Query Evaluation Revised: Parallel, Distributed, via Rewritings

Dissertation

zur Erlangung des Grades eines

D o k t o r s d e r N a t u r w i s s e n s c h a f t e n

der Technischen Universität Dortmund
an der Fakultät für Informatik

von

Christopher Spinrath

Dortmund

2024

Tag der mündlichen Prüfung: 29.01.2024
Dekan: Prof. Dr-Ing. Gernot A. Fink

Gutachter:

- Prof. Dr Thomas Schwentick
- Prof. Dr Reinhard Pichler

Abstract

This is a thesis on query evaluation in parallel and distributed settings, and structurally simple rewritings of queries. It consists of three parts, each highlighting various aspects of query evaluation.

In the first part, the efficiency of constant-time parallel evaluation algorithms is investigated. That is, the number of required processors or, asymptotically equivalent, the *work* required to evaluate queries in constant time. From fundamental results by Codd and Immerman, it easily follows that all relational algebra queries can be evaluated in constant time on an appropriate PRAM model in principle. However, these results do not focus on work-efficiency, and indeed lead to evaluation algorithms with huge (polynomial) bounds on the work, and the tuples of query result that can be extremely scattered in memory. In particular, such algorithm require substantially more work than classical, sequential algorithms require time, for several classes of queries.

We explore work-efficient constant-time evaluation algorithms for classes of queries, for which efficient sequential query evaluation algorithms are known: Acyclic queries, free-connex acyclic queries, semi-join algebra queries, and natural join queries – the latter in the worst-case optimal framework.

Given suitable data structures for the database relations, our algorithms are substantially more work-efficient than naive algorithms. In the case of semi-join queries, the work of our algorithms even matches the running time of the best sequential algorithms, and in the other cases it comes reasonably close.

The second part is about deciding parallel-correctness – a natural problem in the context of distributed query evaluation: In a nutshell, given a query and policies specifying how data is divided and communicated among multiple servers, does the distributed evaluation of the query on the servers yield the same result as evaluating the query, over every database?

Ketsman et al. started the investigation of the distributed evaluation of recursive queries in the Massively Parallel Communication (MPC) model. Among other results, they proved that parallel-correctness for general Datalog programs is undecidable, by a reduction from the undecidable containment problem for Datalog.

We show that the undecidability of parallel-correctness runs deeper: It already holds for fragments of Datalog with a decidable containment problem, specifically monadic and frontier-guarded Datalog; even under relatively simple communication policies. These simple communication policies are defined in terms of data-moving distribution constraints. We then study a property – which can be imposed by syntactic restrictions of the constraints, or semantically, and show that the parallel-correctness problem for frontier-guarded Datalog and constraints with this property is 2ExpTime-complete. Furthermore, we will obtain the same bounds for the parallel-boundedness problem,

which asks whether the number of required communication rounds is bounded, over all databases.

Interestingly, our investigations reveal that not every monadic Datalog query is effectively equivalent to a frontier-guarded one in the distributed setting, although this holds in the classical, sequential setting.

The third part is about structurally simple rewritings. The (classical) rewriting problem asks whether, for a given query Q and a set $\mathcal{V}$ of views, there is a query Q' – called rewriting – over $\mathcal{V}$ that is equivalent to Q. Levy et al. proved that this problem is NP-complete for conjunctive queries and views.

We study the variant of this problem for (subclasses of) conjunctive queries and views that asks for a structurally simple rewriting Q'. Concerning the existence of structurally simple rewritings, we prove that, if the given query Q is acyclic itself, an acyclic rewriting exists if there is any rewriting at all. Analogous statements also hold for free-connex acyclic, hierarchical, and q-hierarchical queries.

We then study the complexity of our variant of the rewriting problem: It turns out that it is NP-hard, even if both Q and the views in $\mathcal{V}$ are acyclic or hierarchical, and the arity of the database schema is bounded. However, it becomes tractable if the views are free-connex acyclic or q-hierarchical (and the arity of the database schema is bounded).

Contents

Chapter 1

Introduction

Evaluating database queries is an ever-evolving topic with many facets. With technical advances and new applications, requirements and circumstances for evaluating queries are constantly subject to change. While sometimes new query languages and database models are meaningful to address these changes, it is often desirable (or even required) to consider well-established query languages and database models in settings with additional prerequisites, restrictions, requirements, etc., which have to be taken into account for query evaluation.

There are numerous examples. For instance, access to the database might be partially restricted due to privacy or fairness reasons. If the data is stored in an external memory or has to be transmitted over a network, accessing data might also be considered expensive – and should therefore be minimized. For huge databases evaluating queries in parallel using multiple processors or servers might be desirable. Query results from "similar" queries evaluated earlier on the same database instance could be provided as prerequisites to optimize the evaluation of the input query. Yet another example are constraints or derivation rules that have to be taken into account when evaluating a query.

A more involved setting is the following. Since the database is huge only the query results of a few selected queries, called *views*, are actually available to evaluate the actual queries formulated for the original database. Since the query results of the views are still large and fast query evaluation is paramount, multiple servers are utilized. Each server has in turn multiple processors. While the processors of a single server can communicate rather efficiently with each other, say via shared memory, communication between the servers is more expensive. It seems sensible to pursue the following three objectives in a scenario like this.

- Queries have to be adapted such that they can be evaluated given the query results of the views. If possible they should retain all their good properties concerning query evaluation.
- The communication and distribution of data between the servers should be restricted. Preferably the database does *not* have to be redistributed among the servers for each individual query.
- Each server should perform its task as fast as possible, utilizing all of its resources and, in particular, all of its processors.

From the perspective of Database Theory, studying these new settings involves aspects of the following kind – phrased here as questions.

(1) How can the setting be properly modelled, and what is a proper framework to study the setting?

(2) Are evaluation algorithms from classical or other settings applicable? If not, (how) can they be adapted, or are new algorithms, potentially influenced by other areas of Computer Science, desirable?

(3) If there are restrictions or constraints, is correctness of query evaluation guaranteed? That is, does the query result computed in the presence of these restrictions and constraints coincide with the (hypothetical) query result computed without them? If not, is it decidable, given a query and (a family of) restrictions and constraints, whether sound and complete evaluation is possible for any database instance? And if it is decidable, what is the complexity?

(4) What are proper measures to rate query evaluation algorithms, and when do we consider an evaluation algorithm or the complexity of a problem to be "good" or "efficient"?

(5) What is the complexity of query evaluation, or, if query evaluation is achieved by some kind of reduction or translation, the complexity of computing this reduction?

(6) Are there characterizations useful for, e.g., designing algorithms, proving complexity bounds, or identifying circumstances that allow for a more efficient evaluation?

(7) How do algorithms and the complexity of problems compare to their analogous in other settings, most notably "classical" settings?

Of course, this list is *not* exclusive, and there might be further aspects, depending on the setting. It also depends on the setting how relevant any of these aspects are – some can possibly be easily or even trivially addressed for one setting but for other settings it might be more involved. Furthermore, the order in which the aspects are best addressed can vary depending on the setting as well. Addressing Aspect (6) can, for example, yield characterizations which, in turn, can be very helpful to design algorithms, and, thereby, addressing Aspect (2). Fully addressing an aspect might also require considering query evaluation in a broad sense. For example, query optimization, which can arguably be considered a topic on its own but is entangled with query evaluation, might play a role for Aspects (5) and (7).

Last but not least, it is potentially meaningful to consider Aspects (2) to (7) for fragments of query languages which are known to be well-behaved in other settings.

In this thesis we delve into three settings (and some variations thereof), each highlighting different aspects among Aspects (1) to (7). We will briefly introduce these settings in the following before we discuss the overall structure and outline of this thesis. Related work will be primarily discussed in the main part of this thesis.

1.1 Settings and Main Results

The underlying database model of our settings is the classic relational model introduced by Codd [Cod70]. Relational databases are widely deployed – ranging from large-scale enterprise deployments to instances embedded into everyday applications for personal computers. Consequently, there is a wide, expanding range of applications and settings for query evaluation on relational databases.

Codd [Cod72] also introduced the relational algebra which is the foundation for SQL – the query language commonly implemented by relational database management systems – and permits precise and mathematical sound reasoning about queries stated in terms of relational expressions. An alternative are rule based query languages, most prominently conjunctive queries defined by Chandra and Merlin [CM77]. They capture some core aspects of SQL; more precisely, conjunctive queries are exactly the queries that can be expressed by relational algebra expressions using only the select, project, and join operators [cf., e.g., Are+21, Theorem 12.7]. The query language Datalog can be seen as an extension of conjunctive queries by recursion which is often desirable to express reachability queries. There are various more query languages for relational databases. Which query language is appropriate for a certain setting depends, in general, on multiple factors: The setting itself, the questions being investigated, prior work, how well results and proofs can be stated, etc. Consequently, which query language(s) we consider will vary depending on the setting.

1.1.1 Work-Efficient Constant-Time Parallel Evaluation

This setting is concerned with the data complexity of parallel query evaluation algorithms that can utilize a vast amount of processors.

Concerning Aspect (1), we will use the *Parallel Random Access Machine (PRAM)* model, which allows for a fine-grained analysis of parallel algorithms. Immerman [Imm89; Imm99] showed that PRAMs with polynomially many processors can evaluate first-order formulas in constant time, i.e. in time $\mathcal{O}(1)$. The same applies to relational algebra queries because they can be translated into equivalent first-order formulas [Cod72]. Comparing solely the running time of parallel algorithms and sequential algorithms neglects, however, that parallelism requires a potentially huge number of processors, which can be understood as another kind of valuable resource.

In the PRAM literature, it is common to compare the *work* of a parallel algorithm with the running time of sequential algorithms instead [cf., e.g., JáJ92]. Here the work is the overall number of computation steps in a PRAM computation. This will also be the primary measure we use in this thesis to address Aspects (4), (5) and (7). Regarding Aspects (4) and (7) in particular, an important goal is to design parallel algorithms that are *work-optimal* in the sense that their work asymptotically matches the running time of the best sequential algorithms. The other measure we will study is the space complexity.

Obviously, for constant-time PRAM algorithms the work and the number of processors are asymptotically equivalent. Thus, the result by Immerman – combined with the translation given by Codd [Cod72] – shows that relational algebra queries can be evaluated

with polynomial work. More precisely, the algorithms obtained this way require work $\mathcal{O}(\mathsf{IN}^k)$ where k is the number of variables of the (intermediate) formula. For many classes of queries this is *not* work-optimal. We note that optimizing the work was *not* an objective of these investigations. Surprisingly however, to the best of our knowledge, "work-efficiency" of constant-time PRAM algorithms for query evaluation has *not* been studied beyond this result in the literature.

Therefore, to investigate Aspects (2), (4), (5) and (7), we will develop PRAM algorithms for queries directly, omitting intermediate translations to, for example, formulas or circuits; thereby, building some foundations for "work-efficient" query evaluation in constant time. While it is not hard to come up with constant-time PRAM algorithms for relational expressions or conjunctive queries in principle, there are some obstacles affecting the "work-efficiency" of naive algorithms: Intuitively, the amount of data a single processor can access and communicate in constant time is very limited. We will see that these limitations manifest in obstacles like the following.

- It is, in general, *not* possible to represent the result of a (sub-)query in a compact form, say, as an array contiguously filled with result tuples.
- It is often necessary to allocate more processors than necessary, only for some to detect that they are superfluous and do *not* actually contribute to the query result. This can be illustrated with an array whose cells are *not* contiguously filled. That is, some cells are empty. To process each data item that is actually stored in the array it is often necessary to allocate one processor for each cell of the array – including the empty ones.
- The work-efficiency of operations like searching and sorting relies heavily on the representation of the data items. For sorting a constant-time PRAM algorithms is – to the best of our knowledge – *not* even known for the most general case.

Since intermediate results are also affected by obstacles like these, evaluating a relational algebra expression by combining algorithms for the individual operators naively, can, in general, lead to large bounds for the required work.

For example, a naive algorithm for the join $R \bowtie S$ of two relations R and S can simply allocate $|R| \cdot |S|$ processors, one for each pair of an R-tuple and an S-tuple. If the tuples assigned to a processor "match" it can write the corresponding result tuple into an array of length $|R| \cdot |S|$. Here we assume that the processors are numbered from 1 to $|R| \cdot |S|$, and each processor uses its number as index for the result array. In general, *not* every cell of the result array will contain a result tuple, and the algorithm is *not* work-optimal: In the sequential setting, sorting the relations suitably yields, for instance, a running time that is sub-quadratic in the input size (and linear in the output size).

We will investigate these obstacles (among others) and possible mitigations. Essential ingredients for our mitigations will be the constant-time PRAM algorithm for computing approximate prefix sums from Goldberg and Zwick [GZ95], and suitable representations of relations.

We consider three settings that differ in how databases can access (and represent) database relations.

- In the most general setting, domain values cannot be directly accessed. Instead, they are represented by tokens, and it can only be tested whether two tokens represent the same value.
- In the *ordered setting* we assume that there is a linear order on the domain values and it can be tested whether one value is smaller than another.
- In the *dictionary setting* domain values are mapped to an initial, linear-sized segment of the natural numbers by a dictionary. These *keys* can be directly accessed by a PRAM and used in computations.

For each of these setting, we will present algorithms for the operators of the relational algebra which are more "work-efficient" than naive algorithms.

Based on this, we will study algorithms for queries of the semi-join algebra (a fragment of the relational algebra), conjunctive queries, and natural join query – the latter in the worst-case optimal framework. Among other results, we will prove that

- there are work-optimal constant-time evaluation algorithms for queries of the semi-join algebra;
- there are parallel versions of the *Yannakakis algorithm* for acyclic conjunctive queries that require work $\mathcal{O}((\mathsf{IN} + \mathsf{IN} \cdot \mathsf{OUT})^{1+\varepsilon})$, for every fixed $\varepsilon > 0$; and
- an "almost" worst-case optimal work algorithm for natural join queries

in the dictionary setting. Except for the work-optimal algorithm for semi-join algebra queries, these results carry over to the ordered setting, if the input relations are suitably ordered a priori. We also present corresponding algorithms for the general setting, although they require (asymptotically) more work.

Regarding Aspect (2) these results will let us conclude that the query evaluation algorithms – which we consider – carry over to the parallel, constant-time setting reasonably well.

1.1.2 Parallel-Correctness of Distributed Query Evaluation

In this setting we shift focus to the communication between entities that participate in the distributed evaluation of queries. To emphasize this, we use the terms *servers* and *distributed* evaluation instead of processors and parallel evaluation; following the intuition that processors are "close" to each other and can access shared memory, while servers are "far apart" and have to communicate over some network. This is somewhat inspired by data management systems like *Apache Spark* [Apab; Xin+13] and *Apache Hadoop* [Apaa] which distribute large volumes of data among a massive amount of servers to speed up query processing.

Regarding Aspect (1), Beame et al. [BKS17a] established the *Massively Parallel Communication (MPC)* model to study distributed query evaluation. In this model query evaluation proceeds in multiple rounds where each round consists of a computation and a communication step. In the computation step, every server operates on its local data in isolation, whereas in the communication step data is exchanged between the servers.

In comparison, in the PRAM model computation and communication (via shared memory) is *not* clearly separated, and it is a "low-level" model in the sense that it stipulates how a processor can perform computations.[1]

The "high-level" MPC is therefore arguably more convenient and suited for studying distributed evaluations where the exact details on how the local computations are performed are *not* crucial.

For single-round algorithms, Ameloot et al. [Ame+17] considered an instantiation of Aspect (3): The problem of *parallel-correctness* asks whether one can always be sure that the corresponding algorithm computes the query result correctly, no matter the actual data, starting from a particular distribution policy. Here a distribution policy asserts the presence of certain data on certain servers. We emphasize that, in difference to the setting discussed in Section 1.1.1 this is a static analysis problem, and we are *not* concerned with data complexity here. Parallel-correctness and related problems were subsequently studied for conjunctive queries (with and without negation) and under set as well as bag semantics by Ameloot et al. [Ame+17], Geck et al. [Gec+16] and Ketsman et al. [KNV18].

Ketsman et al. [KAK20] started the investigation of the distributed evaluation of Datalog queries in the multi-round MPC model. To this end, they introduced a framework entailing *economic policies* as a means to specify data reshuffling in a recursive setting where intermediate derived facts can be communicated between servers. That is, in addition to specifying the initial distribution of the facts in the input database, economic policies also determine which servers can derive and use intensional facts during the evaluation of Datalog queries. Among other things, the authors showed that parallel-correctness for general Datalog queries is undecidable. In light of Aspect (4) they also proved that the parallel-boundedness problem, that asks whether a Datalog query can be evaluated in a bounded number of evaluation rounds is, for general Datalog queries, undecidable as well. Both undecidability results were proved by a reduction from the undecidable containment problem for Datalog.

In this thesis, we revisit the parallel-correctness and parallel-boundedness problem for Datalog queries. Given the prior work discussed above, we will primarily focus on identifying fragments of Datalog and policies with decidable parallel-correctness and parallel-boundedness problems. For this purpose, we build a more generic framework for the distributed evaluation of Datalog within the multi-round MPC model which allows for more general evaluation strategies than the economic policies of Ketsman et al. [KAK20].

Within this framework, we first establish that the undecidability of parallel-correctness runs deeper than the containment problem. We show that parallel-correctness is already undecidable under relatively simple distributed evaluation strategies for fragments of Datalog with a decidable containment problem, namely monadic and frontier-guarded Datalog queries. In a nutshell, monadic Datalog is the variant of Datalog where intensional predicates have arity at most one, whereas in frontier-guarded Datalog every rule should contain an extensional predicate mentioning all the variables occurring in the head of the rule [c.f., e.g., BKR15a; BCS11; BCO12].

[1] For details, we refer to Section 3.1.1.

Given these undecidability results, we will then proceed by identifying syntactical and semantic properties of policies which lead to decidable parallel-correctness and parallel-boundedness problems for monadic and frontier-guarded Datalog. More precisely, we show that these problems are 2ExpTime-complete, for suitable combinations of Datalog fragments and restricted policies. Although this complexity might seem daunting from a practical perspective, it is not unexpected, since the containment problems for monadic and frontier-guarded Datalog queries are 2ExpTime-complete and can easily be reduced to the corresponding parallel-correctness problems.

Our investigations will also reveal that classical results and coherences do *not* always hold in distributed settings: The usual ability to transform monadic Datalog queries into (equivalent) frontier-guarded Datalog queries can break drastically.

1.1.3 Structurally Simple Rewritings

The third and last setting which we consider in this thesis covers scenarios where the access to the database is restricted – for example, due to privacy, data protection, confidentiality reasons, or, as in the introductory example, the database is simply too large. This is somewhat orthogonal to the other two settings, in the sense, that such restrictions can be considered on top of and, to some degree, independent of parallel (or sequential) evaluation settings.[2]

A common way to model access restrictions to databases are *views* [cf., e.g., CY12; Hal01] which are queries with a special role: Instead of accessing the database directly, only the query results for the views can be accessed, or rather, referred to by actual user queries. Note that, in this model, it is *not* strictly necessary to *materialize*, i.e. compute, the query results for views. A database system could, for example, enumerate these results or inline the view definitions if the database system itself has access to the full database but does *not* expose it to its users.

For such settings Aspect (3) essentially boils down to the study of the *query rewriting problem* which asks, for a query Q and a set $\mathcal{V}$ of views, whether there is a query Q' over $\mathcal{V}$ that is equivalent to Q, and to find such a *rewriting* Q'.

If access to the database is only possible via views, it is arguably desirable that rewritings have good properties. As indicated by our results for the other two settings, structurally simple queries, like acyclic conjunctive queries, enjoy many good properties. Most notably, many classes of structurally simple queries allow for efficient query evaluation. This leads to a modified version of the query rewriting problem which does *not* merely ask for *any* rewriting, but rather for rewritings which are structurally simple and allow for efficient evaluation.

Let us emphasize that this modified version of the rewriting problem can also be relevant if access to the database is *not* restricted. For instance, this is the case for *multi-query evaluation* settings where multiple queries are given. A possible question in such a setting is, if one of the given queries can be evaluated more efficiently given the query results of the other (input) queries. Here the "other input queries" take the

[2] In distributed settings restrictions and distribution policies might *not* be fully independent.

role of the views. Building upon this one may ask for an optimal evaluation order of the given queries, minimizing the overall evaluation complexity. Furthermore, note that, if the set $\mathcal{V}$ of views consists of one view for each relation in the database, the question boils down to whether there is a structurally simple query equivalent to the input query – and, therefore, a classical query optimization problem.

In this thesis we are interested in the following two questions, which concern Aspects (2) and (5).

(a) Under which circumstances is it guaranteed that a structurally simple rewriting exists, if there exists a rewriting at all?

(b) What is the complexity to decide whether such a rewriting exists and to compute one?

We study Questions (a) and (b) for classes of structurally simple conjunctive queries, and depending on the structure of the given views and the given query. More precisely, we consider the classes of acyclic conjunctive queries (ACQ), and hierarchical conjunctive queries (HCQ), as well as their slightly stronger versions free-connex acyclic (CCQ) and q-hierarchical (QHCQ) conjunctive queries.

It is well-known that many problems are tractable for acyclic conjunctive queries but (presumably) *not* for conjunctive queries in general. Notably, the evaluation, minimization, and the containment problems are tractable for acyclic queries [Yan81; CR00; GLS01] but NP-complete for the class of conjunctive queries [CM77].

Hierarchical conjunctive queries play a central role in the context of probabilistic databases [DS07; FO14], and distributed query evaluation in the MPC model [KS11]. Free-connex acyclic and q-hierarchical conjunctive queries play a crucial role in the context of the enumeration complexity of queries [BDG07; BKS17b; BGS20]. For further applications of these query classes we refer to the articles of Kara et al. [Kar+20] and Fink and Olteanu [FO16] as well as the thesis of Keppeler [Kep20].

Our answer to Question (a) turns out to be very simple and also quite encouraging: If the original query is acyclic and there is any rewriting of it, then there is also an acyclic rewriting. And the same is true for the three subclasses HCQ, CCQ, and QHCQ of ACQ. To be more precise, let $\textsc{Rewr}(\mathbb{V}, \mathbb{Q}, \mathbb{R})$, for classes $\mathbb{V}$, $\mathbb{Q}$, and $\mathbb{R}$ of conjunctive queries, denote the rewriting problem that asks, given a set $\mathcal{V}$ views from $\mathbb{V}$ and a query Q from $\mathbb{Q}$, whether there is a rewriting of Q over $\mathcal{V}$ in $\mathbb{R}$. Our answer to Question (a) implies, in particular, that for $\mathbb{Q} \in \{\mathsf{ACQ}, \mathsf{CCQ}, \mathsf{HCQ}, \mathsf{QHCQ}\}$ and any set $\mathbb{V}$ of conjunctive queries the decision problems $\textsc{Rewr}(\mathbb{V}, \mathbb{Q}, \mathsf{CQ})$ and $\textsc{Rewr}(\mathbb{V}, \mathbb{Q}, \mathbb{Q})$ – and, thus, their complexities – coincide. This somewhat simplifies our study of Question (b).

Our answer to Question (b) reveals that the complexity of the acyclic rewriting problem depends mainly on two parameters: The views and the arity of the underlying database schema. We denote the restriction of $\textsc{Rewr}(\mathbb{V}, \mathbb{Q}, \mathbb{R})$ to database schemas of arity at most k by $\textsc{Rewr}^k(\mathbb{V}, \mathbb{Q}, \mathbb{R})$, and we indicate by $\mathbb{V}^k$ if the arity of views is at most k.

Our main results for Question (b) are as follows.

- The decision problems $\textsc{Rewr}^3(\mathsf{ACQ}, \mathsf{ACQ}, \mathsf{ACQ})$ and $\textsc{Rewr}^3(\mathsf{HCQ}, \mathsf{HCQ}, \mathsf{HCQ})$ are NP-complete.

- If arity of the views is bounded by some fixed k, and both, views and query are structurally simple, then the rewriting problem becomes tractable. In particular, we have that Rewr(ACQ^k, ACQ, ACQ) is in polynomial time, and an acyclic rewriting can be computed in polynomial time (if it exists). This follows easily with the help of the known *canonical rewriting* approach [see NSV10, Proposition 5.1] and our answer to Question (a).

- The same is true if the views are free-connex acyclic or q-hierarchical, and the arity of the database schema is bounded by some fixed k: The decision problems Rewr^k(CCQ, ACQ, ACQ), and Rewr^k(CCQ, CCQ, CCQ), as well as the decision problems Rewr^k(QHCQ, ACQ, ACQ), and Rewr^k(QHCQ, QHCQ, QHCQ) are in polynomial time.

Roughly speaking, the existence of a structurally simple rewriting thus depends on the input query, and the complexity of finding one depends, in addition, on the views (and the database schema).

We conclude with the remark that, to prove our answers to Questions (a) and (b), we will first address Aspect (6), and present a characterization of rewritability. Similar notions have been used in the literature, but ours is particularly suited for the study of exact (equivalent) rewritings.

1.2 Structure and Outline

In Chapter 2 we will introduce some basic notions, relational databases, and the query languages used throughout this thesis.

The main part of this thesis consists of three chapters, one for each of the aforementioned settings, namely Chapter 3 "Work-Efficient Query Evaluation with PRAMs", Chapter 4 "Distributed Evaluation of Datalog", and Chapter 5 "Structurally Simple Rewritings". Each of these main-part chapters starts with a detailed introduction and formalization of the respective setting, and we conclude each of them with a discussion of the results and related work. We provide a more detailed, individual outline in the opening of each chapter. They can be read in any order.

In Chapter 6 we give an overarching conclusion.

1.3 Publications

This thesis is based on the following publications, each of which lead to one of the main chapters of this thesis.

- Jens Keppeler, Thomas Schwentick and Christopher Spinrath. "Work-Efficient Query Evaluation with PRAMs". In: *26th International Conference on Database Theory, ICDT 2023, March 28-31, 2023, Ioannina, Greece.* [Full reference: KSS23].

- Frank Neven, Thomas Schwentick, Christopher Spinrath and Brecht Vandevoort. "Parallel-Correctness and Parallel-Boundedness for Datalog Programs". In: *22nd International Conference on Database Theory, ICDT 2019, March 26-28, 2019, Lisbon, Portugal*. [Full reference: Nev+19].

- Gaetano Geck, Jens Keppeler, Thomas Schwentick and Christopher Spinrath. "Rewriting with Acyclic Queries: Mind Your Head". In: *25th International Conference on Database Theory, ICDT 2022, March 29 to April 1, 2022, Edinburgh, UK (Virtual Conference)*. [Full reference: Gec+22].

A video presentation [Spi+22] from me[3] has also been published along with the conference paper "Rewriting with Acyclic Queries: Mind Your Head" [Gec+22]. Moreover, a journal version of this paper has been accepted for publication in the *Logical Methods in Computer Science (LMCS)* journal [Gec+23].

The individual contributions of my co-authors and me as well as the differences in content between the publications above and this thesis are discussed in the openings of Chapters 3 to 5.

[3]While the pronoun "we" is used throughout this thesis to involve the reader, the pronouns "I", "me", and "myself" are used to unambiguously refer to *me*, the author of this thesis, when discussing scientific contributions.

Chapter 2

Preliminaries

In this chapter, we fix some notation and recall the basic concepts from database theory that are relevant for this thesis. In Section 2.1 we introduce our notation for relational databases, and in Sections 2.2 to 2.4 we discuss query languages: The relational algebra, conjunctive queries, and Datalog as well as fragments and extensions thereof. In Section 2.5 we provide definitions and some basic facts for some machine and automata models which we use in this thesis to prove some upper and lower bounds. We start with notation for some fundamental mathematical concepts.

Basic Notation. By $\mathbb{N}_0$ we denote the set of non-negative integers. For integers $i, j \in \mathbb{N}_0$, we denote by $[i, j]$ the set $\{i, \dots, j\} = \{k \in \mathbb{N}_0 \mid i \leq k \leq j\}$. In the common case where $i = 1$, we write $[j]$ as a shorthand for $[1, j]$. We write $|S|$ for the number of elements in S when S is a set, a sequence, or a tuple. For a tuple $\bar{x} = (x_1, \dots, x_k)$ and a tuple $\bar{p} = (p_1, \dots, p_\ell) \in [1, k]^\ell$ of *positions* we write $\bar{x}[\bar{p}]$ for the tuple $(x_{p_1}, \dots, x_{p_\ell})$. As usual, we omit parentheses if they are redundant or the tuple consists of only one element. For instance, given the tuple $\bar{x} = (a, b, c)$, we have $|\bar{x}| = 3$, $\bar{x}[(2, 3, 2)] = (b, c, b)$, and $\bar{x}[1] = a$.

We use the natural extensions of mappings onto sets and tuples without notational distinction. That is, for a mapping $f \colon X \to Y$, we write $f(X')$ for $\{f(x) \mid x \in X'\}$ and $f(\bar{x})$ for $(f(x_1), \dots, f(x_k))$ for sets $X' \subseteq X$ and tuples $\bar{x} = (x_1, \dots, x_k) \in X^k$, respectively. The *composition* $f \circ g$ of two functions $g \colon X \to Y$ and $f \colon Y \to Z$ is the function $(f \circ g) \colon X \to Z$ defined by $(f \circ g)(x) = f(g(x))$ for all $x \in X$. For a (partial) function $f \colon X \rightharpoonup Y$, we write $\mathsf{dom}(f)$ for its domain, i.e. the set $X' \subseteq X$ of all elements in X for which f is defined. By id we denote the identity mapping (on any domain).

2.1 Relational Databases

Following Arenas et al. [Are+21, Chapter 2] we consider two perspectives from which databases can be defined: The *unnamed* and the *named* perspective. While the resulting formalisms are equally expressive, the former will be more convenient for studying static analysis problems in Chapters 4 and 5, and the latter will be more convenient for presenting evaluation algorithms in Chapter 3.

The Unnamed Perspective. Let dom be an infinite set of *domain values*.

Databases (and queries) are defined over database schemas. A *database schema* $\mathcal{S}$ is a finite set of *relation schemas*, each represented by a *(relation) symbol* R and associated

with a fixed *arity* $\mathsf{ar}(R) \in \mathbb{N}_0$. The *arity of* $\mathcal{S}$ is $\max_{R \in \mathcal{S}} \mathsf{ar}(R)$. We usually just write *schema* instead of database schema.

A relation symbol $R \in \mathcal{S}$ and a tuple $\bar{a}$ of domain values with $|\bar{a}| = \mathsf{ar}(R)$ constitute a *fact* $R(\bar{a})$ over $\mathcal{S}$. If we want to emphasize the relation symbol of a fact we call it an R-fact. A *database* D over a schema $\mathcal{S}$ is a finite set of facts over $\mathcal{S}$. The *active domain* of a database D, denoted $\mathsf{adom}(D) \subsetneq \mathsf{dom}$, is the set of all domain values that occur in D.

An *R-relation*, for a relation symbol R, is a finite set of tuples $\bar{a}$ of domain values with $|\bar{a}| = \mathsf{ar}(R)$. A database D over a schema $\mathcal{S}$ induces, for each $R \in \mathcal{S}$, an R-relation $D(R)$, namely $D(R) = \{\bar{a} \mid R(\bar{a}) \in D\}$. These relations $D(R)$ are called *database relations* (of D). We usually write R instead of $D(R)$ for a relation if D is understood from the context. That is, we use the same notation for relations and relation symbols.

The Named Perspective. Let att be an infinite set of *attributes* that is disjoint with the set dom of domain values, and equipped with a linear order. Since we are formalizing the same concepts as before, just from another perspective, we overload some notation accordingly.

A *named relation schema* is represented by a relation symbol R which is associated with a finite set $\mathtt{attr}(R) \subsetneq \mathsf{att}$ of attributes. The *arity* of R is $\mathsf{ar}(R) = |\mathtt{attr}(R)|$. A *named database schema* $\mathcal{S}$ is a set of finitely many named relation schemas, and its arity is $\max_{R \in \mathcal{S}} \mathsf{ar}(R)$.

A *named tuple* $\tilde{a}$ over a finite set $\mathcal{X} \subsetneq \mathsf{att}$ of attributes is a function $\tilde{a} \colon \mathcal{X} \to \mathsf{dom}$, and a *named fact* $R(\tilde{a})$ consists of a relation symbol (from a named schema) and a named tuple $\tilde{a}$ over $\mathtt{attr}(R)$. The arity of a named tuple $\tilde{a}$ over $\mathcal{X}$ is $|\mathcal{X}|$. A *database* over a named schema $\mathcal{S}$ is a finite set of named facts over $\mathcal{S}$. For every $R \in \mathcal{S}$, a database D over $\mathcal{S}$ induces the R-relation $D(R) = \{\tilde{a} \mid R(\tilde{a}) \in D\}$ over $\mathtt{attr}(R)$. Here a relation over a set $\mathcal{X}$ of attributes is a set of tuples over $\mathcal{X}$. Like their unnamed counterparts, we call the relations $D(R)$ the *database relation* (of D), and just write R instead of $D(R)$. The *active domain* $\mathsf{adom}(D)$ of a database D is the set of all domain values that occur in D, i.e. in the range of any named tuple that occurs as part of a fact in D.

For a named tuple over $\mathcal{X}$ and a subset or sequence $\mathcal{Y}$ of attributes in $\mathcal{X}$, we write $\tilde{a}[\mathcal{Y}]$ for the restriction of $\tilde{a}$ to $\mathcal{Y}$, that is, the named tuple $\tilde{b} \colon \mathcal{Y} \to \mathsf{dom}$ with $\tilde{b}(Y) = \tilde{a}(Y)$, for all $Y \in \mathcal{Y}$. This notation naturally extends to relations: For a relation R and $\mathcal{Y} \subseteq \mathtt{attr}(R)$, we define $R[\mathcal{Y}] = \{\tilde{a}[\mathcal{Y}] \mid \tilde{a} \in R\}$. If $\mathcal{Y}$ is a singleton set $\{Y\}$, we just write $\tilde{a}[Y]$ and $R[Y]$, instead of $\tilde{a}[\{Y\}]$ and $R[\{Y\}]$, respectively.

Thanks to the linear order on att, we can refer to the j-th attribute of a relation symbol R. To be a bit more precise, we agree that the first attribute of R is the smallest element of $\mathtt{attr}(R)$ with respect to the linear order on att, and the $\mathsf{ar}(R)$-th attribute of R is the largest element in $\mathtt{attr}(R)$. Of course, this also applies to any finite set $\mathcal{X} \subsetneq \mathsf{att}$ of attributes. Thus, we can identify each named tuple $\tilde{a}$ over a set $\mathcal{X}$ of attributes with a tuple $\bar{a} = (a_1, \ldots, a_{|\mathcal{X}|})$, where a_j is the value assigned to the j-th in $\mathcal{X}$ by $\tilde{a}$. This allows us, in particular, to switch to the unnamed perspective (and back) in a straightforward manner, and to interpret (database) relations as sets of (unnamed) tuples.

For more details on these two perspectives and their relationship we refer to the discussion of Arenas et al. [Are+21, Section 2, "The Relational Model"].

2.2 Query Basics

In general, we define a *query* over a (named or unnamed) schema $\mathcal{S}$ as a function that maps databases over $\mathcal{S}$ to databases over another schema $\mathcal{S}'$. If not explicitly noted otherwise, we will require that the schemas $\mathcal{S}$ and $\mathcal{S}'$ are disjoint. For a query Q and a database D we call the image $Q(D)$ the *query result* of Q over D.[1]

Like functions in general, queries can be composed: Given a query Q over schema $\mathcal{S}$, that maps databases over $\mathcal{S}$ to databases over a schema $\mathcal{S}'$, and a query Q' over schema $\mathcal{S}'$, we write $Q' \circ Q$ for the query over schema $\mathcal{S}$ defined by $D \mapsto Q'(Q(D))$.

Queries over the same schema can be compared with respect to the query results they define. Let Q_1 and Q_2 be queries over the same schema. We say that Q_1 is *contained* in Q_2 and write $Q_1 \sqsubseteq Q_2$ if $Q_1(D) \subseteq Q_2(D)$ holds for every database D. We say that Q_1 and Q_2 are *equivalent* and write $Q_1 \equiv Q_2$ if $Q_1 \sqsubseteq Q_2$ and $Q_2 \sqsubseteq Q_1$ hold. We will often employ the containment problem – for various classes of queries. For classes $\mathbb{Q}_1$ and $\mathbb{Q}_2$ of queries, the *containment problem* $\textsc{Cont}(\mathbb{Q}_1, \mathbb{Q}_2)$ is defined as follows.

— $\textsc{Cont}(\mathbb{Q}_1, \mathbb{Q}_2)$ —

Given: Queries $Q_1 \in \mathbb{Q}_1$ and $Q_2 \in \mathbb{Q}_2$

Question: Is Q_1 contained in Q_2? That is, does $Q_1 \sqsubseteq Q_2$ hold?

A query is *monotone* if $Q(D_1) \subseteq Q(D_2)$ holds whenever $D_1 \subseteq D_2$ holds, for all databases D_1, D_2.

Queries are usually defined syntactically in terms of query languages. In the following we will briefly introduce the query languages relevant for this thesis.

2.3 Relational Algebra

A query of the *(named) relational algebra* over a named schema $\mathcal{S}$ is an expression over a set of attributes inductively defined as follows.

- Each relation symbol $R \in \mathcal{S}$ is a relational algebra expression over $\texttt{attr}(R)$.
- If $\mathcal{E}_1$ and $\mathcal{E}_2$ are relational algebra expressions over sets $\mathcal{X}$ and $\mathcal{Y}$, respectively, then $(\mathcal{E}_1 \bowtie \mathcal{E}_2)$ is a relational algebra expression over $\mathcal{X} \cup \mathcal{Y}$. (*join*)
- If $\mathcal{E}_1$ and $\mathcal{E}_2$ are relational algebra expressions over sets $\mathcal{X}$ and $\mathcal{Y}$, respectively, then $(\mathcal{E}_1 \ltimes \mathcal{E}_2)$ is a relational algebra expression over $\mathcal{X}$. (*semi-join*)
- If $\mathcal{E}_1$ and $\mathcal{E}_2$ are relational algebra expressions over the same set $\mathcal{X}$ of attributes, then $(\mathcal{E}_1 \cup \mathcal{E}_2)$ and $(\mathcal{E}_1 - \mathcal{E}_2)$ are relational algebra expressions over $\mathcal{X}$ as well. (*union* and *difference*)

[1] We note that our definition of queries implies that query results are always finite (databases).

- If $\mathcal{E}$ is a relational algebra expression over a set $\mathcal{X}$ of attributes, then $\pi_{\mathcal{Y}}(\mathcal{E})$ is a relational algebra expression over $\mathcal{Y}$, for every $\mathcal{Y} \subseteq \mathcal{X}$. (*projection*)
- If $\mathcal{E}$ is a relational algebra expression over a set $\mathcal{X}$ of attributes, then $\sigma_{X=Y}(\mathcal{E})$ is a relational algebra expression over $\mathcal{X}$ for any two attributes $X, Y \in \mathcal{X}$. (*selection*)
- If $\mathcal{E}$ is a relational algebra expression over a set $\mathcal{X}$ of attributes, then $\rho_{X \to Y}(\mathcal{E})$ is a relational algebra expression over $(\mathcal{X} \setminus \{X\}) \cup \{Y\}$, for every $X \in \mathcal{X}$ and $Y \in \mathsf{att} \setminus \mathcal{X}$. (*rename*)

As usual, we omit superfluous parentheses. Furthermore, for an attribute X, we just write $\pi_X(\mathcal{E})$ instead of $\pi_{\{X\}}(\mathcal{E})$, and sometimes we write $\pi_{\mathcal{Y}}(\mathcal{E})$ instead of $\pi_{\mathcal{Y} \cap \mathcal{X}}(\mathcal{E})$ for an expression $\mathcal{E}$ over $\mathcal{X}$, for convenience. We call $\bowtie, \ltimes, \cup, -, \pi_{\mathcal{Y}}, \sigma_{X=Y}, \rho_{X \to Y}$ the *operators of the relational algebra*.

Every relational algebra expression $\mathcal{E}$ over a set $\mathcal{X}$ of attributes and schema $\mathcal{S}$ defines a relation $R_{\mathcal{E}}(D)$ over $\mathcal{X}$, for every database D over $\mathcal{S}$, as follows.

- If $\mathcal{E} = R$, for some $R \in \mathcal{S}$, then $R_{\mathcal{E}}(D) = D(R)$.
- If $\mathcal{E} = \mathcal{E}_1 \bowtie \mathcal{E}_2$, for relational algebra expressions $\mathcal{E}_1$ and $\mathcal{E}_2$ over sets $\mathcal{X}$ and $\mathcal{Y}$ of attributes, respectively, then

$$R_{\mathcal{E}}(D) = \big\{\tilde{a} \mid \tilde{a} \text{ is a tuple over } \mathcal{X} \cup \mathcal{Y} \text{ with } \tilde{a}[\mathcal{X}] \in R_{\mathcal{E}_1}(D) \text{ and } \tilde{a}[\mathcal{Y}] \in R_{\mathcal{E}_2}(D)\big\}.$$

- If $\mathcal{E} = \mathcal{E}_1 \ltimes \mathcal{E}_2$, for relational algebra expressions $\mathcal{E}_1$ and $\mathcal{E}_2$ over sets $\mathcal{X}$ and $\mathcal{Y}$ of attributes, respectively, then $R_{\mathcal{E}}(D) = \{\tilde{a} \in R_{\mathcal{E}_1}(D) \mid \tilde{a}[\mathcal{X} \cap \mathcal{Y}] \in R_{\mathcal{E}_2}(D)[\mathcal{X} \cap \mathcal{Y}]\}$.
- If $\mathcal{E} = \mathcal{E}_1 \cup \mathcal{E}_2$, then $R_{\mathcal{E}}(D) = R_{\mathcal{E}_1}(D) \cup R_{\mathcal{E}_2}(D)$.
- If $\mathcal{E} = \mathcal{E}_1 - \mathcal{E}_2$, then $R_{\mathcal{E}}(D) = R_{\mathcal{E}_1}(D) \setminus R_{\mathcal{E}_2}(D)$.
- If $\mathcal{E} = \pi_{\mathcal{Y}}(\mathcal{E}')$, then $R_{\mathcal{E}}(D) = R_{\mathcal{E}'}(D)[\mathcal{Y}]$.
- If $\mathcal{E} = \sigma_{X=Y}(\mathcal{E}')$, then $R_{\mathcal{E}}(D) = \{\tilde{a} \in R_{\mathcal{E}'}(D) \mid \tilde{a}[X] = \tilde{a}[Y]\}$.
- If $\mathcal{E} = \rho_{X \to Y}(\mathcal{E}')$, for an relational algebra expression $\mathcal{E}'$ over a set $\mathcal{X}$ of attributes, then

$$R_{\mathcal{E}}(D) = \{\tilde{b} \mid \tilde{b} \text{ is a tuple over } (\mathcal{X} \setminus \{X\}) \cup \{Y\} \text{ with } \tilde{b}[\mathcal{X} \setminus \{X\}] = \tilde{a}[\mathcal{X} \setminus \{X\}] \text{ and } \tilde{b}[Y] = \tilde{a}[X] \text{ for some } \tilde{a} \in R_{\mathcal{E}'}(D)\}.$$

The query Q defined by a relational algebra expression $\mathcal{E}$ is then simply the query that maps every database to the database consisting of the relation defined by $\mathcal{E}$. More formally, $Q(D) = \{R_{\mathcal{E}}(\tilde{a}) \mid \tilde{a} \in R_{\mathcal{E}}(D)\}$, for every database D. We will often identify a query Q of the relational algebra with its defining expression $\mathcal{E}$. For instance, we write $Q = Q_1 \bowtie Q_2$ for queries Q_1 and Q_2 and $R_Q(D)$ instead of $R_{\mathcal{E}}(D)$. In the same spirit, we write $\mathcal{E}$ for the relation $R_{\mathcal{E}}(D)$, if D is understood from the context.

The *semi-join algebra* is the fragment of the relational algebra without the join operator. That is, queries and expressions of the semi-join algebra are defined as for the relational algebra, but *no* subexpression is of the form $\mathcal{E}_1 \bowtie \mathcal{E}_2$.

For more details on the relational algebra and related terms, we refer to the book of Arenas et al. [Are+21, Chapter 4].

2.4 Rule-Based Query Languages

In this section we will introduce our notation for and recall the basic concepts of conjunctive queries and Datalog queries. Both query languages are defined in terms of (query) rules and the unnamed perspective. Therefore, we will first discuss rules and related terminology, and then, based upon this, the query languages.

Let var be an infinite set of *variables* that is disjoint with the sets dom of domain values, and att of attributes.

Atoms. An *atom* of arity r is of the form $R(x_1, \ldots, x_r)$ with a relation symbol R of arity r, and variable set $\{x_1, \ldots, x_r\} \subset \mathsf{var}$. We denote the variable set of an atom A by $\mathsf{vars}(A)$. Similarly, for a set $\mathcal{A}$ we write $\mathsf{vars}(\mathcal{A})$ for the set of variables occurring in atoms in $\mathcal{A}$. That is, we have $\mathsf{vars}(\mathcal{A}) = \bigcup_{A \in \mathcal{A}} \mathsf{vars}(A)$. Analogously to facts, an atom with relation symbol R is called an R-atom if we want to stress the associated relation (symbol). More generally, $\mathcal{S}$-atoms are R-atoms for some symbol R in schema $\mathcal{S}$.

Like for sets and tuples, we also use the natural extension of mappings of variables on atoms without difference in notation: For a mapping $f\colon \mathsf{var} \to Y$ and an atom $A = R(x_1, \ldots, x_r)$, the image $f(A)$ is $R(f(x_1), \ldots, f(x_r))$.

A *valuation* is a partial mapping $\vartheta\colon \mathsf{var} \rightharpoonup \mathsf{dom}$. A database D *satisfies* a set $\mathcal{A}$ of atoms under a valuation ϑ, if $\vartheta(\mathcal{A}) \subseteq D$, that is, for every atom $R(x_1, \ldots, x_r)$ in $\mathcal{A}$, the fact $R(\vartheta(x_1), \ldots, \vartheta(x_r))$ is contained in D.

We define the *size* $\|A\|$ of an atom $A = R(\bar{x})$ as $\mathsf{ar}(R) + 1$. Note that this ensures that atoms $R()$ of arity 0 have a strictly positive size.

Query Rules. A (query) *rule* τ represents a conjunction of atoms and has the form

$$A \leftarrow A_1, \ldots, A_m$$

where $m \geq 1$ is an integer and $A, A_1, \ldots, A_m$ are atoms. The atom A is called the *head* of τ and the set $\{A_1, \ldots, A_m\}$ is called the *body* of τ. We denote the head and body of a rule τ by $\mathsf{head}(\tau)$ and $\mathsf{body}(\tau)$, respectively. By $\mathsf{vars}(\tau)$ we denote the set of variables occurring in τ. That is, $\mathsf{vars}(\tau) = \mathsf{vars}(\mathsf{head}(\tau)) \cup \mathsf{vars}(\mathsf{body}(\tau))$.

A rule τ is *safe* if every variable in its head also occurs in at least one atom of the body, i.e., $\mathsf{vars}(\mathsf{head}(\tau)) \subseteq \mathsf{vars}(\mathsf{body}(\tau))$ holds. A rule τ is *recursive* if the relation symbol of its head also occurs in $\mathsf{body}(\tau)$. More formally, if $\mathsf{head}(\tau) = H(\bar{x})$ for some relation symbol H then $\mathsf{body}(\tau)$ contains an H-atom.

We define the *size* $\|\tau\|$ of a rule τ as $\|\mathsf{head}(\tau)\| + \sum_{A \in \mathsf{body}(\tau)} \|A\|$, and its *length* $|\tau|$ as $|\mathsf{body}(\tau)| + 1$.

2.4.1 Conjunctive Queries

A *conjunctive query* Q over a schema $\mathcal{S}$ is defined by a safe rule τ that satisfies the following condition: The atoms in $\mathsf{body}(\tau)$ are $\mathcal{S}$-atoms and the atom $\mathsf{head}(\tau)$, on the contrary, is not. Since a conjunctive query Q is defined by exactly one rule τ, we usually

identify Q with τ. In particular, we write $\mathsf{head}(Q)$, $\mathsf{body}(Q)$, etc. instead of $\mathsf{head}(\tau)$, $\mathsf{body}(\tau)$, etc., and we have $\|Q\| = \|\tau\|$ as well as $|\tau| = |\tau|$.

Variables that occur in the head of a conjunctive query are called *head variables*; all other variables are called *quantified variables*. A conjunctive query without quantified variables is called a *full* conjunctive query. Conjunctive queries with two or more body atoms that refer to the same relation are said to have *self-joins*. The *arity* of a conjunctive query Q is the arity of its head. Furthermore, a conjunctive query Q is a *Boolean* query if its arity is 0.

The *query result* of a conjunctive query Q over a database D is defined as

$$Q(D) = \{\vartheta(\mathsf{head}(Q)) \mid \vartheta \text{ is a valuation and } D \text{ satisfies } \mathsf{body}(Q) \text{ under } \vartheta\}.$$

We note that conjunctive queries are monotone [cf., e.g., AHV95, Proposition 4.2.2].

Example 2.4.1. Consider the conjunctive query Q defined by the rule

$$H(x, y, x) \leftarrow R(x, z), R(y, z), S(x, y, z).$$

Its head variables are x and y; the variable z is a quantified variable. The arity of Q is 3.

Let $D = \{R(1, 2), R(3, 2), R(4, 2), S(1, 3, 2), S(4, 1, 2)\}$ be a database over the schema $\{R, S\}$. The query result of Q over D is the H-relation

$$Q(D) = \{H(1, 3, 1), H(4, 1, 4)\}. \qquad \triangleleft$$

Homomorphisms, Containment, and Minimality. It is well-known that a conjunctive query Q_1 is contained in a conjunctive query Q_2 if and only if there is a homomorphism from the latter query into the first [cf. CM77, Proof of Lemma 13]. Such a *homomorphism* is a mapping $h\colon \mathsf{vars}(Q_2) \to \mathsf{vars}(Q_1)$ such that

(a) $h(\mathsf{body}(Q_2)) \subseteq \mathsf{body}(Q_1)$ and

(b) $h(\mathsf{head}(Q_2)) = \mathsf{head}(Q_1)$ hold.

We call h a *body homomorphism* if it fulfils Condition (a).

A conjunctive query Q_1 is *minimal* if there is *no* conjunctive query Q_2 such that $Q_2 \equiv Q_1$ and $|\mathsf{body}(Q_2)| < |\mathsf{body}(Q_1)|$ holds.

Structurally Simple Conjunctive Queries. Despite their simplicity and restricted expressiveness, several interesting problems are intractable for conjunctive queries in general. Therefore, different fragments have been studied in the literature. In this thesis, we are particularly concerned with "acyclic" conjunctive queries, which allow, for instance, evaluation in polynomial time. We also consider three subclasses of acyclic conjunctive queries, namely, free-connex acyclic, hierarchical, and q-hierarchical queries.

Acyclic and Free-Connex Acyclic Queries. A *join tree* for a conjunctive query Q is a tree T_Q whose nodes are the atoms in the query's body and that satisfies the following path property: For every two atoms $A, A' \in \mathsf{body}(Q)$ with a common variable x, all atoms on the (shortest) path from A to A' contain x. A conjunctive query Q is *acyclic* if it has a join tree. It is *free-connex acyclic* if Q is acyclic and the Boolean query whose body is $\mathsf{body}(Q) \cup \{\mathsf{head}(Q)\}$ is acyclic as well [BDG07; Bra13].

Hierarchical Queries. For a fixed conjunctive query Q and some variable x that occurs in Q, let $\mathsf{atoms}(x)$ denote the set of atoms in $\mathsf{body}(Q)$ in which x appears.

