FROM R) t
GROUP BY KeyAttri(R)
HAVING COUNT(*) > 1
```

Translation of Rule 3.3. For **Rule 3.3** in the pair-pruning phase, we need to compute all blocks of R containing some fact that does not join with some fact in S_{join} for some child node S of R. This can be achieved through a *left outer join* between R and each of its child nodes $\mathsf{S}^1_{\mathsf{join}}$, $\mathsf{S}^2_{\mathsf{join}}$, …, $\mathsf{S}^k_{\mathsf{join}}$, which are readily computed in the recursive steps. For each $1 \leq i \leq k$, let the attributes of S^i be B^i_1, B^i_2, …, $B^i_{m_i}$, joining with attributes $A_{\alpha^i_1}$, $A_{\alpha^i_2}$, …, $A_{\alpha^i_{m_i}}$ in R respectively. We produce the following rule:

```
SELECT KeyAttri(R) FROM R
LEFT OUTER JOIN $\mathsf{S}^1_{\mathsf{join}}$ ON
  $\mathsf{R}.A_{\alpha^1_1} = \mathsf{S}^1_{\mathsf{join}}.B^1_1$ AND ... AND $\mathsf{R}.A_{\alpha^1_{m_1}} = \mathsf{S}^1_{\mathsf{join}}.B^1_{m_1}$
...
LEFT OUTER JOIN $\mathsf{S}^k_{\mathsf{join}}$ ON
  $\mathsf{R}.A_{\alpha^k_1} = \mathsf{S}^k_{\mathsf{join}}.B^k_1$ AND ... AND $\mathsf{R}.A_{\alpha^k_{m_k}} = \mathsf{S}^k_{\mathsf{join}}.B^k_{m_k}$
WHERE $\mathsf{S}^1_{\mathsf{join}}.B^1_1$ IS NULL OR ... $\mathsf{S}^1_{\mathsf{join}}.B^1_{m_1}$ IS NULL OR
      $\mathsf{S}^2_{\mathsf{join}}.B^2_1$ IS NULL OR ... $\mathsf{S}^2_{\mathsf{join}}.B^2_{m_2}$ IS NULL OR
      ...
      $\mathsf{S}^k_{\mathsf{join}}.B^k_1$ IS NULL OR ... $\mathsf{S}^k_{\mathsf{join}}.B^k_{m_k}$ IS NULL
```

The *inconsistent blocks* represented by the keys found by the above three queries are *combined* using `UNION ALL` (e.g., R_{fkey} in **Rule 3.1, 3.2, 3.3**).

Translation of Rule 3.4. Finally, we translate **Rule 3.4** computing the values on join attributes between *good blocks* in R and its unique parent T if it exists. Let $A_1, A_2, \ldots, A_k$ be the key attributes of R.

```
SELECT JoinAttri(R, T) FROM R WHERE NOT EXISTS (
  SELECT *
  FROM R_fkey
  WHERE R.A_1 = R_fkey.A_1 AND ... AND R.A_k = R_fkey.A_k)
```

If R is the root relation of the PPJT, we replace **JoinAttri**(R, T) with `DISTINCT 1` (i.e. a Boolean query). Otherwise, the results returned from the above query are stored as $\mathsf{R}_{\mathsf{join}}$ and the recursive process continues as described in Algorithm 1.

Extension to non-Boolean queries. Let q be a non-Boolean query. We use **ProjAttri**(q) to denote a sequence of attributes of q to be projected and let **CompPredicate**(q) be the comparison expression in the `WHERE` clause of q. We first produce the SQL query that computes the facts of ground*.

```
SELECT KeyAttri(R_1), ..., KeyAttri(R_k), ProjAttri(q)
FROM R_1, R_2, ..., R_k
WHERE CompPredicate(q)
```

We then modify each SQL statement as follows. Consider a SQL statement whose corresponding Datalog rule is unsafe and let $T(\vec{v}, \vec{w})$ be an atom in the rule body. Let $\vec{u}_T$ be a sequence of free variables in q_{τ_T} and let **FreeAttri**(T) be a sequence of attributes in q_{τ_T} to be projected (i.e., corresponding to the variables in $\vec{u}_T$). Since, as previously mentioned, $T_{\mathsf{join}}(\vec{v})$ and $T_{\mathsf{fkey}}(\vec{v})$ are replaced with $T_{\mathsf{join}}(\vec{v}, \vec{u}_T)$ and $T_{\mathsf{fkey}}(\vec{v}, \vec{u}_T)$ respectively, we first append **FreeAttri**(T) to the `SELECT` clause and then add a `JOIN` between table T and ground on all attributes in **KeyAttri**(T). Finally, for a rule that has some negated IDB containing a free variable corresponding to some attribute in ground (i.e., ground.A),

- if the rule is produced by **Rule 3.3,** in each `LEFT OUTER JOIN` with S^i_{join} we add the expression `ground.A =` $\mathsf{S}^i_{\mathsf{join}}.B$ connected by the `AND` operator, where B is an attribute to be projected in $\mathsf{S}^i_{\mathsf{join}}$. In the `WHERE` clause we also add an expression `ground.A IS NULL`, connected by the `OR` operator; and
- if the rule is produced by **Rule 3.4**, in the `WHERE` clause of the subquery we add an expression `ground.A =` $\mathsf{R}_{\mathsf{fkey}}$`.A`.

3.3.2 Improvements upon existing CQA systems

Conquer Fuxman and Miller [FM07] identified $\mathcal{C}_{\mathsf{forest}}$, a class of CQs whose consistent answers can be computed via an **FO**-rewriting. However, their accompanying system can only handle queries in $\mathcal{C}_{\mathsf{forest}}$ whose join graph is a tree, unable to handle the query in $\mathcal{C}_{\mathsf{forest}}$ whose join graph is not connected [FFM05]. Since we were unable to find the original ConQuer implementation, we re-implemented ConQuer and added an efficient implementation of the method `RewriteConsistent` in Figure 2 of [FM07], enabling us to produce the consistent SQL rewriting for every query in $\mathcal{C}_{\mathsf{forest}}$.

Conquesto Conquesto [KJL+20] produces a non-recursive Datalog program that implements the algorithm in [KW17], targeting all **FO**-rewritable self-join-free CQs. However, it suffers from repeated computation and unnecessary cartesian products. For example, the Conquesto rewriting for the CQ $q(z)$:- $R_1(\underline{x}, y, z), R_2(\underline{y}, v, w)$ is shown as follows, where Rule (3.7) and (3.9) share the common predicate $\mathsf{R}_2(y, v, w)$ in their bodies, resulting in re-computation, and Rule (3.11) involves a Cartesian product.

$$\mathsf{Sr}_{\mathsf{R}_2}(y) \text{ :- } \mathsf{R}_2(y, v, w). \tag{3.7}$$

$$\mathsf{Yes}_{\mathsf{R}_2}(y) \text{ :- } \mathsf{Sr}_{\mathsf{R}_2}(y), \mathsf{R}_2(y, v, w). \tag{3.8}$$

$$\mathsf{Sr}_{\mathsf{R}_1}(z) \text{ :- } \mathsf{R}_1(x, y, z), \mathsf{R}_2(y, v, w). \tag{3.9}$$

$$\mathsf{Gf}_{\mathsf{R}_1}(v_2, x, y, z) \text{ :- } \mathsf{Sr}_{\mathsf{R}_1}(z), \mathsf{R}_1(x, y, v_2), \mathsf{Yes}_{\mathsf{R}_2}(y), v_2 = z. \tag{3.10}$$

$$\mathsf{Bb}_{\mathsf{R}_1}(x, z) \text{ :- } \mathsf{Sr}_{\mathsf{R}_1}(z), \mathsf{R}_1(x, y, v_2), \neg\mathsf{Gf}_{\mathsf{R}_1}(v_2, x, y, z). \tag{3.11}$$

$$\mathsf{Yes}_{\mathsf{R}_1}(z) \text{ :- } \mathsf{Sr}_{\mathsf{R}_1}(z), \mathsf{R}_1(x, y, z), \neg\mathsf{Bb}_{\mathsf{R}_1}(x, z). \tag{3.12}$$

We thus implement FastFO to address the aforementioned issues, incorporating our ideas in Subsection 3.3.1.1. Instead of re-computing the *local safe ranges* such as $\mathsf{Sr}_{\mathsf{R}_1}(y)$ and $\mathsf{Sr}_{\mathsf{R}_2}(z)$, we compute a *global safe range* $\mathsf{Sr}(x, y, z)$, which includes all key variables from all atoms and the free variables. This removes all undesired Cartesian products and the recomputations of the local safe ranges at once. The FastFO rewriting for q is presented below.

$$\mathsf{Sr}(x, y, z) \text{ :- } \mathsf{R}_1(x, y, z), \mathsf{R}_2(y, v, w). \tag{3.13}$$

$$\mathsf{Yes}_{\mathsf{R}_2}(y) \text{ :- } \mathsf{Sr}(x, y, z), \mathsf{R}_2(y, v, w). \tag{3.14}$$

$$\mathsf{Gf}_{\mathsf{R}_1}(v_2, x, y, z) \text{ :- } \mathsf{Sr}(x, \_, z), \mathsf{R}_1(x, y, v_2), \mathsf{Yes}_{\mathsf{R}_2}(y), v_2 = z. \tag{3.15}$$

$$\mathsf{Bb}_{\mathsf{R}_1}(x, z) \text{ :- } \mathsf{Sr}(x, \_, z), \mathsf{R}_1(x, y, v_2), \neg\mathsf{Gf}_{\mathsf{R}_1}(v_2, x, y, z). \tag{3.16}$$

$$\mathsf{Yes}_{\mathsf{R}_1}(z) \text{ :- } \mathsf{Sr}(x, y, z), \mathsf{R}_1(x, y, z), \neg\mathsf{Bb}_{\mathsf{R}_1}(x, z). \tag{3.17}$$

For evaluation, the rules computing each intermediate relation (i.e. all rules except for the one computing $\mathsf{Yes}_{\mathsf{R}_1}(z)$) are then translated to a SQL subquery via a `WITH` clause.

3.4 Experiments

Our experimental evaluation addresses the following questions:

1. How do first-order rewriting techniques perform compared to generic state-of-the-art CQA systems (e.g., CAvSAT)?

2. How does LinCQA perform compared to other existing CQA techniques?

3. How do different CQA techniques behave on inconsistent databases with different properties (e.g., varying inconsistent block sizes, inconsistency)?

4. Are there instances where we can observe the worst-case guarantee of LinCQA that other CQA techniques lack?

To answer these questions, we perform experiments using synthetic benchmarks used in prior works and a large real-world dataset of 400GB. We compare LinCQA against several state-of-the-art CQA systems with improvements. To the best of our knowledge, this is the most comprehensive performance evaluation of existing CQA techniques, and we are the first ones to evaluate different CQA techniques on a real-world dataset of this large scale.

3.4.1 Experimental Setup

We next briefly describe the setup of our experiments.

System configuration. All of our experiments are conducted on a bare-metal server in Cloudlab [Clo18], a large cloud infrastructure. The server runs Ubuntu 18.04.1 LTS and has two Intel Xeon E5-2660 v3 2.60 GHz (Haswell EP) processors. Each processor has 10 cores, and 20 hyper-threading hardware threads. The server has a SATA SSD with 440GB space being available, 160GB memory and each NUMA node is directly attached to 80GB of memory. We run Microsoft SQL Server 2019 Developer Edition (64-bit) on Linux as the relational backend for all CQA systems. For CAvSAT, MaxHS v3.2.1 [DB11] is used as the solver for the output WPMaxSAT instances.

Other CQA systems. We compare the performance of LinCQA with several state-of-the-art CQA methods.

ConQuer: a CQA system that outputs a SQL rewriting for queries that are in $\mathcal{C}_{\mathsf{forest}}$ [FFM05].

FastFO: our own implementation of the general method that can handle any query for which CQA is **FO**-rewritable.

CAvSAT: a recent SAT-based system. It reduces the complement of CQA with arbitrary denial constraints to a SAT problem, which is solved with an efficient SAT solver [DK19].

For LinCQA, ConQuer and FastFO, we only report execution time of **FO**-rewritings, since the rewritings themselves can be produced within 1ms for all our queries. We report the performance of each **FO**-rewriting using the best query plan. The preprocessing time required by CAvSAT *prior* to computing the consistent answers is not reported. For each rewriting and database shown in the experimental results, we run the evaluation five times (unless timed out), discard the first run and report the average time of the last four runs.

3.4.2 Databases and Queries

3.4.2.1 Synthetic workload

We consider the synthetic workload used in previous works [KPT13, DK19, Dix21]. Specifically, we take the seven queries that are consistent first-order rewritable in [DK19, KPT13, Dix21]. These queries feature joins between primary-key attributes and foreign-key attributes, as well as projections on non-key attributes:

$$\begin{aligned}
q_1(z) &\text{ :- } \mathsf{R}_1(\underline{x}, y, z), \mathsf{R}_3(\underline{y}, v, w).\\
q_2(z, w) &\text{ :- } \mathsf{R}_1(\underline{x}, y, z), \mathsf{R}_2(\underline{y}, v, w).\\
q_3(z) &\text{ :- } \mathsf{R}_1(\underline{x}, y, z), \mathsf{R}_2(\underline{y}, v, w), \mathsf{R}_7(\underline{v}, u, d).\\
q_4(z, d) &\text{ :- } \mathsf{R}_1(\underline{x}, y, z), \mathsf{R}_2(\underline{y}, v, w), \mathsf{R}_7(\underline{v}, u, d).\\
q_5(z) &\text{ :- } \mathsf{R}_1(\underline{x}, y, z), \mathsf{R}_8(\underline{y, v}, w).\\
q_6(z) &\text{ :- } \mathsf{R}_1(\underline{x}, y, z), \mathsf{R}_6(\underline{t}, y, w), \mathsf{R}_9(\underline{x}, y, d).\\
q_7(z) &\text{ :- } \mathsf{R}_3(\underline{x}, y, z), \mathsf{R}_4(\underline{y}, x, w), \mathsf{R}_{10}(\underline{x}, y, d).
\end{aligned}$$

The synthetic instances are generated in two phases. In the first phase, we generate the consistent instance, while in the second phase we inject inconsistency. We use the following parameters for data generation: (*i*) rSize: the number of tuples per relation, (*ii*) inRatio: the ratio of the number of tuples that violate primary key constraints (i.e., number of tuples that are in inconsistent blocks) to the total number of tuples of the database, and (*iii*) bSize: the number of inconsistent tuples in each inconsistent block.

Consistent data generation. Each relation in the consistent database has the same number of tuples, so that injecting inconsistency with specified bSize and inRatio makes the total number of tuples in the relation equal to rSize. The data generation is *query-specific*: for each of the seven queries, the data is generated in a way to ensure the output size of the original query on the consistent database is reasonably large. To achieve this purpose, when generating the database instance for one of the seven queries, we ensure that for any two relations that join on some attributes, the number of matching tuples in each relation is approximately 25%; for the third attribute in each ternary relation that does not participate in a join but is sometimes present in the final projection, the values are chosen uniformly from the range $[1, \mathsf{rSize}/10]$.

Inconsistency injection. In each relation, we first select a number of primary keys (or number of inconsistent blocks $\mathsf{inBlockNum}$) from the generated consistent instance. Then, for each selected primary key, the inconsistency is injected by inserting the *same number of additional tuples* ($\mathsf{bSize}-1$) into each block. The parameter $\mathsf{inBlockNum}$ is calculated by the given $\mathsf{rSize}, \mathsf{inRatio}$ and bSize: $\mathsf{inBlockNum} = (\mathsf{inRatio} \cdot \mathsf{rSize})/\mathsf{bSize}$. We remark that there are alternative inconsistency injection methods available [AGM+15, AJKO08].

3.4.2.2 TPC-H benchmark.

We also altered the 22 queries from the original TPC-H benchmark [PF00] by removing aggregation, nested subqueries and selection predicates other than constant constraints, yielding 14 simplified conjunctive queries, namely queries q'_1, q'_2, q'_3, q'_4, q'_6, q'_{10}, q'_{11}, q'_{12}, q'_{14}, q'_{16}, q'_{17}, q'_{18}, q'_{20}, q'_{21}. All of the 14 queries are in $\mathcal{C}_{\mathsf{forest}}$ and hence each query has a PPJT, meaning that they can be handled by both ConQuer and LinCQA.

We generate the inconsistent instances by injecting inconsistency into the TPC-H databases of scale factor (SF) 1 and 10 in the same way as described for the synthetic data. The only difference is that for a given consistent database instance, instead of fixing rSize for the inconsistent database, we determine the number of inconsistent tuples to be injected based on the size of the consistent database instance, the specified $\mathsf{inRatio}$ and bSize.

3.4.2.3 Stackoverflow Dataset.

We obtained the `stackoverflow.com` metadata as of 02/2021 from the Stack Exchange Data Dump, with 551,271,294 rows taking up 400GB.[4] [5] The database tables used are summarized in Table 3.2. We remark that the *Id* attributes in PostHistory, Comments, Badges, and Votes are surrogate keys and therefore not imposed as natural primary keys; instead, we properly choose composite keys as primary keys (or quasi-keys).

[4]`https://archive.org/details/stackexchange`
[5]https://sedeschema.github.io/

Table 3.1: StackOverflow queries

Q_1	`SELECT DISTINCT P.id, P.title FROM Posts P, Votes V WHERE P.Id = V.PostId AND P.OwnerUserId = V.UserId AND BountyAmount > 100`
Q_2	`SELECT DISTINCT U.Id, U.DisplayName FROM Users U, Badges B WHERE U.Id = B.UserId AND B.name = "Illuminator"`
Q_3	`SELECT DISTINCT U.DisplayName FROM Users U, Posts P WHERE U.Id = P.OwnerUserId AND P.Tags LIKE "<c++>"`
Q_4	`SELECT DISTINCT U.Id, U.DisplayName FROM Users U, Posts P, Comments C WHERE C.UserId = U.Id AND C.PostId = P.Id AND P.Tags LIKE "%SQL%" AND C.Score > 5`
Q_5	`SELECT DISTINCT P.Id, P.Title FROM Posts P, PostHistory PH, Votes V, Comments C WHERE P.id = V.PostId AND P.id = PH.PostId AND P.id = C.PostId AND P.Tags LIKE "%SQL%" AND V.BountyAmount > 100 AND PH.PostHistoryTypeId = 2 AND C.score = 0`

Table 3.2: A summary of the Stackoverflow Dataset.

Table	# of rows	inRatio	max. bSize	Attributes
Users	14,839,627	0%	1	Id, AboutMe, Age, CreationDate, DisplayName, DownVotes, EmailHash, LastAccessDate, Location, Reputation, UpVotes, Views, WebsiteUrl, AccountId
Posts	53,086,328	0%	1	Id, AcceptedAnswerId, AnswerCount, Body, ClosedDate, CommentCount, CommunityOwnedDate, CreationDate, FavoriteCount, LastActivityDate, LastEditDate, LastEditorDisplayName, LastEditorUserId, OwnerUserId, ParentId, PostTypeId, Score, Tags, Title, ViewCount
PostLinks	7,499,403	0%	1	Id, CreationDate, PostId, RelatedPostId, LinkTypeId
PostHistory	141,277,451	0.001%	4	Id, PostHistoryTypeId, PostId, RevisionGUID, CreationDate, UserId, UserDisplayName, Comment, Text
Comments	80,673,644	0.0012%	7	Id, CreationDate, PostId, Score, Text, UserId
Badges	40,338,942	0.58%	941	Id, Name, UserId, Date
Votes	213,555,899	30.9%	1441	Id, PostId, UserId, BountyAmount, VoteTypeId, CreationDate

Table 3.1 shows the five queries used in our CQA evaluation, where the number of tables joined together increases from 2 in Q_1 to 4 in Q_5.

3.4.3 Experimental Results

In this section, we report the evaluation of LinCQA and the other CQA systems on synthetic workloads and the StackOverflow dataset. Table 3.3 summarizes the number of consistent and possible answers for each query in the selected datasets.

Fixed inconsistency with varying relation sizes. To compare LinCQA with other CQA systems, we evaluate all systems using both the synthetic workload and the altered TPC-H benchmark with fixed inconsistency ($\mathsf{inRatio} = 10\%$, $\mathsf{bSize} = 2$) as in previous works [KPT13, DK19, Dix21]. We vary the size of each relation ($\mathsf{rSize} \in \{500\text{K}, 1\text{M}, 5\text{M}\}$) in the synthetic data (Figure 3.4) and we evaluate on TPC-H database instances of scale factors 1 and 10 (Figure 3.5). Both figures include the time for running the original query on the inconsistent database (which returns the possible answers).

In the synthetic dataset, all three systems based on **FO**-rewriting techniques outperform CAvSAT, often by an order of magnitude. This observation shows that if $\mathsf{CERTAINTY}(q)$ is **FO**-rewritable, a properly implemented rewriting is more efficient than the generic algorithm in practice, refuting some observations in [DK19, KPT13]. Compared to ConQuer, LinCQA performs better or comparably on q_1 through q_4. LinCQA is also more efficient than ConQuer for q_1, q_2, and q_3. As the database size increases, the relative performance gap between LinCQA and ConQuer reduces for q_4. ConQuer cannot produce the SQL rewritings for queries q_5, q_6, and q_7 since they are not in $\mathcal{C}_{\mathsf{forest}}$. In summary, LinCQA is more efficient and at worst competitive to ConQuer on relatively small databases with less than 5M tuples, and is applicable to a wider class of acyclic queries.

In the TPC-H benchmark, the CQA systems are much closer in terms of performance. In this experiment, we observe that LinCQA almost always produces the fastest rewriting, and even when it is not, its performance is comparable to the other baselines. It is also worth noting that for most queries in the TPC-H benchmark, the overhead over running the SQL query directly is much smaller when compared to the synthetic benchmark. Note that CAvSAT times out after one hour for queries q'_{10} and q'_{18} for both scale 1 and 10, while the systems based on **FO**-rewriting techniques terminate. We also remark that for Boolean queries, CAvSAT will terminate at an early stage without processing the inconsistent part of the database using SAT solvers if the consistent part of the database already satisfies the query (e.g., q'_6, q'_{14}, q'_{17} in TPC-H). Overall, both LinCQA and ConQuer perform better than FastFO, since they both are better at exploiting the structure of the join tree. We also note that ConQuer and LinCQA exhibit comparable performances on

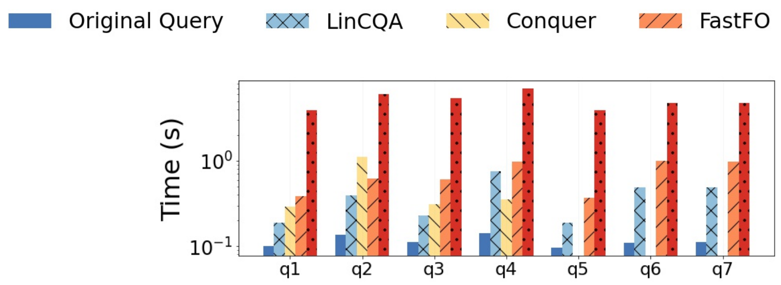
Table 3.3: The number of consistent and possible answers for each query in selected datasets.

	Synthetic (rSize = 5M, inRatio = 10%, bSize = 2)						
	q_1	q_2	q_3	q_4	q_5	q_6	q_7
# cons.	311573	463459	290012	408230	311434	277287	277135
# poss.	571047	572244	534011	534953	574615	504907	474203

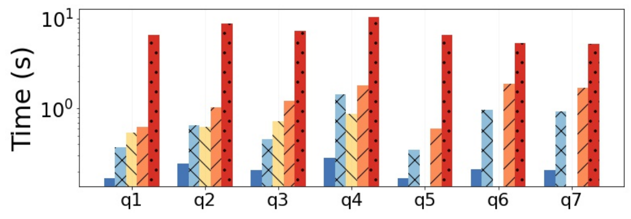
	TPC-H (SF = 10)						
	q'_1	q'_2	q'_3	q'_4	q'_6	q'_{10}	q'_{11}
# cons.	4	28591	0	5	1	901514	289361
# poss.	4	35206	0	5	1	1089754	318015
	q'_{12}	q'_{14}	q'_{16}	q'_{17}	q'_{18}	q'_{20}	q'_{21}
# cons.	7	1	187489	1	13465732	3844	3776
# poss.	7	1	187495	1	16617583	4054	4010

	StackOverflow				
	Q_1	Q_2	Q_3	Q_4	Q_5
# cons.	27578	145	38320	3925	1245
# poss.	27578	145	38320	3925	1250

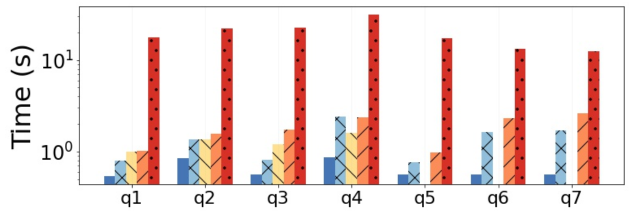
most queries in TPC-H. When computing consistent answers for a given query, we observed that the actual runtime performance heavily depends on the query plan chosen by the query optimizer besides the SQL rewriting given. In view of this observation, rather than comparing different CQA systems on the basis of relatively small performance differences between individual queries, we focus on the overall performance of different CQA systems.



(a) rSize = 500K



(b) rSize = 1M



(c) rSize = 5M

Figure 3.4: Performance comparison of different CQA systems on a synthetic workload with varying relation sizes.

Fixed relation size with varying inconsistency. We perform experiments to observe how different CQA systems react when the inconsistency of the instance changes. Using synthetic data, we first fix rSize = 1M, bSize = 2 and run all CQA systems on databases instances of varying inconsistency ratio from inRatio = 10% to inRatio = 100%. The results are depicted in Figure 3.7. We observe that the running time of CAvSAT increases when the inconsistency ratio of the database instance becomes larger. This happens because the SAT formula grows with larger inconsistency, and hence the SAT solver becomes slower. In contrast, the running time of all **FO**-rewriting techniques is relatively stable across database instances of different inconsistency ratios. More interestingly, the running time of LinCQA decreases when the inconsistency ratio becomes larger. This behavior occurs because of the early pruning on the relations at lower levels of the PPJT, which shrinks the size of the candidate space being considered at higher levels of the PPJT and thus reduces the overall computation time. The overall performance trends of different systems are similar for all queries and thus we present only figures of q_1, q_3, q_5, q_7 here due to the space limit.

In our next experiment, we fix the database instance size with $\mathsf{rSize} = 1\text{M}$ and inconsistency ratio with $\mathsf{inRatio} = 10\%$, running all CQA systems on databases of varying inconsistent block size bSize from 2 to 10. We show the results in Figure 3.8. We observe that the performance of all CQA systems is not very sensitive to the change of inconsistent block sizes. We observe that the performance of all CQA systems is not very sensitive to the change of inconsistent block sizes and thus we omit the results here due to the space limit.

Figure 3.8 summarizes the performance of all seven synthetic queries on varying block sizes.

StackOverflow Dataset. We use a 400GB StackOverflow dataset to evaluate the performance of different systems on large-scale real-world datasets. Another motivation to use such a large dataset is that LinCQA and ConQuer exhibit comparable performance on the medium-sized synthetic and TPC-H datasets. CAvSAT is excluded since it requires extra storage for preprocessing which is beyond the limit of the available disk space. Since Q_1 and Q_5 are not in $\mathcal{C}_{\mathsf{forest}}$, ConQuer cannot handle them and their execution times are marked as "N/A". Query executions that do not finish within one hour are marked as "Time Out". We observe that on all five queries, LinCQA significantly outperforms other competitors. In particular, when the database size is very large, LinCQA is much more scalable than ConQuer due to its more efficient strategy. We intentionally select queries with small possible answer sizes for ease of experiments and presentation. Some queries with possible answer size up to 1M would require hours to be executed and it is prohibitive to measure the performances of our baseline systems. For queries that ConQuer (Q_4) and FastFO (Q_3, Q_5) take long to compute, LinCQA manages to finish execution quickly thanks to its efficient self-pruning and pair-pruning steps.

To see the performance change of different systems when executing in small available memory, we run the experiments on a SQL server with maximum allowed memory of 120GB, 90GB, 60GB, 30GB, and 10GB respectively. Figure 3.9 shows that, despite the memory reduction, LinCQA is still the best performer on all five queries given different amounts of available memory. No obvious performance regression is observed on Q_1 and Q_2 when reducing memory since both queries access only two tables.

Summary. Our experiments show that both LinCQA and ConQuer outperform FastFO and CAvSAT, systems that produce generic **FO**-rewritings and reduce to SAT respectively. Despite LinCQA and ConQuer showing a similar performance on most queries in our experiments, we observe that LinCQA is (1) applicable to a wider class of acyclic queries than ConQuer and (2) more scalable than ConQuer when the database size increases significantly.

3.4.4 Worst-Case Study

To demonstrate the robustness and efficiency of LinCQA as a result of its theoretical guarantees, we generate synthetic *worst-case* inconsistent database instances for the 2-path query $Q_{\mathsf{2-path}}$ and the 3-path query $Q_{\mathsf{3-path}}$:

$$\begin{aligned} Q_{\mathsf{2-path}}(x) &\text{ :- } \mathsf{R}(\underline{x}, y), \mathsf{S}(\underline{y}, z). \\ Q_{\mathsf{3-path}}(x) &\text{ :- } \mathsf{R}(\underline{x}, y), \mathsf{S}(\underline{y}, z), \mathsf{T}(\underline{z}, w). \end{aligned}$$

We compare the performance of LinCQA with ConQuer and FastFO on both queries. CAvSAT does not finish its execution on any instance within one hour, due to the long time it requires to solve the SAT formula. Thus, we do not report the time of CAvSAT.

We define a generic binary relation $\mathcal{D}(x, y, N)$ as

$$\mathcal{D}(x, y, N) = ([x] \times [y]) \cup \{(u, u) \mid xy + 1 \leq u \leq N, u \in \mathbb{Z}^+\},$$

where $x, y, N \in \mathbb{Z}^+$, $[n] = \{1, 2, \ldots, n\}$, and $[a] \times [b]$ denotes the Cartesian product between $[a]$ and $[b]$. To generate the input instances for $Q_{\mathsf{2-path}}$, we generate relations $\mathsf{R} = \mathcal{D}(a, b, N)$ and $\mathsf{S} = \mathcal{D}(b, c, N)$ with integer parameters a, b, c and N. For $Q_{\mathsf{3-path}}$, we additionally generate the relation $\mathsf{T} = \mathcal{D}(c, d, N)$. Intuitively, for R, $[a] \times [b]$ is the set of inconsistent tuples and $\{(u, u) \mid ab + 1 \leq u \leq N, u \in \mathbb{Z}^+\}$ is the set of consistent tuples. The values of a and b control both the number of inconsistent tuples (i.e., ab) and the size of inconsistent blocks (i.e., b). We note that $[a] \times [b]$ and $\{(u, u) \mid ab + 1 \leq u \leq N, u \in \mathbb{Z}^+\}$ are disjoint.

Fixed database inconsistency with varying size. We perform experiments to see how robust different CQA systems are when running queries on an instance of increasing size. For $Q_{\mathsf{2-path}}$, we fix $b = c = 800$, and for each $k = 0, 1, \ldots, 8$, we construct a database instance with $a = 120 + 460k$ and $N = (1 + k/2) \cdot 10^6$. By construction, each database instance has inconsistent block size $\mathsf{bSize} = b = c = 800$ in both relations R and S, and $\mathsf{inRatio} = (ab + bc)/2N = 36.8\%$, with varying relation size $\mathsf{rSize} = N$ ranging from 1M to 5M. Similarly for $Q_{\mathsf{3-path}}$, we fix $b = c = d = 120$, and for each $k = 0, 1, \ldots, 8$, we construct a database instance with $a = 120+180k$ and $N = (1+k/2)\cdot 10^6$. Here the constructed database instances have $\mathsf{inRatio} = (ab + bc + cd)/3N = 1.44\%$. As shown in Figures 3.10a and 3.10b, the performance of LinCQA is much less sensitive to the changes of the relation sizes when compared to other CQA systems. We omit reporting the running time of FastFO for $Q_{3-\mathsf{path}}$ on relatively larger database instances in Figure 3.10b for better contrast with ConQuer and LinCQA.

Fixed database sizes with varying inconsistency. Next, we experiment on instances of varying inconsistency ratio inRatio in which the joining mainly happens between inconsistent blocks of different relations. For $Q_{2-\mathsf{path}}$, we fix $b = c = 800$ and $N = 10^6$, and generate database instances for each $a = 100, 190, 280, \dots, 1000$. All generated database instances have inconsistent block size bSize $= b = c = 800$ for both relations R and S, and the size of each relation rSize $= N = 10^6$ by construction. The inconsistency ratio inRatio varies from 36% to 72%. For $Q_{3-\mathsf{path}}$, we fix $b = c = d = 120$ and $N = 10^6$ and generate database instances with $a = 200, 800, 1400, \dots, 8000$. The inconsistency ratio of the generated database instances varies from 1.76% to 32.96%. Figures 3.10c and 3.10d show that LinCQA is the only system whose performance is agnostic to the change of the inconsistency ratio. The running time of FastFO and Conquer increases when the input database inconsistency increases. Similar to the experiments varying relation sizes, the running times of FastFO for $Q_{3-\mathsf{path}}$ are omitted on relatively larger database instances in Figure 3.10d for better contrast with ConQuer and LinCQA.

3.5 Open Problems

We remark that CERTAINTY(q) remains solvable in linear time for certain acyclic self-join-free CQ that is **FO**-rewritable but does not have a PPJT. It uses techniques from efficient query result enumeration algorithms [DHK21, DHK20].

Proposition 3.5. *Let* $q()$:- $R(\underline{c}, x), S(\underline{c}, y), T(\underline{x, y})$ *where* c *is a constant. Then there exists a linear-time algorithm for* CERTAINTY(q).

Proof. Let **db** be an instance for CERTAINTY(q). We define $X = \{a \mid R(\underline{c}, a) \in \mathbf{db}\}$ and $Y = \{b \mid S(\underline{c}, b) \in \mathbf{db}\}$. It is easy to see that **db** is a "yes"-instance for CERTAINTY(q) if and only if $X \times Y \subseteq T$, where $\times$ denotes the Cartesian product.

Next, consider the following algorithm that computes $X \times Y$ in linear time, exploiting that T is also part of the input to CERTAINTY(q).

```
1 Compute $X = \{a \mid R(\underline{c}, a) \in \mathbf{db}\}$ and $Y = \{b \mid S(\underline{c}, b) \in \mathbf{db}\}$
2 if $|X| \cdot |Y| > |T|$ then
3     return false
```

```
4 **return whether** $X \times Y \subseteq T$
```

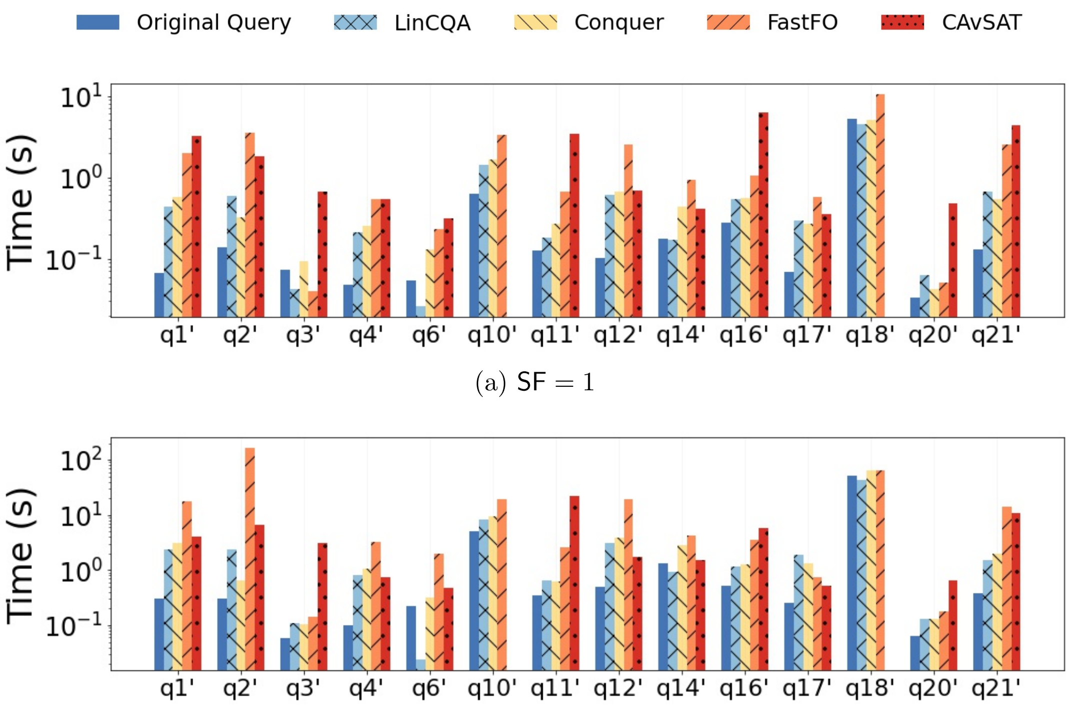
Line 1 and 2 run in time $O(|R|+|S|+|T|)$. If the algorithm terminates at line 3, then the algorithm runs in linear time, or otherwise we must have $|R| \cdot |S| \leq |T|$, and the algorithm thus runs in time $O(|R| + |S| + |T| + |R| \cdot |S|) = O(|R| + |S| + |T|)$.

Note that q does not have a PPJT: in $q_1()$:- $R(\underline{c}, x), T(\underline{x, y})$, R attacks T, and in the query $q_2()$:- $S(\underline{c}, y), T(\underline{x, y})$, S attacks T. $\square$

3.6 Conclusion

In this chapter, we introduce the notion of a pair-pruning join tree (PPJT) and show that if a BCQ has a PPJT, then $\mathsf{CERTAINTY}(q)$ is in **FO** and solvable in linear time in the size of the inconsistent database. We implement this idea in a system called LinCQA that produces a SQL query to compute the consistent answers of q. Our experiments show that LinCQA produces efficient rewritings, is scalable, and robust on worst case instances.

An interesting open question is whether CQA is in linear time for *all* acyclic self-join-free SPJ queries with an acyclic attack graph, including those that do not admit a PPJT. It would also be interesting to study the notion of PPJT for non-acyclic SPJ queries.



(a) SF = 1

(b) SF = 10

Figure 3.5: Performance comparison of different CQA systems on the TPC-H benchmark with varying scale factor (SF).

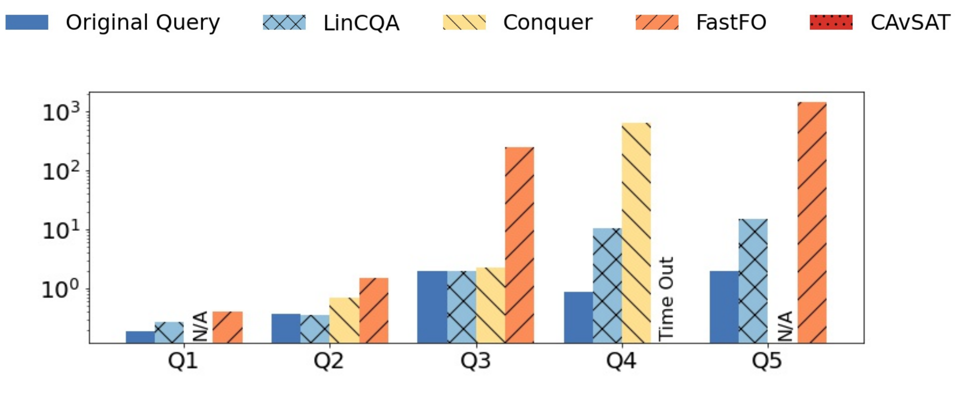
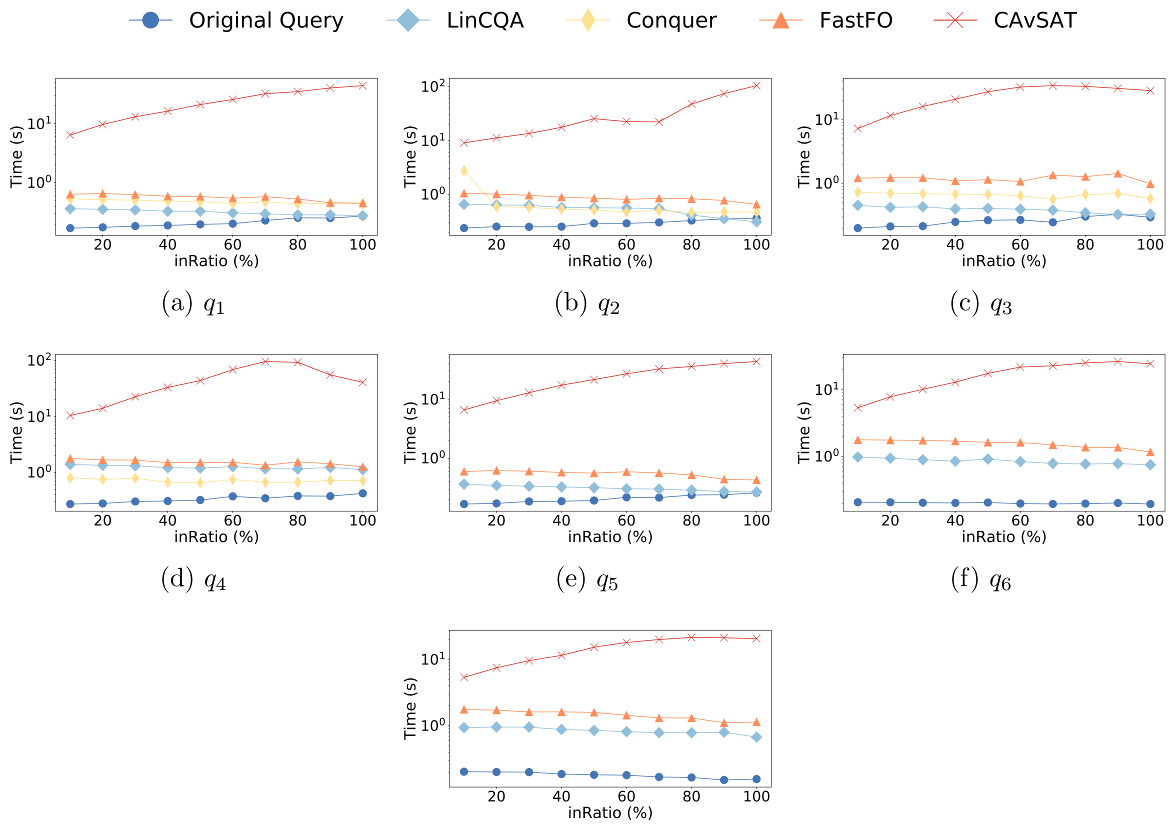


Figure 3.6: Runtime Comparison on StackOverflow



(a) q_1

(b) q_2

(c) q_3

(d) q_4

(e) q_5

(f) q_6

(g) q_7

Figure 3.7: Performance of different systems on inconsistent databases with varying inconsistency ratio

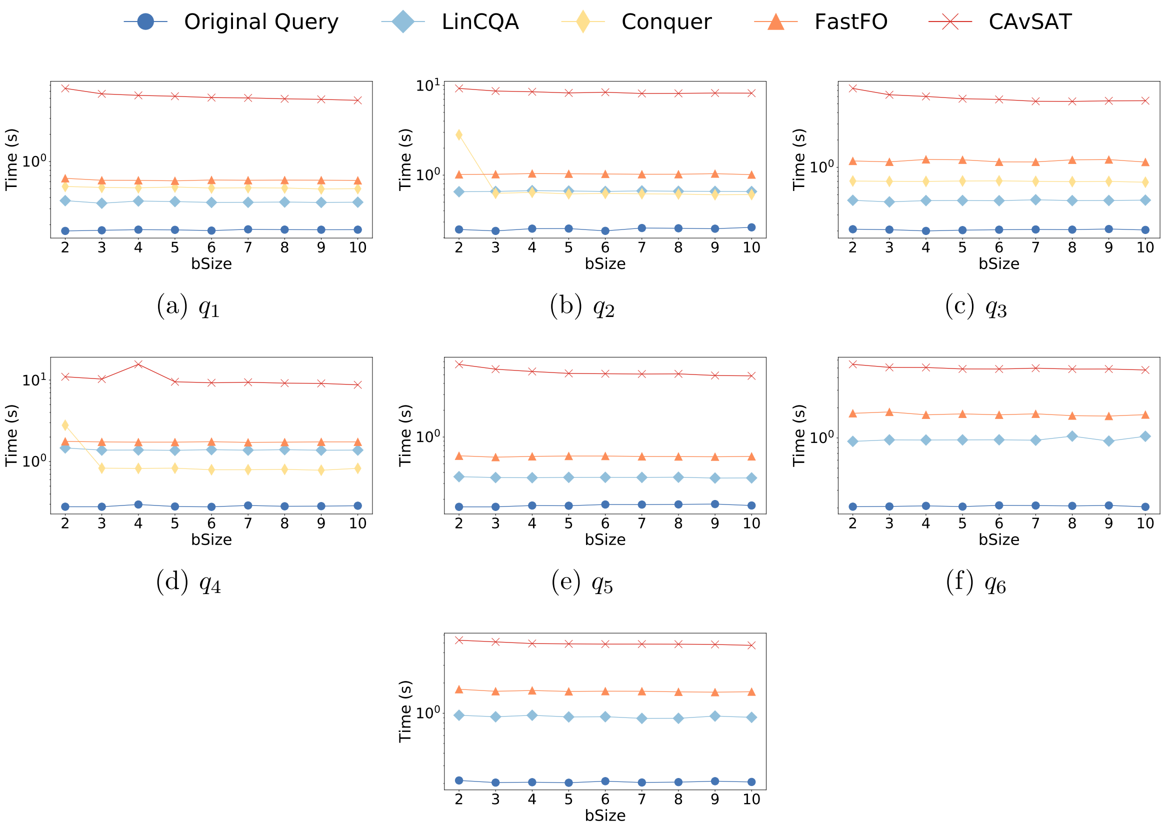
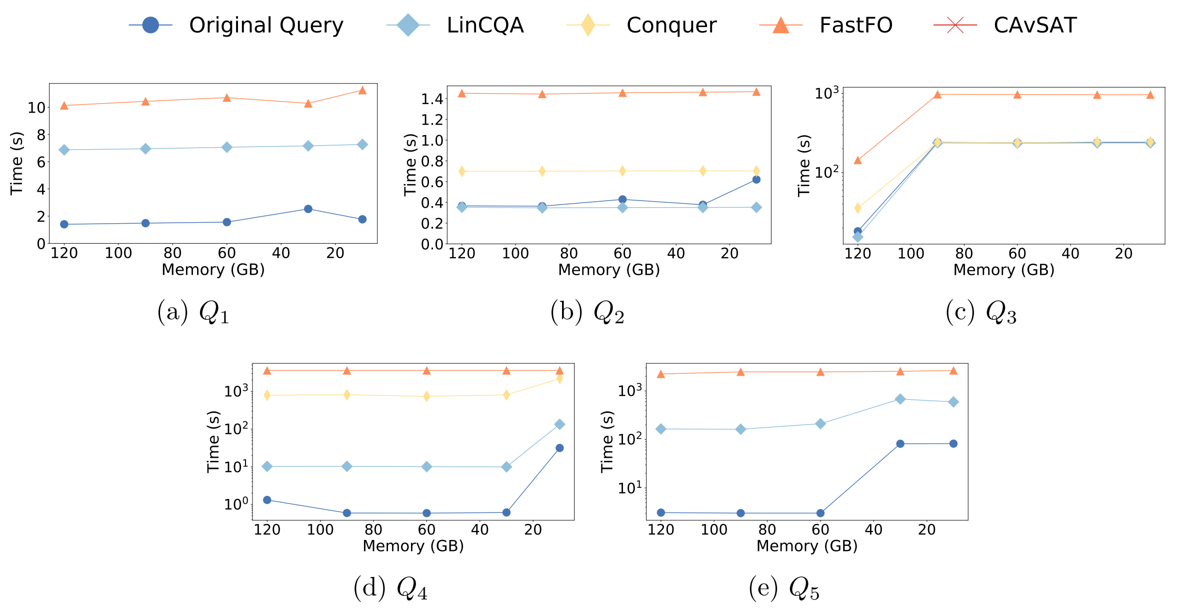


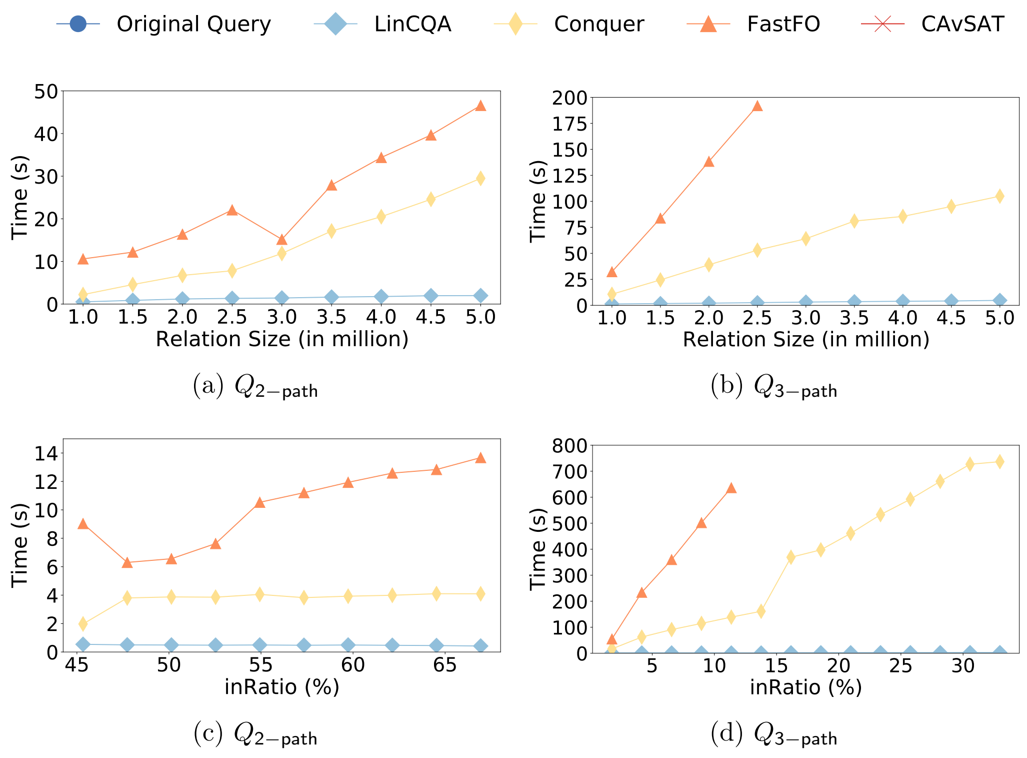
Figure 3.8: Performance of different systems on inconsistent database of varying block size



(a) Q_1 (b) Q_2 (c) Q_3

(d) Q_4 (e) Q_5

Figure 3.9: Performance of StackOverflow queries with varying amount of available memory



(a) $Q_{2-\mathsf{path}}$ (b) $Q_{3-\mathsf{path}}$

(c) $Q_{2-\mathsf{path}}$ (d) $Q_{3-\mathsf{path}}$

Figure 3.10: Performance comparison between different systems on varying relation sizes/inconsistency ratios

Chapter 4

A Tetrachotomy for CQA on Path Queries with Self-joins

衣帶漸寬終不悔
為伊消得人憔悴
—柳永 《蝶戀花》

We discussed in Chapter 3 how to produce an effective linear-time rewriting for CQA based on the existing trichotomy of CQA on self-join-free Boolean conjunctive queries (BCQs), stated in Theorem 1.1 [KW21]. It is thus worthwhile to continue our investigation on the complexity classification of CQA beyond the class of self-join-free BCQs, motivated by the following two questions:

- Why is it so hard to obtain a classification once the queries allow self-joins?
- Where do the existing techniques fall short when we allow self-joins?

In this chapter, we make the first attempt to obtain the complexity classification of $\mathsf{CERTAINTY}(q)$ for path queries with self-joins.

Recall that in computational complexity studies, consistent query answering is commonly defined as the data complexity of the following decision problem, for a fixed Boolean query q:

Problem: $\mathsf{CERTAINTY}(q)$

Input: A database instance **db**.

Question: Does q evaluate to true on every repair of **db**?

For every first-order query q, the problem $\mathsf{CERTAINTY}(q)$ is obviously in **coNP**. However, despite significant research efforts (see Section 1.4), a fine-grained complexity classification is still largely open.

R	$\underline{1}$	2
	a	a
	a	b
	b	a
	b	b

S	$\underline{1}$	2
	a	a
	a	b
	b	a
	b	b

Figure 4.1: An inconsistent database instance **db**.

It has been conjectured that for each Boolean conjunctive query q, the problem $\mathsf{CERTAINTY}(q)$ is either in **PTIME** or **coNP**-complete. On the other hand, for the smaller class of self-join-free Boolean conjunctive queries, the complexity exhibits a trichotomy between **FO**, **L**-complete, and **coNP**-complete, stated in Theorem 1.1 [KW21].

Abandoning the restriction of self-join-freeness turns out to be a major challenge. The difficulty of self-joins is caused by the obvious observation that a single database fact can be used to satisfy more than one atom of a conjunctive query, as illustrated by Example 4.1. Self-joins happen to significantly change the complexity landscape laid down in Theorem 1.1; this is illustrated by Example 4.2. Self-join-freeness is a simplifying assumption that is also used outside CQA (e.g., [FGIM15, BKS17, FGIM20]).

Example 4.1. Take the self-join $q_1 = \exists x \exists y (R(\underline{x}, y) \land R(\underline{y}, x))$ and its self-join-free counterpart $q_2 = \exists x \exists y (R(\underline{x}, y) \land S(\underline{y}, x))$. Consider the inconsistent database instance **db** in Figure 5.5. We have that **db** is a "no"-instance of $\mathsf{CERTAINTY}(q_2)$, because q_2 is not satisfied by the repair $\{R(\underline{a}, a)$, $R(\underline{b}, b)$, $S(\underline{a}, b)$, $S(\underline{b}, a)\}$. However, **db** is a "yes"-instance of $\mathsf{CERTAINTY}(q_1)$. This is because every repair that contains $R(\underline{a}, a)$ or $R(\underline{b}, b)$ will satisfy q_1, while a repair that contains neither of these facts must contain $R(\underline{a}, b)$ and $R(\underline{b}, a)$, which together also satisfy q_1. □

Example 4.2. Take the self-join $q_1 = \exists x \exists y \exists z (R(\underline{x}, z) \land R(\underline{y}, z))$ and its self-join-free counterpart $q_2 = \exists x \exists y \exists z (R(\underline{x}, z) \land S(\underline{y}, z))$. $\mathsf{CERTAINTY}(q_2)$ is known to be **coNP**-complete, whereas it is easily verified that $\mathsf{CERTAINTY}(q_1)$ is in **FO**, by observing that a database instance is a "yes"-instance of $\mathsf{CERTAINTY}(q_1)$ if and only if it satisfies $\exists x \exists y (R(\underline{x}, y))$. □

This chapter makes a contribution to the complexity classification of $\mathsf{CERTAINTY}(q)$ for conjunctive queries, possibly with self-joins, of the form

$$\exists x_1 \cdots \exists x_{k+1} (R_1(\underline{x_1}, x_2) \land R_2(\underline{x_2}, x_3) \land \cdots \land R_k(\underline{x_k}, x_{k+1})),$$

which we call *path queries*. Formally, a *path query* is a Boolean conjunctive query without constants of the following form:

$$q = \{R_1(\underline{x_1}, x_2), R_2(\underline{x_2}, x_3), \ldots, R_k(\underline{x_k}, x_{k+1})\},$$

where x_1, x_2,..., x_{k+1} are distinct variables, and R_1, R_2,..., R_k are (not necessarily distinct) relation names. It will often be convenient to denote this query as a *word* $R_1 R_2 \cdots R_k$ over the alphabet of relation names. This "word" representation is obviously lossless up to a variable renaming. Importantly, path queries may have self-joins, i.e., a relation name may occur multiple times. Path queries containing constants will be discussed in Section 4.6. The treatment of constants is significant, because it allows moving from Boolean to non-Boolean queries, by using that free variables behave like constants.

As will become apparent in our technical treatment, the classification of path queries is already very challenging, even though it is only a first step towards Conjecture 1.1, which is currently beyond reach. If all R_is are distinct (i.e., if there are no self-joins), then $\mathsf{CERTAINTY}(q)$ is known to be in **FO** for path queries q. However, when self-joins are allowed, the complexity landscape of $\mathsf{CERTAINTY}(q)$ for path queries exhibits a tetrachotomy, as stated by the following main result, previously stated in Theorem 1.3.

Theorem 4.1 (Tetrachotomy Theorem)**.** *For each Boolean path query q, $\mathsf{CERTAINTY}(q)$ is in* **FO***,* **NL***-complete,* **PTIME***-complete, or* **coNP***-complete, and it is decidable in polynomial time in the size of q which of the four cases applies.*

Let us provide some intuitions behind Theorem 1.3 and 4.1 by means of examples. Path queries use only binary relation names. A database instance **db** with binary facts can be viewed as a directed edge-colored graph: a fact $R(\underline{a}, b)$ is a directed edge from a to b with color R. A repair of **db** is obtained by choosing, for each vertex, precisely one outgoing edge among all outgoing edges of the same color. We will use the shorthand $q = RR$ to denote the path query

$$q = \exists x \exists y \exists z (R(\underline{x}, y) \land R(\underline{y}, z)).$$

In general, path queries can be represented by words over the alphabet of relation names. Throughout this section, relation names are in uppercase letters, while lowercase letters u, v, w stand for (possibly empty) words. An important operation on words is dubbed *rewinding*: if a word has a factor of the form RvR, then rewinding refers to the operation that replaces this factor with $RvRvR$. That is, rewinding the factor RvR in the word $uRvRw$ yields the longer word $uRvRvRw$. For short, we also say that $uRvRw$ *rewinds to* the word $u{\cdot}Rv{\cdot}\underline{Rv}{\cdot}Rw$, where we used concatenation

$(\cdot)$ and underlining for clarity. For example, $TWITTER$ rewinds to $TWI{\cdot}\underline{TWI}{\cdot}TTER$, but also to $TWIT{\cdot}\underline{TWIT}{\cdot}TER$ and to $TWI{\cdot}T{\cdot}\underline{T}{\cdot}TER$.

Let $q_1 = RR$. It is easily verified that a database instance is a "yes"-instance of $\mathsf{CERTAINTY}(q_1)$ if and only if it satisfies the following first-order formula:

$$\varphi = \exists x(\exists y R(\underline{x}, y) \land \forall y(R(\underline{x}, y) \rightarrow \exists z R(\underline{y}, z))).$$

Informally, every repair contains an R-path of length 2 if and only if there exists some vertex x such that every repair contains a path of length 2 starting in x.

Let $q_2 = RRX$, and consider the database instance in Figure 4.2. Since the only conflicting facts are $R(\underline{1}, 2)$ and $R(\underline{1}, 3)$, this database instance has two repairs. Both repairs satisfy RRX, but unlike the previous example, there is no vertex x such that every repair has a path colored RRX that starts in x. Indeed, in one repair, such path starts in 0; in the other repair it starts in 1. For reasons that will become apparent in our theoretical development, it is significant that both repairs have paths that start in 0 and are colored by a word in the regular language defined by $RR\,(R)^*\,X$. This is exactly the language that contains RRX and is closed under the rewinding operation. In general, it can be verified with some effort that a database instance is a "yes"-instance of $\mathsf{CERTAINTY}(q_2)$ if and only if it contains some vertex x such that every repair has a path that starts in x and is colored by a word in the regular language defined by $RR\,(R)^*\,X$. The latter condition can be tested in **PTIME** (and even in **NL**).

$$\begin{tikzcd}
 & & 2 \arrow[dr, "R"] & & \\
0 \arrow[r, "R"] & 1 \arrow[ur, "R"] \arrow[rr, "R"] & & 3 \arrow[r, "X"] & 4
\end{tikzcd}$$

Figure 4.2: An example database instance **db** for $q_2 = RRX$.

The situation is still different for $q_3 = ARRX$, for which it will be shown that $\mathsf{CERTAINTY}(q_3)$ is **coNP**-complete. Unlike our previous example, repeated rewinding of $ARRX$ into words of the language $ARR\,(R)^*\,X$ is not helpful. For example, in the database instance of Figure 4.3, every repair has a path that starts in 0 and is colored with a word in the language defined by $ARR\,(R)^*\,X$. However, the repair that contains $R(\underline{a}, c)$ does not satisfy q_3. Unlike Figure 4.2, the "bifurcation" in Figure 4.3 can be used as a gadget for showing **coNP**-completeness in Section 5.6.

Organization. Section 5.1 introduces the preliminaries. In Section 4.1, the statement of Theorem 4.2 gives the syntactic conditions for deciding the complexity of $\mathsf{CERTAINTY}(q)$ for path queries q. To prove this theorem, we view the rewinding operator from the perspectives of regular expressions and automata, which are presented in Sections 4.2 and 4.3 respectively. Sections 5.5

$$\begin{tikzcd}
 & & & b \arrow[r, "X"] & {} & \\
0 \arrow[r, "A"] & {} \arrow[r, "R"] & a \arrow[ur, "R"] \arrow[dr, "R"'] & & & \\
 & & & c \arrow[r, "R"'] & {} \arrow[r, "X"'] & {}
\end{tikzcd}$$

Figure 4.3: An example database instance **db** for $q_3 = ARRX$.

and 5.6 present, respectively, complexity upper bounds and lower bounds of our classification. In Section 4.6, we extend our classification result to path queries with constants. Section 7 concludes this chapter.

4.1 The Complexity Classification

We define syntactic conditions $\mathcal{C}_1$, $\mathcal{C}_2$, and $\mathcal{C}_3$ for path queries q. Let R be any relation name in q, and let u, v, and w be (possibly empty) words over the alphabet of relation names of q.

$\mathcal{C}_1$: Whenever $q = uRvRw$, q is a prefix of $uRvRvRw$.

$\mathcal{C}_2$: Whenever $q = uRvRw$, q is a factor of $uRvRvRw$; and whenever $q = uRv_1Rv_2Rw$ for consecutive occurrences of R, $v_1 = v_2$ or Rw is a prefix of Rv_1.

$\mathcal{C}_3$: Whenever $q = uRvRw$, q is a factor of $uRvRvRw$.

It is instructive to think of these conditions in terms of the rewinding operator introduced earlier: $\mathcal{C}_1$ is tantamount to saying that q is a prefix of every word to which q rewinds; and $\mathcal{C}_3$ says that q is a factor of every word to which q rewinds. These conditions can be checked in polynomial time in the length of the word q. The following result has an easy proof.

Proposition 4.1. *Let q be a path query. If q satisfies $\mathcal{C}_1$, then q satisfies $\mathcal{C}_2$; and if q satisfies $\mathcal{C}_2$, then q satisfies $\mathcal{C}_3$.*

The main part of this chapter comprises a proof of the following theorem, which refines the statement of Theorem 1.3 by adding syntactic conditions. The theorem is illustrated by Example 4.3.

Theorem 4.2. *For every path query q, the following complexity upper bounds obtain:*

- *if q satisfies $\mathcal{C}_1$, then $\mathsf{CERTAINTY}(q)$ is in* **FO***;*
- *if q satisfies $\mathcal{C}_2$, then $\mathsf{CERTAINTY}(q)$ is in* **NL***; and*

- *if q satisfies $\mathcal{C}_3$, then* $\mathsf{CERTAINTY}(q)$ *is in* **PTIME**.

Moreover, for every path query q, the following complexity lower bounds obtain:

- *if q violates $\mathcal{C}_1$, then* $\mathsf{CERTAINTY}(q)$ *is* **NL**-*hard;*
- *if q violates $\mathcal{C}_2$, then* $\mathsf{CERTAINTY}(q)$ *is* **PTIME**-*hard; and*
- *if q violates $\mathcal{C}_3$, then* $\mathsf{CERTAINTY}(q)$ *is* **coNP**-*complete.*

Example 4.3. The query $q_1 = RXRX$ rewinds to (and only to) $RX{\cdot}\underline{RX}{\cdot}RX$ and $R{\cdot}XR{\cdot}\underline{XR}{\cdot}X$, which both contain q_1 as a prefix. It is correct to conclude that $\mathsf{CERTAINTY}(q_1)$ is in **FO**.

The query $q_2 = RXRY$ rewinds only to $RX{\cdot}\underline{RX}{\cdot}RY$, which contains q_2 as a factor, but not as a prefix. Therefore, q_2 satisfies $\mathcal{C}_3$, but violates $\mathcal{C}_1$. Since q_2 vacuously satisfies $\mathcal{C}_2$ (because no relation name occurs three times in q_2), it is correct to conclude that $\mathsf{CERTAINTY}(q_2)$ is **NL**-complete.

The query $q_3 = RXRYRY$ rewinds to $RX \cdot \underline{RX} \cdot RYRY$, to $RXRY \cdot \underline{RXRY} \cdot RY$, and to $RX \cdot RY \cdot \underline{RY} \cdot RY = RXR{\cdot}YR{\cdot}\underline{YR}{\cdot}Y$. Since these words contain q_3 as a factor, but not always as a prefix, we have that q_3 satisfies $\mathcal{C}_3$ but violates $\mathcal{C}_1$. It can be verified that q_3 violates $\mathcal{C}_2$ by writing it as follows:

$$q_3 = \underbrace{\varepsilon}_{u}\underbrace{\boldsymbol{RX}}_{Rv_1}\underbrace{\boldsymbol{RY}}_{Rv_2}\underbrace{\boldsymbol{RY}}_{Rw}$$

We have $X = v_1 \neq v_2 = Y$, but $Rw = RY$ is not a prefix of $Rv_1 = RX$. Thus, $\mathsf{CERTAINTY}(q_3)$ is **PTIME**-complete.

Finally, the path query $q_4 = RXRXRYRY$ rewinds, among others, to $RX \cdot RXRY \cdot \underline{RXRY} \cdot RY$, which does not contain q_4 as a factor. It is correct to conclude that $\mathsf{CERTAINTY}(q_4)$ is **coNP**-complete. □

4.2 Regexes for $\mathcal{C}_1$, $\mathcal{C}_2$, and $\mathcal{C}_3$

In this section, we show that the conditions $\mathcal{C}_1$, $\mathcal{C}_2$, and $\mathcal{C}_3$ can be captured by regular expressions (or regexes) on path queries, which will be used in the proof of Theorem 4.2. Since these results are within the field of *combinatorics of words*, we will use the term *word* rather than *path query.*

Definition 4.1. We define four properties $\mathcal{B}_1$, $\mathcal{B}_{2a}$, $\mathcal{B}_{2b}$, $\mathcal{B}_3$ that a word q can possess:

$\mathcal{B}_1$**:** For some integer $k \geq 0$, there are words v, w such that vw is self-join-free and q is a prefix of $w\,(v)^k$.

$\mathcal{B}_{2a}$**:** For some integers $j, k \geq 0$, there are words u, v, w such that uvw is self-join-free and q is a factor of $(u)^j\,w\,(v)^k$.

$\mathcal{B}_{2b}$: For some integer $k \geq 0$, there are words u, v, w such that uvw is self-join-free and q is a factor of $(uv)^k\, wv$.

$\mathcal{B}_3$: For some integer $k \geq 0$, there are words u, v, w such that uvw is self-join-free and q is a factor of $uw\,(uv)^k$. □

We can identify each condition among $\mathcal{C}_1$, $\mathcal{C}_2$, $\mathcal{C}_3$, $\mathcal{B}_1$, $\mathcal{B}_{2a}$, $\mathcal{B}_{2b}$, $\mathcal{B}_3$ with the set of all words satisfying this condition. Note then that $\mathcal{B}_1 \subseteq \mathcal{B}_{2a} \cap \mathcal{B}_3$. The results in the remainder of this section can be summarized as follows:

- $\mathcal{C}_1 = \mathcal{B}_1$ (Lemma 4.1)
- $\mathcal{C}_2 = \mathcal{B}_{2a} \cup \mathcal{B}_{2b}$ (Lemma 4.3)
- $\mathcal{C}_3 = \mathcal{B}_{2a} \cup \mathcal{B}_{2b} \cup \mathcal{B}_3$ (Lemma 4.2)

Moreover, Lemma 4.3 characterizes $\mathcal{C}_3 \setminus \mathcal{C}_2$.

Lemma 4.1. *For every word q, the following are equivalent:*

1. *q satisfies $\mathcal{C}_1$; and*
2. *q satisfies $\mathcal{B}_1$.*

Lemma 4.2. *For every word q, the following are equivalent:*

1. *q satisfies $\mathcal{C}_3$; and*
2. *q satisfies $\mathcal{B}_{2a}$, $\mathcal{B}_{2b}$, or $\mathcal{B}_3$.*

Definition 4.2 (First and last symbol)**.** For a nonempty word u, we write $\mathsf{first}(u)$ and $\mathsf{last}(u)$ for, respectively, the first and the last symbol of u. □

Lemma 4.3. *Let q be a word that satisfies $\mathcal{C}_3$. Then, the following three statements are equivalent:*

1. *q violates $\mathcal{C}_2$;*
2. *q violates both $\mathcal{B}_{2a}$ and $\mathcal{B}_{2b}$; and*
3. *there are words u, v, w with $u \neq \varepsilon$ and uvw self-join-free such that either*
   - *(a) $v \neq \varepsilon$ and $\mathsf{last}(u) \cdot wuvu \cdot \mathsf{first}(v)$ is a factor of q; or*
   - *(b) $v = \varepsilon$, $w \neq \varepsilon$, and $\mathsf{last}(u) \cdot w\,(u)^2 \cdot \mathsf{first}(u)$ is a factor of q.*

The shortest word of the form (3a) in the preceding lemma is $RRSRS$ (let $u = R$, $v = S$, and $w = \varepsilon$), and the shortest word of the form (3b) is $RSRRR$ (let $u = R$, $v = \varepsilon$, and $w = S$). Note that since each of $\mathcal{C}_2$, $\mathcal{B}_{2a}$, and $\mathcal{B}_{2b}$ implies $\mathcal{C}_3$, it is correct to conclude that the equivalence between the first two items in Lemma 4.3 does not need the hypothesis that q must satisfy $\mathcal{C}_3$.

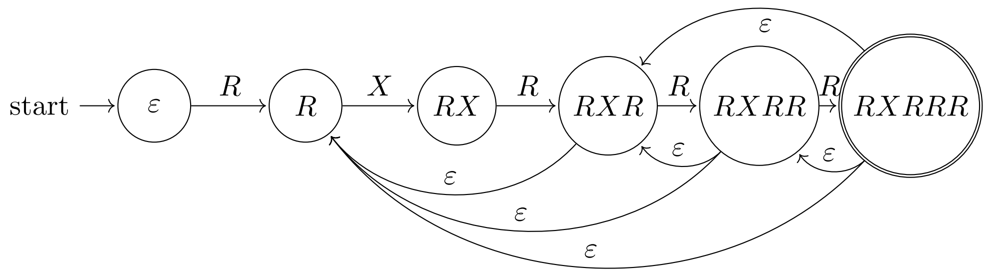


Figure 4.4: The $\mathsf{NFA}(q)$ automaton for the path query $q = RXRRR$.

4.3 Automaton-Based Perspective

In this section, we prove an important lemma, Lemma 4.7, which will be used for proving the complexity upper bounds in Theorem 4.2.

4.3.1 From Path Queries to Finite Automata

We can view a path query q as a word where the alphabet is the set of relation names. We now associate each path query q with a nondeterministic finite automaton (NFA), denoted $\mathsf{NFA}(q)$.

Definition 4.3 ($\mathsf{NFA}(q)$)**.** Every word q gives rise to a nondeterministic finite automaton (NFA) with ε-moves, denoted $\mathsf{NFA}(q)$, as follows.

States: The set of states is the set of prefixes of q. We include the empty word ε in the prefixes of q.

Forward transitions: If u and uR are states, then there is a transition with label R from state u to state uR. These transitions are said to be *forward*.

Backward transitions: If uR and wR are states such that $|u| < |w|$ (and therefore uR is a prefix of w), then there is a transition with label ε from state wR to state uR. These transitions are said to be *backward*, and capture the operation dubbed rewinding.

Initial and accepting states: The initial state is ε and the only accepting state is q. $\square$

Figure 4.4 shows the automaton $\mathsf{NFA}(RXRRR)$. Informally, the forward transitions capture the automaton that would accept the word $RXRRR$, while the backward transitions capture the existence of self-joins that allow an application of the rewind operator. We now take an alternative route for defining the language accepted by $\mathsf{NFA}(q)$, which straightforwardly results in Lemma 4.4. Then, Lemma 4.5 gives alternative ways for expressing $\mathcal{C}_1$ and $\mathcal{C}_3$.

Definition 4.4. Let q be a path query, represented as a word over the alphabet of relation names. We define the language $\mathcal{L}^{\looparrowright}(q)$ as the smallest set of words such that

(a) q belongs to $\mathcal{L}^{\looparrowright}(q)$; and

(b) *Rewinding:* if $uRvRw$ is in $\mathcal{L}^{\looparrowright}(q)$ for some relation name R and (possibly empty) words u, v and w, then $uRvRvRw$ is also in $\mathcal{L}^{\looparrowright}(q)$. $\square$

That is, $\mathcal{L}^{\looparrowright}(q)$ is the smallest language that contains q and is closed under rewinding.

Lemma 4.4. *For every path query q, the automaton* $\mathsf{NFA}(q)$ *accepts the language $\mathcal{L}^{\looparrowright}(q)$.*

Lemma 4.5. *Let q be a path query. Then,*

1. *q satisfies $\mathcal{C}_1$ if and only if q is a prefix of each $p \in \mathcal{L}^{\looparrowright}(q)$;*
2. *q satisfies $\mathcal{C}_3$ if and only if q is a factor of each $p \in \mathcal{L}^{\looparrowright}(q)$.*

Proof. $\boxed{\Longleftarrow \text{ in (1) and (2)}}$ This direction is trivial, because whenever $q = uRvRw$, we have that $uRvRvRw \in \mathcal{L}^{\looparrowright}(q)$.

We now show the $\Longrightarrow$ direction in both items. To this end, we call an application of the rule (b) in Definition 4.4 a *rewind*. By construction, each word in $\mathcal{L}^{\looparrowright}(q)$ can be obtained from q by using k rewinds, for some nonnegative integer k. Let q_k be a word in $\mathcal{L}^{\looparrowright}(q)$ that can be obtained from q by using k rewinds.

$\boxed{\Longrightarrow \text{ in (1)}}$ We use induction on k to show that q is a prefix of q_k. For he induction basis, $k = 0$, we have that q is a prefix of $q_0 = q$. We next show the induction step $k \to k+1$. Let $q_{k+1} = uRvRvRw$ where $q_k = uRvRw$ is a word in $\mathcal{L}^{\looparrowright}(q)$ obtained with k rewinds. By the induction hypothesis, we can assume a word s such that $q_k = q \cdot s$.

- If q is a prefix of $uRvR$, then $q_{k+1} = uRvRvRw$ trivially contains q as a prefix.
- If $uRvR$ is a proper prefix of q, let $q = uRvRt$ where t is nonempty. Since q satisfies $\mathcal{C}_1$, Rt is a prefix of Rv. Then $q_{k+1} = uRvRvRw$ contains $q = u \cdot Rv \cdot Rt$ as a prefix.

$\boxed{\Longrightarrow \text{ in (2)}}$ We use induction on k to show that q is a factor of q_k. For the induction basis, $k = 0$, we have that q is a prefix of $q_0 = q$. For the induction step, $k \to k+1$, let $q_{k+1} = uRvRvRw$ where $q_k = uRvRw$ is a word in $\mathcal{L}^{\looparrowright}(q)$ obtained with k rewinds. By the induction hypothesis, $q_k = uRvRw$ contains q as a factor. If q is a factor of either $uRvR$ or $RvRw$, then $q_{k+1} = uRvRvRw$ contains q as a factor. Otherwise, we may decompose $q_k = u^- q^- RvRq^+ w^+$ where $q = q^- RvRq^+$, $u = u^- q^-$ and $w = q^+ w^+$. Since q satisfies $\mathcal{C}_3$, the word $q^- RvRvRq^+$, which is a factor of q_{k+1}, contains q as a factor. $\square$

In the technical treatment, it will be convenient to consider the automaton obtained from $\mathsf{NFA}(q)$ by changing its start state, as defined next.

Definition 4.5. If u is a prefix of q (and thus u is a state in $\mathsf{NFA}(q)$), then $\mathsf{S\text{-}NFA}(q, u)$ is the automaton obtained from $\mathsf{NFA}(q)$ by letting the initial state be u instead of the empty word. Note that $\mathsf{S\text{-}NFA}(q, \varepsilon) = \mathsf{NFA}(q)$. It may be helpful to think of the first S in $\mathsf{S\text{-}NFA}(q, u)$ as "$\underline{\mathsf{S}}$tart at u." □

4.3.2 Reification Lemma

In this subsection, we first define how an automaton executes on a database instance. We then state an helping lemma which will be used in the proof of Lemma 4.7, which constitutes the main result of Section 4.3. To improve the readability and logical flow of our presentation, we postpone the proof of the helping lemma to Section 4.3.3.

Definition 4.6 (Automata on database instances)**.** Let $\mathbf{db}$ be a database instance. A *path (in* $\mathbf{db}$*)* is defined as a sequence $R_1(\underline{c_1}, c_2)$, $R_2(\underline{c_2}, c_3)$, $\dots$, $R_n(\underline{c_n}, c_{n+1})$ of facts in $\mathbf{db}$. Such a path is said to *start in* c_1. We call $R_1 R_2 \cdots R_n$ the *trace* of this path. A path is said to be *accepted* by an automaton if its trace is accepted by the automaton.

Let q be a path query and $\mathbf{r}$ be a consistent database instance. We define $\mathsf{start}(q, \mathbf{r})$ as the set containing all (and only) constants $c \in \mathsf{adom}(\mathbf{r})$ such that there is a path in $\mathbf{r}$ that starts in c and is accepted by $\mathsf{NFA}(q)$. □

Example 4.4. Consider the query $q_2 = RRX$ and the database instance of Figure 4.2. Let $\mathbf{r}_1$ and $\mathbf{r}_2$ be the repairs containing, respectively, $R(\underline{1}, 2)$ and $R(\underline{1}, 3)$. The only path with trace RRX in $\mathbf{r}_1$ starts in 1; and the only path with trace RRX in $\mathbf{r}_2$ starts in 0. The regular expression for $\mathcal{L}^{\looparrowright}(q)$ is $RR\,(R)^*\,X$. We have $\mathsf{start}(q, \mathbf{r}_1) = \{0, 1\}$ and $\mathsf{start}(q, \mathbf{r}_2) = \{0\}$. □

The following lemma tells us that, among all repairs, there is one that is inclusion-minimal with respect to $\mathsf{start}(q, \cdot)$. In the preceding example, the repair $\mathbf{r}_2$ minimizes $\mathsf{start}(q, \cdot)$.

Lemma 4.6. *Let q be a path query, and $\mathbf{db}$ a database instance. There exists a repair $\mathbf{r}^*$ of $\mathbf{db}$ such that for every repair $\mathbf{r}$ of $\mathbf{db}$, $\mathsf{start}(q, \mathbf{r}^*) \subseteq \mathsf{start}(q, \mathbf{r})$.*

Informally, we think of the next Lemma 4.7 as a *reification lemma.* The notion of *reifiable variable* was coined in [Wij12, Definition 8.5], to refer to a variable x in a query $\exists x\,(\varphi(x))$ such that whenever that query is true in every repair of a database instance, then there is a constant c such that $\varphi(c)$ is true in every repair. The following lemma captures a very similar concept.

Lemma 4.7 (Reification Lemma for $\mathcal{C}_3$)**.** *Let q be a path query that satisfies $\mathcal{C}_3$. Then, for every database instance* $\mathbf{db}$*, the following are equivalent:*

1. $\mathbf{db}$ *is a "yes"-instance of* $\mathsf{CERTAINTY}(q)$*; and*
2. *there exists a constant c (which depends on* $\mathbf{db}$*) such that for every repair* $\mathbf{r}$ *of* $\mathbf{db}$*, $c \in \mathsf{start}(q, \mathbf{r})$.*

Proof. $\boxed{1 \Longrightarrow 2}$ Assume (1). By Lemma 4.6, there exists a repair $\mathbf{r}^*$ of $\mathbf{db}$ such that for every repair $\mathbf{r}$ of $\mathbf{db}$, $\mathsf{start}(q, \mathbf{r}^*) \subseteq \mathsf{start}(q, \mathbf{r})$. Since $\mathbf{r}^*$ satisfies q, there is a path $R_1(\underline{c_1}, c_2)$, $R_2(\underline{c_2}, c_3)$, $\dots$, $R_n(\underline{c_n}, c_{n+1})$ in $\mathbf{r}^*$ such that $q = R_1 R_2 \cdots R_n$. Since q is accepted by $\mathsf{NFA}(q)$, we have $c_1 \in \mathsf{start}(q, \mathbf{r}^*)$. It follows that $c_1 \in \mathsf{start}(q, \mathbf{r})$ for every repair $\mathbf{r}$ of $\mathbf{db}$.

$\boxed{2 \Longrightarrow 1}$ Let $\mathbf{r}$ be any repair of $\mathbf{db}$. By our hypothesis that (2) holds true, there is some $c \in \mathsf{start}(q, \mathbf{r})$. Therefore, there is a path in $\mathbf{r}$ that starts in c and is accepted by $\mathsf{NFA}(q)$. Let p be the trace of this path. By Lemma 4.4, $p \in \mathcal{L}^{\looparrowright}(q)$. Since q satisfies $\mathcal{C}_3$ by the hypothesis of the current lemma, it follows by Lemma 4.5 that q is a factor of p. Consequently, there is a path in $\mathbf{r}$ whose trace is q. It follows that $\mathbf{r}$ satisfies q. $\square$

4.3.3 Proof of Lemma 4.6

We will use the following definition.

Definition 4.7 (States Set)**.** This definition is relative to a path query q. Let $\mathbf{r}$ be a consistent database instance, and let f be an R-fact in $\mathbf{r}$, for some relation name R. The *states set* of f in $\mathbf{r}$, denoted $\mathsf{ST}_q(f, \mathbf{r})$, is defined as the smallest set of states satisfying the following property, for all prefixes u of q:

if $\mathsf{S\text{-}NFA}(q, u)$ accepts a path in $\mathbf{r}$ that starts with f, then uR belongs to $\mathsf{ST}_q(f, \mathbf{r})$.