Definition 2.4.2 [DS07], [BKS17b, Definition 3.1]**.** A conjunctive query Q is *hierarchical* if, for all variables x, y in $\mathsf{vars}(Q)$, one of the following conditions is satisfied.

(1) $\mathsf{atoms}(x) \subseteq \mathsf{atoms}(y)$

(2) $\mathsf{atoms}(x) \supseteq \mathsf{atoms}(y)$

(3) $\mathsf{atoms}(x) \cap \mathsf{atoms}(y) = \emptyset$

Thus, the "hierarchy" is established by the query's variables and the sets of atoms that contain them.

A conjunctive query Q is *q-hierarchical* (short for *quantified-hierarchical*) if it is hierarchical and for all variables $x, y \in \mathsf{vars}(Q)$ the following is satisfied.

(4) If $\mathsf{atoms}(x) \subsetneq \mathsf{atoms}(y)$ holds and x is a head variable of Q, then y is also a head variable of Q.

For brevity, we denote by CQ, ACQ, CCQ, HCQ, and QHCQ the classes of conjunctive queries in general and those conjunctive queries that are acyclic, free-connex acyclic, hierarchical or q-hierarchical, respectively (cf. Table 2.1(a)). The relationships of these classes are depicted in Figure 2.1. We note that, in particular, each q-hierarchical conjunctive query is free-connex acyclic and each hierarchical conjunctive query is acyclic.

Proposition 2.4.3 (various sources, see discussion below)**.** *The inclusions depicted in Figure 2.1 hold.*

The inclusions $\mathsf{QHCQ} \subseteq \mathsf{HCQ}$, $\mathsf{CCQ} \subseteq \mathsf{ACQ}$, and $\mathsf{ACQ} \subseteq \mathsf{CQ}$ hold by definition. Idris et al. [IUV17, Proposition 4.25] proved the inclusion $\mathsf{QHCQ} \subseteq \mathsf{CCQ}$. The inclusion $\mathsf{HCQ} \subseteq \mathsf{ACQ}$ is mentioned by, e.g., Hu and Yi [HY19] and Kara et al. [Kar+20]; it also follows readily from the former inclusion: For acyclicity the head of a query is of no concern, and the Boolean variant of a hierarchical conjunctive query is always q-hierarchical by definition, and, thus, free-connex acyclic which implies the existence of a join tree for the body of the original query.

The strictness of the inclusions can easily be verified by (classical) examples.

$\mathsf{ACQ} \subsetneq \mathsf{CQ}$: The triangle query $T() \leftarrow E(x, y), E(y, z), E(z, x)$ is not acyclic.

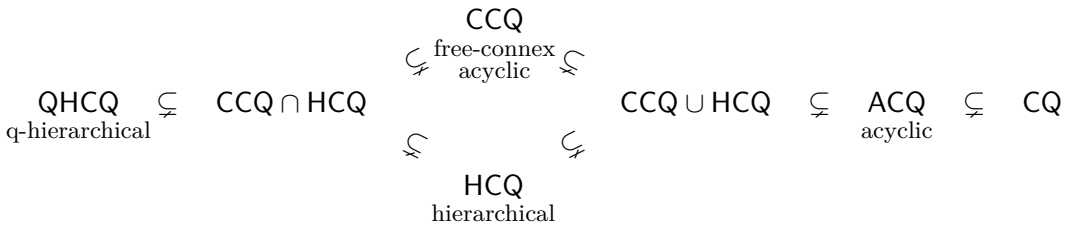

Figure 2.1: Relationships of subclasses of the class CQ of conjunctive queries.

query class	short description
CQ	conjunctive queries
ACQ	acyclic queries
CCQ	free-connex acyclic queries
HCQ	hierarchical queries
QHCQ	q-hierarchical queries

(a) Classes of conjunctive queries.

query class	short description
DL	Datalog queries
MDL	monadic queries
FGDL	frontier-guarded queries

(b) Classes of Datalog queries.

Table 2.1: Overview of rule-based query classes.

CCQ ∪ HCQ ⊊ ACQ: The path query $P_3(x,u) \leftarrow E(x,y), E(y,z), E(z,u)$ is acyclic but neither hierarchical nor free-connex acyclic because the sets $\mathsf{atoms}(y)$ and $\mathsf{atoms}(z)$ are incomparable and not disjoint, and the atoms $P_3(x,u), E(x,y), E(y,z), E(z,u)$ form a circle, respectively.

HCQ ⊊ HCQ ∪ CCQ and HCQ ∩ CCQ ⊊ CCQ: The query $H() \leftarrow S(x), R(x,y), S(y)$ is free-connex acyclic but not hierarchical, since the sets $\mathsf{atoms}(x)$ and $\mathsf{atoms}(y)$ are incomparable and not disjoint.

CCQ ⊊ HCQ ∪ CCQ and HCQ ∩ CCQ ⊊ HCQ: The query $P_2(x,z) \leftarrow E(x,y), E(y,z)$ is hierarchical but not free-connex acyclic, because the atoms $P_2(x,z), E(x,y), E(y,z)$ form a circle.

QHCQ ⊊ HCQ ∩ CCQ: The query $H(x) \leftarrow R(x,y), S(y)$ is free-connex acyclic and hierarchical but not q-hierarchical because $\mathsf{atoms}(x) \subsetneq \mathsf{atoms}(y)$ holds and x is a head variable, but y is not.

Finally, observe that for full and Boolean conjunctive queries the picture becomes simpler: Every full acyclic conjunctive query is free-connex acyclic, and every full hierarchical conjunctive query is q-hierarchical. The same is true for Boolean queries.

Generalized Hypertree Decompositions. In Chapter 3 we will employ a generalization of acyclic conjunctive queries, which we define next.

A *tree decomposition* of a conjunctive query Q is an undirected, rooted tree T, where each node v is associated with a set $\mathsf{bag}_T(v) \subseteq \mathsf{vars}(Q)$ of variables from Q such that the following three conditions are satisfied.

(a) For each variable $x \in \mathsf{vars}(Q)$ there is a node $v \in \mathsf{nodes}(T)$ with $x \in \mathsf{bag}_T(v)$.[2]

(b) For each atom $A \in \mathsf{body}(Q)$ there is a node $v \in \mathsf{nodes}(T)$ with $\mathsf{vars}(A) \subseteq \mathsf{bag}_T(v)$.

(c) For each variable $x \in \mathsf{vars}(Q)$ the set $\{v \in \mathsf{nodes}(T) \mid x \in \mathsf{bag}_T(v)\}$ induces a connected subtree of T.

A *generalized hypertree decomposition* of a conjunctive query Q is tree decomposition T where each node $v \in \mathsf{nodes}(T)$ is additionally associated with a set $\mathsf{cover}_T(v) \subseteq \mathsf{body}(Q)$ of body atoms such that $\mathsf{bag}_T(v) \subseteq \bigcup_{A \in \mathsf{cover}_T(v)} \mathsf{vars}(A)$ holds.

The *width* of a generalized hypertree decomposition T is $\max_{v \in \mathsf{nodes}(T)} |\mathsf{cover}_T(v)|$. The *generalized hypertree width* of a conjunctive query Q is the *minimal* width over all of its generalized hypertree decompositions. We point out that a conjunctive query is acyclic, if and only if it has generalized hypertree width 1 [GLS02, Theorem 4.5].[3]

Following Gottlob et al. [GLS02, Definition 4.2] we say that a generalized hypertree decomposition for a query Q is *complete* if, for every atom $A \in \mathsf{body}(Q)$, there is a node v such that $A \in \mathsf{cover}_T(v)$. Given any generalized hypertree decomposition, it can be transformed into a complete generalized hypertree decomposition with the same width.

Lemma 2.4.4 [GLS02, Lemma 4.4]**.** *For every conjunctive query Q with generalized hypertree width k, there is a complete generalized hypertree decomposition of width k.*

In a nutshell, a complete decomposition can be obtained by creating, for every atom A a new node v with $\mathsf{cover}_T(v) = \{A\}$ and $\mathsf{bag}_T(v) = \mathsf{vars}(A)$, and attaching it to a node w with $\mathsf{bag}_T(w) \supseteq \mathsf{vars}(A)$. Such a node w always exists thanks to Condition (b). For more details we refer to the proof given by Gottlob et al. [GLS02, Lemma 4.4].

For more details on generalized hypertree decompositions and related notions in general we refer to the question and answer article of Gottlob et al. [Got+16].

Free-Connex Conjunctive Queries. The notion of generalized hypertree decompositions also gives rise to a generalization of free-connex acyclic conjunctive queries.[4]

A generalized hypertree decomposition T of a conjunctive query Q is *free-connex* if there is a set of nodes $U \subseteq \mathsf{nodes}(v)$ that induces a connected subtree in T and satisfies $\mathsf{vars}(\mathsf{head}(Q)) = \bigcup_{w \in U} \mathsf{bag}_T(w)$. The *free-connex generalized hypertree width* of a conjunctive query Q is the *minimal* width among its free-connex generalized hypertree decompositions. We will sometimes use the following result. A detailed proof of it has been given by Berkholz et al. [BGS20, Theorem 5.2].

[2] We note that is condition is actually redundant, because we defined conjunctive queries in terms of *safe* rules, and thanks to Condition (b). We include it here to avoid confusion.

[3] We note that the cited proof [GLS02] is for hypertree decompositions which impose an additional condition in comparison to generalized hypertree decompositions. However, it is *not* used in the proof.

[4] We note that such a generalization can be defined on top of various other notions [cf. BGS20], and originally free-connex acyclic conjunctive queries were defined in terms of tree decompositions by Bagan et al. [BDG07, Definition 36].

Proposition 2.4.5 [Bra13, p. 82, Footnote 10]**.** *A conjunctive query is free-connex acyclic if and only if it has free-connex generalized hypertree width* 1*.*

2.4.2 Datalog

A *Datalog program* P is a finite set of Datalog rules. Here a *Datalog rule* is just a safe rule.[5]

We call a relation symbol occurring in P an *extensional* relation symbol if it occurs only in the body of rules in P. All other relation symbols occurring in P, i.e. those which are not extensional, we refer to as the *intensional* relations symbols of P. By $\mathsf{edb}(P)$ and $\mathsf{idb}(P)$, we denote the schemas induced by the extensional and intensional relation symbols of P, respectively. We extend these terms to facts and atoms in the natural way: A fact (or atom) is extensional if its relation symbol is extensional and intensional otherwise.

A *Datalog query* Q over a schema $\mathcal{S}$ is a pair $Q = (P, \mathrm{Out})$ where P is a Datalog program and $\mathrm{Out} \notin \mathcal{S}$ is a relation symbol such that (1) $\mathsf{edb}(P) \subseteq \mathcal{S}$; (2) $\mathsf{idb}(P) \cap \mathcal{S} = \emptyset$; and (3) $\mathrm{Out} \in \mathsf{idb}(P)$. That is, all extensional relation symbols are in $\mathcal{S}$ while intensional relation symbols are not in $\mathcal{S}$. We call the relation symbol Out the *output symbol* of Q. Moreover, we define the *size* $\|Q\|$ of Q as $\|P\| + 1$ where $\|P\| = \sum_{\tau \in P} \|\tau\|$.

Semantics. Let $Q = (P, \mathrm{Out})$ be a Datalog query and D be a database over some schema $\mathcal{S}$. We say that a fact $R(\bar{a})$ *can be derived* by a rule $\tau \in P$ and a valuation ϑ from D if $R(\bar{a}) = \vartheta(\mathsf{head}(\tau))$ and D satisfies $\mathsf{body}(\tau)$ under ϑ. The *immediate consequence operator* for P, denoted c_P, maps databases over the schema $\mathcal{S} \cup \mathsf{idb}(P)$ to databases over $\mathcal{S} \cup \mathsf{idb}(P)$. It is defined by

$$c_P(D') = D' \cup \{\vartheta(\mathsf{head}(\tau)) \mid \tau \in P, \vartheta \text{ is a valuation, and } D' \text{ satisfies } \mathsf{body}(\tau) \text{ under } \vartheta\}.$$

Then the output $P(D)$ *computed by* P *over* D is defined as the (least) fixpoint that is reached after iteratively applying c_P to D. For a fact $R(\bar{a}) \in P(D)$ we also say that $R(\bar{a})$ *can be derived* by P. The immediate consequence operator is monotone [AHV95, Lemma 12.3.1] and, hence, $P(D)$ always exists and is well-defined [AHV95, Theorem 12.3.2].

The *query result* of $Q = (P, \mathrm{Out})$ over D is then the Out-relation of $P(D)$. More formally,

$$Q(D) = \{\mathrm{Out}(\bar{a}) \mid \mathrm{Out}(\bar{a}) \in P(D)\}.$$

Since the immediate consequence operator is monotone, Q is monotone as well. We refer to Abiteboul et al. [AHV95] and Arenas et al. [Are+21] for more details on Datalog.

Let us point out that every conjunctive query Q' constitutes an equivalent Datalog query Q over the same schema: Q consists of a Datalog program with a single rule, namely the rule defining Q', and output symbol is the relation symbol of $\mathsf{head}(Q')$. But not every Datalog rule defines a conjunctive query. In particular, Datalog rules can be recursive while conjunctive queries do *not* allow for recursion.

[5] We call them Datalog rules to emphasize they do not necessarily define a conjunctive query.

Example 2.4.6. Let atoms of the form $E_r(x, y)$ denote red edges and $E_s(x, y)$ denote sea blue edges. The Datalog query $Q = (P, \mathrm{Out})$ where P consists of the rules below asks for all nodes reachable from a starting node x (indicated by the atom $\mathrm{Start}(x)$) by a path containing only red and a path containing only sea blue edges.

$$R(x) \leftarrow \mathrm{Start}(x) \qquad S(x) \leftarrow \mathrm{Start}(x) \qquad \mathrm{Out}(x) \leftarrow R(x), S(x)$$
$$R(x) \leftarrow R(y), E_r(y, x) \qquad S(x) \leftarrow S(y), E_s(y, x)$$

We have $\mathsf{idb}(P) = \{R, S, \mathrm{Out}\}$ and $\mathsf{edb}(P) = \{E_r, E_s\}$. Let further

$$D = \{\mathrm{Start}(1), E_r(1, 3), E_r(1, 4), E_s(1, 2), E_s(2, 3)\}.$$

Then $Q(D) = \{\mathrm{Out}(1), \mathrm{Out}(3)\}$. ◁

Proof Trees. For conjunctive queries Q' a valuation can serve as a "witness" for a fact $R(\bar{a})$ being in a query result $Q'(D)$. An analogue notion for Datalog queries are *proof trees* [cf., e.g., AHV95; Are+21].

For a rooted tree T, we denote its set of nodes by $\mathsf{nodes}(T)$, and its root node by $\mathsf{root}(T)$. We denote the set of children of a node $v \in \mathsf{nodes}(T)$ by $\mathsf{children}_T(v)$.

Definition 2.4.7 (Proof Tree)**.** A *proof tree* T for a fact $R(\bar{a})$ with respect to a Datalog program P is a rooted tree, in which every node $v \in \mathsf{nodes}(T)$ is labelled with a fact $\mathsf{fact}(v)$ over $\mathsf{edb}(P) \cup \mathsf{idb}(P)$, and which has the following properties.

(a) The root node is labelled $\mathsf{fact}(\mathsf{root}(T)) = R(\bar{a})$.

(b) For every inner node v of T, there is a rule $\tau \in P$ and a valuation ϑ such that $\vartheta(\mathsf{head}(\tau)) = \mathsf{fact}(v)$ and $\vartheta(\mathsf{body}(\tau)) = \{\mathsf{fact}(w) \mid w \in \mathsf{children}_T(v)\}$.

(c) Every leaf v is labelled with an extensional fact $\mathsf{fact}(v)$ over $\mathsf{edb}(P)$.

A *proof tree* T for a fact $\mathrm{Out}(\bar{a})$ with respect to a Datalog query $Q = (P, \mathrm{Out})$ is a proof tree with respect to P. We say that T is a *proof tree with respect to a database* D, if all leaves are labelled with facts from D.

We say that a node v of a proof tree is witnessed by a rule τ and a valuation ϑ as shorthand for τ and ϑ witnessing that v has Property (b).

The following well-known result can be proved by induction on the depth of a proof tree (direction from left to right) and on the number of application of the immediate consequence operator (direction from right to left).

Lemma 2.4.8 [e.g. Are+21, Lemma 38.6]**.** *Let $Q = (P, \mathrm{Out})$ be a Datalog query, D be a database, and $R(\bar{a})$ be a fact. Then there is a proof tree for $R(\bar{a})$ with respect to P and D if and only if $R(\bar{a}) \in P(D)$.*

Structurally Simple Datalog Queries. A Datalog query $Q = (P, \text{Out})$ is *monadic* if the arity of every relation in $\mathsf{idb}(P)$ is at most one. It is *frontier-guarded*, if every rule $\tau \in P$ is *frontier-guarded*, that is, if there is an extensional atom in $\mathsf{body}(\tau)$, called *guard atom*, that contains all the variables from $\mathsf{head}(\tau)$ [see, e.g., BKR15a].

We note that a monadic Datalog query is not necessarily frontier-guarded. It is however possible to transform every monadic Datalog query into an equivalent frontier-guarded Datalog query that is only polynomially larger.

Lemma 2.4.9 [BKR15a, last paragraph p. 2828]**.** *For every monadic Datalog query Q there is an equivalent monadic, frontier-guarded Datalog query Q'. Moreover, Q' has size polynomial in $\|Q\|$.*

We denote by DL, MDL, and FGDL, the classes of Datalog queries in general and those Datalog queries that are monadic, or frontier-guarded, respectively (see Table 2.1(b) for an overview).

Example 2.4.10 (Continuation of Example 2.4.6)**.** The Datalog query $Q = (P, \text{Out})$ from Example 2.4.6 is monadic because all intensional relation symbols have arity 1. It is, however, *not* frontier-guarded: The rule $\text{Out}(x) \leftarrow R(x), S(x)$ does not contain any extensional atom and, thus, cannot be frontier-guarded.

Replacing $\text{Out}(x) \leftarrow R(x), S(x)$ with the rules

$$\text{Out}(x) \leftarrow \text{Start}(x), R(x), S(x) \quad \text{and} \quad \text{Out}(x) \leftarrow E_r(y, x), R(x), S(x)$$

yields an equivalent frontier-guarded Datalog query. ◁

2.5 Automata and Machine Models for Upper and Lower Bound Proofs

In this section we provide definitions and some basic facts on machine and automata models, which we will use to prove some upper and lower bounds in Chapter 4. It is also viable to skip this section for now. Back references are given when these models become relevant.

2.5.1 Minsky Machines

A (deterministic) *Minsky machine* $M = (S, s_0, s_f)$ consists of a set S of states, an *initial state* $s_0 \in S$ and a *halting state* $s_f \in S$. Moreover, the machine has two counters numbered 1 and 2. Each state $s \in S$ – except the halting state s_f – is equipped with a specific *instruction* which has one of the following forms.

- $\text{Inc}(i, s')$ which increments counter i and then moves to state s'; or,
- $\text{Dec}(i, s', s'')$ which, if counter i has a non-zero value, decrements it and moves to state s', or else moves to state s''.

A *configuration* is a tuple (s, c_1, c_2) where s is the current state and c_i is the value of counter i. A configuration (s', c_1', c_2') *succeeds* a configuration (s, c_1, c_2) if the former results from the latter in the natural fashion by applying the instruction associated with s. *No* configuration succeeds a configuration (s_f, c_1, c_2) with the halting state. The unique *computation* of M is the (possibly infinite) sequence of configurations where

- the first configuration in the sequence is the *initial configuration* $(s_0, 0, 0)$; and
- the $(i+1)$-th configuration succeeds the i-th configuration, for all (applicable) $i \geq 1$.

A Minsky machine M *halts* if the halting state s_f occurs in its computation (if s_f does occur then the computation is finite and s_f is the state of the last configuration).

In Section 4.2.1 we will utilize the following decision problem as well as the associated undecidability result.

— MinskyHalt —

Given: Minsky machine M

Question: Does M halt?

Proposition 2.5.1 [Min61, Theorem Ia]**.** MinskyHalt *is undecidable.*

For more details on Minsky machines we refer to the primordial work of Minsky [Min61], and to the survey of Sapir [Sap15].

2.5.2 Two-Way Alternating Tree Automata

In Sections 4.3 and 4.4 we will utilize alternating two-way tree automata. They recognize tree languages; that is, sets of finite trees over a ranked alphabet. A *ranked alphabet* Γ is a disjoint union $\Gamma_k \cup \ldots \cup \Gamma_0$ of finitely many finite sets Γ_i. We require Γ_0 to be non-empty (any other Γ_i may be empty). An element of Γ is called a *letter* and it has *rank* i if it is in Γ_i. Since we consider no other kind of alphabet, we will usually just use the term "alphabet".

A *tree over* Γ is a finite, rooted, and ordered tree whose nodes are labelled with letters from Γ such that, for every node, its label is in Γ_i if and only if it has exactly i children. By $\mathcal{T}_\Gamma$ we denote the set of all trees over Γ. A *tree language over* Γ is a subset of $\mathcal{T}_\Gamma$.

For a set X, whose elements we consider to be propositions, we denote by $\mathcal{L}^+(X)$ the set of positive propositional formulas over X and the set $\{\vee, \wedge\}$ of operations. In particular, the formula $\top$ (tautology) is always contained in $\mathcal{L}^+(X)$.

Definition 2.5.2 (Alternating Two-Way Tree Automaton)**.** An *alternating two-way tree automaton* $\mathbb{A}$ is a tuple (S, Γ, ρ, s_0) where

- S is a (finite) set of states;
- $\Gamma = \Gamma_k \cup \ldots \cup \Gamma_1 \cup \Gamma_0$ is a (ranked) alphabet;
- $\rho\colon S \times \Gamma \to \mathcal{L}^+(\{-1, 0, \ldots, i\} \times S)$ is the transition function which maps, for all $i \in [0, k]$, each pair from $S \times \Gamma_i$ to a formula in $\mathcal{L}^+(\{-1, 0, \ldots, i\} \times S)$; and

- $s_0 \in S$ is the initial state.

Following Colcombet and Löding [CL10] we define the semantics of $\mathbb{A}$ on a tree $T \in \mathcal{T}_\Gamma$ in terms of a two-player game denoted by $\mathbb{A} \times T$. The *positions* of the game consist of a state s of the automaton and a node v of the tree T. The *initial position* is $(s_0, \mathsf{root}(T))$. The players are Morgana (the existential player) and Arthur (the universal player). A *play* is a sequence $\pi = (s_0, v_0), (s_1, v_1), \ldots$ of positions where $(s_0, v_0) = (s_0, \mathsf{root}(T))$ is the initial position and the successor of (s_i, v_i) is determined in round $i + 1$ of the game as follows. Let a_i be the label of v_i.

(1) If $\rho(s_i, a_i) = \top$ the game ends. That is, (s_i, v_i) has no successor in π.

(2) Morgana and Arthur determine a proposition from $\varphi = \rho(s_i, a_i)$ by playing a "subgame" on φ: If $\varphi = \psi_1 \wedge \psi_2$, Arthur chooses either ψ_1 or ψ_2; if $\varphi = \psi_1 \vee \psi_2$, Morgana chooses. The subgame continues with the chosen subformula. In that fashion Arthur and Morgana will eventually select a proposition (d, s).

(3) The state of the next *position* is then $s_{i+1} = s$. The next node v_{i+1} is **(i)** the parent node of v_i if $d = -1$; **(ii)** v_i if $d = 0$; or **(iii)** the d-th child of v_i if $d > 0$.

If the position is of the form $(s_i, \mathsf{root}(T))$ – this includes, in particular, the initial position – and a proposition (d, s) with $d = -1$ is selected, the game ends. Morgana *wins the play* π if it ends because $\rho(s_i, a_i) = \top$ holds for the last position in π. Otherwise, she looses and Arthur wins. We can always assume that a play is finite: If a positions repeats, Morgana can either win with fewer moves or she cannot win at all. In other words, if $(s_i, v_i) = (s_j, v_j)$ for some $j < i$, we agree that the game ends (and Morgana looses).

A *partial play* is a strict, finite prefix of a play. A *strategy for Morgana* is function which maps partial plays to moves of Morgana. More precisely, it maps every pair $(\pi, \varphi_1 \ldots \varphi_\ell)$ with $\varphi_\ell = \psi_1 \vee \psi_2$ to either ψ_1 or ψ_2. Here π is a partial play of the game and $\varphi_1 \ldots \varphi_\ell$ is a partial play of the subgame, i.e. φ_j is a conjunction or disjunction of φ_{j+1} with another formula, for all $j \in [1, \ell]$. *Strategies for Arthur* are defined analogously. Together, a strategy for Morgana and a strategy Arthur uniquely determine a play. A *winning strategy* for Morgana is a strategy of hers such that, no matter the strategy of Arthur, Morgana wins the resulting play.

An alternating two-way tree automaton $\mathbb{A}$ *accepts* a tree T if Morgana has a winning strategy for the game $\mathbb{A} \times T$. The tree language *recognized by* $\mathbb{A}$ is

$$\mathcal{T}_\mathbb{A} = \{T \in \mathcal{T}_\Gamma \mid \mathbb{A} \text{ accepts } T\}.$$

Cost Automata. Cost automata are an extension of "classical" automata. Instead of just accepting (or rejecting) trees they also assign a *cost* to trees. For that purpose they are equipped with *counters* which can be incremented, checked, and reset (but, unlike for Minsky machines, there is no decrement operation). In this thesis we only require – and, hence, define – a very simple kind of cost automata: Namely, automata with only

one counter that can only be incremented (and is implicitly checked).[6] We refer to the articles of Colcombet and Löding [CL10], and Benedikt et al. [Ben+15] for more general definitions which allow for more counters as well as more variations of cost assignments and acceptance conditions.

Our definition of alternating two-way tree cost automata is almost the same as for alternating two-way tree automata. The only difference is that every proposition in a transition formula comes with an *action*: Either increment the counter (Inc) or leave the counter value unchanged (Noop).

Definition 2.5.3 (Alternating Two-Way Tree Cost Automaton)**.** An *alternating two-way tree cost automaton* $\mathbb{A}$ is a tuple (S, Γ, ρ, s_0) where

- S is a (finite) set of states;
- $\Gamma = \Gamma_k \cup \ldots \cup \Gamma_1 \cup \Gamma_0$ is a (ranked) alphabet;
- $\rho\colon S \times \Gamma \to \mathcal{L}^+(\{-1, 0, \ldots, i\} \times S \times \{\text{Inc}, \text{Noop}\})$ is the transition function which maps, for all $i \in [0, k]$, each pair from $S \times \Gamma_i$ to a formula in

$$\mathcal{L}^+(\{-1, 0, \ldots, i\} \times S \times \{\text{Inc}, \text{Noop}\}); \text{ and}$$

- $s_0 \in S$ is the initial state.

The semantics of our cost automata are also very similar to the semantics of alternating two-way tree automata. Here the positions of a game $\mathbb{A} \times T$ are triples (s, v, c) where s and v are as before the current state of $\mathbb{A}$ and node of T, respectively; and c is the current value of the counter. A play is a sequence $\pi = (s_0, v_0, c_0), (s_1, v_1, c_1), \ldots$ where the initial configuration is $(s_0, v_0, c_0) = (s_0, \mathsf{root}(T), 0)$. The successor of a position (s_i, v_i, c_i) in π is determined as before, but every time Morgana and Arthur select a proposition of the form (d, s, Inc), the counter is incremented, i.e. $c_{i+1} = c_i + 1$. Otherwise, it is left unchanged. The *cost assigned to a play* is ∞ if the game ends because there are i, j with $j < i$ such that $s_i = s_j$ and $v_i = v_j$, that is, if Morgana looses because Arthur could prolong the game indefinitely. Otherwise, the cost is the largest counter value c_i occurring in π.

The notions of (winning) strategies, acceptance, and recognition carry over from alternating two-way tree automata. A *c-winning strategy* of Morgana is a winning strategy such that, no matter the strategy of Arthur, the cost of the resulting play is at most c. The *cost assigned to a tree* $T \in \mathcal{T}_{\mathbb{A}}$ *by* $\mathbb{A}$ is the smallest non-negative integer c such that Morgana has a c-winning strategy for $\mathbb{A} \times T$. In other words, Morgana has the additional objective to keep the counter value as small as possible, while Arthur's objective is to maximize the counter value.

An alternating two-way tree cost automaton $\mathbb{A}$ is *limited* if there is some constant d such that, for each $T \in \mathcal{T}_{\mathbb{A}}$, the cost assigned to T by $\mathbb{A}$ is at most d. That is, if Morgana has a winning strategy, she always has a d-winning strategy. The *limitedness problem* Limitedness for alternating two-way tree cost automaton is defined as follows.

[6] This is also known as "distance automaton" or "automaton with distance objective" in the literature [cf. e.g. Ben+15].

— Limitedness

Given: *Alternating two-way tree cost automaton* $\mathbb{A}$

Question: Is $\mathbb{A}$ limited?

The following result by Benedikt et al. [Ben+15] is crucial for our main results in Section 4.4.

Proposition 2.5.4 [Ben+15, Theorem 12]**.** *The limitedness problem for alternating two-way tree cost automata with a single counter that is never reset, is decidable in exponential time.*

We note that Benedikt et al. [Ben+15] state Proposition 2.5.4 for alternating two-way tree cost automata on infinite, unranked trees, and for the *boundedness* problem. The encoding of finite trees within infinite tree structures is straightforward, and the universal player Arthur can be given the option to claim and prove encoding errors. The boundedness problem asks whether the counter stays below a constant, for any input tree (where the limitedness problem only asks this for input trees which are accepted). By giving the existential player the option to "resign" (in exchange for never increasing the counter), the seemingly stricter condition for boundedness can always be met, if the original automaton is limited.

Chapter 3

Work-Efficient Query Evaluation with Parallel Random Access Machines

In this chapter we investigate work-efficient $\mathcal{O}(1)$-time parallel algorithms for evaluating queries from various query classes. As mentioned in the introduction, we will do so in three settings: The general setting, the ordered setting, and the dictionary setting – where the latter will be our main setting. The main results of this chapter are summarized in Table 3.1. We emphasize that we are concerned with the data complexity of query evaluation in this chapter. In particular, the query itself is never part of the input.

Outline. This chapter is roughly divided into three parts. In the first part we introduce the PRAM models we use in this chapter formally (Section 3.1.1), and discuss some essential lower and upper bound results for these models (Section 3.1.2). We then introduce our three settings in Section 3.2, which differ in how PRAMs can access and store databases and (intermediate) results in their memory.

In the second part we will present algorithms for operations which we will use as building blocks for our constant-time query evaluation algorithms. We will first introduce algorithms for some basic operations, notably including a search operation, in Section 3.3, and based on that algorithms for the relational operators in Section 3.4.

In the third part, we then study $\mathcal{O}(1)$-time parallel algorithms for evaluating queries of the semi-join algebra, (structurally simple) conjunctive queries, and natural join queries. For the latter we present almost worst-case optimal evaluation algorithms. We first present evaluation algorithms in Section 3.5 for our main setting. For the other two settings we will derive evaluation algorithms by translating them into the main setting in Section 3.6.

We conclude this chapter with a discussion of our results and further related literature in Section 3.7.

Publication and Contributions. This chapter is based on a conference paper [KSS23] authored by my advisor Prof. Dr Thomas Schwentick, my colleague Dr Jens Keppeler, and me. To somewhat live up to its title, it has been written in parallel to a full version of the conference paper.[1]

In comparison with the conference paper, this chapter and the full version draw a more complete and improved picture of $\mathcal{O}(1)$-time parallel algorithms for query evaluation. In

[1] My co-authors and I intend to submit it for publication.

Query class	Work bound in the …		
	general setting	ordered setting	dictionary setting
Semijoin Algebra	$\mathcal{O}(\mathsf{IN}^2)$ Theorem 3.6.2	$\mathcal{O}(\mathsf{IN}^{1+\varepsilon})$ Theorem 3.6.2	$\mathcal{O}(\mathsf{IN})$ Theorem 3.5.1
Acyclic CQs	$\mathcal{O}((\mathsf{IN}+\mathsf{IN}\cdot\mathsf{OUT})^{1+\varepsilon}+\mathsf{IN}^2)$ Corollary 3.6.4	$\mathcal{O}((\mathsf{IN}+\mathsf{IN}\cdot\mathsf{OUT})^{1+\varepsilon})$ Corollary 3.6.4	Theorem 3.5.4
CQs with GHW k	$\mathcal{O}((\mathsf{IN}^k+\mathsf{IN}^k\cdot\mathsf{OUT})^{1+\varepsilon}+\mathsf{IN}^2)$ Corollary 3.6.4	$\mathcal{O}((\mathsf{IN}^k+\mathsf{IN}^k\cdot\mathsf{OUT})^{1+\varepsilon})$ Corollary 3.6.4	Theorem 3.5.5
Free-Connex Acyclic CQs	$\mathcal{O}((\mathsf{IN}+\mathsf{OUT})^{1+\varepsilon}+\mathsf{IN}^2)$ Corollary 3.6.5	$\mathcal{O}((\mathsf{IN}+\mathsf{OUT})^{1+\varepsilon})$ Corollary 3.6.5	Proposition 3.5.6
CQs with Free-Connex GHW k	$\mathcal{O}((\mathsf{IN}^k+\mathsf{OUT})^{1+\varepsilon}+\mathsf{IN}^2)$ Corollary 3.6.5	$\mathcal{O}((\mathsf{IN}^k+\mathsf{OUT})^{1+\varepsilon})$ Corollary 3.6.5	Corollary 3.5.7
Natural Join Queries	$\mathcal{O}((\prod_{i=1}^m \lvert R_i\rvert^{x_i}+\mathsf{IN})^{1+\varepsilon}+\mathsf{IN}^2)$ Corollary 3.6.6	$\mathcal{O}((\prod_{i=1}^m \lvert R_i\rvert^{x_i}+\mathsf{IN})^{1+\varepsilon})$ Corollary 3.6.6	Theorem 3.5.8

Table 3.1: Work bounds for various query classes. Here CQ is short for "conjunctive query", GHW is short for "generalized hypertree width", natural join queries have the form $R_1 \bowtie \ldots \bowtie R_m$, and $x_1, \ldots, x_m$ constitute a fractional edge cover. In the ordered setting the algorithms require suitably ordered input arrays.

the conference paper the evaluation of queries of the semi-join algebra and structurally simple conjunctive queries was only considered in two settings – the general and the dictionary setting – while the (almost) worst-case optimal evaluation of natural join queries was only considered in the ordered setting. Here (and in the full version) the evaluation of all these queries is considered within all three settings. Furthermore, the work and space bounds for evaluating conjunctive queries presented in Section 3.5.2 can be arguably deemed work-efficient, while the bounds presented in the conference paper cannot. These improvements are mainly thanks to the compaction technique of Goldberg and Zwick [GZ95, Theorem 4.2], the padded integer sorting algorithm presented here as Proposition 3.1.11, and the translations into the dictionary setting presented in Section 3.6.

The results presented in this chapter have been developed and were evolved in several joint research sessions. Therefore, it is fair to say that all authors contributed equally to all these results. All results are also included in the full version of the conference paper. Their presentation, however, differs.

3.1 PRAMs and Constant-Time Parallel Algorithms

In this section, we define PRAMs and recall some known results on $\mathcal{O}(1)$-time parallel algorithms for PRAMs from the literature. The latter will include lower bounds that

reveal some obstacles for designing $\mathcal{O}(1)$-time parallel algorithms. On the other hand, we also discuss quite powerful tools to tackle these obstacles.

In addition, we present a $\mathcal{O}(1)$-time parallel algorithm for padded integer sorting – applicable to sequences of integers of polynomial size – that will be useful for our purposes.

3.1.1 Parallel Random Access Machines (PRAMs)

A *Parallel Random Access Machine (PRAM)* consists of a number of processors which operate in parallel[2] and can use a *shared memory*. The processors are consecutively numbered from 1 to $p_{\max}$. Each processor can access its *processor number* and freely use it in computations. For instance, a common application is to use it as an index for an array. The memory consists of *memory cells* which are consecutively numbered as well, starting with 1. The number of a memory cell is called its *address*. An input for a PRAM is a sequence of n numbers, whose binary encodings have length $\mathcal{O}(\log n)$ each, and which are initially stored in the first n memory cells. We presume that the *word size*, that is, the number of bits that can be stored in a memory cell, is of the form $\mathcal{O}(\log n + \log p_{\max})$. Thus, a single memory cell can hold an (arbitrary) input number, and it is always possible to store a processor number in a memory cell.

A processor can access any memory cell in $\mathcal{O}(1)$-time, given its address. Each processor is also equipped with a constant number of *local registers* which it can use for computations. These registers are essentially memory cells but, unlike them, registers are *not* shared. Furthermore, we assume that the usual arithmetic and bitwise operations can be performed in $\mathcal{O}(1)$-time by a single processor.

We mostly use the *Concurrent-Read Concurrent-Write (CRCW)* model which allows processors to read and write concurrently from and to the same memory cell. More precisely, we mainly assume the *arbitrary* CRCW PRAM model: In case multiple processors concurrently write to the same memory cell, exactly one of them, "arbitrarily", succeeds. This is our standard model, and we often refer to it simply by CRCW PRAM.

For some algorithms the weaker *common* model would suffice. In this model all processors need to write the same value, if they attempt to write into the same memory cell. On the other hand, some of the lower bounds which we recall below even hold for the stronger *priority* model, where always the processor with the smallest processor number succeeds in writing into a memory cell. We note that all these models are equally expressive in the sense that they can simulate each other. However, the simulation of a stronger model by a weaker one comes at a cost: It takes $\omega(1)$ time, or it requires additional processors [Rag92].

We sometimes also use the weaker *Exclusive-Read Exclusive-Write (EREW)* model (EREW PRAM), where concurrent access is forbidden. Cook et al. [CDR86, Theorem 7] showed that EREW PRAMs are strictly less "powerful" than the concurrent-write models defined above when it comes to constant-time algorithms, because they cannot compute certain functions, like for instance the logical *or* of n bits, in $\mathcal{O}(1)$ time.

[2] More precisely, the processors operate synchronously, that is, there is a global clock.

The *work* of a PRAM computation is the sum of the number of all computation steps of every processor made during the computation. We note that, for $\mathcal{O}(1)$-time parallel algorithms, asymptotically the work concurs with the number of processors $p_{\max}$. Let m be the maximal address of any memory cell accessed during a computation of a PRAM. Then the *space* required by this PRAM computation is $m+cp_{\max}$ where c is the (constant) number of local registers available to a processor. For most of our algorithms $m \geq p_{\max}$ holds trivially, and we will hence focus on m to determine asymptotic space bounds.

For more details on PRAMs, we refer to the book of JáJá [JáJ92], and to the discussion of Emde Boas [Emd90, Section 2.2.3] on alternative space measures.

Arrays. Many algorithms for PRAMs, including our algorithms, operate on arrays – a very natural data structure for (P)RAMs. An *array* $\mathbf{A}$ is a sequence of consecutive memory cells. The *length* $|\mathbf{A}|$ of $\mathbf{A}$ is the number of memory cells it consists of. By $\mathbf{A}[i]$, for $1 \leq i \leq |\mathbf{A}|$, we refer to the i-th cell of $\mathbf{A}$, and to the (current) content of that cell. We call i the *index* of cell $\mathbf{A}[i]$, and $1, \ldots, |\mathbf{A}|$ the indices of $\mathbf{A}$. Furthermore, we assume that the length of an array is always available to all processors (for instance, it might be stored in a "hidden" cell with index 0). Given an index i, any processor can access cell $\mathbf{A}[i]$ in $\mathcal{O}(1)$ time on its own.

We will also use arrays whose cells correspond to c underlying memory cells each, for some constant integer $c > 0$. In terms of some programming languages this corresponds to an array of *structs* (or *records*). Since a single processor can read and write c memory cells in constant time, we blur the distinction. For instance, if we write $\mathbf{A}[i]$ we actually refer to c underlying memory cells and an array of length n actually consists of cn underlying memory cells.

3.1.2 Lower and Upper Bounds for Constant-Time CRCW PRAM Algorithms

Next, we recall some known results on PRAM algorithms from the literature. The first result implies that we cannot expect, in general, that query results can be stored in a compact fashion. A bit more precisely, the result $Q(D)$ of a query Q over a database D cannot be stored in an array of length $|Q(D)|$. This follows from the following lower bound for computing the parity function on CRCW PRAMs.

Proposition 3.1.1 [JáJ92, Theorem 10.8]**.** *Any algorithm that, given an array* $\mathbf{A}$ *of length* n *whose cells contain either* 0 *or* 1*, outputs whether the number of cells containing* 1 *is even (or not) and uses a polynomial number of processors requires* $\Omega\left(\frac{\log n}{\log \log n}\right)$ *time on a priority CRCW PRAM.*

Next, we make the term "compact" precise. We consider arrays which may have *empty cells*.[3] An array *without* empty cells is *compact*. Furthermore, we say that an algorithm *compacts* an array $\mathbf{A}$ with k non-empty cells, if it computes a compact array $\mathbf{B}$ of length k that contains the same values as $\mathbf{A}$. Here two arrays $\mathbf{A}$ and $\mathbf{B}$ *contain the same values*

[3] In terms of some programming languages an empty cell corresponds to a cell containing a special `null` value.

if they have the same number of non-empty cells and there is a bijection f between the indices of these non-empty cells such that $\mathbf{B}[f(i)] = \mathbf{A}[i]$ holds for every non-empty cell $\mathbf{A}[i]$ of $\mathbf{A}$.

Proposition 3.1.1 implies that an algorithm that compacts arrays cannot run in constant time on our PRAM models. Indeed, given an array $\mathbf{A}$ whose cells contain either 0 or 1, even the exact number of cells containing 1 can be determined by **(1)** interpreting all cells containing 0 as empty and **(2)** compacting the array – the length of the resulting array is then the number of cells of the original array containing 1.

Corollary 3.1.2. *Any algorithm that compacts an array of length n, and uses a polynomial number of processors, requires $\Omega(\frac{\log n}{\log \log n})$ time on a priority CRCW PRAM.*

Proposition 3.1.1 also implies a lower bound for sorting arrays. Here it suffices to sort the given array $\mathbf{A}$. The number of cells containing 1 can then be determined from the index of the first cell containing 1. This index can easily be found by $|\mathbf{A}| - 1$ processors in constant time: Processor i simply checks cells $\mathbf{A}[i]$ and $\mathbf{A}[i+1]$.

Corollary 3.1.3. *Any algorithm that sorts an array of length n containing natural numbers and uses a polynomial number of processors requires $\Omega(\frac{\log n}{\log \log n})$ time on a priority CRCW PRAM. This holds even if all numbers in the array are from $\{0, 1\}$.*

Of course, the lower bounds stated in Corollaries 3.1.2 and 3.1.3 also apply to the arbitrary and common CRCW PRAM models, since they are weaker than the priority model [cf., e.g., JáJ92]. We will see in Section 3.4.1 that these lower bounds transfer to query evaluation algorithms in the sense that they cannot output the query result in a compact array. This is because, in a nutshell, even very simple queries can effectively ask for the indices of all cells containing 1. In the remainder of this section we have a closer look at further lower bounds as well as at almost matching upper bounds for approximate compaction and sorting.

Approximate Compaction. Corollary 3.1.2 rules out any constant-time algorithms for compacting arrays (for our PRAM models). Therefore, the best we can hope for is approximate compaction, that is, compaction procedures that guarantee some maximum size of the result array in terms of the number of non-empty cells.

An *accuracy function* is a function that maps natural numbers to positive real numbers. Let, in the following, λ denote such an accuracy function. An array $\mathbf{A}$ of length n with k non-empty cells is *λ-compact* if $n \leq (1 + \lambda(n))k$ holds. We say that an algorithm achieves *λ-compaction*, if it, given an array $\mathbf{A}$, computes a λ-compact array $\mathbf{B}$ that contains the same values as $\mathbf{A}$. We call it *order-preserving*, if the relative order of values does *not* change. That is, if there is a bijection f between the indices of non-empty cells of $\mathbf{A}$ and $\mathbf{B}$ such that **(1)** $\mathbf{B}[f(i)] = \mathbf{A}[i]$ holds for every non-empty cell $\mathbf{A}[i]$ of $\mathbf{A}$, and **(2)** $f(i) < f(j)$ whenever $i < j$ and i, j are indices of non-empty cells of $\mathbf{A}$.

Note that, for accuracy functions λ with $\lambda(n) \leq \frac{1}{n+1}$ for all positive integers n, a λ-compact array is always (perfectly) compact. Indeed, for such λ and λ-compact arrays

of length n with k non-empty cells we have that

$$n \leq (1 + \lambda(n))k \leq \left(1 + \frac{1}{n+1}\right) k = k + \frac{k}{n+1} = k + \varepsilon$$

for $\varepsilon = \frac{n}{n+1} < 1$, because $k \leq n$. But then we have $k = n$ since n is an integer. In other words, the array is compact. We thus always assume that $\lambda(n) > \frac{1}{n+1}$ holds. In fact, in this chapter λ will either be a constant or of the form $(\log n)^{-c}$, for some constant c.

We will see that approximate compaction can be done quite well, but the following result implies a trade-off between time and number of processors.

Proposition 3.1.4 [Cha96, Theorem 5.3]**.** *Any algorithm that achieves λ-approximate compaction, for an accuracy function λ, and uses $\mu(n)n$ processors on a priority CRCW PRAM, for some function μ, requires time $\Omega\big(\log \frac{\log n}{\log(\mu(n)\lambda(n)+2)}\big)$.*

Corollary 3.1.5. *Any algorithm that achieves λ-approximate compaction for a constant $\lambda > 0$ within constant time requires $\Omega(n^{1+\varepsilon})$ processors, for some constant $\varepsilon > 0$.*

Proof. Consider an algorithm that achieves λ-compaction in constant time t using $\mu(n)n$ processors. Due to Proposition 3.1.4 there is a constant c such that $t \geq c \log \frac{\log n}{\log(\mu(n)\lambda+2)}$ holds. Therefore, we get the following chain of inequalities.

$$2^t \geq \Big(\frac{\log n}{\log(\mu(n)\lambda + 2)}\Big)^c \quad \Rightarrow \quad 2^{t/c} \geq \frac{\log n}{\log(\mu(n)\lambda + 2)} \quad \Rightarrow \quad \log(\mu(n)\lambda + 2) \geq \frac{1}{2^{t/c}} \log n$$

Consequently, we have $\mu(n)\lambda + 2 \geq n^{\varepsilon}$, where $\varepsilon = \frac{1}{2^{t/c}}$ is a constant. Since we assumed λ to be constant, we can conclude that $\mu(n) \in \Omega(n^{\varepsilon})$, and thus, $\mu(n)n \in \Omega(n^{1+\varepsilon})$ holds. □

As a consequence of Corollaries 3.1.2 and 3.1.5, our main data structures are λ-compact arrays that (possibly) have *empty cells*. In particular, our algorithms use such arrays to represent query results – and, maybe more importantly, intermediate results. The latter means that our algorithms also have to deal with λ-compact input arrays. Furthermore, the best work bounds we can expect for algorithms that output λ-compact arrays are of the form $\mathcal{O}(n^{1+\varepsilon})$, for some $\varepsilon > 0$.

The good news is that there are algorithms that achieve λ-compaction and match the lower bound of Corollary 3.1.5, even order-preserving ones, and for $\lambda \in o(1)$.

Proposition 3.1.6 [GZ95]**.** *For every $\varepsilon > 0$ and $c > 0$ there is a $\mathcal{O}(1)$-time parallel algorithm that achieves order-preserving λ-approximate compaction for $\lambda(n) = (\log n)^{-c}$. The algorithm requires work and space $\mathcal{O}(n^{1+\varepsilon})$ on a common CRCW PRAM.*

Proposition 3.1.6 is not stated in this form by Goldberg and Zwick [GZ95], but it readily follows from their result on consistent (approximate) prefix sums, which we discuss next.

Definition 3.1.7. Let $\mathbf{A}$ be an array of length n whose cells contain integers. An array $\mathbf{B}$ of the same length contains *λ-consistent prefix sums* for $\mathbf{A}$, if, for each $i \in [1, n]$, the following two conditions hold.

(a) $\sum_{j=1}^{i} \mathbf{A}[j] \leq \mathbf{B}[i] \leq (1+\lambda(n)) \sum_{j=1}^{i} \mathbf{A}[j]$

(b) $\mathbf{B}[i] - \mathbf{B}[i-1] \geq \mathbf{A}[i]$, if $i > 1$

Condition (a) ensures that each $\mathbf{B}[i]$ is an approximation of the *exact prefix sum* $\sum_{j=1}^{i} \mathbf{A}[j]$ and Condition (b) ensures *consistency*.

Goldberg and Zwick [GZ95] proved the following result. We note that they did not state the space bound explicitly, and therefore, we revisit their proof in Appendix A.

Proposition 3.1.8 [GZ95, Theorem 4.2]**.** *For every $\varepsilon > 0$ and $c > 0$ there is a $\mathcal{O}(1)$-time parallel algorithm that, given an array $\mathbf{A}$ of length n with $(\log n)$-bit integers, computes an array containing λ-consistent prefix sums for $\mathbf{A}$, where $\lambda(n) = (\log n)^{-c}$. The algorithm requires work and space $\mathcal{O}(n^{1+\varepsilon})$ on a common CRCW PRAM.*

Indeed, Proposition 3.1.6 follows by using an array $\mathbf{A}$ with $\mathbf{A}[j] = 1$ if the j-th cell of the array that is to be compacted is *not* empty and 0, if it is empty. The λ-consistent prefix sums then yield the new indices for the contents of the non-empty cells.

We emphasize that thus the size of the cell *contents* does *not* matter in Proposition 3.1.6.

Padded Sorting. Another notorious obstacle for $\mathcal{O}(1)$-time parallel algorithms is pointed out by Corollary 3.1.3: They cannot sort (compactly) in constant time, at all. And similarly to Corollary 3.1.5 linearly many processors do not suffice to sort in a (slightly) non-compact fashion. The *λ-padded integer sorting problem* asks, given an array $\mathbf{A}$ of length n whose cells contain integers, to sort the integers in $\mathbf{A}$ into an array of length $(1+\lambda)n$. The cells of the result array that do *not* contain an integer from $\mathbf{A}$ are required to be empty.

Proposition 3.1.9 [Cha96, Theorem 5.4]**.** *Solving the λ-padded sorting problem on a priority CRCW PRAM using $\mu(n)n$ processors requires $\Omega\big(\log \frac{\log n}{\log((\lambda(n)+2)(\mu(n)+1))}\big)$ time.*

The following corollary follows from Proposition 3.1.9 analogously to how Corollary 3.1.5 follows from Proposition 3.1.4.

Corollary 3.1.10. *Any constant-time algorithm for the λ-padded sorting problem, for a constant λ, requires $\Omega(n^{1+\varepsilon})$ processors, for some constant $\varepsilon > 0$.*

Thus, we cannot rely on sorting in constant time with a linear number of processors.

However, with the help of the algorithms guaranteed by Propositions 3.1.6 and 3.1.8, it is possible to sort n numbers of polynomial size in constant time with $\mathcal{O}(n^{1+\varepsilon})$ work. We will see later that this is very useful for our purposes since we will often have to deal with fixed-length tuples of numbers of linear size, which can be viewed as numbers of polynomial size.

Proposition 3.1.11. *For all constants $\varepsilon > 0$, $\lambda > 0$, and $c > 0$ there is a $\mathcal{O}(1)$-time parallel algorithm that solves the λ-padded integer sorting problem for integers in the range $[0, n^c - 1]$, where n is the length of the input array. It requires work and space $\mathcal{O}(n^{1+\varepsilon})$ on a common CRCW PRAM.*

Proof. Let $a_1, \ldots, a_n$ be the sequence of integers stored in the input array $\mathbf{A}$ of length n. In a preliminary step, each number a_i is replaced by $a_i(n+1)+i$ to guarantee that all numbers are pairwise distinct. We also choose δ as $\min\{\frac{1}{3}, \frac{\varepsilon}{3}\}$.