Note that if f is an R-fact, then all states in $\mathsf{S\text{-}NFA}(q, \mathbf{r})$ have R as their last relation name. $\square$

Example 4.5. Let $q = RRX$ and $\mathbf{r} = \{R(\underline{a}, b), R(\underline{b}, c), R(\underline{c}, d), X(\underline{d}, e), R(\underline{d}, e)\}$. Then $\mathsf{NFA}(q)$ has states $\{\varepsilon, R, RR, RRX\}$ and accepts the regular language $RR\,(R)^*\,X$. Since $\mathsf{S\text{-}NFA}(q, \varepsilon)$ accepts the path $R(\underline{b}, c)$, $R(\underline{c}, d)$, $X(\underline{c}, d)$, the states set $\mathsf{ST}_q(R(\underline{b}, c), \mathbf{r})$ contains $\varepsilon \cdot R = R$. Since the latter path is also accepted by $\mathsf{S\text{-}NFA}(q, R)$, we also have $R \cdot R \in \mathsf{ST}_q(R(\underline{b}, c), \mathbf{r})$. Finally, note that $\mathsf{ST}_q(R(\underline{d}, e), \mathbf{r}) = \emptyset$, because there is no path that contains $R(\underline{d}, e)$ and is accepted by $\mathsf{NFA}(q)$. $\square$

Lemma 4.8. *Let q be a path query, and* $\mathbf{r}$ *a consistent database instance. If $\mathsf{ST}_q(f, \mathbf{r})$ contains state uR, then it contains every state of the form vR with $|v| \geq |u|$.*

Proof. Assume $uR \in \mathsf{ST}_q(f, \mathbf{r})$. Then f is an R-fact and there is a path $f \cdot \pi$ in $\mathbf{r}$ that is accepted by $\mathsf{S\text{-}NFA}(q, u)$. Let vR be a state with $|v| > |u|$. Thus, by construction, $\mathsf{NFA}(q)$ has a backward transition with label ε from state vR to state uR.

It suffices to show that $f \cdot \pi$ is accepted by $\mathsf{S\text{-}NFA}(q, v)$. Starting in state v, $\mathsf{S\text{-}NFA}(q, v)$ traverses f (reaching state vR) and then uses the backward transition (with label ε) to reach the state uR. From there on, $\mathsf{S\text{-}NFA}(q, v)$ behaves like $\mathsf{S\text{-}NFA}(q, u)$. $\square$

From Lemma 4.8, it follows that $\mathsf{ST}_q(f, \mathbf{r})$ is completely determined by the shortest word in it.

Definition 4.8. Let q be a path query and $\mathbf{db}$ a database instance. For every fact $f \in \mathbf{db}$, we define:

$$\mathsf{cqaST}_q(f, \mathbf{db}) := \bigcap \{\mathsf{ST}_q(f, \mathbf{r}) \mid \mathbf{r} \text{ is a repair of } \mathbf{db} \text{ that contains } f\},$$

where $\bigcap X = \bigcap_{S \in X} S$. $\square$

It is to be noted here that whenever $\mathbf{r}_1$ and $\mathbf{r}_2$ are repairs containing f, then by Lemma 4.8, $\mathsf{ST}_q(f, \mathbf{r}_1)$ and $\mathsf{ST}_q(f, \mathbf{r}_2)$ are comparable by set inclusion. Therefore, informally, $\mathsf{cqaST}_q(f, \mathbf{db})$ is the $\subseteq$-minimal states set of f over all repairs that contain f.

Definition 4.9 (Preorder $\preceq_q$ on repairs)**.** Let $\mathbf{db}$ be a database instance. For all repairs $\mathbf{r}, \mathbf{s}$ of $\mathbf{db}$, we define $\mathbf{r} \preceq_q \mathbf{s}$ if for every $f \in \mathbf{r}$ and $g \in \mathbf{s}$ such that f and g are key-equal, we have $\mathsf{ST}_q(f, \mathbf{r}) \subseteq \mathsf{ST}_q(g, \mathbf{s})$.

Clearly, $\preceq_q$ is a reflexive and transitive binary relation on the set of repairs of $\mathbf{db}$. We write $\mathbf{r} \prec_q \mathbf{s}$ if both $\mathbf{r} \preceq_q \mathbf{s}$ and for some $f \in \mathbf{r}$ and $g \in \mathbf{s}$ such that f and g are key-equal, $\mathsf{ST}_q(f, \mathbf{r}) \subsetneq \mathsf{ST}_q(g, \mathbf{s})$. $\square$

Lemma 4.9. *Let q be a path query. For every database instance $\mathbf{db}$, there is a repair $\mathbf{r}^*$ of $\mathbf{db}$ such that for every repair $\mathbf{r}$ of $\mathbf{db}$, $\mathbf{r}^* \preceq_q \mathbf{r}$.*

Proof. Construct a repair $\mathbf{r}^*$ as follows. For every block $\mathbf{b}$ of $\mathbf{db}$, insert into $\mathbf{r}^*$ a fact f of $\mathbf{b}$ such that $\mathsf{cqaST}_q(f, \mathbf{db}) = \bigcap \{\mathsf{cqaST}_q(g, \mathbf{db}) \mid g \in \mathbf{b}\}$. More informally, we insert into $\mathbf{r}^*$ a fact f from $\mathbf{b}$ with a states set that is $\subseteq$-minimal over all repairs and all facts of $\mathbf{b}$. We first show the following claim.

Claim 4.1. For every fact f in $\mathbf{r}^*$, we have $\mathsf{ST}_q(f, \mathbf{r}^*) = \mathsf{cqaST}_q(f, \mathbf{db})$.

Proof. Let f_1 be an arbitrary fact in $\mathbf{r}^*$. We show $\mathsf{ST}_q(f_1, \mathbf{r}^*) = \mathsf{cqaST}_q(f_1, \mathbf{db})$.

$\boxed{\supseteq}$ Obvious, because $\mathbf{r}^*$ is itself a repair of $\mathbf{db}$ that contains f_1.

$\boxed{\subseteq}$ Let $f_1 = R_1(\underline{c_0}, c_1)$. Assume by way of a contradiction that there is $p_1 \in \mathsf{ST}_q(f_1, \mathbf{r}^*)$ such that $p_1 \notin \mathsf{cqaST}_q(f_1, \mathbf{db})$. Then, for some (possibly empty) prefix p_0 of q, there is a sequence:

$$p_0 \xrightarrow{\varepsilon} p_0' \xrightarrow{f_1 = R_1(\underline{c_0}, c_1)} p_1 \xrightarrow{\varepsilon} p_1' \xrightarrow{f_2 = R_2(\underline{c_1}, c_2)} p_2 \cdots p_{n-1} \xrightarrow{\varepsilon} p_{n-1}' \xrightarrow{f_n = R_n(\underline{c_{n-1}}, c_n)} p_n = q, \tag{4.1}$$

where $f_1, f_2, \ldots, f_n \in \mathbf{r}^*$, for each $i \in \{1, \ldots, n\}$, $p_i = p'_{i-1}R_i$, and for each $i \in \{0, \ldots, n-1\}$, either $p'_i = p_i$ or p'_i is a strict prefix of p_i such that p'_i and p_i agree on their rightmost relation name. Informally, the sequence (4.1) represents an accepting run of $\mathsf{S\text{-}NFA}(q, p_0)$ in $\mathbf{r}^*$. Since $p_1 \in \mathsf{ST}_q(f_1, \mathbf{r}^*) \setminus \mathsf{cqaST}_q(f_1, \mathbf{db})$, we can assume a largest index $\ell \in \{1, \ldots, n\}$ such that $p_\ell \in \mathsf{ST}_q(f_\ell, \mathbf{r}^*) \setminus \mathsf{cqaST}_q(f_\ell, \mathbf{db})$. By construction of $\mathbf{r}^*$, there is a repair $\mathbf{s}$ such that $f_\ell \in \mathbf{s}$ and $\mathsf{ST}_q(f_\ell, \mathbf{s}) = \mathsf{cqaST}_q(f_\ell, \mathbf{db})$. Consequently, $p_\ell \notin \mathsf{ST}_q(f_\ell, \mathbf{s})$. We distinguish two cases:

Case that $\ell = n$. Thus, the run (4.1) ends with

$$\cdots \quad p_{\ell-1} \xrightarrow{\varepsilon} p'_{\ell-1} \xrightarrow{f_\ell = R_\ell(\underline{c_{\ell-1}}, c_\ell)} p_\ell = q.$$

Thus, the rightmost relation name in q is R_ℓ. Since $f_\ell \in \mathbf{s}$, it is clear that $p_\ell \in \mathsf{ST}_q(f_\ell, \mathbf{s})$, a contradiction.

Case that $\ell < n$. Thus, the run (4.1) includes

$$\cdots \quad p_{\ell-1} \xrightarrow{\varepsilon} p'_{\ell-1} \xrightarrow{f_\ell = R_\ell(\underline{c_{\ell-1}}, c_\ell)} p_\ell \xrightarrow{\varepsilon} p'_\ell \xrightarrow{f_{\ell+1} = R_{\ell+1}(\underline{c_\ell}, c_{\ell+1})} p_{\ell+1} \quad \cdots,$$

where $\ell+1$ can be equal to n. Clearly, $p_{\ell+1} \in \mathsf{ST}_q(f_{\ell+1}, \mathbf{r}^*)$. Assume without loss of generality that $\mathbf{s}$ contains $f'_{\ell+1} := R_{\ell+1}(\underline{c_\ell}, c'_{\ell+1})$, which is key-equal to $f_{\ell+1}$ (possibly $c'_{\ell+1} = c_{\ell+1}$). From $p_\ell \notin \mathsf{ST}_q(f_\ell, \mathbf{s})$, it follows $p_{\ell+1} \notin \mathsf{ST}_q(f'_{\ell+1}, \mathbf{s})$. Consequently, $p_{\ell+1} \notin \mathsf{cqaST}_q(f'_{\ell+1}, \mathbf{db})$. By our construction of r^*, we have $p_{\ell+1} \notin \mathsf{cqaST}_q(f_{\ell+1}, \mathbf{db})$. Consequently, $p_{\ell+1} \in \mathsf{ST}_q(f_{\ell+1}, \mathbf{r}^*) \setminus \mathsf{cqaST}_q(f_{\ell+1}, \mathbf{db})$, which contradicts that ℓ was chosen to be the largest such an index possible.

The proof of Claim 4.1 is now concluded. $\triangleleft$

To conclude the proof of the lemma, let $\mathbf{r}$ be any repair of $\mathbf{db}$, and let $f \in \mathbf{r}^*$ and $f' \in \mathbf{r}$ be two key-equal facts in $\mathbf{db}$. By Claim 4.1 and the construction of $\mathbf{r}^*$, we have that $\mathsf{ST}_q(f, \mathbf{r}^*) = \mathsf{cqaST}_q(f, \mathbf{db}) \subseteq \mathsf{cqaST}_q(f', \mathbf{db}) \subseteq \mathsf{ST}_q(f', \mathbf{r})$, as desired. $\square$

We can now give the proof of Lemma 4.6.

Proof of Lemma 4.6. Let $\mathbf{db}$ be a database instance. Then by Lemma 4.9, there is a repair $\mathbf{r}^*$ of $\mathbf{db}$ such that for every repair $\mathbf{r}$ of $\mathbf{db}$, $\mathbf{r}^* \preceq_q \mathbf{r}$. It suffices to show that for every repair $\mathbf{r}$ of $\mathbf{db}$, $\mathsf{start}(q, \mathbf{r}^*) \subseteq \mathsf{start}(q, \mathbf{r})$. To this end, consider any repair $\mathbf{r}$ and $c \in \mathsf{start}(q, \mathbf{r}^*)$. Let R be the first relation name of q. Since $c \in \mathsf{start}(q, \mathbf{r}^*)$, there is $d \in \mathsf{adom}(\mathbf{r}^*)$ such that $R \in \mathsf{ST}_q(R(\underline{c}, d), \mathbf{r}^*)$. Then, there is a unique $d' \in \mathsf{adom}(\mathbf{r})$ such that $R(\underline{c}, d') \in \mathbf{r}$, where it is possible that $d' = d$. From $\mathbf{r}^* \preceq_q \mathbf{r}$, it follows $\mathsf{ST}_q(R(\underline{c}, d), \mathbf{r}^*) \subseteq \mathsf{ST}_q(R(\underline{c}, d'), \mathbf{r})$. Consequently, $R \in \mathsf{ST}_q(R(\underline{c}, d'), \mathbf{r})$, which implies $c \in \mathsf{start}(q, \mathbf{r})$. This conclude the proof. $\square$

```
Initialization Step:   $N \leftarrow \{\langle c, q\rangle \mid c \in \mathsf{adom}(\mathbf{db})\}$.
     Iterative Rule:        if   $uR$ is a prefix of $q$, and $R(\underline{c}, *)$ is a nonempty block in $\mathbf{db}$ s.t.
                                 for every $R(\underline{c}, y) \in \mathbf{db}$, $\langle y, uR\rangle \in N$
                          then
                                 $N \leftarrow N \cup \underbrace{\{\langle c, u\rangle\}}_{\text{forward}} \cup \underbrace{\{\langle c, w\rangle \mid \mathsf{NFA}(q) \text{ has an } \varepsilon\text{-transition from } w \text{ to } u\}}_{\text{backward}}$
```

Figure 4.5: Polynomial-time algorithm for computing $\{\langle c, u\rangle \mid \mathbf{db} \vdash_q \langle c, u\rangle\}$, for a fixed path query q satisfying $\mathcal{C}_3$.

4.4 Complexity Upper Bounds

We now show the complexity upper bounds of Theorem 4.2.

4.4.1 A PTIME Algorithm for $\mathcal{C}_3$

We now specify a polynomial-time algorithm for $\mathsf{CERTAINTY}(q)$, for path queries q that satisfy condition $\mathcal{C}_3$. The algorithm is based on the automata defined in Definition 4.5, and uses the concept defined next.

Definition 4.10 (Relation $\vdash_q$)**.** Let q be a path query and $\mathbf{db}$ a database instance. For every $c \in \mathsf{adom}(q)$ and every prefix u of q, we write $\mathbf{db} \vdash_q \langle c, u\rangle$ if every repair of $\mathbf{db}$ has a path that starts in c and is accepted by $\mathsf{S\text{-}NFA}(q, u)$. □

An algorithm that decides the relation $\vdash_q$ can be used to solve $\mathsf{CERTAINTY}(q)$ for path queries satisfying $\mathcal{C}_3$. Indeed, by Lemma 4.7, for path queries satisfying $\mathcal{C}_3$, $\mathbf{db}$ is a "yes"-instance for the problem $\mathsf{CERTAINTY}(q)$ if and only if there is a constant $c \in \mathsf{adom}(\mathbf{db})$ such that $\mathbf{db} \vdash_q \langle c, u\rangle$ with $u = \varepsilon$.

Figure 4.5 shows an algorithm that computes $\{\langle c, u\rangle \mid \mathbf{db} \vdash_q \langle c, u\rangle\}$ as the fixed point of a binary relation N. The *Initialization Step* inserts into N all pairs $\langle c, q\rangle$, which is correct because $\mathbf{db} \vdash_q \langle c, q\rangle$ holds vacuously, as q is the accepting state of $\mathsf{S\text{-}NFA}(q, q)$. Then, the *Iterative Rule* is executed until N remains unchanged; it intrinsically reflects the constructive proof of Lemma 4.9: $\mathbf{db} \vdash_q \langle c, u\rangle$ if and only if for every fact $f = R(\underline{c}, d) \in \mathbf{db}$, we have $uR \in \mathsf{cqaST}_q(f, \mathbf{db})$. Figure 4.6 shows an example run of the algorithm in Figure 4.5. The next lemma states the correctness of the algorithm.

Lemma 4.10. *Let q be a path query. Let $\mathbf{db}$ be a database instance. Let N be the output relation returned by the algorithm in Figure 4.5 on input $\mathbf{db}$. Then, for every $c \in \mathsf{adom}(\mathbf{db})$ and every prefix u of q,*

Iteration	Tuples added to N
init.	<0, RRX>, <1, RRX>, <2, RRX>, <3, RRX>, <4, RRX>, <5, RRX>
1	<4, RR>
2	<3, R>, <3, RR>
3	<2, R>, <2, RR>
4	<1, R>, <1, RR>
5	<0, R>, <0, RR>, <0, ε>

$$\begin{tikzcd}
0 \arrow[r, "R"] & 1 \arrow[r, "R"] \arrow[dr, "R"] & 2 \arrow[r, "R"] \arrow[d, "R"] & 3 \arrow[dl, "R"] \\
 & & 4 \arrow[r, "X"'] & 5
\end{tikzcd}$$

Figure 4.6: Example run of our algorithm for $q = RRX$, on the database instance **db** shown below.

$$\langle c, u\rangle \in N \textit{ if and only if } \mathbf{db} \vdash_q \langle c, u\rangle.$$

Proof. $\boxed{\Longleftarrow}$ Proof by contraposition. Assume $\langle c, u\rangle \notin N$. The proof shows the construction of a repair $\mathbf{r}$ of $\mathbf{db}$ such that $\mathbf{r}$ has no path that starts in c and is accepted by $\mathsf{S\text{-}NFA}(q, u)$. Such a repair shows $\mathbf{db} \nvdash_q \langle c, u\rangle$.

We explain which fact of an arbitrary block $R(\underline{a}, *)$ of $\mathbf{db}$ will be inserted in $\mathbf{r}$. Among all prefixes of q that end with R, let u_0R be the longest prefix such that $\langle a, u_0\rangle \notin N$. If such u_0R does not exist, then an arbitrarily picked fact of the block $R(\underline{a}, *)$ is inserted in $\mathbf{r}$. Otherwise, the *Iterative Rule* in Figure 4.5 entails the existence of a fact $R(\underline{a}, b)$ such that $\langle b, u_0R\rangle \notin N$. Then, $R(\underline{a}, b)$ is inserted in $\mathbf{r}$. We remark that this repair $\mathbf{r}$ is constructed in exactly the same way as the repair $\mathbf{r}^*$ built in the proof of Lemma 4.9.

Assume for the sake of contradiction that there is a path π in $\mathbf{r}$ that starts in c and is accepted by $\mathsf{S\text{-}NFA}(q, u)$. Let $\pi := R_1(\underline{c_0}, c_1)$, $R_2(\underline{c_1}, c_2)$, $\ldots$, $R_n(\underline{c_{n-1}}, c_n)$ where $c_0 = c$. Since $\langle c_0, u\rangle \notin N$ and $\langle c_n, q\rangle \in N$, there is a longest prefix u_0 of q, where $|u_0| \geq |u|$, and $i \in \{1, \ldots, n\}$ such that $\langle c_{i-1}, u_0\rangle \notin N$ and $\langle c_i, u_0R_i\rangle \in N$. From $\langle c_{i-1}, u_0\rangle \notin N$, it follows that $\mathbf{db}$ contains a fact $R_i(\underline{c_{i-1}}, d)$ such that $\langle d, u_0R_i\rangle \notin N$. Then $R_i(\underline{c_{i-1}}, c_i)$ would not be chosen in a repair, contradicting $R_i(\underline{c_{i-1}}, c_i) \in \mathbf{r}$.

$\boxed{\Longrightarrow}$ Assume that $\langle c, u\rangle \in N$. Let ℓ be the number of executions of the *Iterative Rule* that were used to insert $\langle c, u\rangle$ in N. We show $\mathbf{db} \vdash_q \langle c, u\rangle$ by induction on ℓ.

The basis of the induction, $\ell = 0$, holds because the *Initialization Step* is obviously correct. Indeed, since q is an accepting state of $\mathsf{S\text{-}NFA}(q, q)$, we have $\mathbf{db} \vdash_q \langle c, q\rangle$. For the inductive step, $\ell \to \ell + 1$, we distinguish two cases.

Case that $\langle c, u\rangle$ is added to N by the *forward* part of the *Iterative Rule*. That is, $\langle c, u\rangle$ is added because $\mathbf{db}$ has a block $\{R(\underline{c}, d_1), \ldots, R(\underline{c}, d_k)\}$ with $k \geq 1$ and for every $i \in \{1, \ldots, k\}$, we have that $\langle d_i, uR\rangle$ was added to N by a previous execution of the *Iterative Rule*. Let $\mathbf{r}$ be an arbitrary repair of $\mathbf{db}$. Since every repair contains exactly one fact from each block, we can assume $i \in \{1, \ldots, k\}$ such that $R(\underline{c}, d_i) \in \mathbf{r}$. By the induction hypothesis, $\mathbf{db} \vdash_q \langle d_i, uR\rangle$ and thus $\mathbf{r}$ has a path that starts in d_i and is accepted by $\mathsf{S\text{-}NFA}(q, uR)$. Clearly, this path can be left extended with $R(\underline{c}, d_i)$, and this left extended path is accepted by $\mathsf{S\text{-}NFA}(q, u)$. Note incidentally that the path in $\mathbf{r}$ may already use $R(\underline{c}, d_i)$, in which case the path is cyclic. Since $\mathbf{r}$ is an arbitrary repair, it is correct to conclude $\mathbf{db} \vdash_q \langle c, u\rangle$.

Case that $\langle c, u\rangle$ is added to N by the *backward* part of the *Iterative Rule*. Then, there exists a relation name S and words v, w such that $u = vSwS$, and $\langle c, u\rangle$ is added because $\langle c, vS\rangle$ was added in the same iteration. Then, $\mathsf{S\text{-}NFA}(q, u)$ has an ε-transition from state u to vS. Let $\mathbf{r}$

$$\varphi_q(N,x,z) := \left(\begin{array}{ll} & (\alpha(x) \wedge z = \text{‘}\mathtt{q}\text{’}) \\ \vee & \left(\bigvee_{\mathtt{uR} \leq \mathtt{q}} ((z = \text{‘}\mathtt{u}\text{’}) \wedge \exists y R(\underline{x}, y) \wedge \forall y\, (R(\underline{x}, y) \rightarrow N(y, \text{‘}\mathtt{uR}\text{’})))\right) \\ \vee & \left(\bigvee_{\substack{\varepsilon < \mathtt{u} < \mathtt{uv} \leq \mathtt{q} \\ \mathsf{last}(\mathtt{u}) = \mathsf{last}(\mathtt{v})}} (N(x, \text{‘}\mathtt{u}\text{’}) \wedge z = \text{‘}\mathtt{uv}\text{’})\right) \end{array}\right)$$

Figure 4.7: Definition of $\varphi_q(N,x,z)$. The predicate $\alpha(x)$ states that x is in the active domain, and $<$ is shorthand for *"is a strict prefix of"*.

be an arbitrary repair of **db**. By the reasoning in the previous case, **r** has a path that starts in c and is accepted by $\mathsf{S\text{-}NFA}(q, vS)$. We claim that **r** has a path that starts in c and is accepted by $\mathsf{S\text{-}NFA}(q, u)$. Indeed, $\mathsf{S\text{-}NFA}(q, u)$ can use the ε-transition to reach the state vS, and then behave like $\mathsf{S\text{-}NFA}(q, vS)$. This concludes the proof. □

The following corollary is now immediate.

Corollary 4.1. *Let q be a path query. Let **db** be a database instance, and $c \in \mathsf{adom}(\mathbf{db})$. Then, the following are equivalent:*

1. *$c \in \mathsf{start}(q, \mathbf{r})$ for every repair **r** of **db**; and*
2. *$\langle c, \epsilon \rangle \in N$, where N is the output of the algorithm in Figure 4.5.*

Finally, we obtain the following tractability result.

Lemma 4.11. *For each path query q satisfying $\mathcal{C}_3$, $\mathsf{CERTAINTY}(q)$ is expressible in Least Fixpoint Logic, and hence is in* **PTIME**.

Proof. For a path query q, define the following formula in LFP [Lib04]:

$$\psi_q(s,t) := \left[\mathbf{lfp}_{N,x,z}\varphi_q(N,x,z)\right](s,t), \tag{4.2}$$

where $\varphi_q(N,x,z)$ is given in Figure 4.7. Herein, $\alpha(x)$ denotes a first-order query that computes the active domain. That is, for every database instance **db** and constant c, $\mathbf{db} \models \alpha(c)$ if and only if $c \in \mathsf{adom}(\mathbf{db})$. Further, $u \leq v$ means that u is a prefix of v; and $u < v$ means that u is a proper prefix of v. Thus, $u < v$ if and only if $u \leq v$ and $u \neq v$. The formula $\varphi_q(N,x,z)$ is positive in N, which is a 2-ary predicate symbol. It is understood that the middle disjunction ranges over all nonempty prefixes uR of q (possibly $u = \varepsilon$). The last disjunction ranges over all pairs (u, uv) of distinct nonempty prefixes of q that agree on their last symbol. We used a different typesetting to distinguish the constant words $\mathtt{q}$, $\mathtt{uR}$, $\mathtt{uv}$ from first-order variables x, z. It is easily verified that the LFP query (5.7) expresses the algorithm of Figure 4.5. □

Since the formula (5.7) in the proof of Lemma 4.11 uses universal quantification, it is not in Existential Least Fixpoint Logic, which is equal to $\mathsf{DATALOG}_{\neg}$ [Lib04, Theorem 10.18].

4.4.2 FO-Rewritability for $\mathcal{C}_1$

We now show that if a path query q satisfies $\mathcal{C}_1$, then $\mathsf{CERTAINTY}(q)$ is in **FO**, and a first-order rewriting for q can be effectively constructed.

Definition 4.11 (First-order rewriting)**.** If q is a Boolean query such that $\mathsf{CERTAINTY}(q)$ is in **FO**, then a *(consistent) first-order rewriting* for q is a first-order sentence ψ such that for every database instance **db**, the following are equivalent:

1. **db** is a "yes"-instance of $\mathsf{CERTAINTY}(q)$; and
2. **db** satisfies ψ. □

Definition 4.12. If $q = \{R_1(\underline{x_1}, x_2),\ R_2(\underline{x_2}, x_3),\ \dots,\ R_k(\underline{x_k}, x_{k+1})\}$, $k \geq 1$, and c is a constant, then $q_{[c]}$ is the Boolean conjunctive query $q_{[c]} := \{R_1(\underline{c}, x_2), R_2(\underline{x_2}, x_3), \dots, R_k(\underline{x_k}, x_{k+1})\}$. □

Lemma 4.12. *For every nonempty path query q and constant c, the problem $\mathsf{CERTAINTY}(q_{[c]})$ is in* **FO**. *Moreover, it is possible to construct a first-order formula $\psi(x)$, with free variable x, such that for every constant c, the sentence $\exists x\,(\psi(x) \wedge x = c)$ is a first-order rewriting for $q_{[c]}$.*

Proof. The proof inductively constructs a first-order rewriting for $q_{[c]}$, where the induction is on the number n of atoms in q. For the basis of the induction, $n = 1$, we have $q_{[c]} = R(\underline{c}, y)$. Then, the first-order formula $\psi(x) = \exists y R(\underline{x}, y)$ obviously satisfies the statement of the lemma.

We next show the induction step, $n \to n+1$. Let $R(\underline{x_1}, x_2)$ be the left-most atom of q, and assume that $p := q \setminus \{R(\underline{x_1}, x_2)\}$ is a path query with $n \geq 1$ atoms. By the induction hypothesis, it is possible to construct a first-order formula $\varphi(z)$, with free variable z, such that for every constant d,

$$\exists z\,(\varphi(z) \wedge z = d) \text{ is a first-order rewriting for } p_{[d]}. \tag{4.3}$$

We now define $\psi(x)$ as follows:

$$\psi(x) = \exists y\,(R(\underline{x}, y)) \wedge \forall z\,(R(\underline{x}, z) \to \varphi(z))\,. \tag{4.4}$$

We will show that for every constant c, $\exists x\,(\psi(x) \wedge x = c)$ is a first-order rewriting for $q_{[c]}$. To this end, let **db** be a database instance. It remains to be shown that **db** is a "yes"-instance of $\mathsf{CERTAINTY}(q_{[c]})$ if and only if **db** satisfies $\exists x\,(\psi(x) \wedge x = c)$.

$\boxed{\Longleftarrow}$ Assume **db** satisfies $\exists x\,(\psi(x) \wedge x = c)$. Because of the conjunct $\exists y\,(R(\underline{x}, y))$ in (4.4), we have that **db** includes a block $R(\underline{c}, *)$. Let **r** be a repair of **db**. We need to show that **r** satisfies $q_{[c]}$.

Clearly, $\mathbf{r}$ contains $R(\underline{c}, d)$ for some constant d. Since $\mathbf{db}$ satisfies $\exists z\,(\varphi(z) \wedge z = d)$, the induction hypothesis (4.3) tells us that $\mathbf{r}$ satisfies $p_{[d]}$. It is then obvious that $\mathbf{r}$ satisfies $q_{[c]}$.

$\boxed{\Longrightarrow}$ Assume $\mathbf{db}$ is a "yes"-instance for $\mathsf{CERTAINTY}(q_{[c]})$. Then $\mathbf{db}$ must obviously satisfy $\exists y\,(R(\underline{c}, y))$. Therefore, $\mathbf{db}$ includes a block $R(\underline{c}, *)$. Let $\mathbf{r}$ be an arbitrary repair of $\mathbf{db}$. There exists d such that $R(\underline{c}, d) \in \mathbf{r}$. Since $\mathbf{r}$ satisfies $q_{[c]}$, it follows that $\mathbf{r}$ satisfies $p_{[d]}$. Since $\mathbf{r}$ is an arbitrary repair, the induction hypothesis (4.3) tells us that $\mathbf{db}$ satisfies $\exists z\,(\varphi(z) \wedge z = d)$. It is then clear that $\mathbf{db}$ satisfies $\exists x\,(\psi(x) \wedge x = c)$. □

Lemma 4.13. *For every path query q that satisfies $\mathcal{C}_1$, the problem $\mathsf{CERTAINTY}(q)$ is in* **FO**, *and a first-order rewriting for q can be effectively constructed.*

Proof. By Lemmas 4.5 and 4.7, a database instance $\mathbf{db}$ is a "yes"-instance for $\mathsf{CERTAINTY}(q)$ if and only if there is a constant c (which depends on $\mathbf{db}$) such that $\mathbf{db}$ is a "yes"-instance for $\mathsf{CERTAINTY}(q_{[c]})$. By Lemma 4.12, it is possible to construct a first-order rewriting $\exists x\,(\psi(x) \wedge x = c)$ for $q_{[c]}$. It is then clear that $\exists x\,(\psi(x))$ is a first-order rewriting for q. □

4.4.3 An NL Algorithm for $\mathcal{C}_2$

We show that $\mathsf{CERTAINTY}(q)$ is in **NL** if q satisfies $\mathcal{C}_2$ by expressing it in linear Datalog with stratified negation. The proof will use the syntactic characterization of $\mathcal{C}_2$ established in Lemma 4.3.

Lemma 4.14. *For every path query q that satisfies $\mathcal{C}_2$, the problem $\mathsf{CERTAINTY}(q)$ is in linear Datalog with stratified negation (and hence in* **NL***).*

In the remainder of this section, we develop the proof of Lemma 4.14.

Definition 4.13. Let q be a path query. We define $\mathsf{NFA}^{\min}(q)$ as the automaton that accepts w if w is accepted by $\mathsf{NFA}(q)$ and no proper prefix of w is accepted by $\mathsf{NFA}(q)$. □

It is well-known that such an automaton $\mathsf{NFA}^{\min}(q)$ exists.

Example 4.6. Let $q = RXRYR$. Then, $RXRYRYR$ is accepted by $\mathsf{NFA}(q)$, but not by $\mathsf{NFA}^{\min}(q)$, because the proper prefix $RXRYR$ is also accepted by $\mathsf{NFA}(q)$. □

Definition 4.14. Let q be a path query and $\mathbf{r}$ be a consistent database instance. We define $\mathsf{start}^{\min}(q, \mathbf{r})$ as the set containing all (and only) constants $c \in \mathsf{adom}(\mathbf{r})$ such that there is a path in $\mathbf{r}$ that starts in c and is accepted by $\mathsf{NFA}^{\min}(q)$. □

Lemma 4.15. *Let q be a path query. For every consistent database instance $\mathbf{r}$, we have that* $\mathsf{start}(q, \mathbf{r}) = \mathsf{start}^{\min}(q, \mathbf{r})$.

Proof. By construction, $\mathsf{start}^{\min}(q, \mathbf{r}) \subseteq \mathsf{start}(q, \mathbf{r})$. Next assume that $c \in \mathsf{start}(q, \mathbf{r})$ and let π be the path that starts in c and is accepted by $\mathsf{NFA}(q)$. Let π^{-} be the shortest prefix of π that is accepted by $\mathsf{NFA}(q)$. Since π^{-} starts in c and is accepted by $\mathsf{NFA}^{\min}(q)$, it follows $c \in \mathsf{start}^{\min}(q, \mathbf{r})$. □

Lemma 4.16. *Let $u \cdot v \cdot w$ be a self-join-free word over the alphabet of relation names. Let s be a suffix of uv that is distinct from uv. For every integer $k \geq 0$, $\mathsf{NFA}^{\min}(s\,(uv)^{k}\,wv)$ accepts the language of the regular expression $s\,(uv)^{k}\,(uv)^{*}\,wv$.*

Proof. Let $q = s\,(uv)^{k}\,wv$. Since $u{\cdot}v{\cdot}w$ is self-join-free, applying the rewinding operation, zero, one, or more times, in the part of q that precedes w will repeat the factor uv. This gives words of the form $s\,(uv)^{\ell}\,wv$ with $\ell \geq k$. The difficult case is where we rewind a factor of q that itself contains w as a factor. In this case, the rewinding operation will repeat a factor of the form $v_2\,(uv)^{\ell}\,wv_1$ such that $v = v_1v_2$ and $v_2 \neq \varepsilon$, which results in words of one of the following forms ($s = s_1 \cdot v_2$):

$$\begin{aligned}&\left(s\,(uv)^{\ell_1}\,uv_1\right)\cdot v_2\,(uv)^{\ell_2}\,wv_1\cdot\underline{v_2\,(uv)^{\ell_2}\,wv_1}\cdot(v_2);\ \text{or}\\ &(s_1)\cdot v_2\,(uv)^{\ell}\,wv_1\cdot\underline{v_2\,(uv)^{\ell}\,wv_1}\cdot(v_2).\end{aligned}$$

These words have a prefix belonging to the language of the regular expression $s\,(uv)^{k}\,(uv)^{*}\,wv$. □

Definition 4.15. Let $\mathbf{db}$ be a database instance, and q a path query.

For $a, b \in \mathsf{adom}(\mathbf{db})$, we write $\mathbf{db} \models a \xrightarrow{q} b$ if there exists a path in $\mathbf{db}$ from a to b with trace q. Even more formally, $\mathbf{db} \models a \xrightarrow{q} b$ if $\mathbf{db}$ contains facts $R_1(\underline{a_1}, a_2), R_2(\underline{a_2}, a_3), \ldots, R_{|q|}(\underline{a_{|q|}}, a_{|q|+1})$ such that $R_1R_2\cdots R_{|q|} = q$. We write $\mathbf{db} \models a \xrightarrow{q_1} b \xrightarrow{q_2} c$ as a shorthand for $\mathbf{db} \models a \xrightarrow{q_1} b$ and $\mathbf{db} \models b \xrightarrow{q_2} c$.

We write $\mathbf{db} \models a \overset{q}{\twoheadrightarrow} b$ if there exists a *consistent path* in $\mathbf{db}$ from a to b with trace q, where a path is called consistent if it does not contain two distinct key-equal facts.

A constant $c \in \mathsf{adom}(\mathbf{db})$ is called *terminal for q in* $\mathbf{db}$ if for some (possibly empty) proper prefix p of q, there is a consistent path in $\mathbf{db}$ with trace p that cannot be right extended to a consistent path in $\mathbf{db}$ with trace q. □

Note that for every $c \in \mathsf{adom}(\mathbf{db})$, we have $c \overset{\varepsilon}{\twoheadrightarrow} c$. Clearly, if q is self-join-free, then $c \xrightarrow{q} d$ implies $c \overset{q}{\twoheadrightarrow} d$ (the converse implication holds vacuously true).

Example 4.7. Let $\mathbf{db} = \{R(\underline{c}, d), S(\underline{d}, c), R(\underline{c}, e), T(\underline{e}, f)\}$. Then, c is terminal for $RSRT$ in $\mathbf{db}$ because the path $R(\underline{c}, d), S(\underline{d}, c)$ cannot be right extended to a consistent path with trace $RSRT$, because d has no outgoing T-edge. Note incidentally that $\mathbf{db} \models c \overset{RS}{\twoheadrightarrow} c \overset{RT}{\twoheadrightarrow} f$, but $\mathbf{db} \not\models c \overset{RSRT}{\twoheadrightarrow} f$. □

Lemma 4.17. *Let* **db** *be a database instance, and* $c \in \mathsf{adom}(\mathbf{db})$. *Let* q *be a path query. Then,* c *is terminal for* q *in* **db** *if and only if* **db** *is a "no"-instance of* $\mathsf{CERTAINTY}(q_{[c]})$, *with* $q_{[c]}$ *as defined by Definition 4.12.*

Proof. $\boxed{\Longrightarrow}$ Straightforward. $\boxed{\Longleftarrow}$ Assume **db** is a "no"-instance of $\mathsf{CERTAINTY}(q_{[c]})$. Then, there is a repair $\mathbf{r}$ of **db** such that $\mathbf{r} \not\models q_{[c]}$. The empty path is a path in $\mathbf{r}$ that starts in c and has trace ε, which is a prefix of q. We can therefore assume a longest prefix p of q such there exists a path π in $\mathbf{r}$ that starts in c and has trace p. Since $\mathbf{r}$ is consistent, π is consistent. From $\mathbf{r} \not\models q_{[c]}$, it follows that p is a proper prefix of q. By Definition 4.15, c is terminal for q in **db**. □

We can now give the proof of Lemma 4.14.

Proof of Lemma 4.14. Assume q satisfies $\mathcal{C}_2$. By Lemma 4.3, q satisfies $\mathcal{B}_{2a}$ or $\mathcal{B}_{2b}$. We treat the case that q satisfies $\mathcal{B}_{2b}$ (the case that q satisfies $\mathcal{B}_{2a}$ is even easier). We have that q is a factor of $(uv)^k\, wv$, where k is chosen as small as possible, and uvw is self-join-free. The proof is straightforward if $k = 0$; we assume $k \geq 1$ from here on. To simplify notation, we will show the case where q is a suffix of $(uv)^k\, wv$; our proof can be easily extended to the case where q is not a suffix, at the price of some extra notation. There is a suffix s of uv such that $q = s\,(uv)^{k-1}\, wv$.

We first define a unary predicate P (which depends on q) such that $\mathbf{db} \models P(d)$ if for some $\ell \geq 0$, there are constants $d_0, d_1, \dots, d_\ell \in \mathsf{adom}(\mathbf{db})$ with $d_0 = d$ such that:

(i) $\mathbf{db} \models d_0 \xrightarrow{uv} d_1 \xrightarrow{uv} d_2 \xrightarrow{uv} \cdots \xrightarrow{uv} d_\ell$;

(ii) for every $i \in \{0, 1, \dots, \ell\}$, d_i is terminal for wv in **db**; and

(iii) either d_ℓ is terminal for uv in **db**, or $d_\ell \in \{d_0, \dots, d_{\ell-1}\}$.

Claim 4.2. The definition of the predicate P does not change if we replace item (i) by the stronger requirement that for every $i \in \{0, 1, \dots, \ell - 1\}$, there exists a path π_i from d_i to d_{i+1} with trace uv such that the composed path $\pi_0 \cdot \pi_1 \cdots \pi_{\ell-1}$ is consistent.

Proof. It suffices to show the following statement by induction on increasing l:

> whenever there exist $l \geq 1$ and constants $d_0, d_1, \dots, d_l$ with $d_0 = d$ such that conditions (i), (ii), and (iii) hold, there exist another constant $k \geq 1$ and constants $c_0, c_1, \dots, c_k$ with $c_0 = d$ such that conditions (i), (ii), and (iii) hold, and, moreover, for each $i \in \{0, 1, \dots, k-1\}$, there exists a path π_i from c_i to c_{i+1} such that the composed path $\pi_0 \cdot \pi_1 \cdots \pi_{k-1}$ is consistent.

Basis $l = 1$. Then we have $\mathbf{db} \models d_0 \xrightarrow{uv} d_1$, witnessed by a path π_0. Since uv is self-join-free, the path π_0 is consistent. The claim thus follows with $k = l = 1$, $c_0 = d_0$ and $c_1 = d_1$.

Inductive step $l \rightarrow l+1$. Assume that the statement holds for any integer in $\{1, 2, \ldots, l\}$. Suppose that there exist $l \geq 2$ and constants $d_0, d_1, \ldots, d_{l+1}$ with $d_0 = d$ such that conditions (i), (ii), and (iii) hold.

For $i \in \{0, \ldots, l\}$, let π_i be a path with trace uv from d_i to d_{i+1} in **db**. The claim holds if the composed path $\pi_0 \cdot \pi_1 \cdots \pi_l$ is consistent, with $k = l+1$ and $c_i = d_i$ for $i \in \{0, 1, \ldots, l+1\}$. Now, assume that for some $i < j$, the paths that show $\mathbf{db} \models d_i \xrightarrow{uv} d_{i+1}$ and $\mathbf{db} \models d_j \xrightarrow{uv} d_{j+1}$ contain, respectively, $R(\underline{a}, b_1)$ and $R(\underline{a}, b_2)$ with $b_1 \neq b_2$. It is easily verified that

$$\mathbf{db} \models d_0 \xrightarrow{uv} d_1 \xrightarrow{uv} d_2 \xrightarrow{uv} \cdots \xrightarrow{uv} d_i \xrightarrow{uv} d_{j+1} \xrightarrow{uv} \cdots \xrightarrow{uv} d_{l+1},$$

where the number of uv-steps is strictly less than $l+1$. Informally, we follow the original path until we reach $R(\underline{a}, b_1)$, but then follow $R(\underline{a}, b_2)$ instead of $R(\underline{a}, b_1)$, and continue on the path that proves $\mathbf{db} \models d_j \xrightarrow{uv} d_{j+1}$. Then the claim holds by applying the inductive hypothesis on constants $d_0, d_1, \ldots, d_i, d_{j+1}, \ldots, d_{l+1}$.

The proof is now complete. □

Since we care about the expressibility of the predicate P in Datalog, Claim 4.2 is not cooked into the definition of P. The idea is the same as in an **NL**-algorithm for reachability: if there exists a directed path from s to t, then there is such a path without repeated vertices; but we do not care for repeated vertices when computing reachability.

Claim 4.3. The definition of predicate P does not change if we require that for $i \in \{0, 1, \ldots, \ell-1\}$, d_i is not terminal for uv in **db**.

Proof. Assume that for some $0 \leq i < \ell$, d_i is terminal for uv in **db**. Then, all conditions in the definition are satisfied by choosing ℓ equal to j. □

Claim 4.3 is not cooked into the definition of P to simplify the the encoding of P in Datalog.

Next, we define a unary predicate O such that $\mathbf{db} \models O(c)$ for a constant c if $c \in \mathsf{adom}(\mathbf{db})$ and one of the following holds true:

1. c is terminal for $s\,(uv)^{k-1}$ in **db**; or
2. there is a constant $d \in \mathsf{adom}(\mathbf{db})$ such that both $\mathbf{db} \models c \xrightarrow{s(uv)^{k-1}} d$ and $\mathbf{db} \models P(d)$.

Claim 4.4. Let $c \in \mathsf{adom}(\mathbf{db})$. The following are equivalent:

(I) there is a repair $\mathbf{r}$ of **db** that contains no path that starts in c and whose trace is in the language of the regular expression $s\,(uv)^{k-1}\,(uv)^*\,wv$; and

(II) $\mathbf{db} \models O(c)$.

Proof. Let $wv = S_0S_1 \cdots S_{m-1}$ and $uv = R_0R_1 \cdots R_{n-1}$.

$\boxed{\text{(I)} \implies \text{(II)}}$ Assume that item (I) holds true. Let the first relation name of s be R_i. Starting from c, let π be a maximal (possibly infinite) path in $\mathbf{r}$ that starts in c and has trace $R_iR_{i+1}R_{i+2}\cdots$, where addition is modulo n. Since $\mathbf{r}$ is consistent, π is deterministic. Since $\mathbf{r}$ is finite, π contains only finitely many distinct edges. Therefore, π ends either in a loop or in an edge $R_j(\underline{d}, e)$ such that $\mathbf{db} \models \neg\exists y R_{j+1}(\underline{e}, y)$ (recall that $\mathbf{r}$ contains a fact from every block of $\mathbf{db}$). Assume that π has a prefix π' with trace $s\,(uv)^{k-1}$; if e occurs at the non-primary key position of the last R_{n-1}-fact of π' or of any R_{n-1}-fact occurring afterwards in π, then it follows from item (I) that there exist a (possibly empty) prefix pS_j of wv and a constant $f \in \mathsf{adom}(\mathbf{r})$ such that $\mathbf{r} \models e \xrightarrow{p} f$ and $\mathbf{db} \models \neg\exists y S_j(\underline{f}, y)$. It is now easily verified that $\mathbf{db} \models O(c)$.

$\boxed{\text{(II)} \implies \text{(I)}}$ Assume $\mathbf{db} \models O(c)$. It is easily verified that the desired result holds true if c is terminal for $s\,(uv)^{k-1}$ in $\mathbf{db}$. Assume from here on that c is not terminal for $s\,(uv)^{k-1}$ in $\mathbf{db}$. That is, for every repair $\mathbf{r}$ of $\mathbf{db}$, there is a constant d such that $\mathbf{r} \models c \xrightarrow{s(uv)^{k-1}} d$. Then, there is a consistent path α with trace $s\,(uv)^{k-1}$ from c to some constant $d \in \mathsf{adom}(\mathbf{db})$ such that $\mathbf{db} \models P(d)$, using the stronger definition of P implied by Claims 4.2 and 4.3. Let $d_0, \dots, d_\ell$ be as in our (stronger) definition of $P(d)$, that is, first, $d_1, \dots, d_{\ell-1}$ are not terminal for uv in $\mathbf{db}$ (cf. Claim 4.3), and second, there is a $\subseteq$-minimal consistent subset π of $\mathbf{db}$ such that $\pi \models d_0 \xrightarrow{uv} d_1 \xrightarrow{uv} d_2 \xrightarrow{uv} \cdots \xrightarrow{uv} d_\ell$ (cf. Claim 4.2). We construct a repair $\mathbf{r}$ as follows:

1. insert into $\mathbf{r}$ all facts of π;
2. for every $i \in \{0, \dots, \ell\}$, d_i is terminal for wv in $\mathbf{db}$. We ensure that $\mathbf{r} \models d_i \xrightarrow{S_0S_1\cdots S_{j_i}} e_i$ for some $j_i \in \{0, \dots, m-2\}$ and some constant e_i such that $\mathbf{db} \models \neg\exists y S_{j_i+1}(\underline{e_i}, y)$;
3. if d_ℓ is terminal for uv in $\mathbf{db}$, then we ensure that $\mathbf{r} \models d_\ell \xrightarrow{R_0R_1\cdots R_j} e$ for some $j \in \{0, \dots, n-2\}$ and some constant e such that $\mathbf{db} \models \neg\exists y S_{j+1}(\underline{e}, y)$;
4. insert into $\mathbf{r}$ the facts of α that are not key-equal to a fact already in $\mathbf{r}$; and
5. complete $\mathbf{r}$ into a $\subseteq$-maximal consistent subset of $\mathbf{db}$.

Since $\mathbf{r}$ is a repair of $\mathbf{db}$, there exists a path δ with trace $s\,(uv)^{k-1}$ in $\mathbf{r}$ that starts from c. If $\delta \neq \alpha$, then δ must contain a fact of π that was inserted in step 1. Consequently, no matter whether $\delta = \alpha$ or $\delta \neq \alpha$, the endpoint of δ belongs to $\{d_0, \dots, d_\ell\}$. It follows that there is a (possibly empty) path from δ's endpoint to d_ℓ whose trace is of the form $(uv)^*$. Two cases can occur:

- d_ℓ is terminal for uv in $\mathbf{db}$.
- d_ℓ is not terminal for uv in $\mathbf{db}$. Then there is $j \in \{0, \dots, \ell-1\}$ such that $d_j = d_\ell$. Then, there is a path of the form $(uv)^*$ that starts from δ's endpoint and eventually loops.

Since, by construction, each d_i is terminal for wv in $\mathbf{r}$, it will be the case that δ cannot be extended to a path in $\mathbf{r}$ whose trace is of the form $s\,(uv)^k\,(uv)^*\,wv$. $\square$

Claim 4.5. The unary predicate O is expressible in linear Datalog with stratified negation.

Proof. The construction of the linear Datalog program is straightforward. Concerning the computation of predicates P and O, note that it can be checked in **FO** whether or not a constant c is terminal for some path query q, by Lemmas 4.12 and 4.17. The only need for recursion comes from condition (i) in the definition of the predicate P, which searches for a directed path of a particular form. It is easily seen that any path query satisfying $\mathcal{B}_{2b}$ admits such a program for the predicate O. $\square$

By Lemmas 4.7, 4.15, and 4.16, the following are equivalent:

(a) **db** is a "no"-instance of $\mathsf{CERTAINTY}(q)$; and

(b) for every constant $c_i \in \mathsf{adom}(q)$, there is a repair $\mathbf{r}$ of **db** that contains no path that starts in c_i and whose trace is in the language of the regular expression $s\,(uv)^{k-1}\,(uv)^*\,wv$.

By Claim 4.4, item (b) holds true if and only if for every $c \in \mathsf{adom}(\mathbf{db})$, $\mathbf{db} \models \neg O(c)$. It follows from Claim 4.5 that the latter test is in linear Datalog with stratified negation, which concludes the proof of Lemma 4.14. $\square$

4.5 Complexity Lower Bounds

In this section, we show the complexity lower bounds of Theorem 4.2. For a path query $q = \{R_1(x_1, x_2), \ldots, R_k(x_k, x_{k+1})\}$ and constants a, b, we define the following database instances:

$$\begin{aligned}
\phi_a^b[q] &:= \{R_1(a, \square_2), R_2(\square_2, \square_3), \ldots, R_k(\square_k, b)\}\\
\phi_a^\perp[q] &:= \{R_1(a, \square_2), R_2(\square_2, \square_3), \ldots, R_k(\square_k, \square_{k+1})\}\\
\phi_\perp^b[q] &:= \{R_1(\square_1, \square_2), R_2(\square_2, \square_3), \ldots, R_k(\square_k, b)\}
\end{aligned}$$

where the symbols $\square_i$ denoted fresh constants not occurring elsewhere. Significantly, two occurrences of $\square_i$ will represent different constants.

4.5.1 NL-Hardness

We first show that if a path query violates $\mathcal{C}_1$, then $\mathsf{CERTAINTY}(q)$ is **NL**-hard, and therefore not in **FO**.

Lemma 4.18. *If a path query q violates $\mathcal{C}_1$, then* $\mathsf{CERTAINTY}(q)$ *is* **NL**-*hard.*

Proof. Assume that q does not satisfy $\mathcal{C}_1$. Then, there exists a relation name R such that $q = uRvRw$ and q is not a prefix of $uRvRvRw$. It follows that Rw is not a prefix of $RvRw$. Since $Rv \neq \varepsilon$, there exists no (conjunctive query) homomorphism from q to uRw.

The problem $\mathsf{REACHABILITY}$ takes as input a directed graph $G(V, E)$ and two vertices $s, t \in V$, and asks whether G has a directed path from s to t. This problem is **NL**-complete and remains **NL**-complete when the inputs are acyclic graphs. Recall that **NL** is closed under complement. We present a first-order reduction from $\mathsf{REACHABILITY}$ to the complement of $\mathsf{CERTAINTY}(q)$, for acyclic directed graphs.

Let $G = (V, E)$ be an acyclic directed graph and $s, t \in V$. Let $G' = (V \cup \{s', t'\}, E \cup \{(s', s), (t, t')\})$, where s', t' are fresh vertices. We construct an input instance $\mathbf{db}$ for $\mathsf{CERTAINTY}(q)$ as follows:

- for each vertex $x \in V \cup \{s'\}$, we add $\phi_{\perp}^{x}[u]$;
- for each edge $(x, y) \in E \cup \{(s', s), (t, t')\}$, we add $\phi_{x}^{y}[Rv]$; and
- for each vertex $x \in V$, we add $\phi_{x}^{\perp}[Rw]$.

This construction can be executed in **FO**. Figure 4.8 shows an example of the above construction. Observe that the only conflicts in $\mathbf{db}$ occur in R-facts outgoing from a same vertex.

$$\begin{tikzcd}
{} \arrow[d, "u"] & {} \arrow[d, "u"] & {} \arrow[d, "u"] & {} \arrow[d, "u"] & \\
s' \arrow[r, "Rv"] & s \arrow[r, "Rv"] \arrow[d, "Rw"] & a \arrow[r, "Rv"] \arrow[d, "Rw"] & t \arrow[r, "Rv"] \arrow[d, "Rw"] & t' \\
 & {} & {} & {} &
\end{tikzcd}$$

Figure 4.8: Database instance for the **NL**-hardness reduction from the graph G with $V = \{s, a, t\}$ and $E = \{(s, a), (a, t)\}$.

We now show that there exists a directed path from s to t in G if and only if there exists a repair of $\mathbf{db}$ that does not satisfy q.

$\boxed{\Longrightarrow}$ Suppose that there is a directed path from s to t in G. Then, G' has a directed path $P = s, x_0, x_1, \ldots, t, t'$. Then, consider the repair $\mathbf{r}$ that chooses the first R-fact from $\phi_{x}^{y}[Rv]$ for each edge (x, y) on the path P, and the first R-fact from $\phi_{y}^{\perp}[Rw]$ for each y not on the path P. We show that $\mathbf{r}$ falsifies q. Assume for the sake of contradiction that $\mathbf{r}$ satisfies q. Then, there exists a valuation θ for the variables in q such that $\theta(q) \subseteq \mathbf{r}$. Since, as argued in the beginning of this proof, there exists no (conjunctive query) homomorphism from q to uRw, it must be that all facts in $\theta(q)$ belong to a path in $\mathbf{r}$ with trace $u\,(Rv)^{k}$, for some $k \geq 0$. Since, by construction, no constants are repeated on such paths, there exists a (conjunctive query) homomorphism from q to

$u\,(Rv)^k$, which implies that Rw is a prefix of $RvRw$, a contradiction. We conclude by contradiction that $\mathbf{r}$ falsifies q.

$\boxed{\Longleftarrow}$ Proof by contradiction. Suppose that there is no directed path from s to t in G. Let $\mathbf{r}$ be any repair of $\mathbf{db}$; we will show that $\mathbf{r}$ satisfies q. Indeed, there exists a maximal path $P = x_0, x_1, \ldots, x_n$ such that $x_0 = s'$, $x_1 = s$, and $\phi_{x_i}^{x_{i+1}}[Rv] \subseteq \mathbf{r}$. By construction, s' cannot reach t' in G', and thus $x_n \neq t'$. Since P is maximal, we must have $\phi_{x_n}^{\perp}[Rw] \subseteq \mathbf{r}$. Then $\phi_{\perp}^{x_{n-1}}[u] \cup \phi_{x_{n-1}}^{x_n}[Rv] \cup \phi_{x_n}^{\perp}[Rw]$ satisfies q. □

4.5.2 coNP-Hardness

Next, we show the **coNP**-hard lower bound.

Lemma 4.19. *If a path query q violates $\mathcal{C}_3$, then* $\mathsf{CERTAINTY}(q)$ *is* **coNP***-hard.*

Proof. If q does not satisfy $\mathcal{C}_3$, then there exists a relation R such that $q = uRvRw$ and q is not a factor of $uRvRvRw$. Note that this means that there is no homomorphism from q to $uRvRvRw$. Also, u must be nonempty (otherwise, $q = RvRw$ is trivially a suffix of $RvRvRw$). Let S be the first relation of u.

The proof is a first-order reduction from SAT to the complement of $\mathsf{CERTAINTY}(q)$. The problem SAT asks whether a given propositional formula in CNF has a satisfying truth assignment.

Given any formula ψ for SAT, we construct an input instance $\mathbf{db}$ for $\mathsf{CERTAINTY}(q)$ as follows:

- for each variable z, we add $\phi_z^{\perp}[Rw]$ and $\phi_z^{\perp}[RvRw]$;
- for each clause C and positive literal z of C, we add $\phi_C^z[u]$;
- for each clause C and variable z that occurs in a negative literal of C, we add $\phi_C^z[uRv]$.

This construction can be executed in **FO**. Figure 4.9 depicts an example of the above construction. Intuitively, $\phi_z^{\perp}[Rw]$ corresponds to setting the variable z to true, and $\phi_z^{\perp}[RvRw]$ to false. There are two types of conflicts that occur in $\mathbf{db}$. First, we have conflicting facts of the form $S(\underline{C}, *)$; resolving this conflict corresponds to the clause C choosing one of its literals. Moreover, for each variable z, we have conflicting facts of the form $R(\underline{z}, *)$; resolving this conflict corresponds to the variable z choosing a truth assignment.

We show now that ψ has a satisfying truth assignment if and only if there exists a repair of $\mathbf{db}$ that does not satisfy q.

$\boxed{\Longrightarrow}$ Assume that there exists a satisfying truth assignment σ for ψ. Then for any clause C, there exists a variable $z_C \in C$ whose corresponding literal is true in C under σ. Consider the repair $\mathbf{r}$ that:

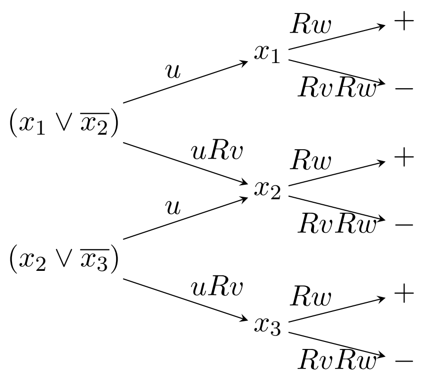


Figure 4.9: Database instance for the **coNP**-hardness reduction from the formula $\psi = (x_1 \lor \overline{x_2}) \land (x_2 \lor \overline{x_3})$.

- for each variable z, it chooses the first R-fact of $\phi_z^{\perp}[Rw]$ if $\sigma(z)$ is true, otherwise the first R-fact of $\phi_z^{\perp}[RvRw]$;
- for each clause C, it chooses the first S-fact of $\phi_C^{z}[u]$ if z_C is positive in C, or the first S-fact of $\phi_C^{z}[uRv]$ if z_C is negative in C.

Assume for the sake of contradiction that $\mathbf{r}$ satisfies q. Then we must have a homomorphism from q to either uRw or $uRvRvRw$. But the former is not possible, while the latter contradicts $\mathcal{C}_3$. We conclude by contradiction that $\mathbf{r}$ falsifies q.

$\boxed{\Longleftarrow}$ Suppose that there exists a repair $\mathbf{r}$ of **db** that falsifies q. Consider the assignment σ:

$$\sigma(z) = \begin{cases} \text{true} & \text{if } \phi_z^{\perp}[Rw] \subseteq \mathbf{r} \\ \text{false} & \text{if } \phi_z^{\perp}[RvRw] \subseteq \mathbf{r} \end{cases}$$

We claim that σ is a satisfying truth assignment for ψ. Indeed, for each clause C, the repair must have chosen a variable z in C. If z appears as a positive literal in C, then $\phi_C^{z}[u] \subseteq \mathbf{r}$. Since $\mathbf{r}$ falsifies q, we must have $\phi_z^{\perp}[Rw] \subseteq \mathbf{r}$. Thus, $\sigma(z)$ is true and C is satisfied. If z appears in a negative literal, then $\phi_C^{z}[uRv] \subseteq \mathbf{r}$. Since $\mathbf{r}$ falsifies q, we must have $\phi_z^{\perp}[RvRw] \subseteq \mathbf{r}$. Thus, $\sigma(z)$ is false and C is again satisfied. □

4.5.3 PTIME-Hardness

Finally, we show the **PTIME**-hard lower bound.

Lemma 4.20. *If a path query q violates $\mathcal{C}_2$, then* $\mathsf{CERTAINTY}(p)$ *is* **PTIME**-*hard.*

Proof. Suppose q violates $\mathcal{C}_2$. If q also violates $\mathcal{C}_3$, then the problem $\mathsf{CERTAINTY}(q)$ is **PTIME**-hard since it is **coNP**-hard by Lemma 4.19. Otherwise, it is possible to write $q = uRv_1Rv_2Rw$, with three consecutive occurrences of R such that $v_1 \neq v_2$ and Rw is not a prefix of Rv_1. Let v be the maximal path query such that $v_1 = vv_1^+$ and $v_2 = vv_2^+$. Thus $v_1^+ \neq v_2^+$ and the first relation names of v_1^+ and v_2^+ are different.

Our proof is a reduction from the Monotone Circuit Value Problem (MCVP) known to be **PTIME**-complete [Gol77]:

Problem: MCVP

Input: A monotone Boolean circuit C on inputs $x_1, x_2, \ldots, x_n$ and output gate o; an assignment $\sigma : \{x_i \mid 1 \leq i \leq n\} \to \{0, 1\}$.

Question: What is the value of the output o under σ?

We construct an instance **db** for CERTAINTY(q) as follows:

- for the output gate o, we add $\phi_{\perp}^{o}[uRv_1]$;
- for each input variable x with $\sigma(x) = 1$, we add $\phi_{x}^{\perp}[Rv_2Rw]$;
- for each gate g, we add $\phi_{\perp}^{g}[u]$ and $\phi_{g}^{\perp}[Rv_2Rw]$;
- for each AND gate $g = g_1 \wedge g_2$, we add

$$\phi_{g}^{g_1}[Rv_1] \cup \phi_{g}^{g_2}[Rv_1].$$

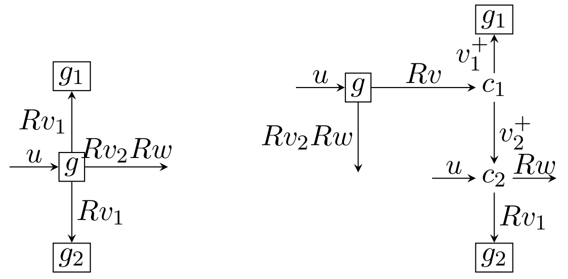
  Here, g_1 and g_2 can be gates or input variables; and
- for each OR gate $g = g_1 \vee g_2$, we add

$$\begin{aligned} \phi_{g}^{c_1}[Rv] \cup \phi_{c_1}^{g_1}[v_1^+] \quad &\cup \phi_{c_1}^{c_2}[v_2^+] \\ \cup\, \phi_{\perp}^{c_2}[u] \quad \cup \phi_{c_2}^{g_2}[Rv_1] &\cup \phi_{c_2}^{\perp}[Rw] \end{aligned}$$

  where c_1, c_2 are fresh constants.

This construction can be executed in **FO**. An example of the gadget constructions is shown in Figure 4.10. We next show that the output gate o is evaluated to 1 under σ if and only if each repair of **db** satisfies q.

$\boxed{\Longrightarrow}$ Suppose the output gate o is evaluated to 1 under σ. Consider any repair $\mathbf{r}$. We construct a sequence of gates starting from o, with the invariant that every gate g evaluates to 1, and there is a path of the form uRv_1 in $\mathbf{r}$ that ends in g. The output gate o evaluates to 1, and also we have that $\phi_{\perp}^{o}[uRv_1] \subseteq \mathbf{r}$ by construction. Suppose that we are at gate g. If there is a Rv_2Rw path in $\mathbf{r}$ that starts in g, the sequence ends and the query q is satisfied. Otherwise, we distinguish two cases:



(a) AND gate (b) OR gate

Figure 4.10: Gadgets for the **PTIME**-hardness reduction.

1. $g = g_1 \wedge g_2$. Then, we choose the gate with $\phi_g^{g_i}[Rv_1] \subseteq \mathbf{r}$. Since both gates evaluate to 1 and $\phi_\perp^g[u] \subseteq \mathbf{r}$, the invariant holds for the chosen gate.
2. $g = g_1 \vee g_2$. If g_1 evaluates to 1, we choose g_1. Observe that $\phi_\perp^g[u] \cup \phi_g^{c_1}[Rv] \cup \phi_{c_1}^{g_1}[v_1^+]$ creates the desired uRv_1 path. Otherwise g_2 evaluates to 1. If $\phi_{c_2}^\perp[Rw] \subseteq \mathbf{r}$, then there is a path with trace uRv_1 ending in g, and a path with trace Rv_2Rw starting in g, and therefore $\mathbf{r}$ satisfies q. If $\phi_{c_2}^\perp[Rw] \nsubseteq \mathbf{r}$, we choose g_2 and the invariant holds.

If the query is not satisfied at any point in the sequence, we will reach an input variable x evaluated at 1. But then there is an outgoing Rv_2Rw path from x, which means that q must be satisfied.

$\boxed{\Longleftarrow}$ Proof by contraposition. Assume that o is evaluated to 0 under σ. We construct a repair $\mathbf{r}$ as follows, for each gate g:

- if g is evaluated to 1, we choose the first R-fact in $\phi_g^\perp[Rv_2Rw]$;
- if $g = g_1 \wedge g_2$ and g is evaluated to 0, let g_i be the gate or input variable evaluated to 0. We then choose $\phi_g^{g_i}[Rv_1]$;
- if $g = g_1 \vee g_2$ and g is evaluated to 0, we choose $\phi_g^{c_1}[Rv]$; and
- if $g = g_1 \vee g_2$, we choose $\phi_{c_2}^{g_2}[Rv_1]$.

For a path query p, we write $\mathsf{head}(p)$ for the variable at the key-position of the first atom, and $\mathsf{rear}(p)$ for the variable at the non-key position of the last atom.

Assume for the sake of contradiction that $\mathbf{r}$ satisfies q. Then, there exists some valuation θ such that $\theta(uRv_1Rv_2Rw) \subseteq \mathbf{r}$. Then the gate $g^* := \theta(\mathsf{head}(Rv_1))$ is evaluated to 0 by construction. Let $g_1 := \theta(\mathsf{rear}(Rv_1))$. By construction, for $g^* = g_1 \wedge g_2$ or $g^* = g_1 \vee g_2$, we must have $\phi_g^{g_1}[Rv_1] \subseteq \mathbf{r}$ and g_1 is a gate or an input variable also evaluated to 0. By our construction of $\mathbf{r}$, there is no path with trace Rv_2Rw outgoing from g_1. However, $\theta(Rv_2Rw) \subseteq \mathbf{r}$, this can only happen when g_1 is an OR gate, and one of the following occurs:

- Case that $|Rw| \leq |Rv_1|$, and the trace of $\theta(Rv_2Rw)$ is a prefix of $Rvv_2^+Rv_1$. Then Rw is a prefix of Rv_1, a contradiction.
- Case that $|Rw| > |Rv_1|$, and $Rvv_2^+Rv_1$ is a prefix of the trace of $\theta(Rv_2Rw)$. Consequently, Rv_1 is a prefix of Rw. Then, for every $k \geq 1$, $\mathcal{L}^{\looparrowright}(q)$ contains $uRv_1\left(Rv_2\right)^k Rw$. It is now easily verified that for large enough values of k, uRv_1Rv_2w is not a factor of $uRv_1\left(Rv_2\right)^k Rw$. By Lemmas 4.5 and 4.19, $\mathsf{CERTAINTY}(q)$ is **coNP**-hard. □

4.6 Path Queries with Constants

We now extend our complexity classification of $\mathsf{CERTAINTY}(q)$ to path queries in which constants can occur.

Definition 4.16 (Generalized path queries)**.** A *generalized path query* is a Boolean conjunctive query of the following form:

$$q = \{R_1(\underline{s_1}, s_2), R_2(\underline{s_2}, s_3), \dots, R_k(\underline{s_k}, s_{k+1})\}, \tag{4.5}$$

where s_1, s_2,..., s_{k+1} are constants or variables, all distinct, and R_1, R_2,..., R_k are (not necessarily distinct) relation names. Significantly, every constant can occur at most twice: at a non-primary-key position and the next primary-key-position.

The *characteristic prefix* of q, denoted by $\mathsf{char}(q)$, is the longest prefix

$$\{R_1(\underline{s_1}, s_2), R_2(\underline{s_2}, s_3), \dots, R_\ell(\underline{s_\ell}, s_{\ell+1})\}, 0 \leq \ell \leq k$$

such that no constant occurs among s_1, s_2, ..., s_ℓ (but $s_{\ell+1}$ can be a constant). Clearly, if q is constant-free, then $\mathsf{char}(q) = q$. □

Example 4.8. If $q = \{R(\underline{x}, y),\ S(\underline{y}, 0),\ T(\underline{0}, 1),\ R(\underline{1}, w)\}$, where 0 and 1 are constants, then $\mathsf{char}(q) = \{R(\underline{x}, y),\ S(\underline{y}, 0)\}$. □

The following lemma implies that if a generalized path query q starts with a constant, then $\mathsf{CERTAINTY}(q)$ is in $\mathbf{FO}$. This explains why the complexity classification in the remainder of this section will only depend on $\mathsf{char}(q)$.

Lemma 4.21. *For any generalized path query q, $\mathsf{CERTAINTY}(p)$ is in $\mathbf{FO}$, where $p := q \setminus \mathsf{char}(q)$.*

Lemma 4.21 is an immediate corollary of Lemma 4.24, which states that whenever a generalized path query starts with a constant, then $\mathsf{CERTAINTY}(q)$ is in $\mathbf{FO}$. Its proof needs two helping lemmas.

Lemma 4.22. *Let $q = q_1 \cup q_2 \cup \cdots \cup q_k$ be a Boolean conjunctive query such that for all $1 \leq i < j \leq k$, $\mathsf{vars}(q_i) \cap \mathsf{vars}(q_j) = \emptyset$. Then, the following are equivalent for every database instance $\mathbf{db}$:*

1. *$\mathbf{db}$ is a "yes"-instance for $\mathsf{CERTAINTY}(q)$; and*
2. *for each $1 \leq i \leq k$, $\mathbf{db}$ is a "yes"-instance for $\mathsf{CERTAINTY}(q_i)$.*

Proof. We give the proof for $k = 2$. The generalization to larger k is straightforward.

$\boxed{1 \implies 2}$ Assume that (1) holds true. Then each repair $\mathbf{r}$ of $\mathbf{db}$ satisfies q, and therefore satisfies both q_1 and q_2. Therefore, $\mathbf{db}$ is a "yes"-instance for both $\mathsf{CERTAINTY}(q_1)$ and $\mathsf{CERTAINTY}(q_2)$.

$\boxed{2 \implies 1}$ Assume that (2) holds true. Let $\mathbf{r}$ be any repair of $\mathbf{db}$. Then there are valuations μ from $\mathsf{vars}(q_1)$ to $\mathsf{adom}(\mathbf{db})$, and θ from $\mathsf{vars}(q_2)$ to $\mathsf{adom}(\mathbf{db})$ such that $\mu(q_1) \subseteq \mathbf{r}$ and $\theta(q_2) \subseteq \mathbf{r}$. Since $\mathsf{vars}(q_1) \cap \mathsf{vars}(q_2) = \emptyset$ by construction, we can define a valuation σ as follows, for every variable $z \in \mathsf{vars}(q_1) \cup \mathsf{vars}(q_2)$:

$$\sigma(z) = \begin{cases} \mu(z) & \text{if } z \in \mathsf{vars}(q_1) \\ \theta(z) & \text{if } z \in \mathsf{vars}(q_2) \end{cases}$$

From $\sigma(q) = \sigma(q_1) \cup \sigma(q_2) = \mu(q_1) \cup \theta(q_2) \subseteq \mathbf{r}$, it follows that $\mathbf{r}$ satisfies q. Therefore, $\mathbf{db}$ is a "yes"-instance for $\mathsf{CERTAINTY}(q)$. □

Lemma 4.23. *Let q be a generalized path query with*

$$q = \{R_1(\underline{s_1}, s_2), R_2(\underline{s_2}, s_3), \ldots, R_k(\underline{s_k}, c)\},$$

where c is a constant, and each s_i is either a constant or a variable for all $i \in \{1, \ldots, k\}$. Let

$$p = \{R_1(\underline{s_1}, s_2), R_2(\underline{s_2}, s_3), \ldots, R_k(\underline{s_k}, s_{k+1}), N(\underline{s_{k+1}}, s_{k+2})\},$$

where s_{k+1}, s_{k+2} are fresh variables to q and N is a fresh relation to q. Then there exists a first-order reduction from $\mathsf{CERTAINTY}(q)$ to $\mathsf{CERTAINTY}(p)$.

Proof. Let $\mathbf{db}$ be an instance for $\mathsf{CERTAINTY}(q)$ and consider the instance $\mathbf{db} \cup \{N(\underline{c}, d)\}$ for $\mathsf{CERTAINTY}(p)$ where d is a fresh constant to $\mathsf{adom}(\mathbf{db})$.

We show that $\mathbf{db}$ is a "yes"-instance for $\mathsf{CERTAINTY}(q)$ if and only if $\mathbf{db} \cup \{N(\underline{c}, d)\}$ is a "yes"-instance for $\mathsf{CERTAINTY}(p)$.

$\boxed{\implies}$ Assume $\mathbf{db}$ is a "yes"-instance for $\mathsf{CERTAINTY}(q)$. Let $\mathbf{r}$ be any repair of $\mathbf{db} \cup \{N(\underline{c}, d)\}$, and thus $\mathbf{r} \setminus \{N(\underline{c}, d)\}$ is a repair for $\mathbf{db}$. Then there exists a valuation μ with $\mu(q) \subseteq \mathbf{r} \setminus \{N(\underline{c}, d)\}$. Consider the valuation μ^+ from $\mathsf{vars}(q) \cup \{s_{k+1}, s_{k+2}\}$ to $\mathsf{adom}(\mathbf{db}) \cup \{c, d\}$ that agrees with μ on

$\mathsf{vars}(q)$ and maps additionally $\mu^+(s_{k+1}) = c$ and $\mu^+(s_{k+2}) = d$. We thus have $\mu^+(p) \subseteq \mathbf{r}$. It is correct to conclude that $\mathbf{db} \cup \{N(\underline{c}, d)\}$ is a "yes"-instance for $\mathsf{CERTAINTY}(p)$.

$\boxed{\Longleftarrow}$ Assume that $\mathbf{db} \cup \{N(\underline{c}, d)\}$ is a "yes"-instance for the problem $\mathsf{CERTAINTY}(p)$. Let $\mathbf{r}$ be any repair of $\mathbf{db}$. Then $\mathbf{r} \cup \{N(\underline{c}, d)\}$ is a repair of $\mathbf{db} \cup \{N(\underline{c}, d)\}$, and thus there exists some valuation θ with $\theta(p) \subseteq \mathbf{r} \cup \{N(\underline{c}, d)\}$. Since $\mathbf{db}$ contains only one N-fact, we have $\theta(s_{k+1}) = c$. It follows that $\theta(q) \subseteq \mathbf{r}$, as desired. □

Lemma 4.24. *Let q be a generalized path query with*

$$q = \{R_1(\underline{s_1}, s_2), R_2(\underline{s_2}, s_3), \ldots, R_k(\underline{s_k}, s_{k+1})\}$$

where s_1 is a constant, and each s_i is either a constant or a variable for all $i \in \{2, \ldots, k+1\}$. Then the problem $\mathsf{CERTAINTY}(q)$ is in **FO**.

Proof. Let the $1 = j_1 < j_2 < \cdots < j_\ell \leq k+1$ be all the indexes j such that s_j is a constant for some $\ell \geq 1$. Let $j_{\ell+1} = k+1$. Then for each $i \in \{1, 2, \ldots, \ell\}$, the query

$$q_i = \bigcup_{j_i \leq j < j_{i+1}} \{R_j(\underline{s_j}, s_{j+1})\}$$

is a generalized path query where each s_{j_i} is a constant.

We claim that $\mathsf{CERTAINTY}(q_i)$ is in **FO** for each $1 \leq i \leq \ell$. Indeed, if $s_{j_{i+1}}$ is a variable, then the claim follows by Lemma 4.12; if $s_{j_{i+1}}$ is a constant, then the claim follows by Lemma 4.23 and Lemma 4.12.

Since by construction, $q = q_1 \cup q_2 \cup \cdots \cup q_\ell$, we conclude that $\mathsf{CERTAINTY}(q)$ is in **FO** by Lemma 4.22. □

The proof of Lemma 4.21 is now simple.

Proof of Lemma 4.21. If q contains no constants, the lemma holds trivially. Otherwise, the problem $\mathsf{CERTAINTY}(p)$ is in **FO** by Lemma 4.24. □

We now introduce some definitions and notations used in our complexity classification. The following definition introduces a convenient syntactic shorthand for characteristic prefixes previously defined in Definition 4.16.

Definition 4.17. Let $q = \{R_1(\underline{x_1}, x_2),\ R_2(\underline{x_2}, x_3),\ \ldots,\ R_k(\underline{x_k}, x_{k+1})\}$ be a path query. We write $[\![q, c]\!]$ for the generalized path query obtained from q by replacing x_{k+1} with the constant c. The constant-free path query q will be denoted by $[\![q, \top]\!]$, where $\top$ is a distinguished special symbol. □

Definition 4.18 (Prefix homomorphism)**.** Let

$$\begin{aligned} q &= \{R_1(\underline{s_1}, s_2), R_2(\underline{s_2}, s_3), \dots, R_k(\underline{s_k}, s_{k+1})\} \\ p &= \{S_1(\underline{t_1}, t_2), S_2(\underline{t_2}, t_3), \dots, R_\ell(\underline{s_\ell}, s_{\ell+1})\} \end{aligned}$$

be generalized path queries. A *homomorphism from q to p* is a substitution θ for the variables in q, extended to be the identity on constants, such that for every $i \in \{1, \dots, k\}$, $R_i(\underline{\theta(s_i)}, \theta(s_{i+1})) \in p$. Such a homomorphism is a *prefix homomorphism* if $\theta(s_1) = t_1$. □

Example 4.9. Let $q = \{R(\underline{x}, y),\ R(\underline{y}, 1),\ S(\underline{1}, z)\}$, and $p = \{R(\underline{x}, y),\ R(\underline{y}, z),\ R(\underline{y}, 1)\}$. Then $\mathsf{char}(q) = \{R(\underline{x}, y), R(\underline{y}, 1)\} = [\![RR, 1]\!]$ and $p = [\![RRR, 1]\!]$. There is a homomorphism from $\mathsf{char}(q)$ to p, but there is no prefix homomorphism from $\mathsf{char}(q)$ to p. □

The following conditions generalize $\mathcal{C}_1$, $\mathcal{C}_2$, and $\mathcal{C}_3$ from constant-free path queries to generalized path queries. Let γ be either a constant or the distinguished symbol $\top$.

$\mathcal{D}_1$**:** Whenever $\mathsf{char}(q) = [\![uRvRw, \gamma]\!]$, there is a prefix homomorphism from $\mathsf{char}(q)$ to the generalized path query $[\![uRvRvRw, \gamma]\!]$.

$\mathcal{D}_2$**:** Whenever $\mathsf{char}(q) = [\![uRvRw, \gamma]\!]$, there is a homomorphism from $\mathsf{char}(q)$ to $[\![uRvRvRw, \gamma]\!]$; and whenever $\mathsf{char}(q) = [\![uRv_1Rv_2Rw, \gamma]\!]$ for consecutive occurrences of R, $v_1 = v_2$ or there is a prefix homomorphism from $[\![Rw, \gamma]\!]$ to $[\![Rv_1, \gamma]\!]$.

$\mathcal{D}_3$**:** Whenever $\mathsf{char}(q) = [\![uRvRw, \gamma]\!]$, there is a homomorphism from $\mathsf{char}(q)$ to $[\![uRvRvRw, \gamma]\!]$.

It is easily verified that if $\gamma = \top$, then $\mathcal{D}_1$, $\mathcal{D}_2$, and $\mathcal{D}_3$ are equivalent to, respectively, $\mathcal{C}_1$, $\mathcal{C}_2$, and $\mathcal{C}_3$. Likewise, the following theorem degenerates to Theorem 4.2 for path queries without constants.

Theorem 4.3. *For every generalized path query q, the following complexity upper bounds obtain:*

- *if q satisfies $\mathcal{D}_1$, then* $\mathsf{CERTAINTY}(q)$ *is in* **FO***;*
- *if q satisfies $\mathcal{D}_2$, then* $\mathsf{CERTAINTY}(q)$ *is in* **NL***; and*
- *if q satisfies $\mathcal{D}_3$, then* $\mathsf{CERTAINTY}(q)$ *is in* **PTIME***.*

The following complexity lower bounds obtain:

- *if q violates $\mathcal{D}_1$, then* $\mathsf{CERTAINTY}(q)$ *is* **NL***-hard;*
- *if q violates $\mathcal{D}_2$, then* $\mathsf{CERTAINTY}(q)$ *is* **PTIME***-hard; and*
- *if q violates $\mathcal{D}_3$, then* $\mathsf{CERTAINTY}(q)$ *is* **coNP***-complete.*

4.6.1 Upper bounds in Theorem 4.3

The proof of Theorem 4.3 requires the notion of an *extended query* of a generalized path query.

Definition 4.19 (Extended query)**.** Let q be a generalized path query. The *extended query* of q, denoted by $\mathsf{ext}(q)$, is defined as follows:

- if q does not contain any constant, then $\mathsf{ext}(q) := q$;
- otherwise, $\mathsf{char}(q) = \{R_1(\underline{x_1}, x_2),\ R_2(\underline{x_2}, x_3),\ \dots,\ R_\ell(\underline{x_\ell}, c)\}$ for some constant c. In this case, we define
$$\mathsf{ext}(q) := \{R_1(\underline{x_1}, x_2), \dots, R_\ell(\underline{x_\ell}, x_{\ell+1}), N(\underline{x_{\ell+1}}, x_{\ell+2})\},$$
where $x_{\ell+1}$ and $x_{\ell+2}$ are fresh variables and N is a fresh relation name not occurring in q. □

By definition, $\mathsf{ext}(q)$ does not contain any constant.

Example 4.10. Let $q = R(\underline{x}, y), S(\underline{y}, 0), T(\underline{0}, 1), R(\underline{1}, w)$ where 0 and 1 are constants. We have $\mathsf{ext}(q) = R(\underline{x}, y), S(\underline{y}, z), N(\underline{z}, u)$. □

We show two lemmas which, taken together, show that the problem $\mathsf{CERTAINTY}(q)$ is first-order reducible to $\mathsf{CERTAINTY}(\mathsf{ext}(q))$, for every generalized path query q.