The algorithm proceeds in two phases. In the first phase, it computes numbers $b_1, \ldots, b_n$ of size at most $\mathcal{O}(n^{1+\delta})$ such that, for all i, j, it holds $b_i < b_j$ if and only if $a_i < a_j$. These numbers are then used in the second step to order the original sequence $a_1, \ldots, a_n$.

Towards the first phase, we can assume that $n^c > n^{1+\delta}$ holds, since otherwise, the first phase can be skipped.

The algorithm performs multiple rounds of a bucket-like sort, reducing the range by applying a factor of $(1+\lambda)\frac{1}{n^\delta}$ in each round.

In the following, we describe the algorithm for the first round. More precisely, we show that in constant time with work $\mathcal{O}(n^{1+\delta})$ numbers $b_1, \ldots, b_n$ can be computed (and stored in an array $\mathbf{B}$), such that, **(1)** for all indices i, j, we have $b_i < b_j$ if and only if $a_i < a_j$ and, **(2)** for each index $i \leq n$, we have $b_i \leq (1+\lambda)n^{c-\delta}$.

First, for each $i \leq n$, the algorithm determines c_i, d_i such that $a_i = c_i n^{c-1-\delta} + d_i$ and $0 \leq d_i < n^{c-1-\delta}$. Since, $a_i \leq n^c - 1$, we have $c_i \leq n^{1+\delta} - 1$. Here c_i is the "bucket" for integer a_i.

Next, each c_i is replaced by a number of size at most $(1+\lambda)n$. To this end, let $\mathbf{C}$ be an array of length $n^{1+\delta}$, all of whose cells are initially empty. For each i, the number c_i is stored in cell $\mathbf{C}[c_i]$. Next, the array $\mathbf{C}$ is compacted into an array of length at most $(1+\lambda)n-1$ using Proposition 3.1.6 with a suitable $\lambda' < \lambda$. Let $\mathbf{C}'$ denote this compacted version of $\mathbf{C}$. Each c_i is then assigned to its index in $\mathbf{C}'$ as follows: For each index j of $\mathbf{C}'$, if the $\mathbf{C}'[j]$ is *not* empty and contains c_i, then j is stored in $\mathbf{C}[c_i]$. We note that, since Proposition 3.1.6 guarantees order-preserving compaction, we have that $\mathbf{C}[c_i] < \mathbf{C}[c_k]$ if and only if $c_i < c_k$.

Finally, for each $i \leq n$, we define b_i as $\mathbf{C}[c_i]n^{c-1-\delta} + d_i$. Since $\mathbf{C}$ only contains numbers of size at most $(1+\lambda)n - 1$ and $d_i < n^{c-1-\delta}$, we have that

$$b_i \leq ((1+\lambda)n - 1)n^{c-1-\delta} + n^{c-1-\delta} = (1+\lambda)n^{c-\delta}.$$

By repeating this procedure for at most $\mathcal{O}(c)$ times, we obtain an array $\mathbf{B}$ with numbers $b_1, \ldots, b_n$ of size at most $\mathcal{O}(n^{1+\delta})$ such that, for each $i \leq n$, it holds $b_i < b_j$ if and only if $a_i < a_j$. This concludes the first phase of the algorithm.

In the second phase, a new array $\mathbf{D}$ of size $\mathcal{O}(n^{1+\delta})$ is allocated and initialized by setting, for each $i \leq n$, $\mathbf{D}[b_i] = i$. Every other cell of $\mathbf{D}$, i.e. cells for which *no* such i exists, is empty. Compacting $\mathbf{D}$ with the algorithm guaranteed by Proposition 3.1.6 yields an array $\mathbf{D}'$ of size $(1+\lambda)n$. Since the compaction is order-preserving, the sequence $i_1, \ldots, i_n$ of numbers stored in the non-empty cells of $\mathbf{D}'$ fulfils $a_{i_1} < \cdots < a_{i_n}$.

It remains to analyse the required work and space. The work in the second phase is dominated by the compaction. Compacting an array of size $\mathcal{O}(n^{1+\delta})$ can be done with work and space $\mathcal{O}(n^{(1+\delta)^2})$ thanks to Proposition 3.1.6. Since we chose δ such that $\delta \leq 1$ and $\delta \leq \frac{\varepsilon}{3}$ hold, we have

$$n^{(1+\delta)^2} = n^{1+2\delta+\delta^2} \leq n^{1+3\delta} = n^{1+\varepsilon}.$$

Thus, the second phase requires work and space $\mathcal{O}(n^{1+\varepsilon})$. This is also an upper bound for each of the constantly many rounds of the first phase. □

3.2 PRAMs Meet Databases: Settings and Representations

In this section, we first discuss how PRAMs can interact with (input) databases. Besides a very general setting, we consider two more specific settings, namely the ordered and the dictionary setting.

In the second part of this section, we explain how (database) relations can be represented by arrays. While the concrete representation will depend on the setting, this will provide us with a unified interface which allows us to, for instance, represent (intermediate) results and compose database operations.

Throughout this chapter, we always assume a fixed named schema $\mathcal{S}$, and therefore a fixed maximal arity $\mathsf{ar}(\mathcal{S})$. In particular, there is a fixed maximal arity of tuples occurring in a database over $\mathcal{S}$. Recall that, thanks to the linear order on the set att of attributes, we can refer to the j-th attribute of a relation R, for every $j \in [1, \mathsf{ar}(R)]$.

Settings. In the most *general setting* we consider in this chapter, we do *not* specify how databases are actually stored. Rather we only assume that the tuples in every database relation R are numbered, from 1 to $|R|$, and that, for all $R, S \in \mathcal{S}$, the following elemental operations can be carried out in constant time by a single processor.

- $\texttt{\#Tuples}_R$ returns $|R|$, that is the number of tuples in relation R.
- $\texttt{Equal}_{R,S}(i_1, j_1, i_2, j_2)$ tests whether the j_1-th attribute of the i_1-th tuple of relation R has the same value as the j_2-th attribute of the i_2-th tuple of relation S.
- $\texttt{Output}_R(i_1, j_1, i_2, j_2)$ outputs the value of the j_1-th attribute of the i_1-th tuple of R as the value of the j_2-th attribute of the i_2-th output tuple.

The second setting we consider is the *ordered setting*. Here we assume that there is a linear order on the active domain $\mathsf{adom}(D)$ of the given database D, and that the following additional elemental operation is available to access this order.

- $\texttt{LessThan}_{R,S}(i_1, j_1, i_2, j_2)$ tests whether the j_1-th attribute of the i_1-th tuple of relation R is less or equal than the value of the j_2-th attribute of the i_2-th tuple of relation S.

The third and main setting that we consider is in the spirit of dictionary-based compressed databases [see, e.g., CGK01]. In a nutshell, such a database has a dictionary that maps domain values to natural numbers and internally stores and manipulates tuples over these numbers to improve performance. Such dictionaries are often defined attribute-wise, but for our purposes this does *not* matter. Query evaluation then does *not* need to touch the actual dictionary, it only works with the numbers.

For our purposes, we are interested in dictionaries that yield *small numbers*. More precisely, we are interested in an *injective* mapping $\mathsf{key} \colon \mathsf{adom}(D) \to [1, c_{\mathcal{S}}|D|]$, for some constant $c_{\mathcal{S}}$ that may depend on the fixed database schema $\mathcal{S}$ but, of course, *not* on the database.

In the *dictionary setting* we assume the presence of such an injective mapping key, and that a single processor can, in addition to the elemental operations of the general setting, carry out the following two elemental operations for every relation R of the database in constant time.

- $\texttt{KeyOf}_R(i, j)$ returns $\mathsf{key}(a)$ where a is the value of the j-th attribute of the i-th tuple of relation R.
- $\texttt{KeyOutput}(k, i, j)$ outputs the value $\mathsf{key}^{-1}(k)$ as the value of the j-th attribute of the i-th output tuple.

Here $\mathsf{key}^{-1} \colon [1, c_{\mathcal{S}}|D|] \to \mathsf{adom}(D)$ denotes the *inverse* of key on its range, which exists because key is injective. We note that, for relation symbols $R, S \in \mathcal{S}$ and indices i_1, j_1, i_2, j_2, the invocations $\texttt{KeyOf}_R(i_1, j_1)$ and $\texttt{KeyOf}_S(i_2, j_2)$ return the same value if and only if $\texttt{Equal}_{R,S}(i_1, j_1, i_2, j_2)$ returns $\texttt{true}$, that is, if i_1, j_1 and i_2, j_2 with respect to R and S, respectively, refer to the same value.

Moreover, let us emphasize that the dictionary setting is a specialization of the ordered setting, because the mapping key induces a linear order on the domain values. Namely, the linear order where, for all pairs a_1, a_2 of domain values, a_1 is smaller or equal than a_2, if $\mathsf{key}(a_1) \leq \mathsf{key}(a_2)$ holds. Here $\leq$ is the natural, linear order on the natural numbers in $[1, c_{\mathcal{S}}|D|]$. Furthermore, there is a straightforward, constant-time implementation of the operation $\texttt{LessThan}_{R,S}$. Indeed, a single processor can simply obtain $\mathsf{key}(a_1)$ and $\mathsf{key}(a_2)$ in constant time using $\texttt{KeyOf}_R$ and $\texttt{KeyOf}_S$, and then test whether $\mathsf{key}(a_1) \leq \mathsf{key}(a_2)$ holds.

We will see in Section 3.6 that, in the general and the ordered settings, it is possible to compute a data structure which allows constant-time, single processor implementations of $\texttt{KeyOf}_R$ and $\texttt{KeyOutput}$ in these settings. This effectively yields a translation from these settings into the dictionary setting. And while computing this data structure comes at a cost, this cost will turn out to be somewhat reasonable.

Next, we will explain how domain values, tuples, and relations can be represented by a PRAM in its memory. This is, in particular, intended for storing (intermediate) results and having an interface for composing (database) operations.

Representing Databases. We first discuss the representations in the general and ordered setting. Since it is *not* possible to access the domain values directly, we represent domain values by *tokens* which are triples of the form (R, i, j), where R is a relation symbol, $i \in [1, |R|]$, and $j \in [1, \mathsf{ar}(R)]$. More precisely, a token (R, i, j) represents the value of the j-th attribute of the i-th tuple of relation R. We emphasize that different tokens (R, i_1, j_1) and (S, i_2, j_2), with possibly $R = S$, can represent the same domain value. Note that a single processor can test in constant time whether two such tokens represent the same domain value using the operation $\texttt{Equal}_{R,S}(i_1, i_2, j_1, j_2)$. In the ordered setting the domain values represented can also be compared using $\texttt{LessThan}_{R,S}(i_1, i_2, j_1, j_2)$.

This encoding extends to tuples in the natural fashion. Furthermore, such tokens can be stored in the memory cells of a PRAM. We will refer to this encoding as *token representation* of a domain value or tuple.

Example 3.2.1. Consider the ternary relation R over the set $\{X_1, X_2, X_3\}$ of attributes given by

$$R = \big\{(\texttt{"Hello"}, 3.14, \texttt{blob}_1), (\texttt{"World"}, 6000, \texttt{blob}_2), (\texttt{"Hello"}, 3.14, \texttt{blob}_3)\big\}.$$

Here we assume that, for each $j \in [1, 3]$, X_j is the j-th attribute of R. For instance, $(\texttt{"Hello"}, 3.14, \texttt{blob}_1)$ corresponds to the named tuple

$$\tilde{a}_1 = \big\{X_1 \mapsto \texttt{"Hello"}, X_2 \mapsto 3.14, X_3 \mapsto \texttt{blob}_1\big\}.$$

From the perspective of a PRAM, the tuples of R can be represented by the token sequences

$$((R,1,1),(R,1,2),(R,1,3)), ((R,2,1),(R,2,2),(R,2,3)),$$
$$\text{and } ((R,3,1),(R,3,2),(R,3,3)).$$

Observe that $(R,1,1)$ and $(R,3,1)$ as well as $(R,1,2)$ and $(R,3,2)$ represent the same domain values, namely, `"Hello"` and 3.14, respectively. Thus, the third tuple could also be represented by $((R,1,1),(R,1,2),(R,3,3))$.

It is also possible to represent tuples *not* in R. For instance, $((R,3,1),(R,2,1))$ represents the tuple $(\texttt{"Hello"}, \texttt{"World"})$. ◁

In the dictionary setting a PRAM can simply work with the (small) numbers guaranteed by the underlying mapping key. That is, instead of representing a domain value a by a token of the form (R, i, j), a PRAM can invoke $\texttt{KeyOf}_R(i, j)$ and use the return value, which is small enough to be stored in a memory cell, in place of a. In fact, we can view a database in the dictionary setting as a database whose active domain consists of small numbers, i.e. $\mathsf{key}(\mathsf{adom}(D))$. After an initial lookup phase, a PRAM can just work with the return values of the $\texttt{KeyOf}_R$ operations. In particular, it can evaluate queries over $\mathsf{key}(D)$. Thanks to the `KeyOutput` operation, the final query result can always be "translated" back into the original domain. We note that, for this purpose, it is also *not* necessary to keep track of the origin of these small numbers.

Example 3.2.2. Consider the relation R from Example 3.2.1 and suppose key maps the values occurring in R as follows.

$$\texttt{"Hello"} \mapsto 1 \qquad \texttt{"World"} \mapsto 2 \qquad 3.14 \mapsto 6 \qquad 6000 \mapsto 7$$
$$\texttt{blob}_1 \mapsto 20 \qquad \texttt{blob}_2 \mapsto 26 \qquad \texttt{blob}_3 \mapsto 42$$

A PRAM can then operate on

$$\mathsf{key}(R) = \big\{(1,6,20),(2,7,26),(1,6,42)\big\}.$$

Since the numbers in the range of key are small enough, i.e. in $\mathcal{O}(|D|)$, they can be stored in one memory cell each, and each tuple in three memory cells. ◁

As mentioned before, we represent relations by one-dimensional arrays, whose cells contain tuples – or token representations thereof – and might be augmented by additional data. For this purpose, a cell of an array might actually consist of a constant number of memory cells, depending only on the fixed database schema.

Due to the limitations of $\mathcal{O}(1)$-time parallel algorithms imposed by the impossibility results for (perfect) compaction and sorting discussed in Section 3.1.2, our algorithms will often yield result arrays, in which *not* all cells contain result tuples.

Therefore, a cell can either be *inhabited*, if it represents a "useful" tuple, or *uninhabited*, indicated by some Boolean flag.[4] In the dictionary setting, an inhabited cell $\mathbf{A}[i]$ of an array $\mathbf{A}$ always contains a named tuple $\tilde{a}$ (over some relation schema) whose values are small numbers, i.e. the image of a tuple under $\mathtt{key}$. In the other two settings, an inhabited cell contains a token representation of a tuple. We blur the distinction between these representations as well as the represented tuples, and write $\mathbf{A}[i].\mathtt{t}$ for both, the representation stored in $\mathbf{A}[i]$ and the represented tuple. As discussed before, we assume in the dictionary setting that the values occurring in $\mathbf{A}[i].\mathtt{t}$ are small numbers.

Note that operations like setting $\mathbf{A}[i].\mathtt{t}[X] = \mathbf{B}[j].\mathtt{t}[Y]$ or testing whether $\mathbf{A}[i].\mathtt{t}[X]$ equals $\mathbf{B}[j].\mathtt{t}[Y]$ can be performed by a single processor in constant time (and space), in any setting. In the dictionary setting these kinds of operations can be performed directly. In the other two settings, the former kind of operation can be achieved by changing the token representation accordingly, and the latter using the elemental operations $\mathtt{Equal}_{R,S}$. Since the arity of tuples is always fixed in this chapter, this is not limited to single attributes but extends to tuples. In the ordered setting, comparing values w.r.t. the underlying linear order is, of course, also possible. For instance, testing whether $\mathbf{A}[i].\mathtt{t}[X] < \mathbf{B}[j].\mathtt{t}[Y]$ holds can be done by a single processor in constant time. In the same spirit we say that a tuple occurs in a cell or a cell contains a tuple when the cell actually contains a token representation of the tuple.

There might be additional data stored in a cell, for instance, further Boolean flags and pointers to other cells of (possibly) other arrays. However, the number of items is always bounded by a constant. Thus, the contents of a cell can still be read or written in constant time.

We say that an array *represents* a relation R, if for each tuple $\tilde{a}$ in R, there is some inhabited cell that contains $\tilde{a}$, and *no* inhabited cell contains a tuple *not* in R. This definition allows that a tuple occurs more than once. An array represents R *concisely*, if each tuple occurs in exactly one inhabited cell. To indicate that an array represents a relation R we usually denote it by $\mathbf{R}$, $\mathbf{R}'$, etc.

We adapt the compactness notions and call an array representing a relation *compact* if it has *no* uninhabited cells. We call it λ-*compact*, for some $\lambda > 0$, if it has at length at most $(1 + \lambda)k$ where k is the number of inhabited cells.

Remark 3.2.3. Observe that, for each relation R of the input database, an array $\mathbf{R}$

[4] In difference to an empty cell, an uninhabited cell contains data, at the very least the Boolean flag indicating that it is, in fact, uninhabited. Our algorithms will always be able to initialize arrays accordingly without affecting their work and space bounds. That being said, sometimes it will be convenient to interpret uninhabited cells as empty cells.

representing R concisely can easily be compiled with $|R|$ processors in constant time: In the general setting and the ordered setting, processor i simply writes the token representation $((R, i, 1), \ldots, (R, i, \mathsf{ar}(R)))$ into cell $\mathbf{R}[i]$. The number of required processors can be obtained by invoking $\texttt{\#Tuples}_R$.

In the dictionary setting the numbers returned by $\texttt{KeyOf}_R(i, 1), \ldots, \texttt{KeyOf}_R(i, \mathsf{ar}(R))$ are written instead. Although the arrays obtained in this fashion are even compact, we will actually only assume λ-compactness, for some constant λ, for the arrays representing the relations of the input database. Indeed, this does *not* affect our algorithms or their work (and space) bounds. It has, however, the advantage that our algorithms are still applicable if the arrays have been obtained by other means. In particular, our translations from the general and ordered settings into the dictionary setting in Section 3.6 do *not* guarantee that the arrays representing the (translated) database are (perfectly) compact.

We often consider the *induced tuple sequence* $\tilde{a}_1, \ldots, \tilde{a}_n$ of an array $\mathbf{A}$ of length n. Here, for each index $i \in [1, n]$, $\tilde{a}_i = \mathbf{A}[i].\texttt{t}$ is either a *proper tuple*, if the cell $\mathbf{A}[i]$ is inhabited, or otherwise, $\tilde{a}_i$ is the *empty tuple*, which we denote by $\perp$.

Example 3.2.4. The tuple sequence $(1, 5), \perp, (3, 4), (8, 3), (1, 5), \perp, \perp, (7, 3)$ of an array $\mathbf{R}$ of length eight consists of five proper tuples and the empty tuple occurs three times. The second, sixth, and seventh cell of $\mathbf{R}$ are hence uninhabited. It represents the relation $R = \{(1, 5), (3, 4), (8, 3), (7, 3)\}$, but *not* concisely, because $(1, 5)$ occurs twice in $\mathbf{R}$.

The sequence $(1, 5), (3, 4), (7, 3), \perp, (8, 3)$ represents R concisely. The corresponding array is *not* compact but it is $\frac{1}{4}$-compact. ◁

Ordered Arrays. In the ordered and dictionary setting we consider (lexicographically) ordered arrays. Recall that in these two settings a linear order on the domain values is available. We consider the extension of this order to the lexicographical order on tuples. To be precise, for an (ordered) sequence $\mathcal{X} = (X_1, \ldots, X_k)$ of attributes[5], and two named tuples $\tilde{a}$ and $\tilde{b}$ over $\mathcal{X}$ we have $\tilde{a} <_{\mathcal{X}} \tilde{b}$ if, for some $j \in [1, k]$, $\tilde{a}[X_j] < \tilde{b}[X_j]$ and $\tilde{a}[X_\ell] = \tilde{b}[X_\ell]$ holds for all $\ell \in [1, j-1]$. We often omit the subscript $\mathcal{X}$ if it is clear from the context.

An array $\mathbf{R}$ that represents a relation R is $\mathcal{X}$-*ordered*, for some sequence $\mathcal{X}$ of attributes from $\texttt{attr}(R)$, if $i < j$ implies $\mathbf{R}[i].\texttt{t}[\mathcal{X}] \leq_{\mathcal{X}} \mathbf{R}[j].\texttt{t}[\mathcal{X}]$, for all indices i, j. Thus, tuples that agree on the attributes of $\mathcal{X}$ might occur in any order. We call $\mathbf{R}$ *fully ordered*, if it is $\mathcal{X}$-ordered, for a sequence $\mathcal{X}$ that contains all attributes in $\texttt{attr}(R)$. In this case, the relative order is uniquely determined for all tuples.

We will primarily use ordered arrays to search efficiently for tuples in relations. In the sequential settings, various kinds of *index structures* – for instance, based on search trees – are used for purposes like this; particularly, if multiple orders of the same relation play a role. We chose to use ordered arrays because for almost all our algorithms one order suffices, and it lead to a simpler presentation of our PRAM algorithms. We will discuss some alternatives in Section 3.7.

[5]We note that the order of attributes in $\mathcal{X}$ does *not* necessarily agree with the linear order on the set att of all attributes.

3.3 Algorithmic Techniques and Basic Array Operations

In this section, we present some basic operations on arrays which we will use as building blocks to implement database operations and the translation from the general and ordered settings into the dictionary setting. For some of these operations it will suffice to apply the results discussed in Section 3.1.2 appropriately. However, to obtain work-efficient algorithms for our search and deduplication operations, which will turn out to be essential, we require some setting-dependent algorithmic techniques.

We will proceed as follows. We start with an introduction of the basic operations and related terminology. In Section 3.3.1 we will present the setting-dependent algorithmic techniques, before we present our algorithms for the basic operations in Section 3.3.2.

Operations and Pointers. Before we define our basic operations we illustrate some aspects by means of examples.

Example 3.3.1. We sketch a naive algorithm for evaluating the semi-join $R \ltimes S$ of two binary relations R, S over attributes $\{X_1, X_2\}$ and $\{X_2, X_3\}$, respectively. Let R be given by the array $\mathbf{R}$ induced by the tuple sequence $(1,5), (3,4), (7,3), \bot, (8,3)$ from Example 3.2.4. Moreover, let S be given by the array $\mathbf{S}$ induced by the tuple sequence $(3,2), (3,6), (5,5)$. Note that $\mathbf{R}$ and $\mathbf{S}$ represent R and S concisely, respectively.

The algorithm proceeds in three steps to compute an array representing $R \ltimes S$ concisely. In the first step, it computes *not* necessarily concise arrays $\mathbf{R}'$ and $\mathbf{S}'$ that represent the projections of R and S to the common attribute X_2.

Indeed, $\mathbf{R}'$ can be easily computed using $|\mathbf{R}|$ processors: It is initialized as an array of the same length as $\mathbf{R}$. For each $i \in [1, |\mathbf{R}|]$, processor i then reads $\tilde{a}_i = \mathbf{R}[i].\mathtt{t}$ and writes $\tilde{a}_i[X_2]$ into cell $\mathbf{R}'[i]$, if $\mathbf{R}[i]$ is inhabited. We stress that the number and indices of uninhabited cells of $\mathbf{R}$ are *not* known, in the general case, and we thus assign a processor to each cell of $\mathbf{R}$ – inhabited or *not*. The array $\mathbf{S}'$ can be obtained analogously. In this example, $\mathbf{R}'$ and $\mathbf{S}'$ correspond to the sequences $5, 4, 3, \bot, 3$ and $3, 3, 5$, respectively. Overall this step takes $\mathcal{O}(|\mathbf{R}| + |\mathbf{S}|)$ work and space.

In the second step, the algorithm searches for every proper tuple of $\mathbf{R}'$ in the array $\mathbf{S}'$ in parallel. This can be done using $|\mathbf{R}'| \cdot |\mathbf{S}'|$ processors as follows. To each cell $\mathbf{R}'[i]$ of $\mathbf{R}'$ the algorithm assigns $|\mathbf{S}'|$ processors which check, in parallel, each cell $\mathbf{S}'[j]$ of $\mathbf{S}'$. If $\mathbf{R}'[i].\mathtt{t} = \mathbf{S}'[j].\mathtt{t}$ and $\mathbf{R}'[i]$ is inhabited, then a *pointer* from $\mathbf{R}'[i]$ to $\mathbf{S}'[j]$ is inserted. We note that multiple processors might attempt to insert a tuple in parallel, but due to the semantics of an arbitrary CRCW PRAM only one will succeed. For instance, for $\mathbf{R}'[5].\mathtt{t} = 3$, the processors for $\mathbf{S}'[1].\mathtt{t} = 3$ and $\mathbf{S}'[2].\mathtt{t} = 3$ assigned to $\mathbf{R}'[5]$ find a "match". Thus, $\mathbf{R}'[5]$ is augmented by a pointer, either to $\mathbf{S}'[1]$ or to $\mathbf{S}'[2]$, depending on which processor succeeds in writing. In our case, all inhabited cells of $\mathbf{R}'$ are augmented by a pointer, except for $\mathbf{R}'[2]$, since $\mathbf{R}'[2].\mathtt{t} = 4$ does *not* occur in $\mathbf{S}'$.

In the last step, the algorithm uses $|\mathbf{R}'|$ processors to mark every cell $\mathbf{R}[i]$ as uninhabited for which $\mathbf{R}'[i]$ is *not* augmented by a pointer to $\mathbf{S}'$. The resulting array, with the induced tuple sequence $(1,5), \bot, (7,3), \bot, (8,3)$, represents $R \ltimes S$ concisely.

The work and space required by this naive algorithm are dominated by the search step,

which requires $\mathcal{O}(|\mathbf{R}'| \cdot |\mathbf{S}'|) = |\mathbf{R}| \cdot |\mathbf{S}|$ work and linear space. We will see that, given an $\{X_2\}$-ordered array $\mathbf{S}$ for S, this step can be done with work $\mathcal{O}(|\mathbf{R}| \cdot |\mathbf{S}|^{\varepsilon})$, for arbitrary but fixed $\varepsilon > 0$. In the dictionary setting it can even be done with work $\mathcal{O}(|\mathbf{R}| + |\mathbf{S}|)$. ◁

We note that the first two steps of the algorithm sketched in Example 3.3.1 could be condensed into a single step by checking for $\mathbf{R}'[i].\mathtt{t}[X_2] = \mathbf{S}'[j].\mathtt{t}[X_2]$ instead of $\mathbf{R}'[i].\mathtt{t} = \mathbf{S}'[j].\mathtt{t}$ in the second step. However, this would unnecessarily lead to a more involved interface for our search operation, which is why we refrain from doing so. Furthermore, it would suffice to mark cells of $\mathbf{R}'$ if there is a match in $\mathbf{S}'$. But for our purposes it is often useful to have pointers to such matches.

We model a *pointer* to a cell $\mathbf{A}[i]$ of an array $\mathbf{A}$ as a pair (m, i) where m is the address of the first memory cell of $\mathbf{A}$. Given a pointer, a processor can thus not only determine the contents of $\mathbf{A}[i]$ but also (properties of) the array $\mathbf{A}$, in particular, its length. Furthermore, a pointer can be stored in two memory cells, and we can hence assume that a constant number of pointers can be stored in each cell $\mathbf{B}[j]$ of an array $\mathbf{B}$ whose cells correspond to multiple (but constantly many) memory cells. If a cell $\mathbf{B}[j]$ is inhabited and contains a pointer to an inhabited cell $\mathbf{A}[i]$, we also say that $\mathbf{B}[j].\mathtt{t}$ is *linked* to $\mathbf{A}[i].\mathtt{t}$. If $\mathbf{A}[i]$ also contains a pointer to $\mathbf{B}[j]$, we say that the tuples $\mathbf{A}[i].\mathtt{t}$ and $\mathbf{B}[j].\mathtt{t}$ are *mutually linked*.

We will often "chain" pointers. In Example 3.3.1, for instance, the cells $\mathbf{R}[i]$ and $\mathbf{R}'[i]$ can easily be augmented by pointers to each other, for each inhabited cell $\mathbf{R}[i]$. The same is true for the cells of $\mathbf{S}$ and $\mathbf{S}'$. It is then possible to follow the chain of pointers from $\mathbf{R}$ to matching tuples in $\mathbf{S}$. For example, $\mathbf{R}[1].\mathtt{t} = (1, 5)$ is mutually linked to $\mathbf{R}'[1].\mathtt{t} = 5$, which is in turn linked to $\mathbf{S}'[3].\mathtt{t} = 5$. Since $\mathbf{S}'[3].\mathtt{t} = 5$ is linked to $\mathbf{S}[3].\mathtt{t} = (5, 5)$, a single processor can obtain the matching tuple $(5, 5)$ in $\mathbf{S}$ for $(1, 5)$ in three steps.

The next example illustrates that it is desirable that our search operations links all occurrences of a proper tuple in an array $\mathbf{A}$ to the same occurrence in the "target" array $\mathbf{B}$.

Example 3.3.2. We sketch a "high-level" $\mathcal{O}(1)$-time parallel algorithm to evaluate the projection $\pi_{X_2}(R)$ where R is again the relation from Example 3.2.4. As in Example 3.3.1 we assume that R is given by the array $\mathbf{R}$ with the induced tuple sequence $(1, 5), (3, 4), (7, 3), \bot, (8, 3)$.

First, the algorithm computes the array $\mathbf{R}'$ with induced tuple sequence $5, 4, 3, \bot, 3$ which represents $\pi_{X_2}(R)$. This can be done exactly as in the first step of the algorithm sketched in Example 3.3.1.

While $\mathbf{R}'$ represents $\pi_{X_2}(R)$, it does *not* do so concisely. To get an array that represents $\pi_{X_2}(R)$ concisely, the algorithm has to eliminate duplicates. For this purpose, we will utilize a search operation that, given two arrays $\mathbf{A}$ and $\mathbf{B}$, links all occurrences of a proper tuple $\tilde{a}$ in $\mathbf{A}$ to the same occurrence of $\tilde{a}$ in $\mathbf{B}$, if it occurs at all. Invoking this search operation with both input arrays set to $\mathbf{R}'$, yields pointers from the inhabited cells of $\mathbf{R}'$ to inhabited cells of $\mathbf{R}'$. More precisely, $\mathbf{R}'[1].\mathtt{t} = 5$ and $\mathbf{R}'[2].\mathtt{t} = 4$ are linked to themselves, since they occur exactly once in $\mathbf{R}'$. The tuples $\mathbf{R}'[3].\mathtt{t} = 3$ and $\mathbf{R}'[5].\mathtt{t} = 3$ are either *both* linked to $\mathbf{R}'[3].\mathtt{t}$ or *both* linked to $\mathbf{R}'[5].\mathtt{t}$.

To complete the deduplication, it then suffices to flag all cells of $\mathbf{R}'$ as uninhabited that do *not* contain pointers to themselves. Depending on the output of the search operation the induced tuple sequence of the resulting array is either $5, 4, \perp, \perp, 3$ or $5, 4, 3, \perp, \perp$. In either case, it represents $\pi_{X_2}(R)$ concisely.

We observe that the cells flagged as uninhabited by this deduplication procedure still contain pointers to the cell containing the "representative", that is, a cell with the same tuple as previously stored in the now uninhabited cell. This allows us to effectively "redirect" incoming pointers. Suppose that the resulting array has the induced tuple sequence $5, 4, \perp, \perp, 3$. Thanks to the search operation $\mathbf{R}'[3]$ contains a pointer to $\mathbf{R}'[5]$. Thus, all incoming pointers to $\mathbf{R}'[3]$ can be "redirected" to $\mathbf{R}'[5]$, which contains the same tuple, namely $\mathbf{R}'[5].\mathtt{t} = 3$, as $\mathbf{R}'[3]$ did before the deduplication step. ◁

Our search operation will satisfy the requirement in Example 3.3.2. Indeed, it will be a convenient feature for other procedure besides deduplication as well, and it is "free" in the sense that it does *not* affect our work and space bounds negatively.

Basic Array Operations. We now describe our basic array operations which we will use as building blocks in the remainder of this chapter for query evaluation algorithms.

Just as in Examples 3.3.1 and 3.3.2, the input for our operations usually consists of arrays representing relations, and the output is again an array whose cells are augmented by pointers. In fact, each time a tuple of an output array results from some tuple of an input array, they are linked.

We emphasize that the input (and output) arrays of the following operations do *not* necessarily have to represent relations concisely, some operations are even only meaningful if they do *not*. Moreover, in this section, we do *not* make any assumption on the compactness of arrays.

In the following the parameter $\lambda > 0$ is always a constant. Furthermore, we say that an array $\mathbf{B}$ *contains all proper tuples of an array* $\mathbf{A}$ if there is a bijection f between the inhabited cells of $\mathbf{B}$ and $\mathbf{A}$ such that $\mathbf{A}[i].\mathtt{t} = \mathbf{B}[f(i)].\mathtt{t}$ holds for all inhabited cells $\mathbf{A}[i]$. We call $\mathbf{B}[f(i)]$ the *cell of* $\mathbf{B}$ *corresponding to* $\mathbf{A}[i]$.

- $\mathtt{Compact}_\lambda(\mathbf{A})$ computes a λ-compact array $\mathbf{B}$ that contains all proper tuples of $\mathbf{A}$. Pointers are added – in both directions – between each inhabited cell $\mathbf{A}[i]$ and the corresponding cell of $\mathbf{B}$. Furthermore, this operation preserves the relative order of proper tuples.

- $\mathtt{Sort}_\lambda(\mathbf{A}, \mathcal{X})$ returns an $\mathcal{X}$-ordered array $\mathbf{B}$ of length $(1 + \lambda)|\mathbf{A}|$ which contains all proper tuples of $\mathbf{A}$. Here $\mathcal{X}$ is an ordered list of attributes from the relation represented by $\mathbf{A}$. Pointers are added – in both directions – between each inhabited cell $\mathbf{A}[i]$ and the corresponding cell of $\mathbf{B}$.

- $\mathtt{SearchRepresentatives}(\mathbf{A}, \mathbf{B})$ augments every inhabited cell $\mathbf{A}[i]$ by a pointer to an inhabited *representative cell* $\mathbf{B}[j]$, such that $\mathbf{B}[j].\mathtt{t} = \mathbf{A}[i].\mathtt{t}$ holds, if such a cell exists. Furthermore, for all indices i_1, i_2 with $\mathbf{A}[i_1].\mathtt{t} = \mathbf{A}[i_2].\mathtt{t} \neq \perp$, both $\mathbf{A}[i_1]$ and $\mathbf{A}[i_2]$ are augmented by a pointer to the same representative.

We stress that $\mathbf{B}$ does *not* have to represent its relation concisely, and, in fact, the operation is used to remove duplicates.

Moreover, we note that it is not required that the computed representatives are the same for different arrays representing the same relation.[6]

- `Deduplicate`($\mathbf{A}$) determines, for each maximal set $\mathcal{I}$ of indices of inhabited cells of $\mathbf{A}$ with $\mathbf{A}[i_1].\mathtt{t} = \mathbf{A}[i_2].\mathtt{t}$ for all $i_1, i_2 \in \mathcal{I}$, a representative $j \in \mathcal{I}$. It adds pointers from each cell $\mathbf{A}[j]$ with $j \in \mathcal{I}$ to $\mathbf{A}[i]$ and flags every cell $\mathbf{A}[j]$ with $j \in \mathcal{I}$, $j \neq i$ as uninhabited.

3.3.1 Algorithmic Techniques

In this section, we describe some important algorithmic techniques which we will use to implement algorithms for our basic array operations, in particular, `SearchRepresentatives`.

Here and in the remainder of this chapter, D always denotes the underlying database. Note that the size of an input relation R (or array) and D can vary drastically. For instance, D may contain another relation besides R whose size is exponential in $|R|$.

Array Hash Tables. In the dictionary setting we use *array hash tables* which associate each inhabited cell of $\mathbf{A}$ with an integer from $[1, |\mathbf{A}|]$, such that two cells $\mathbf{A}[i], \mathbf{A}[j]$ get the same number if and only if $\mathbf{A}[i].\mathtt{t} = \mathbf{A}[j].\mathtt{t}$ holds. They follow a similar idea as the injective mapping key, which associates every domain value with an integer in the dictionary setting. However, their range depends on the array, *not* necessarily on the database, and even for arrays representing unary relations, the associated values are independent of the mapping key. Array hash tables can be efficiently computed.

Lemma 3.3.3. *For every $\varepsilon > 0$, there is a $\mathcal{O}(1)$-time parallel algorithm that, in the dictionary setting, computes an array hash table for a given array $\mathbf{A}$. It requires work $\mathcal{O}(|\mathbf{A}|)$ and space $\mathcal{O}(|\mathbf{A}| \cdot |D|^{\varepsilon})$ on an arbitrary CRCW PRAM.*[7]

We note that due to the "arbitrary" resolution of concurrent write, the result of the algorithm guaranteed by Lemma 3.3.3 is not uniquely determined by in the input array $\mathbf{A}$.

Recall that, in the dictionary setting, we understand all domain values as small numbers of size at most $c_{\mathcal{S}}|D|$, for some constant $c_{\mathcal{S}}$.

Proof. We first present the algorithm for $\varepsilon \geq 1$ and, afterwards, discuss how it can be adapted for the case $\varepsilon < 1$.

Let $X_1, \ldots, X_\ell$ be the attributes of the relation R represented by $\mathbf{A}$ in an arbitrary but fixed order and let $\mathcal{X}_j = \{X_1, \ldots, X_j\}$, for all $j \in [1, \ell]$. The algorithm inductively computes hash values for proper tuples in $R[\mathcal{X}_j]$ for increasing j from 1 to ℓ.

[6] In the dictionary setting with arbitrary CRCW PRAMs it is even possible that different applications for the *same* input arrays yield different representatives.

[7] We note that the constant hidden by the asymptotic bounds depends on the arity of the relation represented by $\mathbf{A}$.

The idea is to assign, to each tuple $\tilde{a} \in R[\mathcal{X}_j]$, a processor number in the range $[1, |\mathbf{A}|]$ as hash value and augment each cell of $\mathbf{A}$ containing a proper tuple $\tilde{a}'$ with $\tilde{a}'[\mathcal{X}] = \tilde{a}$ by this hash value. Since the same (projected) tuple $\tilde{a} \in R[\mathcal{X}_j]$ might occur in multiple, pairwise different, cells of $\mathbf{A}$, it does not suffice to load all tuples in $\mathbf{A}$ to $|\mathbf{A}|$ processors and let each processor augment the tuple loaded to it by its processor number: Multiple (different) numbers might get assigned to the same tuple (in different cells of $\mathbf{A}$). To resolve these conflicts, the algorithm utilizes that the domain values in the dictionary setting are small numbers and the hash values for tuples in $R[\mathcal{X}_{j-1}]$.

For the base case $j = 1$ the algorithm allocates an auxiliary array of size $c_{\mathcal{S}}|D|$, where $c_{\mathcal{S}}$ is the constant guaranteed in the dictionary setting such that every domain value is smaller or equal to $c_{\mathcal{S}}|D|$. For each $i \in [1, |\mathbf{A}|]$, processor i is assigned to the tuple $\tilde{a}_i$ in cell $\mathbf{A}[i]$. The projection $\tilde{a}[X_1]$ is a domain value that can be used as index for the auxiliary array. Thus, each processor i with a proper tuple $\tilde{a}_i$ can write its processor number i into cell $\tilde{a}_i[X_1]$ of the auxiliary array. Then it assigns to $\tilde{a}_i$ the value actually written to the cell with index $\tilde{a}_i[X_1]$. Note that, for each value a, all processors i with $\tilde{a}_i[X_1] = a$ will assign the same value to their tuple $\tilde{a}_i$, since precisely one processor among the processors with $\tilde{a}_i[X_1] = a$ succeeds in writing its number to the cell with index $\tilde{a}_i[X_1]$ on an arbitrary CRCW PRAM. This can be done with work $\mathcal{O}(|\mathbf{A}|)$ and space $\mathcal{O}(|\mathbf{A}| + |D|)$.

For $j > 1$ the algorithm proceeds similarly but also takes, for a tuple $\tilde{a}$, the hash value $h_{j-1}(\tilde{a}[\mathcal{X}_{j-1}])$ for $\tilde{a}[\mathcal{X}_{j-1}]$ into account, in addition to $\tilde{a}[X_j]$. For this purpose, the algorithm first computes the hash values for $R[\mathcal{X}_{j-1}]$. It then allocates an auxiliary array of size $|\mathbf{A}| \cdot c_{\mathcal{S}}|D|$ which is interpreted as a two-dimensional array and each processor i writes its number into the cell with index $(h_{j-1}(\tilde{a}_i[\mathcal{X}_{j-1}]), \tilde{a}_i[\mathcal{X}_j])$ of the auxiliary array, if $\tilde{a}_i$ is a proper tuple. The number in this cell is then the hash value for $\tilde{a}_i[\mathcal{X}_j]$.

Writing and reading back the processor numbers requires work $\mathcal{O}(|\mathbf{A}|)$ and the auxiliary array requires space $\mathcal{O}(|\mathbf{A}| \cdot |D|)$. We note that it is *not* necessary to initialize the auxiliary array, since all cells read are written to before. The same bounds hold for the recursive invocations. Since the recursion depth is ℓ, the procedure requires work $\mathcal{O}(\ell \cdot |\mathbf{A}|) = \mathcal{O}(|\mathbf{A}|)$ and, because the space for the auxiliary arrays can be reused, space $\mathcal{O}(|\mathbf{A}| \cdot |D|)$ in total. Since we assumed $\varepsilon \geq 1$, this implies the statement of the lemma for this case.

For the case $\varepsilon < 1$, the algorithm replaces each attribute X by multiple attributes in a preprocessing phase. To this end, we observe that every domain value $\tilde{a}[X]$ can be written as a sum $\sum_{i=0}^{k} b_i \cdot |D|^{i\varepsilon}$, where $k = \lceil \frac{1}{\varepsilon} \rceil$ and the b_i have size at most $c_{\mathcal{S}} \cdot |D|^{\varepsilon}$. Therefore, the attribute X can be replaced by a sequence $Y_0, \ldots, Y_k$ of attributes and the values for X in $\mathbf{A}$ by values of size at most $\mathcal{O}(|D|^{\varepsilon})$ for $Y_0, \ldots, Y_k$. Doing this for every original attribute yields tuples that can be understood as tuples from a database with domain values of size $\mathcal{O}(|D|^{\varepsilon})$. Applying the procedure to compute hash values as detailed above, then yields an array hash table, which is also an array hash table for the original array X. Since k is a constant, the procedure operates within the desired work and space bounds. □

In the dictionary setting, array hash tables will provide the means for an efficient implementation of `SearchRepresentatives`. Alternatively, searching in ordered arrays

is also possible quite efficiently, and they are also available in the ordered setting.

Search in Ordered Arrays. To deal with uninhabited cells our search algorithm for ordered arrays requires pointers from each cell to the next and previous inhabited cell. We refer to those pointers as predecessor and successor links, respectively, and say that an array is *fully linked* if all its cells are augmented by predecessor and successor links.

Proposition 3.3.4. *For every $\varepsilon > 0$, there is a $\mathcal{O}(1)$-time parallel algorithm that computes, for an array $\mathbf{A}$, predecessor and successor pointers with work and space $\mathcal{O}(|\mathbf{A}|^{1+\varepsilon})$ on a common CRCW PRAM.*

Proof. We describe the computation of predecessor links. Successor links can be computed analogously. Let $n = |\mathbf{A}|$ and $\delta = \frac{\varepsilon}{2}$. In the first round, the algorithm considers (non-overlapping) subarrays of length at most n^δ and establishes predecessor links within them. To this end, it uses, for each subarray, a two-dimensional auxiliary array of length $n^\delta \times n^\delta$. For each pair (i, j) with $i < j$ the cell at (i, j) in the auxiliary array is initialized with value 1, if $\mathbf{A}[i]$ is inhabited and, otherwise, by 0. Next, for each triple i, j, k, the cell at (i, j) is set to 0 if $i < k < j$ and $\mathbf{A}[k]$ is inhabited. It is easy to see that afterwards the cell at (i, j) still carries value 1 if and only if i is the predecessor of j. Therefore, for all such pairs (i, j), the cell $\mathbf{A}[j]$ is augmented by a pointer to $\mathbf{A}[i]$.

For every subarray, $(n^\delta)^3 = n^{3\delta}$ processors suffice for this computation, i.e. one processor for each triple i, j, k. Since there are $\lceil \frac{n}{n^\delta} \rceil = \lceil n^{1-\delta} \rceil$ subarrays, the first round requires overall work $\mathcal{O}(n^{1-\delta} \cdot n^{3\delta}) = \mathcal{O}(n^{1+2\delta}) = \mathcal{O}(n^{1+\varepsilon})$. Similarly, the algorithm uses $\lceil n^{1-\delta} \rceil$ many auxiliary arrays of length $n^{2\delta}$ each. Thus, it requires space $\mathcal{O}(n^{1+\delta})$ which is bounded by $\mathcal{O}(n^{1+\varepsilon})$.

In the next round, subarrays of length $n^{2\delta}$ are considered and each is viewed as an array of n^δ smaller subarrays of length n^δ. The goal in the second round is to establish predecessor pointers for the minimum cells of each of the smaller subarrays. This can be done similarly with the same asymptotic work and space as round 1. After $\lceil \frac{1}{\delta} \rceil$ rounds, this process has established predecessor links for all cells (besides for the minimum cells without a predecessor). □

To search for a single tuple in an ordered array in constant time, we will employ a generalization of the classical binary search. Instead of dividing the search space in half in every step, our generalization will divide it by a magnitude of n^ε, which yields a constant recursion depth.

Proposition 3.3.5. *For every $\varepsilon > 0$, there is a $\mathcal{O}(1)$-time parallel algorithm that computes, for a given tuple $\tilde{a}$ over some ordered list $\mathcal{X}$ of attributes, and an $\mathcal{X}$-ordered, fully linked array $\mathbf{A}$, the largest index i with $\mathbf{A}[i].t[\mathcal{X}] \leq \tilde{a}$ (or the smallest index j with $\mathbf{A}[j].t[\mathcal{X}] \geq \tilde{a}$). It requires work and space $\mathcal{O}(|\mathbf{A}|^\varepsilon)$ on a common CRCW PRAM.*

Proof. Let $n = |\mathbf{A}|$. In the first round, using n^ε processors, the algorithm tests for all cells with indices $k = in^{1-\varepsilon}$ for $0 \leq i < n^\varepsilon$ whether $\mathbf{A}[k]$, or its predecessor if it is uninhabited, contains a tuple $\tilde{b}$ such that $\tilde{b}[\mathcal{X}] \leq \tilde{a}$ holds, and whether this does *not* hold

for index $(i+1)n^{1-\varepsilon}$ or its successor. The search then continues recursively in the thus identified subarray of length $n^{1-\varepsilon}$. After at most

$$\lceil \log_{n^\varepsilon} n \rceil = \left\lceil \frac{\log n}{\log n^\varepsilon} \right\rceil = \left\lceil \frac{\log n}{\varepsilon \log n} \right\rceil = \left\lceil \frac{1}{\varepsilon} \right\rceil$$

rounds it terminates. Since, in each round, n^ε processors are used, the statement follows. □

We note that analogously it is possible to search for m tuples in parallel with work and space $\mathcal{O}(m|\mathbf{A}|^\varepsilon)$.

3.3.2 Algorithms for Basic Array Operations

We are now ready to present algorithms for our basic array operations.

For `SearchRepresentatives`$(\mathbf{A}, \mathbf{B})$ we present four algorithms, depending on the setting and, for ordered array, whether $\mathbf{A}$ or $\mathbf{B}$ is suitably ordered. This operation is crucial for deduplication, semi-join and join; and the upper bounds impact the bounds for those operations as well. We stress that ordered arrays are *not* available in the general setting.

Lemma 3.3.6. *For every $\varepsilon > 0$, there are $\mathcal{O}(1)$-time parallel algorithms for* `SearchRepresentatives` *that, given arrays $\mathbf{A}$ and $\mathbf{B}$, have the following bounds on an arbitrary CRCW PRAM.*

- **(a)** *Work $\mathcal{O}(|\mathbf{A}| + |\mathbf{B}|)$ and space $\mathcal{O}((|\mathbf{A}| + |\mathbf{B}|) \cdot |D|^\varepsilon)$ in the dictionary setting.*
- **(b)** *Work $\mathcal{O}(|\mathbf{A}| \cdot |\mathbf{B}|^\varepsilon)$ and space $\mathcal{O}(|\mathbf{A}| \cdot |\mathbf{B}|^\varepsilon + |\mathbf{B}|)$, if $\mathbf{B}$ is fully ordered and fully linked.*
- **(c)** *Work $\mathcal{O}((|\mathbf{A}| + |\mathbf{B}|) \cdot |\mathbf{A}|^\varepsilon)$ and space $\mathcal{O}((|\mathbf{A}| + |\mathbf{B}|) \cdot |\mathbf{A}|^\varepsilon)$, if $\mathbf{A}$ is fully ordered.*
- **(d)** *Work $\mathcal{O}((|\mathbf{A}| + |\mathbf{B}|) \cdot |\mathbf{B}|)$ and space $\mathcal{O}(|\mathbf{A}| + |\mathbf{B}|)$ in the general setting.*

Proof. For Statement (a), first an array hash table for (the concatenation of) $\mathbf{A}$ and $\mathbf{B}$ is computed. The concatenation of can easily be constructed by copying the cell contents with $|\mathbf{A}| + |\mathbf{B}|$ processors in parallel. Thanks to Lemma 3.3.3 the array hash table can thus be computed with work $\mathcal{O}(|\mathbf{A}| + |\mathbf{B}|)$ and space $\mathcal{O}((|\mathbf{A}| + |\mathbf{B}|) \cdot |D|^\varepsilon)$ in total.

For a proper tuple $\tilde{a}$, let $h(\tilde{a})$ denote the hash value in the range $[1, |\mathbf{A}| + |\mathbf{B}|]$ assigned to $\tilde{a}$. The algorithm then allocates an auxiliary array of length $|\mathbf{A}| + |\mathbf{B}|$ and, for each inhabited cell $\mathbf{B}[j]$, it writes, in parallel, j into cell $h(\mathbf{B}[j].\mathtt{t})$ of the auxiliary array. Other processors might attempt to write an index to cell $h(\mathbf{B}[j].\mathtt{t})$ as well, but only one will succeed. This requires work $\mathcal{O}(|\mathbf{B}|)$, and space $\mathcal{O}(|\mathbf{A}| + |\mathbf{B}|)$.

For each inhabited cell $\mathbf{A}[i]$ of $\mathbf{A}$ it is then checked in parallel, if cell $h(\mathbf{A}[i].\mathtt{t})$ contains an index j of $\mathbf{B}$. If it does, then $\mathbf{A}[i]$ augmented by a pointer to cell $\mathbf{B}[j]$, since $\mathbf{B}[j].\mathtt{t} = \mathbf{A}[i].\mathtt{t}$. If *not*, then $\mathbf{A}[i].\mathtt{t}$ does *not* occur in $\mathbf{B}$, and thus is *not* augmented by a pointer.