Lemma 4.25. *For every generalized path query q, there is a first-order reduction from the problem $\mathsf{CERTAINTY}(q)$ to $\mathsf{CERTAINTY}(\mathsf{char}(q))$.*

Proof. Let $p := q \setminus \mathsf{char}(q)$. Since $\mathsf{vars}(\mathsf{char}(q)) \cap \mathsf{vars}(p) = \emptyset$, Lemmas 4.22 and 4.24 imply that the following are equivalent for every database instance $\mathbf{db}$:

1. $\mathbf{db}$ is a "yes"-instance for $\mathsf{CERTAINTY}(q)$; and
2. $\mathbf{db}$ is a "yes"-instance for $\mathsf{CERTAINTY}(\mathsf{char}(q))$ and a "yes"-instance for $\mathsf{CERTAINTY}(p)$.

To conclude the proof, it suffices to observe that $\mathsf{CERTAINTY}(p)$ is in **FO** by Lemma 4.24. □

Lemma 4.26. *For every generalized path query q, there is a first-order reduction from the problem $\mathsf{CERTAINTY}(\mathsf{char}(q))$ to $\mathsf{CERTAINTY}(\mathsf{ext}(q))$.*

Proof. Let q be a generalized path query. If q contains no constants, the lemma trivially obtains because $\mathsf{char}(q) = \mathsf{ext}(q) = q$. If q contains at least one constant, then there exists a first-order reduction from $\mathsf{CERTAINTY}(\mathsf{char}(q))$ to $\mathsf{CERTAINTY}(\mathsf{ext}(q))$ by Lemma 4.23. □

Lemma 4.27. *Let q be a generalized path query that contains at least one constant. If q satisfies $\mathcal{D}_3$, then q satisfies $\mathcal{D}_2$ and $\mathsf{ext}(q)$ satisfies $\mathcal{C}_2$.*

Proof. Assume that q satisfies $\mathcal{D}_3$. Let $\mathsf{char}(q) = [\![p, c]\!]$ for some constant c. We have $\mathsf{ext}(q) = p \cdot N$ where N is a fresh relation name not occurring in p.

We first argue that $\mathsf{ext}(q)$ is a factor of every word to which $\mathsf{ext}(q)$ rewinds. To this end, let $\mathsf{ext}(q) = uRvRwN$ where $p = uRvRw$. Since q satisfies $\mathcal{D}_3$, there exists a homomorphism from $\mathsf{char}(q) = [\![uRvRw, c]\!]$ to $[\![uRvRvRw, c]\!]$, implying that $uRvRw$ is a suffix of $uRvRvRw$. It follows that $uRvRwN$ is a suffix of $uRvRvRwN$. Hence $\mathsf{ext}(q)$ satisfies $\mathcal{C}_3$.

The remaining test for $\mathcal{C}_2$ is where $\mathsf{ext}(q) = uRv_1Rv_2RwN$ for consecutive occurrences of R. We need to show that either $v_1 = v_2$ or RwN is a prefix of Rv_1 (or both). We have $p = uRv_1Rv_2Rw$. Since q satisfies $\mathcal{D}_3$, there exists a homomorphism from $\mathsf{char}(q) = [\![uRv_1Rv_2Rw, c]\!]$ to $[\![uRv_1Rv_2Rv_2Rw, c]\!]$. Since c is a constant, the homomorphism must map Rv_1 to Rv_2, implying that $v_1 = v_2$. It is correct to conclude that q satisfies $\mathcal{D}_2$ and $\mathsf{ext}(q)$ satisfies $\mathcal{C}_2$. □

Lemma 4.28. *For every generalized path query q,*

- *if q satisfies $\mathcal{D}_1$, then $\mathsf{ext}(q)$ satisfies $\mathcal{C}_1$;*
- *if q satisfies $\mathcal{D}_2$, then $\mathsf{ext}(q)$ satisfies $\mathcal{C}_2$; and*
- *if q satisfies $\mathcal{D}_3$, then $\mathsf{ext}(q)$ satisfies $\mathcal{C}_3$.*

Proof. The lemma holds trivially if q contains no constant. Assume from here on that q contains at least one constant.

Assume that q satisfies $\mathcal{D}_1$. Then $\mathsf{char}(q)$ must be self-join-free. In this case, $\mathsf{ext}(q)$ is self-join-free, and thus $\mathsf{ext}(q)$ satisfies $\mathcal{C}_1$.

For the two remaining items, assume that q satisfies $\mathcal{D}_2$ or $\mathcal{D}_3$. Since $\mathcal{D}_2$ logically implies $\mathcal{D}_3$, q satisfies $\mathcal{D}_3$. By Lemma 4.27, $\mathsf{ext}(q)$ satisfies $\mathcal{C}_2$. Since $\mathcal{C}_2$ logically implies $\mathcal{C}_3$, q satisfies $\mathcal{C}_3$. □

We can now prove the upper bounds in Theorem 4.3.

Proof of upper bounds in Theorem 4.3. Since first-order reductions compose, there is a first-order reduction from the problem $\mathsf{CERTAINTY}(q)$ to $\mathsf{CERTAINTY}(\mathsf{ext}(q))$ by Lemmas 4.25 and 4.26. The upper bound results then follow by Lemma 4.28. □

Finally, the proof of Theorem 4.3 reveals that for generalized path queries q containing at least one constant, the complexity of $\mathsf{CERTAINTY}(q)$ exhibits a trichotomy (instead of a tetrachotomy as in Theorem 4.3).

Theorem 4.4. *For any generalized path query q containing at least one constant, the problem $\mathsf{CERTAINTY}(q)$ is either in* **FO**, **NL***-complete, or* **coNP***-complete.*

Proof. Immediate from Theorem 4.3 and Lemma 4.27. □

4.6.2 Lower bounds in Theorem 4.3

The complexity lower bounds in Theorem 4.3 can be proved by slight modifications of the proofs in Sections 4.5.1 and 4.5.2. We explain these modifications below for a generalized path query q containing at least one constant. Note incidentally that the proof in Section 4.5.3 needs no revisiting, because, by Lemma 4.27, a violation of $\mathcal{D}_2$ implies a violation of $\mathcal{D}_3$.

In the proof of Lemma 4.18, let $\mathsf{char}(q) = [\![uRvRw, c]\!]$ where c is a constant and there is no prefix homomorphism from $\mathsf{char}(q)$ to $[\![uRvRvRw, c]\!]$. Let $p = q \setminus \mathsf{char}(q)$. Note that the path query uRv does not contain any constant. We revise the reduction description in Lemma 4.18 to be

- for each vertex $x \in V \cup \{s'\}$, we add $\phi_{\perp}^{x}[u]$;
- for each edge $(x, y) \in E \cup \{(s', s), (t, t')\}$, we add $\phi_{x}^{y}[Rv]$;
- for each vertex $x \in V$, we add $\phi_{x}^{c}[Rw]$; and
- add a canonical copy of p (which starts in the constant c).

An example is shown in Figure 4.11. Since the constant c occurs at most twice in q by Definition 4.16, the query q can only be satisfied by a repair including each of $\phi_{\perp}^{x}[u]$, $\phi_{x}^{y}[Rv]$, $\phi_{y}^{c}[Rw]$, and the canonical copy of p. **NL**-hardness can now be proved as in the proof of Lemma 4.18.

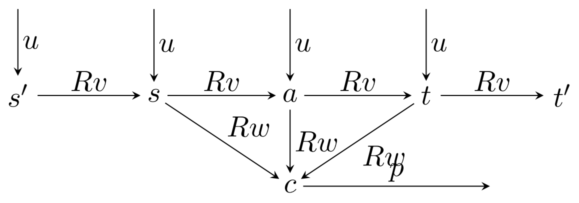

$$\begin{tikzcd}
{} \arrow[d, "u"] & {} \arrow[d, "u"] & {} \arrow[d, "u"] & {} \arrow[d, "u"] & \\
s' \arrow[r, "Rv"] & s \arrow[r, "Rv"] \arrow[dr, "Rw"'] & a \arrow[r, "Rv"] \arrow[d, "Rw"] & t \arrow[r, "Rv"] \arrow[dl, "Rw"] & t' \\
& & c \arrow[r, "p"] & {} &
\end{tikzcd}$$

Figure 4.11: Database instance for the revised **NL**-hardness reduction from the graph G with $V = \{s, a, t\}$ and $E = \{(s, a), (a, t)\}$.

In the proof of Lemma 4.19, let $\mathsf{char}(q) = [\![uRvRw, c]\!]$ where c is a constant and there is no homomorphism from $\mathsf{char}(q)$ to $[\![uRvRvRw, c]\!]$. Let $p = q \setminus \mathsf{char}(q)$. Note that both path queries uRv and u do not contain any constant. We revise the reduction description in Lemma 4.19 to be

- for each variable z, we add $\phi_{z}^{c}[Rw]$ and $\phi_{z}^{c}[RvRw]$;
- for each clause C and positive literal z of C, we add $\phi_{C}^{z}[u]$;
- for each clause C and variable z that occurs in a negative literal of C, we add $\phi_{C}^{z}[uRv]$; and
- add a canonical copy of p (which starts in the constant c).

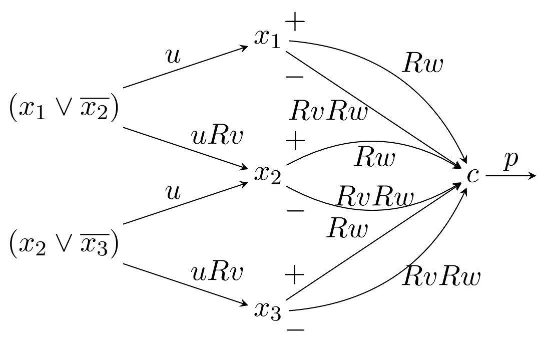


Figure 4.12: Database instance for the revised **coNP**-hardness reduction from the formula $\psi = (x_1 \vee \overline{x_2}) \wedge (x_2 \vee \overline{x_3})$.

An example is shown in Figure 4.12. Since the constant c occurs at most twice in q, the query q can only be satisfied by a repair $\mathbf{r}$ such that either

- $\mathbf{r}$ contains $\phi_C^z[uRv]$, $\phi_z^c[Rw]$, and the canonical copy of p; or
- $\mathbf{r}$ contains $\phi_C^z[u]$, $\phi_z^c[RvRw]$, and the canonical copy of p.

coNP-hardness can now be proved as in the proof of Lemma 4.19.

4.7 Conclusion

We established a complexity classification in consistent query answering relative to primary keys, for path queries that can have self-joins: for every path query q, the problem $\mathsf{CERTAINTY}(q)$ is in **FO**, **NL**-complete, **PTIME**-complete, or **coNP**-complete, and it is decidable in polynomial time in the size of q which of the four cases applies. If $\mathsf{CERTAINTY}(q)$ is in **FO** or in **PTIME**, rewritings of q can be effectively constructed in, respectively, first-order logic and Least Fixpoint Logic . Notably, we extend the existing classification of $\mathsf{CERTAINTY}(q)$ beyond self-join-free BCQs.

Chapter 5

A Trichotomy for CQA on Rooted Tree Queries and Beyond

眾里尋他千百度
驀然回首
那人卻在燈火闌珊處
—辛棄疾 《青玉案》

In this chapter, we further our existing tetrachotomy classification for CQA on path queries to a more general class of query, called *rooted tree queries.* We then show that the classification on rooted tree queries can be extended to other classes of queries with self-joins.

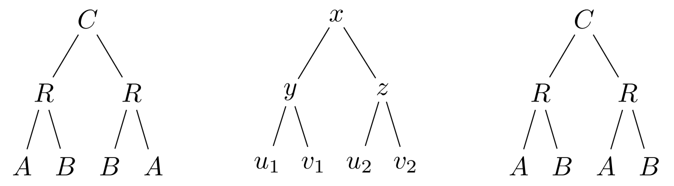
Past research has indicated that the tools used for proving Theorem 1.1 largely fall short in dealing with difficulties caused by self-joins. A notable example concerns *path queries*, which we discussed in Chapter 4, of the form

$$\exists x_1 \cdots \exists x_{k+1}(R_1(\underline{x_1}, x_2) \wedge R_2(\underline{x_2}, x_3) \wedge \cdots \wedge R_k(\underline{x_k}, x_{k+1})).$$

If a query of this form is self-join-free (i.e., if $R_i \neq R_j$ whenever $i \neq j$), then the "attack graph" tool [KW21] immediately tells us that $\mathsf{CERTAINTY}(q)$ is in **FO**. However, for path queries q with self-joins, $\mathsf{CERTAINTY}(q)$ exhibits a tetrachotomy between **FO**, **NL**-complete, **PTIME**-complete, and **coNP**-complete [KOW21], and the complexity classification requires sophisticated tools that concern "word rewinding".

A natural question addressed in this chapter is to extend the complexity classification for path queries to queries that are syntactically less constrained. In particular, while path queries are restricted to binary relation names, we aim for unrestricted arities, as in practical database systems, which brings us to the construct of tree queries.

A query q in BCQ is a *rooted (ordered) tree query* if it is uniquely (up to a variable renaming) representable by a rooted ordered tree in which each non-leaf vertex is labeled by a relation name,



(a) A rooted ordered tree representing q_1.

(b) Each vertex is associated with a fresh variable.

(c) A rooted ordered tree representing q_2.

Figure 5.1: The left rooted ordered tree represents (up to a variable renaming) the Boolean conjunctive query q_1 with atoms $C(\underline{x}, y, z)$, $R(\underline{y}, u_1, v_1)$, $A(\underline{u_1})$, $B(\underline{v_1})$, $R(\underline{z}, u_2, v_2)$, $B(\underline{u_2})$, $A(\underline{v_2})$. The right rooted ordered tree represents q_2 with atoms $C(\underline{x}, y, z)$, $R(\underline{y}, u_1, v_1)$, $A(\underline{u_1})$, $B(\underline{v_1})$, $R(\underline{z}, u_2, v_2)$, $A(\underline{u_2})$, $B(\underline{v_2})$.

and each leaf vertex is labeled by a unary relation name, a constant, or $\perp$. The query q is read from this tree as follows: each vertex labeled by either a relation name or $\perp$ is first associated with a fresh variable, and each vertex labeled by a constant is associated with that same constant; then, a vertex labeled with relation name R and associated with variable x represents the query atom $R(\underline{x}, y_1, \dots, y_n)$, where $y_1, \dots, y_n$ are the symbols (variables or constants) associated with the left-to-right ordered children of the vertex x. The underlined position is the primary key. Note that a vertex labeled with a relation name of arity $n+1$ must have n children. For example, consider the rooted tree in Figure 5.1(a) and associate fresh variables to its vertices as depicted in Figure 5.1(b). The rooted tree thus represents a query q_1 that contains, among others, the atoms $C(\underline{x}, y, z)$ and $R(\underline{y}, u_1, v_1)$. It is easy to see that every path query is a rooted tree query. The class of all rooted tree queries is denoted $\mathsf{TreeBCQ}$. We can now present our main results, previously stated in Theorem 1.4.

Theorem 5.1. *For every query q in* $\mathsf{TreeBCQ}$*,* $\mathsf{CERTAINTY}(q)$ *is in* **FO***,* **NL***-hard* $\cap$ **LFPL***, or* **coNP***-complete, and it is decidable in polynomial time in the size of q which of the three cases applies.*

Here **LFPL** denotes the class of problems expressible in *Least Fixed-point Logic*. The classification criteria implied in Theorem 1.4 and 5.1 are explicitly stated in Theorem 5.4.

It will turn out that subtree homomorphisms play a crucial role in the complexity classification of $\mathsf{CERTAINTY}(q)$ for queries q in $\mathsf{TreeBCQ}$. For example, our results show that for the queries q_1 and q_2 represented in, respectively, Figure 5.1(a) and (c), $\mathsf{CERTAINTY}(q_1)$ is **coNP**-complete, while $\mathsf{CERTAINTY}(q_2)$ is in **FO**. The difference occurs because the two ordered subtrees rooted at R are isomorphic in q_2 (A precedes B in both subtrees), but not in q_1. Another novel and useful tool in the complexity classification is a tree automaton that generalizes the NFA for path queries used in [KOW21].

Once Theorem 1.4 is proved, it is quite natural to generalize rooted tree queries further by allowing queries that can be represented by graphs that are not trees. Toward this generalization, we next define a new subclass of BCQ.

Definition 5.1 (GraphBCQ)**.** GraphBCQ is the class of Boolean conjunctive queries q satisfying the following conditions:

1. every atom in q is of the form $R(\underline{x}, y_1, \dots, y_n)$ where x is a variable and $y_1, \dots, y_n$ are symbols (variables or constants) such that no variable occurs twice in the atom; and
2. if $R(\underline{x}, y_1, \dots, y_n)$ and $S(\underline{u}, v_1, \dots, v_m)$ are distinct atoms of q, then $x \neq u$. Note that R and S need not be distinct.

Every rooted tree query belongs to GraphBCQ. Furthermore, the graph representation of rooted tree queries naturally extends to all queries q in GraphBCQ: there is a vertex for every variable or constant occurring in q; and for every atom $R(\underline{x}, y_1, \dots, y_n)$ of q, the vertex x is labeled by R and has left-to-right ordered outgoing edges to vertices $y_1, \dots, y_n$. Every sink vertex s that is not labeled by a relation name is labeled by "$\perp$" if s is a variable, and by "s" if s is a constant.

Significantly, we were able to establish the **FO**-boundary in the set $\{\mathsf{CERTAINTY}(q) \mid q \in \mathsf{GraphBCQ}\}$.

Theorem 5.2. *For every query q in* GraphBCQ*, it is decidable whether or not* $\mathsf{CERTAINTY}(q)$ *is in* **FO***; and when it is, a first-order rewriting can be effectively constructed.*

So far, we have not achieved a fine-grained complexity classification of all problems in the set $\{\mathsf{CERTAINTY}(q) \mid q \in \mathsf{GraphBCQ}\}$. However, we were able to do so for the set of Berge-acyclic queries in GraphBCQ, denoted $\mathsf{Graph}_{\mathsf{Berge}}\mathsf{BCQ}$. Recall that a conjunctive query is Berge-acyclic if its incidence graph (i.e., the undirected bipartite graph that connects every variable x to all query atoms in which x occurs) is acyclic.

Theorem 5.3. *For every query q in* $\mathsf{Graph}_{\mathsf{Berge}}\mathsf{BCQ}$*, the decision problem* $\mathsf{CERTAINTY}(q)$ *is in* **FO***,* **NL***-hard* $\cap$ **LFPL***, or* **coNP***-complete, and it is decidable in polynomial time in the size of q which of the three cases applies.*

Since

$$\mathsf{TreeBCQ} \subsetneq \mathsf{Graph}_{\mathsf{Berge}}\mathsf{BCQ} \subsetneq \mathsf{GraphBCQ} \tag{5.1}$$

is easily verified, Theorem 1.4 is subsumed by Theorem 5.3. We nevertheless provide Theorem 1.4 explicitly, because its proof makes up the main part of this chapter.

5.1 Preliminaries

A *Boolean conjunctive query* can also be represented by a finite set $q = \{R_1(\underline{\vec{x}_1}, \vec{y}_1), \ldots, R_n(\underline{\vec{x}_n}, \vec{y}_n)\}$ of atoms. The set q represents the first-order sentence with no free-variables

$$\exists u_1 \cdots \exists u_k (R_1(\underline{\vec{x}_1}, \vec{y}_1) \wedge \cdots \wedge R_n(\underline{\vec{x}_n}, \vec{y}_n)),$$

and we denote $\mathsf{vars}(q) = \{u_1, \ldots, u_k\}$, the set of variables that occur in q. We write BCQ for the class of Boolean conjunctive queries.

Rooted relation trees. A *rooted relation tree* is a (directed) rooted ordered tree where each internal vertex is labeled by a relation name, and each leaf vertex is labeled with either a unary relation name, a constant, or $\perp$, such that every two vertices sharing the same label have the same number of children. We denote by $\tau^u_\triangle$ the subtree rooted at vertex u in τ. Any rooted relation tree τ has a succinct representation recursively defined as follows:

- if τ contains a single vertex labeled ℓ, then $\tau = \ell$;
- if the root of τ is labeled R and has the following ordered children $v_1, v_2, \ldots, v_n$, then $\tau = R(\tau_1, \tau_2, \ldots, \tau_n)$, where τ_i is the succinct representation of $\tau^{v_i}_\triangle$.

Rooted tree query and rooted tree sets. A *querification* of a rooted relation tree τ is a total function f with domain τ's vertex set that maps each vertex labeled by a constant to that same constant, and injectively maps all other vertices to variables. Such a querification naturally extends to a mapping $f(\tau)$ of the entire tree: if u is a vertex in τ with label R and children v_1, v_2, $\ldots$, v_n, then $f(\tau)$ contains the atom $R(\underline{f(u)}, f(v_1), f(v_2), \ldots, f(v_n))$. A Boolean conjunctive query is a *rooted tree query* if it is equal to $f(\tau)$ for some querification f of some rooted relation tree τ. If $q = f(\tau)$, we also say that q is *represented* by τ. It can be verified that every rooted tree query belongs to $\mathsf{GraphBCQ}$, as stated by (5.1). We write $R[x]$ for the unique R-atom in q with primary key variable x. We write $\mathsf{TreeBCQ}$ for the class of rooted tree queries.

Every query q in $\mathsf{TreeBCQ}$ is represented by a unique rooted relation tree. Conversely, every rooted relation tree represents a query in $\mathsf{TreeBCQ}$ that is unique up to a variable renaming. When $f(\tau) = q$, by a slight abuse of terminology, we may use q to refer to τ, and use the query variable x

(or the expression $R[x]$) to refer to the vertex u in τ that satisfies $f(u) = x$ and whose label is R. The variable r is the *root variable* of a query q in $\mathsf{TreeBCQ}$ if r is the root vertex of q's rooted relation tree. For two distinct vertices x and y, we write $x <_q y$ if the vertex x is an ancestor of y in q, and write $x \parallel_q y$ if neither $x <_q y$ nor $y <_q x$. When x and y have the same label R, we can also write $R[x] <_q R[y]$ and $R[x] \parallel_q R[y]$ instead of $x <_q y$ and $x \parallel_q y$ respectively. For every variable x in a rooted tree query q, we write $q^x_\triangle$ for the subquery of q whose rooted relation tree is the subtree rooted at vertex x in q. A variable x is a leaf variable in q if $q^x_\triangle = \perp$, $q^x_\triangle = c$, or $q^x_\triangle = A$, for some constant c or unary relation name A.

An *instantiation* of a rooted relation tree τ is a total function g from τ's vertex set to constants such that each vertex labeled by a constant c is mapped to c. As before, we define $g(\tau)$ as the $\subseteq$-minimal set that contains $R(\underline{g(u)}, g(v_1), g(v_2), \ldots, g(v_n))$ whenever u is a vertex in τ with label R and children v_1, v_2, $\ldots$, v_n. A subset S of $\mathbf{db}$ is a *rooted tree set in* $\mathbf{db}$ *starting in* c if $S = g(\tau)$ for some instantiation g of τ that maps τ's root to c. A case of particular interest is when $\mathbf{db}$ is consistent, in particular, when $\mathbf{db}$ is a repair. It can be verified that a rooted tree set in a repair $\mathbf{r}$ is uniquely determined by a constant c and a rooted tree τ (because only one instantiation is possible); by overloading terminology, τ is also called a rooted tree set in $\mathbf{r}$ starting in c. For convenience, an empty rooted tree set, denoted by $\perp$, starts in any constant c.

Example 5.1. Figure 5.2 depicts a rooted relation tree that corresponds to the following rooted tree query

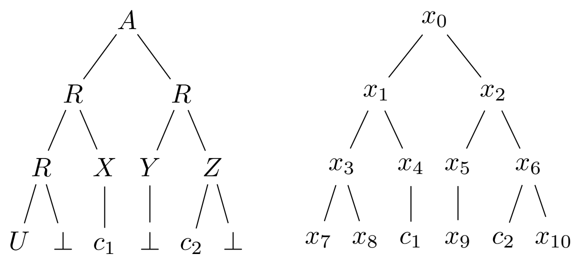
$$\begin{aligned} q = \{&A(\underline{x_0}, x_1, x_2), R(\underline{x_1}, x_3, x_4), R(\underline{x_2}, x_5, x_6),\\ &R(\underline{x_3}, x_7, x_8), U(\underline{x_7}),\\ &X(\underline{x_4}, c_1), Y(\underline{x_5}, x_9), Z(\underline{x_6}, c_2, x_{10})\} \end{aligned}$$

for constants c_1 and c_2, which has a succinct representation

$$q = A(R(R(U, \perp), X(c_1)), R(Y(\perp), Z(c_2, \perp))),$$

where

$$\begin{aligned} q^{x_1}_\triangle &= R(\underline{x_1}, x_3, x_4), R(\underline{x_3}, x_7, x_8), U(\underline{x_7}), X(\underline{x_4}, c_1)\\ &= R(R(U, \perp), X(c_1)),\\ q^{x_2}_\triangle &= R(\underline{x_2}, x_5, x_6), Y(\underline{x_5}, x_9), Z(\underline{x_6}, c_2, x_{10})\\ &= R(Y(\perp), Z(c_2, \perp)),\\ q^{x_3}_\triangle &= R(\underline{x_3}, x_7, x_8), U(\underline{x_7})\\ &= R(U, \perp). \end{aligned}$$


A
R R
R X Y Z
U $\perp$ c_1 $\perp$ c_2 $\perp$
x_0
x_1 x_2
x_3 x_4 x_5 x_6
x_7 x_8 c_1 x_9 c_2 x_{10}

(a) A rooted relation tree τ

(b) A mapping f from vertices in τ to variables in q

Figure 5.2: An example rooted relation tree, where c_1 and c_2 are constants.

In this query q, we have $R[x_1] \parallel_q R[x_2]$, $R[x_1] <_q R[x_3]$ and $R[x_2] \parallel_q R[x_3]$.

It can now be verified that for rooted tree queries p and q, there is a homomorphism h from p to q if and only if there is a label-preserving homomorphism from the rooted relation tree of p to that of q (we assume that a leaf vertex with label $\perp$ can map to a vertex with any label). In particular, the order of child vertices must be preserved under such homomorphism: for example, there is no homomorphism between the rooted trees in Figure 5.1(a) and (c).

5.2 The Complexity Classification

Our classification focuses on rooted tree queries ($\mathsf{TreeBCQ}$). We will extend to $\mathsf{Graph}_{\mathsf{Berge}}\mathsf{BCQ}$ and $\mathsf{GraphBCQ}$ in Section 5.7. The classification of path queries in [KOW21] concerns a notion of "rewinding": if a path query q, when treated as a word, can be written as $q = u \cdot Rv \cdot Rw$, then q rewinds to $u \cdot Rv \cdot Rv \cdot Rw$. We generalize the notion of rewinding from path queries to rooted tree queries.

Definition 5.2 (Rewinding)**.** Let q be a query in $\mathsf{TreeBCQ}$. Let $R(\underline{x}, \ldots)$ and $R(\underline{y}, \ldots)$ be two (not necessarily distinct) atoms in q. We define $q^{R:y \looparrowright x}$ as the following rooted tree query

$$q^{R:y \looparrowright x} := (q \setminus q^y_\triangle) \cup f(q^x_\triangle),$$

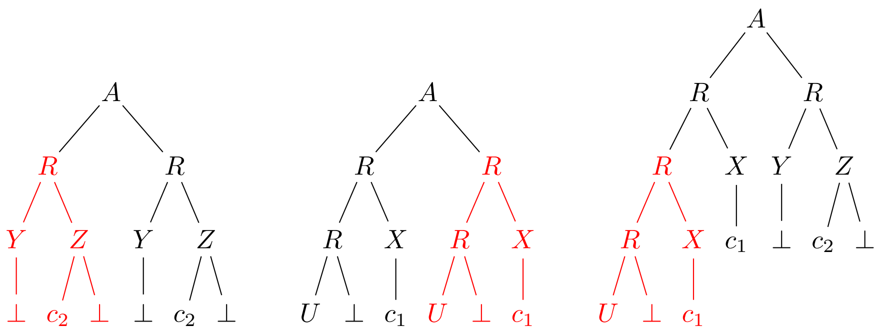
for some isomorphism f that maps x to y (i.e., $f(x) = y$), and maps every other variable in $q^x_\triangle$ to a fresh variable.

Intuitively, the rooted tree query $q^{R:y \looparrowright x}$ can be obtained by replacing $q^y_\triangle$ with a fresh copy of $q^x_\triangle$. Figure 5.3 presents some rooted tree queries obtained from rewinding on the rooted tree q in Figure 5.2.

The classification criteria in [KOW21] uses the notions of factors and prefixes that are specific to words, which can be generalized using homomorphism on rooted tree queries. Consider the following syntactic conditions on a rooted tree query q with root variable r:

- $\mathcal{C}_2$: for every two atoms $R(\underline{x}, \ldots)$ and $R(\underline{y}, \ldots)$ in q, either $q \leq_{\rightarrow} q^{R:y \looparrowright x}$ or $q \leq_{\rightarrow} q^{R:x \looparrowright y}$.
- $\mathcal{C}_1$: for every two atoms $R(\underline{x}, \ldots)$ and $R(\underline{y}, \ldots)$ in q, either $q \leq_{r \rightarrow r} q^{R:y \looparrowright x}$ or $q \leq_{r \rightarrow r} q^{R:x \looparrowright y}$.

It is easy to see that conditions $\mathcal{C}_1$ and $\mathcal{C}_2$ are decidable in polynomial time in the size of the query. The conditions $\mathcal{C}_1$ and $\mathcal{C}_2$ introduced in this chapter can be viewed as generalizations of conditions $\mathcal{C}_1$ and $\mathcal{C}_3$ in Chapter 4 respectively. We may restate $\mathcal{C}_2$ and $\mathcal{C}_1$ using more fine-grained syntactic conditions below.



(a) $q^{R:x_1 \looparrowright x_2}$ (b) $q^{R:x_2 \looparrowright x_1}$ (c) $q^{R:x_3 \looparrowright x_1}$

Figure 5.3: An illustration of rewinding for the query of Figure 5.2; the modified subtrees are highlighted in red.

- $\mathcal{C}_{\mathsf{branch}}$: for every two atoms $R[x] \parallel_q R[y]$ in q, either $q^y_\triangle \leq_{y \to x} q^x_\triangle$ or $q^x_\triangle \leq_{x \to y} q^y_\triangle$.
- $\mathcal{C}_{\mathsf{factor}}$: for every two atoms $R[x] <_q R[y]$ in q, we have $q \leq_{\to} q^{R:y \looparrowright x}$.
- $\mathcal{C}_{\mathsf{prefix}}$: for every two atoms $R[x] <_q R[y]$ in q, we have $q \leq_{r \to r} q^{R:y \looparrowright x}$.

Lemma 5.1. *For every two atoms $R[x] \parallel_q R[y]$ in a rooted tree query q, we have $q \leq_{\to} q^{R:y \looparrowright x}$ if and only if $q^y_\triangle \leq_{y \to x} q^x_\triangle$.*

Proof. We denote

$$p = q^{R:y \looparrowright x} = (q \setminus q^y_\triangle) \cup f(q^x_\triangle),$$

for some isomorphism f that maps every variable in $q^x_\triangle$ to a fresh variable, except for x, which we have $f(x) = y$.

Assume first that $q^y_\triangle \leq_{y \to x} q^x_\triangle$, witnessed by the homomorphism h with $h(y) = x$. It is easy to verify that the homomorphism $g : \mathsf{vars}(q) \to \mathsf{vars}(p)$ with

$$g(z) = \begin{cases} z & \text{if } z \in \mathsf{vars}(q \setminus q^y_\triangle), \\ f(h(z)) & \text{otherwise} \end{cases}$$

is a homomorphism from q to p.

Assume there is a homomorphism $h : \mathsf{vars}(q) \to \mathsf{vars}(p)$ from q to p. Hence

$$h(q) = h(q \setminus q^y_\triangle) \cup h(q^y_\triangle) \subseteq (q \setminus q^y_\triangle) \cup f(q^x_\triangle).$$

Note that q is minimal, i.e., there is no automorphism α such that $\alpha(q) \subsetneq q$. If $h(y) = y$, since q is minimal, we have $h(q \setminus q^y_\triangle) = q \setminus q^y_\triangle$, and we have $h(q^y_\triangle) \subseteq f(q^x_\triangle)$. Thus $q^y_\triangle \leq_{y \to x} q^x_\triangle$, witnessed by the homomorphism $g = f^{-1} \circ h$ with $g(y) = f^{-1}(h(y)) = f^{-1}(y) = x$, as desired.

Suppose for contradiction that $h(y) \neq y$. In this case, we have $h(q \setminus q^y_\triangle) \cap f(q^x_\triangle) \neq \emptyset$.

We argue that $y = f(x) <_p h(y)$. Case (i) Assume $h(y) <_p y = f(x)$ holds. Then h maps the unique path of nodes from r to y in q to the unique path from $h(r)$ to $h(y)$ in p. While we have $r = h(r)$ or $r <_p h(r)$, but since $h(y) <_p y$, this is not possible because the path from $h(r)$ to $h(y)$ in p is strictly shorter than the path from r to y in q. Case (ii) Assume $h(y) \parallel_p y = f(x)$ holds. Let $y_0 = y$, and for each $i \geq 1$, $y_i = h(y_{i-1})$. We argue that variables y_0, y_1, $\ldots$ are all distinct, thereby reaching a contradiction to the finite size of q. Assume first that y_1 is a left sibling of y_0 in q: for the greatest common ancestor y^* of y_1 and y_0, there is an atom $R(\underline{y^*}, .., y_l, .., y_r)$ such that y_l and y_r are ancestors of y_1 and y_0. The arguments for the case where y_1 is a right sibling of y_0 in q is similar. Note that y_1 appears in both p and q and its subtree is not affected by the rewinding operation since $y_1 \parallel_p y_0$. Since y_1 is a left sibling of y_0 and that the children of rooted trees are ordered, $h(y_1)$ is a left sibling of $h(y_0)$, that is y_2 is a left sibling of y_1 in q, and this process continues. Since each y_{i+1} is a left sibling of y_i, the variables need to be distinct, or otherwise there is some y_{j+1} is a right sibling of y_j, a contradiction.

Let $T[z]$ be the greatest common ancestor of $R[x]$ and $R[y]$ in q and let u and v be variables in $T[z]$ such that $u <_q x$ and $v <_q y$ and $u \parallel_q v$. Hence z appears in both q and p. Hence $z <_p h(z)$ but $h(z) \neq z$. We have

$$|q^u_\triangle| + |q^v_\triangle| + 1 \leq |q^z_\triangle| \leq |p^{h(z)}_\triangle|,$$

because the homomorphism maps $q^z_\triangle$ to the subtree of p, rooted at $h(z)$.

We show that $v <_q h(z)$. Since $z <_q h(z)$ and v is the immediate child of z, we can have either $v <_q h(z)$ or $v \parallel_q h(z)$. Suppose for contradiction that $v \parallel_q h(z)$, then $h(z) \notin \{u, v\}$. Then, $p^{h(z)}_\triangle = q^{h(z)}_\triangle$ since the rewinding leaves $q^{h(z)}_\triangle$ intact. But that implies $h(q^z_\triangle) \subseteq q^{h(z)}_\triangle$ with $z <_q h(z)$, a contradiction.

Therefore, we have

$$|q^u_\triangle| + |q^v_\triangle| + 1 \leq |p^{h(z)}_\triangle| \leq |p^v_\triangle| \leq |q^v_\triangle| - |q^y_\triangle| + |q^x_\triangle|,$$

where the second inequality follows by construction of rewinding that replaces $q^y_\triangle$ with $q^x_\triangle$.

This yields

$$0 \leq |q^u_\triangle| - |q^x_\triangle| \leq -|q^y_\triangle| - 1 < 0,$$

a contradiction. □

Proposition 5.1. $\mathcal{C}_2 = \mathcal{C}_{\mathsf{factor}} \wedge \mathcal{C}_{\mathsf{branch}}$, $\mathcal{C}_1 = \mathcal{C}_{\mathsf{prefix}} \wedge \mathcal{C}_{\mathsf{branch}}$.

Proof. Immediate from Lemma 5.1. □

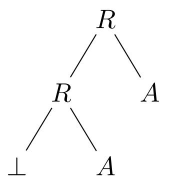
Example 5.2. Let q be as in Figure 5.2. We have that q violates $\mathcal{C}_{\mathsf{branch}}$ (and therefore $\mathcal{C}_2$), since there is no homomorphism from q to neither $q^{R:x_1 \hookrightarrow x_2}$ nor $q^{R:x_2 \hookrightarrow x_1}$.

Figure 5.4 shows some example rooted relation trees annotated with the syntactic conditions they satisfy or violate.

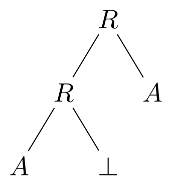
Our main classification result can now be stated.

Theorem 5.4 (Trichotomy Theorem)**.** *For every query q in* TreeBCQ*,*

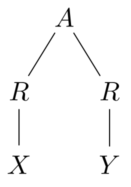
- *if q satisfies $\mathcal{C}_2$, then the problem* CERTAINTY(q) *is in* **LFPL***; otherwise it is* **coNP***-complete; and*
- *if q satisfies $\mathcal{C}_1$, then the problem* CERTAINTY(q) *is in* **FO***; otherwise it is* **NL***-hard.*



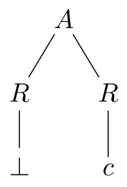
(a) $\mathcal{C}_1$



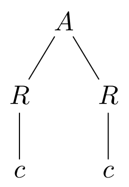
(b) $\mathcal{C}_2, \neg\mathcal{C}_{\mathsf{prefix}}$



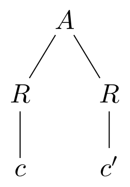
(c) $\neg\mathcal{C}_{\mathsf{branch}}$



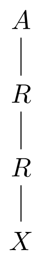
(d) $\mathcal{C}_1$



(e) $\mathcal{C}_1$



(f) $\neg\mathcal{C}_{\mathsf{branch}}$



(g) $\neg\mathcal{C}_{\mathsf{factor}}$

Figure 5.4: Examples of rooted relation trees. Trees annotated with $\neg\mathsf{C}$ violate syntactic condition C, while trees annotated with C satisfy C. For example, the tree in (a) satisfies $\mathcal{C}_1$; and the tree (b) satisfies $\mathcal{C}_2$ but violates $\mathcal{C}_{\mathsf{prefix}}$.

	C	$\underline{1}$	2	3
$*$		c_1	x_1	z_-
		c_1	x_2	z_-
		c_2	z_+	x_1
$*$		c_2	z_+	x_2

	R	$\underline{1}$	2	3
		x_1	a	b
$*$		x_1	b	a
$*$		x_2	a	b
		x_2	b	a
$*$		z_+	a	b
$*$		z_-	b	a

	A	$\underline{1}$
$*$		a

	B	$\underline{1}$
$*$		b

Figure 5.5: An inconsistent database instance **db** for $\mathsf{CERTAINTY}(q_1)$, where q_1 is represented in Figure 5.1(a). Blocks are separated by dashed lines. The facts with $*$ form a repair that falsifies q_1, corresponding to a satisfying truth assignment $x_1 = 1$ and $x_2 = 0$.

Let us provide some intuitions behind Theorem 5.4. Both $\mathcal{C}_{\mathsf{prefix}}$ and $\mathcal{C}_{\mathsf{factor}}$ concern the homomorphism from q to the rooted tree query obtained by rewinding from a subtree to its ancestor subtree, which resembles the case on path queries. The condition $\mathcal{C}_{\mathsf{branch}}$ is vacuously satisfied for path queries, but is crucial to the classification of rooted tree queries.

For the complexity lower bound, if q violates $\mathcal{C}_{\mathsf{branch}}$, then $\mathsf{CERTAINTY}(q)$ is **coNP**-hard. Intuitively, this is because if $q^x_\triangle$ and $q^y_\triangle$ are not homomorphically comparable and appear in different branches, then the facts in their common ancestor relation may "choose" which branch to satisfy, which allows us to reduce from SAT in Lemma 5.14. For example, consider the query q_1 as in Figure 5.1(a) and the example database instance **db** in Figure 5.5. It can be shown that there is a repair of **db** that falsifies q_1 if and only if the following CNF formula is satisfiable:

$$\underbrace{(x_1 \lor x_2)}_{C_1} \land \underbrace{(\overline{x_1} \lor \overline{x_2})}_{C_2}.$$

For the complexity upper bound, if $q^y_\triangle \leq_{y \to x} q^x_\triangle$, the arguments above fail because the facts in their common ancestor relation cannot "choose" which branch to satisfy anymore: informally, whenever $q^x_\triangle$ is satisfied, $q^y_\triangle$ will be satisfied due to the homomorphism. This crucial observation from $\mathcal{C}_{\mathsf{branch}}$ also leads to a total preorder on all self-joining atoms, which allows us to deal with self-joining atoms in different branches as if they were on a path.

Definition 5.3 (Relation $\preceq_q$)**.** Let q be a query in $\mathsf{TreeBCQ}$. Let $R[x]$ and $R[y]$ be two atoms in q. We write $R[x] \preceq_q R[y]$ if either $R[x] <_q R[y]$ or $q^y_\triangle \leq_{y \to x} q^x_\triangle$.

Proposition 5.2. *Let q be a query in* $\mathsf{TreeBCQ}$ *satisfying* $\mathcal{C}_{\mathsf{branch}}$. *For every relation name R, the relation $\preceq_q$ is a total preorder on all R-atoms in q.*

Proof. We first show that every two distinct atoms $R[x]$ and $R[x]$ are comparable by $\preceq_q$. Let $R[x]$ and $R[y]$ be two distinct atoms in q. The claim holds if $R[x] <_q R[y]$ or $R[y] <_q R[x]$. Otherwise, we have $R[x] \parallel_q R[y]$, and since q satisfies $\mathcal{C}_{\mathsf{branch}}$, we have either $q_\triangle^x \leq_{x \to y} q_\triangle^y$ or $q_\triangle^y \leq_{y \to x} q_\triangle^x$, as desired.

Next we show show that $\preceq_q$ is transitive. Assume that $R[x] \preceq_q R[y]$ and $R[y] \preceq_q R[z]$. We distinguish four cases.

- Case that $R[x] <_q R[y]$ and $R[y] <_q R[z]$. Then we have $R[x] <_q R[z]$, as desired.
- Case that $q_\triangle^y \leq_{y \to x} q_\triangle^x$ and $q_\triangle^z \leq_{z \to y} q_\triangle^y$. Then we have $q_\triangle^z \leq_{z \to x} q_\triangle^x$, as desired.
- Case that $R[x] <_q R[y]$ and $q_\triangle^z \leq_{z \to y} q_\triangle^y$. The claim follows if $R[x] <_q R[z]$. Suppose for contradiction that $R[z] <_q R[x]$. Then $R[z] <_q R[y]$, and $q_\triangle^z$ contains more atoms than $q_\triangle^y$. However, we have $q_\triangle^z \leq_{z \to y} q_\triangle^y$, a contradiction. It then must be that $R[x] \parallel_q R[z]$. Suppose for contradiction that $q_\triangle^x \leq_{x \to z} q_\triangle^z$. Then we have $q_\triangle^x \leq_{x \to y} q_\triangle^y$, but $R[x] <_q R[y]$, a contradiction. Since q satisfies $\mathcal{C}_{\mathsf{branch}}$, we have $q_\triangle^z \leq_{z \to x} q_\triangle^x$, as desired.
- Case that $q_\triangle^y \leq_{y \to x} q_\triangle^x$ and $R[y] <_q R[z]$. The claim follows if $R[x] <_q R[z]$. Suppose for contradiction that $R[z] <_q R[x]$. Then $R[y] <_q R[x]$, and $q_\triangle^y$ contains more atoms than $q_\triangle^x$. However, we have $q_\triangle^y \leq_{y \to x} q_\triangle^x$, a contradiction. It then must be that $R[x] \parallel_q R[z]$. Suppose for contradiction that $q_\triangle^x \leq_{x \to z} q_\triangle^z$. Then we have $q_\triangle^y \leq_{y \to z} q_\triangle^z$, but $R[y] <_q R[z]$, a contradiction. Since q satisfies $\mathcal{C}_{\mathsf{branch}}$, it follows $q_\triangle^z \leq_{z \to x} q_\triangle^x$.

This concludes the proof. □

The remainder of this chapter is organized as follows.

- In Section 5.3, we define a tree automaton $\mathsf{NFA}^{\clubsuit}(q)$ for each $q \in \mathsf{TreeBCQ}$, and the problem $\mathsf{CERTAIN}_{\mathsf{tr}}(q)$ that concerns $\mathsf{NFA}^{\clubsuit}(q)$. Lemma 5.2 concludes the equivalence of $\mathsf{CERTAINTY}(q)$ and $\mathsf{CERTAIN}_{\mathsf{tr}}(q)$ if q satisfies $\mathcal{C}_2$ (or $\mathcal{C}_1$).
- In Section 5.4, we show that $\mathsf{CERTAIN}_{\mathsf{tr}}(q)$ is in **LFPL** (and in **PTIME**) if q satisfies $\mathcal{C}_{\mathsf{branch}}$.
- In Sections 5.5 and 5.6, we show the upper bounds and lower bounds in Theorem 5.4 respectively.
- In Section 5.7, we prove Theorems 5.2 and 5.3.

5.3 Context-Free Grammar

We first generalize NFAs used in the study of path queries [KOW21] to context-free grammars (CFGs).

Definition 5.4 ($\mathsf{CFG}^{\clubsuit}(q)$)**.** Let q be a query in $\mathsf{TreeBCQ}$ with root variable r. We define a context-free grammar $\mathsf{CFG}^{\clubsuit}(q)$ over the string representations of rooted relation trees for each rooted tree query q. The alphabet Σ of $\mathsf{CFG}^{\clubsuit}(q)$ contains every relation symbol and constant in q, open/close parentheses, $\perp$ and comma.

Whenever v is a variable or a constant in q, there is a nonterminal symbol S_v. Every symbol in Σ is a terminal symbol. The rules of $\mathsf{CFG}^{\clubsuit}(q)$ are as follows:

- for each atom $R[y] = R(\underline{y}, y_1, y_2, \dots, y_n)$ in q, there is a forward production rule
$$S_y \rightarrow_q R(S_{y_1}, S_{y_2}, \dots, S_{y_n}) \tag{5.2}$$
- whenever $R[x]$ and $R[y]$ are atoms in q such that $R[x] <_q R[y]$, there is a backward production rule
$$S_y \rightarrow_q S_x \tag{5.3}$$
- for every leaf variable u whose label L is either $\perp$ or a unary relation name, there is a rule
$$S_u \rightarrow_q L \tag{5.4}$$
- for each constant c in q, there is a rule
$$S_c \rightarrow_q c \tag{5.5}$$

The starting symbol of $\mathsf{CFG}^{\clubsuit}(q)$ is S_r where r is the root variable of q. A rooted relation tree τ is accepted by $\mathsf{CFG}^{\clubsuit}(q)$, denoted as $\tau \in \mathsf{CFG}^{\clubsuit}(q)$, if the string representation of τ can be derived from S_r, written as $S_r \overset{*}{\rightarrow}_q \tau$.

Example 5.3. Let q be as in Figure 5.2(a) with variables labeled as in Figure 5.2(b). The rooted relation tree τ in Figure 5.3(c) has string representation $\tau = A(\tau_1, \tau_2)$ where

$$\begin{aligned}\tau_1 &= R(R(R(U, \perp), X(c_1)), X(c_1)),\\ \tau_2 &= R(Y(\perp), Z(c_2, \perp)).\end{aligned}$$

We have $S_{x_2} \overset{*}{\rightarrow}_q \tau_2$ by applying only forward rewrite rules. We show next $S_{x_1} \overset{*}{\rightarrow}_q \tau_1$, using a backward rewrite rule $S_{x_3} \rightarrow_q S_{x_1}$ at some point:

$$\begin{aligned}
S_{x_1} &\to_q R(S_{x_3}, S_{x_4}) \\
&\to_q R(S_{x_1}, X(S_{c_1})) \\
&\to_q R(R(S_{x_3}, S_{x_4}), X(c_1)) \\
&\to_q R(R(R(S_{x_7}, S_{x_8}), X(S_{c_1})), X(c_1)) \\
&\to_q R(R(R(U, \bot)), X(c_1)), X(c_1)) = \tau_1.
\end{aligned}$$

Thus $S_{x_0} \to_q A(S_{x_1}, S_{x_2}) \overset{*}{\to}_q A(\tau_1, \tau_2) = \tau$. So it is correct to conclude that τ is accepted by $\mathsf{CFG}^{\clubsuit}(q)$.

Recall from Section 5.1 that a rooted tree set in a repair $\mathbf{r}$ is uniquely determined by a rooted tree τ and a constant c; such a rooted tree set is said to be accepted by $\mathsf{CFG}^{\clubsuit}(q)$ if $\tau \in \mathsf{CFG}^{\clubsuit}(q)$. For our technical treatment later, we next define modifications of $\mathsf{CFG}^{\clubsuit}(q)$ by changing its starting terminal.

Definition 5.5 ($\mathsf{S\text{-}CFG}^{\clubsuit}(q, u)$)**.** For a query q in $\mathsf{TreeBCQ}$ and a variable u in q, we define $\mathsf{S\text{-}CFG}^{\clubsuit}(q, u)$ as the context-free grammar that accepts a rooted relation tree τ if and only if $S_u \overset{*}{\to}_q \tau$.

We now introduce the *certain trace problem.* For each q in $\mathsf{TreeBCQ}$, $\mathsf{CERTAIN}_{\mathsf{tr}}(q)$ is defined as the following decision problem:

> **PROBLEM** $\mathsf{CERTAIN}_{\mathsf{tr}}(q)$
>
> **Input**: A database instance $\mathbf{db}$.
>
> **Question**: Is there a constant $c \in \mathbf{db}$, such that for every repair $\mathbf{r}$ of $\mathbf{db}$, there is a rooted tree set τ in $\mathbf{r}$ starting in c with $\tau \in \mathsf{CFG}^{\clubsuit}(q)$?

The problems $\mathsf{CERTAINTY}(q)$ and $\mathsf{CERTAIN}_{\mathsf{tr}}(q)$ reduce to each other if q satisfies $\mathcal{C}_2$.

Lemma 5.2. *Let q be a query in* $\mathsf{TreeBCQ}$ *satisfying $\mathcal{C}_2$. Let* $\mathbf{db}$ *be a database instance. Then the following statements are equivalent:*

1. $\mathbf{db}$ *is a "yes"-instance of* $\mathsf{CERTAINTY}(q)$*; and*
2. $\mathbf{db}$ *is a "yes"-instance of* $\mathsf{CERTAIN}_{\mathsf{tr}}(q)$*.*

The proof of Lemma 5.2 is deferred to Section 5.5 since it requires some useful results to be developed in the subsequent sections.

Initialization Step: **for every** $c \in \mathsf{adom}(\mathbf{db})$ and leaf variable or constant u in q
add $\langle c, u\rangle$ to B **if** $u = c$ is a constant,
or the label of variable u in q is either $\perp$,
or L with $L(\underline{c}) \in \mathbf{db}$.

Iterative Rule: **for every** $c \in \mathsf{adom}(\mathbf{db})$ and atom $R(\underline{y}, y_1, y_2, \dots, y_n)$ in q
add $\langle c, y\rangle$ to B **if** the following formula holds:

$$\exists \vec{d} : R(\underline{c}, \vec{d}) \in \mathbf{db} \land \forall \vec{d} : \left(R(\underline{c}, \vec{d}) \in \mathbf{db} \rightarrow \mathsf{fact}(R(\underline{c}, \vec{d}), y)\right),$$

where

$$\mathsf{fact}(R(\underline{c}, \vec{d}), y) = \underbrace{\left(\bigwedge_{1 \leq i \leq n} \langle d_i, y_i\rangle \in B\right)}_{\text{forward production}} \lor \underbrace{\left(\bigvee_{R[x] <_q R[y]} \mathsf{fact}(R(\underline{c}, \vec{d}), x)\right)}_{\text{backward production}}$$

and $\vec{d} = \langle d_1, d_2, \dots, d_n\rangle$.

Figure 5.6: A fixpoint algorithm for computing a set B, for a fixed rooted tree q.

5.4 Membership of $\mathsf{CERTAIN}_{\mathsf{tr}}(q)$ in LFPL

In this section, we show that the problem $\mathsf{CERTAIN}_{\mathsf{tr}}(q)$ is expressible in **LFPL** (and thus in **PTIME**) if q satisfies $\mathcal{C}_{\mathsf{branch}}$. Let $\mathbf{db}$ be a database instance. Consider the algorithm in Figure 5.6, following a dynamic programming fashion. The algorithm iteratively computes a set B of pairs $\langle c, y\rangle$ until it reaches a fixpoint, ensuring that

whenever $\langle c, y\rangle$ is added to B, then every repair of $\mathbf{db}$ contains a rooted tree set starting in c that is accepted by $\mathsf{S\text{-}CFG}^{\clubsuit}(q, y)$.

Intuitively, this holds true because $\langle c, y\rangle$ is added to B if for every possible fact $f = R(\underline{c}, \vec{d})$ that can be chosen by a repair of $\mathbf{db}$, the context-free grammar $\mathsf{S\text{-}CFG}^{\clubsuit}(q, y)$ can proceed by firing forward rule with nonterminal S_y that consumes f from the rooted tree set, or by non-deterministically firing some backward rule of the form $S_y \rightarrow_q S_x$.

The formal semantics for each pair $\langle c, y\rangle$ is stated in Lemma 5.3.

Lemma 5.3. *Let q be a query in* $\mathsf{TreeBCQ}$ *satisfying $\mathcal{C}_{\mathsf{branch}}$. Let $\mathbf{db}$ be a database instance. Let B be the output of the algorithm in Figure 5.6. Then for any constant $c \in \mathsf{adom}(\mathbf{db})$ and a variable y in q, the following statements are equivalent:*

1. *$\langle c, y\rangle \in B$; and*
2. *for every repair $\mathbf{r}$ of $\mathbf{db}$, there exists a rooted tree set τ in $\mathbf{r}$ starting in c such that $\tau \in \mathsf{S\text{-}CFG}^{\clubsuit}(q, y)$.*

The crux in the proof of Lemma 5.3 relies on the existence of repairs called *frugal*: to show item (2) of Lemma 5.3, it will be sufficient to show that it holds true for frugal repairs. Frugal repairs also turn out to be useful in proving Lemma 5.2 and offer an alternative perspective to the algorithm, as stated in Corollary 5.1.

5.4.1 Frugal repairs

We show that the evaluation result of the predicate "fact" and the membership in B in the algorithm of Figure 5.6 propagate along the total preorder $\preceq_q$. The first step is to show that the formula in Figure 5.6 propagate on root homomorphism.

Lemma 5.4. *Let q be a rooted tree query and* $\mathbf{db}$ *a database instance. Then for constants $c, d_1, d_2, \ldots, d_n \in \mathsf{adom}(\mathbf{db})$ where $\vec{d} = \langle d_1, d_2, \ldots, d_n \rangle$ and any two atoms $R[x]$ and $R[y]$ with $q_\triangle^y \leq_{y \to x} q_\triangle^x$, the following statements hold:*

1. *if $\mathsf{fact}(R(\underline{c}, \vec{d}), x)$ is true, then $\mathsf{fact}(R(\underline{c}, \vec{d}), y)$ is true; and*
2. *if $\langle c, x \rangle \in B$, then $\langle c, y \rangle \in B$.*

Proof. We show both (1) and (2) by an induction on the height k of the atom $R[y]$ in q.

- Basis $k = 0$. In this case, y is a leaf variable of q and (1) holds vacuously. Assume that the label of y is L, then there is an atom $L(\underline{y})$ in q. Then there must be an atom $L(\underline{x})$ in q. From $\langle c, x \rangle \in B$, we have $L(\underline{c}) \in \mathbf{db}$, and thus $\langle c, y \rangle \in B$ by the initialization step.
- Inductive step. Assume that both (1) and (2) holds if the height of $q_\triangle^y$ is less than k. Consider the case where the height of $q_\triangle^y$ is k.
  First we show (1) holds. Assume that $\mathsf{fact}(R(\underline{c}, \vec{d}), x)$ holds. Let $R[x] = R(\underline{x}, x_1, x_2, \ldots, x_n)$ and $R[y] = R(\underline{y}, y_1, y_2, \ldots, y_n)$. Consider two cases.
  - Case (I) that the following formula is true
  $$\bigwedge_{1 \leq i \leq n} \langle d_i, x_i \rangle \in B. \tag{5.6}$$
  To show $\mathsf{fact}(R(\underline{c}, \vec{d}), y)$ holds, it suffices to show
  $$\bigwedge_{1 \leq i \leq n} \langle d_i, y_i \rangle \in B.$$
  Consider any y_i. If y_i is a leaf variable with label $\perp$, then $\langle d_i, y_i \rangle \in B$ by the initialization step. Otherwise, there is an atom $T[y_i]$ in q. Since $q_\triangle^y \leq_{y \to x} q_\triangle^x$, there is some atom $T[x_i]$ in q such that $q_\triangle^{y_i} \leq_{y_i \to x_i} q_\triangle^{x_i}$ and $\langle d_i, x_i \rangle \in B$, by Equation (5.6). Since the height of $T[y_i]$ is less than k, by the inductive hypothesis for (2), we have $\langle d_i, y_i \rangle \in B$.

– Case (II) that there is some atom $R[u] <_q R[x]$, such that $\mathsf{fact}(R(\underline{c}, \vec{d}), u)$ is true.

  If $R[u] <_q R[y]$, then $\mathsf{fact}(R(\underline{c}, \vec{d}), y)$ holds. Otherwise, we must have $R[u] \parallel_q R[y]$. Indeed, if not, we would have $R[y] <_q R[u] <_q R[x]$, but $q_\Delta^y \leq_{y \to x} q_\Delta^x$, a contradiction.

  We argue that $q_\Delta^y \leq_{y \to u} q_\Delta^u$. If not, then by $\mathcal{C}_{\mathsf{branch}}$, we have $q_\Delta^u \leq_{u \to y} q_\Delta^y \leq_{y \to x} q_\Delta^x$, but $R[u] <_q R[x]$, a contradiction.

  Note that we just established $\mathsf{fact}(R(\underline{c}, \vec{d}), u)$ is true and $q_\Delta^y \leq_{y \to u} q_\Delta^u$ for $R[u] <_q R[x]$. If Case (I) holds when $\mathsf{fact}(R(\underline{c}, \vec{d}), u)$ is true, then $\mathsf{fact}(R(\underline{c}, \vec{d}), y)$ is true, as desired. Otherwise, by the previous argument in Case (II), either $\mathsf{fact}(R(\underline{c}, \vec{d}), y)$ is true as desired, or there is another atom $R[w]$ such that $R[w] <_q R[u] <_q R[x]$ and $q_\Delta^y \leq_{y \to w} q_\Delta^w$. Since there are only finitely many R-atoms in q, this process must terminate and show that $\mathsf{fact}(R(\underline{c}, \vec{d}), y)$ is true.

For (2), assume that $\langle c, x \rangle \in B$. For every fact $R(\underline{c}, \vec{d}) \in \mathbf{db}$, $\mathsf{fact}(R(\underline{c}, \vec{d}), x)$ holds. By (1), $\mathsf{fact}(R(\underline{c}, \vec{d}), y)$ holds for every $R(\underline{c}, \vec{d}) \in \mathbf{db}$. Hence $\langle c, y \rangle \in B$.

The proof is now complete. □

Lemma 5.5. *Let q be a query in* TreeBCQ *satisfying $\mathcal{C}_{\mathsf{branch}}$, and* **db** *a database instance. Let $R[x], R[y]$ be two atoms of q. Then for every fact $R(\underline{c}, \vec{d})$ in* **db** *and two atoms $R[x] \preceq_q R[y]$,*

1. *if $\mathsf{fact}(R(\underline{c}, \vec{d}), x)$ is true, then $\mathsf{fact}(R(\underline{c}, \vec{d}), y)$ is true, where* fact *is defined by the algorithm of Figure 5.6; and*
2. *if $\langle c, x \rangle \in B$, then $\langle c, y \rangle \in B$, where B is the output of the algorithm of Figure 5.6; and*

Proof. The lemma follows from Lemma 5.4 if $q_\Delta^y \leq_{y \to x} q_\Delta^x$. Assume that $R[x] <_q R[y]$, and both (1) and (2) are straightforward by definition of $\mathsf{fact}(R(\underline{c}, \vec{d}), y)$. □

Definition 5.6 (Frugal Set)**.** Let q be a query in TreeBCQ satisfying $\mathcal{C}_{\mathsf{branch}}$, and **db** a database instance. Let $f = R(\underline{c}, \vec{d})$ be an R-fact in **db**. We define the frugal set of f in **db** with respect to q as

$$\mathsf{ST}_q(f, \mathbf{db}) = \{R[x] \in \mathsf{atoms}(q) \mid \mathsf{fact}(R(\underline{c}, \vec{d}), x) \text{ is true}\}.$$

Lemma 5.6. *Let q be a query in* TreeBCQ *satisfying $\mathcal{C}_{\mathsf{branch}}$, and* **db** *a database instance. For every two key-equal facts f and g in* **db**, *the sets $\mathsf{ST}_q(f, \mathbf{db})$ and $\mathsf{ST}_q(g, \mathbf{db})$ are comparable by $\subseteq$.*

Proof. Suppose for contradiction that there exists two key-equal facts $f = R(\underline{c}, \vec{d_1})$ and $g = R(\underline{c}, \vec{d_2})$ in **db** such that $R[x] \in \mathsf{ST}_q(f, \mathbf{db}) \setminus \mathsf{ST}_q(g, \mathbf{db})$ and $R[y] \in \mathsf{ST}_q(g, \mathbf{db}) \setminus \mathsf{ST}_q(f, \mathbf{db})$. By Proposition 5.2, assume without loss of generality that $R[x] \preceq_q R[y]$. Then since $R[x] \in \mathsf{ST}_q(f, \mathbf{db})$, we have $\mathsf{fact}(R(\underline{c}, \vec{d_1}), x)$ is true, and thus $\mathsf{fact}(R(\underline{c}, \vec{d_1}), y)$ is true by Lemma 5.5, and hence $R[y] \in \mathsf{ST}_q(f, \mathbf{db})$, a contradiction. A similar contradiction can also be reached if $R[y] \preceq_q R[x]$. This completes the proof. $\square$

Informally, by Lemma 5.6, among all facts of a non-empty block $R(\underline{c}, *)$ in $\mathbf{db}$, there is a (not necessarily unique) fact $R(\underline{c}, \vec{d})$ with a $\subseteq$-minimal frugal set in $\mathbf{db}$. The repair $\mathbf{r}^*$ of $\mathbf{db}$ containing all such facts is frugal in the sense that each fact in it satisfies as few R-atoms as possible; and if $\mathbf{r}^*$ contains a rooted tree set τ starting in c accepted by $\mathsf{S\text{-}CFG}^{\clubsuit}(q, y)$, so should every repair of $\mathbf{db}$. We now formalize this idea.

Definition 5.7 (Frugal repair)**.** Let q be a query in $\mathsf{TreeBCQ}$ satisfying $\mathcal{C}_{\mathsf{branch}}$. Let $\mathbf{db}$ be a database instance. A *frugal repair* $\mathbf{r}^*$ of $\mathbf{db}$ with respect to q is constructed by picking, from each block $R(\underline{c}, *)$ of $\mathbf{db}$, a fact $R(\underline{c}, \vec{d})$ which $\subseteq$-minimizes $\mathsf{ST}_q(R(\underline{c}, \vec{d}), \mathbf{db})$.

Lemma 5.7 is then straightforward by construction of a frugal repair.

Lemma 5.7. *Let q be a rooted tree query satisfying $\mathcal{C}_{\mathsf{branch}}$. Let $\mathbf{db}$ be a database instance. Let $\mathbf{r}^*$ be a frugal repair of $\mathbf{db}$ with respect to q and let $R(\underline{c}, \vec{d}) \in \mathbf{r}^*$. Let $R[u]$ be an atom in q. If $\mathsf{fact}(R(\underline{c}, \vec{d}), u)$ is true, then $\langle c, u\rangle \in B$.*

Proof. Let $R(\underline{c}, \vec{b})$ be an arbitrary fact in the block $R(\underline{c}, *)$ in $\mathbf{db}$. By construction of a frugal repair, we have that $\mathsf{ST}_q(R(\underline{c}, \vec{d}), \mathbf{db}) \subseteq \mathsf{ST}_q(R(\underline{c}, \vec{b}), \mathbf{db})$. Since $R(\underline{c}, \vec{d}) \in \mathbf{r}^*$ and $\mathsf{fact}(R(\underline{c}, \vec{d}), u)$ is true, we have $R[u] \in \mathsf{ST}_q(R(\underline{c}, \vec{d}), \mathbf{db})$. Thus, $R[u] \in \mathsf{ST}_q(R(\underline{c}, \vec{b}), \mathbf{db})$ and $\mathsf{fact}(R(\underline{c}, \vec{b}), u)$ is true. Hence $\langle c, u\rangle \in B$. $\square$

Lemma 5.8 shows a desirable property of frugal repairs.

Lemma 5.8. *Let q be a query in $\mathsf{TreeBCQ}$ satisfying $\mathcal{C}_{\mathsf{branch}}$. Let $\mathbf{db}$ be a database instance. Let $\mathbf{r}^*$ be a frugal repair of $\mathbf{db}$ with respect to q. If there is a rooted tree set τ in $\mathbf{r}^*$ starting in c such that $\tau \in \mathsf{S\text{-}CFG}^{\clubsuit}(q, y)$, then $\langle c, y\rangle \in B$.*

Proof. Let τ be a rooted tree set starting in c in $\mathbf{r}^*$ such that $\tau \in \mathsf{S\text{-}CFG}^{\clubsuit}(q, y)$. We recursively define a tree trace $\mathcal{T}$ on nodes of the form (c, x), where $c \in \mathsf{adom}(\mathbf{r}^*)$ and x is a variable in q, as follows:

- the root node of $\mathcal{T}$ is (c, y, τ); and
- whenever (a, u, σ) is a node in $\mathcal{T}$ with a rooted tree set σ starting in a in $\mathbf{r}^*$ for an atom $R(\underline{u}, u_1, u_2, \dots, u_n)$ in q and fact $R(\underline{a}, b_1, b_2, \dots, b_n)$ in $\mathbf{r}^*$,

(i) if $\mathsf{S\text{-}CFG}^{\clubsuit}(q,y)$ invokes a forward production rule

$$S_u \to_q R(S_{u_1}, S_{u_2}, \dots, S_{u_n}),$$

then the node (a,u,σ) has n outgoing R-edges to its children (b_1,u_1,τ_1), (b_2,u_2,τ_2), $\dots$, (b_n,u_n,τ_n); or

(ii) if $\mathsf{S\text{-}CFG}^{\clubsuit}(q,y)$ invokes a backward production rule $S_u \to_q S_v$, then the node (a,u,σ) has a single outgoing ε-edge to its only child (a,v,σ).

The tree trace $\mathcal{T}$ succinctly records the rule productions that witness $\tau \in \mathsf{S\text{-}CFG}^{\clubsuit}(q,y)$ in $\mathbf{r}^*$. We use a structural induction to show that for every node (a,u,σ) in $\mathcal{T}$, $\langle a,u\rangle \in B$.

- Basis. The claim holds for every leaf node (a,u,σ) in $\mathcal{T}$, since if $\sigma = \bot$, then $\langle a,u\rangle \in B$, or otherwise $\sigma = L$ starting in a in $\mathbf{r}^*$ for some unary relation name L, and we have $L(a)$ is in $\mathbf{db}$.
- Inductive step. Let (a,u,σ) be a node in $\mathcal{T}$. Assume that for any child node (b,w,σ') of (a,u) in $\mathcal{T}$ (possibly $b=a$), $\langle b,w\rangle \in B$. It suffices to argue that for the atom $R[u] = R(\underline{u}, u_1, u_2, \dots, u_n)$ in q, $\langle a,u\rangle \in B$.
  (i) Case that (a,u,σ) has child nodes (b_1,u_1,τ_1), (b_2,u_2,τ_2), $\dots$, (b_n,u_n,τ_n) in $\mathcal{T}$ with $\sigma = R(\tau_1,\tau_2,\dots,\tau_n)$. By the inductive hypothesis $\langle b_i,u_i\rangle \in B$ for every $1 \le i \le n$, which yields that $\mathsf{fact}(R(\underline{a},\vec{b}),u)$ is true, where $\vec{b} = \langle b_1,b_2,\dots,b_n\rangle$. Then by Lemma 5.7, $\langle a,u\rangle \in B$.
  (ii) Case that (a,u,σ) has a child node (a,v,σ) in $\mathcal{T}$ connected with an ε-edge. Then there is some atom $R[v]$ with $R[v] <_q R[u]$. By the inductive hypothesis on the child (a,v,σ), $\langle a,v\rangle \in B$. Hence $\langle a,u\rangle \in B$ by Lemma 5.5.

This completes the proof. □

The proof of Lemma 5.3 can now be given.

Proof of Lemma 5.3. $\boxed{2 \Longrightarrow 1}$ Let $\mathbf{r}^*$ be a frugal repair of $\mathbf{db}$ with respect to q. Then there is a rooted tree set τ starting in c in $\mathbf{r}^*$ with $\tau \in \mathsf{S\text{-}CFG}^{\clubsuit}(q,y)$. The claim follows by Lemma 5.8.

$\boxed{1 \Longrightarrow 2}$ Assume that $\langle c,y\rangle \in B$. We use induction on k to show that if $\langle c,y\rangle$ is added to B at the k-th iteration, then for any repair $\mathbf{r}$ of $\mathbf{db}$, there exists a rooted tree set τ starting in c in τ with $\tau \in \mathsf{S\text{-}CFG}^{\clubsuit}(q,y)$.

- Basis $k=0$. Then every $\langle c,u\rangle$ is added to B for every leaf variable u of q such that either the label of u in q is $\bot$, or a unary relation name L. If the label of u is $\bot$, the empty rooted tree set $\tau = \emptyset$ starting in c with string representation $\bot$ is accepted by $\mathsf{S\text{-}CFG}^{\clubsuit}(q,u)$, Otherwise, we must have $L(c) \in \mathbf{db}$, and the rooted tree set $\tau = L$ starting in c is accepted by $\mathsf{S\text{-}CFG}^{\clubsuit}(q,u)$.

- Inductive step. Assume that $\langle c, y\rangle$ is added to B in the k-th iteration, and for any tuple $\langle b, x\rangle$ added to B prior to the addition of $\langle c, y\rangle$, any repair of $\mathbf{db}$ contains a rooted tree set $\tau \in \mathsf{S\text{-}CFG}^{\clubsuit}(q, x)$ starting in b. Let $\mathbf{r}$ be any repair of $\mathbf{db}$. It suffices to construct a rooted tree set τ in $\mathbf{r}$ starting in c such that $\tau \in \mathsf{S\text{-}CFG}^{\clubsuit}(q, y)$. Let $R[y] = R(\underline{y}, y_1, y_2, \ldots, y_n)$. Let $R(\underline{c}, d_1, d_2, \ldots, d_n) \in \mathbf{r}$ and let $\vec{d} = \langle d_1, d_2, \ldots, d_n\rangle$. Since $\langle c, y\rangle \in B$, $\mathsf{fact}(R(\underline{c}, \vec{d}), y)$ is true. Consider two cases.
  - Case that $\langle d_i, y_i\rangle \in B$ for every $1 \leq i \leq n$. Since each $\langle d_i, y_i\rangle$ was added to B in an iteration $< k$, by the inductive hypothesis, there is a rooted tree set τ_i starting in d_i in $\mathbf{r}$ with $\tau_i \in \mathsf{S\text{-}CFG}^{\clubsuit}(q, y_i)$, i.e., $S_{y_i} \overset{*}{\rightarrow}_q \tau_i$. Consider the rooted tree set $\tau = \{R(\underline{c}, \vec{d})\} \cup \bigcup_{1 \leq i \leq n} \tau_i$, starting in c in $\mathbf{r}$ with a string representation $\tau = R(\tau_1, \tau_2, \ldots, \tau_n)$. From

    $$\begin{aligned} S_y &\rightarrow_q R(S_{y_1}, S_{y_2}, \ldots, S_{y_n}) \\ &\overset{*}{\rightarrow}_q R(\tau_1, \tau_2, \ldots, \tau_n) = \tau, \end{aligned}$$

    we conclude that $\tau \in \mathsf{S\text{-}CFG}^{\clubsuit}(q, y)$.
  - Case that $\mathsf{fact}(R(\underline{c}, \vec{d}), x)$ is true for some $R[x] <_q R[y]$. Without loss of generality, we assume that x is the smallest with respect to $<_q$ for the atom $R(\underline{x}, x_1, x_2, \ldots, x_n)$. Hence we must have $\langle d_i, x_i\rangle \in B$ for every $1 \leq i \leq n$, and by the previous case, there exists a rooted tree set τ_i starting in d_i such that $\tau_i \in \mathsf{S\text{-}CFG}^{\clubsuit}(q, x_i)$, i.e., $S_{x_i} \overset{*}{\rightarrow}_q \tau_i$. Since $R[x] <_q R[y]$, we have

    $$\begin{aligned} S_y &\rightarrow_q S_x \\ &\rightarrow_q R(S_{x_1}, S_{x_2}, \ldots, S_{x_n}) \\ &\overset{*}{\rightarrow}_q R(\tau_1, \tau_2, \ldots, \tau_n) = \tau, \end{aligned}$$

    and therefore $\tau \in \mathsf{S\text{-}CFG}^{\clubsuit}(q, y)$.

The proof is hence complete. $\square$

5.4.2 Expressibility in LFPL and FO

Lemma 5.9. *For every query q in* TreeBCQ *that satisfies* $\mathcal{C}_{\mathsf{branch}}$*,* $\mathsf{CERTAIN}_{\mathsf{tr}}(q)$ *is expressible in* **LFPL** *(and thus is in* **PTIME***).*

Proof. Let r be the root variable of q. Our algorithm first computes the set B, and then checks $\exists c : \langle c, r\rangle \in B$. The algorithm is correct by Lemma 5.3.

For a rooted tree query q, define the following formula in **LFPL** [Lib04]:

$$\psi_q(s,t) := \left[\mathbf{lfp}_{B,x,z}\varphi_q(B,x,z)\right](s,t), \tag{5.7}$$

where we have

$$\varphi_q(B,x,z) := \left(\begin{array}{ll} & \alpha(x) \wedge (z = R[u]) \\ \wedge & \exists y R(\underline{x}, \vec{y}) \\ \wedge & \forall \vec{y}\,(R(\underline{x},\vec{y}) \rightarrow \mathsf{fact}(R(\underline{x},\vec{y}), u)) \end{array}\right)$$

and the formula $\mathsf{fact}(R(\underline{x},\vec{y}),u)$ is defined in Figure 5.6. The initialization step of B in Figure 5.6 is expressible in **FO**. Herein, $\alpha(x)$ denotes a first-order query that computes the active domain. That is, for every database instance $\mathbf{db}$ and constant c, $\mathbf{db} \models \alpha(c)$ if and only if $c \in \mathsf{adom}(\mathbf{db})$. It is easy to verify that the LFP formula in (5.7) computes the set B in Figure 5.6. □

We now show that if q satisfies $\mathcal{C}_1$, we can safely remove the recursion from the algorithm in Figure 5.6.

Lemma 5.10. *Let q be a rooted tree query satisfying $\mathcal{C}_1$. Let $\mathbf{db}$ be a database. Let $R[y] = R(\underline{y}, y_1, y_2, \ldots, y_n)$ be an atom in q and let $R(\underline{c}, \vec{d}) = R(\underline{c}, d_1, d_2, \ldots, d_n)$ be a fact in $\mathbf{db}$. Then $\mathsf{fact}(R(\underline{c},\vec{d}), y)$ is true if and only if for every atom $T_i[y_i]$ in q, $\langle d_i, y_i\rangle \in B$.*

Proof. $\boxed{\Longleftarrow}$ Immediate by definition of $\mathsf{fact}(R(\underline{c},\vec{d}), y)$.

$\boxed{\Longrightarrow}$ Assume that $\mathsf{fact}(R(\underline{c},\vec{d}), y)$ is true. Let $R[x]$ be a minimal atom with respect to $<_q$ such that $R[x] <_q R[y]$ and $\mathsf{fact}(R(\underline{c},\vec{d}), x)$ is true. If such an atom $R[x]$ does not exist, then the claim follows by definition of $\mathsf{fact}(R(\underline{c},\vec{d}), y)$. Otherwise, since $R[x]$ is minimal with respect to $<_q$, for every atom $T_i[x_i]$ in q, $\langle d_i, x_i\rangle \in B$, where $R(\underline{x},\vec{x}) = R(\underline{x}, x_1, x_2, \ldots, x_n)$.

It suffices to show that for every atom $T_i[y_i]$ in q, $\langle d_i, y_i\rangle \in B$. Let $T_i[y_i]$ be an atom in q. From $\mathcal{C}_1$ and $R[x] <_q R[y]$, $q_{\triangle}^{y_i} \leq_{y_i \rightarrow x_i} q_{\triangle}^{x_i}$. Thus there is some atom $T_i[x_i]$ in q with $T_i[x_i] <_q T_i[y_i]$. Since $\langle d_i, x_i\rangle \in B$, by Lemma 5.4, $\langle d_i, y_i\rangle \in B$. □

Lemma 5.11. $\mathsf{CERTAIN}_{\mathsf{tr}}(q)$ *is in* **FO** *for each q in* $\mathsf{TreeBCQ}$ *that satisfies $\mathcal{C}_1$.*

Proof. Consider the following variant of the algorithm in Figure 5.6, where we simply have

$$\mathsf{fact}(R(\underline{c},\vec{d}), y) = \bigwedge_{1 \leq i \leq n} \langle d_i, y_i\rangle \in B.$$

The variant algorithm is correct for $\mathsf{CERTAIN}_{\mathsf{tr}}(q)$ by Lemma 5.10. Since the size of the query q is fixed, for every constant c and variable y in q, deciding whether $\langle c, y\rangle \in B$ is in **FO** since the algorithm in Figure 5.6 can be expanded into a sentence of fixed size. So is our algorithm, which checks $\exists c : \langle c, r\rangle \in B$. □

5.5 Complexity Upper Bounds

In this section, we prove the upper bound results in Theorem 5.4. First, we shall prove Lemma 5.2.

Lemma 5.12. *Let q be a rooted tree query. Then q satisfies $\mathcal{C}_{\mathsf{factor}}$ if and only if $q \leq_{\rightarrow} \tau$ for every $\tau \in \mathsf{NFA}^{\clubsuit}(q)$.*

Proof of Lemma 5.12. Consider two directions.

$\boxed{\Longleftarrow}$ Let $R[x]$ and $R[y]$ be two atoms in q with $R[x] <_q R[y]$. It suffices to show that $q^{R:y\looparrowright x} \in \mathsf{NFA}^{\clubsuit}(q)$. Indeed, there is an execution of $S_r(q^{R:y\looparrowright x})$ that follows exactly $S_r(q)$, until it invokes $S_y(q^x_\triangle)$, instead of $S_y(q^y_\triangle)$ in $S_r(q)$. Note that $S_y(q^x_\triangle) = S_x(q^x_\triangle) = q^x_\triangle$. Thus $S_r(q^{R:y\looparrowright x}) = q^{R:y\looparrowright x}$, concluding that $q^{R:y\looparrowright x} \in \mathsf{NFA}^{\clubsuit}(q)$.

$\boxed{\Longrightarrow}$ Let $\tau \in \mathsf{NFA}^{\clubsuit}(q)$ with $S_r(\tau) = \tau$. We use an induction on the number k of backward transitions in $S_r(\tau) = \tau$ to show that $q \leq_{\rightarrow} \tau$.

- Basis $k = 0$. We have $\tau = q$, and the claim follows.
- Inductive step $k \rightarrow k+1$. Assume that if $S_r(\sigma) = \sigma$ uses k backward transitions, then $q \leq_{\rightarrow} \sigma$. Let $\tau \in \mathsf{NFA}^{\clubsuit}(q)$ such that $S_r(\tau) = \tau$ uses $k+1$ backward transitions. Let σ be a subtree of τ such that the execution of $S_r(\sigma)$ invokes exactly 1 backward transition $S_y(\sigma) = S_x(\sigma) = \sigma$. Hence $\sigma = q^x_\triangle$. Consider the rooted tree τ^*, obtained by replacing $\sigma = q^x_\triangle$ with $\sigma^* = q^y_\triangle$. We have $\tau^* \in \mathsf{NFA}^{\clubsuit}(q)$, since $S_r(\tau^*)$ would invoke $S_y(\sigma^*) = \sigma^*$ and use exactly k backward transitions. By the inductive hypothesis, there is a homomorphism h from q to τ^*. If $h(q) \cap \sigma^* = \emptyset$, then $h(q)$ is still present in τ, and thus $q \leq_{\rightarrow} \tau$. Otherwise, assume that the homomorphism h maps $q^z_\triangle$ in q to σ^*. Hence $R[x] <_q R[y] <_q R[z]$. Since q satisfies $\mathcal{C}_{\mathsf{factor}}$, there is a homomorphism g from q to $q^{R:z\looparrowright x}$, and thus a homomorphism from q to τ.

The proof is now complete. $\square$

The following definition is helpful in our exposition.

Definition 5.8. Let q be a rooted tree query. Let $\mathbf{db}$ be a database. For each repair $\mathbf{r}$ of $\mathbf{db}$, we define $\mathsf{start}(q, \mathbf{r})$ as the set containing all (and only) constants $c \in \mathsf{adom}(\mathbf{r})$ such that there is a rooted tree set τ in $\mathbf{r}$ starting in c with $\tau \in \mathsf{NFA}^{\clubsuit}(q)$.

The problem $\mathsf{CERTAIN}_{\mathsf{tr}}(q)$ essentially asks whether there is some constant c such that for every repair $\mathbf{r}$ of $\mathbf{db}$, $c \in \mathsf{start}(q, r)$. Surprisingly, the frugal repair $\mathbf{r}^*$ of $\mathbf{db}$ minimizes $\mathsf{start}(q, \cdot)$ across all repairs of $\mathbf{db}$.