Towards Statement (b), the algorithm identifies, for each inhabited cell $\mathbf{A}[i]$ of $\mathbf{A}$, the smallest index j of $\mathbf{B}$ such that $\mathbf{A}[i].\mathtt{t} \leq \mathbf{B}[j].\mathtt{t}$. If $\mathbf{A}[i].\mathtt{t} = \mathbf{B}[j].\mathtt{t}$ holds, $\mathbf{A}[i]$ is

augmented by a pointer to the cell $\mathbf{B}[j]$. For each inhabited cell of $\mathbf{A}$, this can be done with work and space $|\mathbf{B}|^\varepsilon$, thanks to Proposition 3.3.5 and these searches can be done in parallel by assigning $|\mathbf{B}|^\varepsilon$ processors per cell of $\mathbf{A}$.

For Statement (c), the algorithm first computes predecessor and successor pointers for $\mathbf{A}$, if necessary. In this case this can be done within the stated bounds, thanks to Proposition 3.3.4. The algorithm then determines, for each inhabited cell $\mathbf{B}[j]$ of $\mathbf{B}$, the smallest index i of $\mathbf{A}$ with $\mathbf{A}[i].\mathtt{t} \geq \mathbf{B}[j].\mathtt{t}$. If $\mathbf{A}[i].\mathtt{t} = \mathbf{B}[j].\mathtt{t}$ then the cell $\mathbf{A}[i]$ is augmented by a pointer to the cell $\mathbf{B}[j]$. We note that, if $\mathbf{B}$ is *not* concise, multiple processors might attempt to write a pointer to a cell $\mathbf{A}[i]$. However, only one processor succeeds and thereby determines the representative of $\mathbf{A}[i].\mathtt{t}$ in $\mathbf{B}$. If $\mathbf{A}$ is guaranteed to be concise, that's all. Otherwise, for each inhabited cell $\mathbf{A}[i]$ of $\mathbf{A}$ the smallest index i' of an inhabited of $\mathbf{A}$ with $\mathbf{A}[i'].\mathtt{t} = \mathbf{A}[i].\mathtt{t}$ is determined. If it was augmented by a pointer to $\mathbf{B}$, then $\mathbf{A}[i]$ is also augmented by this pointer. The work and space bounds are again thanks to Proposition 3.3.5.

For Statement (d), the naive algorithm can be used. Since $\mathbf{B}$ is possibly *not* concise the algorithm first determines representatives for the tuples in $\mathbf{B}$. For that purpose, it uses one processor for each pair (i, j) of indices of $\mathbf{B}$ to flag duplicates in $\mathbf{B}$: If $i < j$ and $\mathbf{B}[i].\mathtt{t} = \mathbf{B}[j].\mathtt{t}$, then $\mathbf{B}[j]$ is flagged. It then uses one processor for each pair (i, j) of indices for $\mathbf{A}$ and $\mathbf{B}$, respectively, to establish pointers from $\mathbf{A}[i]$ to $\mathbf{B}[j]$ if $\mathbf{A}[i].\mathtt{t} = \mathbf{B}[j].\mathtt{t}$ and $\mathbf{B}[j]$ is *not* flagged. □

As suggested in Example 3.3.2, `SearchRepresentatives` can be used to implement `Deduplicate`.

Lemma 3.3.7. *For every $\varepsilon > 0$, there are $\mathcal{O}(1)$-time parallel algorithms for* `Deduplicate` *that, given an array $\mathbf{A}$, have the following bounds on an arbitrary CRCW PRAM.*

(a) *Work $\mathcal{O}(|\mathbf{A}|)$ and space $\mathcal{O}(|\mathbf{A}| \cdot |D|^\varepsilon)$ in the dictionary setting.*

(b) *Work $\mathcal{O}(|\mathbf{A}|^{1+\varepsilon})$ and space $\mathcal{O}(|\mathbf{A}|^{1+\varepsilon})$, if $\mathbf{A}$ is fully ordered.*

(c) *Work $\mathcal{O}(|\mathbf{A}|^2))$ and space $\mathcal{O}(|\mathbf{A}|)$ in the general setting.*

Proof. In all three cases, `SearchRepresentatives`$(\mathbf{A}, \mathbf{A})$ is invoked and afterwards all inhabited cells that were augmented by a pointer to a different cell are made uninhabited, leaving exactly one cell for each tuple that occurs in $\mathbf{A}$ inhabited. For Statement (b), predecessor and successor pointers can be computed, if necessary, thanks to Proposition 3.3.4. Incoming pointers can be redirected using the pointers established by `SearchRepresentatives`$(\mathbf{A}, \mathbf{A})$. The bounds follow with Lemma 3.3.6. □

The algorithm for $\mathtt{Compact}_\lambda$ and $\mathtt{Sort}_\lambda$ are implied by the results already discussed in Section 3.1.2.

Lemma 3.3.8 [GZ95, implied by Theorem 4.2]**.** *For every constant $\lambda > 0$, there is a $\mathcal{O}(1)$-time parallel algorithm for $\mathtt{Compact}_\lambda$ that, given an array $\mathbf{A}$, requires work and space $\mathcal{O}(|\mathbf{A}|^{1+\varepsilon})$ on a common CRCW PRAM.*

Lemma 3.3.8 follows from Proposition 3.1.6 by observing that for any constant $\lambda > 0$ there is an integer c such that $\lambda < (\log n)^{-c}$ holds for all $n > 2$. We should, however, explain how the pointers are established: Instead of compacting $\mathbf{A}$ directly, the algorithm allocates an array $\mathbf{A}'$ of the same length as $\mathbf{A}$ and sets $\mathbf{A}'[i] = i$ if $\mathbf{A}$ is inhabited, and leaves $\mathbf{A}'[i]$ empty otherwise. Then $\mathbf{A}'$ is compacted using Proposition 3.1.6. Let $\mathbf{B}'$ be the resulting array. Using $|\mathbf{B}'|$ many processors the desired output array $\mathbf{B}$ can then be compiled as follows: Processor j checks whether $\mathbf{B}'[j]$ contains an index i. If yes, it sets $\mathbf{B}[j] = \mathbf{A}[i]$ and establishes pointers – in both directions – between $\mathbf{B}[j]$ and $\mathbf{A}[i]$. This procedure requires only work and space $\mathcal{O}(|\mathbf{A}|)$ in addition to the compaction. The bounds are therefore inherited from Proposition 3.1.6.

For the $\texttt{Sort}_\lambda$ we only provide an algorithm for the dictionary setting. We discuss the situation for sorting in the ordered setting in Section 3.7. In short, we are *not* aware of a constant-time, comparison-based sort algorithm for PRAMs.

Lemma 3.3.9. *For every constant $\lambda > 0$, there is a $\mathcal{O}(1)$-time parallel algorithm for $\texttt{Sort}_\lambda$ that, given an array $\mathbf{A}$ and list $\mathcal{X}$ of attributes, requires work and space $\mathcal{O}((|\mathbf{A}| + |D|)^{1+\varepsilon})$ on a common CRCW PRAM in the dictionary setting.*[8]

Proof. Let $\tilde{a}_1, \dots, \tilde{a}_n$ with $n = |\mathbf{A}|$ be the induced tuple sequence of $\mathbf{A}$. We assume that $\mathcal{X} = (X_1, \dots, X_k)$ contains all attributes of the tuples in $\mathbf{A}$. Otherwise, $\mathcal{X}$ can be extended arbitrarily. Recall that we denote the lexicographic order induced by $\mathcal{X}$ on the tuples occurring in $\mathbf{A}$ by $\leq_{\mathcal{X}}$.

For each proper tuple $\tilde{a}_i$, the algorithm computes its characteristic number

$$c_i = (|\mathbf{A}| + 1) \sum_{j=1}^{k} \tilde{a}_i[X_j] m^{k-j} + i$$

where $m = c_{\mathcal{S}}|D| + 1$. Here, the purpose of the factor $|\mathbf{A}| + 1$ and the addend i is to keep track of the original index i. Observe that, since domain values in the dictionary setting are small numbers, m is larger than any value in $\mathbf{A}$. Thus, a processor can derive $\tilde{a}_i$ from c_i in constant time. Furthermore, we have $\tilde{a}_i \leq_{\mathcal{X}} \tilde{a}_j$ if and only if $c_i \leq c_j$ for all $i, j \in [1, n]$. For uninhabited cells, we can just fix $c_i = 0$.

Thanks to Proposition 3.1.11 an array $\mathbf{B}$ of length $(1 + \lambda) \max\{|\mathbf{A}|, |D|\}$ containing the c_i in order can be computed in constant time with work and space $\mathcal{O}((|\mathbf{A}| + |D|)^{1+\varepsilon})$. The length depends on D because to apply Proposition 3.1.11 the c_i have to be bounded by a polynomial in the length of the input array. Thus, it might be necessary to pad $\mathbf{A}$ with zeros to yield an array of length $|D|$, since the c_i depend on $|\mathbf{A}|$ *and* $|D|$. If necessary, $\mathbf{B}$ can be compacted using $\texttt{Compact}_{\lambda'}$ for a sufficiently small λ' to yield an array of the desired length.

In a finishing step, the algorithm replaces every $c_i > 0$ in $\mathbf{B}$ with the tuple $\tilde{a}_i$ it encodes. Furthermore, it can derive i from c_i and thus establish pointers between $\mathbf{A}[i]$ and the cell of $\mathbf{B}$ containing $\tilde{a}_i = \mathbf{A}[i].\texttt{t}$. □

[8] Like for the algorithm for computing array hash tables, the constant hidden in the asymptotic bounds depends on the number of attributes.

3.4 Database Operations

In this section, we proceed similarly as in Section 3.3. We start by defining array-based operations for the operators of the relational algebra. The main contribution of this section is the presentation of $\mathcal{O}(1)$-time parallel algorithms for these operations and the analysis of their complexity with respect to the work and space they require. Before we do that in Sections 3.4.2 and 3.4.3, we will first present some lower bounds in Section 3.4.1 that are consequences of the impossibility results of Section 3.1.2.

Array-Based Database Operations. For database operations, unlike for the basic array operations, we require that input relations are given by arrays representing them concisely. Likewise, all algorithms produce output arrays which represent the result relation concisely. However, neither for input nor for output relations we make any assumptions about the compactness of the representations.

More precisely, we actually present $\mathcal{O}(1)$-time parallel algorithms for the following operations (on arrays) which correspond to the operators of the relational algebra (on relations).[9] Similar to the basic array operations, some operations establish pointers which will be convenient when using the operations as building blocks for query evaluation algorithms.

Recall that we denote the arrays representing relations R, S, etc. by $\mathbf{R}$, $\mathbf{S}$, etc.

- $\texttt{Select}_{X=Y}(\mathbf{R})$ returns a new array $\mathbf{R}'$ that represents $\sigma_{X=Y}(R)$ concisely. Here X and Y are attributes of R. Every inhabited cell $\mathbf{R}'[i]$ is augmented by a pointer to a cell $\mathbf{R}[j]$ with $\mathbf{R}[j].\texttt{t} = \mathbf{R}'[i].\texttt{t}$.
- $\texttt{Project}_{\mathcal{X}}(\mathbf{R})$ returns a new array $\mathbf{R}'$ that represents $\pi_{\mathcal{X}}(R)$ concisely. Here $\mathcal{X}$ is a set or sequence of attributes from R. Every inhabited cell $\mathbf{R}'[i]$ is augmented by a pointer to a cell $\mathbf{R}[j]$ with $\mathbf{R}[j].\texttt{t}[\mathcal{X}] = \mathbf{R}'[i].\texttt{t}$. Conversely, every inhabited cell $\mathbf{R}[j]$ is augmented by a pointer to an inhabited representative cell $\mathbf{R}'[k]$ such that $\mathbf{R}'[k].\texttt{t} = \mathbf{R}[j].\texttt{t}[\mathcal{X}]$ holds.
- $\texttt{Difference}(\mathbf{R}, \mathbf{S})$ returns a new array that represents $R \setminus S$ concisely. Here R and S are relations over the same set of attributes.
- $\texttt{Union}(\mathbf{R}, \mathbf{S})$ returns a new array that represents $R \cup S$ concisely. Here R and S are relations over the same set of attributes.
- $\texttt{SemiJoin}(\mathbf{R}, \mathbf{S})$ returns a new array $\mathbf{R}'$ that represents $R \ltimes S$ concisely. Every inhabited cell $\mathbf{R}'[i]$ is augmented by a pointer to a cell $\mathbf{S}[j]$ such that $\mathbf{S}[j].\texttt{t}[\mathcal{X}] = \mathbf{R}'[i].\texttt{t}[\mathcal{X}]$ holds where $\mathcal{X}$ is the set of common attributes of R and S.
- $\texttt{Join}(\mathbf{R}, \mathbf{S})$ returns a new array that represents $R \bowtie S$ concisely.

We note that for the rename operator $\rho_{X \to Y}(R)$ of the relational algebra, of course, *no* parallel algorithm is required.

[9] We believe that distinguishing between operations on arrays and operations on relations also improves the readability of algorithms and proofs.

3.4.1 Lower Bounds

Before we present our algorithms, we first derive some lower bounds for our database operations from the results discussed in Section 3.1.2.

On the one hand our lower bounds state that there is *no* $\mathcal{O}(1)$-time parallel algorithm using a polynomial number of processors, that computes the semi-join of two unary relations R and S, and stores the result in an array of length $|R \ltimes S|$.

Hence, it is required that the output arrays of algorithms for `SemiJoin` have length $(1+\lambda)|R \ltimes S|$, for some λ. But, if we want to guarantee such a length, then any constant-time algorithm requires $\Omega(n^{1+\varepsilon})$ processors for any $\varepsilon \geq 0$. The same applies to the (natural) join of two unary relations R and S.

Theorem 3.4.1. *Let $c \geq 1$ and $\lambda > 0$ be constants. For every CRCW PRAM algorithm for the operation* `SemiJoin` *(or* `Join`*), the following statements hold.*

(a) *If the algorithm uses a polynomial number of processors and the output array is compact, for every possible input, then it has running time $\Omega\big(\frac{\log n}{\log\log n}\big)$.*

(b) *If the algorithm only needs constant time and the output array is λ-compact, for every possible input, then it requires $\Omega(n^{1+\varepsilon})$ processors, for some constant $\varepsilon > 0$.*

Here n is the size of the input arrays. The lower bounds hold even if the input relations R and S are unary relations, the arrays representing R and S are compact and fully ordered, and in the dictionary setting.

Proof. The lower bounds rely on the lower bounds for (exact) compaction and approximate compaction, and basically use the same reduction.

We first prove Statement (b). Suppose that there is a CRCW PRAM algorithm for `SemiJoin` (or `Join`) that guarantees a λ-compact output array and runs in constant time using $\mu(n)n$ processors, where $\mu(n) = o(n^{\varepsilon})$, for every $\varepsilon > 0$. We show how this algorithm can be used to achieve approximate compaction with $\mu(n)n$ processors in constant time.

Let $\mathbf{A}$ be the input array of length n with k non-empty cells. An algorithm that achieves λ-approximate compaction can proceed as follows.

In the first step it creates an array $\mathbf{R}$ of length n for the unary relation R that consists of the even numbers from 2 to $2n$. This can be easily done in parallel in constant time with n processors.

In the second step, it initializes an array $\mathbf{S}$ of length n as follows. For every $i \in [1, n]$, it stores the value $2i$ in $\mathbf{S}[i]$, if $\mathbf{B}[i]$ is *not* empty, and otherwise the value $2i+1$. Hence, the relation defined by $R \ltimes S$ and $R \bowtie S$ consists of exactly all even numbers $2i$, for which $\mathbf{A}[i]$ is *not* empty.

Note that, by construction, the arrays for the relations R and S are concise, compact, and fully ordered.

From the λ-compact array representing $R \ltimes S$ a λ-compact array containing all values from $\mathbf{B}$ can be obtained by replacing every value $2i$ by $\mathbf{A}[i]$. Indeed, the length of the array is at most $(1+\lambda)|R \ltimes S| = (1+\lambda)|R \bowtie S| = (1+\lambda)k$.

All in all, the algorithm takes constant time and uses $\mu(n)n$ processors to achieve linear approximate compaction. Hence, due to Proposition 3.1.4 it holds $\mu(n)n \in \Omega(n^{1+\varepsilon})$, for some $\varepsilon > 0$.

By the same construction, an algorithm for `SemiJoin` (or `Join`) according to Statement (a) yields an algorithm achieving (perfect) compaction that works in constant time with polynomially many processors, contradicting Corollary 3.1.2. □

The next result states that the best we can hope for in the general setting are quadratic work bounds.

Lemma 3.4.2. *Any $\mathcal{O}(1)$-time parallel algorithm for* `SemiJoin` *or* `Join` *in the general setting requires $\Omega(|D|^2)$ work for infinitely many input databases on a priority CRCW PRAM.*

Proof. We define a family of databases D_n and show that the elemental operation $\texttt{Equal}_{R,S}$ has to be invoked at least n^2 times to evaluate $R \ltimes S$ (or $R \bowtie S$) correctly. For any $n > 0$, let $R_n = \{2, \ldots, 2n\}$ be the set of all even numbers from 2 to $2n$ and $S_n = \{1, \ldots, 2(n-1)+1\}$ be the set of all odd numbers from 1 to $2(n-1)+1$. Note that $|R_n| = |S_n| = n$ and let D_n be the database over schema $\{R, S\}$ with $D_n(R) = R_n$ and $D_n(S) = S_n$. The query result of $R \ltimes S$ on D_n is empty.

Assume for the sake of a contradiction that there is a $\mathcal{O}(1)$-time parallel algorithm for the general setting that evaluates $R \ltimes S$ on D_n with less than n^2 invocations of $\texttt{Equal}_{R,S}$. Then there are i_1, i_2 such that $\texttt{Equal}_{R,S}(i_1, 1, i_2, 1)$ is *never* invoked. Furthermore, by construction any invocation of $\texttt{Equal}_{R,S}$ returns `false` on D_n, and invocations $\texttt{Equal}_{R,R}(i'_1, 1, i'_2, 1)$ or $\texttt{Equal}_{S,S}(i'_1, 1, i'_2, 1)$ return `true` if and only if $i'_1 = i'_2$. Now consider the database D'_n that is defined like D_n except that the i_1-th value of R_n and the i_2-th value of S_n are set to $2n + 1$.

Since every invocation of $\texttt{Equal}_{R,S}(R, S)$ except $\texttt{Equal}_{R,S}(i_1, 1, i_2, 1)$ still returns `false`, and the output for $\texttt{\#Tuples}_R$ and $\texttt{\#Tuples}_S$ is also the same on D'_n as for D_n, we can conclude that the computation on D'_n does *not* diverge from the computation on D_n. In other words, $\texttt{Equal}_{R,S}(i_1, 1, i_2, 1)$ is never invoked, and the algorithm outputs the empty query result. This is a contradiction because the query result of $R \ltimes S$ on D'_n is the singleton set $\{2n + 1\}$. □

3.4.2 Algorithms for the Operations of the Semi-Join Algebra

In this subsection we present our algorithms for the operations that correspond to the operators of the semi-join algebra. They are essentially simple combinations of the algorithms of Section 3.3. The algorithms for the `Join` operation are more involved and are deferred to Section 3.4.3.

To be a bit more precise, we present "high-level" algorithms for the operations of the semi-join algebra which are built upon the basic array operations introduced in Section 3.3. Plugging in the concrete algorithms for the basic operations then yields different algorithms depending on the setting and whether the input arrays are suitably ordered. Therefore, we also state complexity bounds for the ordered setting and the

Table 3.2: Complexity bounds for the operations of the semi-join algebra in the general setting.

Operation	Result	Work bound	Space bound
$\texttt{Select}_{X=Y}(\mathbf{R})$	Proposition 3.4.3	$\mathcal{O}(\lvert\mathbf{R}\rvert)$	$\mathcal{O}(\lvert\mathbf{R}\rvert)$
$\texttt{Project}_{\mathcal{X}}(\mathbf{R})$	Proposition 3.4.4	$\mathcal{O}(\lvert\mathbf{R}\rvert^2)$	$\mathcal{O}(\lvert\mathbf{R}\rvert)$
$\texttt{SemiJoin}(\mathbf{R},\mathbf{S})$	Proposition 3.4.5	$\mathcal{O}((\lvert\mathbf{R}\rvert+\lvert\mathbf{S}\rvert)\cdot\lvert\mathbf{S}\rvert)$	$\mathcal{O}(\lvert\mathbf{R}\rvert+\lvert\mathbf{S}\rvert)$
$\texttt{Difference}(\mathbf{R},\mathbf{S})$	Proposition 3.4.6	$\mathcal{O}((\lvert\mathbf{R}\rvert+\lvert\mathbf{S}\rvert)\cdot\lvert\mathbf{S}\rvert)$	$\mathcal{O}(\lvert\mathbf{R}\rvert+\lvert\mathbf{S}\rvert)$
$\texttt{Union}(\mathbf{R},\mathbf{S})$	Proposition 3.4.7	$\mathcal{O}((\lvert\mathbf{R}\rvert+\lvert\mathbf{S}\rvert)\cdot\lvert\mathbf{S}\rvert)$	$\mathcal{O}(\lvert\mathbf{R}\rvert+\lvert\mathbf{S}\rvert)$

Table 3.3: Complexity bounds for the operations of the semi-join algebra in the dictionary setting.

Operation	Result	Work bound	Space bound
$\texttt{Select}_{X=Y}(\mathbf{R})$	Proposition 3.4.3	$\mathcal{O}(\lvert\mathbf{R}\rvert)$	$\mathcal{O}(\lvert\mathbf{R}\rvert)$
$\texttt{Project}_{\mathcal{X}}(\mathbf{R})$	Proposition 3.4.4	$\mathcal{O}(\lvert\mathbf{R}\rvert)$	$\mathcal{O}(\lvert\mathbf{R}\rvert\cdot\lvert D\rvert^{\varepsilon})$
$\texttt{SemiJoin}(\mathbf{R},\mathbf{S})$	Proposition 3.4.5	$\mathcal{O}(\lvert\mathbf{R}\rvert+\lvert\mathbf{S}\rvert)$	$\mathcal{O}((\lvert\mathbf{R}\rvert+\lvert\mathbf{S}\rvert)\cdot\lvert D\rvert^{\varepsilon})$
$\texttt{Difference}(\mathbf{R},\mathbf{S})$	Proposition 3.4.6	$\mathcal{O}(\lvert\mathbf{R}\rvert+\lvert\mathbf{S}\rvert)$	$\mathcal{O}((\lvert\mathbf{R}\rvert+\lvert\mathbf{S}\rvert)\cdot\lvert D\rvert^{\varepsilon})$
$\texttt{Union}(\mathbf{R},\mathbf{S})$	Proposition 3.4.7	$\mathcal{O}(\lvert\mathbf{R}\rvert+\lvert\mathbf{S}\rvert)$	$\mathcal{O}((\lvert\mathbf{R}\rvert+\lvert\mathbf{S}\rvert)\cdot\lvert D\rvert^{\varepsilon})$

general setting, although we will *not* utilize the algorithms we present here for building query evaluation algorithms for these settings.[10] We stress again that the algorithms assuming ordered input arrays and their bounds do *not* apply to the general setting, since we simply do *not* have access to an order in this setting.

An overview of the work and space bounds of our algorithms for the general, the dictionary, and the ordered setting is given in Table 3.2, Table 3.3, and Table 3.4, respectively.

Let us point out that some of our results in this section state a linear upper bound for the work, in particular for the dictionary setting. These bounds do *not* contradict the lower bounds stated in Theorem 3.4.1 because the results state only trivial constraints on the length (and hence compactness) of the output arrays. Indeed, compacting the output arrays yields the bounds to be expected given Theorem 3.4.1.

Our algorithm for $\texttt{Select}_{X=Y}$ is very simple and does *not* depend on the setting at all.

Proposition 3.4.3. *For all attributes X, Y there is a $\mathcal{O}(1)$-time parallel algorithm for $\texttt{Select}_{X=Y}$ that, given an array $\mathbf{R}$, requires work and space $\mathcal{O}(\lvert\mathbf{R}\rvert)$ on an EREW PRAM. The output array has length $\lvert\mathbf{R}\rvert$. If $\mathbf{R}$ is $\mathcal{X}$-ordered, then the output array is $\mathcal{X}$-ordered.*

Proof. The algorithm simply initializes a new array $\mathbf{R}'$ of length $\lvert\mathbf{R}\rvert$, and then, using $\lvert\mathbf{R}\rvert$ processors, copies $\mathbf{R}[i].\texttt{t}$ to $\mathbf{R}'[i]$ if $\mathbf{R}[i].\texttt{t}[X] = \mathbf{R}[i].\texttt{t}[Y]$ holds. Furthermore, it is trivial to augment every inhabited cell $\mathbf{R}'[i]$ by a pointer to $\mathbf{R}[i]$. Clearly, this takes work and space $\mathcal{O}(\lvert\mathbf{R}\rvert)$, and preserves the order of tuples. □

[10] We will instead opt for a translation into the dictionary setting.

Table 3.4: Complexity bounds for the operations of the semi-join algebra in the ordered (and dictionary) setting. In the conditions for the `SemiJoin` operation, $\mathcal{X}$ is the set of common attributes of R and S.

Operation	Result	Work bound	Space bound
$\texttt{Select}_{X=Y}(\mathbf{R})$	Proposition 3.4.3	$\mathcal{O}(\lvert\mathbf{R}\rvert)$	$\mathcal{O}(\lvert\mathbf{R}\rvert)$
$\texttt{Project}_{\mathcal{X}}(\mathbf{R})$	Proposition 3.4.4	$\mathcal{O}(\lvert\mathbf{R}\rvert^{1+\varepsilon})$	$\mathcal{O}(\lvert\mathbf{R}\rvert^{1+\varepsilon})$ ↪ if $\mathbf{R}$ is $\mathcal{X}$-ordered
$\texttt{SemiJoin}(\mathbf{R},\mathbf{S})$	Proposition 3.4.5	$\mathcal{O}(\lvert\mathbf{R}\rvert \cdot \lvert\mathbf{S}\rvert^{\varepsilon})$ ↪ if $\mathbf{S}$ is $\mathcal{X}$-ordered and fully linked	$\mathcal{O}(\lvert\mathbf{R}\rvert \cdot \lvert\mathbf{S}\rvert^{\varepsilon} + \lvert\mathbf{S}\rvert)$
		$\mathcal{O}((\lvert\mathbf{R}\rvert + \lvert\mathbf{S}\rvert) \cdot \lvert\mathbf{R}\rvert^{\varepsilon})$	$\mathcal{O}((\lvert\mathbf{R}\rvert + \lvert\mathbf{S}\rvert) \cdot \lvert\mathbf{R}\rvert^{\varepsilon})$ ↪ if $\mathbf{R}$ is $\mathcal{X}$-ordered
$\texttt{Difference}(\mathbf{R},\mathbf{S})$	Proposition 3.4.6	$\mathcal{O}(\lvert\mathbf{R}\rvert \cdot \lvert\mathbf{S}\rvert^{\varepsilon})$ ↪ if $\mathbf{S}$ is fully ordered and fully linked	$\mathcal{O}(\lvert\mathbf{R}\rvert \cdot \lvert\mathbf{S}\rvert^{\varepsilon} + \lvert\mathbf{S}\rvert)$
		$\mathcal{O}((\lvert\mathbf{R}\rvert + \lvert\mathbf{S}\rvert) \cdot \lvert\mathbf{R}\rvert^{\varepsilon})$	$\mathcal{O}((\lvert\mathbf{R}\rvert + \lvert\mathbf{S}\rvert) \cdot \lvert\mathbf{R}\rvert^{\varepsilon})$ ↪ if $\mathbf{R}$ is fully ordered
$\texttt{Union}(\mathbf{R},\mathbf{S})$	Proposition 3.4.7	$\mathcal{O}(\lvert\mathbf{R}\rvert \cdot \lvert\mathbf{S}\rvert^{\varepsilon} + \lvert\mathbf{S}\rvert)$ ↪ if $\mathbf{S}$ is fully ordered and fully linked	$\mathcal{O}(\lvert\mathbf{R}\rvert \cdot \lvert\mathbf{S}\rvert^{\varepsilon} + \lvert\mathbf{S}\rvert)$

Proposition 3.4.4. *For every $\varepsilon > 0$ and every set or sequence $\mathcal{X}$ of attributes, there are $\mathcal{O}(1)$-time parallel algorithms for $\texttt{Project}_{\mathcal{X}}$ that, given an array $\mathbf{R}$, have the following bounds on an arbitrary CRCW PRAM.*

- **(a)** *Work $\mathcal{O}(\lvert\mathbf{R}\rvert)$ and space $\mathcal{O}(\lvert\mathbf{R}\rvert \cdot \lvert D\rvert^{\varepsilon})$ in the dictionary setting.*
- **(b)** *Work $\mathcal{O}(\lvert\mathbf{R}\rvert^{1+\varepsilon})$ and space $\mathcal{O}(\lvert\mathbf{R}\rvert^{1+\varepsilon})$, if $\mathbf{R}$ is $\mathcal{X}$-ordered.*
- **(c)** *Work $\mathcal{O}(\lvert\mathbf{R}\rvert^{2})$ and space $\mathcal{O}(\lvert\mathbf{R}\rvert)$ in the general setting.*

The output array has length $\lvert\mathbf{R}\rvert$. If $\mathbf{R}$ is $\mathcal{Y}$-ordered, for some subsequence $\mathcal{Y}$ of $\mathcal{X}$ then the output array is $\mathcal{Y}$-ordered, too.

Proof. The algorithm first allocates an array $\mathbf{R}'$ of length $\lvert\mathbf{R}\rvert$. It then sets $\mathbf{R}'[i].\texttt{t} = \mathbf{R}[i].\texttt{t}[\mathcal{X}]$ and augments $\mathbf{R}'[i]$ by a pointer to $\mathbf{R}[i]$, for every inhabited cell $\mathbf{R}[i]$, using $\lvert\mathbf{R}\rvert$ processors, one for each cell of $\mathbf{R}$. Clearly, this requires work and space $\mathcal{O}(\lvert\mathbf{R}\rvert)$.

The array $\mathbf{R}'$ represents $\pi_{\mathcal{X}}(R)$, but *not* concisely, which is a requirement. Thus, the algorithm invokes $\texttt{Deduplicate}(\mathbf{R}')$ to remove multiple appearances of a tuple in $\mathbf{R}'$, and returns the resulting array. The work and space bounds are dominated by the invocation of `Deduplicate` in any case, and therefore the bounds follow from Lemma 3.3.7.

In the ordered case it is crucial that $\mathbf{R}$ is $\mathcal{X}$-ordered. It implies that $\mathbf{R}'$ is fully ordered (with respect to $\mathcal{X}$). □

Proposition 3.4.5. *For every $\varepsilon > 0$, there are $\mathcal{O}(1)$-time parallel algorithms for* `SemiJoin` *that, given arrays $\mathbf{R}$ and $\mathbf{S}$ have the following bounds on an arbitrary CRCW*

PRAM. Here, $\mathcal{X}$ denotes the joint attributes of the relations R and S represented by $\mathbf{R}$ and $\mathbf{S}$, respectively.

(a) *Work $\mathcal{O}(|\mathbf{R}| + |\mathbf{S}|)$ and space $\mathcal{O}((|\mathbf{R}| + |\mathbf{S}|) \cdot |D|^\varepsilon)$ in the dictionary setting.*

(b) *Work $\mathcal{O}(|\mathbf{R}| \cdot |\mathbf{S}|^\varepsilon)$ and space $\mathcal{O}(|\mathbf{R}| \cdot |\mathbf{S}|^\varepsilon + |\mathbf{S}|)$, if $\mathbf{S}$ is $\mathcal{X}$-ordered and fully linked.*

(c) *Work $\mathcal{O}((|\mathbf{R}| + |\mathbf{S}|) \cdot |\mathbf{R}|^\varepsilon)$ and space $\mathcal{O}((|\mathbf{R}| + |\mathbf{S}|) \cdot |\mathbf{R}|^\varepsilon)$, if $\mathbf{R}$ is $\mathcal{X}$-ordered.*

(d) *Work $\mathcal{O}((|\mathbf{R}| + |\mathbf{S}|) \cdot |\mathbf{S}|)$ and space $\mathcal{O}(|\mathbf{R}| + |\mathbf{S}|)$ in the general setting.*

The output array has length $|\mathbf{R}|$. If $\mathbf{R}$ is $\mathcal{Y}$-ordered, then the output array is $\mathcal{Y}$-ordered.

Proof. We will first describe the "high-level" algorithm for Statements (a), (c) and (d). Afterwards, we discuss how it can be adapted for Statement (b).

The algorithm first computes arrays $\mathbf{R}'$ and $\mathbf{S}'$ which represent $R' = \pi_{\mathcal{X}}(R)$ and $S' = \pi_{\mathcal{X}}(S)$, respectively, but possibly *not* concisely.[11] This can be done as detailed in the proof for Proposition 3.4.4, using $|\mathbf{R}|$ and $|\mathbf{S}|$ processors, respectively. In a nutshell, it sets $\mathbf{R}'[i].\mathtt{t} = \mathbf{R}[i].\mathtt{t}[\mathcal{X}]$, for every inhabited cell $\mathbf{R}[i]$. The same applies to $\mathbf{S}'$ and $\mathbf{S}$. Moreover, if $\mathbf{S}$ (or $\mathbf{R}$) is $\mathcal{X}$-ordered then $\mathbf{S}'$ (resp. $\mathbf{R}'$) is fully ordered (with respect to $\mathcal{X}$).

The algorithm then uses $\mathtt{SearchRepresentatives}(\mathbf{R}', \mathbf{S}')$ to augment every inhabited $\mathbf{R}'[i]$ by a pointer to an inhabited cell $\mathbf{S}'[j]$ with $\mathbf{S}'[j].\mathtt{t} = \mathbf{R}'[i]$, if there is such a cell. Finally, using $|\mathbf{R}|$ processors, the algorithm augments an inhabited cell $\mathbf{R}[i]$ by a pointer to $\mathbf{S}[j]$, if $\mathbf{R}'[i]$ was augmented by a pointer to $\mathbf{S}'[j]$. If *not*, $\mathbf{R}[i]$ is flagged uninhabited.[12] Afterwards, each inhabited cell $\mathbf{R}[i]$ is equipped with a pointer to an inhabited cell $\mathbf{S}[j]$ with $\mathbf{S}[j].\mathtt{t}[\mathcal{X}] = \mathbf{R}[i].\mathtt{t}[\mathcal{X}]$. In particular, $\mathbf{R}$ represents $R \ltimes S$ concisely.

For Statements (a), (c) and (d), the overall work and space are dominated by the invocation of $\mathtt{SearchRepresentatives}$. Hence, the bounds follow from Lemma 3.3.6.

For Statement (b), the array $\mathbf{S}'$ cannot be computed within the stated work bound $\mathcal{O}(|\mathbf{R}| \cdot |\mathbf{S}|^\varepsilon)$. Thus, the algorithm invokes $\mathtt{SearchRepresentatives}(\mathbf{R}', \mathbf{S})$ instead of $\mathtt{SearchRepresentatives}(\mathbf{R}', \mathbf{S}')$. Thereby, it intercepts every read instruction to a cell $\mathbf{S}[i]$ and replaces $\mathbf{S}[i].\mathtt{t}$ with $\mathbf{S}[i].\mathtt{t}[\mathcal{X}]$ "on-the-fly". □

Proposition 3.4.6. *For every $\varepsilon > 0$, there are $\mathcal{O}(1)$-time parallel algorithms for* `Difference` *that, given arrays $\mathbf{R}$ and $\mathbf{S}$ have the following bounds on an arbitrary CRCW PRAM.*

(a) *Work $\mathcal{O}(|\mathbf{R}| + |\mathbf{S}|)$ and space $\mathcal{O}((|\mathbf{R}| + |\mathbf{S}|) \cdot |D|^\varepsilon)$ in the dictionary setting.*

(b) *Work $\mathcal{O}(|\mathbf{R}| \cdot |\mathbf{S}|^\varepsilon)$ and space $\mathcal{O}(|\mathbf{R}| \cdot |\mathbf{S}|^\varepsilon + |\mathbf{S}|)$, if $\mathbf{S}$ is fully ordered and fully linked.*

(c) *Work $\mathcal{O}((|\mathbf{R}| + |\mathbf{S}|) \cdot |\mathbf{R}|^\varepsilon)$ and space $\mathcal{O}((|\mathbf{R}| + |\mathbf{S}|) \cdot |\mathbf{R}|^\varepsilon)$, if $\mathbf{R}$ is fully ordered.*

(d) *Work $\mathcal{O}((|\mathbf{R}| + |\mathbf{S}|) \cdot |\mathbf{S}|)$ and space $\mathcal{O}(|\mathbf{R}| + |\mathbf{S}|)$ in the general setting.*

The output array has length $|\mathbf{R}|$. If $\mathbf{R}$ is $\mathcal{Y}$-ordered, then the output array is $\mathcal{Y}$-ordered.

[11] While it seems natural to compute $\mathbf{R}'$ and $\mathbf{S}'$ using $\mathtt{Project}_{\mathcal{X}}$ this is *not* possible for Statements (b) and (c) in general because $\mathbf{R}'$ or $\mathbf{S}'$ might *not* be $\mathcal{X}$-ordered.

[12] To preserve the input array, the algorithm may actually create a copy of $\mathbf{R}$ first.

Proof. The algorithm uses `SearchRepresentatives`$(\mathbf{R}, \mathbf{S})$ to augment every inhabited cell $\mathbf{R}[i]$ by a pointer to an inhabited cell $\mathbf{S}[j]$ with $\mathbf{S}[j].\mathtt{t} = \mathbf{R}[i].\mathtt{t}$, if there is such a cell. The tuples that belong to the difference $R - S$ are then precisely the tuples stored in inhabited cells of $\mathbf{R}$ that were *not* augmented by a pointer. Thus, the algorithm allocates an output array $\mathbf{R}'$ and, using one processor for each cell $\mathbf{R}[i]$, copies $\mathbf{R}[i].\mathtt{t}$ to $\mathbf{R}'[i]$ if $\mathbf{R}[i]$ is inhabited and is *not* augmented by a pointer.

In every case, the work and space for invoking `SearchRepresentatives` dominates the overall work and space. Thus, the bounds follow from Lemma 3.3.6. We note that the algorithm effectively just removes tuples from $\mathbf{R}$. It is therefore order-preserving. □

Proposition 3.4.7. *For every $\varepsilon > 0$, there are $\mathcal{O}(1)$-time parallel algorithms for* `Union` *that, given arrays $\mathbf{R}$ and $\mathbf{S}$ have the following bounds on an arbitrary CRCW PRAM.*

(a) *Work $\mathcal{O}(|\mathbf{R}| + |\mathbf{S}|)$ and space $\mathcal{O}((|\mathbf{R}| + |\mathbf{S}|) \cdot |D|^{\varepsilon})$ in the dictionary setting.*

(b) *Work $\mathcal{O}(|\mathbf{R}| \cdot |\mathbf{S}|^{\varepsilon} + |\mathbf{S}|)$ and space $\mathcal{O}(|\mathbf{R}| \cdot |\mathbf{S}|^{\varepsilon} + |\mathbf{S}|)$, if $\mathbf{S}$ is fully ordered and fully linked.*

(c) *Work $\mathcal{O}((|\mathbf{R}| + |\mathbf{S}|) \cdot |\mathbf{S}|)$ and space $\mathcal{O}(|\mathbf{R}| + |\mathbf{S}|)$ in the general setting.*

The output array has length $|\mathbf{R}| + |\mathbf{S}|$.

The algorithms basically concatenate the arrays representing $R - S$ and S. We note that thanks to the symmetry of union, the algorithm for Statement (b) can also be applied if $\mathbf{R}$ is fully ordered.

Proof for Proposition 3.4.7. The "high-level" algorithm uses `Difference` to compute an array $\mathbf{R}'$ representing $R' = R - S$, concisely. In a second step it concatenates $\mathbf{R}'$ and $\mathbf{S}$. For that purpose, it allocates an output array of length $|\mathbf{R}'| + |\mathbf{S}| = |\mathbf{R}| + |\mathbf{S}|$, and then copies the proper tuples from $\mathbf{R}'$ and $\mathbf{S}$ into the output array using $|\mathbf{R}'| + |\mathbf{S}|$ processors. To be precise, for each inhabited cell $\mathbf{R}'[i]$, the tuple $\mathbf{R}'[i].\mathtt{t}$ is copied to the i-th cell of the output array and, for each inhabited cell $\mathbf{S}[j]$, the tuple $\mathbf{S}[i].\mathtt{t}$ is copied to the cell with index $|\mathbf{R}'| + j$.

For Statements (a) and (c), the required work and space for `Difference` dominate the work and space required for the second step. Thus, the bounds are inherited from Proposition 3.4.6 in these cases. For Statement (b) the additional addend $|\mathbf{S}|$ is due to the concatenation in the second step which requires work and space $\mathcal{O}(|\mathbf{R}| + |\mathbf{S}|)$, since $|\mathbf{R}'| = |\mathbf{R}|$. □

3.4.3 Algorithms for the Join Operation

In this section, we present our algorithms for the computing the join $R \bowtie S$ of two relations R and S. In a nutshell, they group the tuples in S according to the values of the common attributes of R and S, and then join each tuple $\tilde{a}$ in R with the group for $\tilde{a}[\mathcal{X}]$.

If the input array $\mathbf{S}$ for S is $\mathcal{X}$-ordered, then the tuples in $\mathbf{S}$ are already grouped and it suffices to find the first and last index of each group. For a tuple $\tilde{a} \in R$, we denote by $\mathbf{G}_{\tilde{a}}$ the subarray of $\mathbf{S}$ which contains all tuples $\tilde{b}$ of S with $\tilde{b}[\mathcal{X}] = \tilde{a}[\mathcal{X}]$, i.e. the tuples

matching $\tilde{a}$. Observe that multiple tuples from R might be associated with the same subarray, i.e. if they have the same values for the attributes in $\mathcal{X}$.

Given $\tilde{a}$ and the associated group array $\mathbf{G}_{\tilde{a}}$, the output tuples resulting from $\tilde{a}$ can be written into the output array using $|\mathbf{G}|$ processors. However, if $\mathbf{G}_{\tilde{a}}$ is *not* λ-compact w.r.t. its own length $|\mathbf{G}_{\tilde{a}}|$, this can result in undesirable work bounds (and a large output array). Indeed, even if $\mathbf{S}$ is λ-compact, there might be a group array $\mathbf{G}_{\tilde{a}}$ with $\lambda|S|$ uninhabited cells and two inhabited cells. Assuming that every tuple in R has to be joined with every tuple in this group array $\mathbf{G}_{\tilde{a}}$, writing the output array naively as described above then requires $|R|(2 + \lambda|S|) = 2|R| + \lambda|R||S|$ processors. That is, the work of this procedure is quadratic, even though there are only $2|R|$ output tuples.

To obtain a better work bound, our algorithm compacts each group array $\mathbf{G}_{\tilde{a}}$ in $\mathbf{S}$ in parallel. Since the number of groups is, in general, *not* bounded by a constant, the underlying obstacle for this objective is *processor allocation*: Each processor has to be assigned to a group it then helps to compact. This entails that a single processor has to be able to determine its group in constant time. All while not allocating too many processors in total. Let us point out that our algorithms for the `Join` operation has to resolve processor allocation twice: Once to compact each group as discussed above and then a second time to copy, for each $\tilde{a} \in R$, the output tuples $\{\tilde{a}\} \bowtie \{\tilde{b} \in S \mid \tilde{b}[\mathcal{X}] = \tilde{a}[\mathcal{X}]\}$ into the output array.

Processor allocation can be formalized as follows. A *task description* $d = (m, \bar{x})$ consists of a number m specifying how many processors are required for the task, and a constant number of additional numbers $\bar{x} = (x_1, \ldots, x_k)$ serving as "input" for the task, e.g. $\bar{x}$ may contain pointers to input arrays or a number that indicates which algorithm is used to solve the task. A *task schedule* for a sequence $d_1, \ldots, d_n$ of task descriptions is an array $\mathbf{C}$ of length at least $\sum_{i=1}^{n} m_i$ such that, for every $i \in [1, n]$, there are at least m_i consecutive cells $\mathbf{C}[j_i], \ldots, \mathbf{C}[j_i + m_i - 1]$ with content d_i and each of these cells is augmented by a pointer to $\mathbf{C}[j_i]$, i.e. the cell with the smallest index in the sequence. The tasks specified by $d_1, \ldots, d_n$ can then be solved in parallel using $|\mathbf{C}|$ processors: Processor j can lookup the task it helps to solve in cell $\mathbf{C}[j]$, and using the pointer it can also determine its relative processor number for the task. If a cell $\mathbf{C}[j]$ is empty, processor j does nothing.

The processor allocation problem is closely related to the prefix sums and the *interval allocation* problems. We believe the following lemma is folklore. The relationship of these problems and an analogous result for randomized PRAMs and time $\mathcal{O}(\log^* n)$ are, for instance, discussed by Hagerup [Hag92b, Section 2]. For the sake of completeness we provide a proof for the deterministic constant-time case here.

Lemma 3.4.8. *For every $\varepsilon > 0$ and $\lambda > 0$ there is a $\mathcal{O}(1)$-time parallel algorithm that, given an array $\mathbf{T}$ that contains a sequence of task descriptions $d_1 = (m_1, \bar{x}_1), \ldots, d_n = (m_n, \bar{x}_n)$, computes a task schedule $\mathbf{C}$ for $d_1, \ldots, d_n$ of size $(1 + \lambda) \sum_{i=1} m_i$. It requires work and space $\mathcal{O}(|\mathbf{T}|^{1+\varepsilon} + |\mathbf{C}|^{1+\varepsilon})$ on a common CRCW PRAM.*

Proof. We assume in the following that the input array $\mathbf{T}$ is compact. If *not*, a task description $(0, ())$ asking for zero processors can be written to the empty cells. These placeholders can also easily be removed from the computed task schedule.

The algorithm first determines, for each task, a "lead processor", which will be the processor with the lowest processor number assigned to the task.

For this purpose, it computes consistent λ-approximate prefix sums $s_1, \ldots, s_n$ for the sequence $m_1, \ldots, m_n$ using Proposition 3.1.8. It then assigns the first task d_1 to processor 1, and, for $i \in [2, n]$, task d_i to processor $s_{i-1} + 1$. Consequently, the algorithm allocates an array $\mathbf{C}$ of size s_n for the task schedule and sets $\mathbf{C}[0] = d_1$, and $\mathbf{C}[s_{i-1}+1] = d_i$, for $i \geq 2$. All remaining cells are initially empty. Since $s_n \leq (1+\lambda)\sum_{i=1}^{n} m_i$, the array $\mathbf{C}$ has the desired size. Thanks to Proposition 3.1.8 computing the prefix sums requires work and space $\mathcal{O}(|\mathbf{T}|^{1+\varepsilon})$.

To assign the remaining $m_i - 1$ processors to task d_i, we observe that, for each i, there are at least $m_i - 1$ empty cells in $\mathbf{C}$ between the cell containing d_i and the cell containing d_{i+1}, because the prefix sums $s_1, \ldots, s_n$ are consistent. Thus, it suffices to compute predecessor pointers for $\mathbf{C}$ and then copy, with $|\mathbf{C}|$ processors in parallel, the task description d_i to a cell $\mathbf{C}[j]$ if $\mathbf{C}[j]$ is empty and d_i is stored in the non-empty cell $\mathbf{C}[k]$ with maximal k that precedes $\mathbf{C}[j]$, i.e. cell the predecessor pointer of $\mathbf{C}[j]$ points to. Due to Proposition 3.3.4 this requires work and space $\mathcal{O}(|\mathbf{C}|^{1+\varepsilon})$. □

We are now ready to present algorithms for the `Join` operation in the dictionary and the ordered setting.

Proposition 3.4.9. *For every $\varepsilon > 0$ and every $\lambda > 0$, there are $\mathcal{O}(1)$-time parallel algorithms for* `Join` *that, given arrays $\mathbf{R}$ and $\mathbf{S}$, have the following bounds on an arbitrary CRCW PRAM. Here, $\mathcal{X}$ denotes the common attributes of R and S.*

(a) *Work and space $\mathcal{O}((|\mathbf{S}| + |D|)^{1+\varepsilon} + |\mathbf{R}|^{1+\varepsilon} + |\mathbf{R} \bowtie \mathbf{S}|^{1+\varepsilon})$ in the dictionary setting.*

(b) *Work and space $\mathcal{O}(|\mathbf{S}|^{1+\varepsilon} + |\mathbf{R}|^{1+\varepsilon} + |R \bowtie S|^{1+\varepsilon})$ if $\mathbf{S}$ is $\mathcal{X}$-ordered.*

The length of the output array is bounded by $(1+\lambda)|R \bowtie S|$.

Proof. For Statement (a) the algorithm first sorts $\mathbf{S}$ w.r.t. $\mathcal{X}$. Due to Lemma 3.3.9, this takes work and space $\mathcal{O}((|\mathbf{S}| + |D|)^{1+\varepsilon})$. The remainder of the algorithm is the same as for Statement (b), which we describe next. We set $\delta = \min\{\frac{1}{3}, \frac{\varepsilon}{3}\}$ and $\lambda' = \min\{\frac{1}{3}, \frac{\lambda}{3}\}$.

The algorithm proceeds in three phases: The grouping phase, the pairing phase, and the join phase.

Towards the grouping phase, we observe that, since $\mathbf{S}$ is $\mathcal{X}$-ordered, the tuples in $\mathbf{S}$ are, in particular, grouped by $\mathcal{X}$. In the grouping phase, the algorithm determines, the boundaries of these groups within $\mathbf{S}$, and λ-compacts each group.

For this purpose, it first establishes predecessor and successor links for $\mathbf{S}$, and then searches for each inhabited cell $\mathbf{S}[k]$ in parallel, the smallest index i and largest index j with $i \leq k \leq j$ such that $\mathbf{S}[i].\mathtt{t}[\mathcal{X}] = \mathbf{S}[k].\mathtt{t}[\mathcal{X}] = \mathbf{S}[j].\mathtt{t}[\mathcal{X}]$ holds, and augments $\mathbf{S}[k]$ by respective pointers. Thanks to Propositions 3.3.4 and 3.3.5 this requires overall work and space $\mathcal{O}(|\mathbf{S}|^{1+\varepsilon})$.

For each tuple $\tilde{a} \in \pi_{\mathcal{X}}(S)$, let $\mathbf{G}_{\tilde{a}}$ denote the subarray of $\mathbf{S}$ that contains the group for $\tilde{a}$. Because there are up to $|\mathbf{S}|$ many groups, the algorithm uses Lemma 3.4.8 to allocate processors for compacting the groups. To obtain proper task descriptions, the

algorithm computes an array $\mathbf{T}$ representing $\pi_{\mathcal{X}}(S)$ concisely using Proposition 3.4.4. Thanks to $\mathbf{S}$ being $\mathcal{X}$-ordered this can be done with work and space $\mathcal{O}(|\mathbf{S}|^{1+\varepsilon})$.

Note that each proper tuple $\tilde{a}$ in $\mathbf{T}$ is linked to $\mathbf{G}_{\tilde{a}}$ – via pointers from $\mathbf{T}$ to $\mathbf{S}$, and from each inhabited cell of $\mathbf{S}$ to the group boundaries – and a single processor can determine the length of $\mathbf{G}_{\tilde{a}}$ in constant time. The task description for a proper tuple $\tilde{a}$ consists of $m_{\tilde{a}} = |\mathbf{G}_{\tilde{a}}|^{1+\delta}$ and pointers to $\mathbf{G}_{\tilde{a}}$. The task schedule guaranteed by Lemma 3.4.8 has length at most

$$(1+\lambda)\sum_{\tilde{a}\in\pi_{\mathcal{X}}(S)} m_{\tilde{a}} = (1+\lambda)\sum_{\tilde{a}\in\pi_{\mathcal{X}}(S)} |\mathbf{G}_{\tilde{a}}|^{1+\delta} \leq (1+\lambda)\left(\sum_{\tilde{a}\in\pi_{\mathcal{X}}(S)} |\mathbf{G}_{\tilde{a}}|\right)^{1+\delta} \in \mathcal{O}(|\mathbf{S}|^{1+\delta})$$

and computing it, thus, requires work and space $\mathcal{O}(|\mathbf{S}|^{1+\varepsilon})$ because $(1+\delta)^2 \leq (1+\varepsilon)$.