Lemma 5.13. *Let q be a rooted tree query satisfying $\mathcal{C}_{\mathsf{branch}}$. Let* $\mathbf{db}$ *be a database. Let* $\mathbf{r}^*$ *be a frugal repair of* $\mathbf{db}$*. Then for any repair* $\mathbf{r}$ *of* $\mathbf{db}$*,* $\mathsf{start}(q, \mathbf{r}^*) \subseteq \mathsf{start}(q, \mathbf{r})$*.*

Proof of Lemma 5.13. Let B be the output of the algorithm in Figure 5.6. Let $\mathbf{r}^*$ be a frugal repair of $\mathbf{db}$. Let $\mathbf{r}$ be any repair of $\mathbf{db}$. We show that $\mathsf{start}(q, \mathbf{r}^*) \subseteq \mathsf{start}(q, \mathbf{r})$. Let r be the root variable of q. Assume that $c \in \mathsf{start}(q, \mathbf{r}^*)$. Then there exists a rooted tree set τ starting in c in $\mathbf{r}^*$ with $\tau \in \mathsf{NFA}^{\clubsuit}(q) = \mathsf{S\text{-}NFA}^{\clubsuit}(q, r)$. By Lemma 5.8, we have $\langle c, r \rangle \in B$. By Lemma 5.3, there exists a rooted tree set τ' starting in c in $\mathbf{r}$ with $\tau' \in \mathsf{S\text{-}NFA}^{\clubsuit}(q, r) = \mathsf{NFA}^{\clubsuit}(q)$. Thus $c \in \mathsf{start}(q, \mathbf{r})$. □

The proof of Lemma 5.2 can now be given.

Proof of Lemma 5.2. $\boxed{1 \Longrightarrow 2}$ Assume (1). Let $\mathbf{r}^*$ be a frugal repair of $\mathbf{db}$. Since $\mathbf{r}^*$ satisfies q, there is a rooted tree set starting in c isomorphic to q in $\mathbf{r}^*$. Since $q \in \mathsf{NFA}^{\clubsuit}(q)$, we have $c \in \mathsf{start}(q, \mathbf{r}^*)$. By Lemma 5.13, for every repair $\mathbf{r}$ of $\mathbf{db}$, $\mathsf{start}(q, \mathbf{r}^*) \subseteq \mathsf{start}(q, \mathbf{r})$. It follows that $c \in \mathsf{start}(q, \mathbf{r})$ for every repair $\mathbf{r}$ of $\mathbf{db}$.

$\boxed{2 \Longrightarrow 1}$ Let $\mathbf{r}$ be any repair of $\mathbf{db}$. By our hypothesis that (2) holds true, there is some $c \in \mathsf{start}(q, \mathbf{r})$. Let τ be a rooted tree set in $\mathbf{r}$ starting in c with $\tau \in \mathsf{NFA}^{\clubsuit}(q)$. Since q satisfies $\mathcal{C}_2$ by the hypothesis of the current lemma, it follows by Lemma 5.12 that $q \leq_{\rightarrow} \tau$. Consequently, $\mathbf{r}$ satisfies q. □

Proposition 5.3. *For every q in* $\mathsf{TreeBCQ}$*,*

1. *if q satisfies $\mathcal{C}_2$, then* $\mathsf{CERTAINTY}(q)$ *is in* **LFPL***; and*
2. *if q satisfies $\mathcal{C}_1$, then* $\mathsf{CERTAINTY}(q)$ *is in* **FO***.*

Proof. Immediate from Lemmas 5.2, 5.9, and 5.11 by noting that $\mathcal{C}_1$ implies $\mathcal{C}_2$. □

Interestingly, for each query q in $\mathsf{TreeBCQ}$ satisfying $\mathcal{C}_2$, "checking the frugal repair is all you need".

Corollary 5.1. *Let q be a query in* $\mathsf{TreeBCQ}$*, and let* $\mathbf{db}$ *be a database instance. Let* $\mathbf{r}^*$ *be a frugal repair of* $\mathbf{db}$ *with respect to q. If q satisfies $\mathcal{C}_2$, then* $\mathbf{db}$ *is a "yes"-instance of* $\mathsf{CERTAINTY}(q)$ *if and only if* $\mathbf{r}^*$ *satisfies q.*

Proof. Let $\mathbf{r}^*$ be a frugal repair of $\mathbf{db}$ with respect to q.

$\boxed{\Longrightarrow}$ This direction is straightforward. $\boxed{\Longleftarrow}$ Assume that $\mathbf{r}^*$ satisfies q. Let r be the root variable of q. Hence there exists a constant c in $\mathbf{db}$, such that there exists a rooted relation tree τ in $\mathbf{r}^*$ that is isomorphic to q accepted by $\mathsf{S\text{-}NFA}(q, r)$. Then by Lemma 5.8, $\langle c, r \rangle \in B$, where B is the output of algorithm in Figure 5.6. Hence $\mathbf{db}$ is a "yes"-instance for $\mathsf{CERTAIN}_{\mathsf{tr}}(q)$, and by Lemma 5.2, a "yes"-instance for $\mathsf{CERTAINTY}(q)$. □

5.6 Complexity Lower Bounds

In this section, we present the hardness results in Theorem 5.4.

We define a *canonical copy* of a query q as a set of facts $\mu(q)$, where μ maps each variable in q to a unique constant. The following notation will be central in all our reductions. For a query q, variables x_i in q and distinct constants c_i, we denote

$$\langle q, [x_1, x_2, \ldots, x_n \to c_1, c_2, \ldots, c_n] \rangle$$

as the canonical copy $\mu(q)$, where

$$\mu(z) = \begin{cases} c_i & \text{if } z = x_i \\ \text{a fresh distinct constant} & \text{otherwise.} \end{cases}$$

Lemma 5.14. *$\mathsf{CERTAINTY}(q)$ is* **coNP***-hard for each q in $\mathsf{TreeBCQ}$ that violates $\mathcal{C}_2$.*

Proof. Since q violates $\mathcal{C}_2$, there exist two atoms $R(\underline{p}, \ldots)$ and $R(\underline{n}, \ldots)$ in q such that there is no homomorphism from q to neither $q^{R:p \looparrowright n}$ nor $q^{R:n \looparrowright p}$.

Consider now the root atom $A(\underline{r}, \ldots)$. It must be that $r \neq p$, since otherwise, there would be a homomorphism from q to $q^{R:n \looparrowright p}$, a contradiction. Similarly, we have that $r \neq n$. Hence, the root atom is distinct from $R(\underline{p}, \ldots)$ and $R(\underline{n}, \ldots)$. We also have that $r <_q p$ and $r <_q n$.

We present a reduction from $\mathsf{MonotoneSAT}$: Given a monotone CNF formula φ, i.e., each clause in φ contains either all positive literals or all negative literals, does φ has a satisfying assignment?

Let φ be a monotone CNF formula. We construct an instance **db** for $\mathsf{CERTAINTY}(q)$ as follows.

- for each variable z in φ, we introduce the facts $\langle q^p_\Delta, [p \to z] \rangle$ and $\langle q^n_\Delta, [n \to z] \rangle$;
- for each positive literal z in clause C, we introduce the facts $\langle q \setminus q^p_\Delta, [r, p \to C, z] \rangle$;
- for each negative literal z in clause $\overline{C}$, we introduce the facts $\langle q \setminus q^n_\Delta, [r, n \to \overline{C}, z] \rangle$;

Observe that the instance **db** has two types of inconsistent blocks. For relation A, we have a block for each positive or negative clause, where the primary key position is the clause. For relation R, for every variable z we have a block of size two, which corresponds to choosing a true/false assignment for z. All the other relations are consistent.

Additionally, for a positive literal $z \in C$, the set of facts $\langle q^p_\Delta, [p \to z] \rangle \cup \langle q \setminus q^p_\Delta, [r, p \to C, z] \rangle$ make q true; similarly for a negative literal $z \in \overline{C}$, the facts $\langle q^n_\Delta, [n \to z] \rangle \cup \langle q \setminus q^n_\Delta, [n, p \to \overline{C}, z] \rangle$ make q true. Note also that $\langle q^n_\Delta, [n \to z] \rangle \cup \langle q \setminus q^p_\Delta, [r, p \to C, z] \rangle$ is a canonical copy of $q^{R:p \looparrowright n}$ (and hence cannot satisfy q), while $\langle q^p_\Delta, [p \to z] \rangle \cup \langle q \setminus q^n_\Delta, [r, n \to \overline{C}, z] \rangle$ is a canonical copy of $q^{R:n \looparrowright p}$ (which also cannot satisfy q).

Now we argue that φ has a satisfying assignment χ if and only if **db** has a repair **r** that does not satisfy q.

$\boxed{\Longrightarrow}$ Assume that φ has a satisfying assignment χ. Consider the repair **r** of **db** that

- for each variable z, if $\chi(z) = \mathsf{true}$, picks $\langle q^n_\Delta, [n \to z]\rangle$, or otherwise $\langle q^p_\Delta, [p \to z]\rangle$;
- for each positive clause C, picks $\langle q \setminus q^p_\Delta, [r, p \to C, z]\rangle$ where z is a positive literal in C with $\chi(z) = \mathsf{true}$; and
- for each negative clause $\overline{C}$, picks $\langle q \setminus q^n_\Delta, [r, n \to \overline{C}, \overline{z}]\rangle$ where $\overline{z}$ is a negative literal in $\overline{C}$ with $\chi(z) = \mathsf{false}$.

We show that **r** does not satisfy q. Indeed, for each positive clause C, there is a literal $z \in C$ with $\chi(z) = \mathsf{true}$, and thus $\langle q \setminus q^p_\Delta, [r, p \to C, z]\rangle \subseteq \mathbf{r}$. However, we have $\langle q^n_\Delta, [n \to z]\rangle \subseteq \mathbf{r}$, and thus q is not satisfied. Similarly, for each negative clause $\overline{C}$, there is a literal $\overline{z} \in \overline{C}$ with $\chi(z) = \mathsf{false}$, and thus $\langle q \setminus q^n_\Delta, [n, p \to \overline{C}, z]\rangle \subseteq \mathbf{r}$. However, we have $\langle q^p_\Delta, [p \to z]\rangle \subseteq \mathbf{r}$ and hence this part also cannot satisfy q. Hence **r** does not satisfy q.

$\boxed{\Longleftarrow}$ Now assume that **db** has a repair **r** that does not satisfy q. Consider the assignment χ that sets $\chi(z) = \mathsf{true}$ if $\langle q^n_\Delta, [n \to z]\rangle \subseteq \mathbf{r}$, or otherwise $\chi(z) = \mathsf{false}$. We argue that φ is satisfied. For each positive clause C, there exists some $z \in C$ such that $\langle q \setminus q^p_\Delta, [r, p \to C, z]\rangle \subseteq \mathbf{r}$. Since **r** does not satisfy q, it must be that $\langle q^p_\Delta, [p \to z]\rangle \nsubseteq \mathbf{r}$ and thus $\langle q^n_\Delta, [n \to z]\rangle \subseteq \mathbf{r}$. By construction, z is true and the clause C is satisfied. Similarly, the negative clauses are all satisfied by the assignment. □

Lemma 5.15. *Let q be a rooted tree query. If there exist two distinct atoms $R(\underline{x}, \dots)$ and $R(\underline{y}, \dots)$ such that $x <_q y$ and there is no root homomorphism from q^y_Δ to q^x_Δ (i.e., it does not hold that $q^y_\Delta \leq_{y \to x} q^x_\Delta$), then* $\mathsf{CERTAINTY}(q)$ *is* **NL**-*hard.*

Proof. We may assume without loss of generality two things (i) there is no atom $R(\underline{z}, \dots)$ such that $z \notin \{x, y\}$, $x <_q z <_q y$ (we then say that the two R-atoms are consecutive), and (ii) for any $y <_q z$, $z \neq y$, we have $q^z_\Delta \leq_{z \to y} q^y_\Delta$. Indeed, we can pick $R(\underline{x}, \dots)$ and $R(\underline{y}, \dots)$ to be the pair of consecutive R-atoms that violates the root homomorphism condition and occurs lowest in the rooted tree. Such a pair must always exists, since the root homomorphism property is transitive, i.e., if $q^y_\Delta \leq_{y \to z} q^z_\Delta$ and $q^z_\Delta \leq_{z \to x} q^x_\Delta$, then we also have that $q^y_\Delta \leq_{y \to x} q^x_\Delta$.

We present a reduction from the complement of $\mathsf{REACHABILITY}$ problem, which is **NL**-hard: Given a directed acyclic graph $G = (V, E)$ and $s, t \in V$, is there a directed path from s to t in G?

We construct an instance **db** for $\mathsf{CERTAINTY}(q)$ as follows. First, we introduce two new constants s' and t'. Then:

- for each $u \in V \cup \{s'\}$, introduce $\langle q \setminus q^x_\Delta, [x \to u]\rangle$;

- for every edge $(u, v) \in E \cup \{(s', s), (t, t')\}$, introduce $\langle q^x_\triangle \setminus q^y_\triangle, [x, y \to u, v]\rangle$;
- for every vertex $u \in V$, introduce $\langle q^y_\triangle, [y \to u]\rangle$.

Note that the above construction guarantees that only R has inconsistent blocks.

We now argue that there is a directed path $(u_1, u_2, \dots, u_k)$ with $(u_i, u_{i+1}) \in E$, $u_1 = s$ and $u_k = t$ in G if and only if there is a repair of $\mathbf{db}$ that does not satisfy q.

$\boxed{\Longrightarrow}$ Assume that there exists a directed path $(u_1, u_2, \dots, u_k)$ with $(u_i, u_{i+1}) \in E$, $u_1 = s$ and $u_k = t$ in G. Denote $u_0 = s'$ and $u_{k+1} = t'$. Let $\mathbf{r}$ be the repair that picks $\langle q^x_\triangle \setminus q^y_\triangle, [x, y \to u_i, u_{i+1}]\rangle$ for every $1 \leq i \leq k-1$, and $\langle q^y_\triangle, [y \to u]\rangle$ for any other vertex u. Suppose for contradiction that $\mathbf{r}$ satisfies q with a valuation θ. It is not possible that $\theta(q) \subseteq \langle q^y_\triangle, [y \to u]\rangle$ for any $u \notin V$ since the size does not fit.

We argue that we must have $\theta(x) = u_i$ and $\theta(y) = u_{i+1}$ for some $0 \leq i < k$. If $\theta(x) = u_i \in \{u_0, u_1, \dots, u_k\}$, then we must have $\theta(y) = u_{i+1}$ since $\langle q^x_\triangle \setminus q^y_\triangle, [x, y \to u, v]\rangle$ is a canonical copy. Suppose for contradiction that $\theta(x) \notin \{u_0, u_1, \dots, u_k\}$. It is not possible that $\theta(x) = u_{k+1} = t'$ since by construction, there is no rooted tree set rooted at t'. Note that there is no atom $R(\underline{z}, \dots)$ such that $z \notin x, y$, $x <_q z <_q y$. Hence $\theta(x)$ cannot fall on the path connecting any u_i and u_{i+1}, and $\theta(q^x_\triangle)$ must be contained in some $\langle q^x_\triangle \setminus q^y_\triangle, [x, y \to u_i, u_{i+1}]\rangle$. Then, there must be an atom $R(\underline{z}, \dots)$ such that (i) $x <_q z$, (ii) $z \parallel_q y$ and (iii) $\theta(q^x_\triangle)$ is contained in $\langle q^z_\triangle, [z \to \theta(x)]\rangle$, which is impossible since the sizes do not fit.

By construction, there is a canonical copy of $q^y_\triangle$ rooted at u_{i+1}. If this canonical copy is contained in $\langle q^x_\triangle \setminus q^y_\triangle, [x, y \to u_{i+1}, u_{i+2}]\rangle$, then there is a root homomorphism from $q^y_\triangle$ to $q^x_\triangle \setminus q^y_\triangle$, and so from $q^y_\triangle$ to $q^x_\triangle$, a contradiction. Otherwise, there exists some atom $R(\underline{z}, \dots)$ such that *(i)* $y <_q z$ and *(ii)* $q^y_\triangle \setminus q^z_\triangle$ has a root homomorphism to $q^x_\triangle \setminus q^y_\triangle$. Recall now that from our initial assumption we must have that $q^z_\triangle \leq_{z \to y} q^y_\triangle$. This implies that we can now generate a root homomorphism from $q^y_\triangle$ to $q^x_\triangle$, a contradiction.

$\boxed{\Longleftarrow}$ Assume that there is no directed path from s to t in G. Consider any repair $\mathbf{r}$ of $\mathbf{db}$. Since G is acyclic, there exists a maximal sequence $u_0, u_1, \dots, u_k$ with $k \geq 1$ such that $u_0 = s'$, $u_1 = s$, $\langle q^x_\triangle \setminus q^y_\triangle, [x, y \to u_i, u_{i+1}]\rangle \subseteq \mathbf{r}$ for $0 \leq i < k$ and $\langle q^y_\triangle, [y \to u_k]\rangle \subseteq \mathbf{r}$. Then, the following set of facts satisfies q:

$$\begin{aligned}
&\langle q \setminus q^x_\triangle, [x \to u_{k-1}]\rangle \cup \\
&\langle q^x_\triangle \setminus q^y_\triangle, [x, y \to u_{k-1}, u_k]\rangle \cup \\
&\langle q^y_\triangle, [y \to u_k]\rangle.
\end{aligned}$$

This shows that $\mathsf{CERTAINTY}(q)$ is $\mathbf{NL}$-hard since $\mathbf{NL}$ is closed under complement. $\square$

Lemma 5.16. *$\mathsf{CERTAINTY}(q)$ is $\mathbf{NL}$-hard for each q in $\mathsf{TreeBCQ}$ that violates $\mathcal{C}_1$.*

Proof. Assume that q violates $\mathcal{C}_1$. Then there exist two distinct atoms $R(\underline{x}, \dots)$ and $R(\underline{y}, \dots)$ in q such that there is no root homomorphism from $q^y_\triangle$ to $q^x_\triangle$ or from $q^x_\triangle$ to $q^y_\triangle$. If $x \parallel_q y$, Lemma 5.1 we implies that $\mathcal{C}_2$ is also violated, so $\mathsf{CERTAINTY}(q)$ is **coNP**-hard from Lemma 5.14. Otherwise, $\mathsf{CERTAINTY}(q)$ is **NL**-hard by Lemma 5.15. □

Proposition 5.4. *For every q in* TreeBCQ*,*

1. *if q violates $\mathcal{C}_2$, then* $\mathsf{CERTAINTY}(q)$ *is* **coNP***-hard; and*
2. *if q violates $\mathcal{C}_1$, then* $\mathsf{CERTAINTY}(q)$ *is* **NL***-hard.*

Proof. Immediate from Lemma 5.14 and 5.16. □

5.7 Extending the Trichotomy

In this section, we extend the complexity classification for rooted tree queries to larger classes of Boolean conjunctive queries.

Lemma 5.17. *Let q be a minimal query in* BCQ *with connected components $q_1, q_2, \dots, q_n$. Then:*

1. *for every $1 \leq i \leq n$, there exists a first order reduction from the problem* $\mathsf{CERTAINTY}(q_i)$ *to* $\mathsf{CERTAINTY}(q)$*; and*
2. *for every database instance* **db***,* **db** *is a "yes"-instance of the problem* $\mathsf{CERTAINTY}(q)$ *if and only if for every $1 \leq i \leq n$,* **db** *is a "yes"-instance of* $\mathsf{CERTAINTY}(q_i)$*.*

Proof. For item (1), let $\mathbf{db}$ be an instance for $\mathsf{CERTAINTY}(q_i)$ and construct an instance $\mathbf{db}' = \mathbf{db} \cup \bigcup_{j \neq i} D[q_j]$ for $\mathsf{CERTAINTY}(q)$, where $D[q_j]$ is a canonical copy of q_j. Clearly, $\mathbf{db}'$ can be constructed in **FO**. Next, we show that $\mathsf{CERTAINTY}(q_i)$ is true on $\mathbf{db}$ if and only if $\mathsf{CERTAINTY}(q)$ is true on $\mathbf{db}'$.

Assume that $\mathbf{db}$ is a "yes"-instance for $\mathsf{CERTAINTY}(q_i)$. Let $\mathbf{r}'$ be any repair of $\mathbf{db}'$. We have $\bigcup_{j \neq i} D[q_j] \subseteq \mathbf{r}'$, since each $D[q_j]$ is consistent. Thus $\mathbf{r} = \mathbf{r}' \setminus \bigcup_{j \neq i} D[q_j]$ is a repair of $\mathbf{db}$, and we have that $\mathbf{r}$ satisfies q_i. Then $\mathbf{r}'$ satisfies q, since $\mathbf{r}$ also contains $D[q_j]$, which satisfies q_j for $j \neq i$.

Assume that $\mathbf{db}'$ is a "yes"-instance for $\mathsf{CERTAINTY}(q)$. Let $\mathbf{r}$ be any repair of $\mathbf{db}$. It remains to show that $\mathbf{r}$ satisfy q_i. Let $\mathbf{r}' = \mathbf{r} \cup \bigcup_{j \neq i} D[q_j]$. Hence $\mathbf{r}'$ is a repair of $\mathbf{db}'$, and and there is a valuation μ such that $\mu(q) \subseteq \mathbf{r}'$. It suffices to show that $\mu(q_i) \subseteq \mathbf{r}$. Suppose not, since each q_i is connected and each $D[q_j]$ is connected, we must have $\mu(q_i) \subseteq D[q_j]$ for some $j \neq i$, which implies a homomorphism from q_i to q_j, contradicting that q is minimal. Thus $\mu(q_i) \subseteq \mathbf{r}$, as desired.

Item (2) is proved in Lemma B.1 in [KOW21]. □

Lemma 5.17 implies that the complexity of $\mathsf{CERTAINTY}(q)$ is equal to the highest complexity of $\mathsf{CERTAINTY}(q')$ over every connected component q' of q.

Attacks. Let q be a self-join-free Boolean CQ. For every atom $F \in q$, we define $F^{+,q}$ as the set of all variables in q that are functionally determined by $\mathsf{key}(F)$ with respect to all functional dependencies of the form $\mathsf{key}(G) \rightarrow \mathsf{vars}(G)$ with $G \in q \setminus \{F\}$. Following [KW15], the *attack graph* of q is a directed graph whose vertices are the atoms of q. There is a directed edge, called *attack*, from F to G ($F \neq G$), written $F \overset{q}{\rightsquigarrow} G$, if there exists a path between F and G in the query such that every two adjacent atoms share a variable not in $F^{+,q}$. The attack is called *weak* if $\mathsf{key}(F) \rightarrow \mathsf{key}(G)$, otherwise it is called *strong*. It was proved in [KW15] that for a self-join-free Boolean CQ q, $\mathsf{CERTAINTY}(q)$ is **coNP**-hard if and only if there exist two atoms $F \neq G$ that attack each other and at least one of the attacks is strong.

We can now prove the proposition.

Proposition 5.5. *If q is a connected minimal conjunctive query in* $\mathsf{GraphBCQ} \setminus \mathsf{TreeBCQ}$*, then* $\mathsf{CERTAINTY}(q)$ *is* **L***-hard (and not in* **FO***); if q is also Berge-acyclic, then* $\mathsf{CERTAINTY}(q)$ *is* **coNP***-hard.*

Proof. Let q be a minimal connected query in $\mathsf{GraphBCQ}$.

Assume that q is not a rooted tree query. Then, there exist two atoms $R(\underline{x}, \dots, z, \dots)$ and $S(\underline{y}, \dots, z, \dots)$ with $x \neq y$ (and possibly $R = S$) in q. Consider now q^{sjf}, and let R_0 and S_0 be the corresponding atoms of R and S in q^{sjf}.

Since q satisfies (1) and (3), so does q^{sjf}, and we have $R_0^{+,q^{\mathsf{sjf}}} = \{x\}$ and $S_0^{+,q^{\mathsf{sjf}}} = \{y\}$. Hence $R_0 \overset{q^{\mathsf{sjf}}}{\rightsquigarrow} S_0$, and similarly, $S_0 \overset{q^{\mathsf{sjf}}}{\rightsquigarrow} R_0$. By [KW15] $\mathsf{CERTAINTY}(q^{\mathsf{sjf}})$ is **L**-hard (due to this cycle in the attack graph of q^{sjf}), and so is $\mathsf{CERTAINTY}(q)$ by Lemma 5.18.

Next we additionally assume that q is Berge-acyclic, that is, $q \in \mathsf{Graph_{Berge}BCQ}$. We thus have either $x \not\lessdot_q y$ or $y \not\lessdot_q x$. Indeed, otherwise there exist atoms $R_0, R_1, \dots, R_n$ and $S_0, S_1, \dots, S_m$ and variables $x_0, x_1, x_2, \dots, x_{n+1}$, $y_0, y_1, \dots, y_{m+1}$ in q^{sjf} where $x_0 = x$, $x_{n+1} = y$, $y_0 = y$, $y_{m+1} = x$ such that q^{sjf} contains atoms $R_i(\underline{x_i}, \dots, x_{i+1}, \dots)$ for every $0 \leq i \leq n$ and $S_i(\underline{y_i}, y_{i+1})$ for every $0 \leq i \leq m$. Then,

$$\{x_0, R_0, x_1, R_1, \dots, R_n, x_{n+1}(= y = y_0), S_0, y_1, S_1, \dots, S_m, y_{m+1}(= x = x_0)\}$$

is a Berge-cycle in q^{sjf}, a contradiction to that q^{sjf} (and q) are Berge-acyclic. This implies that at least one of the two attacks is strong. Hence, applying the result from [KW15], $\mathsf{CERTAINTY}(q^{\mathsf{sjf}})$ is **coNP**-hard (due to this strong cycle in the attack graph of q^{sjf}), and so is $\mathsf{CERTAINTY}(q)$ by Lemma 5.18. $\square$

Proof of Theorems 5.2 and 5.3. Let q be a query in $\mathsf{GraphBCQ}$. Then the minimal query q^* of q is also in $\mathsf{GraphBCQ}$. If every connected component of q^* is in $\mathsf{TreeBCQ}$ and satisfies $\mathcal{C}_1$, then $\mathsf{CERTAINTY}(q)$ is in **FO**. Otherwise, there exists some connected component q' of q^* that is either not in $\mathsf{TreeBCQ}$, or violates $\mathcal{C}_1$, and $\mathsf{CERTAINTY}(q)$ is **L**-hard or **NL**-hard by Lemma 5.17, Proposition 5.5, and Theorem 5.4. Assume that q is also Berge-acyclic. If some connected component q' of q^* is not in $\mathsf{TreeBCQ}$, then $\mathsf{CERTAINTY}(q)$ is **coNP**-complete; or otherwise, $\mathsf{CERTAINTY}(q)$ exhibits a trichotomy by Theorem 5.4. $\square$

Lemma 5.18 (adapted from [Wij19a]) is essential to the proof of Proposition 5.5, but is of independent interest. It relates the complexity of CQA on queries with self-joins to that on self-join-free queries.

Given a query q in BCQ, a *self-join-free version of* q, denoted q^{sjf}, is any self-join-free Boolean conjunctive query obtained from q by (only) renaming relation names. For example, a self-join-free version of $\{R(\underline{x}, y), R(\underline{y}, x)\}$ is $\{R(\underline{x}, y), S(\underline{y}, x)\}$.

Lemma 5.18 (Bridging Lemma)**.** *Let q be a minimal query in* BCQ *and $\mathcal{C}$ a complexity class. If* $\mathsf{CERTAINTY}(q^{\mathsf{sjf}})$ *is $\mathcal{C}$-hard, then* $\mathsf{CERTAINTY}(q)$ *is $\mathcal{C}$-hard.*

Proof. For each atom $N(\underline{\vec{x}}, \vec{y})$ in q^{sjf}, we denote $\pi(N) = R$ if $R(\underline{\vec{x}}, \vec{y})$ in q.

We present a reduction from $\mathsf{CERTAINTY}(q^{\mathsf{sjf}})$ to $\mathsf{CERTAINTY}(q)$ in **FO**.

Let $\mathbf{db}^{\mathsf{sjf}}$ be an instance for $\mathsf{CERTAINTY}(q^{\mathsf{sjf}})$ and $N(\alpha_1, \alpha_2, \ldots, \alpha_n)$ an atom in q^{sjf}. Consider a mapping ϕ from facts to facts that for any fact $f = N(f_1, f_2, \ldots, f_n)$ in $\mathbf{db}^{\mathsf{sjf}}$,

$$\phi(f) = \pi(N)(\langle f_1, \alpha_1\rangle, \langle f_2, \alpha_2\rangle, \ldots, \langle f_n, \alpha_n\rangle)$$

where each $\langle f_i, \alpha_i\rangle$ is a fresh constant such that $\langle f_i, \alpha_i\rangle = \langle f_j, \alpha_j\rangle$ if and only if $f_i = f_j$ and $\alpha_i = \alpha_j$. Let $\mathbf{db} = \{\phi(f) \mid f \in \mathbf{db}^{\mathsf{sjf}}\}$.

We first show that ϕ is bijective from $\mathbf{db}^{\mathsf{sjf}}$ to $\mathbf{db}$. By construction, ϕ is onto. Suppose ϕ is not injective, then there exist two distinct facts $f = R(f_1, f_2, \ldots, f_n)$ and $g = S(g_1, g_2, \ldots, g_m)$ from atoms $R(\alpha_1, \alpha_2, \ldots, \alpha_n)$ and $S(\beta_1, \beta_2, \ldots, \beta_m)$ in q^{sjf} such that $\phi(f) = \phi(g)$. We then have $m = n$, $\pi(R) = \pi(S)$ and for each $1 \leq i \leq n$, $\langle f_i, \alpha_i\rangle = \langle g_i, \beta_i\rangle$, implying that $\pi(R)(\alpha_1, \alpha_2, \ldots, \alpha_n) = \pi(S)(\beta_1, \beta_2, \ldots, \beta_m)$ in q, but q is minimal, a contradiction.

We show that $\mathbf{db}^{\mathsf{sjf}}$ is a "yes"-instance for $\mathsf{CERTAINTY}(q^{\mathsf{sjf}})$ if and only if $\mathbf{db}$ is a "yes"-instance for $\mathsf{CERTAINTY}(q)$. Towards this end, let $\mathbf{r}^{\mathsf{sjf}}$ be a repair of $\mathbf{db}^{\mathsf{sjf}}$, and consider the set $\mathbf{r} = \{\phi(f) \mid f \in \mathbf{r}^{\mathsf{sjf}}\}$. It is easy to see that $\mathbf{r}$ is a repair of $\mathbf{db}$. Hence it is sufficent to show that $\mathbf{r}^{\mathsf{sjf}}$ satisfies q^{sjf} if and only if $\mathbf{r}$ satisfies q.

Therefore, $\mathbf{r}^{\mathsf{sjf}}$ satisfies q^{sjf} if and only if there exists a valuation μ such that $\mu(q^{\mathsf{sjf}}) \subseteq \mathbf{r}^{\mathsf{sjf}}$. That is, for every fact $f = R(\mu(\alpha_1), \mu(\alpha_2), \ldots, \mu(\alpha_n))$ in $\mathbf{r}^{\mathsf{sjf}}$, $\phi(f) \in \mathbf{r}$. Since ϕ is bijective, this is equivalent to $\phi(\mu(q)) \subseteq \mathbf{r}$. This shows that $\mathbf{r}$ satisfies q. The other direction follows similarly since ϕ is bijective. $\square$

Example 5.4. For $q_1 = \{R(\underline{x}, y, z), R(\underline{z}, x, y)\}$, we have ${q_1}^{\mathsf{sjf}} = \{R_1(\underline{x}, y, z), R_2(\underline{z}, x, y)\}$. By Theorem 1.1 [KW15], $\mathsf{CERTAINTY}({q_1}^{\mathsf{sjf}})$ is **L**-complete, and thus $\mathsf{CERTAINTY}(q_1)$ is **L**-hard by Lemma 5.18.

For $q_2 = \{R(\underline{x}, z), R(\underline{y}, z)\}$, we have ${q_2}^{\mathsf{sjf}} = \{R_1(\underline{x}, z), R_2(\underline{y}, z)\}$. By Theorem 1.1 [KW15], $\mathsf{CERTAINTY}({q_2}^{\mathsf{sjf}})$ is **coNP**-complete. In fact, $\mathsf{CERTAINTY}(q_2)$ is in **FO** because $q_2 \equiv q_2'$ where $q_2' = \{R(\underline{x}, z)\}$. Lemma 5.18 does not apply here because q_2 is not minimal.

5.8 Open Problems

So far, we have established the **FO**-boundary of $\mathsf{CERTAINTY}(q)$ for all queries q in $\mathsf{GraphBCQ}$, and a fine-grained complexity classification for all Berge-acyclic queries in $\mathsf{GraphBCQ}$, which include all rooted tree queries. We briefly discuss the remaining syntactic restrictions and the challenges in extending Theorem 5.3 beyond $\mathsf{Graph}_{\mathsf{Berge}}\mathsf{BCQ}$.

The complexity classification of $\mathsf{CERTAINTY}(q)$ for queries q in $\mathsf{GraphBCQ}$ that are not Berge-acyclic is likely to impose new challenges. In particular, Figueira et al. [FPSS23] showed that for q_1 in Example 5.4 (that is not Berge-acyclic), the complement of $\mathsf{CERTAINTY}(q_1)$ is complete for $\mathsf{Bipartite\ Matching}$ under LOGSPACE-reductions.

The restriction imposed by $\mathsf{GraphBCQ}$ that every variable occurs at most once at a primary-key position allows for an elegant graph representation. We found that dropping this requirement imposes serious challenges. The following Proposition 5.6 hints at the difficulty of having to "correlate two rooted tree branches" that share the same primary-key variable.

Proposition 5.6. *Consider the following queries:*

- $q_1 = \{R(\underline{u}, x_1), R(\underline{x_1}, x_2), S(\underline{u}, y_1), S(\underline{y_1}, y_2)\}$;
- $q_2 = q_1 \cup \{X(\underline{x_2}, x_3)\}$; *and*
- $q_3 = q_1 \cup \{X(\underline{x_2}, x_3), Y(\underline{y_2}, y_3)\}$.

Then it follows that $\mathsf{CERTAINTY}(q_1)$ *is in* **FO**, $\mathsf{CERTAINTY}(q_2)$ *is in* **NL**-*hard* $\cap$ **LFPL**, *and* $\mathsf{CERTAINTY}(q_3)$ *is* **coNP**-*complete.*

We define two first-order formula $\varphi_R(u)$ and $\varphi_S(u)$:

$$\varphi_R(u) = \exists y R(\underline{u}, y) \wedge \forall y \left(R(\underline{u}, y) \rightarrow \exists z R(\underline{y}, z)\right)$$
$$\varphi_S(u) = \exists y S(\underline{u}, y) \wedge \forall y \left(S(\underline{u}, y) \rightarrow \exists z S(\underline{y}, z)\right)$$

Lemma 5.19 ([KOW21])**.** *For $q = R(\underline{c}, y), R(\underline{y}, z)$, the* **FO***-rewriting of the query* $\mathsf{CERTAINTY}(q)$ *is* $\varphi_R(c)$.

Lemma 5.20. *Let* $q = \{R(\underline{u}, x_1), R(\underline{x_1}, x_2), S(\underline{u}, y_1), S(\underline{y_1}, y_2)\}$*. Then* $\mathsf{CERTAINTY}(q)$ *is in* **FO**.

Proof. Let **db** be a database instance. We show that **db** is a "yes"-instance for $\mathsf{CERTAINTY}(q)$ if and only if **db** satisfies the formula

$$\exists c : \varphi_R(c) \wedge \varphi_S(c).$$

$\boxed{\Longleftarrow}$ This direction is straightforward. Let **r** be any repair of **db**. By Lemma 5.19, **r** contains a path of RR starting in c and a path SS starting in c. Hence **r** satisfies q.

$\boxed{\Longrightarrow}$ Assume that **db** does not satisfy **r**. We construct a falsifying repair **r** of **db** as follows: For each constant $c \in \mathsf{adom}(\mathbf{db})$:

- if $R(\underline{c}, *)$ is empty and $S(\underline{c}, *)$ is nonempty, pick an arbitrary fact from $S(\underline{c}, *)$;
- if $R(\underline{c}, *)$ is nonempty and $S(\underline{c}, *)$ is empty, pick an arbitrary fact from $R(\underline{c}, *)$;
- Assume that $R(\underline{c}, *)$ and $S(\underline{c}, *)$ are nonempty. If $\varphi_R(c)$ is false, we pick $R(\underline{c}, d_1)$ such that $R(\underline{d_1}, *)$ is empty; or otherwise $\varphi_S(c)$ is false, we pick $S(\underline{c}, d_2)$ such that $S(\underline{d_2}, *)$ is empty.

We argue that **r** is a falsifying repair. Suppose for contradiction that **r** satisfies q and u is mapped to c. Assume that $R(\underline{c}, d_1), S(\underline{c}, d_2) \in \mathbf{r}$. If $\varphi_R(c)$ is false, then we would have picked a fact $R(\underline{c}, d_1)$ such that $R(\underline{d_1}, *)$ is empty, a contradiction, and so is the other case where $\varphi_S(c)$ is false. □

Lemma 5.21. *Let* $q = \{R(\underline{u}, x_1), R(\underline{x_1}, x_2), X(\underline{x_2}, x_3), S(\underline{u}, y_1), S(\underline{y_1}, y_2)\}$*. Then* $\mathsf{CERTAINTY}(q)$ *is in* **NL***-hard and in* **LFPL**.

Proof. The **NL**-hardness proof follows by modifying the proof of Lemma 7.1 in [KOW21] to also add a copy of SS path starting in every vertex $v \in \{s'\} \cup V$.

Let **db** be a database instance. We revise the algorithm in Figure 5.6 for $\mathsf{CERTAINTY}(RRX)$ as follows:

- Initialize $N = \{\langle c, x_2\rangle \mid X(\underline{c}, *) \neq \emptyset\}$
- while N is not fixed:
- add $\langle c, x_1\rangle$ to N if

$$\exists d R(\underline{c}, d) \wedge \forall d \left(R(\underline{c}, d) \rightarrow f_2(R(\underline{c}, d), R(\underline{x_1}, x_2))\right),$$

where

$$f_2(R(\underline{c}, d), R(\underline{x_1}, x_2)) = \langle d, x_2 \rangle \in N \vee \underbrace{(\varphi_S(c) \wedge f_1(R(\underline{c}, d), R(\underline{u}, x_1)))}_{\text{rewinding to } R(\underline{u}, x_1) \text{ allowed only when } \varphi_S(c) \text{ is true}}$$

- add $\langle c, u \rangle$ to N if

$$\varphi_S(c) \wedge \exists d R(\underline{c}, d) \wedge \forall d \left(R(\underline{c}, d) \rightarrow f_1(R(\underline{c}, d), R(\underline{u}, x_1))\right),$$

where

$$f_1(R(\underline{c}, d), R(\underline{u}, x_1)) = \langle d, x_1 \rangle \in N.$$

Our algorithm essentially first computes the set N and then checks $\exists c : \langle c, u \rangle \in N$. Notice that this algorithm is in **LFPL**. To show correctness, we argue that there exists a constant c such that $\langle c, u \rangle \in N$ if and only if **db** is a "yes"-instance for $\mathsf{CERTAINTY}(q)$.

To this end, we need to define a refined notion of frugal repairs that take into account of $\varphi_S(u)$ constructed as follows:

- pick an arbitrary fact from every nonempty block $X(\underline{c}, *)$;
- for every nonempty block $S(\underline{c}, *)$, if $\varphi_S(c)$ is true, pick an arbitrary fact; or otherwise, pick $S(\underline{c}, d)$ such that $S(\underline{d}, *)$ is empty; and
- for every fact $R(\underline{c}, d)$ in the block $R(\underline{c}, *)$, we define the frugal index of $R(\underline{c}, d)$ to be
  - 0, if $\varphi_S(c) \wedge f_1(R(\underline{c}, d), R(\underline{u}, x_1))$ is true;
  - 1, if $\varphi_S(c) \wedge f_1(R(\underline{c}, d), R(\underline{u}, x_1))$ is false and $f_2(R(\underline{c}, d), R(\underline{x_1}, x_2))$ is true; or
  - 2, otherwise.

We then pick the fact $R(\underline{c}, d)$ from each nonempty block $R(\underline{c}, *)$ with the largest frugal index.

Definition 5.9. An extended RRX-path in a repair $\mathbf{r}$ is a sequence of facts

$$R(\underline{c_0}, c_1), R(\underline{c_1}, c_2), \ldots, R(\underline{c_{n-1}}, c_n), X(\underline{c_n}, c_{n+1})$$

in $\mathbf{r}$ for some $n \geq 2$ such that for every $0 \leq i \leq n-2$, there exists an SS-path in $\mathbf{r}$ starting in c_i.

The following claim concludes the proof.

Claim 5.1. The following statements are equivalent:

1. $\langle c, u \rangle \in N$;
2. there exists an extended RRX-path in $\mathbf{r}^*$ starting in c; and

3. for every repair $\mathbf{r}$ in $\mathbf{db}$, there exists an extended RRX-path in $\mathbf{r}$ starting in c.

Proof. $\boxed{(3) \implies (2)}$ Straightforward. $\boxed{(2) \implies (1)}$ Let $R(\underline{c_0}, c_1)$, $R(\underline{c_1}, c_2)$, $\ldots$, $R(\underline{c_{n-1}}, c_n)$, $X(\underline{c_n}, c_{n+1})$ be an extended RRX-path in $\mathbf{r}^*$ for some $n \geq 2$ and $c_0 = c$. Since for every $0 \leq i \leq n-2$, there exists an SS-path in $\mathbf{r}^*$ starting in c_i, by construction of $\mathbf{r}^*$, we have that $\varphi_S(c_i)$ is true for every $0 \leq i \leq n-2$.