Since $m_{\tilde{a}} = |\mathbf{G}_{\tilde{a}}|^{1+\delta}$ processors suffice to λ'-compact the group array $\mathbf{G}_{\tilde{a}}$ due to Lemma 3.3.8, all groups can be λ'-compacted using work and space $\mathcal{O}(|\mathbf{S}|^{1+\varepsilon})$ in total. This concludes the grouping phase.

In the pairing phase, the algorithm computes an array $\mathbf{R}'$ representing the relation $R' = R \ltimes S$ concisely by invoking $\texttt{SemiJoin}(\mathbf{R}, \mathbf{S})$. This has two effects. First, the proper tuples in $\mathbf{R}'$ are exactly those tuples from R that have at least one matching tuple in S. Second, each proper tuple $\tilde{a}$ in $\mathbf{R}'$ is (indirectly) linked to the λ'-compact group array for $\tilde{a}[\mathcal{X}]$. Since $\mathbf{S}$ is $\mathcal{X}$-ordered and fully linked this phase can be done with work $\mathcal{O}(|\mathbf{R}| \cdot |\mathbf{S}|^{\varepsilon})$ and space $\mathcal{O}(|\mathbf{R}| \cdot |\mathbf{S}|^{\varepsilon} + |\mathbf{S}|)$, thanks to Proposition 3.4.5. Note that the work and space bounds of this phase are dominated by $\mathcal{O}(|\mathbf{S}|^{1+\varepsilon} + |\mathbf{R}|^{1+\varepsilon})$.

In the join phase, the algorithm uses $|\mathbf{G}_{\tilde{a}[\mathcal{X}]}|$ processors for each proper tuple $\tilde{a}$ in $\mathbf{R}'$ to write $\tilde{a} \bowtie \tilde{a}'$, for all proper tuples $\tilde{a}'$ from $\mathbf{G}_{\tilde{a}}$, into the output array. Since there may be up to $|\mathbf{S}|$ groups of pairwise different length, we again employ Lemma 3.4.8 to assign processors.

The task description for a proper tuple $\tilde{a}$ from $\mathbf{R}'$ consists of the length $|\mathbf{G}_{\tilde{a}[\mathcal{X}]}|$ of its λ'-compact group array, a pointer to its cell in $\mathbf{R}'$, and pointers to its group $\mathbf{G}_{\tilde{a}[\mathcal{X}]}$. The length of the task schedule guaranteed by Lemma 3.4.8 can be bounded by $M = (1+\lambda')\sum_{\tilde{a}\in R\ltimes S} |\mathbf{G}_{\tilde{a}[\mathcal{X}]}|$.

The algorithm then uses M processors to assemble the output array of length M: If processor j is the i-th processor assigned to a proper tuple $\tilde{a}$ in $\mathbf{R}'$, it writes $\tilde{a} \bowtie \mathbf{G}_{\tilde{a}[\mathcal{X}]}[i].\mathtt{t}$ into the j-th cell of the output array, if $\mathbf{G}_{\tilde{a}[\mathcal{X}]}[i]$ is inhabited.

It remains to show that the output array has the desired size and the join phase can be carried out within the desired bounds. Let $n_{\tilde{a}}$ be the number of proper tuples in $\mathbf{G}_{\tilde{a}}$, for each $\tilde{a} \in \pi_{\mathcal{X}}(S)$, and let $R' = R \ltimes S$ denote the relation represented by $\mathbf{R}'$. Thanks to the group arrays being λ'-compact, we can deduce that

$$\begin{aligned} M = (1+\lambda')\sum_{\tilde{a}\in R'} |\mathbf{G}_{\tilde{a}[\mathcal{X}]}| \leq (1+\lambda')\sum_{\tilde{a}\in R'} (1+\lambda')n_{\tilde{a}[\mathcal{X}]} \\ = (1+\lambda')^2 \sum_{\tilde{a}\in R'} n_{\tilde{a}[\mathcal{X}]} = (1+\lambda')^2 |R \bowtie S| \end{aligned}$$

where the last equality holds because R' contains exactly those tuples $\tilde{a}$ from R that match with all tuples proper tuples in $\mathbf{G}_{\tilde{a}[\mathcal{X}]}$. Since further $|\mathbf{R}'| \in \mathcal{O}(|\mathbf{R}|)$, the task

schedule for this phase can be computed with work and space $\mathcal{O}(|\mathbf{R}|^{1+\varepsilon} + |R \ltimes S|^{1+\varepsilon})$. The output array can then be assembled with work and space $\mathcal{O}(|R \ltimes S|)$.

Lastly, the output array has the desired length of $(1+\lambda)|R \ltimes S|$ because we have $(1+\lambda')^2 \leq (1+\lambda)$ thanks to our choice of λ'. □

Since ordered arrays are *not* available in the general setting, the "high-level" algorithm for Proposition 3.4.9 does *not* apply to the general setting as is. Indeed, for the grouping phase, the algorithm relies on $\mathbf{S}$ being $\mathcal{X}$-ordered. Recall that an algorithm for the `Join` operation in the general setting requires at least quadratic work due to Lemma 3.4.2. Since our translation into the dictionary setting can also be done with quadratic work, we refrain from presenting a dedicated algorithm for the `Join` operation in the general setting, and instead refer to the translation which we will present in Section 3.6.

3.5 Query Evaluation in the Dictionary Setting

After studying algorithms for basic array operations and operators of the relational algebra, we are now prepared to investigate the complexity of $\mathcal{O}(1)$-time parallel algorithms for query evaluation. In this section we will focus on query evaluation in the dictionary setting – the results are summarized in the right-hand column of Table 3.1. For the other two settings, we will derive algorithms and bounds by means of a translation into the dictionary setting in Section 3.6.

Although every query of the relational algebra can be evaluated by a $\mathcal{O}(1)$-time parallel algorithms with polynomial work, the polynomials can be arbitrarily bad. In fact, that a graph has a k-clique can be expressed by a conjunctive query with k variables. Namely, the Boolean conjunctive query Q_k with $\mathsf{body}(Q_k) = \{E(x_i, x_j) \mid 1 \leq i < j \leq k\}$. Rossman [Ros08, Theorem 1.2] proved a $\omega(n^{k/4})$ lower bound on the size of constant-depth, unbounded fan-in circuit families for the k-clique decision problem. This implies that any $\mathcal{O}(1)$-time parallel algorithm that evaluates Q_k requires work $\omega(n^{k/4})$.

With this in mind, we rather study $\mathcal{O}(1)$-time parallel algorithms for fragments of query languages, for which efficient sequential algorithms are known. Concretely, we consider queries of the semi-join algebra (Section 3.5.1) and various subclasses of conjunctive queries, notably acyclic conjunctive queries (Section 3.5.2). Furthermore, we present algorithms for conjunctive queries with work (and space) bounds depending on their generalized hypertree width – effectively yielding evaluation algorithms for all conjunctive queries. Lastly, in Section 3.5.3, we present a $\mathcal{O}(1)$-time parallel version of worst-case optimal algorithms for natural join queries.

Conventions and Assumptions. For convenience and notational simplicity we use the following notation and conventions for the input arrays representing the database relations. By IN we always denote the maximum number of tuples in any relation of the underlying database that is addressed by the given query. Note that we have $|D| \in \mathcal{O}(\mathsf{IN})$ for any database D. Similarly, OUT denotes the number of tuples in the query result. We assume that, for some constant λ, every relation R of the input database is given by a λ-compact

array $\mathbf{R}$ that represents R concisely. In particular, we have $|\mathbf{R}| \in \mathcal{O}(\mathsf{IN})$ for all database relations.

3.5.1 Evaluation of Semi-Join Algebra Queries

Recall that the semi-join algebra is the fragment of the relational algebra that uses only selection, projection, rename, union, set difference and, not least, semi-join. It is well-known that query results of semi-join algebra queries have size $\mathcal{O}(\mathsf{IN})$ [cf. LV07, Corollary 16]. This is also reflected by our results for the dictionary setting in Section 3.4: The output arrays have length $\mathcal{O}(\mathsf{IN})$, if the input array has length $\mathcal{O}(\mathsf{IN})$. Since they further do *not* require the input arrays to be λ-compact, we can conclude the following.

Theorem 3.5.1. *For every $\varepsilon > 0$, $\lambda > 0$, and each query Q of the semi-join algebra there is a $\mathcal{O}(1)$-time parallel algorithm that, given λ-compact arrays representing the database relations concisely, evaluates Q, and requires work $\mathcal{O}(\mathsf{IN})$ and space $\mathcal{O}(\mathsf{IN}^{1+\varepsilon})$ on an arbitrary CRCW PRAM in the dictionary setting. The output array has length $\mathcal{O}(\mathsf{IN})$.*

Note that this result does *not* contradict our lower bounds stated in Theorem 3.4.1, since it does *not* make any assertions on the compactness of the output array. Indeed, compacting the output array using $\texttt{Compact}_\lambda$ yields the expected work bound $\mathcal{O}(\mathsf{IN}^{1+\varepsilon})$.

Leinders et al. [Lei+05, Theorem 19] proved that semi-join algebra queries can be evaluated in time $\mathcal{O}(\mathsf{IN})$. We emphasize that the work bound stated by Theorem 3.5.1 matches this time bound. Moreover, Theorem 3.5.1 states, in particular, that there are work-optimal $\mathcal{O}(1)$-time parallel algorithms for evaluating semi-join algebra queries in the dictionary setting.

3.5.2 Evaluation of Conjunctive Queries

In this section we present $\mathcal{O}(1)$-time parallel algorithms for evaluating (subclasses of) conjunctive queries. Since conjunctive queries are rule-based queries defined from the unnamed perspective, but our databases operations from Section 3.4 are defined in terms of the named perspective, we briefly discuss how they fit together.

First, we assume, in this section, that there are *no* variable repetitions in any atom of the query, that is the variables $x_1, \ldots, x_k$ of an atom $R(x_1, \ldots, x_k)$ are pairwise distinct. Indeed, this requisite can always be established by proper applications of `Select` operations to the arrays representing the database relations, which only requires linear work and space, thanks to Proposition 3.4.3. We further associate each variable x_i with a distinguished attribute X_i. A *full* conjunctive query Q is then equivalent to joins $R_1 \bowtie R_2 \bowtie \cdots \bowtie R_m$ of relations $R_1, \ldots, R_m$ where each relation R_i is obtained by renaming attributes according to an atom $R(x_1, \ldots, x_k) \in \mathsf{body}(Q)$ of the database relation R. In case the conjunctive query is *not* full, an additional projection to the attributes that correspond to the head variables of the query is applied. By $\mathsf{free}(Q)$ we denote the *set of attributes* associated to the head variables of the conjunctive query Q.

We proceed as follows. First we represent $\mathcal{O}(1)$-time parallel algorithms for evaluating *acyclic* conjunctive queries. Then we will show that, as in the sequential setting, the evaluation of conjunctive queries with bounded generalized hypertree width can be reduced to evaluating acyclic conjunctive queries, over a likely larger database. Since every conjunctive query has *some* generalized hypertree width, this effectively yields $\mathcal{O}(1)$-time parallel algorithms for evaluating any conjunctive query. The work and space bounds depend on the generalized hypertree width. Lastly, we show that better bounds can be achieved for conjunctive queries with small free-connex generalized hypertree width.

Acyclic Conjunctive Queries. We implement the well-known *Yannakakis algorithm* [Yan81] using our database operations from Section 3.4. The algorithm evaluates an acyclic conjunctive query Q along a join tree T_Q for Q. For that purpose, it associates, with each node $v \in \mathsf{nodes}(T_Q)$, a relation S_v. Initially, $S_v = D(R_v)$, where R_v is the relation symbol of v.[13] The algorithm then proceeds in three phases.

(1) bottom-up semi-join reduction: All nodes of T_Q are visited in bottom-up traversal order. Upon visiting a node v, S_v is iteratively updated by setting $S_v = S_v \ltimes S_w$, for every child w of v in T_Q.

(2) top-down semi-join reduction: All nodes of T_Q are visited in top-down traversal order. Upon visiting a node v, the relation S_v is updated by setting $S_w = S_w \ltimes S_v$, for every child w of v in T_Q.

(3) All nodes of T_Q are visited in bottom-up traversal order. Upon visiting a node v, the relation S_v is iteratively updated by setting $S_v = \pi_{\mathsf{free}(Q) \cup \mathtt{attr}(R_v)}(S_v \bowtie S_w)$, for every child w of v in T_Q.

After Phase (3), the query result can be obtained by evaluating $\pi_{\mathsf{free}(Q)}(S_v)$ for the root node $v = \mathsf{root}(T_Q)$. Bernstein and Goodman [BG81, Sections 2.4 and 3] proved the following.

Proposition 3.5.2 [BG81, Implied by Theorem 1]**.** *Let Q be an acyclic conjunctive query and T_Q be a join tree for Q. After Phase (2) of the Yannakakis algorithm, $|\pi_{\mathsf{free}(Q)}(S_v)| \leq \mathit{OUT}$ holds for every node v of T_Q.*

Proposition 3.5.2 implies the following invariants.

Corollary 3.5.3. *Let Q be an acyclic conjunctive query and T_Q be a join tree for Q. During Phase (3) of the Yannakakis algorithm, the following invariants hold, for every node v of T_Q.*

(a) $|S_v| \leq \mathit{IN} \cdot \mathit{OUT}$

(b) $|S_v| \leq \mathit{OUT}$*, if Q is a full query.*

[13] Recall that the nodes of T_Q are the atoms from the body of Q.

Proof. Statement (b) follows immediately from Proposition 3.5.2, since, for full queries, we have $\mathtt{attr}(S_v) \subseteq \mathsf{free}(Q)$ and thus $\pi_{\mathsf{free}(Q)}(S_v) = S_v$, for every node v.

For Statement (a) we observe that initially, after Phase (2), $|S_v| \leq \mathsf{IN}$ holds, and that Proposition 3.5.2 implies $\mathsf{OUT} \geq 1$, if $|S_v| \geq 1$. Moreover, each tuple $\tilde{a}$ derived by an expression of the form $\pi_{\mathsf{free}(Q) \cup \mathtt{attr}(R_v)}(S_v \ltimes S_w)$ can be partitioned into subtuples $\tilde{a}[\mathtt{attr}(R_v)]$ and $\tilde{a}[\mathsf{free}(Q)]$. The former is a tuple of the input relation R_v and the latter is contained in $\pi_{\mathsf{free}(Q)}(S_v)$ after the update. Thus, there at most $\mathsf{IN} \cdot \mathsf{OUT}$ tuples in S_v after each update step. □

We are now prepared to implement our parallel version of the Yannakakis algorithms, with the help of our database operations from Section 3.4.

Theorem 3.5.4. *For every $\varepsilon > 0$, $\lambda > 0$, and each acyclic conjunctive query Q there are $\mathcal{O}(1)$-time parallel algorithms that, given λ-compact arrays representing the database relations concisely, evaluate Q, and have the following bounds on an arbitrary CRCW PRAM in the dictionary setting.*

(a) *Work and space $\mathcal{O}((IN + IN \cdot OUT)^{1+\varepsilon})$, and*

(b) *Work and space $\mathcal{O}((IN + OUT)^{1+\varepsilon})$, if Q is a full acyclic conjunctive query.*

Proof. The algorithms for Statements (a) and (b) are almost identical – both are a parallel version of the Yannakakis algorithm. We present the algorithm for Statement (a) first.

For each node v of a join tree T_Q, let $\mathbf{S}_v$ be the array representing the relation S_v associated with v concisely. These arrays are initialized with copies of the input arrays representing the database relations – with their attributes properly renamed.

Phases (1) and (2) of the Yannakakis algorithm can be done with work $\mathcal{O}(\mathsf{IN})$ and space $\mathcal{O}(\mathsf{IN}^{1+\varepsilon})$ thanks to Theorem 3.5.1.

In Phase (3), the algorithm maintains $|\mathbf{S}_v| \in \mathcal{O}(\mathsf{IN} \cdot \mathsf{OUT})$ for all nodes v of T_Q. Initially this holds, since the arrays have length $\mathcal{O}(\mathsf{IN})$ after the first two phases, thanks to Theorem 3.5.1.

We describe the procedure for an update step of a node v and one of its child nodes w. Let $\mathcal{Y} = (\mathsf{free}(Q) \cup \mathtt{attr}(R_v)) \cap \mathtt{attr}(S_w)$ be the set of all attributes of S_w that are also attributes of R_v or correspond to free variables. Observe that

$$S_v \ltimes \pi_{\mathcal{Y}}(S_w) = \pi_{\mathsf{free}(Q) \cup \mathtt{attr}(R_v)}(S_v \ltimes S_w),$$

since the attributes of S_v are guaranteed to be free variables or to originate from the database relation R_v. The algorithm uses this observation and first computes an array $\mathbf{S}'_w$ representing $S'_w = \pi_{\mathcal{Y}}(S_w)$ concisely using $\mathtt{Project}_{\mathcal{Y}}$. $\mathtt{Project}_{\mathcal{Y}}$ can be carried out with work $\mathcal{O}(\mathsf{IN} \cdot \mathsf{OUT})$ and space $\mathcal{O}(\mathsf{IN}^{1+\varepsilon} \cdot \mathsf{OUT})$ thanks to Proposition 3.4.4 and $|\mathbf{S}_w| \in \mathcal{O}(\mathsf{IN} \cdot \mathsf{OUT})$. Furthermore, the array $\mathbf{S}'_w$ has length $\mathcal{O}(\mathsf{IN} \cdot \mathsf{OUT})$.

It then remains to compute $S_v \ltimes S'_w$ to complete the update step. Thanks to Proposition 3.4.9 this requires work and space

$$\mathcal{O}((|\mathbf{S}'_w| + |D|)^{1+\varepsilon} + |\mathbf{S}_v|^{1+\varepsilon} + |\mathbf{S}_v \bowtie \mathbf{S}'_w|^{1+\varepsilon}).$$

Thanks to Statement (a) of Corollary 3.5.3 we have $|S_v \ltimes S'_w| \in \mathcal{O}(\mathsf{IN} \cdot \mathsf{OUT})$ because $S_v \ltimes S'_w$ is the new relation associated with v after the update step. Furthermore, the new array has length $\mathcal{O}(\mathsf{IN} \cdot \mathsf{OUT})$. Overall, the algorithm therefore requires work and space $\mathcal{O}((\mathsf{IN} + \mathsf{IN} \cdot \mathsf{OUT})^{1+\varepsilon})$.

For Statement (b) the algorithm is essentially the same. However, since Q is a full query, it is *not* necessary to compute arrays $\mathbf{S}'_w$ for the projections $\pi_{\mathcal{Y}}(S_v)$, since $\pi_{\mathcal{Y}}(S_v) = S_v$. Furthermore, the algorithm can maintain $|\mathbf{S}_v| \in \mathcal{O}(\mathsf{OUT})$ in Phase (3), thanks to Statement (b) of Corollary 3.5.3. Together with the bounds for Phases (1) and (2) this yields the claimed work and space bound $\mathcal{O}((\mathsf{IN} + \mathsf{OUT})^{1+\varepsilon})$.

For further details on the correctness we refer to the primordial paper of Yannakakis [Yan81]. □

Generalized Hypertree Width. In the sequential setting the evaluation of any conjunctive query can be reduced to the evaluation of an acyclic conjunctive query [GLS02, Section 4.2]. Overall, the time it takes for evaluating the query then depends on the generalized hypertree width. As it turns out, the reduction can also be computed on a PRAM in constant time, yielding the following result.

Theorem 3.5.5. *For every $\varepsilon > 0$, $\lambda > 0$, and each conjunctive query Q with generalized hypertree width k there is a $\mathcal{O}(1)$-time parallel algorithm that, given λ-compact arrays representing the database relations concisely, evaluates Q, and requires work and space $\mathcal{O}((IN^k + IN^k \cdot OUT)^{1+\varepsilon})$ on an arbitrary CRCW PRAM in the dictionary setting.*

Proof. Let T be a generalized hypertree decomposition witnessing that Q has generalized hypertree width k. Thanks to Lemma 2.4.4 we can assume that T is complete. Following the reduction in the sequential setting [GLS02, Section 4.2], the algorithm computes an acyclic conjunctive query Q' and arrays representing the relations of a database D' such that $Q'(D') = Q(D)$, where D denotes the input database. It then invokes the algorithm guaranteed by Statement (a) of Theorem 3.5.4 to compute the query result $Q'(D)$.

For every $v \in \mathsf{nodes}(T)$, we define a new relation R_v by means of the conjunctive query Q_v with $\mathsf{vars}(\mathsf{head}(Q_v)) = \mathsf{bag}_T(v)$ and $\mathsf{body}(Q_v) = \mathsf{cover}_T(v)$. The database D' consists of all these relations R_v. More precisely, the schema of D' is $\{R_v \mid v \in \mathsf{nodes}(T)\}$ and $D'(R_v) = Q_v(D)$, for all $v \in \mathsf{nodes}(T)$.

The query Q' is the conjunctive query with

$$\mathsf{head}(Q') = \mathsf{head}(Q) \text{ and } \mathsf{body}(Q') = \{\mathsf{head}(Q_v) \mid v \in \mathsf{nodes}(T)\}.$$

It is straightforward to verify that $Q(D) = Q'(D')$ holds by inlining the definitions of the queries Q_v and observing that every atom of Q occurs in one of the Q_v thanks to the decomposition being complete. Moreover, Q' is acyclic because the generalized hypertree decomposition T' for Q' obtained from T by setting $\mathsf{cover}_{T'}(v) = \{R_v\}$, for every node v, has width 1.

Arrays $\mathbf{R}_v$ representing the relations R_v concisely can be computed using at most $|\mathsf{cover}_T(v)|$ many `Join` operations, followed by a $\texttt{Project}_{\mathcal{Y}}$ operation, where $\mathcal{Y} = \mathsf{free}(Q_v)$. The arrays representing the database relations in $\mathsf{cover}_T(v)$ have length $\mathcal{O}(\mathsf{IN})$ and, thanks

to Proposition 3.4.9 and T having width k, the arrays representing the (intermediate) results have length $\mathcal{O}(\mathsf{IN}^k)$. The `Join` operations can thus be carried out with work and space $\mathcal{O}(\mathsf{IN}^{(1+\varepsilon)k})$. Afterwards the projection can be computed with work $\mathcal{O}(\mathsf{IN}^k)$ and space $\mathcal{O}(\mathsf{IN}^{k+\varepsilon})$ thanks to Proposition 3.4.4.

The overall statement then follows thanks to Theorem 3.5.4. □

Free-Connex Conjunctive Queries. It turns out that the bounds from Theorem 3.5.5 can be improved, if the queries have a small free-connex generalized hypertree width.

We first consider the case of free-connex acyclic conjunctive queries. Evaluation of such a query can be reduced to the evaluation of a *full* acyclic conjunctive query [BGS20, Implicit in the proof of Theorem 4.1]. Again, we show that this reduction can be done in constant time by a PRAM.

Proposition 3.5.6. *For every $\varepsilon > 0$, $\lambda > 0$, and each free-connex acyclic conjunctive query Q there is a $\mathcal{O}(1)$-time parallel algorithm that, given λ-compact arrays representing the database relations concisely, evaluates Q, and requires work and space $\mathcal{O}((\mathit{IN} + \mathit{OUT})^{1+\varepsilon})$ on an arbitrary CRCW PRAM in the dictionary setting.*

Proof. Thanks to Proposition 2.4.5 there is a free-connex generalized hypertree decomposition T of Q that has width 1. Further, let $U \subseteq \mathsf{nodes}(T)$ be the set with $\mathsf{vars}(\mathsf{head}(Q)) = \bigcup_{w \in U} \mathsf{bag}_T(w)$ that induces a connected subtree of T. Let w be an arbitrary node in U, and orient the nodes in T such that w becomes the root in T. We can assume that T is complete thanks to Lemma 2.4.4 and adding new nodes to T *not* affecting U.

Since T has width 1, for each node $v \in \mathsf{nodes}(T)$, the set $\mathsf{cover}_T(v)$ contains exactly one atom A_v which we call the *guard* of v. As in the proof for Theorem 3.5.5, let, for every node v of T, Q_v be the conjunctive query with $\mathsf{vars}(\mathsf{head}(Q_v)) = \mathsf{bag}_T(v)$ and body $\mathsf{cover}_T(v)$. Note that Q_v is a projection of the database relation for (the only) relation symbol in $\mathsf{cover}_T(v)$ to the variables in $\mathsf{bag}_T(v)$. We associate every node v with the relation R_v defined by Q_v; that is, $R_v = Q_v(D)$. An array representing R_v concisely can be computed using $\texttt{Project}_{\mathcal{X}}$ with linear work and space $\mathcal{O}(\mathsf{IN}^{1+\varepsilon})$ thanks to Proposition 3.4.4.

In a second phase, the algorithm performs a bottom-up semi-join reduction, similar to Phase (1) of the Yannakakis algorithm. Upon visiting a node v, R_v is updated to $R_v \ltimes R_w$, for every child w of v in T. This can be computed with linear work and space $\mathcal{O}(\mathsf{IN}^{1+\varepsilon})$, thanks to Proposition 3.4.5. The resulting arrays have length $\mathcal{O}(\mathsf{IN})$.

Let D' be the database that consists of the relations R_w with $w \in U$ after the second phase. Further, let Q' be the conjunctive query with

$$\mathsf{head}(Q') = \mathsf{head}(Q) \text{ and } \mathsf{body}(Q') = \{\mathsf{head}(Q_w) \mid w \in U\}.$$

Recall, that U induces a connected subtree of T with $\mathsf{vars}(\mathsf{head}(Q)) = \bigcup_{w \in U} \mathsf{bag}_T(w)$. Thus, the heads of the queries Q_w contain only head variables of Q, and Q' is therefore a full query. Consider the generalized hypertree decomposition T' for Q' that consists of the subtree of T induced by U, and for which $\mathsf{bag}_{T'}(w) = \mathsf{bag}_T(w)$, and $\mathsf{cover}_{T'}(w) =$

$\{\mathsf{head}(Q_w)\}$ hold, for all $w \in U$. This generalized hypertree decomposition has width 1. Thus, Q' is a full acyclic conjunctive query.

It is straightforward to verify that $Q(D) = Q'(D')$ holds. Thus, applying the algorithm for Statement (b) of Theorem 3.5.4 yields the desired outcome. For more details we refer to Claims 1 and 2 in the proof for the sequential version [BGS20, Theorem 4.1]. □

The evaluation of conjunctive queries with free-connex generalized hypertree width k can be reduced to the evaluation of free-connex acyclic conjunctive queries [BGS20, Lemma 4.4]. In fact, the reduction is the same as we utilized for Theorem 3.5.5: If the generalized hypertree decomposition T of the original query is free-connex, then so is the derived generalized hypertree decomposition T' of Q' with width 1, since the underlying tree decomposition does *not* change. Thanks to Proposition 2.4.5 the query Q' is thus free-connex acyclic, which allows us to draw the following conclusion.

Corollary 3.5.7. *For every $\varepsilon > 0$, $\lambda > 0$, and each conjunctive query Q with free-connex generalized hypertree width k there is a $\mathcal{O}(1)$-time parallel algorithm that, given λ-compact arrays representing the database relations concisely, evaluates Q, and requires work and space $\mathcal{O}((\mathsf{IN}^k + \mathsf{OUT})^{1+\varepsilon})$ on an arbitrary CRCW PRAM in the dictionary setting.*

3.5.3 Weakly Worst-Case Optimal Work for Natural Joins

This section is concerned with the evaluation of *natural join queries*[14]

$$Q = R_1 \bowtie \ldots \bowtie R_m$$

over some schema $\mathcal{S} = \{R_1, \ldots, R_m\}$ with attributes $\mathtt{attr}(Q) = \bigcup_{i=1}^{m} \mathtt{attr}(R_i)$. Atserias et al. [AGM13] proved that $|Q(D)| \leq \prod_{i=1}^{m} |R_i|^{x_i}$ holds for every database D and that this bound is tight for infinitely many databases. This result is also known as the *AGM bound*. Here $x_1, \ldots, x_m$ is a *fractional edge cover* of Q defined as a solution of the following linear program.

$$\text{minimize} \sum_{i=1}^{m} x_i \text{ subject to } \sum_{i : X \in \mathtt{attr}(R_i)} x_i \geq 1 \text{ for all } X \in \mathtt{attr}(q)$$
$$\text{and } x_i \geq 0 \text{ for all } 1 \leq i \leq m$$

In the sequential setting, there are evaluation algorithms for natural join queries which run in time $\mathcal{O}(\prod_{i=1}^{m} |R_i|^{x_i} + \mathsf{IN})$ [e.g., Ngo+18, Theorem 6.1]. These algorithms are called *worst-case optimal* [cf. Ngo+18, Definition 3.2].

We say that a natural join query Q has *weakly worst-case optimal* $\mathcal{O}(1)$-time parallel algorithms, if, for every $\varepsilon > 0$, there is a $\mathcal{O}(1)$-time parallel algorithm that evaluates Q with work $\mathcal{O}((\prod_{i=1}^{m} |R_i|^{x_i} + \mathsf{IN})^{1+\varepsilon})$. Our next result states that natural join queries indeed have weakly worst-case optimal $\mathcal{O}(1)$-time parallel algorithms in the dictionary setting.

[14] We note that natural join queries correspond to full conjunctive queries. However, for this subsection we decided to use the notation and terminology commonly used in the context of worst-case optimal algorithms [cf., e.g., AGM13; Are+21].

Theorem 3.5.8. *For every $\varepsilon > 0$, $\lambda > 0$, and each natural join query $Q = R_1 \bowtie \ldots \bowtie R_m$ there is a $\mathcal{O}(1)$-time parallel algorithm that, given λ-compact arrays representing $R_1, \ldots, R_m$ concisely, evaluates Q, and requires work and space $\mathcal{O}\big(\big(\prod_{i=1}^{m} |R_i|^{x_i} + \mathit{IN}\big)^{1+\varepsilon}\big)$ on an arbitrary CRCW PRAM in the dictionary setting. Here $x_1, \ldots, x_m$ constitute a fractional edge cover of Q.*

A $\mathcal{O}(1)$-time parallel algorithm can proceed, from a high-level perspective, similarly to the sequential *attribute elimination join algorithm* [see, e.g., Are+21, Algorithm 10]. The core of this algorithm is captured by the following result.

Proposition 3.5.9 [Are+21, Algorithm 10 and Proposition 26.1]**.** *For a natural join query $Q = R_1 \bowtie \ldots \bowtie R_m$ with attributes $\mathcal{X} = (X_1, \ldots, X_k)$ and each database D, let*

- $L_1 = \bigcap_{i:X_1 \in \mathtt{attr}(R_i)} \pi_{X_1}(R_i)$ *and,*
- *for each j with $1 < j \leq k$, L_j be the union of all relations*

$$V_{\tilde{a}} = \{\tilde{a}\} \times \bigcap\nolimits_{i:X_j \in \mathtt{attr}(R_i)} \pi_{X_j}(R_i \ltimes \{\tilde{a}\})$$

for each $\tilde{a} \in L_{j-1}$.

Then the relation L_k is the query result $Q(D)$.

We will also use the following inequalities in our complexity analysis.

Lemma 3.5.10 [Are+21, pp. 228-229]**.** *Let $Q = R_1 \bowtie \ldots \bowtie R_m$ be a natural join query with attributes $\mathcal{X} = (X_1, \ldots, X_k)$ and $x_1, \ldots, x_m$ be a fractional edge cover of Q. Furthermore, let $L_1, \ldots, L_k$ be defined as in Proposition 3.5.9. For every database D it holds that*

(a) $\min_{i:X_1 \in \mathtt{attr}(R_i)} |R_i| \leq \prod_{i=1}^{m} |R_i|^{x_i}$*, and*

(b) $\sum_{\tilde{a} \in L_{j-1}} \min_{i:X_j \in \mathtt{attr}(R_i)} |R_i \ltimes \{\tilde{a}\}| \leq \prod_{i=1}^{m} |R_i|^{x_i}$ *for all $j \in [2, k]$.*

Proof for Theorem 3.5.8. Let $\mathcal{X} = (X_1, \ldots, X_k)$ be a sequence of all attributes occurring in Q, in some arbitrary but fixed order. Furthermore, we assume without loss of generality that $\lambda < \frac{1}{2}$. In a nutshell, the algorithm computes iteratively, for increasing j from 1 to k arrays $\mathbf{L}_j$ representing the relations L_j defined in Proposition 3.5.9 concisely and outputs $\mathbf{L}_k$. The correctness is then implied by Proposition 3.5.9.

Further on, we discuss how the arrays $\mathbf{L}_j$ can be computed and that the algorithm can ensure that $|\mathbf{L}_j| \leq (1 + \lambda) \prod_{i=1}^{m} |R_i|^{x_i}$ holds for every j.

Let $\mathcal{X}_j = (X_1, \ldots, X_j)$, for each $j \in [1, k]$, be the prefix of $\mathcal{X}$ up to attribute X_j. Furthermore, let, for each $i \in [1, m]$ and $j \in [1, k]$, $\mathcal{Y}_{i,j}$ be the subsequence obtained from $\mathcal{X}_j$ by removing all attributes not in $\mathtt{attr}(R_i)$. Note that, for every i, $\mathtt{attr}(R_i) = \mathcal{Y}_{i,k}$ and, for every j, L_j is a relation over the set of attributes from $\mathcal{X}_j$.

Let, for each $i \in [1, m]$, $\mathbf{R}_i$ be the λ-compact array that represents the input relation R_i concisely. During an initialization phase – outlined in Algorithm 3.1 – the algorithm

Algorithm 3.1: Initialization phase of the algorithm for Theorem 3.5.8.

```
1  foreach i ∈ [1,m] do // sequential loop
2      P_{i,k} ← Sort_λ(R_i, Y_{i,k})                          // Use Lemma 3.3.9
3
4      for j = k downto 2 do // sequential loop
5        if X_j ∈ attr(R_i) then
6          P_{i,j-1} ← Project_{Y_{i,j-1}}(P_{i,j})           // Use Proposition 3.4.4
7        else // Y_{i,j-1} = Y_{i,j}
8          P_{i,j-1} ← P_{i,j}
9        endif
10     endfor
11 endfor
```

Algorithm 3.2: Computation of L_1; part of the algorithm for Theorem 3.5.8.

```
1  foreach i ∈ [1,m] with X_1 ∈ attr(R_i) do // sequential loop
2    if L_1 is uninitialized then
3      L_1 ← P_{i,1}
4    else
5      L_1 ← SemiJoin(L_1, P_{i,1})                            // Use Proposition 3.4.5
6    endif
7  endfor
8
9  L_1 ← Compact_λ(L_1)                                        // Use Lemma 3.3.8
```

computes $\mathcal{Y}_{i,j}$-ordered, fully linked arrays $\mathbf{P}_{i,j}$, for each $i \in [1,m]$ and each $j \in [1,k]$, which represent $\pi_{\mathcal{Y}_{i,j}}(R_i)$ concisely, respectively. This can be done, for each i, by sorting $\mathbf{R_i}$ according to $\mathcal{Y}_{i,k}$, establishing predecessor and successor pointers, and then applying $\texttt{Project}_{\mathcal{Y}_{i,j}}$ for up to k values of j. By doing this in decreasing order of j, each proper tuple in $\mathbf{P}_{i,j}$ can be linked to its projection in $\mathbf{P}_{i,j-1}$.[15]

This initial phase requires work and space $\mathcal{O}(\sum_{i=1}^{m} |\mathbf{R_i}|^{1+\varepsilon}) = \mathcal{O}(\mathsf{IN}^{1+\varepsilon})$, for each i, thanks to Lemma 3.3.9, Proposition 3.3.4, and Proposition 3.4.4. The arrays $\mathbf{P}_{i,j}$ have length $\mathcal{O}(\mathsf{IN})$.

The computation of L_1 is outlined in Algorithm 3.2. It suffices to perform semi-joins, since all $\pi_{X_1}(R_i)$ involved have the same attribute. This requires work $\mathcal{O}(\mathsf{IN}^{1+\varepsilon})$ and space $\mathcal{O}(\mathsf{IN})$ since the arrays $\mathbf{P}_{i,1}$ are fully linked and ordered. Furthermore, the output array $\mathbf{L}_1$ has length at most IN. Thus, compacting $\mathbf{L}_1$ with $\texttt{Compact}_\lambda$ yields an array representing L_1 of length at most $(1+\lambda)|L_1|$ and requires work and space $\mathcal{O}(\mathsf{IN}^{1+\varepsilon})$. Clearly, $|L_1| \leq \min_{1\leq i\leq m} |\pi_{X_1}(R_i)| \leq \min_{1\leq i\leq m} |R_i|$. Thanks to Lemma 3.5.10 we have that $\min_{1\leq i\leq m} |R_i| \leq \prod_{i=1}^{m} |R_i|^{x_i}$. Thus, the length of the compacted array $\mathbf{L}_1$ is bounded by $(1+\lambda)\prod_{i=1}^{m} |R_i|^{x_i}$.

[15] If X_j does not occur in $\mathcal{Y}_{i,j}$, then $\mathbf{P}_{i,j-1}$ is just $\mathbf{P}_{i,j}$ and no computation is necessary.

To compute $\mathbf{L}_j$, for $j > 1$, the algorithm operates in two phases: A grouping phase, and an intersection phase. In these phases only those input relations R_i with $X_j \in \mathtt{attr}(R_i)$ participate. Let $\mathcal{R}_j$ be the set of all $i \in [1, m]$ such that X_j occurs in $\mathcal{Y}_{i,j}$. Note that for these $i \in \mathcal{R}_j$ we have, in particular, that X_j occurs in $\mathcal{Y}_{i,j}$ but *not* in $\mathcal{Y}_{i,j-1}$.

Towards the grouping phase, observe that the tuples in the arrays $\mathbf{P}_{i,j}$ are grouped by $\mathcal{Y}_{i,j-1}$ since they are even $\mathcal{Y}_{i,j-1}$-ordered. Furthermore, note that the group in $\mathbf{P}_{i,j}$ for a tuple $\tilde{a} \in L_{j-1}$ is essentially a list of the values in $\pi_{X_j}(R_i \ltimes \{\tilde{a}\})$ annotated with $\tilde{a}[\mathcal{Y}_{i,j-1}]$. Similar to the grouping phase of our algorithm for the `Join` operation, each group in $\mathbf{P}_{i,j}$ is compacted and each inhabited cell of $\mathbf{L}_{j-1}$ is augmented by pointers to (the first and last cell of) the group for $\mathbf{L}_{j-1}.\mathtt{t}$ in $\mathbf{P}_{i,j}$ as follows.

(1) Determine the minimum and maximum indices of each group in $\mathbf{P}_{i,j}$ with $|\mathbf{P}_{i,j}|$ processors: Processor p checks whether $\mathbf{P}_{i,j}[p].\mathtt{t}$ is a proper tuple and differs from its predecessor (resp. successor) w.r.t. to attributes in $\mathcal{Y}_{i,j-1}$. The representative in $\mathbf{P}_{i,j-1}$ is augmented with these indices. Since $\mathbf{P}_{i,j}$ is $\mathcal{Y}_{i,j}$-ordered, the proper tuples between the minimum and maximum assigned to a proper tuple $\tilde{b}$ in $\mathbf{P}_{i,j-1}$ are then precisely the tuples $\tilde{a}$ with $\tilde{a}[\mathcal{Y}_{i,j-1}] = \tilde{b}$. Note that in the special case of $\mathcal{Y}_{i,j}$ consisting solely of X_j, there is exactly one group. Then the first cell of, e.g., $\mathbf{P}_{i,j-1}$ can be chosen as representative of this group.

For every proper tuple $\tilde{a}$ in $\mathbf{P}_{i,j-1}$ let $\mathbf{G}_{\tilde{a},i,j}$ denote the (sub)array of the group for $\tilde{a}$ in $\mathbf{P}_{i,j}$, for every $i \in \mathcal{R}_j$.

(2) Compact each group $\mathbf{G}_{\tilde{a},i,j}$ in parallel using $\mathtt{Compact}_{\lambda'}$ with $\lambda' = \min\{\frac{1}{3}, \frac{\lambda}{3}\}$ and parameter $\delta = \min\{\frac{1}{3}, \frac{\varepsilon}{3}\}$. The choice of λ' is required for the complexity bounds in the intersection phase. To this end, the algorithm creates, for each proper tuple $\tilde{a}$ in $\mathbf{P}_{i,j-1}$ a task description $d_{\tilde{a}}$ for $m_{\tilde{a}} = |\mathbf{G}_{\tilde{a},i,j}|^{1+\delta}$ processors and with pointers to $\mathbf{G}_{\tilde{a},i,j}$ and the tuple $\tilde{a}$ in $\mathbf{P}_{i,j-1}$ itself.

Invoking the algorithm guaranteed by Lemma 3.4.8 yields a task schedule of length

$$(1+\lambda)\sum_{\tilde{a}\in\mathbf{P}_{i,j-1}} m_{\tilde{a}} = (1+\lambda)\sum_{\tilde{a}\in\mathbf{P}_{i,j-1}} |\mathbf{G}_{\tilde{a},i,j}|^{1+\delta} \leq (1+\lambda)\Big(\sum_{\tilde{a}\in\mathbf{P}_{i,j-1}} |\mathbf{G}_{\tilde{a},i,j}|\Big)^{1+\delta} = (1+\lambda)|\mathbf{P}_{i,j}|^{1+\delta}.$$

Thus, and due to $(1+\delta)^2 \leq (1+\varepsilon)$ computing the schedule requires work and space $\mathcal{O}(\mathsf{IN}^{1+\varepsilon})$. Moreover, the arrays $\mathbf{G}_{\tilde{a},i,j}$ can be compacted with $\mathtt{Compact}_{\lambda'}$, and the inhabited cells of $\mathbf{P}_{i,j}$ can be augmented by pointers to the λ'-compact arrays within the same bounds.

(3) Finally, the proper tuples in $\mathbf{L}_{j-1}$ are linked to their respective cell in $\mathbf{P}_{i,j-1}$, for every $i \in \mathcal{R}_j$ as follows. Since L_{j-1} is a relation over $\mathcal{X}_{j-1}$ which, in general, is a superset of $\mathcal{Y}_{i,j-1}$, the algorithm first computes a (possibly) non-concise representation of $\pi_{Y_{i,j-1}}(L_{j-1})$ with mutual pointers to/from $\mathbf{L}_{j-1}$. This can be done with $|\mathbf{L}_{j-1}|$ processors in a straightforward manner, and requires work and space $\mathcal{O}(|\mathbf{L}_{j-1}|)$.

Then, for each $i \in \mathcal{R}_j$ `SearchRepresentatives` is applied to link, for each proper tuple $\tilde{a}$ in $\mathbf{L}_{j-1}$, the respective tuple $\tilde{a}[\mathcal{Y}_{i,j-1}]$ in $\mathbf{P}_{i,j-1}$. Each of the (constantly many) applications of `SearchRepresentatives` requires work and space $\mathcal{O}(|\mathbf{L}_{j-1}| \cdot |\mathbf{P}_{i,j-1}|^{\varepsilon}) = \mathcal{O}((\prod_{i=1}^{m} |R_i|^{x_i}) \cdot \mathsf{IN}^{\varepsilon})$, because $\mathbf{P}_{i,j-1}$ is fully ordered (w.r.t. $\mathcal{Y}_{i,j-1}$) and fully linked. The pointers established by `SearchRepresentatives`, together with the pointers in $\mathbf{P}_{i,j-1}$ to the λ'-compact group arrays $\mathbf{G}_{\tilde{a},i,j}$ allow to determine, for each tuple $\tilde{a}$ in L_{j-1}, the smallest group array $\mathbf{G}_{\tilde{b},j}$ where $\tilde{b} = t[\mathcal{Y}_{i,j-1}]$ among the group arrays $\mathbf{G}_{\tilde{b},i,j}$ with work $\mathcal{O}(|\mathbf{L}_{j-1}|) = \mathcal{O}(\prod_{i=1}^{m} |R_i|^{x_i})$. Note that, in general, multiple tuples in $\mathbf{L}_{j-1}$ may be linked to the same group.

To ease notation, we will write $\mathbf{G}_{\tilde{a},i,j}$ and $\mathbf{G}_{\tilde{a},j}$ for the groups $\mathbf{G}_{\tilde{b},i,j}$ and $\mathbf{G}_{\tilde{b},j}$, respectively, for tuples $\tilde{a} \in L_{j-1}$ and where $\tilde{b} = \tilde{a}[\mathcal{Y}_{i,j}]$.

This phase requires work and space $\mathcal{O}((\prod_{i=1}^{m} |R_i|^{x_i} + \mathsf{IN}) \cdot \mathsf{IN}^{\varepsilon})$ in total.

Observe that a group array $\mathbf{G}_{\tilde{a},i,j}$ for some $\tilde{a} \in L_{j-1}$ represents $\pi_{Y_{i,j}}(R_i) \ltimes \{\tilde{a}\}$ concisely. Since $\tilde{a}$ determines the values for all attributes except X_j, $\mathbf{G}_{\tilde{a},i,j}$ can also be viewed as an array representing $\pi_{X_j}(R_i \ltimes \{\tilde{a}\})$ concisely, where every value is annotated with $\tilde{a}[\mathcal{Y}_{i,j-1}]$. In particular, a group $\mathbf{G}_{\tilde{a},j}$ represents the $\pi_{X_j}(R_i \ltimes \{\tilde{a}\})$ of minimal length among all $i \in \mathcal{R}_j$.

In the intersection phase, the algorithm computes the union of all $V_{\tilde{a}}$ in parallel. It proceeds in two steps. First, it will compute, for all $\tilde{a} \in L_{j-1}$, an array for the relation $\{\tilde{a}\} \times \pi_{X_j}(R_i \ltimes \{\tilde{a}\})$ for the i minimizing the size of $\pi_{X_j}(R_i \ltimes \{\tilde{a}\})$. In other words, each $\tilde{a}$ is joined with all proper tuples in $\mathbf{G}_{\tilde{a},j}$. Then, in the second step, it will perform semi-joins with all (remaining) group arrays $\mathbf{G}_{\tilde{a},i,j}$ to effectively compute the intersection of the sets $\{\tilde{a}\} \times \pi_{X_j}(R_i \ltimes \{\tilde{a}\})$ for all $i \in \mathcal{R}_j$.

For the first step, the algorithm has to assign $|\mathbf{G}_{\tilde{a},j}|$ processors to each tuple $\tilde{a} \in L_{j-1}$; recall that $\mathbf{G}_{\tilde{a},j}$ is the smallest group array for t.[16] To this end, it creates, for each $\tilde{a} \in L_{j-1}$, a task description $d_{\tilde{a}}$ for $m_{\tilde{a}} = |\mathbf{G}_{\tilde{a},j}|$ processors and, as usual, with pointers to the group array $\mathbf{G}_{\tilde{a},j}$ and the cell of $\tilde{a}$ in $\mathbf{L}_{j-1}$. Since $\mathbf{L}_{j-1}$ already contains pointers to the group arrays, these task descriptions can be computed with work and space linear in $|\mathbf{L}_{j-1}|$ and stored in an array of length $|\mathbf{L}_{j-1}|$. Thanks to Lemma 3.4.8 it is then possible to compute a schedule of length

$$M = (1 + \lambda') \sum_{\tilde{a} \in L_{j-1}} m_{\tilde{a}} \tag{1}$$

in constant time and with work and space $|\mathbf{L}_{j-1}|^{1+\varepsilon} + M^{1+\varepsilon}$.

To finalize the first step, the algorithm then allocates an array $\mathbf{L}_j$ of length M and uses M processors to write, for all $\tilde{a} \in L_{j-1}$, the join of $\{\tilde{a}\}$ with all tuples in $\mathbf{G}_{\tilde{a},j}$ into $\mathbf{L}_j$: If the m-th processor is the ℓ-th processor assigned to $\tilde{a}$, it writes $\{\tilde{a}\} \bowtie \mathbf{G}_{\tilde{a},j}[\ell].\mathtt{t}$ into $\mathbf{L}_j[m]$ if $\mathbf{G}_{\tilde{a},j}[\ell]$ is inhabited.

[16] We note that this corresponds (up to a logarithmic factor) to the running time stated by [Are+21, Claim 26.3] and required for the complexity analysis of [Are+21, Algorithm 10]; an implementation for the operation described in [Are+21, Claim 26.3] is, e.g., the Leapfrog-Join [Vel14], [Are+21, Proposition 27.10].

Algorithm 3.3: Second step of the intersection phase of the algorithm for Theorem 3.5.8.

```
1 foreach i ∈ [1, m] with X_j ∈ attr(R_i) do //sequential loop
2   L_j ← SemiJoin(L_j, P_{i,j})                    // Use Proposition 3.4.5
3 endfor
```

In the second step of the intersection phase, it remains to compute semi-joins with all remaining group arrays $\mathbf{G}_{\tilde{a},i,j}$ which are *not* minimal. Recall that every group array $\mathbf{G}_{\tilde{a},i,j}$ originates from $\mathbf{P}_{i,j}$. More precisely, the proper tuples in $\mathbf{G}_{\tilde{a},i,j}$ are exactly those tuples in $\mathbf{P}_{i,j}$ that match $\tilde{a}$. Thus, it suffices to compute semi-joins of $\mathbf{L}_j$ with $\mathbf{P}_{i,j}$ for all $i \in \mathcal{R}_j$ as outlined in Algorithm 3.3.[17] Because the arrays $\mathbf{P}_{i,j}$ are $\mathcal{Y}_{i,j}$-ordered and have length $\mathcal{O}(\mathsf{IN})$ the second step requires work $\mathcal{O}(M \cdot \mathsf{IN}^{\varepsilon})$ and space $\mathcal{O}(M + \mathsf{IN})$ thanks to Proposition 3.4.5.

To obtain the desired work and space bounds for the intersection phase as well as upper bounds for the length of $\mathbf{L}_j$, we prove that $M \leq (1+\lambda)\prod_{i=1}^{m} |R_i|^{x_i}$. As pointed out above, an array $\mathbf{G}_{\tilde{a},i,j}$ represents $\pi_{\mathcal{Y}_{i,j}}(R_i) \ltimes \{\tilde{a}\} = \pi_{\mathcal{Y}_{i,j}}(R_i \ltimes \{\tilde{a}\})$ concisely. Further, we have that $|\pi_{\mathcal{Y}_{i,j}}(R_i) \ltimes \{\tilde{a}\}| = |\pi_{\mathcal{Y}_{i,j}}(R_i \ltimes \{\tilde{a}\})| \leq |R_i \ltimes \{\tilde{a}\}|$. Since the groups where compacted using $\texttt{Compact}_{\lambda'}$ in the grouping phase, we can conclude that $(1+\lambda')|R_i \ltimes \{\tilde{a}\}|$ is an upper bound for $|\mathbf{G}_{\tilde{a},i,j}|$. In particular, for the group arrays $\mathbf{G}_{\tilde{a},j}$ of minimal length, we have

$$|\mathbf{G}_{\tilde{a},j}| \leq (1+\lambda') \min_{i \in \mathcal{R}_j} |R_i \ltimes \{\tilde{a}\}|. \tag{2}$$

Therefore, we have

$$\begin{aligned}
M &\overset{(1)}{=} (1+\lambda') \sum_{\tilde{a} \in L_{j-1}} m_{\tilde{a}} \\
&= (1+\lambda') \sum_{\tilde{a} \in L_{j-1}} |\mathbf{G}_{\tilde{a},j}| \\
&\overset{(2)}{\leq} (1+\lambda') \sum_{\tilde{a} \in L_{j-1}} (1+\lambda') \min_{i \in \mathcal{R}_j} |R_i \ltimes \{\tilde{a}\}| \\
&\overset{(3)}{\leq} (1+\lambda) \sum_{\tilde{a} \in L_{j-1}} \min_{i \in \mathcal{R}_j} |R_i \ltimes \{\tilde{a}\}| \\
&\overset{(4)}{\leq} (1+\lambda) \prod_{i=1}^{m} |R_i|^{x_i}
\end{aligned}$$

where Inequality (3) holds because $(1+\lambda')^2 \leq (1+\lambda)$; and Inequality (4) thanks to Lemma 3.5.10. Therefore, altogether $M = |\mathbf{L}_j| \leq (1+\lambda)\prod_{i=1}^{m} |R_i|^{x_i}$. This concludes the analysis of the arrays $\mathbf{L}_j$. □

[17] Note that this makes processor allocation straightforward.

3.6 Evaluation via Translation

The goal for this section is to obtain $\mathcal{O}(1)$-time parallel algorithms for evaluating queries in the general setting and in the ordered setting. For this purpose, we will first present translations from these settings into the dictionary setting in Section 3.6.1. In Section 3.6.2 we will then combine these translations with our evaluation algorithms from Section 3.5.

3.6.1 Into the Dictionary Setting

To translate a database D in the general or ordered setting into a representation for the dictionary setting, the domain values occurring in D have to be mapped to small numbers. More precisely, we are interested in an injective mapping $\mathsf{key}\colon \mathsf{adom}(D) \to [1, c_{\mathcal{S}}|D|]$ for some constant $c_{\mathcal{S}}$ that depends only on the fixed schema $\mathcal{S}$. Furthermore, we have to provide implementations of the operations $\texttt{KeyOf}_R(i,j)$ and $\texttt{KeyOutput}(k,i,j)$ such that a single processor can carry them out in constant time. For that purpose, our translations yield a *dictionary* – that is, a data structure which allows determining $\mathsf{key}(a)$ and $\mathsf{key}^{-1}(k)$ in constant time, given a token representation of a domain value a. Recall that token representations are *not* unique, i.e. there may be multiple token representations for a domain value a. All these representations have to be mapped to the same *key* $\mathsf{key}(a)$.

Recall that, using $\texttt{KeyOf}_R$ an array representing a relation R in the general or ordered setting, can be translated into an array representing $\mathsf{key}(R)$ in the dictionary setting in constant time with linear work. Thus, it suffices to compute a dictionary to effectively yield a translation. The following result states that dictionaries can indeed be computed in constant time.

Lemma 3.6.1. *For every $\varepsilon > 0$ and $\lambda > 0$, there are $\mathcal{O}(1)$-time parallel algorithms that compute a dictionary and have the following bounds on an arbitrary CRCW PRAM.*

- **(a)** *Work $\mathcal{O}(|D|^2)$ and space $\mathcal{O}(|D|)$ in the general setting.*
- **(b)** *Work $\mathcal{O}(|D|^{1+\varepsilon})$ and space $\mathcal{O}(|D|^{1+\varepsilon})$ in the ordered setting, given an $\{X\}$-ordered, λ-compact array representing R concisely, for every database relation R and every attribute $X \in \mathtt{attr}(R)$.*

Proof. For both settings, the idea is to compute an array $\mathbf{D}$ of length $\mathcal{O}(|D|)$, whose cells contain all (token representations of) domain values occurring in D. This array will be the underlying data structure of the dictionary. We note that $\mathbf{D}$ is (most likely) *not* concise. The key $\mathsf{key}(a)$ for a domain value a is then the index of a fixed representative of a in $\mathbf{D}$. In the following we will detail how to compile $\mathbf{D}$, and how the operations $\texttt{KeyOf}_R$, and $\texttt{KeyOutput}$ can be implemented on top of it.

We first present the algorithm for the general setting and then show how to adapt it for the ordered setting. For each input relation R and $j \in [1, \mathsf{ar}(R)]$ an array $\mathbf{R}_j$ of length $|R|$ is computed using $|R|$ processors as follows: Processor i writes the token (R, i, j) to cell $\mathbf{R}_j[i]$. Thus, $\mathbf{R}_j$ is a compact array that contains (token representations for) all domain values occurring for the j-th attribute in any tuple of R. Concatenating all arrays $\mathbf{R}_j$, where R ranges over all database relations and j over $[1, \mathsf{ar}(R)]$ yields the

array $\mathbf{D}$. Note that we can assume that $\mathbf{D}$ is compact, since there are only a constant number of input relations and a PRAM can determine the exact number of tuples in each database relation R using $\texttt{\#Tuples}_R$. The length of $\mathbf{D}$ is clearly bounded by $\sum_{R\in\mathcal{S}} \mathsf{ar}(R) \cdot |R| \leq c_{\mathcal{S}}|D|$ where $c_{\mathcal{S}} = \sum_{R\in\mathcal{S}} \mathsf{ar}(R)$. This step requires work and space $\mathcal{O}(|D|)$.

Next, the algorithm uses `SearchRepresentatives` to augment every cell $\mathbf{D}[\ell]$ with a pointer to a representative cell $\mathbf{D}[k]$ with $\mathbf{D}[k].\mathtt{t} = \mathbf{D}[\ell].\mathtt{t}$. The key for the domain value $a = \mathbf{D}[\ell].\mathtt{t}$ is then $\mathsf{key}(a) = k$, that is, the index of the representative cell. Thanks to Lemma 3.3.6 this requires work $\mathcal{O}(|D|^2)$ and space $\mathcal{O}(|D|)$ in the general setting.

It remains to show that $\texttt{KeyOf}_R$ and `KeyOutput` can be carried out in constant time by a single processor given $\mathbf{D}$. For $\texttt{KeyOf}_R(i,j)$ we observe that a single processor can determine the index ℓ of the token (R,i,j) in $\mathbf{D}$: Indeed, (R,i,j) has been stored in $\mathbf{R}_j[i]$, $\mathbf{D}$ is the (compact) concatenation of constantly many $\mathbf{S}_m$, and a single processor can obtain the size of any relation (and, hence, $|\mathbf{S}_m|$) using $\texttt{\#Tuples}_S$. Given ℓ it is then easy to determine the representative cell $\mathbf{D}[k]$ for $\mathbf{D}[\ell]$ using the pointers established by `SearchRepresentatives`. Of course, the special case $k = \ell$ can occur. The index k is then returned as the unique key for the domain value represented by (R,i,j).

The implementation of $\texttt{KeyOutput}(k,i,j)$ is straightforward. Indeed, k is an index of $\mathbf{D}$ and $\mathbf{D}[k]$ contains a token representation (R,i_1,j_1) for the domain value with the unique key k. Thus, invoking $\texttt{Output}_R(i_1,j_1,i,j)$ has the desired effect.

In the ordered setting an algorithm can essentially proceed similarly to the algorithm for the general setting detailed above. For the algorithm to require work $\mathcal{O}(|D|^{1+\varepsilon})$ it would suffice to use the algorithm for `SearchRepresentatives` for ordered arrays, since this is the only operation with a non-linear work bound. However, the array $\mathbf{D}$ is not necessarily ordered.

We will show how to construct another array $\mathbf{C}$ that, like $\mathbf{D}$, contains all (token representations of) domain values in D, and is *piecewise ordered*. It is then possible to search, for each cell of $\mathbf{D}$, a representative cell in $\mathbf{C}$. We note that the unordered version $\mathbf{D}$ is still required to carry out $\texttt{KeyOf}_R$ in constant time (by a single processor).

To construct $\mathbf{C}$, the algorithm first computes ordered arrays $\mathbf{R}'_j$ which correspond to the arrays $\mathbf{R}_j$ but are ordered and "only" λ-compact. Since, for every database relation R and every attribute X of R, a $\{X\}$-ordered array is given as input, $\mathbf{R}'_j$ can simply be derived from it with $\mathcal{O}|R|$ processors: Processor p reads the token representation for the j-th attribute X_j of R from the p-th cell of the $\{X_j\}$-ordered array and writes it into cell $\mathbf{R}'_j[p]$ if the cell is inhabited. Since the input array for R is $\{X\}$-ordered, $\mathbf{R}'_j$ is fully ordered. Furthermore, it has length $\mathcal{O}(\mathsf{IN})$. The array $\mathbf{C}$ is then the concatenation of all $\mathbf{R}'_j$. Computing $\mathbf{C}$ requires linear work and space.

To determine representatives, fix an arbitrary linear order on the set of all pairs (R,j) where R is a relation symbol from the schema $\mathcal{S}$ and $j \in [1, \mathsf{ar}(R)]$. For each pair (R,j) the algorithm invokes $\texttt{SearchRepresentatives}(\mathbf{D}, \mathbf{R}'_j)$ to find representatives for all domain values in the subarray $\mathbf{R}'_j$ of $\mathbf{C}$. Since there are only constantly many pairs (R,j) this can be done in constant time with work and space $\mathcal{O}(|D|^{1+\varepsilon})$ thanks to Lemma 3.3.6 and each $\mathbf{R}'_j$ being fully ordered. The representative for an inhabited cell $\mathbf{D}[\ell]$ is then

the representative cell for $\mathbf{D}[\ell]$ of the subarray $\mathbf{R}'_j$ for the smallest pair (R, j) for which such a representative cell exists in $\mathbf{R}'_j$.

The operations $\texttt{KeyOf}_R$ and $\texttt{KeyOutput}$ can then be implemented analogously to the general setting. That is, instead of returning an index of $\mathbf{D}$, $\texttt{KeyOf}_R(i, j)$ returns the index of the representative cell in $\mathbf{C}$ for the cell of $\mathbf{D}$ containing (R, i, j), and $\texttt{KeyOutput}(k, i, j)$ can use the token representation in $\mathbf{C}[k]$ to output the proper value. □

We note that the work bound for the general setting of Lemma 3.6.1 is optimal in the sense that a sub-quadratic work bound of the form $o(|D|^2)$ would contradict Lemma 3.4.2 because semi-joins can be evaluated with linear work in the dictionary setting thanks to Proposition 3.4.5.

3.6.2 Query Evaluation in the General and Ordered Setting

In this subsection we finally present algorithm (and bounds) for evaluating queries in the general setting and the ordered setting.

As for the dictionary setting, we start with algorithms for semi-join algebra queries.

Theorem 3.6.2. *For every $\varepsilon > 0$, $\lambda > 0$, and each query Q of the semi-join algebra there are $\mathcal{O}(1)$-time parallel algorithms that evaluate Q, and have the following bounds and requisites on an arbitrary CRCW PRAM.*

- **(a)** *Work $\mathcal{O}(IN^2)$ and space $\mathcal{O}(IN)$ in the general setting, given λ-compact arrays representing the database relations concisely.*
- **(b)** *Work and space $\mathcal{O}(IN^{1+\varepsilon})$ in the ordered setting, given an $\{X\}$-ordered, λ-compact array representing R concisely, for every database relation R and every attribute $X \in \mathit{attr}(R)$.*

Statement (a) of Theorem 3.6.2 actually follows from our results for the operators of the semi-join algebra in Section 3.4. We note that translating into the dictionary setting and then using the algorithm guaranteed by Theorem 3.5.1 would result in the same work bound, *but* a worse space bound. In any case, in light of Lemma 3.4.2 this work bound is the best we can hope for in the general setting.

Statement (b) of Theorem 3.6.2 is an immediate consequence of Theorem 3.5.1 and Lemma 3.6.1. It is tempting to also use our results from Section 3.4 to obtain algorithms to evaluate semi-join algebra queries in the ordered setting – that have less demanding requirements. However, the following example suggests that this is *not* straightforward, because our algorithm for $\texttt{Sort}_\lambda$ is only applicable in the dictionary setting. We discuss this further in Section 3.7.

Example 3.6.3. Consider the semi-join algebra query $Q = \pi_Y(\pi_{\{X,Y\}}(R))$ and the ternary relation

$$R = \{(a, f, b), (a, g, c), (b, d, d), (b, d, e), (d, c, e), (d, h, c), (e, a, h)\}.$$

Here the values for X are, as usual, given in the first component, and the values for Y and Z in the second and third, respectively. To evaluate the subquery $Q' = \pi_{\{X,Y\}}(R)$ with the

algorithm for the ordered setting guaranteed by Statement (b) of Proposition 3.4.4 we require a (X, Y)-ordered array representing R. The output array is then (X, Y)-ordered. But evaluating $Q = \pi_Y(Q')$ requires an $\{Y\}$-ordered array. And indeed, in our example, the output array for Q' is *not* $\{Y\}$-ordered – observe that the tuples of R are given in order w.r.t. (X, Y, Z) above. ◁

Onwards to the evaluation of conjunctive queries, we obtain the following results by combining Lemma 3.6.1 with Theorem 3.5.5, and Corollary 3.5.7, respectively. Note that these results also apply to acyclic and free-connex acyclic queries, respectively.

Corollary 3.6.4. *For every $\varepsilon > 0$, $\lambda > 0$, and each conjunctive query Q with generalized hypertree width k there is a $\mathcal{O}(1)$-time parallel algorithm that evaluates Q, and has the following bounds and requisites on an arbitrary CRCW PRAM.*

- **(a)** *Work $\mathcal{O}((\mathsf{IN}^k + \mathsf{IN}^k \cdot \mathsf{OUT})^{1+\varepsilon} + \mathsf{IN}^2)$ and space $\mathcal{O}((\mathsf{IN}^k + \mathsf{IN}^k \cdot \mathsf{OUT})^{1+\varepsilon})$ in the general setting, given λ-compact arrays representing the database relations concisely.*
- **(b)** *Work and space $\mathcal{O}((\mathsf{IN}^k + \mathsf{IN}^k \cdot \mathsf{OUT})^{1+\varepsilon})$ in the ordered setting, given an $\{X\}$-ordered, λ-compact array representing R concisely, for every database relation R and every attribute $X \in \mathtt{attr}(R)$.*

Corollary 3.6.5. *For every $\varepsilon > 0$, $\lambda > 0$, and each conjunctive query Q with free-connex generalized hypertree width k there is a $\mathcal{O}(1)$-time parallel algorithm that evaluates Q, and has the following bounds and requisites on an arbitrary CRCW PRAM.*

- **(a)** *Work $\mathcal{O}((\mathsf{IN}^k + \mathsf{OUT})^{1+\varepsilon} + \mathsf{IN}^2)$ and space $\mathcal{O}((\mathsf{IN}^k + \mathsf{OUT})^{1+\varepsilon})$ in the general setting, given λ-compact arrays representing the database relations concisely.*
- **(b)** *Work and space $\mathcal{O}((\mathsf{IN}^k + \mathsf{OUT})^{1+\varepsilon})$ in the ordered setting, given an $\{X\}$-ordered, λ-compact array representing R concisely, for every database relation R and every attribute $X \in \mathtt{attr}(R)$.*

We note that the additional addend for the general setting in Corollaries 3.6.4 and 3.6.5 only matters for $k = 1$, that is, acyclic and free-connex acyclic queries. For $k \geq 2$, it is swallowed by the other term.

Finally, we also obtain weakly worst-case optimal $\mathcal{O}(1)$-time parallel algorithm for the ordered setting by combining Theorem 3.5.8 and Lemma 3.6.1. In the general setting, there is an additional addend of $\mathcal{O}(\mathsf{IN}^2)$ due to the translation. In light of our lower bound stated in Lemma 3.4.2 this is, however, to be expected.

Corollary 3.6.6. *For every $\varepsilon > 0$, $\lambda > 0$, and each natural join query $Q = R_1 \bowtie \ldots \bowtie R_m$ there is a $\mathcal{O}(1)$-time parallel algorithm that evaluates Q, and has the following bounds and requisites on an arbitrary CRCW PRAM. Here $x_1, \ldots, x_m$ constitute a fractional edge cover of Q.*

- **(a)** *Work $\mathcal{O}((\prod_{i=1}^m |R_i|^{x_i} + \mathsf{IN})^{1+\varepsilon} + \mathsf{IN}^2)$ and space $\mathcal{O}((\prod_{i=1}^m |R_i|^{x_i} + \mathsf{IN})^{1+\varepsilon})$ in the general setting, given λ-compact arrays representing $R_1, \ldots, R_m$ concisely.*
- **(b)** *Work and space $\mathcal{O}((\prod_{i=1}^m |R_i|^{x_i} + \mathsf{IN})^{1+\varepsilon})$ in the ordered setting, given an $\{X\}$-ordered, λ-compact array representing R_i concisely, for each relation R_i and every attribute $X \in \mathtt{attr}(R_i)$.*

3.7 Discussion and Related Work

We conclude this chapter with a discussion of our results and related work.

Parallel Query Evaluation. As stated in the introduction the best known work bounds for evaluating queries of the relational algebra in constant time with PRAMs resulted – to the best of our knowledge – from translating queries into first-order formulas [Cod72] and then into PRAM algorithms [Imm89; Imm99]. The resulting algorithms require work $\mathcal{O}(\mathsf{IN}^k)$ where k is the number of variables that occur in the intermediate first-order formula. Our results show that better work bounds can be achieved for several queries, in particular, for classes that are known to allow for efficient query evaluation in the sequential setting.

For semi-join algebra queries we were able to obtain work-optimal algorithms (Theorem 3.5.1) in the dictionary setting, their work bound matches the running time $\mathcal{O}(\mathsf{IN})$ of the best sequential algorithm [Lei+05, Theorem 19]. However, this relies on intermediate results to be of size at most $\mathcal{O}(\mathsf{IN})$.

For queries that do *not* satisfy this requirement it seems crucial to keep the size of intermediate results in check. Using linear approximate compaction techniques for this purpose leads inevitably to work bounds of the shape $\mathcal{O}(m^{1+\varepsilon})$ for some $\varepsilon > 0$, due to the lower bounds discussed in Sections 3.1.2 and 3.4.1.

An alternative is to use *non-linear* approximate compaction. Hagerup [Hag92a, Unnamed Theorem] proved that an array of length n with k non-empty cells can be compacted into an array of length $k^{1+\varepsilon}$ in constant time with work $\mathcal{O}(n)$.[18] In fact, this technique is utilized in the conference paper [KSS23] this chapter is based on, instead of Proposition 3.1.6. Although this technique requires less work than linear compaction, processing an intermediate result of non-linear size nullifies this advantage.[19] More importantly, however, the (non-linear) compaction technique of Hagerup is *not* order-preserving. Of course, it can still be utilized to compact the final query result, if it is *not* already compact. In particular, this applies to the results of semi-join algebra queries: Unlike Proposition 3.1.6, non-linear compaction does *not* increase the required work in the dictionary setting, while compacting the result array at least somewhat.

Overall, our impression is that $\mathcal{O}(1)$-time parallel algorithms for query evaluation should be considered *work-efficient* for a query language, if they require work $\mathcal{O}(T^{1+\varepsilon})$, for every $\varepsilon > 0$, where T is the best sequential time of an evaluation algorithm. Of course, it would be nice if this impression could be substantiated by lower bound results which are independent of the representation of (intermediate) results. But obtaining such lower bounds seems to be quite challenging.

Let us mention two closely related articles on parallel query evaluation. Recently, Wang and Yi [WY22] studied query evaluation by circuits. In particular, they also presented a parallel version of the Yannakakis algorithm. However, the depth of the circuits is polylogarithmic, and hence they do *not* correspond to $\mathcal{O}(1)$-time parallel algorithms.

[18] We note that this corresponds to choosing $\lambda = n^{\varepsilon}$ in Corollary 3.1.5, and thus does *not* contradict the lower bound, since λ is *not* a constant.

[19] In our experience, it also results in more technical complexity analyses.

Denninghoff and Vianu [DV91] proposed *database method schemas* as model for studying parallel query evaluation. While they are defined for *object-oriented databases* they can also be applied to relational databases. However, although they consider constant-time parallel evaluation, they do not study the work of $\mathcal{O}(1)$-time parallel algorithms. Notably, they prove that PRAMs and database method schemas can simulate each other within a logarithmic time factor under some (reasonable) assumptions [DV91, Section 5].

Dynamic Complexity. Our study of work-efficient $\mathcal{O}(1)$-time parallel algorithms for evaluating queries was actually motivated by recent developments in the area of dynamic complexity theory within the *DynFO-framework* introduced by Patnaik and Immerman [PI97] and, similarly, by Dong and Su [DS95].

In this framework the database is subject to change. For instance, facts may be added or removed. It is then natural to ask whether a previously computed query result can be updated accordingly efficiently – and thus effectively be *maintained*. In this context efficiency usually refers to $\mathcal{O}(1)$-time parallel algorithms.[20] Since queries can even be reevaluated in constant time, the research focused on queries beyond the expressiveness of the relational algebra, for instance reachability queries [Dat+18]. For more details we refer to the survey of Schwentick et al. [SVZ20].

Recently, Schmidt et al. [Sch+21] started to investigate the work required by algorithms for maintaining query results in this framework [cf., also SS23; SST23]. In this case it then becomes meaningful to consider queries of the relational algebra.

To this end, the research presented in this chapter is meant to lay a foundation, by studying the work required for evaluation queries of the relational algebra in the *static*, i.e. "non-dynamic", setting.

Comparison-Based Sorting. Our algorithms for the ordered setting all require that at least one of the input array is suitably ordered. For the query evaluation algorithms in Section 3.6.2, it is even required that, for all relations R and all attributes X of R, an $\{X\}$-ordered array representing R is available.

These requirements could be lifted, if we had an implementation for $\texttt{Sort}_\lambda$ in the ordered setting. Since, in this setting, the domain values cannot be directly accessed but only compared using the operations $\texttt{Equal}_{R,S}$ and $\texttt{LessThan}_{R,S}$, such an implementation boils down to an algorithm for *comparison-based padded sorting*. To the best of our knowledge it is unknown whether there is such an algorithm for PRAMs. For details on that matter we refer to the discussion of Chong and Ramos [CR98, Section 1]. We point out, however, that comparison-based padded sorting in constant time is possible using a *randomized* PRAM with work and space $\mathcal{O}(n^{1+\varepsilon})$ [HR92, Corollary 3.5]. This matches a known lower bound [AV87, Corollary 1]. We therefore do *not* expect a $\mathcal{O}(1)$-time parallel algorithm for comparison-based padded sorting to yield better work and space bounds for query evaluation in the ordered setting.

[20] In the literature, algorithms in this setting are often represented by first-order formulas. But they can be translated into $\mathcal{O}(1)$-time parallel algorithms for PRAMs [Imm89; Imm99].

We emphasize that translating into the dictionary setting using the procedure for the general setting and then applying $\texttt{Sort}_\lambda$ does *not* yield a sorting algorithm for the original values. Instead, merely the keys assigned to the domain values by the translation are sorted.

Data Structures. As mentioned in Section 3.2 we chose ordered arrays for convenience. In the sequential setting, various kinds of *index structures* are utilized instead.

One example are *index arrays*. An index array $\mathbf{I}_{R,\mathcal{X}}$ for a relation R and a sequence $\mathcal{X}$ of attributes of R is an array whose cells contain each number from $[1, |R|]$ exactly once such that, for all indices i, j of $\mathbf{I}_{R,\mathcal{X}}$ with $i < j$, the $\mathbf{I}_{R,\mathcal{X}}[i]$-th tuple is smaller than the $\mathbf{I}_{R,\mathcal{X}}[j]$-th tuple of R w.r.t. the lexicographical order induced by $\mathcal{X}$, if both cells are *not* empty. Of course, instead of numbers in $[1, |R|]$ an index array may also refer to the indices of an array representing R. It is straightforward to derive an ordered array from an index array: The content of a cell $\mathbf{I}_{R,\mathcal{X}}[i]$ is simply replaced by the $\mathbf{I}_{R,\mathcal{X}}[j]$-th tuple of R.

In the dictionary setting they can also be computed work-efficiently in constant time. The algorithm simply invokes $\texttt{Sort}_\lambda$, and replaces in the output array each tuple with its index in the input array $\mathbf{R}$. Recall that the index for $\mathbf{R}$ can be obtained in constant time by a single processor thanks to the pointers guaranteed by $\texttt{Sort}_\lambda$.

It is also not hard to see that other operations like $\texttt{Compact}_\lambda$ can carry over index arrays for their input array(s) to the output array.

Another alternative are search trees. For instance, a *B-tree* with page size $\mathcal{O}(n^\varepsilon)$ allows for single tuple insertions and lookups in constant time with $\mathcal{O}(n^\varepsilon)$ work [BM72]. However, inserting multiple tuples in parallel will probably require some kind of scheduling.

Lastly, let us point out that the predecessor and successor pointers established by Proposition 3.3.4 effectively yield a doubly linked list. Thus, a query result could also be represented by such a list. In particular, as usually expected from sequential algorithms, it allows for (trivial) constant-delay enumeration [BGS20].

Chapter 4

Distributed Evaluation of Datalog

In this chapter we study the parallel-correctness and the parallel-boundedness problem for Datalog queries in the MPC model. Recall that we will use the term *server* rather than processor, to emphasize the distributed nature of this setting.

Let us emphasize that, in difference to Chapter 3 this chapter is concerned with static analysis problems. In particular, it is no longer justifiable that the database schema has a fixed arity.

Outline. We will proceed as follows. In Section 4.1 we will present preliminaries on distributed databases, formalize the setting, and present our MPC-based framework to reason about distributed evaluation of Datalog queries. This includes, in particular, the introduction of the concrete formalisms we employ to specify hash-based distribution and constraint-based communication policies.

For these kinds of policies, we will study the parallel-correctness and the parallel-boundedness problems in Section 4.2 and Section 4.4, respectively. In between, in Section 4.3, we revisit the decision procedure for the containment problem for frontier-guarded Datalog queries of Bourhis et al. [BKR15a]. In a sense Section 4.3 will conclude our study of parallel-correctness, because our upper bound results for parallel-correctness in Section 4.2 rely on a fine graded complexity analysis of this decision procedure. Moreover, we will build upon it in Section 4.4 to obtain our upper bound results for parallel-boundedness. Finally, we will discuss related work in Section 4.5.

Although it is technically not a part of this chapter, let us point out that we briefly present an alternative to constraint-based communication policies – namely hash-based communication policies – in Appendix B.

Publication and Contributions. This chapter is based on a conference paper [Nev+19] which I co-authored with my advisor, Prof. Dr Thomas Schwentick, as well as Prof. Dr Frank Neven and Dr Brecht Vandevoort from the Hasselt University, Belgium. In difference to this chapter, this paper also contains results on parallel-correctness with respect to hash-based *communication* policies and in the "locally restrained setting". Results for hash-based communication policies are presented in Appendix B, though, and the locally restrained setting is briefly discussed in Section 4.5.

My main contributions to this paper were the parallel-correctness results for hash-based distribution policies and constraint-based communication policies presented here in Sections 4.2.2 to 4.2.4, the complexity analysis of the containment problem (Section 4.3),

and the results on parallel-boundedness (Section 4.4). The undecidability result presented in Section 4.2.1, the simulation of non-transitive (Section 4.2.5), and locally restrained settings within our framework, and the results on hash-based policies (for communication *and* distribution) were mainly contributed by my co-authors. It should be noted that the proof ideas for our 2ExpTime-completeness results for parallel-correctness are all similar and the proof for hash-based communication policies, obtained by Dr Brecht Vandevoort, is the original. The contents of Section 4.1 were derived in an iterative process to which all have contributed equally.

The reason for presenting also (most of the) main contributions of my co-authors in this chapter (and in Appendix B) is twofold. Firstly, it results in a coherent representation. In some cases results of my co-authors are even strict requirements for my own (this is particularly true for results on hash-based distribution policies). The other motivation is the publication of full proofs for the results of the conference paper. To better distinguish the main contributions of my co-authors, they are marked with a star (e.g. Definition*, Theorem*, etc.) throughout this chapter.

I reworked the proofs of all results, based on (mostly) unpublished proof (sketches). Notably, for the results in Section 4.2.5, I opted for a slightly different definition of the semantics in the non-transitive setting (Definition 4.2.30) and more general constructions for the proofs of Theorem 4.2.31 and Theorem 4.2.33, which allow for a "smoother" integration into our framework. The 2ExpTime-hardness lower bound stated in Theorem 4.2.31, the main result of Section 4.2.2 (Proposition 4.2.16), and the 2ExpTime-*completeness* result for the parallel-boundedness problem in the non-transitive communication setting are new, unpublished results.

4.1 Setting and Framework

We adapt the *Massively Parallel Communication (MPC)* model [BKS17a] and define a generic framework that allows us to reason about parallel evaluation of Datalog queries in distributed settings. We start with some basic terminology, namely networks of servers and distributed databases. Then we will introduce the core of our framework: Policy pairs (δ, γ), which consist of a (partial) specification of the initial data distribution δ and a communication policy γ over the same network. In Section 4.1.1 we will then formalize the distributed evaluation of Datalog queries in our framework. Lastly, we will introduce our concrete formalisms to specify distribution policies and communication policies in Sections 4.1.2 and 4.1.3, respectively. For a comparison with the framework of Ketsman et al. [KAK20] we refer to Section 4.5.

Networks and Distributed Databases. We model a *network* of database servers as a finite, non-empty set $\mathcal{N}$ of *servers*. Following Geck et al. [Gec+16] we take a "local-as-view" approach for the representation of distributed databases and keep track of all facts via a "global" database, in addition to "local", per server databases. Formally, a *distributed database* $\mathcal{D} = (G, \mathcal{I})$ over a network $\mathcal{N}$ and a schema $\mathcal{S}$ consists of a database G

over $\mathcal{S}$ and a family $\mathcal{I} = (I_k)_{k \in \mathcal{N}}$ of databases over $\mathcal{S}$, one for each server of $\mathcal{N}$, such that

$$\bigcup_{k \in \mathcal{N}} I_k = G.$$

We call G the *global database* of $\mathcal{D}$; and the I_k are called *local databases*. We write $R(\bar{a})@k$ for a fact $R(\bar{a})$ in I_k. Particularly, $R(\bar{a})@k \in \mathcal{D}$ is synonymous with $R(\bar{a}) \in I_k$. For a fact $R(\bar{a})@k$ we also say that $R(\bar{a})$ *resides* on server k. We emphasize that we do *not* allow facts to be "skipped". That is, every fact in G should occur somewhere as a *local fact*, i.e. in at least one local database. In the literature this is not always required and referred to as a *complete* distributed database [cf., e.g., Gec19, Section 2.2.1]. For two distributed databases $\mathcal{D} = (G, \mathcal{I})$ and $\mathcal{D}' = (G', \mathcal{I}')$ over the same network $\mathcal{N}$ we write $\mathcal{D} \subseteq \mathcal{D}'$, if $G \subseteq G'$ and $I_k \subseteq I'_k$ hold, for every $k \in \mathcal{N}$.

Policies. As mentioned before, the core of our framework are *policy pairs* (δ, γ) consisting of a distribution policy δ and a communication policy γ over the same network.

Before we define these terms, let us emphasize that we require distribution policies to describe only the distribution of extensional facts and communication policies to be restricted to intensional facts.

In general, a *distribution policy* δ over a network $\mathcal{N}$ and a schema $\mathcal{S}$ is a function mapping databases G over $\mathcal{S}$ to distributed databases $\mathcal{D} = (G, \mathcal{I})$ over $\mathcal{N}$ and $\mathcal{S}$. A distributed database $\mathcal{D} = (G, \mathcal{I})$ over $\mathcal{N}$ and $\mathcal{S}$ *complies* with a distribution policy δ over $\mathcal{N}$ and $\mathcal{S}$ if $\delta(G) \subseteq \mathcal{D}$ holds. Thus, we allow facts to reside on more servers than required by δ. This is because we do not understand δ as a specification of a communication round but rather as a constraint that is met at the beginning of the evaluation of a Datalog query. However, by definition the global database of the distributed database $\delta(G)$ is precisely G. In other words, δ cannot postulate the existence of facts *not* in G. We define the specific (hash-based) distribution policies used in this thesis below in Section 4.1.2.

For a network $\mathcal{N}$, and a schema $\mathcal{S}$, a *communicated fact* $R(\bar{a})@k \triangleright \ell$ over $\mathcal{N}$ and $\mathcal{S}$ consists of an $\mathcal{S}$-fact $R(\bar{a})$ and two servers k and ℓ from $\mathcal{N}$. It has the intended meaning that $R(\bar{a})$ is communicated from k to ℓ. A *communication policy* γ over $\mathcal{N}$ and $\mathcal{S}$ is a monotone mapping that assigns, to every distributed database $\mathcal{D}$ over $\mathcal{N}$ and a schema $\mathcal{S}' \supseteq \mathcal{S}$, a set of communicated facts $R(\bar{a})@k \triangleright \ell$ over $\mathcal{N}$ and $\mathcal{S}$ with $R(\bar{a})@k \in \mathcal{D}$. Here, monotonicity means that $\gamma(\mathcal{D}) \subseteq \gamma(\mathcal{D}')$ holds whenever $\mathcal{D} \subseteq \mathcal{D}'$ does. The specific (constraint-based) communication policies studied in this thesis are defined in Section 4.1.3. Another formalism to specify communication policies in our framework, resulting in hash-based communication policies, is defined in Appendix B.

As in Chapter 3, we assume a fixed database schema $\mathcal{S}$ for the underlying database and queries throughout the remainder of this chapter. Moreover, we always require that the schemas of Datalog queries, distribution, and communication policies "match". That is, we consider a Datalog query $Q = (P, \mathrm{Out})$ over $\mathcal{S}$ only in combination with distribution policies over $\mathcal{S}$ – recall that $\mathsf{edb}(P) \subseteq \mathcal{S}$ – and communication policies over the schema $\mathsf{idb}(P)$.

4.1.1 Distributed Evaluation of Datalog Programs

A communication policy γ induces a distributed multi-round evaluation strategy for a Datalog query $Q = (P, \text{Out})$ over a distributed database $\mathcal{D} = (G, \mathcal{I})$ as follows.[1] Each *round* consists of a computation and a communication phase. By $\mathcal{D}^i = (G^i, \mathcal{I}^i)$ we denote the distributed database after the communication phase of the i-th round. The initial distributed database $\mathcal{D}^0$ is just $\mathcal{D}$. Then, for $i \geq 1$, the following phases are performed.

- *Computation phase*: Every server computes the *local (least) fixpoint* of P over its local database. That is, the intermediate result after this phase is $\mathcal{D}' = (G', \mathcal{J})$ where $G' = \bigcup_{k \in \mathcal{N}} J_k$ and, for each $k \in \mathcal{N}$, $J_k = P(I_k^{i-1})$.
- *Communication phase*: For each communicated fact $R(\bar{a})@k \triangleright \ell \in \gamma(\mathcal{D}')$, the fact $R(\bar{a})$ is copied from k to ℓ. That is, for each $\ell \in \mathcal{N}$,

$$I_\ell^i = J_\ell \cup \big\{R(\bar{a}) \mid R(\bar{a})@k \triangleright \ell \in \gamma(\mathcal{D}')\big\}.$$

  Then, $\mathcal{D}^i = (G^i, \mathcal{I}^i)$ where $G^i = \bigcup_{k \in \mathcal{N}} I_k^i$.

The distributed evaluation terminates when a *global fixpoint* is reached, i.e. $\mathcal{D}^r = \mathcal{D}^{r+1}$ holds for some $r \geq 0$. We denote this fixpoint by $[P, \gamma](\mathcal{D})$, and note that it always exists, because Datalog queries are monotone and communication policies only copy facts. That is, $[P, \gamma](\mathcal{D}) = \mathcal{D}^r$ where r is the smallest integer such that $\mathcal{D}^r = \mathcal{D}^{r+1}$ holds. The *parallel query result* $[Q, \gamma](\mathcal{D})$ *of* Q *over* $\mathcal{D}$ *according to* γ, is the union of all output facts occurring in $[P, \gamma](\mathcal{D})$. More precisely,

$$[Q, \gamma](\mathcal{D}) = \big\{\text{Out}(\bar{a}) \mid \text{Out}(\bar{a})@k \in [P, \gamma](\mathcal{D}) \text{ for some server } k\big\}.$$

We note that our setting differs slightly from the one of Ketsman et al. [KAK20]. We provide more details in Section 4.5.

Example 4.1.1. Consider the monadic Datalog query $Q = (P, \text{Out})$ and the database D from Example 2.4.6 which we briefly recall here for convenience. The query Q asks for all nodes reachable from a starting node by a path containing only red as well as a path containing only sea blue edges. Its Datalog program P consists of the following rules.

$$R(x) \leftarrow \text{Start}(x) \qquad S(x) \leftarrow \text{Start}(x) \qquad \text{Out}(x) \leftarrow R(x), S(x)$$
$$R(x) \leftarrow R(y), E_r(y, x) \qquad S(x) \leftarrow S(y), E_s(y, x)$$

The database D is defined as

$$D = \big\{\text{Start}(1), E_r(1,3), E_r(1,4), E_s(1,2), E_s(2,3)\big\}.$$

Let $\mathcal{D} = (D, \mathcal{I})$ be the distributed database over the network $\mathcal{N} = [1, 4]$ with the local databases

$$I_1 = \big\{\text{Start}(1), E_r(1,3), E_r(1,4)\big\}, I_2 = \big\{\text{Start}(1), E_s(1,2), E_s(2,3)\big\}, \text{ and } I_3 = I_4 = \emptyset.$$

[1] We recall that we do not view a distribution policy as a specification of a communication round.

That is, Start-facts are duplicated over servers 1 and 2, while red edges reside on server 1 and sea blue edges on server 2.

Let γ be the communication policy that maps a distributed database $\mathcal{D}' = (G', \mathcal{I}')$ to the set of communicated facts

$$\{R(a)@1 \triangleright f(a) \mid R(a) \in I'_1\} \cup \{S(a)@2 \triangleright f(a) \mid S(a) \in I'_2\}$$

where f is the function mapping each value a to $((a-1) \bmod 4) + 1$. That is, when a fact $R(a)$ can be derived on server 1, it is sent to server $f(a)$. A fact $S(a)$ derived on server 2 is also sent to server $f(a)$.

The distributed evaluation then proceeds as follows. Rounds are depicted from left to right and underlined facts are received through communication.

$$\begin{array}{lll}
I^1_1 = I_1 \cup \{R(1), R(3), R(4), \underline{S(1)}\}, & I^2_1 = I^1_1 \cup \{\mathrm{Out}(1)\}, & I^3_1 = I^2_1, \\
I^1_2 = I_2 \cup \{S(1), S(2), S(3)\}, & I^2_2 = I^1_2, & I^3_2 = I^2_2, \\
I^1_3 = I_3 \cup \{\underline{R(3)}, \underline{S(3)}\}, & I^2_3 = I^1_3 \cup \{\mathrm{Out}(3)\}, & I^3_3 = I^2_3, \\
I^1_4 = I_4 \cup \{\underline{R(4)}\} & I^2_4 = I^1_4 & I^3_4 = I^2_4
\end{array}$$

A global fixpoint is reached after the third round. The parallel query result is $[Q, \gamma](\mathcal{D}) = \{\mathrm{Out}(1), \mathrm{Out}(3)\}$. Thus, for this example, the distributed evaluation according to γ yields the same result as the classical evaluation of Q on D, i.e. we have $[Q, \gamma](\mathcal{D}) = Q(D)$, cf. Example 2.4.6. ◁

We next point out a crucial truth about (our) distributed evaluation strategies. Namely, that the parallel query results of two equivalent Datalog queries Q_1 and Q_2 are not necessarily the same. In other words, the notion of equivalence does not carry over from the sequential to our distributed setting. This affects, in particular, the relationship between the query classes MDL and FGDL of monadic and frontier-guarded Datalog queries, respectively. For every monadic Datalog query an equivalent frontier-guarded Datalog query can be obtained by adding suitable guard atoms to every rule, cf. Lemma 2.4.9. The next example illustrates that such a transformation can change the parallel query result. Therefore, we cannot simply rely on this transformation to deduce results for monadic Datalog queries from corresponding results for frontier-guarded Datalog queries (or vice versa, for lower bounds). In fact, we will see in Section 4.2.5 that the relationship of MDL and FGDL in a distributed setting can be drastically different from the classical setting.[2]

Example 4.1.2. Consider the monadic Datalog query Q, the distributed database $\mathcal{D}$, and the communication policy γ from Example 4.1.1. The rule $\mathrm{Out}(x) \leftarrow R(x), S(x)$ is not frontier-guarded. Replacing it with the following set of frontier-guarded rules yields a frontier-guarded Datalog query $Q' = (P', \mathrm{Out})$.

$$\begin{array}{ll}
\mathrm{Out}(x) \leftarrow R(x), S(x), \mathrm{Start}(x) & \mathrm{Out}(x) \leftarrow R(x), S(x), E_s(x, z) \\
\mathrm{Out}(x) \leftarrow R(x), S(x), E_r(x, z) & \mathrm{Out}(x) \leftarrow R(x), S(x), E_s(z, x) \\
\mathrm{Out}(x) \leftarrow R(x), S(x), E_r(z, x) &
\end{array}$$

[2] For spoilers, see Theorem 4.2.31 and Theorem 4.2.33.

Then we have $Q' \equiv Q$ because, for every fact $\text{Out}(\bar{a})$ that can be derived by the original rule $\text{Out}(x) \leftarrow R(x), S(x)$, the domain value a has to originate from some extensional fact.

But $[Q', \gamma](\mathcal{D}) \neq [Q, \gamma](\mathcal{D})$, since $\text{Out}(3) \in [Q, \gamma](\mathcal{D}) \setminus [Q', \gamma](\mathcal{D})$. Indeed, as outlined in Example 4.1.1, the fact $\text{Out}(3)$ can be derived on server 3 (and no other server) in the distributed evaluation of Q. This is not possible in the distributed evaluation of Q' because *no* extensional facts reside on server 3. ◁

4.1.2 Hash-Based Distribution Policies

As mentioned before, we only consider hash-based distribution policies for the specification of initial distributions. Such a distribution policy is composed of two components: A tuple $H = (h_1, \ldots, h_m)$ of *hash functions* over a network $\mathcal{N}$ and a *hash policy scheme* which describes *how* these hash functions are applied. We will define these terms next.

We define a hash function h over a network $\mathcal{N}$ as a function $h\colon \mathsf{dom}^n \to 2^{\mathcal{N}} \setminus \{\emptyset\}$ for some integer $n \geq 0$. The integer n is called the *arity* of h and is also denoted by $\mathsf{ar}(h)$.

A *hash directive for a relation symbol* R is a triple of the form $(R, i, \bar{u})$ where R is a relation symbol, $i \geq 1$ is an integer used to select a hash function, and $\bar{u} \in [1, \mathsf{ar}(R)]^n$ for some $n \geq 0$ is a tuple of indices. A *hash policy scheme* Z over schema $\mathcal{S}$ is a finite, non-empty set of hash directives that satisfies the following two conditions.

(a) For every $R \in \mathcal{S}$, there is *at least* one hash directive for R in Z.

(b) All pairs $(R, i, \bar{u}), (S, i, \bar{v})$ of hash directives from Z agree on the arity of the i-th hash function. That is, $|\bar{u}| = |\bar{v}|$.

We also refer to Condition (b) by saying that Z is *consistent*. The *size* $\|Z\|$ of a hash policy scheme Z is $\sum_{(R,i,\bar{u}) \in Z} 1 + \|i\| + |\bar{u}|$ where $\|i\|$ is the length of the binary coding of i.

Intuitively, a hash directive $(R, i, \bar{u})$ stipulates that each R-fact $R(\bar{a})$ is hashed using the i-th hash function. Thereby the indices $\bar{u}$ determine which values of $\bar{a}$ and in which order they are "fed" into h_i. More precisely, the intent is to invoke h_i with arguments $\bar{a}[\bar{u}] = (a_{u_1}, \ldots, a_{u_n})$. This is only well-defined if $|\bar{u}| = \mathsf{ar}(h_i)$ holds, however. In the following we will make this precise.

We say that Z is *compatible with* a tuple $H = (h_1, \ldots, h_m)$ of hash functions over a (common) network, if $i \leq m$ and $|\bar{u}| = \mathsf{ar}(h_i)$ holds for every hash directive $(R, i, \bar{u}) \in Z$. Observe that, for every hash policy scheme Z, there are compatible tuples of hash functions, thanks to Z being consistent (Condition (b)).

If Z compatible with a tuple $H = (h_1, \ldots, h_m)$ of hash functions, they induce the distribution policy $\delta_{Z,H}$ that maps every fact $R(\bar{a})$ to the set of all servers k, for which there is a hash directive $(R, i, \bar{u}) \in Z$ such that $k \in h_i(\bar{a}[\bar{u}])$. Let us point out that $\delta_{Z,H}$ maps indeed every database G to a distributed database $\mathcal{D} = (G, \mathcal{I})$ with global database G, thanks to Condition (a).

We note that $\delta_{Z,H}$ is *fact-based*, in the sense that it maps each fact over schema $\mathcal{S}$ to a set of servers from $\mathcal{N}$, independent of the other facts in a database.

Example 4.1.3. Consider the database

$$G = \{R(1,3), R(2,3), R(2,4), R(1,6), S(3,3), S(4,6), S(4,2), S(6,6), T(3), T(6)\}$$

and the hash policy scheme

$$Z = \{(R, 1, (2)), (S, 1, (1)), (S, 2, (1,2)), (T, 2, (1,1))\}$$

which is compatible with the tuple $H = (h_1, h_2)$ where h_1 and h_2 are the hash functions over the network $\mathcal{N} = [1,3]$ defined by

$$h_1(a) = \{(a \bmod 3) + 1\} \quad \text{and} \quad h_2(a,b) = \{((a+b) \bmod 2) + 2, (b \bmod 2) + 2\}$$

for all domain values a, b.

Then $\delta_{Z,H}(G)$ is the distributed database over $\mathcal{N}$ with the local databases

$$\begin{aligned} I_1 &= \{R(1,3), R(2,3), R(1,6), S(3,3), S(4,6), S(6,6)\}, \\ I_2 &= \{R(2,4), S(3,3), S(4,6), S(4,2), S(6,6), T(3), T(6)\}, \text{ and} \\ I_3 &= \{S(4,2), S(3,3), T(3)\}. \end{aligned}$$ ◁

A special case of hash-based distribution policies are *value-independent* distribution policies [cf. KAK20, p. 975]. That is, distribution policies which specify the servers for facts solely based on their relation symbols. In particular, if all hash directives of a hash policy scheme Z have the form $(R, i, ())$, it is only compatible with tuples of hash functions of arity 0. In that case every distribution policy $\delta_{Z,H}$ is value-independent. Some of our lower bounds already hold for these families of distribution policies.

Example 4.1.4. Consider the network $\mathcal{N} = [1,4]$ and the database

$$D = \{\text{Start}(1), E_r(1,3), E_r(1,4), E_s(1,2), E_s(2,3)\}$$

from Example 4.1.1 and the hash policy scheme

$$Z = \{(\text{Start}, 1, ()), (\text{Start}, 2, ()), (E_r, 1, ()), (E_s, 2, ())\}.$$

Then Z compatible with any tuple $H = (h_1, h_2)$ of hash functions where both h_1 and h_2 have arity 0. That is, both hash functions are effectively constants and $\delta_{Z,H}$ maps facts to servers solely based on their relation symbol.

For the concrete hash functions h_1 and h_2 with images $\{1\}$ and $\{2\}$, respectively, $\delta_{Z,H}(D)$ coincides with the distributed database $\mathcal{D}$ defined in Example 4.1.1. ◁

4.1.3 Constraint-Based Communication Policies

In this section we introduce the constraint-based formalism to define the communication policies which we study in this thesis. We present an alternative to this approach, namely hash-based communication policies, in Appendix B.

We borrow the formalism of distribution constraint introduced by Geck et al. [GNS20]. It is important to note that the distribution constraints from Geck et al. [GNS20] are far more general and targeted at specifying classes of distributions (including co-partitionings). We defer a more detailed comparison to Section 4.5.

Distribution constraints are syntactically very similar to query rules. The major (syntactical) difference is that they refer to so called distributed atoms instead of (classical) atoms.

Let svar be an infinite set of *server variables* (disjoint from the sets var, dom, and att). To avoid confusion we will often refer to the variables in var as *data variables* in this chapter. A *distributed atom* $A@\kappa$ consists of an atom A and a server variable $\kappa \in \mathsf{svar}$. This notation extends to sets $\mathcal{A}$ of atoms: $\mathcal{A}@k$ denotes the set $\{A@k \mid A \in \mathcal{A}\}$. A *distribution constraint* σ is a rule of the form

$$A_1@\kappa_1, \ldots, A_m@\kappa_m \to B@\lambda.$$

The set of distributed atoms $\{A_1@\kappa_1, \ldots, A_m@\kappa_m\}$ forms the *body* and the distributed atom $B@\lambda$ is the *head* of σ. We denote the body and head of σ by $\mathsf{body}(\sigma)$ and $\mathsf{head}(\sigma)$, respectively. We further require that $\mathsf{body}(\sigma)$ contains a distributed atom of the form $A@\lambda$ when $\mathsf{head}(\sigma) = B@\lambda$, that is, $\mathsf{body}(\sigma)$ contains a distributed atom with the server variable λ of the head atom. This will ensure that communication policies cannot "create" new servers. We define the *size* $\|\sigma\|$ of a distribution constraint σ of the same form as above as $\|B\| + 1 + \sum_{i=1}^{m}(\|A_i\| + 1)$, and the *size* $\|\Sigma\|$ of a finite set Σ of distribution constraints as $\sum_{\sigma\in\Sigma} \|\sigma\|$.

A distribution constraint σ is *data-moving* if the atom occurring in $\mathsf{head}(\sigma)$ also occurs in a distributed atom in the body. That is, when its head equals $B@\lambda$ then its body contains a distributed atom of the form $B@\kappa$ (possibly $\kappa = \lambda$). This will then imply that the atom B will be sent from server κ to server λ.

A *network aware valuation* ϑ *over a network* $\mathcal{N}$ is a partial mapping

$$\vartheta \colon \mathsf{var} \cup \mathsf{svar} \rightharpoonup \mathsf{dom} \cup \mathcal{N}$$

that maps server variables to servers in $\mathcal{N}$, and data variables to domain values. Given a distributed database $\mathcal{D} = (G, \mathcal{I})$ we write $\vartheta(A@\kappa) \in \mathcal{D}$ if $\vartheta(A) \in I_{\vartheta(\kappa)}$ holds.

Each finite set Σ of data-moving distribution constraints induces, for every network $\mathcal{N}$, a communication policy $\gamma_{\Sigma,\mathcal{N}}$ as follows: For each distributed database $\mathcal{D}$ over $\mathcal{N}$, a communicated fact $R(\bar{a})@k \triangleright \ell$ is in $\gamma_{\Sigma,\mathcal{N}}(\mathcal{D})$ if and only if there is a network aware valuation ϑ and a distribution constraint $\sigma \in \Sigma$ such that

$$\vartheta(\mathsf{head}(\sigma)) = R(\bar{a})@\ell,\ R(\bar{a})@k \in \vartheta(\mathsf{body}(\sigma)),\ \text{and}\ \vartheta(\mathsf{body}(\sigma)) \subseteq \mathcal{D}$$

hold. Observe that $\gamma_{\Sigma,\mathcal{N}}(\mathcal{D})$ is a communication policy over the schema of all relation symbols occurring in head atoms in Σ. Since we only consider communication policies over $\mathsf{idb}(P)$, for some Datalog program P, we thus implicitly require that all head atoms are intensional. On the other hand, we allow extensional atoms in the bodies of distribution constraint. Contrary to our hash-based distribution policies, communication policies induced by data-moving distribution constraints are *not* fact-based.