We have that $\langle c_n, x_2\rangle \in N$. Then for $R(\underline{c_{n-1}}, *)$, we have that $f_2(R(\underline{c_{n-1}}, c_n), R(\underline{x_1}, x_2))$ is true. Then by the choice of the frugal repair $\mathbf{r}^*$, the frugal index of $R(\underline{c_{n-1}}, c_n)$ is at most 1, and thus $f_2(R(\underline{c_{n-1}}, c'_n), R(\underline{x_1}, x_2))$ is true for every fact $R(\underline{c_{n-1}}, c'_n)$ in the block $R(\underline{c_{n-1}}, *)$. Hence $\langle c_{n-1}, x_1\rangle \in N$.

Then, we must have $f_1(R(\underline{c_{n-2}}, c_{n-1}, R(\underline{u}, x_1))$ is true, because $\langle c_{n-1}, x_1\rangle \in N$. Notice that $\varphi_S(c_{n-2})$ is true, the frugal index of $R(\underline{c_{n-2}}, c_{n-1})$ is 0. Then by construction of $\mathbf{r}^*$, for every fact $R(\underline{c_{n-2}}, c'_{n-1})$ in block $R(\underline{c_{n-2}}, *)$, $f_1(R(\underline{c_{n-2}}, c_{n-1}, R(\underline{u}, x_1))$ is true. Hence $\langle c_{n-2}, u\rangle \in N$. Additionally, for every fact $R(\underline{c_{n-2}}, c'_{n-1})$ in block $R(\underline{c_{n-2}}, *)$, $\varphi_S(c_{n-2}) \wedge f_1(R(\underline{c_{n-2}}, c_{n-1}, R(\underline{u}, x_1))$ is true, and thus $f_2(R(\underline{c_{n-2}}, c'_{n-1}, R(\underline{x_1}, x_2))$ is also true. This gives $\langle c_{n-2}, x_1\rangle \in N$.

This argument may continue, until we yield that $\langle c_0, u\rangle = \langle c, u\rangle \in N$.

$\boxed{(1) \implies (3)}$ Assume that $\langle c, u\rangle$ in N. Let $\mathbf{r}$ be any repair of $\mathbf{db}$. We inductively construct an extended RRX-path in $\mathbf{r}$ starting in c. Assume that $R(\underline{c_0}, c_1) \in \mathbf{r}$ with $c_0 = c$. Hence $\varphi_S(c_0)$ is true, and that $\langle c_1, x_1\rangle \in N$. Let $R(\underline{c_1}, c_2) \in \mathbf{r}$, and we have $f_2(R(\underline{c_1}, c_2), R(\underline{x_1}, x_2))$ is true. If $\langle c_2, x_2\rangle \in N$, let $X(\underline{c_2}, c_3) \in \mathbf{r}$ and then the proof is complete since $R(\underline{c_0}, c_1), R(\underline{c_1}, c_2), X(\underline{c_2}, c_3)$ is an extended RRX-path in $\mathbf{r}$. Otherwise, we have that $\varphi_S(c_1) \wedge f_1(R(\underline{c_1}, c_2), R(\underline{u}, x_1))$ is true. Therefore, $\varphi_S(c_1)$ is true and $\langle c_2, x_1\rangle \in N$. This process thus continues, until we have produced facts $R(\underline{c_0}, c_1), R(\underline{c_1}, c_2), \ldots, R(\underline{c_{n-1}}, c_n)$, such that $\varphi_S(c_i)$ is true for $0 \leq i \leq n-2$ and $\langle c_n, x_2\rangle \in N$, which gives a fact $X(\underline{c_n}, c_{n+1}) \in \mathbf{r}$. Hence there exists an extended RRX-path in $\mathbf{r}$ starting in $c_0 = c$. □

Now, assume that $\langle c, u\rangle \in N$. Then by Claim 5.1, every repair $\mathbf{r}$ of $\mathbf{db}$ contains an extended RRX-path starting in c, which satisfies q. Assume that $\langle c, u\rangle \notin N$. Then by Claim 5.1, there is no extended RRX-path in $\mathbf{r}^*$, and thus $\mathbf{r}^*$ does not satisfy q. □

Lemma 5.22. *Let* $q = \{R(\underline{u}, x_1), R(\underline{x_1}, x_2), X(\underline{x_2}, x_3), S(\underline{u}, y_1), S(\underline{y_1}, y_2), Y(\underline{y_2}, y_3)\}$. *Then the problem* $\mathsf{CERTAINTY}(q)$ *is* **coNP***-complete.*

Proof. For **coNP**-hardness, we present a reduction from Monotone SAT: given a monotone CNF formula φ, does φ have a satisfying assignment?

Let φ be a monotone CNF formula. We construct a database $\mathbf{db}$ for $\mathsf{CERTAINTY}(q)$ as follows:

- for each variable z in φ, introduce a copy of

$$z^- = \langle\{R(\underline{u}, x_1), R(\underline{x_1}, x_2), X(\underline{x_2}, x_3), S(\underline{u}, y_1), Y(\underline{y_1}, y_2)\}, [u \to z]\rangle$$

  and a copy of

$$z^+ = \langle\{R(\underline{u}, x_1), X(\underline{x_1}, x_2), S(\underline{u}, y_1), S(\underline{y_1}, y_2), Y(\underline{y_2}, y_3)\}, [u \to z]\rangle;$$

- for each positive literal z in a positive clause C, introduce a copy of

$$C_z = \langle\{R(\underline{u}, x_1), R(\underline{x_1}, x_2), X(\underline{x_2}, x_3), S(\underline{u}, y_1)\}, [u, y_1 \to C, z]\rangle,$$

- for each negative literal $\overline{z}$ in a negative clause $\overline{C}$, introduce a copy of

$$\overline{C}_z = \langle\{R(\underline{u}, x_1), S(\underline{u}, y_1), S(\underline{y_1}, y_2), Y(\underline{y_2}, y_3)\}, [u, x_1 \to \overline{C}, z]\rangle.$$

Note that by construction, the only inconsistencies happen at each block $R(\underline{z}, *)$ and $S(\underline{z}, *)$, because each R-block may choose either an R-edge or an RX-edge; and each S-block may choose either an S-edge or an SY-edge. Surprisingly, there are 4 possible repairs for each variable z, but it is sufficient to encode a Boolean choice between $z = 0$ and $z = 1$. An example gadget is shown in Figure 5.7.

We show that φ has a satisfying assignment if and only if there is a repair of **db** that does not satisfy q.

$\boxed{\implies}$ Assume that σ is a satisfying assignment to φ. We now construct a repair $\mathbf{r}$ of **db** as follows. For each variable z, if $\sigma(z) = 1$, then we pick $z^+ \subseteq \mathbf{r}$, or otherwise we pick $z^- \subseteq \mathbf{r}$; and for each positive clause C, there must be some literal $z \in C$ with $\sigma(z) = 1$, we pick $C_z \subseteq \mathbf{r}$; and for each negative clause $\overline{C}$, there must be literal $\overline{z} \in \overline{C}$ with $\sigma(z) = 0$, we pick $\overline{C}_z \subseteq \mathbf{r}$.

We argue that $\mathbf{r}$ does not satisfy q. Indeed, by construction, for each clause C, we have $C_z \subseteq \mathbf{r}$, but for the variable z, we picked $z^+ \subseteq \mathbf{r}$, and they cannot satisfy the query q. Similarly, every negative clause $\overline{C}$ cannot satisfy q. Each variables are picked so that they also do not satisfy q. We conclude that $\mathbf{r}$ does not satisfy q.

$\boxed{\impliedby}$ Assume that $\mathbf{r}$ is a repair of **db** that does not satisfy q. Consider the assignment σ that assigns $\sigma(z) = 1$, for every variable z with $z^+ \subseteq \mathbf{r}$, and assigns $\sigma(z) = 0$ for every $z^- \subseteq \mathbf{r}$, and assign all other variables arbitrarily.

We now argue that σ is a satisfying assignment. Let C be any positive clause and assume $C_z \subseteq \mathbf{r}$. Then the repair must choose the path SSY starting in z (or otherwise $\mathbf{r}$ satisfies q rooted at the clause constant C), and consequently the path RX starting in z (or otherwise $\mathbf{r}$ satisfies q rooted at the variable constant z). Hence $z^+ \subseteq \mathbf{r}$, and we set $\sigma(z) = 1$. Hence every positive clause

C is satisfied. Let $\overline{C}$ be any positive clause and assume $\overline{C}_z \subseteq \mathbf{r}$. Then the repair must similarly choose the path RRX and SY starting in z, or otherwise $\mathbf{r}$ satisfies q. Then $z^- \subseteq \mathbf{r}$, and we set $\sigma(z) = 0$. Hence every negative clause $\overline{C}$ is satisfied. $\square$

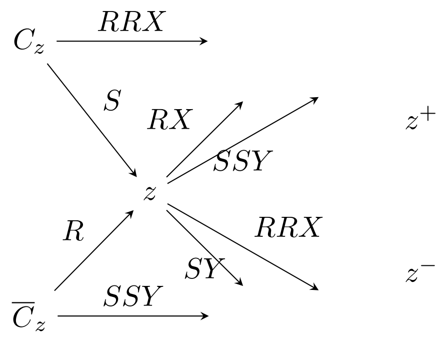


Figure 5.7: The gadget used in Lemma 5.22

The restrictions that no atom contains repeated variables, and that no constant occurs at a primary-key position ease the technical treatment, but it is likely that they can be dropped at the price of some technical involvement. On the other hand, all our techniques fundamentally rely on the restriction that primary keys are simple.

5.9 Conclusion

We established a complexity classification in consistent query answering for primary keys on rooted tree queries that can have self-joins: for every rooted tree query q, the problem $\mathsf{CERTAINTY}(q)$ is in **FO**, **LFPL** $\cap$ **NL**-hard, or **coNP**-complete, and it is decidable in polynomial time in the size of q which of the three cases applies. With some minor effort, this complexity classification can be lifted to DAG queries.

For singleton primary keys, an intriguing open problem is to generalize the form of the queries, from rooted trees to a more general class of conjunctive query. The ultimate open problem is Conjecture 1.1, which conjectures that for every Boolean conjunctive query q, $\mathsf{CERTAINTY}(q)$ is either in **PTIME** or **coNP**-complete.

Chapter 6

Future Directions and Preliminary Results

"Wir müssen wissen. Wir werden wissen."
—David Hilbert

In this Chapter, we identify some open challenges in additional to the possible future work already discussed in previous chapters.

6.1 CQA Systems

From a system perspective, an easy extension is to develop a system that unifies the first-order rewriting techniques and the generic solver methods. Specifically, such a system would first decide, upon receiving an input query q, the complexity of $\mathsf{CERTAINTY}(q)$. If $\mathsf{CERTAINTY}(q)$ is **FO**-rewritable, it runs the best known rewriting possible; or otherwise, default to the solver approach. It has also been discovered that $\mathsf{CERTAINTY}(q)$ can sometimes be expressible in stratified Datalog. Hence systems that use Datalog engines as the backend are potentially valuable.

6.2 CQA for Data Cleaning

We also argue that CQA systems can in turn support data cleaning systems with the help of provenance. Consider an inconsistent database $\mathbf{db}$, Σ the integrity constraint imposed on $\mathbf{db}$, and a given CQ q. Assume that $\mathbf{r}^*$ is the correct repair of $\mathbf{db}$ with respect to Σ. Since $\mathbf{r} \subseteq \mathbf{db}$ for every repair $\mathbf{r} \in \mathsf{repairs}(\mathbf{db}, \Sigma)$ and every CQ q is monotone, it follows that

$$\bigcap_{\mathbf{r} \in \mathsf{repairs}(\mathbf{db},\Sigma)} q(\mathbf{db}) \subseteq q(\mathbf{r}^*) \subseteq q(\mathbf{db}). \tag{6.1}$$

Equation 6.1 reveals that it suffices for data cleaning to resolve the portion of inconsistent data that result in the answers in the set difference

$$q(\mathbf{db}) \setminus \bigcap_{\mathbf{r} \in \mathsf{repairs}(\mathbf{db}, \Sigma)} q(\mathbf{db}), \tag{6.2}$$

with the help of provenance. This is because if the set difference (6.2) becomes empty, Equation 6.1 immediately yields that $q(\mathbf{db}) = q(\mathbf{r}^*)$. This has also been observed in our experiments in Chapter 3, Table 3.3: for the StackOverflow dataset, data cleaning is unnecessary for queries Q_1, Q_2, Q_3 and Q_4, because the set difference is empty! For other queries, the set difference is in fact rather small, which could potentially drastically reduce the workload of data cleaning systems. We refer to [DFAA23] for related techniques.

6.3 Complexity Classification

From a theoretical perspective, the complexity classification beyond rooted tree queries remains largely open. Note that rooted tree queries are still "acyclic" in nature, and thus any result on CQA over "cyclic" queries would be very inspiring. We remark that a **PTIME**/**coNP**-complete dichotomy for CQA on CQs with two self-joining atoms is recently established [PSS23], which provides some insights on how to handle primary keys of arbitrary arities.

In the remainder of this section, we report some preliminary progress towards classifying $\mathsf{CERTAINTY}(q)$ for Boolean conjunctive queries with binary atoms, which we call *graph queries*.

6.3.1 Cycle queries

A *cycle query* is a Boolean conjunctive query of the form

$$q = \exists x_0, x_1, \dots, x_{n-1} : R_0(\underline{x_0}, x_1) \wedge R_1(\underline{x_1}, x_2) \wedge \cdots \wedge R_{n-1}(\underline{x_{n-1}}, x_0).$$

Consider the following syntactic condition $\mathsf{C}_{\mathsf{reg}}$:

- $\mathsf{C}_{\mathsf{reg}}$: whenever $R_i = R_j$, we have $R_{i+1 \mod n} = R_{j+1 \mod n}$.

We have the following conjecture.

Conjecture 6.1. *Let q be a cycle query. If q satisfies $\mathsf{C}_{\mathsf{reg}}$, then $\mathsf{CERTAINTY}(q)$ is* **L***-complete; or otherwise* **PTIME***-complete.*

6.3.2 Query-Agnostic Polynomial-time Algorithms

We remark that a simple polynomial-time algorithm that is agnostic to the syntactic structure of queries for CQA is discovered in [FPSS23], which is shown to be correct for the tractable cases of self-join-free BCQs and path queries. The dichotomy in [PSS23] also relies on that simple algorithm, as well as a new matching-based polynomial-time algorithm. The extent to which these two algorithms correctly capture the **PTIME/coNP**-complete boundary is also intriguingly open.

Chapter 7

Conclusions

"It's all about the journey."

—Dane County Regional Airport Parking Ticket

This dissertation primarily focuses on methods to answer queries in the presence of inconsistent data. Specifically, we consider data that could possibly violate the primary-key constraints and methods to support answering selection-projection-join queries over it. We study consistent query answering (CQA), with the goal of returning the answers that are guaranteed to be returned, regardless of which repair the query is executed on.

We investigate the system aspect of the problem. For the self-join-free selection-projection-join queries that admit a pair-pruning join tree, their consistent answers can be computed by running another SQL query directly on the inconsistent data. We show that these rewritings are guaranteed to run in linear time (Theorem 1.2), exhibiting no asymptotic overhead to the classical Yannakakis' algorithm on consistent data. We also implemented LinCQA, which will produce a first-order rewriting, available in both SQL and Datalog, for every query with a pair-pruning join tree. Our experiments show that LinCQA outperforms existing CQA systems, sometimes by orders of magnitudes.

From a theoretical perspective, we obtain fine-grained complexity classifications of CQA on conjunctive queries possibly with self-joins, namely path queries and rooted tree queries. For every path query q, we show that $\mathsf{CERTAINTY}(q)$ is in **FO**, **NL**-complete, **PTIME**-complete, or **coNP**-complete (Theorem 1.3); and for every rooted tree query q, $\mathsf{CERTAINTY}(q)$ is in **FO**, **NL**-hard $\cap$ **LFPL**, or **coNP**-complete (Theorem 1.4). The complexity classification for rooted tree queries can also be extended to other class of queries (Theorem 5.2 and 5.3). These endeavors reveal that homomorphisms among a set of queries obtained by copying subqueries that are related by self-joins may play a critical role in the ultimate complexity classification of CQA.

LIST OF REFERENCES

[ABC99] Marcelo Arenas, Leopoldo E. Bertossi, and Jan Chomicki. Consistent query answers in inconsistent databases. In *PODS*, pages 68–79. ACM Press, 1999.

[ABL^{+}22] Marcelo Arenas, Pablo Barceló, Leonid Libkin, Wim Martens, and Andreas Pieris. *Database Theory*. Open source at https://github.com/pdm-book/community, 2022.

[AG94] Miklós Ajtai and Yuri Gurevich. Datalog vs first-order logic. *J. Comput. Syst. Sci.*, 49(3):562–588, 1994.

[AGM^{+}15] Patricia C. Arocena, Boris Glavic, Giansalvatore Mecca, Renée J. Miller, Paolo Papotti, and Donatello Santoro. Messing up with BART: error generation for evaluating data-cleaning algorithms. *Proc. VLDB Endow.*, 9(2):36–47, 2015.

[AHV95] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[AJKO08] Lyublena Antova, Thomas Jansen, Christoph Koch, and Dan Olteanu. Fast and simple relational processing of uncertain data. In Gustavo Alonso, José A. Blakeley, and Arbee L. P. Chen, editors, *Proceedings of the 24th International Conference on Data Engineering, ICDE 2008, April 7-12, 2008, Cancún, Mexico*, pages 983–992. IEEE Computer Society, 2008.

[AK09] Arvind Arasu and Raghav Kaushik. A grammar-based entity representation framework for data cleaning. In *SIGMOD Conference*, pages 233–244. ACM, 2009.

[AKV15] Foto N. Afrati, Phokion G. Kolaitis, and Angelos Vasilakopoulos. Consistent answers of conjunctive queries on graphs. *CoRR*, abs/1503.00650, 2015.

[BDG07] 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.

[Ber19] Leopoldo E. Bertossi. Database repairs and consistent query answering: Origins and further developments. In *PODS*, pages 48–58. ACM, 2019.

[BFG$^+$07] Philip Bohannon, Wenfei Fan, Floris Geerts, Xibei Jia, and Anastasios Kementsietsidis. Conditional functional dependencies for data cleaning. In *ICDE*, pages 746–755. IEEE Computer Society, 2007.

[BFMY83] Catriel Beeri, Ronald Fagin, David Maier, and Mihalis Yannakakis. On the desirability of acyclic database schemes. *J. ACM*, 30(3):479–513, 1983.

[BKL13] Leopoldo E. Bertossi, Solmaz Kolahi, and Laks V. S. Lakshmanan. Data cleaning and query answering with matching dependencies and matching functions. *Theory Comput. Syst.*, 52(3):441–482, 2013.

[BKS17] Christoph Berkholz, Jens Keppeler, and Nicole Schweikardt. Answering conjunctive queries under updates. In *PODS*, pages 303–318. ACM, 2017.

[BMNT15] Moria Bergman, Tova Milo, Slava Novgorodov, and Wang Chiew Tan. Query-oriented data cleaning with oracles. In *SIGMOD Conference*, pages 1199–1214. ACM, 2015.

[BOFK23] Song Bian, Xiating Ouyang, Zhiwei Fan, and Paraschos Koutris. Certifiable robustness for naive bayes classifiers. *CoRR*, abs/2303.04811, 2023.

[Bul11] Andrei A. Bulatov. Complexity of conservative constraint satisfaction problems. *ACM Trans. Comput. Log.*, 12(4):24:1–24:66, 2011.

[Bul17] Andrei A. Bulatov. A dichotomy theorem for nonuniform csps. In Chris Umans, editor, *58th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2017, Berkeley, CA, USA, October 15-17, 2017*, pages 319–330. IEEE Computer Society, 2017.

[CCP19] Marco Calautti, Marco Console, and Andreas Pieris. Counting database repairs under primary keys revisited. In *PODS*, pages 104–118. ACM, 2019.

[CCP21] Marco Calautti, Marco Console, and Andreas Pieris. Benchmarking approximate consistent query answering. In *PODS*, pages 233–246. ACM, 2021.

[CCX08] Reynold Cheng, Jinchuan Chen, and Xike Xie. Cleaning uncertain data with quality guarantees. *Proc. VLDB Endow.*, 1(1):722–735, 2008.

[CIKW16] Xu Chu, Ihab F. Ilyas, Sanjay Krishnan, and Jiannan Wang. Data cleaning: Overview and emerging challenges. In *SIGMOD Conference*, pages 2201–2206. ACM, 2016.

[CIP13] Xu Chu, Ihab F. Ilyas, and Paolo Papotti. Holistic data cleaning: Putting violations into context. In *ICDE*, pages 458–469. IEEE Computer Society, 2013.

[Clo18] https://www.cloudlab.us/, 2018.

[CLP18] Marco Calautti, Leonid Libkin, and Andreas Pieris. An operational approach to consistent query answering. In *PODS*, pages 239–251. ACM, 2018.

[CLPS22a] 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.

[CLPS22b] Marco Calautti, Ester Livshits, Andreas Pieris, and Markus Schneider. Uniform operational consistent query answering. In *PODS*, pages 393–402. ACM, 2022.

[CMI$^+$15] Xu Chu, John Morcos, Ihab F. Ilyas, Mourad Ouzzani, Paolo Papotti, Nan Tang, and Yin Ye. KATARA: A data cleaning system powered by knowledge bases and crowdsourcing. In *SIGMOD Conference*, pages 1247–1261. ACM, 2015.

[Cod70] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, 1970.

[CZS$^+$19] Hongge Chen, Huan Zhang, Si Si, Yang Li, Duane Boning, and Cho-Jui Hsieh. Robustness verification of tree-based models. *arXiv:1906.03849*, 2019.

[DB11] Jessica Davies and Fahiem Bacchus. Solving MAXSAT by solving a sequence of simpler SAT instances. In *CP*, volume 6876 of *Lecture Notes in Computer Science*, pages 225–239. Springer, 2011.

[DFAA23] Osnat Drien, Matanya Freiman, Antoine Amarilli, and Yael Amsterdamer. Query-guided resolution in uncertain databases. *Proc. ACM Manag. Data*, 1(2):180:1–180:27, 2023.

[DHK20] Shaleen Deep, Xiao Hu, and Paraschos Koutris. Fast join project query evaluation using matrix multiplication. In *SIGMOD Conference*, pages 1213–1223. ACM, 2020.

[DHK21] Shaleen Deep, Xiao Hu, and Paraschos Koutris. Enumeration algorithms for conjunctive queries with projection. In *ICDT*, volume 186 of *LIPIcs*, pages 14:1–14:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

[Dix21] Akhil Anand Dixit. *Answering Queries Over Inconsistent Databases Using SAT Solvers*. PhD thesis, UC Santa Cruz, 2021.

[DK19] 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.

[DK21a] Akhil A. Dixit and Phokion G. Kolaitis. Cavsat: Answering aggregation queries over inconsistent databases via SAT solving. In *SIGMOD Conference*, pages 2701–2705. ACM, 2021.

[DK21b] Akhil A. Dixit and Phokion G. Kolaitis. Consistent answers of aggregation queries using SAT solvers. *CoRR*, abs/2103.03314, 2021.

[EEI$^+$13] Amr Ebaid, Ahmed K. Elmagarmid, Ihab F. Ilyas, Mourad Ouzzani, Jorge-Arnulfo Quiané-Ruiz, Nan Tang, and Si Yin. NADEEF: A generalized data cleaning system. *Proc. VLDB Endow.*, 6(12):1218–1221, 2013.

[Fan22] Zhiwei Fan. *Building Datalog Systems for Scalable and Efficient Data Analytics*. PhD thesis, The University of Wisconsin-Madison, 2022.

[FFM05] Ariel Fuxman, Elham Fazli, and Renée J Miller. Conquer: Efficient management of inconsistent databases. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 155–166, 2005.

[FGIM15] Cibele Freire, Wolfgang Gatterbauer, Neil Immerman, and Alexandra Meliou. The complexity of resilience and responsibility for self-join-free conjunctive queries. *Proc. VLDB Endow.*, 9(3):180–191, 2015.

[FGIM20] Cibele Freire, Wolfgang Gatterbauer, Neil Immerman, and Alexandra Meliou. New results for the complexity of resilience for binary conjunctive queries with self-joins. In *PODS*, pages 271–284. ACM, 2020.

[FK22] Austen Z Fan and Paraschos Koutris. Certifiable robustness for nearest neighbor classifiers. *arXiv:2201.04770*, 2022.

[FKOW23] Zhiwei Fan, Paraschos Koutris, Xiating Ouyang, and Jef Wijsen. Lincqa: Faster consistent query answering with linear time guarantees. *Proc. ACM Manag. Data*, 1(1):38:1–38:25, 2023.

[FKZ23] Austen Z. Fan, Paraschos Koutris, and Hangdong Zhao. The fine-grained complexity of boolean conjunctive queries and sum-product problems. In Kousha Etessami, Uriel Feige, and Gabriele Puppis, editors, *50th International Colloquium on Automata, Languages, and Programming, ICALP 2023, July 10-14, 2023, Paderborn, Germany*, volume 261 of *LIPIcs*, pages 127:1–127:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023.

[FM07] Ariel Fuxman and Renée J. Miller. First-order query rewriting for inconsistent databases. *J. Comput. Syst. Sci.*, 73(4):610–635, 2007.

[FMK22] Zhiwei Fan, Sunil Mallireddy, and Paraschos Koutris. Towards better understanding of the performance and design of datalog systems. In Mario Alviano and Andreas Pieris, editors, *Proceedings of the 4th International Workshop on the Resurgence of Datalog in Academia and Industry (Datalog-2.0 2022) co-located with the 16th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2022), Genova-Nervi, Italy, September 5, 2022*, volume 3203 of *CEUR Workshop Proceedings*, pages 166–180. CEUR-WS.org, 2022.

[Fon15] 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.

[FPSS23] Diego Figueira, Anantha Padmanabha, Luc Segoufin, and Cristina Sirangelo. A simple algorithm for consistent query answering under primary keys. In *ICDT*, volume 255 of *LIPIcs*, pages 24:1–24:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023.

[Fux07] Ariel Damian Fuxman. *Efficient Query Processing Over Inconsistent Databases.* PhD thesis, University of Toronto, 2007.

[FZZ$^+$19] Zhiwei Fan, Jianqiao Zhu, Zuyu Zhang, Aws Albarghouthi, Paraschos Koutris, and Jignesh M. Patel. Scaling-up in-memory datalog processing: Observations and techniques. *Proc. VLDB Endow.*, 12(6):695–708, 2019.

[GGM$^+$21] Congcong Ge, Yunjun Gao, Xiaoye Miao, Bin Yao, and Haobo Wang. A hybrid data cleaning framework using markov logic networks (extended abstract). In *ICDE*, pages 2344–2345. IEEE, 2021.

[GGZ03] Gianluigi Greco, Sergio Greco, and Ester Zumpano. A logical framework for querying and repairing inconsistent databases. *IEEE Trans. Knowl. Data Eng.*, 15(6):1389–1408, 2003.

[GMPS13] Floris Geerts, Giansalvatore Mecca, Paolo Papotti, and Donatello Santoro. The LLUNATIC data-cleaning framework. *Proc. VLDB Endow.*, 6(9):625–636, 2013.

[Gol77] Leslie M. Goldschlager. The monotone and planar circuit value problems are log space complete for P. *SIGACT News*, 9(2):25–29, July 1977.

[GWYY21] Zhihan Guo, Kan Wu, Cong Yan, and Xiangyao Yu. Releasing locks as early as you can: Reducing contention of hotspots by violating two-phase locking. 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 658–670. ACM, 2021.

[HCG$^+$18] Yeye He, Xu Chu, Kris Ganjam, Yudian Zheng, Vivek R. Narasayya, and Surajit Chaudhuri. Transform-data-by-example (TDE): an extensible search engine for data transformations. *Proc. VLDB Endow.*, 11(10):1165–1177, 2018.

[Hsi60] Harrison Hsia. *Economic Decision Making in Hog Feeding - A New Approach.* PhD thesis, University of Wisconsin - Madison, 1960.

[HW23] Xiao Hu and Qichen Wang. Computing the difference of conjunctive queries efficiently. *Proc. ACM Manag. Data*, 1(2):153:1–153:26, 2023.

[Imm88] Neil Immerman. Nondeterministic space is closed under complementation. *SIAM J. Comput.*, 17(5):935–938, oct 1988.

[KIJ$^+$15] Zuhair Khayyat, Ihab F. Ilyas, Alekh Jindal, Samuel Madden, Mourad Ouzzani, Paolo Papotti, Jorge-Arnulfo Quiané-Ruiz, Nan Tang, and Si Yin. Bigdansing: A system for big data cleansing. In *SIGMOD Conference*, pages 1215–1230. ACM, 2015.

[KJL$^+$20] 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.

[KL21] Henning Kohler and Sebastian Link. Possibilistic data cleaning. *IEEE Transactions on Knowledge and Data Engineering*, 2021.

[KLP20] Benny Kimelfeld, Ester Livshits, and Liat Peterfreund. Counting and enumerating preferred database repairs. *Theor. Comput. Sci.*, 837:115–157, 2020.

[KLW$^+$20] Bojan Karlaš, Peng Li, Renzhi Wu, Nezihe Merve Gürel, Xu Chu, Wentao Wu, and Ce Zhang. Nearest neighbor classifiers over incomplete information: From certain answers to certain predictions. *arXiv:2005.05117*, 2020.

[KNP$^+$22a] Mahmoud Abo Khamis, Hung Q. Ngo, Reinhard Pichler, Dan Suciu, and Yisu Remy Wang. Convergence of datalog over (pre-) semirings. In Leonid Libkin and Pablo Barceló, editors, *PODS '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, pages 105–117. ACM, 2022.

[KNP$^+$22b] Mahmoud Abo Khamis, Hung Q. Ngo, Reinhard Pichler, Dan Suciu, and Yisu Remy Wang. Datalog in wonderland. *SIGMOD Rec.*, 51(2):6–17, 2022.

[KNP$^+$23] Mahmoud Abo Khamis, Hung Q. Ngo, Reinhard Pichler, Dan Suciu, and Yisu Remy Wang. Convergence of datalog over (pre-) semirings. *SIGMOD Rec.*, 52(1):75–82, 2023.

[KNR$^+$11] Jay Kreps, Neha Narkhede, Jun Rao, et al. Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB*, volume 11, pages 1–7. Athens, Greece, 2011.

[KNS17] 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.

[KOW21] Paraschos Koutris, Xiating Ouyang, and Jef Wijsen. Consistent query answering for primary keys on path queries. In *PODS*, pages 215–232. ACM, 2021.

[KOW23] Paraschos Koutris, Xiating Ouyang, and Jef Wijsen. Consistent query answering for primary keys on rooted tree queries. *CoRR*, abs/2310.19642, 2023.

[KP12] 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.

[KPT13] 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.

[KS14] Paraschos Koutris and Dan Suciu. A dichotomy on the complexity of consistent query answering for atoms with simple keys. In *ICDT*, pages 165–176. OpenProceedings.org, 2014.

[KW15] Paraschos Koutris and Jef Wijsen. The data complexity of consistent query answering for self-join-free conjunctive queries under primary key constraints. In *PODS*, pages 17–29. ACM, 2015.

[KW17] 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.

[KW18] Paraschos Koutris and Jef Wijsen. Consistent query answering for primary keys and conjunctive queries with negated atoms. In *PODS*, pages 209–224. ACM, 2018.

[KW19] Paraschos Koutris and Jef Wijsen. Consistent query answering for primary keys in logspace. In *ICDT*, volume 127 of *LIPIcs*, pages 23:1–23:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

[KW20] Paraschos Koutris and Jef Wijsen. First-order rewritability in consistent query answering with respect to multiple keys. In *PODS*, pages 113–129. ACM, 2020.

[KW21] Paraschos Koutris and Jef Wijsen. Consistent query answering for primary keys in datalog. *Theory Comput. Syst.*, 65(1):122–178, 2021.

[KW23] Aziz Amezian El Khalfioui and Jef Wijsen. Consistent query answering for primary keys and conjunctive queries with counting. In Floris Geerts and Brecht Vandevoort, editors, *26th International Conference on Database Theory, ICDT 2023, March 28-31, 2023, Ioannina, Greece*, volume 255 of *LIPIcs*, pages 23:1–23:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023.

[KWW$^+$16] Sanjay Krishnan, Jiannan Wang, Eugene Wu, Michael J. Franklin, and Ken Goldberg. Activeclean: Interactive data cleaning for statistical modeling. *Proc. VLDB Endow.*, 9(12):948–959, 2016.

[Lad75] Richard E. Ladner. On the structure of polynomial time reducibility. *J. ACM*, 22(1):155–171, jan 1975.

[Lib04] Leonid Libkin. *Elements of Finite Model Theory.* Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004.

[LRB$^+$21] Peng Li, Xi Rao, Jennifer Blase, Yue Zhang, Xu Chu, and Ce Zhang. Cleanml: A study for evaluating the impact of data cleaning on ML classification tasks. In *ICDE*, pages 13–24. IEEE, 2021.

[LS79] Butler W Lampson and Howard E Sturgis. *Crash recovery in a distributed data storage system.* Xerox Palo Alto Research Center Palo Alto, California, 1979.

[LW15] Carsten Lutz and Frank Wolter. On the relationship between consistent query answering and constraint satisfaction problems. In *ICDT*, volume 31 of *LIPIcs*, pages 363–379. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2015.

[MB10] Mónica Caniupán Marileo and Leopoldo E. Bertossi. The consistency extractor system: Answer set programs for consistent query answering in databases. *Data Knowl. Eng.*, 69(6):545–572, 2010.

[MRT15] Marco Manna, Francesco Ricca, and Giorgio Terracina. Taming primary key violations to query large inconsistent data via ASP. *Theory Pract. Log. Program.*, 15(4-5):696–710, 2015.

[MW13] Dany Maslowski and Jef Wijsen. A dichotomy in the complexity of counting database repairs. *J. Comput. Syst. Sci.*, 79(6):958–983, 2013.

[MW14] Dany Maslowski and Jef Wijsen. Counting database repairs that satisfy conjunctive queries with self-joins. In *ICDT*, pages 155–164. OpenProceedings.org, 2014.

[NPRR12] Hung Q. Ngo, Ely Porat, Christopher Ré, and Atri Rudra. Worst-case optimal join algorithms: [extended abstract]. In Michael Benedikt, Markus Krötzsch, and Maurizio Lenzerini, editors, *Proceedings of the 31st ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2012, Scottsdale, AZ, USA, May 20-24, 2012*, pages 37–48. ACM, 2012.

[OOCR09] Patrick E. O'Neil, Elizabeth J. O'Neil, Xuedong Chen, and Stephen Revilak. The star schema benchmark and augmented fact table indexing. In Raghunath Othayoth Nambiar and Meikel Poess, editors, *Performance Evaluation and Benchmarking, First TPC Technology Conference, TPCTC 2009, Lyon, France, August 24-28, 2009, Revised Selected Papers*, volume 5895 of *Lecture Notes in Computer Science*, pages 237–252. Springer, 2009.

[PF00] Meikel Poess and Chris Floyd. New tpc benchmarks for decision support and web commerce. *ACM Sigmod Record*, 29(4):64–71, 2000.

[PSC$^+$15] Nataliya Prokoshyna, Jaroslaw Szlichta, Fei Chiang, Renée J. Miller, and Divesh Srivastava. Combining quantitative and logical data cleaning. *Proc. VLDB Endow.*, 9(4):300–311, 2015.

[PSS23] Anantha Padmanabha, Luc Segoufin, and Cristina Sirangelo. A dichotomy in the complexity of consistent query answering for two atom queries with self-join. *CoRR*, abs/2309.12059, 2023.

[PTH23] Yeonsu Park, Byungchul Tak, and Wook-Shin Han. Qaad (query-as-a-data): Scalable execution of massive number of small queries in spark. *Proc. ACM Manag. Data*, 1(2):134:1–134:26, 2023.

[RCIR17] Theodoros Rekatsinas, Xu Chu, Ihab F. Ilyas, and Christopher Ré. Holoclean: Holistic data repairs with probabilistic inference. *Proc. VLDB Endow.*, 10(11):1190–1201, 2017.

[RD00] Erhard Rahm and Hong Hai Do. Data cleaning: Problems and current approaches. *IEEE Data Eng. Bull.*, 23(4):3–13, 2000.

[RG03] Raghu Ramakrishnan and Johannes Gehrke. *Database management systems*, volume 3. McGraw-Hill New York, 2003.

[RJG$^+$21] Abhishek Roy, Alekh Jindal, Priyanka Gomatam, Xiating Ouyang, Ashit Gosalia, Nishkam Ravi, Swinky Mann, and Prakhar Jain. Sparkcruise: Workload optimization in managed spark clusters at microsoft. *Proc. VLDB Endow.*, 14(12):3122–3134, 2021.

[ROA$^+$21] El Kindi Rezig, Mourad Ouzzani, Walid G. Aref, Ahmed K. Elmagarmid, Ahmed R. Mahmood, and Michael Stonebraker. Horizon: Scalable dependency-driven data cleaning. *Proc. VLDB Endow.*, 14(11):2546–2554, 2021.

[SCM12] 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.

[SKRC10] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In Mohammed G. Khatib, Xubin He, and Michael Factor, editors, *IEEE 26th Symposium on Mass Storage Systems and Technologies, MSST 2012, Lake Tahoe, Nevada, USA, May 3-7, 2010*, pages 1–10. IEEE Computer Society, 2010.

[SWZ$^+$21] Zhouxing Shi, Yihan Wang, Huan Zhang, Jinfeng Yi, and Cho-Jui Hsieh. Fast certified robust training with short warmup. *NeurIPS*, 2021.

[SYI$^+$16] Alexander Shkapsky, Mohan Yang, Matteo Interlandi, Hsuan Chiu, Tyson Condie, and Carlo Zaniolo. Big data analytics with datalog queries on spark. In Fatma Özcan, Georgia Koutrika, and Sam Madden, editors, *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 1135–1149. ACM, 2016.

[Sze88] Róbert Szelepcsényi. The method of forced enumeration for nondeterministic automata. *Acta Informatica*, 26(3):279–284, 1988.

[TCZ$^+$14] Yongxin Tong, Caleb Chen Cao, Chen Jason Zhang, Yatao Li, and Lei Chen. Crowdcleaner: Data cleaning for multi-version data on the web via crowdsourcing. In *ICDE*, pages 1182–1185. IEEE Computer Society, 2014.

[VV21] Daniël Vos and Sicco Verwer. Efficient training of robust decision trees against adversarial examples. In *ICML*, 2021.

[Wij10] Jef Wijsen. On the first-order expressibility of computing certain answers to conjunctive queries over uncertain databases. In Jan Paredaens and Dirk Van Gucht, editors, *Proceedings of the Twenty-Ninth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2010, June 6-11, 2010, Indianapolis, Indiana, USA*, pages 179–190. ACM, 2010.

[Wij12] Jef Wijsen. Certain conjunctive query answering in first-order logic. *ACM Trans. Database Syst.*, 37(2):9:1–9:35, 2012.

[Wij19a] Jef Wijsen. Corrigendum to "counting database repairs that satisfy conjunctive queries with self-joins". *CoRR*, abs/1903.12469, 2019.

[Wij19b] Jef Wijsen. Foundations of query answering on inconsistent databases. *SIGMOD Rec.*, 48(3):6–16, 2019.

[WWS23] Yisu Remy Wang, Max Willsey, and Dan Suciu. Free join: Unifying worst-case optimal and traditional joins. *Proc. ACM Manag. Data*, 1(2):150:1–150:23, 2023.

[Yan81] 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.

[ZCF$^+$10] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In Erich M. Nahum and Dongyan Xu, editors, *2nd USENIX Workshop on Hot Topics in Cloud Computing, HotCloud'10, Boston, MA, USA, June 22, 2010.* USENIX Association, 2010.

[ZCT23] Cheng Zhen, Amandeep Singh Chabada, and Arash Termehchy. When can we ignore missing data in model training? In *Proceedings of the Seventh Workshop on Data Management for End-to-End Machine Learning, DEEM 2023, Seattle, WA, USA, 18 June 2023*, pages 4:1–4:4. ACM, 2023.

[ZDK23] Hangdong Zhao, Shaleen Deep, and Paraschos Koutris. Space-time tradeoffs for conjunctive queries with access patterns. In Floris Geerts, Hung Q. Ngo, and Stavros Sintos, editors, *Proceedings of the 42nd ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2023, Seattle, WA, USA, June 18-23, 2023*, pages 59–68. ACM, 2023.

[ZFOK23] Hangdong Zhao, Austen Z. Fan, Xiating Ouyang, and Paraschos Koutris. Conjunctive queries with negation and aggregation: A linear time characterization. *CoRR*, abs/2310.05385, 2023.

[Zhu20] Dmitriy Zhuk. A proof of the CSP dichotomy conjecture. *J. ACM*, 67(5):30:1–30:78, 2020.