Example 4.1.5. Consider the set Σ consisting of the following two data-moving distribution constraints.

$$R(y)@\kappa, E_r(y,x)@\lambda \to R(y)@\lambda \qquad S(y)@\kappa, E_s(y,x)@\lambda \to S(y)@\lambda$$

Intuitively, for an arbitrary network $\mathcal{N}$, the induced communication policy $\gamma_{\Sigma,\mathcal{N}}$ sends each fact $R(b)$ to every server containing a matching fact $E_r(b,a)$. Analogously, $S(b)$ is sent to every server containing at least one fact $E_s(b,a)$. ◁

We conclude this section with the definition of families of policy pairs induced by hash policy schemes and data-moving distribution constraints. Recall that a policy pair consists of a distribution policy and a communication policy *over the same network*. Let Σ be a finite set of data-moving distribution constraints and let Z be a hash policy scheme. By $\mathcal{F}(Z,\Sigma)$ we denote the set of all policy pairs $(\delta_{Z,H}, \gamma_{\Sigma,\mathcal{N}})$, where H is a tuple of hash functions over network $\mathcal{N}$ and Z is compatible with H. By Hash-Constraints we denote the class of families $\mathcal{F}(Z,\Sigma)$, where Z is a hash policy scheme and Σ is a set of data-moving distribution constraints.

4.2 Parallel-Correctness

This section is dedicated to the study of the parallel-correctness problem. We start by formally defining parallel-correctness and the associated decision problem for Datalog queries and (families of) arbitrary policy pairs. Our first step is then to show that, to decide parallel-correctness, it suffices to consider certain combinations of distribution policies and databases which yield "scattered" distributed databases. Equipped with this insight, we study the parallel-correctness problem for the hash-based distribution policies and constrained-based communication policies introduced in Sections 4.1.2 and 4.1.3. More specifically, we show in Section 4.2.1 that for the fragments of monadic and frontier-guarded Datalog queries, which both enjoy a decidable containment problem, parallel-correctness is undecidable. Even for seemingly simple distribution policies. In the bulk of this chapter, which entails Sections 4.2.2 to 4.2.5, we then study restrictions of our distribution and communication policies that allow for a decidable parallel-correctness problem for monadic and frontier-guarded Datalog queries.

As mentioned, we start by formally defining parallel-correctness for Datalog queries and policy pairs.

Definition 4.2.1 (Parallel-Correctness)**.** A Datalog query Q is *parallel-correct* w.r.t. a policy pair (δ,γ) if $[Q,\gamma](\mathcal{D}) = Q(G)$ holds for every distributed database $\mathcal{D} = (G,\mathcal{I})$ that complies with δ.

A Datalog query Q is *parallel-correct* w.r.t. a family $\mathcal{F}$ of policy pairs if it is parallel-correct w.r.t. every policy pair in $\mathcal{F}$.

Example 4.2.2. Let Σ be the set of data-moving distribution constraints consisting of the rules

$$R(y)@\kappa, E_r(y,x)@\lambda \to R(y)@\lambda \qquad \text{and} \qquad S(y)@\kappa, E_s(y,x)@\lambda \to S(y)@\lambda$$

from Example 4.1.5. Furthermore, consider the hash policy scheme

$$Z = \big\{(\text{Start}, 1, (1)), (E_r, 1, (2)), (E_s, 1, (2))\big\},$$

and let the Datalog query $Q = (P, \text{Out})$ be as in Examples 2.4.6 and 4.1.1. Then Q is parallel-correct w.r.t. every policy pair $(\delta_{Z,H}, \gamma_{\Sigma,\mathcal{N}})$ where $\mathcal{N}$ is a network and H is a tuple consisting of a single unary hash function h over $\mathcal{N}$ with which Z is compatible. In other words, Q is parallel-correct w.r.t. $\mathcal{F}(Z, \Sigma)$.

Notice in particular that Σ does not have a distribution constraint enforcing facts $R(a)$ and $S(a)$ to end up on the same server, as this already follows from Z and Q. Indeed, every derivation of a fact $R(a)$ is witnessed by either a fact $\text{Start}(a)$ or some fact $E_r(b, a)$, and every derivation of a fact $S(a)$ is witnessed by some fact $\text{Start}(a)$ or some fact $E_s(b, a)$. By construction of Z, these facts always reside on the set of servers $h(a)$. As a result, every R-fact and its corresponding S-fact are always derived on the same set of servers $h(a)$, if at all. ◁

For classes $\mathbb{Q}$ of Datalog queries, and classes $\mathbb{F}$ of families of policy pairs, we write $\text{PC}(\mathbb{Q}, \mathbb{F})$ for the parallel-correctness problem which is defined as follows.

— $\text{PC}(\mathbb{Q}, \mathbb{F})$ —

Given: Datalog query $Q \in \mathbb{Q}$ and family $\mathcal{F} \in \mathbb{F}$ of policy pairs

Question: Is Q parallel-correct with respect to $\mathcal{F}$?

Since Datalog queries are monotone and neither distribution policies nor communication policies can "create" new facts, our distributed evaluation strategies are sound.

Observation 4.2.3. For every Datalog query Q, and communication policy γ over a network $\mathcal{N}$ the distributed evaluation strategy for Q induced by γ is sound. That is, $[Q, \gamma](\mathcal{D}) \subseteq Q(G)$ holds for all distributed databases $\mathcal{D} = (G, \mathcal{I})$ over $\mathcal{N}$.

Thanks to this observation, parallel-correctness can be decided by testing *parallel-completeness*: It suffices to test whether $Q(G) \subseteq [Q, \gamma](\mathcal{D})$ holds for all distributed databases $\mathcal{D} = (G, \mathcal{I})$. In the following we show that for parallel-completeness it is, in turn, even enough to consider distributed databases where facts are maximally *scattered* across all servers. We make the notion of scatteredness precise for our hash-based distribution policies next.

Definition 4.2.4. A tuple $H = (h_1, \ldots, h_m)$ of hash functions *scatters* a global database G if the following two conditions hold.

(a) $h_i(\bar{a}) \cap h_j(\bar{b}) = \emptyset$, for all $i, j \in [1, m]$ with $i \neq j$, and all tuples $\bar{a} \in \mathsf{adom}(G)^{\mathsf{ar}(h_i)}$, $\bar{b} \in \mathsf{adom}(G)^{\mathsf{ar}(h_j)}$; and

(b) $h_i(\bar{a}) \cap h_i(\bar{b}) = \emptyset$, for all $i \in [1, m]$, and tuples $\bar{a}, \bar{b} \in \mathsf{adom}(G)^{\mathsf{ar}(h_i)}$ with $\bar{a} \neq \bar{b}$.

A distribution policy $\delta_{Z,H}$ *scatters* G if H scatters G.

Thus, if $\delta_{Z,H}$ scatters a global database G, then two facts $R(\bar{a})$ and $S(\bar{b})$ are in the same local database of $\delta_{Z,H}(G)$ if and only if there is some i and triples $(R, i, \bar{u})$ and $(S, i, \bar{v})$ such that $\bar{a}[\bar{u}] = \bar{b}[\bar{v}]$.

Example 4.2.5. Consider the family $(G_m)_{m\geq 0}$ of global databases where, for every $m \geq 0$,

$$G_m = \{R(i) \mid i \in [0, m]\} \cup \{S(i) \mid i \in [0, m]\} \cup \{E(i-1, i) \mid i \in [1, m]\}.$$

Further, let Z be the hash policy scheme

$$Z = \{(R, 1, 1), (S, 1, 1), (E, 2, ())\}.$$

The tuple $H_m = (h_1, h_2)$ of hash functions over the network $\mathcal{N}_m = [0, m+1]$ with $h_1(i) = \{i\}$ for all $i \in [0, m]$, and $h_2(()) = m + 1$, scatters G_m. The distributed database $\mathcal{D}_m = \delta_{Z,H_m}(G_m)$ consists of the global database G_m and the local databases $I_i = \{R(i), S(i)\}$ for all $i \in [0, m]$ and $I_{m+1} = \{E(i-1, i) \mid i \in [1, m]\}$.

The tuple $H'_m = (h'_1, h'_2)$ over the same network $\mathcal{N}_m$ with $h_1(i) = \{i, i+1\}$ for all $i \in [0, m]$, and $h_2(()) = m + 1$, does *not* scatter G_m. However, note that the distributed database $\mathcal{D}'_m = \delta_{Z,H'_m}(G_m)$ with the local databases $I'_0 = \{R(0), S(0)\}$, $I'_i = \{R(i-1), S(i-1), R(i), S(i)\}$ for all $i \in [1, m]$, and $I'_{m+1} = \{R(m), S(m)\} \cup I_{m+1}$ complies with the distribution policy $\delta_{Z,H}$ that scatters G_m. ◁

We are now ready to state a characterization of parallel-correctness in terms of scattered databases which we will exploit to prove our main results in this chapter.

Lemma 4.2.6. *Let Q be a Datalog query, Z be a hash policy scheme, and Σ be a set of data-moving distribution constraints. Then Q is parallel-correct w.r.t. $\mathcal{F}(Z, \Sigma)$ if and only if, for all global databases G, there is a policy pair $(\delta, \gamma) \in \mathcal{F}(Z, \Sigma)$ such that δ scatters G and $[Q, \gamma](\delta(G)) \supseteq Q(G)$ holds.*

To prove Lemma 4.2.6 we require three ingredients. The first one is simply that every database can be scattered.

Lemma* 4.2.7. *For every database G and hash policy scheme Z, there is a tuple H of hash functions compatible with Z such that $\delta_{Z,H}$ scatters G.*

Proof. Let m be the maximum among all integers that occur in the second component of any hash directive in Z, and n be the maximal arity among all tuples $\bar{u}$ occurring in the third component of any hash directive in Z. That is, for every triple $(R, i, \bar{u}) \in Z$ we have $i \leq m$ and $|\bar{u}| \leq n$.

We fix the network $\mathcal{N} = \{(i, \bar{a}) \mid i \in [1, m], \bar{a} \in \mathsf{adom}(G)^j, j \in [0, n]\}$. Furthermore, we define, for every $i \leq m$, a hash function h_i of arity $\mathsf{ar}(h_i) = |\bar{u}|$ by setting $h_i(\bar{a}) = \{(i, \bar{a})\}$ for every $\bar{a} \in \mathsf{adom}(G)^{\mathsf{ar}(h_i)}$ if there is a triple $(R, i, \bar{u}) \in Z$. Note that h_i is well-defined due to Z being consistent. If, for some i, there is no matching triple, h_i does not play any role, and we can just define h_i as above but with arity 1. The tuple $H = (h_1, \ldots, h_m)$ is then compatible with Z.

It is easy to see that $h_i(\bar{a}) \cap h_j(\bar{b}) \neq \emptyset$ holds if and only if $i = j$ and $\bar{a} = \bar{b}$. Thus, $\delta_{Z,H}$ scatters G. □

While Lemma 4.2.7 states that every database G is scattered by some distribution policy $\delta_{Z,H}$, it makes no guarantees regarding the size of the underlying network. In fact, it is evident that the size of the network depends not only on Z but also on the size of G, in general.

Example 4.2.8. Consider the hash policy scheme $Z = \{(R, 1, 1)\}$ and the family $(G_i)_{i\geq 1}$ of databases with $G_i = \{R(a) \mid a \in [1, i]\}$ for all $i \geq 1$. Every hash function h_1 that scatters G_i is defined over a network with at least i servers. Otherwise, h_1 would map two values a_1, a_2, $a_1 \neq a_2$ to a common server by the pigeon-hole principle; violating Definition 4.2.4 (b). ◁

This observation leads to the second ingredient: We need a way to relate distributed databases over different networks. In particular, scattered databases with distributed databases over arbitrary networks.

For distributed databases $\mathcal{D} = (G, \mathcal{I})$ and $\mathcal{D}' = (G', \mathcal{I}')$ over networks $\mathcal{N}$ and $\mathcal{N}'$, respectively, we say that $\mathcal{D}'$ *covers* $\mathcal{D}$, if for each server $k \in \mathcal{N}$, there is a server $\ell \in \mathcal{N}'$, such that $I_k \subseteq I'_\ell$.

Example 4.2.9. Let $\mathcal{D} = (G, \mathcal{I})$ be the distributed database over the network $\{1, 2, 3\}$ with the local databases

$$\begin{aligned} I_1 &= \{R(1,2), R(5,2), E(2,1)\}, \\ I_2 &= \{E(2,1), S(3,7)\}, \text{ and} \\ I_3 &= \{S(1,5), R(1,5)\}; \end{aligned}$$

and $\mathcal{D}' = (G', \mathcal{J})$ be the distributed database over the network $\{a, b, c, d\}$ with the local databases

$$\begin{aligned} J_a &= \{R(1,2), R(5,2), E(2,1), S(3,7)\}, \\ J_b &= \{S(1,5), R(1,5), E(1,5)\}, \\ J_c &= \{E(2,1)\}, \text{ and} \\ J_d &= \{S(1,5), R(1,5)\}. \end{aligned}$$

We have that $\mathcal{D}'$ covers $\mathcal{D}$ because $I_1 \subseteq J_a$, $I_2 \subseteq J_a$, and $I_3 \subseteq J_b$ (or, alternatively, $I_3 \subseteq J_d$). Moreover, $\mathcal{D}$ is also covered by the distributed database $\mathcal{D}''$ over the network $\{a, b\}$ which consists of the local databases J_a and J_b as given above. ◁

Let us note that for two distributed databases $\mathcal{D}, \mathcal{D}'$ over the same network with $\mathcal{D} \subseteq \mathcal{D}'$ we always have that $\mathcal{D}'$ covers $\mathcal{D}$. However, the reserve is not true. For instance, if we identify the servers a, b in Example 4.2.9 with 1 and 2, respectively, then the distributed database $\mathcal{D}''$ covers $\mathcal{D}$ but $\mathcal{D} \subseteq \mathcal{D}''$ does *not* hold.

Combined with Lemma 4.2.7, the next lemma states that every database that complies with some hash-based distribution policy covers a scattered database.

Lemma* 4.2.10. *Let Z be a hash policy scheme, $\mathcal{D}' = (G', \mathcal{I}')$ be a distributed database, $G \subseteq G'$, and H, H' be tuples of hash functions compatible with Z. If $\delta_{Z,H}$ scatters G and $\mathcal{D}'$ complies with $\delta_{Z,H'}$ then $\mathcal{D}'$ covers $\delta_{Z,H}(G)$.*

Proof. Let $H = (h_1, \ldots, h_m)$, $H' = (h'_1, \ldots, h'_m)$, and I_k be a local database of $\delta_{Z,H}(G)$. We have to show that $I_k \subseteq I'_\ell$ holds for some local database I'_ℓ of $\mathcal{D}'$. If I_k contains only one (or no) fact, this is holds trivially, because $G \subseteq G'$ and every fact resides on some server. Thus, we can assume that I_k contains at least two facts.

Since $\delta_{Z,H}$ scatters G there is an i and a tuple $\bar{a}$ of domain values such that, for every fact $R(\bar{b}) \in I_k$, there is a hash directive $(R, i, \bar{u}) \in Z$ with $\bar{b}[\bar{u}] = \bar{a}$. Otherwise, there would be two tuples $\bar{a}_1, \bar{a}_2$ and i, j such that $k \in h_i(\bar{a}_1) \cap h_j(\bar{a}_2)$ with $i = j$ or $\bar{a}_1 = \bar{a}_2$; a contradiction to G being scattered by $\delta_{Z,H}$.

We can conclude that every fact in I_k also resides on all servers in $h'_i(\bar{a})$ because $\mathcal{D}'$ complies with $\delta_{Z,H'}$ and $G \subseteq G'$. Hence, we have $I_k \subseteq I'_\ell$ for any $\ell \in h'_i(\bar{a})$. □

Finally, the last ingredient is a monotonicity condition on our distributed evaluation strategies.

Lemma 4.2.11. *Let $Q = (P, \mathrm{Out})$ be a Datalog query, and Σ be a set of data-moving distribution constraints. For all distributed databases $\mathcal{D}$ and $\mathcal{E}$ over networks $\mathcal{N}$ and $\mathcal{N}'$, respectively, such that $\mathcal{E}$ covers $\mathcal{D}$ we have that*

$$[Q, \gamma_{\Sigma,\mathcal{N}}](\mathcal{D}) \subseteq [Q, \gamma_{\Sigma,\mathcal{N}'}](\mathcal{E}).$$

Furthermore, if a fact $R(\bar{a})$ is derived in round r in the distributed evaluation over $\mathcal{D}$, it is derived after at most r rounds in the evaluation over $\mathcal{E}$.

Proof. Let $\mathcal{D}$ and $\mathcal{E}$ be distributed databases over networks $\mathcal{N}$ and $\mathcal{N}'$, respectively, such that $\mathcal{E}$ covers $\mathcal{D}$. For each $r \geq 0$, we denote by $\mathcal{D}^r$ and $\mathcal{E}^r$ the distributed databases after the r-th communication phase of the distributed evaluation over $\mathcal{D}$ and $\mathcal{E}$, respectively. We show by induction on the number r of rounds that, for each $r \geq 0$, $\mathcal{E}^r$ covers $\mathcal{D}^r$. The statement of the lemma then follows immediately.

Initially, $\mathcal{E}^0 = \mathcal{E}$ covers $\mathcal{D}^0 = \mathcal{D}$ by presumption.

For the induction step, we assume that $\mathcal{E}^{r-1}$ covers $\mathcal{D}^{r-1}$. Let $\mathcal{I}^{r-1}$ and $\mathcal{J}^{r-1}$ denote the families of local databases of $\mathcal{D}^{r-1}$ and $\mathcal{E}^{r-1}$, respectively. Then there is a mapping $s \colon \mathcal{N} \to \mathcal{N}'$ such that, for every server $k \in \mathcal{N}$, we have $I_k^{r-1} \subseteq J_{s(k)}^{r-1}$. But then we also have

$$P(I_k^{r-1}) \subseteq P(J_{s(k)}^{r-1}),$$

for all servers k, because Datalog queries (and Datalog programs) are monotone.[3] In other words, $\mathcal{E}'$ covers $\mathcal{D}'$ where $\mathcal{D}'$ and $\mathcal{E}'$ are the distributed databases obtained from $\mathcal{D}$ and $\mathcal{E}$, respectively, after the r-th computation phase.

For the communication phase, it suffices to show that $R(\bar{a})@k \triangleright \ell \in \gamma_{\Sigma,\mathcal{N}}(\mathcal{D}')$ implies $R(\bar{a})@s(k) \triangleright s(\ell) \in \gamma_{\Sigma,\mathcal{N}'}(\mathcal{E}')$, for every communicated fact $R(\bar{a})@k \triangleright \ell$. For this purpose, let $R(\bar{a})@k \triangleright \ell$ be a communicated fact in $\gamma_{\Sigma,\mathcal{N}}(\mathcal{D}')$. Then there is distribution constraint $\sigma \in \Sigma$ and a network aware valuation ϑ such that $\vartheta(\mathsf{body}(\sigma)) \subseteq \mathcal{D}'$, $\vartheta(\mathsf{head}(\sigma)) = R(\bar{a})@\ell$, and $R(\bar{a})@k \in \vartheta(\mathsf{body}(\sigma))$ hold. But then we also have $\vartheta'(\mathsf{body}(\sigma)) \subseteq \mathcal{E}'$, $\vartheta'(\mathsf{head}(\sigma)) = R(\bar{a})@s(\ell)$, and $R(\bar{a})@s(k) \in \vartheta'(\mathsf{body}(\sigma))$ for $\vartheta' = s \circ \vartheta$. Here $\vartheta'(\mathsf{body}(\sigma)) \subseteq \mathcal{E}'$ holds

[3] We note that s is, in general, neither injective nor surjective.

because s witnesses that $\mathcal{E}'$ covers $\mathcal{D}'$. Thus, $R(\bar{a})@s(k) \triangleright s(\ell) \in \gamma_{\Sigma,\mathcal{N}'}(\mathcal{E}')$, and, overall, we can conclude that $\mathcal{E}^r$ covers $\mathcal{D}^r$. □

We note that Lemma 4.2.11 also entails, as special case, a formal proof for Observation 4.2.3 for constraint-based communication policies. Indeed, every $\mathcal{D} = (G, \mathcal{I})$ is covered by the distributed database $\mathcal{D}' = (G, (I_1))$ with $I_1 = G$ over the network $\{1\}$ consisting of a single server. Since the communication phases in the distributed evaluation over $\mathcal{D}'$ cannot change the only local database, we have $[Q, \gamma_{\Sigma,\{1\}}](\mathcal{D}') = Q(G)$. Thus, Lemma 4.2.11 implies $[Q, \gamma_{\Sigma,\mathcal{N}}](\mathcal{D}) \subseteq Q(G)$.

Further on, we are now ready to prove Lemma 4.2.6 by combining our ingredients stated as Lemma 4.2.7, Lemma 4.2.10, and Lemma 4.2.11.

Proof of Lemma 4.2.6. For the only-if direction suppose that Q is parallel-correct w.r.t. $\mathcal{F}(Z, \Sigma)$ and let G be an arbitrary global database. Thanks to Lemma 4.2.7 there is a tuple H of hash functions over a network $\mathcal{N}$ such that $\delta_{Z,H}$ scatters G. Since Q is parallel-correct w.r.t. $\mathcal{F}(Z, \Sigma)$ we can conclude that $[Q, \gamma_{\Sigma,\mathcal{N}}](\delta_{Z,H}(G)) \supseteq Q(G)$ holds.

For the converse, suppose that for every database G there is policy pair (δ, γ) such that δ scatters G and $[Q, \gamma](\delta(G)) \supseteq Q(G)$ holds. We have to show that Q is parallel-correct w.r.t. $\mathcal{F}(Z, \Sigma)$.

Let $(\delta', \gamma') \in \mathcal{F}(Z, \Sigma)$ and let $\mathcal{D} = (G, \mathcal{I})$ be an arbitrary distributed database that complies with δ'. Thanks to Observation 4.2.3 we have $[Q, \gamma'](\mathcal{D}) \subseteq Q(G)$. Hence, it suffices to establish parallel-completeness, by showing $[Q, \gamma'](\mathcal{D}) \supseteq Q(G)$.

By assumption there is a policy pair $(\delta, \gamma) \in \mathcal{F}(Z, \Sigma)$ such that δ scatters G and $[Q, \gamma](\delta(G)) \supseteq Q(G)$. Thanks to Lemma 4.2.10, $\mathcal{D}$ covers $\delta(G)$ because δ scatters G. Therefore, we have $[Q, \gamma](\delta(G)) \subseteq [Q, \gamma'](\mathcal{D})$ thanks to Lemma 4.2.11.

Altogether, we get $[Q, \gamma'](\mathcal{D}) \supseteq [Q, \gamma](\delta(G)) \supseteq Q(G)$. □

4.2.1 Undecidability for Hash-Constraints

The following result sharpens the undecidability result for parallel-correctness for Datalog from Ketsman et al. [KAK20, Theorem 1], since it states undecidability for fragments of Datalog for which the containment problem is decidable.

Theorem* 4.2.12. PC(FGDL, Hash-Constraints) *and* PC(MDL, Hash-Constraints) *are undecidable.*

The proof is by reduction from the complement of the halting problem for deterministic Minsky machines MinskyHalt that is well-known to be undecidable. We refer to Section 2.5.1 for references and the precise model we use.

Proof of Theorem 4.2.12. Let M be a Minsky machine with states $s_0, \ldots, s_n$ for some $n \geq 1$. Without loss of generality, we assume that s_0 is the initial state and s_n is the designated halting state. We construct a Datalog query Q, a hash policy scheme Z, and a set Σ of data-moving distribution constraints such that Q is parallel-correct w.r.t. $\mathcal{F}(Z, \Sigma)$ if and only if M does *not* halt.

Since the Datalog query Q which we define is monadic as well as frontier-guarded, our construction yields reductions from the complement of the halting problem for Minsky machines to the parallel-correctness problem for monadic Datalog queries and families of policy pairs in Hash-Constraints as well as to the parallel-correctness problem for frontier-guarded Datalog queries and families of policy pairs in Hash-Constraints.

The Idea. Assume a network with ℓ servers. The fundamental idea is to assign the i-th configuration of the computation of M to server i. Each server can then derive a distinguished fact – except if the state of its configuration is the halting state.

For this purpose, it is crucial that distribution constraints can "address" the "i-th server". With this in mind, we fix the underlying database schema: It consists of a binary relation symbol E, two unary relation symbols N and I, and one nullary relation symbol K. Our intention is for N to be a (continuous) set of numbers, i.e. $[0, \ell]$ for some $\ell > 0$, and E to be the (direct) successor relation over N. The intention for I is to identify the server to which the initial configuration gets assigned, and for K to identify a special auxiliary server whose purpose we will explain later.

More concretely, we are particularly interested in distributed databases $\mathcal{D}_\ell$ (for $\ell \geq 0$), with global database

$$G_\ell = \big\{K(), I(0)\big\} \cup \big\{N(i) \mid i \in [0, \ell]\big\} \cup \big\{E(i-1, i) \mid i \in [1, \ell]\big\}$$

and which are distributed as follows.

- The network $\mathcal{N}_\ell$ is $[0, \ell] \cup \{\mathsf{copy}\}$ where copy is a special auxiliary server we refer to as the *copy server*;
- the local database of the copy server copy is

$$\big\{K(), I(0)\big\} \cup \big\{E(i, i+1) \mid i \in [0, \ell-1]\big\};$$

- the local database of server 0 contains the facts $I(0)$ and $N(0)$; and,
- for $i \in [1, \ell]$, the local database of server i is $\{N(i)\}$.

Considering the local databases, each fact $N(i)$ resides on server i. Thus, they effectively assign a number to each server that is determined by the successor relation defined by E. We note that this not only allows us to identify the i-th server but also to increase and decrease counters.

Configurations will be represented using intensional relations computed by the Datalog program. More precisely, a configuration (s_j, c_1, c_2) will be represented by three facts $S_j(i)$, $C_1(c_1)$, and $C_2(c_2)$ residing on server i. We emphasize that, to assign the configuration to server i, it is not only required that $S_j(i)$, $C_1(c_1)$, and $C_2(c_2)$ reside on server i, but also that no other S_ℓ-, C_1-, or C_2-facts reside on i. Proper distribution of these intensional facts will be handled by the communication policy.

The Datalog Query. We start with the definition of the Datalog query $Q = (P, \mathrm{Out})$. As already mentioned, the intensional relations symbols of the Datalog program P include the unary symbols C_1 and C_2 for counter values, and, for each $j \in [0, n]$, the unary symbol S_j for state s_j. Additionally, Out serves as the output symbol and N' serves as a copy of N. The frontier-guarded, monadic Datalog program P consists of the following rules.

$$
\begin{aligned}
N'(x) &\leftarrow N(x) && \\
R(x) &\leftarrow N'(x), I(x), K() && \text{for all } R \in \{S_j \mid j \in [0,n]\} \cup \{C_1, C_2\} \\
R(y) &\leftarrow N'(y), R(x), E(x,y), K() && \text{for all } R \in \{S_j \mid j \in [0,n]\} \cup \{C_1, C_2\} \\
\mathrm{Out}(x) &\leftarrow S_j(x), N(x) && \text{for all } j \in [0, n-1]
\end{aligned}
$$

Let us briefly discuss the behaviour of P over a (global) database (not necessarily G_ℓ). In general, the binary relation E can be viewed as the edge relation of a graph whose nodes may be labelled with N or I. If $K()$ is not in the database, no Out-facts can be derived because no S_j-facts can be derived. If $K()$ is in the database, the first three rules determine all nodes that are reachable from a I-labelled node via a N-labelled path. Indeed, the first rule copies N into N'. The rules of the second kind initialize the computation from a I- and N-labelled node, while rules of the third kind recursively traverse over N-labelled edges. The computed vertices are stored in the intensional relations C_1, C_2, and S_j for all $j \in [0, n]$. In the following this is stated more formally.

Claim 4.2.13. *Let $R \in \{S_j \mid j \in [0,n]\} \cup \{C_1, C_2\}$ and D be a (global) database. Then $R(b) \in P(D)$ if and only if $K() \in D$ and there is a sequence of domain values $a_0, \ldots, a_\ell$ with $\ell \geq 0$ such that $a_\ell = b$, $I(a_0) \in D$, $\{N(a_i) \mid i \in [0, \ell]\} \subseteq D$, and $\{E(a_{i-1}, a_i) \mid i \in [1, \ell]\} \subseteq D$.*

Notably, the relations C_1, C_2, and S_j computed by P over a global database are identical and considering the evaluation over G_ℓ they end up being copies of N. Finally, the rules of the last kind "output" every node derived by other rules as described above. In particular, over G_ℓ, the query result is $\{\mathrm{Out}(0), \mathrm{Out}(1), \ldots, \mathrm{Out}(\ell)\}$.

We emphasize that there is *no* rule $\mathrm{Out}(x) \leftarrow S_n(x), N(x)$ for the halting state s_n. While this does not matter in the sequential setting – because all the S_j are copies of each other – it is crucial for the distributed evaluation and the correctness of the reduction. In a nutshell, on a server to which a S_n-fact but no other S_j-facts get assigned, no Out-facts can be derived. It will also matter that the rules of the second and third kind refer to the intensional relation symbol N' and the rules of the last kind, on the other hand, to N.

The Hash Policy Scheme. Next, we define the hash policy scheme Z for the distribution policies. Our intention here is to obtain the distributed databases $\mathcal{D}_\ell$ if the global databases G_ℓ is scattered. We set

$$Z = \big\{(E, 1, ()), (I, 1, ()), (K, 1, ()), (I, 2, (1)), (N, 2, (1))\big\}.$$

Indeed, for every tuple H of hash functions that is compatible with Z and scatters a global database G the following holds. All E-, I-, and K-facts are mapped to a single

server due to the first three hash directives in Z. Furthermore, due to the last two hash directives in Z, for all domain values a, the facts $N(a)$ and $I(a)$ are assigned to a distinguished server for a, if they exist. In particular, for the global databases G_ℓ we have the following.

Claim 4.2.14. *For every $\ell \geq 0$, there is a tuple H_ℓ of hash functions compatible with Z such that δ_{Z,H_ℓ} scatters G_ℓ and $\delta_{Z,H_\ell}(G_\ell) = \mathcal{D}_\ell$.*

In fact, a tuple H_ℓ attesting Claim 4.2.14 is $H_\ell = (h_1, h_2)$ where h_1 is a nullary hash function that maps every tuple to the server copy and h_2 is a unary hash function that maps every $i \in [0, \ell]$ to server i. The latter implies that δ_{Z,H_ℓ} maps all facts $N(i)$ and $I(i)$ to server i.

The Distribution Constraints. It remains to construct a set of data-moving distribution constraints. This is the most involved part of the construction because the distribution constraints actually simulate the Minsky machine M. A bit more precisely, at the end of the distributed evaluation we intend for the following conditions to hold.

(a) A fact $C_j(c)$ resides on server i if and only if the value of counter j in the i-th configuration in the computation of M is c; and,

(b) a fact $S_j(i)$ resides on server i if and only if the state of i-th configuration in the computation of M is s_j.

Recall that the hash policy scheme Z guarantees that all E-, I-facts, and, if present, the K-fact, reside together on (at least) one server, which we will refer to as the copy server. The following constraint makes the N'-facts available to the copy server as well.

$$K()@\kappa, N(x)@\lambda \to N'(x)@\kappa$$

Observe that this allows the copy server to derive all C_1-, C_2-, and, for all $j \in [0, n]$, S_j-facts that can be derived during the sequential evaluation. If the distributed database is scattered, the copy server can, however, not derive any output facts, because *no* N-facts will reside on it. We always intend for the server variable κ to be mapped to the copy server.

The following constraints assign the initial configuration of M to a server.

$$I(x)@\lambda, N(x)@\lambda, S_0(x)@\kappa \to S_0(x)@\lambda$$
$$I(x)@\lambda, N(x)@\lambda, C_1(x)@\kappa \to C_1(x)@\lambda$$
$$I(x)@\lambda, N(x)@\lambda, C_2(x)@\kappa \to C_2(x)@\lambda$$

Indeed, in the distributed evaluation over $\mathcal{D}_\ell$, the facts $S_0(0)$, $C_1(0)$, and $C_2(0)$ are present on the copy server after the second computation phase and the only matching pair of I-, and N-facts is $I(0), N(0)$ on server 0. Thus, the facts $S_0(0)$, $C_1(0)$, and $C_2(0)$ are sent to server 0 and no other server.

We model an instruction $\mathrm{Inc}(1, s_h)$ of M which transitions from state s_j to state s_h by incrementing the first counter by the three constraints

$$\mathcal{A}^+, C_1(v)@\kappa \to C_1(v)@\mu, \quad \mathcal{A}^+, C_2(w)@\kappa \to C_2(w)@\mu, \quad \mathcal{A}^+, S_h(z)@\kappa \to S_h(z)@\mu,$$

where $\mathcal{A}^+$ is

$$\underbrace{N(y)@\lambda, S_j(y)@\lambda, C_1(u)@\lambda, C_2(w)@\lambda,}_{(1)} \quad \underbrace{N(z)@\mu,}_{(2)} \quad \underbrace{E(u,v)@\kappa, E(y,z)@\kappa}_{(3)}.$$

Intuitively, $\mathcal{A}^+$ expresses that

(1) number y is assigned to server λ, and the y-th configuration in the computation is (s_j, u, w), i.e. M is in state s_j and the counters have value u and w, respectively;

(2) number z is assigned to server μ; and,

(3) (the copy) server κ asserts that $z = y + 1$, i.e. server μ is the server for the $(y+1)$-th configuration, and the new value of the first counter is $v = u + 1$ hold.

Incrementing the second counter can be modelled analogously by swapping C_1 and C_2.

Modelling a decrement instruction $\mathrm{Dec}(1, s_h, s_k)$ requires constraints for two cases, as the result depends on whether the first counter is zero or not. Recall that M either transitions to the state s_h and decrements the first counter, if it is non-zero, or transitions to state s_k without changing any counter, otherwise.

The former case is modelled similarly to the increment instruction. We underline the atom(s) that differ between $\mathcal{A}^+$ and $\mathcal{A}^-$ in the following.

$$\mathcal{A}^-, C_1(v)@\kappa \to C_1(v)@\mu, \quad \mathcal{A}^-, C_2(w)@\kappa \to C_2(w)@\mu, \quad \mathcal{A}^-, S_h(z)@\kappa \to S_h(z)@\mu,$$

where $\mathcal{A}^-$ is

$$N(y)@\lambda, S_j(y)@\lambda, C_1(u)@\lambda, C_2(w)@\lambda, \quad N(z)@\mu, \quad \underline{E(v,u)@\kappa}, E(y,z)@\kappa.$$

Note that $\mathcal{A}^-$ is almost the same as $\mathcal{A}^+$; it lets server κ assert $v = u - 1$ instead of $v = u + 1$. This also implies that u is not zero (in the sense that it has a predecessor).

Finally, we model the latter case with the following constraints. Again we underline differences to the former case.

$$\mathcal{A}^0, C_1(v)@\kappa \to C_1(v)@\mu, \quad \mathcal{A}^0, C_2(w)@\kappa \to C_2(w)@\mu, \quad \mathcal{A}^0, \underline{S_k(z)@\kappa} \to \underline{S_k(z)@\mu},$$

where $\mathcal{A}^0$ is

$$N(y)@\lambda, S_j(y)@\lambda, \underline{C_1(v)@\lambda}, C_2(w)@\lambda, \quad N(z)@\mu, \quad \underline{I(v)@\kappa}, E(y,z)@\kappa.$$

Here $\mathcal{A}^0$ lets server κ assert that the value v of the first counter is zero and is not changed.

Decrementing the second counter can again be modelled analogously.

We can now prove the following which implies Conditions (a) and (b). As usual $\mathcal{D}_\ell^r$ denotes the distributed database obtained after the r-th communication phase of the distributed evaluation over $\mathcal{D}_\ell$ induced by $\gamma_{\Sigma,\mathcal{N}_\ell}$.

Claim 4.2.15. *For every $\ell \geq 0$ and $i \in [0, \ell]$, the i-th configuration in the computation of M is (s_j, c_1, c_2) if and only if $S_j(i)@i$, $C(c_1)@i$, and $C_2(c_2)@i$ are in $\mathcal{D}^r_\ell$ for all $r \geq i+2$.*

The proof of Claim 4.2.15 is by induction on $i \geq 0$. For the induction start observe that, by construction, all facts $N'(i)$ reside on the copy server after the first round. Further, the copy server can then derive all facts $R(i)$ for $R \in \{S_j \mid j \in [0, n]\} \cup \{C_1, C_2\}$. This follows analogously to Claim 4.2.13 and by the observation that N' is a copy of N and the relevant Datalog rules actually refer to N'. Note that, up to this point, besides the communication of the N'-facts to the copy server, no other facts have been communicated because no other facts had even been derived. In the second communication phase the facts $S_0(0)$, $C_1(0)$, and $C_2(0)$ are then communicated to server 0 (and no other server) thanks to the constraints for the initial configuration and $I(0)$ and $N(0)$ residing on server 0 (and no other server).

For the induction step it suffices to observe that, as detailed earlier, facts $S_h(i)$, $C_1(c_1)$, and $C_2(c_2)$ are communicated to server i if and only if $S_j(i-1)$, $C_1(c'_1)$, and $C_2(c'_2)$ reside on server $i-1$ and (s_k, c_1, c_2) is the successor configuration of (s_j, c'_1, c'_2) in the unique computation of M.

Correctness. It remains to prove that M does *not* halt if and only if Q is parallel-correct w.r.t. $\mathcal{F}(Z, \Sigma)$.

Suppose M halts. Then there is an $\ell > 0$ such that the state of ℓ-th configuration in the computation of M is the halting state s_n. To show that Q is *not* parallel-correct, we consider the global database G_ℓ. Thanks to Claim 4.2.14 there is a tuple H_ℓ of hash functions such that $\delta_{Z,H_\ell}(G_\ell) = \mathcal{D}_\ell$. Furthermore, Claim 4.2.13 implies that $\mathrm{Out}(\ell) \in Q(G_\ell)$.

It suffices to show that $\mathrm{Out}(\ell) \notin [Q, \gamma_{\Sigma,\mathcal{N}_\ell}](\mathcal{D}_\ell)$. Due to Claim 4.2.15 and the state of the ℓ-th configuration being the halting state s_n, there will be no fact $S_j(\ell)$ with $j < n$ residing on server ℓ after the distributed evaluation has reached a global fixpoint. But then $\mathrm{Out}(\ell)$ cannot be derived on any server because $N(i)$ resides only on server ℓ and Out-facts can only be derived with rules of the form $\mathrm{Out}(x) \leftarrow S_j(x), N(x)$ for $j < n$. We can conclude $\mathrm{Out}(\ell) \notin [Q, \gamma_{\Sigma,\mathcal{N}_\ell}](\mathcal{D}_\ell)$, and, thus Q is *not* parallel-correct w.r.t. $\mathcal{F}(Z, \Sigma)$.

For the converse, suppose M does *not* halt. To prove parallel-correctness it suffices to show $[Q, \gamma](\mathcal{D}) \supseteq Q(G)$ for all $(\delta, \gamma) \in \mathcal{F}(Z, \Sigma)$ and all distributed database $\mathcal{D} = (G, \mathcal{I})$ that comply with δ, thanks to Observation 4.2.3. For this purpose, pick such δ, γ and $\mathcal{D} = (G, \mathcal{I})$.

Suppose $\mathrm{Out}(b) \in Q(G)$. By construction $P(G)$ then contains a fact $S_j(b)$. Due to Claim 4.2.13 this implies that G contains $K()$, and there is a sequence of domain values $a_0, \ldots, a_\ell$ such that $a_\ell = b$, $I(a_0) \in G$, $\{N(a_i) \mid i \in [0, \ell]\} \subseteq G$ and $\{E(a_{i-1}, a_i) \mid i \in [1, \ell]\} \subseteq G$. Let $G' \subseteq G$ be the database consisting of these facts. Observe that G' is isomorphic to G_ℓ. Indeed, each a_i can just be identified with i.

Further, let δ_{Z,H_ℓ} be the distribution policy with $\delta_{Z,H_\ell}(G_\ell) = \mathcal{D}_\ell$ that scatters G_ℓ guaranteed by Claim 4.2.14. Since M does *not* halt and thanks to Claim 4.2.15 we can conclude that after the distributed evaluation, a fact $S_j(\ell)$ for $j < n$ resides on server ℓ. Thus, $\mathrm{Out}(\ell) \in [Q, \gamma_{\Sigma,\mathcal{N}_\ell}](\mathcal{D}_\ell)$.

Since G' is isomorphic to G_ℓ and ℓ can be identified with $b = a_\ell$, we know that there is a policy pair $(\delta', \gamma') \in \mathcal{F}(Z, \Sigma)$ such that δ' scatters G' and $\mathrm{Out}(b) \in [Q, \gamma'](\delta'(G'))$. Furthermore, $\mathcal{D}$ covers $\delta'(G')$ due to Lemma 4.2.10 because δ' scatters G' and $G' \subseteq G$.

Finally, thanks to the monotonicity guaranteed by Lemma 4.2.11, we can conclude that $\mathrm{Out}(b) \in [Q, \gamma](\mathcal{D})$. □

4.2.2 Value-Independent Distribution Policies

Theorem 4.2.12 sharpens the undecidability result for parallel-correctness of Datalog queries from Ketsman et al. [KAK20, Theorem 1] in the sense that it states undecidability for fragments of Datalog for which the containment problem is decidable. On the other hand, the undecidability result from Ketsman et al. is stated for value-independent policies, i.e. policies which map facts to servers solely based on their relation symbol (cf. Example 4.1.4 and the discussion preceding it).

Recall that, in our setting, a hash policy scheme Z induces a family of value-independent distribution policies if and only if every triple in Z has the form $(R, i, ())$, i.e. all compatible hash functions have arity 0. We call such a hash policy scheme Z *primitive*. By Ind-Constraints we denote the class of all families $\mathcal{F}(Z, \Sigma)$ where Z is a primitive hash policy scheme and Σ is a set of data-moving distribution constraints.

The hash policy scheme and the distribution constraints constructed in the proof of Theorem 4.2.12 do *not* induce a family in Ind-Constraints. In fact, the correctness of the reduction relies on every extensional fact $N(i)$ residing on a distinguished server for the domain value i. We show next that for (our) fragments of Datalog with a decidable containment problem and value-independent distribution policies, parallel-correctness is decidable. The complexity is inherited from the containment problem.

Proposition 4.2.16. PC(FGDL, Ind-Constraints) *and* PC(MDL, Ind-Constraints) *are* 2ExpTime*-complete.*

We prove Proposition 4.2.16 by reductions from and to the containment problems for frontier-guarded and monadic Datalog queries. We focus on the upper bound first. For the reduction to the containment problem we will construct a Datalog program that simulates the distributed evaluation of the original query over a scattered database.

To argue about the correctness of our construction, we will utilize proof trees for facts derived in the distributed evaluation. We define them in the straightforward way by combining proof trees for facts with respect to the Datalog program with proof trees for communicated facts, whose communication is specified by distribution constraints. Recall that, for a rooted tree T, we denote its set of nodes by $\mathsf{nodes}(T)$, its root by $\mathsf{root}(T)$, and, for all nodes $v \in \mathsf{nodes}(T)$, the set of children of v by $\mathsf{children}_T(v)$.

Definition 4.2.17. A *proof tree* T over a network $\mathcal{N}$ for a fact $R(\bar{a})@k$ with $k \in \mathcal{N}$ with respect to a Datalog program P, and a set Σ of data-moving distribution constraints, is a rooted tree, in which every node v is labelled with a fact $S(\bar{b})@\ell = \mathsf{fact}(v)$ with $\ell \in \mathcal{N}$, and which has the following properties.

(a) The root node is labelled with $\mathsf{fact}(\mathsf{root}(T)) = R(\bar{a})@k$.

(b) For every inner node v labelled $S(\bar{b})@\ell$, there is either

- a Datalog rule $\tau \in P$ and a valuation ϑ such that $\vartheta(\mathsf{head}(\tau)) = S(\bar{b})$ and $\vartheta(\mathsf{body}(\tau))@\ell = \{\mathsf{fact}(w) \mid w \in \mathsf{children}_T(v)\}$, or
- a distribution constraint $\sigma \in \Sigma$ and a network aware valuation ϑ such that $\vartheta(\mathsf{head}(\sigma)) = S(\bar{b})@\ell$ and $\vartheta(\mathsf{body}(\sigma)) = \{\mathsf{fact}(w) \mid w \in \mathsf{children}_T(v)\}$.

(c) Every leaf is labelled with a fact $E(\bar{c})@\ell$ where $E(\bar{c})$ is an *extensional* fact over $\mathsf{edb}(P)$.

We say that an inner node v of a proof tree is witnessed by a Datalog rule τ and a valuation ϑ as shorthand for τ and ϑ witnessing v having Property (b). Analogously, we say that v is witnessed by a constraint σ and a network aware valuation ϑ if they witness v having Property (b).

A *partial proof tree* over $\mathcal{N}$ for a fact with respect to P and Σ is a rooted, labelled tree that satisfies Properties (a) and (b) of Definition 4.2.17 but not necessarily Property (c). In other words, its leaves may be labelled with intensional facts.

A (partial) proof tree T with respect to a distributed database $\mathcal{D} = (G, \mathcal{I})$ is a (partial) proof tree, whose leaves are labelled with facts $S(\bar{b})@\ell$ in $\mathcal{D}$.

Analogously to the sequential setting, for a Datalog program P, a set Σ of data-moving distribution constraints, and a distributed database $\mathcal{D}$, $[P, \gamma_{\Sigma,\mathcal{N}}](\mathcal{D})$ consists of exactly those facts $R(\bar{a})@k$ for which there is a proof tree over $\mathcal{N}$ with respect to P, Σ, and $\mathcal{D}$.

Lemma 4.2.18. *Let $Q = (P, \mathrm{Out})$ be a Datalog query, Σ be a set of data-moving distribution constraints, and $\mathcal{D}$ be a distributed database over a network $\mathcal{N}$. For every fact $R(\bar{a})@k$ there is a proof tree over $\mathcal{N}$ with respect to P, Σ, and $\mathcal{D}$ if and only if $R(\bar{a})@k \in [P, \gamma_{\Sigma,\mathcal{N}}](\mathcal{D})$.*

In particular, $R(\bar{a})$ can be derived in at most r rounds in the distributed evaluation of P over $\mathcal{D}$ according to $\gamma_{\Sigma,\mathcal{N}}$ if and only if there is a proof tree over $\mathcal{N}$ with respect to P, Σ, and $\mathcal{D}$ such that, on every root-to-leaf path, at most r nodes are witnessed by a constraint from Σ.

Proof. The proof for the only-if direction is by induction over the structure of a proof tree. The induction start is trivial and, for the induction step, it suffices to observe that a fact $\mathsf{fact}(v)$ can be derived by the Datalog rule (or communicated due to a distribution constraint), and (network aware) valuation witnessing the node v. The induction hypothesis guarantees that the required facts reside on the proper servers. We note that, since the computation phase precedes the communication phase and involves a local fixpoint computation, the evaluation only proceeds to the next round if v is witnessed by a distribution constraint (this has to be taken into account for the induction hypothesis).

For the converse, the proof is by induction over the number of rounds. The induction step involves a nested induction over the number of applications of the immediate consequence operator for P. The Datalog rule (or distribution constraint) and (network aware) valuation used to derive (respectively, communicate) a fact are proper witnesses for the root node of the proof tree. The existence of suitable proof trees for the facts

required to satisfy the body of the rule (or constraint) is guaranteed by the induction hypothesis. □

We are now ready to state our reduction for the upper bound of Proposition 4.2.16.

Lemma 4.2.19. *For every Datalog query Q, primitive hash policy scheme Z, and set Σ of data-moving distribution constraints, a Datalog query Q' can be constructed in exponential time such that the following holds. For every global database G there is a policy pair $(\delta, \gamma) \in \mathcal{F}(Z, \Sigma)$ such that δ scatters G and $[Q, \gamma](\delta(G)) = Q'(G)$.*

The number of variables and the length of rules of Q' is polynomial in $\|Q\|$, $\|Z\|$, and $\|\Sigma\|$; and the number of rules is at most exponential. Furthermore, if Q is frontier-guarded (or monadic), then Q' is frontier-guarded (or monadic, respectively).

Proof. Let $Q = (P, \mathrm{Out})$, and m be such that, for every hash directive $(R, i, ()) \in Z$, we have $i \leq m$. We first construct a Datalog query $Q'' = (P'', \mathrm{Out})$ that has the desired properties except that it may not be frontier-guarded or monadic, even if Q is. We will rectify this in a second step.

In addition to the output symbol Out, the Datalog program P'' uses an intensional relation symbol S^i of arity $\mathsf{ar}(S)$ for every $S \in \mathsf{edb}(P) \cup \mathsf{idb}(P)$ and $i \in [1, m]$. The idea being that a fact $S^i(\bar{a})$ corresponds to a fact $S(\bar{a})@i$ in the distributed evaluation over the network $[1, m]$. The Datalog program P'' has the following rules for Z, P, Σ, and Out respectively.[4]

- For every $(R, i, ()) \in Z$, P'' has a rule $R^i(\bar{x}) \leftarrow R(\bar{x})$ where $\bar{x}$ is some tuple of pairwise different variables.
- For every rule $R(\bar{x}) \leftarrow S_1(\bar{y}_1), \ldots, S_n(\bar{y}_n)$ of P and $i \in [1, m]$, P'' has a rule
$$R^i(\bar{x}) \leftarrow S_1^i(\bar{y}_1), \ldots, S_n^i(\bar{y}_n).$$
- For every distribution constraint $S_1(\bar{y}_1)@\kappa_1, \ldots, S_n(\bar{y}_n)@\kappa_n \to R(\bar{x})@\lambda$ in Σ and mapping $s\colon \mathsf{svars} \to [1, m]$, P'' has a rule
$$R^{s(\lambda)}(\bar{x}) \leftarrow S_1^{s(\kappa_1)}(\bar{y}_1), \ldots, S_n^{s(\kappa_n)}(\bar{y}_n).$$
- For every $i \in [1, m]$, P'' has a rule $\mathrm{Out}(\bar{x}) \leftarrow \mathrm{Out}^i(\bar{x})$ where $\bar{x}$ is some tuple of pairwise different variables.

The number of variables and length of rules in P'' is indeed polynomial in the size of Q, Z, and Σ. The number of rules constructed for Z, P, and Out is polynomial as well. For Σ the number of rules is exponential in the number of server variables in Σ.

For the correctness, consider the network $\mathcal{N} = [1, m]$ and the tuple $H = (h_1, \ldots, h_m)$ of nullary hash functions over $\mathcal{N}$ with $h_i(()) = i$ for all $i \in [1, m]$. Observe that the distribution policy $\delta_{Z,H}$ scatters every global database G.

[4] Strictly speaking, all symbols S^i for which *no* rule is constructed are removed, along with all rules where S^i occurs in the body, and recursively.

It is straightforward to translate a proof tree for a fact $\text{Out}^i(\bar{a})$ with respect to P'' and G into a proof tree over $\mathcal{N}$ for $\text{Out}(\bar{a})@i$ with respect to P, Σ and $\delta_{Z,H}(G)$; and vice versa. Indeed, it suffices to replace every label $R^k(\bar{b})$ of an inner node with $R(\bar{b})@k$. The leaves, which are all labelled with $\mathsf{edb}(P)$-facts, can just be removed, since, by construction, the parent node of a leaf labelled $E(\bar{a})$ is labelled with $E^k(\bar{a})$ and $(E, k, ()) \in Z$. The latter implies that $E(\bar{a})@k \in \delta_{Z,H}(G)$. Note that no Out-fact can appear in a proof tree for $\text{Out}^i(\bar{a})$ with respect to P'' because Out does *not* occur in the body of any rule in P''.

The translation in the other direction is analogous. Here proper leaves have to be added, which is possible thanks to the rules for Z.

We can conclude that $[Q, \gamma](\delta(G)) = Q''(G)$ holds thanks to the additional rules for the output symbol Out in P''.

It remains to show that, if Q is frontier-guarded or monadic then we can obtain Datalog query Q' – equivalent to Q'' – that is frontier-guarded or monadic, respectively.

To this end, suppose that Q is monadic. Then every S^i with $S \in \mathsf{idb}(P)$ is unary. But the symbols E^i with $E \in \mathsf{edb}(P)$ are, in general, not. However, the only rules where such a symbol E^i occurs in the head are the rules of the form $E^i(\bar{x}) \leftarrow E(\bar{x})$. The monadic Datalog query Q' can thus be obtained from Q'' by removing the rules $E^i(\bar{x}) \leftarrow E(\bar{x})$ for Z and replacing every occurrence of E^i in the body of any other rule by E.

Next, suppose that Q is frontier-guarded. The rules $E^i(\bar{x}) \leftarrow E(\bar{x})$ of P'' for Z are frontier-guarded because Z only refers to extensional symbols E. In the following we show how to replace the rules for P, Σ, and Out by frontier-guarded rules step by step.

For the first step, consider a rule τ of the form $R^i(\bar{x}) \leftarrow S_1^i(\bar{y}_1), \ldots, S_n^i(\bar{y}_n)$ constructed for a rule $R(\bar{x}) \leftarrow S_1(\bar{y}_1), \ldots, S_n(\bar{y}_n)$ of P. Since Q is frontier-guarded, there is a j such that $S_j(\bar{y}_j)$ is extensional and $\bar{y}_j$ contains all variables from $\bar{x}$. Hence, there also is a rule $S_j^i(\bar{z}) \leftarrow S_j(\bar{z})$ with $\bar{z}$ being a tuple of pairwise different variables. A frontier-guarded replacement of τ can therefore be obtained by replacing the atom $S_j^i(\bar{y}_j)$ in $\mathsf{body}(\tau)$ with $S_j(\bar{y}_j)$.

In the second step, rules constructed for a data-moving distribution constraint are replaced. For every rule τ with head $R^i(\bar{x})$ constructed for a data-moving distribution constraint σ, there is an atom $R^j(\bar{x}) \in \mathsf{body}(\tau)$ because σ is data-moving. Since this is true for every rule constructed for constraints in Σ, we have that in every proof tree for a fact $R^i(\bar{a})$ there is a node labelled $R^k(\bar{a})$ for some k and it is witnessed by a rule for P. Thanks to the previous step, there then is also an extensional fact $E(\bar{b})$ in this proof tree for some $\bar{b}$ that contains all values of $\bar{a}$. Therefore, the rule τ can be replaced with the set of rules defined as follows. For all extensional relation symbols E and tuples $\bar{z}$ of variables with $|\bar{z}| = \mathsf{ar}(E)$ which contains all variables in $\bar{x}$, the frontier-guarded rule $\tau_{E,\bar{z}}$ is obtained from τ by adding the atom $E(\bar{z})$ to $\mathsf{body}(\tau)$. The rule τ is then replaced with the set of all rules $\tau_{E,\bar{z}}$ obtained this way. Note that the number of rules $\tau_{E,\bar{z}}$ is at most exponential in $\|Q\|$, $\|Z\|$, and $\|\Sigma\|$; and their length is still polynomial. The number of new variables can be limited by $\max_{E \in \mathsf{edb}(P)} \mathsf{ar}(E)$.

Finally, every rule of the form $\text{Out}(\bar{x}) \leftarrow \text{Out}^i(\bar{x})$ can be replaced by the set of rules $\text{Out}(\bar{z}) \leftarrow \mathsf{body}(\tau)$ for every (frontier-guarded) rule τ with head $\text{Out}^i(\bar{z})$ constructed in the previous steps. □

As a final ingredient, we require a containment test for monadic and frontier-guarded Datalog queries over global databases. Bourhis et al. [BKR15a, Theorem 7] have shown that the containment problem Cont(DL, FGDL) is in 2ExpTime. Since the query guaranteed by Lemma 4.2.19 consists of exponentially many rules, simply combining these statements results in a triply exponential upper bound. To achieve a doubly exponential upper bound, we will instead use the following refined statement of the result by Bourhis et al.

Theorem 4.2.20 [BKR15a, Theorem 7]**.** *The containment problem* Cont(DL, FGDL) *is decidable in time*

- *doubly exponential in the number of variables in P and P';*
- *doubly exponential in the maximal size of a rule of P and P'; and*
- *singly exponential in the number of rules of P and P'*

given queries $Q = (P, \text{Out}) \in \mathit{DL}$ and $Q' = (P', \text{Out}) \in \mathit{FGDL}$.

We will prove in Section 4.3 that the procedure of Bourhis et al. [BKR15a] indeed yields Theorem 4.2.20.

We are now ready to prove that the decision problems PC(FGDL, Ind-Constraints) and PC(MDL, Ind-Constraints) are 2ExpTime-complete.

Proof of Proposition 4.2.16. We first prove that the parallel-correctness problems for the classes MDL and FGDL, and Ind-Constraints are in 2ExpTime.

Upper Bound. We first provide the proof for frontier-guarded Datalog queries. Let Q be a frontier-guarded Datalog query, Z be a primitive hash policy scheme, and Σ a set data-moving distribution constraints.

Let Q' be the frontier-guarded Datalog query guaranteed by Lemma 4.2.19. We claim that Q is parallel-correct w.r.t. $\mathcal{F}(Z, \Sigma)$ if and only if $Q \sqsubseteq Q'$ holds. Suppose $Q \sqsubseteq Q'$ holds and let G be a global database. Thanks to Lemma 4.2.19 there is a policy pair (δ, γ) such that δ scatters G and $[Q, \gamma](\delta(G)) = Q'(G)$. By assumption we have $Q(G) \subseteq Q'(G)$, and, thus, $Q(G) \subseteq [Q, \gamma](\delta(G))$. Thanks to Lemma 4.2.6 we can conclude that Q is parallel-correct w.r.t. $\mathcal{F}(Z, \Sigma)$.

Conversely, suppose Q is parallel-correct w.r.t. $\mathcal{F}(Z, \Sigma)$. Let G be an arbitrary database and (δ, γ) be the policy pair satisfying $[Q, \gamma](\delta(G)) = Q'(G)$ due to Lemma 4.2.19. Since Q is parallel-correct we have $Q(G) = [Q, \gamma](\delta(G))$. Together we have, in particular, $Q(G) \subseteq Q'(G)$. We can conclude that $Q \sqsubseteq Q'$ holds.

All in all, parallel-correctness of Q can thus be decided by testing $Q \sqsubseteq Q'$. Since the number of rules of Q' is at most exponential and the number of variables and the maximal length of rules is polynomial, $Q \sqsubseteq Q'$ can be tested in doubly exponential time in the size of Q, Z, and Σ thanks to Lemma 4.2.19 and Theorem 4.2.20.

The proof for monadic Datalog queries is exactly the same. For the containment test (over global databases), recall that every monadic Datalog queries can be transformed into an equivalent frontier-guarded Datalog queries in polynomial time, cf. Lemma 2.4.9.

Lower Bound. We prove the 2ExpTime-hardness results by polynomial time reductions from the containment problem monadic Datalog and frontier-guarded Datalog, respectively. The containment problem Cont(MDL, MDL) is 2ExpTime-hard [BBS12, Theorem 2], and since monadic Datalog queries can be transformed into equivalent frontier-guarded Datalog queries in polynomial time [BKR15a], the same is true for Cont(FGDL, FGDL). The reductions are essentially the same for both cases. Again, we focus on frontier-guarded Datalog queries first.

Let $Q_1 = (P_1, \mathrm{Out})$ and $Q_2 = (P_2, \mathrm{Out})$ be two frontier-guarded Datalog queries. We construct a frontier-guarded Datalog query Q, a primitive hash policy scheme Z, and a set Σ of data-moving distribution constraints such that $Q_1 \sqsubseteq Q_2$ holds if and only if Q is parallel-correct w.r.t. $\mathcal{F}(Z, \Sigma)$.

Without loss of generality, we assume that $\mathsf{idb}(P_1) \cap \mathsf{idb}(P_2) = \{\mathrm{Out}\}$; that is, the only intensional symbol P_1 and P_2 have in common is Out.

The frontier-guarded Datalog query Q is obtained by combining Q_1 and Q_2 as follows. Let P_1' and P_2' be the Datalog programs obtained by replacing Out with Out_1 and Out_2 in P_1 and P_2, respectively. The output symbol of Q is Out, which does not occur in P_1' or P_2'; and the Datalog program is

$$P = P_1' \cup P_2' \cup \big\{\mathrm{Out}(\bar{x}) \leftarrow \mathrm{Out}_1(\bar{x}), E(\bar{x})\big\} \cup \big\{\mathrm{Out}(\bar{x}) \leftarrow \mathrm{Out}_2(\bar{x}), E(\bar{x})\big\},$$

where E is a fresh extensional symbol not appearing in $\mathsf{edb}(P_1') \cup \mathsf{edb}(P_2')$. Clearly, Q is frontier-guarded if Q_1 and Q_2 are.

Over a global database G, the query result of Q is the union of $Q_1(G)$ and $Q_2(G)$ intersected with the relation E.

The hash policy scheme Z for facts over $\mathsf{edb}(P)$ consists of a hash directive $(R, 1, ())$ for each $R \in \mathsf{edb}(P) \setminus \{E\}$ and an additional hash directive $(E, 2, ()))$. Finally, the set Σ of distribution constraints consists of exactly one distribution constraint, namely

$$\mathrm{Out}_2(\bar{x})@\kappa, E(\bar{x})@\lambda \to \mathrm{Out}_2(\bar{x})@\lambda.$$

Let G be a global database and $(\delta, \gamma) \in \mathcal{F}(Z, \Sigma)$ such that δ scatters G. Then, considering the distributed database $\delta(G)$, there are two distinct servers k and ℓ such that all facts over $\mathsf{edb}(P) \setminus \{E\}$ reside on k and all E-facts reside on server ℓ. Notably, there is *no* local database of $\delta(G)$ that contains E-facts and facts over $\mathsf{edb}(P) \setminus \{E\}$.

Therefore, the outputs of P_1' and P_2' can be computed on server k because neither program refers to E. However, no Out-facts can be derived on k since no E-fact resides on k. Due to the communication policy γ all Out_2-facts computed on k are sent to ℓ. Hence, the intersection of Out_2 with E can be computed on ℓ thanks to the rule $\mathrm{Out}(\bar{x}) \leftarrow \mathrm{Out}_2(\bar{x}), E(\bar{x})$. On the other hand, the rule $\mathrm{Out}(\bar{x}) \leftarrow \mathrm{Out}_1(\bar{x}), E(\bar{x})$ is *never* used to derive a fact in the distributed evaluation, because Out_1-facts cannot be communicated. Thus, the distributed evaluation yields the intersection of $Q_2(G)$ with the relation E.

It remains to argue that $Q_1 \sqsubseteq Q_2$ if and only if Q is parallel-correct w.r.t. $\mathcal{F}(Z, \Sigma)$. By Lemma 4.2.6 it suffices to consider global databases G and distribution policies δ such that δ scatters G. As argued before $[Q, \gamma](\delta(G))$ is the intersection of $Q_2(G)$ with the

relation E and $Q(G)$ is the intersection of $Q_1(G) \cup Q_2(G)$ with the relation E for such databases and distribution policies. Thus, if $Q_1 \sqsubseteq Q_2$ does *not* hold, there is a G such that $[Q, \gamma](\delta(G)) \subsetneq Q(G)$ holds for any policy pair (δ, γ) such that δ scatters G. Since Lemma 4.2.7 guarantees that such a policy pair exists, we can conclude that Q is *not* parallel-correct w.r.t. $\mathcal{F}(Z, \Sigma)$.

For the converse suppose $Q_1 \sqsubseteq Q_2$ holds. Let G be a global database and δ be the distribution policy guaranteed by Lemma 4.2.7 that scatters G, and γ be the communication policy such that $(\delta, \gamma) \in \mathcal{F}(Z, \Sigma)$. By assumption we have that $Q_1(G) \subseteq Q_2(G)$ and, hence, $Q_2(G) = Q_1(G) \cup Q_2(G)$. Therefore, $[Q, \gamma](\delta(G)) = Q(G)$. Then Lemma 4.2.6 allows us to conclude that Q is parallel-correct w.r.t. $\mathcal{F}(Z, \Sigma)$. □

In principle, Proposition 4.2.16 is arguably good news, because it identifies classes of Datalog queries and a class of policy pair families for which the parallel-correctness problem is decidable. On the other hand, the families $\mathcal{F}(Z, \Sigma)$ in Ind-Constraints are not very interesting because, intuitively, the underlying networks do not "scale" with the size of the database. Indeed, the proof of Lemma 4.2.19 entails that there is a distribution policy δ over a network $\mathcal{N} = [1, m]$ that scatters every global database. The number m, and hence the size of $\mathcal{N}$, does only depend on Z. Hence, there is at least one local database of $\delta(G)$ that contains $\Omega(|G|)$ many facts for any global database G.

The same is true for all distributed databases that comply with a distribution policy $\delta_{Z,H}$, since according to Lemma 4.2.10 such distributed databases cover $\delta(G)$. Moreover, m servers suffice to compute the query result, if the query is parallel-correct. Any other server is redundant. This contrasts the idea of distributing facts among a massive amount of servers.

Therefore, we will study restrictions of distribution constraints in the following subsections, leaving the hash policy schemes unrestricted. Since the distribution policies are not restricted, the underlying networks may then scale meaningfully. We conclude this subsection with the observation that the set of distribution constraints constructed in the proof for the lower bound of Proposition 4.2.16 has a very simple structure. The 2ExpTime-hardness of the parallel-correctness problem will thus carry over.

4.2.3 The Polynomial Communication Property

In this subsection we introduce and study the *polynomial communication property*. We show that, for subclasses of frontier-guarded Datalog queries and distribution constraints having this property, distributed evaluations over scattered databases can be simulated by Datalog programs – akin to Lemma 4.2.19. As before, in Section 4.2.2, such a simulation effectively yields a reduction from the parallel-correctness problem to the containment problem. We note that our construction fails for monadic Datalog queries – we illustrate this by means of an example at the end of this section.

In Section 4.2.4 we will present a syntactical restriction of data-moving distribution constraints which guarantees the polynomial communication property, even in combination with the class DL of all Datalog queries. We will prove that, in combination with the class FGDL of frontier-guarded Datalog queries, this results in a 2ExpTime-complete

parallel-correctness problem. Alternatively, the polynomial communication property can also be asserted semantically, yielding the same outcome for frontier-guarded Datalog queries. This will be the topic of Section 4.2.5.

The idea behind the polynomial communication property is to bound the number of rounds it takes for an (already computed) intensional fact to be communicated to every server that "requires" it. We make this precise with the help of proof trees.

Let P be a Datalog program, $\mathcal{N}$ be a network, and Σ be set of data-moving distribution constraints. A *subtree T' of a proof tree T* over $\mathcal{N}$ with respect to P and Σ is a partial proof tree that consists of a connected subset of $\mathsf{nodes}(T)$, and all edges from T between these nodes. Furthermore, the labels and witnesses are inherited from T; that is, every node v of T' has the same label in T' as in T and every inner node of T' is witnessed by the same rule or constraint as in T. Notably this implies that, for every node $v \in \mathsf{nodes}(T')$, the subtree T' contains either all children of v in T, or none. We call a (subtree of a) proof tree over $\mathcal{N}$ for a fact $R(\bar{a})@k$ with respect to P and Σ *computation-free*, if all its inner nodes are witnessed by distribution constraints from Σ. The *size* of a (partial) proof tree T is the number of its nodes $|\mathsf{nodes}(T)|$.

Definition 4.2.21 (Polynomial Communication Property)**.** A class $\mathbb{Q}$ of Datalog queries and a class $\mathbb{C}$ of sets of data-moving distribution constraints have the *polynomial communication property* if there is a polynomial p such that the following holds, for each query $Q = (P, \mathrm{Out}) \in \mathbb{Q}$, each network $\mathcal{N}$, each finite set $\Sigma \in \mathbb{C}$, and each distributed database $\mathcal{D}$ over $\mathcal{N}$: If a fact has a proof tree over $\mathcal{N}$ with respect to P, Σ, and $\mathcal{D}$, then there is such a proof tree whose computation-free subtrees have size at most $p(\|Q\|, \|\Sigma\|)$.

The main contribution of this section is the result that a distributed evaluation over a scattered database can be simulated by a frontier-guarded Datalog query, if the polynomial communication property applies. That is, we want to prove the following result.

Lemma 4.2.22. *Let $\mathbb{Q}$ be a class of Datalog queries and $\mathbb{C}$ be a class of sets of data-moving distribution constraints that have the polynomial communication property. For every frontier-guarded Datalog query $Q \in \mathbb{Q}$, hash policy scheme Z, and set $\Sigma \in \mathbb{C}$ of data-moving distribution constraints, a frontier-guarded Datalog query Q' can be constructed in exponential time such that the following holds: For every global database G there is a policy pair $(\delta, \gamma) \in \mathcal{F}(Z, \Sigma)$ such that δ scatters G and $[Q, \gamma](\delta(G)) = Q'(G)$.*

The number of variables and the length of rules of Q' is polynomial in $\|Q\|$, $\|Z\|$, and $\|\Sigma\|$; and the number of rules is at most exponential.

Our construction for Lemma 4.2.22 is similar to the one employed for Lemma 4.2.19. Recall that the Datalog program constructed for Lemma 4.2.19 has, for every symbol R occurring in the original query, a relation symbol R^i with the same arity as R. The intended meaning is that a fact $R^i(\bar{a})$ is derived in the simulation, if the fact $R(\bar{a})$ can be derived on server i in the distributed evaluation. Here this no longer suffices, simply because the number of servers cannot be bounded by the size of Q, Z, and Σ if Z does not induce (only) value-independent distribution policies.

Therefore, we use relation symbols R^i with (strictly) higher arity than R instead. A fact $R^i(\bar{c}, \bar{a})$ then signifies that the fact $R(\bar{a})$ resides on the server identified with $(i, \bar{c})$.

Consequently, the Datalog query uses intensional atoms of the form $R^i(\bar{z}, \bar{x})$. Obtaining frontier-guarded rules for the simulation of constraints is then more involved.

In fact, instead of constructing dedicated rules for constraints, we will incorporate constraints into rules for the Datalog programs. To this end, we define the *composition of two data-moving distribution constraints σ and σ'*. We assume without loss of generality that σ and σ' have disjoint sets of variables. Otherwise, the variables in σ' can just be renamed. We say that σ and σ' are *unifiable* if there is an atom $R(\bar{x})@\kappa \in \mathsf{body}(\sigma)$ and $\mathsf{head}(\sigma') = R(\bar{y})@\lambda$.

Let $\alpha\colon \mathsf{vars} \cup \mathsf{svars} \to \mathsf{vars} \cup \mathsf{svars}$ be a mapping that maps variables in $\bar{x}, \bar{y}$ to variables in $\bar{x}$, λ to κ, and any other variable to itself, such that $\alpha(R(\bar{y})@\lambda) = \alpha(R(\bar{x})@\kappa)$ holds.[5] Then σ and σ' can be composed into a new data-moving distribution constraint $\sigma \circ_\alpha \sigma'$ with head $\alpha(\mathsf{head}(\sigma))$ and body $\alpha(\mathsf{body}(\sigma) \setminus \{R(\bar{x})@\kappa\}) \cup \alpha(\mathsf{body}(\sigma'))$. In other words, $\sigma \circ_\alpha \sigma'$ has the head of σ and its body consists of all atoms in the bodies of σ and σ' besides $R(\bar{x})@\kappa$, after application of α. Let us point out that $\sigma \circ_\alpha \sigma'$ is indeed data-moving: Let A be the atom occurring in $\mathsf{head}(\sigma)$. If $A \neq R(\bar{x})$ then A occurs in $\mathsf{body}(\sigma) \setminus \{R(\bar{x})@\kappa\}$ and, consequently, $\alpha(A)$ occurs in the body of $\sigma \circ_\alpha \sigma'$. If, on the other hand, $A = R(\bar{x})$ then $\alpha(A)$ occurs in $\mathsf{body}(\sigma')$ because, since σ' is data-moving, there has to be an atom $R(\bar{y})@\mu \in \mathsf{body}(\sigma')$.

Example 4.2.23. Consider the two constraints σ and σ' defined by the rules

$$\sigma\colon E(x_1, x_3)@\kappa, R(x_2, x_3, x_3)@\kappa, S(x_1, x_2)@\mu \to S(x_1, x_2)@\kappa \qquad \text{and}$$
$$\sigma'\colon E(y_2, y_4)@\lambda, R(y_1, y_2, y_3)@\mu' \to R(y_1, y_2, y_3)@\lambda.$$

The mapping α with $\alpha(y_1) = x_2, \alpha(y_2) = x_3, \alpha(y_3) = x_3$, $\alpha(\lambda) = \kappa$, and $\alpha(v) = v$ for any other variable v yields the composed constraint

$$E(x_1, x_3)@\kappa, S(x_1, x_2)@\mu, E(x_3, y_4)@\kappa, R(x_2, x_3, x_3)@\mu' \to S(x_1, x_2)@\kappa. \qquad \triangleleft$$

For a set Σ of data-moving distribution constraints, we write Σ^* for the (possibly infinite) set of data-moving distribution constraints that can be obtained by iteratively composing unifiable constraints from Σ. The possibly infinite size of Σ^* is not problematic for our application as we will only consider constraints in Σ^* of size polynomial in Σ of which there are at most exponentially many.

The next result states that computation-free proof trees can be "condensed" into very simple proof trees with only one inner node – which is then necessarily the root node.

Lemma 4.2.24. *Let P be a Datalog program, Σ be a finite set of data-moving distribution constraints, and $\mathcal{N}$ be a network. A fact $R(\bar{a})@k$ has a (partial) computation-free proof tree over $\mathcal{N}$ with respect to P and Σ if and only if it has a (partial) computation-free proof tree over $\mathcal{N}$ with respect to P and Σ^* that has the same leaf nodes (including the same labels) and only one inner node.*

[5] The mapping α is also called a *unifier*; if σ and σ' are unifiable a unifier always exists since constraints do not contain any constants.

Proof. For the direction from left to right, we show that, for every partial computation-free proof tree T with respect to Σ^* with two or more inner nodes, there is a partial computation-free proof tree T^* with fewer inner nodes. Since $\Sigma \subseteq \Sigma^*$ this implies the claim.

To this end, let v be an inner node and v' be a child of v that is an inner node as well. We prove that v and v' can be merged into a new node v^* resulting in the desired partial proof tree T^*. The new node v^* has the same label as v and its children are all children from v and v' except for v'. More precisely, $\mathsf{children}_{T^*}(v^*) = (\mathsf{children}_T(v) \setminus \{v'\}) \cup \mathsf{children}_T(v')$.

We have to argue that the proof tree property of v^* is witnessed by a constraint from Σ^*. For this purpose, let σ and σ' be the distribution constraints, and ϑ and ϑ' be the network aware valuations which are the witnesses for v and v', respectively. We can assume that σ and σ' have no variable in common, since otherwise, we could rename all variables in σ' (and adapt ϑ' accordingly).

Since v' is a child of v, there is a distributed atom $S(\bar{x})@\kappa \in \mathsf{body}(\sigma)$ such that v' is labelled with the fact $\vartheta(S(\bar{x})@\kappa)$ and the head of σ is $S(\bar{y})@\lambda$ for some tuple $\bar{y}$ of variables. Thus, σ and σ' are unifiable. Furthermore, we have $\vartheta'(S(\bar{y})@\lambda) = \vartheta(S(\bar{x})@\kappa)$. Hence, there is a mapping α that maps variables in $\bar{x}, \bar{y}$ to variables in $\bar{x}$, λ to κ, and any other variable to itself, such that $\alpha(S(\bar{y})@\lambda) = \alpha(S(\bar{x})@\kappa)$, $\vartheta(\alpha(\bar{x})) = \vartheta(\bar{x})$, and $\vartheta(\alpha(\bar{y})) = \vartheta'(\bar{y})$ hold.

Let ϑ^* be the network aware valuation defined as ϑ on variables in σ and as ϑ' on variables in σ'. Then we have

$$\vartheta^*(\alpha(\mathsf{head}(\sigma))) = \vartheta(\mathsf{head}(\sigma)),\ \vartheta^*(\alpha(\mathsf{body}(\sigma))) = \vartheta(\mathsf{body}(\sigma)),$$
$$\text{and } \vartheta^*(\alpha(\mathsf{body}(\sigma'))) = \vartheta'(\mathsf{body}(\sigma')).$$

We can conclude that $\sigma \circ_\alpha \sigma'$ and ϑ^* witness the proof tree property of v^*.

For the direction from right to left, consider a node v^* of a partial proof tree with respect to Σ^* whose proof tree property is witnessed by a constraint $\sigma^* \in \Sigma^* \setminus \Sigma$ and a network aware valuation ϑ^*. Then $\sigma^* = \sigma \circ_\alpha \sigma'$ for some mapping α and $\sigma, \sigma' \in \Sigma^*$. More precisely, we have that $\alpha(\mathsf{head}(\sigma)) = \mathsf{head}(\sigma^*)$, the body of σ^* is $\alpha(\mathsf{body}(\sigma) \setminus \{S(\bar{x})@\kappa\}) \cup \alpha(\mathsf{body}(\sigma'))$, $\mathsf{head}(\sigma') = S(\bar{y})@\lambda$, and $\alpha(S(\bar{x})@\kappa) = \alpha(S(\bar{y})@\lambda)$ for some relation symbol S, server variables κ, λ, and tuples $\bar{x}, \bar{y}$.

Let $\vartheta = \vartheta^* \circ \alpha$. The node v^* can then be replaced by a new node v that has the same label as v^*, i.e. $\vartheta(\mathsf{head}(\sigma))$, all children of v^* with labels from $\vartheta(\mathsf{body}(\sigma) \setminus \{S(\bar{x})@\kappa\})$, and a new child v' labelled with $\vartheta(S(\bar{x})@\kappa)$. The children of v' are all children of v^* with labels in $\vartheta(\mathsf{body}(\sigma'))$. The leaves of the subtree with inner nodes v, v' are then exactly the children of v^*. Furthermore, the proof tree property for v is witnessed by σ and ϑ, and the proof tree property for v' by σ' and ϑ.

Since every constraint $\sigma^* \in \Sigma^*$ is composed of finitely many constraints from Σ, applying this procedure finitely often yields the desired (partial) computation-free proof tree with respect to Σ. □

Note that Lemma 4.2.24 implies that, for a (partial) computation-free proof tree with respect to Σ of polynomial size, the constraint from Σ^* witnessing the proof tree property

for the only inner node in the corresponding proof tree with respect to Σ^* has polynomial size. This is because every node in the latter tree is also present in the former.

Equipped with Lemma 4.2.24 we can now prove Lemma 4.2.22.

Proof of Lemma 4.2.22. Let $Q = (P, \text{Out})$ be a Datalog query and Σ be a set of data-moving distribution constraints that belong to classes having the polynomial communication property, and Z be a hash policy scheme. We construct a Datalog query $Q' = (P', \text{Out})$ that simulates the distributed evaluation of Q over scattered databases in three steps. Similarly to Lemma 4.2.19, we first construct an intermediate Datalog program P''' whose rules are not necessarily frontier-guarded. However, P''' does not have any rules for the constraints in Σ. In the second step constraints from Σ^* are incorporated into the rules of P''', resulting in a Datalog program P'' with the desired properties – except that its rules are not necessarily frontier-guarded. The third step then replaces the rules in P'' with equivalent frontier-guarded rules.

Construction. The Datalog program P''' uses one intensional relation symbol R^i of arity $|\bar{u}| + \mathsf{ar}(R)$, for every relation symbol $R \in \mathsf{edb}(P) \cup \mathsf{idb}(P)$ and every hash directive $(E, i, \bar{u})$ – with possibly $R \neq E$ – from Z. It consists of the following rules for Z, P, and Out, respectively.[6]

- For every $(E, i, \bar{u}) \in Z$, P''' has a rule $E^i(\bar{z}, \bar{x}) \leftarrow E(\bar{x})$, where $\bar{x}$ is a tuple of pairwise different variables, and $\bar{z} = \bar{x}[\bar{u}]$.
- For every rule $R(\bar{x}) \leftarrow S_1(\bar{y}_1), \ldots, S_n(\bar{y}_n)$ of P and triple $(E, i, \bar{u})$, P''' has a rule

$$R^i(\bar{z}, \bar{x}) \leftarrow S_1^i(\bar{z}, \bar{y}_1), \ldots, S_n^i(\bar{z}, \bar{y}_n),$$

  where $\bar{z}$ is a tuple of pairwise different variables not occurring in any of the $\bar{y}_i$ or $\bar{x}$.
- For every hash directive $(E, i, \bar{u})$, P''' has a rule $\text{Out}(\bar{x}) \leftarrow \text{Out}^i(\bar{z}, \bar{x})$ where $\bar{z}$ and $\bar{x}$ are tuples of pairwise different variables that share no variable, and $|\bar{z}| = |\bar{u}|$.

The rules for Z basically define a scattered instance of a global database G with respect to Z, where a fact $R^i(\bar{c}, \bar{a})$ corresponds to $R(\bar{a})$ being at server $(i, \bar{c})$. The rules for P mimic the local evaluation of P at each such server. The rules for Out allow for the derivation of output facts.

Let p be a polynomial that bounds the size of computation-free proof trees with respect to $\|P\|$ and $\|\Sigma\|$. The program P'' results from P''' by replacing, in all possible ways, some intensional atoms in bodies of rules with (translated) bodies of constraints from Σ^* of size at most $p(\|P\|, \|\Sigma\|)$. The communication of facts in the distributed evaluation of P, which can be witnessed by a single constraint from Σ^* thanks to Lemma 4.2.24, is thus incorporated into P''.

Replacing intensional atoms in a rule τ of P''' is done similarly to composing two distribution constraints. The main difference is that the server variables occurring in

[6] As in the construction for Lemma 4.2.19 all symbols R^i for which *no* rule is constructed are removed, along with all rules where R^i occurs in the body, and recursively.

the constraints have to be substituted with tuples $\bar{z}$ of data variables and a number i. In the following, we assume that all rules and distribution constraints have pairwise different variables; that is, the variables in a rule (or distribution constraint) do *not* occur in any other rule or distribution constraint. Let $S_1^i(\bar{z}, \bar{y}_1), \ldots, S_m^i(\bar{z}, \bar{y}_m) \in \mathsf{body}(\tau)$ be intensional atoms of τ that are to be replaced, and $\sigma_1, \ldots, \sigma_m \in \Sigma^*$ be distribution constraints with head atoms $S_1(\bar{x}_1)@\kappa_1, \ldots, S_m(\bar{x}_m)@\kappa_m$. Further, let $\alpha\colon \mathsf{var} \to \mathsf{var}$ be a mapping that maps variables in any of the tuples $\bar{x}_j$ and $\bar{y}_j$ to variables in $\bar{x}_1, \ldots, \bar{x}_m$ such that $\alpha(\bar{y}_j) = \alpha(\bar{x}_j)$ holds for all $j \in [1, m]$. On all other variables α is the identity. Additionally, let $s\colon \mathsf{svar} \to \{k \mid (E, k, \bar{u}) \in Z\}$. Then each atom $S_j^i(\bar{z}, \bar{y}_j)$ can be replaced as follows.

- For every distributed atom $S(\bar{y}')@\kappa_j$ in $\mathsf{body}(\sigma_j)$ the atom $S^i(\bar{z}, \alpha(\bar{y}'))$ is added to $\mathsf{body}(\tau)$.
- For every other server variable λ of σ_j, a fresh variable tuple $\bar{z}_\lambda$ is used and for every distributed atom $F(\bar{y}')@\lambda$ in $\mathsf{body}(\sigma_j)$ the atom $F^{s(\lambda)}(\bar{z}_\lambda, \alpha(\bar{y}'))$ is added to $\mathsf{body}(\tau)$.

Finally, every original atom $E^i(\bar{z}, \bar{y})$ in $\mathsf{body}(\tau)$ that has not been replaced by a constraint, is replaced with $E^i(\bar{z}, \alpha(\bar{y}))$ and the head $R^i(\bar{z}, \bar{x})$ of τ is replaced with $R^i(\bar{z}, \alpha(\bar{x}))$.

Since the length of the resulting rules is bounded by $\mathcal{O}\big((\|P\| + \|Z\|) \cdot p(\|P\|, \|\Sigma\|)\big)$, and the domain of s can be restricted to server variables occurring in Σ, the overall number of resulting rules is at most exponential in $\|P\|$ and $\|\Sigma\|$.

Correctness. Before we show how the rules of P'' can be guarded, we prove the correctness of our construction. That is, we prove that, for every global database G, there is a policy pair $(\delta, \gamma) \in \mathcal{F}(Z, \Sigma)$ such that δ scatters G and $[Q, \gamma](\delta(G)) = Q''(G)$ holds for $Q'' = (P'', \mathrm{Out})$. To this end, let G be a global database and $\mathcal{N}$ be the network with servers $(i, \bar{c})$ for every $(E, i, \bar{u})$ and $\bar{c} \in \mathsf{adom}(G)^{|\bar{u}|}$. Clearly, the tuple H consisting of hash functions h_i of arity $|\bar{u}|$ defined by $h_i(\bar{c}) = (i, \bar{c})$ for each $(E, i, \bar{u}) \in Z$ scatters G and Z is compatible with H. The communication policy is simply $\gamma = \gamma_{\Sigma,\mathcal{N}}$.

It suffices to show that there is a proof tree over $\mathcal{N}$ with respect to P, Σ, and $\delta(G)$ for a fact $R(\bar{a})@(i, \bar{c})$ if and only if there is a proof tree for $R^i(\bar{c}, \bar{a})$ with respect to P'' and G. In fact, it even suffices to consider proof trees whose root node is witnessed by a Datalog rule (or is a leaf). Moreover, thanks to Lemma 4.2.24, we can consider proof trees for $R(\bar{a})@(i, \bar{c})$ with respect Σ^* instead of Σ whose computation-free subtrees have only a single inner node. Both directions can be shown by induction over the structure of a proof tree.

We prove the direction from left to right by induction over the structure of a proof tree over $\mathcal{N}$ for a fact $R(\bar{a})@(i, \bar{c})$ with respect to P and Σ^*. Thanks to Lemma 4.2.24 we can assume that all computation-free subtrees have only a single inner node. If the tree consists of a single leaf node labelled $R(\bar{a})@(i, \bar{c})$, then there is a hash directive $(R, i, \bar{u}) \in Z$ such that $\bar{a}[\bar{u}] = \bar{c}$. By construction P'' has a rule $R^i(\bar{z}, \bar{x}) \leftarrow R(\bar{x})$ with $\bar{x}[\bar{u}] = \bar{z}$ and all variables in $\bar{x}$ are pairwise different. Thus, there is a valuation ϑ with $\vartheta(\bar{x}) = \bar{a}$ and $\vartheta(\bar{z}) = \bar{c}$. The tree whose root is labelled $R^i(\bar{c}, \bar{a})$ and which has a single child node labelled $R(\bar{a})$ is then a proof tree for $R^i(\bar{c}, \bar{a})$ with respect to P'' and G.

For the induction step it suffices, as mentioned above, to consider nodes v whose proof tree property is witnessed by a Datalog rule $\tau = R(\bar{x}) \leftarrow S_1(\bar{y}_1), \ldots, S_n(\bar{y}_n)$ and a valuation ϑ. Let $v_1, \ldots, v_n$ be the children of v and $\vartheta(S_1(\bar{y}_1))@(i, \bar{c}), \ldots, \vartheta(S_n(\bar{y}_n))@(i, \bar{c})$ be their labels, respectively. We can assume that $v_1, \ldots, v_m$, for some $m \leq n$, are witnessed by constraints, and $v_{m+1}, \ldots, v_n$ by some Datalog rules. Let $\sigma_1, \ldots, \sigma_m \in \Sigma^*$ be the constraints and $\vartheta_1, \ldots, \vartheta_m$ be the network aware valuations witnessing the proof tree property of the nodes $v_1, \ldots, v_m$. Further, let $S_1(\bar{x}_1)@\kappa_1, \ldots, S_m(\bar{x}_m)@\kappa_m$ be the head atoms of $\sigma_1, \ldots, \sigma_m$. Note that $\vartheta_j(S_j(\bar{x}_j)@\kappa_1) = \vartheta(S_j(\bar{y}_j))@(i, \bar{c})$ holds for all $j \in [1, m]$. Let ϑ^* be the network aware valuation that maps every variable that agrees with ϑ on every variable occurring in the Datalog rule τ and with ϑ_j on every variable occurring in σ_j, for all $j \in [1, m]$.[7] Then there is a variable mapping α such that $\alpha(\bar{y}_j) = \alpha(\bar{x}_j)$ and the image under ϑ^* is preserved. That is, ϑ^* has the following properties.

(a) $\vartheta^*(\alpha(\mathsf{head}(\tau))) = \vartheta(\mathsf{head}(\tau))$,

(b) $\vartheta^*(\alpha(S_j(\bar{y}_j))) = \vartheta(S_j(\bar{y}_j))$ for all $j \in [1, n]$, and

(c) $\vartheta^*(\alpha(\mathsf{body}(\sigma_j))) = \vartheta_j(\mathsf{body}(\sigma_j))$ for all $j \in [1, m]$.

Further, let s be the mapping with $s(\lambda) = k$ if $\vartheta^*(\lambda) = (k, \bar{c})$ for some $\bar{c}$.

Let now τ^* be the rule obtained from $R^i(\bar{z}, \bar{x}) \leftarrow S_1^i(\bar{z}, \bar{y}_1), \ldots, S_n^i(\bar{z}, \bar{y}_n)$ by replacing the intensional atoms $S_1^i(\bar{z}, \bar{y}_1), \ldots, S_m^i(\bar{z}, \bar{y}_m)$ with the bodies of $\sigma_1, \ldots, \sigma_m$ using α and s.

We construct a proof tree T^* for $R^i(\bar{c}, \bar{a})$ whose root node v^* is witnessed by τ^* and an extension of ϑ^*. First, we observe that the head of τ^* is $R^i(\bar{z}, \alpha(\bar{x}))$ and that, by construction, $\bar{z}$ shares no variable with τ or any of the constraints σ_j. Thus, we can extend ϑ^* such that we have $\vartheta^*(\bar{z}) = \bar{c}$. Thanks to Property (a) and $\vartheta(\bar{x}) = \bar{a}$ we then have $\vartheta^*(R^i(\bar{z}, \alpha(\bar{x}))) = R^i(\bar{c}, \bar{a})$, as required.

In the following we extend T^* and ϑ^* such that

$$\{\mathsf{fact}(w) \mid w \in \mathsf{children}_{T^*}(v^*)\} = \vartheta^*(\mathsf{body}(\tau^*))$$

holds. Altogether, we can then conclude that T^* is a proof tree for $R^i(\bar{c}, \bar{a})$ with respect to P'' and G.

- Consider an atom $S_j^i(\bar{z}, \alpha(\bar{y}_j))) \in \mathsf{body}(\tau^*)$ for $j > m$ that originates from an atom $S_j(\bar{y}_j)$ of the original Datalog rule τ from P. By assumption, there is a proof tree for $\vartheta(S_j(\bar{y}_j))@(i, \bar{c})$ with respect to P and Σ^* whose root node is witnessed by a Datalog rule.

  Thanks to the induction hypothesis, there is then also a proof tree T_j for $S_j^i(\bar{c}, \vartheta(\bar{y}_j))$. Since we already extended ϑ^* such that $\vartheta^*(\bar{z}) = \bar{c}$ holds, and thanks to Property (b), we have $\vartheta^*(S_j^i(\bar{z}, \alpha(\bar{y}_j))) = S_j^i(\bar{c}, \vartheta(\bar{y}_j))$. Thus, appending the root of T_j to v^* yields, in particular, a child node of v^* for $S_j^i(\bar{z}, \alpha(\bar{y}_j))$.

[7] As above we assume here w.l.o.g. that the sets of variables occurring in the constraints and the rule are pairwise disjoint; and ϑ^* is thus well-defined. This can always be achieved by renaming variables.

- Any other atom in $\mathsf{body}(\tau^*)$ is of the from $F^{s(\lambda)}(\bar{z}_\lambda, \alpha(\bar{y}'))$ and originates from a distributed atom $F(\bar{y}')@\lambda$ in the body of some constraint σ_j. By definition of s, we have that $\vartheta^*(\lambda) = (s(\lambda), \bar{c}_\lambda)$ for some $\bar{c}_\lambda$. Thanks to Property (c), we also have $\vartheta^*(F(\alpha(\bar{y}'))@\lambda) = \vartheta_j(F(\bar{y}')@\lambda)$. Altogether, $\vartheta_j(F(\bar{y}')@\lambda) = \vartheta^*(F(\alpha(\bar{y}'))@(s(\lambda), \bar{c}_\lambda)$. Since the constraint σ_j and ϑ_j are witnesses for a node in T, there is a proof tree over $\mathcal{N}$ for $\vartheta^*(F(\alpha(\bar{y}'))@(s(\lambda), \bar{c}_\lambda))$ with respect to P and Σ^*. Further, since we can assume that all computation-free subtrees of T have at most one inner node thanks to Lemma 4.2.24, there is such a proof tree for $\vartheta^*(F(\alpha(\bar{y}'))@(s(\lambda), \bar{c}_\lambda))$ where the root node is witnessed by a Datalog rule.

  By induction hypothesis there is then a proof tree for $F^{s(\lambda)}(\bar{c}_\lambda, \vartheta^*(\alpha(\bar{y}')))$ with respect to P'' and G. We attach this proof tree to v^* and extend ϑ^* such that $\vartheta^*(\bar{z}_\lambda) = \bar{c}_\lambda$ holds. As in the former case, this then yields a child node of v^* for $F^{s(\lambda)}(\bar{z}_\lambda, \alpha(\bar{y}'))$. Lastly, let us note that ϑ^* is well-defined because the variables in $\bar{z}_\lambda$ do not occur outside of $\bar{z}_\lambda$ and $\bar{z}_\lambda$ occurs only in atoms derived from distributed atoms with the server variable λ.

The direction from right to left can be proved similarly by induction over a proof tree for a fact $R^i(\bar{c}, \bar{a})$ with respect to P'' and G.

In the base case, a proof tree for a fact $R^i(\bar{c}, \bar{a})$ with respect to P'' and G consists of exactly two nodes. The root is labelled with a fact $R^i(\bar{c}, \bar{a})$ and its only child is labelled with $R(\bar{a})$. Furthermore, the root node is witnessed by a rule of the form $R^i(\bar{z}, \bar{x}) \leftarrow R(\bar{x})$ constructed for some triple $(R, i, \bar{u}) \in Z$. Note that, by construction, $\bar{z} = \bar{x}[\bar{u}]$. Thus, we also have $\bar{c} = \bar{a}[\bar{u}]$. Since, $h_i(\bar{c}) = (i, \bar{c})$ by definition of h_i, we can conclude that $R(\bar{a})@(i, \bar{c}) \in \delta(G)$. The proof tree over $\mathcal{N}$ with a single node labelled $R(\bar{a})@(i, \bar{c})$ is thus a proof tree with respect to $\delta(G)$.

For the induction step consider a proof tree for a fact $R^i(\bar{c}, \bar{a})$ with respect to P'' and G. Let τ^* be rule and ϑ^* be the valuation which are witnesses for the root node. The rule τ^* originates from a rule $\tau' \colon R^i(\bar{z}, \bar{x}) \leftarrow S_1^i(\bar{z}, \bar{y}_1), \ldots, S_n^i(\bar{z}, \bar{y}_n)$ from P'''. Without loss of generality, we can assume that the atoms $S_1^i(\bar{z}, \bar{y}_1), \ldots, S_m^i(\bar{z}, \bar{y}_m)$ have been replaced with the (translated) bodies of constraints $\sigma_1, \ldots, \sigma_m \in \Sigma^*$ using mappings α and s for data variables and server variables, respectively. Further, let $\tau \colon R(\bar{x}) \leftarrow S_1(\bar{y}_1), \ldots, S_n(\bar{y}_n)$ be the rule of P from which τ' originates.

We show that there is a proof tree T over $\mathcal{N}$ for $R(\bar{a})@(i, \bar{c})$ with respect to P, Σ^*, and $\delta(G)$. More precisely, we show that the root node $v = \mathsf{root}(T)$ is witnessed by τ. Recall that every atom (including the head atom) of τ^* is of the form $F^j(\bar{z}', \alpha(\bar{y}'))$ where the variables in $\bar{y}'$ originate from the Datalog rule τ or from the body of some constraint σ_j while the variables from $\bar{z}'$ do *not*. Let ϑ be the valuation that is defined as $\vartheta^* \circ \alpha$ on variables from τ or from any constraint; and as ϑ^* on any other data variable. Then we have, in particular, $\vartheta(R(\bar{x})) = R(\bar{a})$ for the head $R(\bar{x})$ of τ, as required for the proof tree property of v. In the following we extend T to a proof tree over $\mathcal{N}$ with respect to $\delta(G)$.

- Consider an atom $S_j(\bar{y})$ in $\mathsf{body}(\tau)$ for some $j > m$, i.e. an atom that was not replaced by the (translated) body of a constraint. Then $S_j^i(\bar{z}, \alpha(\bar{y}_j))$ is in $\mathsf{body}(\tau^*)$. Let $\bar{c}$ and $\bar{a}$ be such that $S_j^i(\bar{c}, \bar{a}) = \vartheta^*(S_j^i(\bar{z}, \alpha(\bar{y}_j)))$. Then there is a proof tree for $S_j^i(\bar{c}, \bar{a})$

with respect to P'' and G since v^* must have a child labelled $S_j^i(\bar{c}, \bar{a})$. Thanks to the induction hypothesis there then is also a proof tree over $\mathcal{N}$ for $S_j(\bar{a})@(i, \bar{c})$ with respect to P, Σ^*, and $\delta(G)$.

Note that, by definition of ϑ, we also have $\vartheta(S_j(\bar{y})) = S_j(\bar{a})$. Thus, appending the root of the tree for $S_j(\bar{a})@(i, \bar{c})$ to v yields, in particular, a child node for $S_j(\bar{y}_j)$.

- For an atom $S_j(\bar{y}_j)$ in $\mathsf{body}(\tau)$ whose corresponding atom $S_j^i(\bar{z}, \bar{y}_j)$ in $\mathsf{body}(\tau')$ got replaced with the (translated) body of the constraint σ_j, a new node v_j with label $\vartheta(S_j(\bar{y}))@(i, \bar{c})$ is attached as a child node for $S_j(\bar{y}_j)$ to v.

  We next describe the tree below v_j. Let $F^k(\bar{z}_\lambda, \alpha(\bar{y}'))$ with $k = s(\lambda)$ be an atom in $\mathsf{body}(\tau^*)$ which originates from a distributed atom $F(\bar{y}')@\lambda$ in $\mathsf{body}(\sigma_j)$. Then there is a proof tree for $F^k(\bar{c}', \bar{a}') = \vartheta^*(F^k(\bar{z}', \alpha(\bar{y}')))$ with respect to P'' and G. By induction hypothesis, there is thus a proof tree over $\mathcal{N}$ for $F(\bar{a}')@(k, \bar{c}')$ with respect to P, Σ^*, and $\delta(G)$. We attach the root of this tree to v_j as child node for $F(\bar{y}')@\lambda$. Indeed, we already have $\vartheta(\bar{y}') = \bar{a}'$. Extending ϑ by setting $\vartheta(\lambda) = (k, \bar{c}')$ yields $\vartheta(F(\bar{y}')@\lambda) = F(\bar{a}')@(k, \bar{c}')$, as required. Note that extending ϑ in this fashion yields a well-defined network aware valuations ϑ_j since λ was consistently replaced with $s(\lambda) = k$ and a fresh tuple $\bar{z}_\lambda$ of variables.

  Finally, observe that we also have $\vartheta_j(\mathsf{head}(\sigma_j)) = \vartheta(S_j(\bar{y}))@(i, \bar{c})$ since σ_j is data-moving. That is, all data variables and the server variable of $\mathsf{head}(\sigma_j)$ occur in $\mathsf{body}(\sigma_j)$; and, hence, ϑ_j maps $\mathsf{head}(\sigma_j)$ properly. We can conclude that σ_j and ϑ_j are proper witnesses for v_j.

This concludes the correctness proof. It remains to show how the rules of P'' can be replaced by frontier-guarded rules, yielding the desired frontier-guarded Datalog query $Q' = (P', \mathrm{Out})$.

Guarding. The rules for Z of the form $E^i(\bar{z}, \bar{x}) \leftarrow E(\bar{x})$ are already frontier-guarded because E is extensional and, by construction, all variables in $\bar{z}$ occur in $\bar{x}$.

The second kind of rules can replaced with frontier-guarded rules as follows. Let τ^* be a rule in P'' obtained from a rule $R^i(\bar{z}, \bar{x}) \leftarrow S_1^i(\bar{z}, \bar{y}_1), \ldots, S_n^i(\bar{z}, \bar{y}_n)$ by replacing atoms with (translated) bodies of constrains using a variable mapping α. Pick j such that S_j is extensional. Such a j exists, because $R^i(\bar{z}, \bar{x}) \leftarrow S_1^i(\bar{z}, \bar{y}_1), \ldots, S_n^i(\bar{z}, \bar{y}_n)$ was itself obtained from a rule $R(\bar{x}) \leftarrow S_1(\bar{y}_1), \ldots, S_n(\bar{y}_n)$ in P; and all rules in P are frontier-guarded. Then $S_j^i(\bar{z}, \alpha(\bar{y}_j))$ is in $\mathsf{body}(\tau^*)$, since there is no constraint for any extensional S_j-atom. Further, the head of τ^* is $R^i(\bar{z}, \alpha(\bar{x}))$, and every variable in $\alpha(\bar{x})$ appears in $\alpha(\bar{y}_j)$. The latter holds because we know that every variable of $\bar{x}$ appears in $\bar{y}_j$ since $S_j(\bar{y}_j)$ is the guard atom of the original rule in P. Recall that, by construction, the variables in $\bar{z}$ do *not* occur elsewhere, i.e. they only occur as part of $\bar{z}$.

Then, for each hash directive $(S_j, i, \bar{u}) \in Z$, a frontier-guarded rule can be constructed by replacing every occurrence of $\bar{z}$ in τ^* with $\alpha(\bar{y}_j[\bar{u}])$, and adding the extensional atom $S_j(\alpha(\bar{y}_j))$ to $\mathsf{body}(\tau^*)$. Note that this corresponds to inlining the rule $S_j^i(\bar{z}', \bar{y}') \leftarrow S_j(\bar{y}')$ with $\bar{z}' = \bar{y}'[\bar{u}]$ constructed for $(S_j, i, \bar{u})$ into τ^*. In particular, the new head is then

$R^i(\alpha(\bar{y}_j[\bar{u}]), \alpha(\bar{x}))$. All its variables occur in the guard atom $S_j(\alpha(\bar{y}_j))$. Hence, the new rule is frontier-guarded. The rule τ^* is then replaced with all frontier-guarded rules obtained in this fashion.

A rule $\text{Out}(\bar{x}) \leftarrow \text{Out}^i(\bar{z}, \bar{x})$ can be replaced by frontier-guarded rules as in the proof for Lemma 4.2.19. That is, it is replaced with the set of rules $\text{Out}(\bar{x}') \leftarrow \mathsf{body}(\tau)$ for all frontier-guarded rules τ with a head of the form $\text{Out}^i(\bar{z}', \bar{x}')$. □

Towards the end of this section, let us briefly point out why our construction for Lemma 4.2.22 fails for monadic Datalog queries by means of an example.

Example 4.2.25. Consider the monadic Datalog query $Q = (P, \text{Out})$ with the following two rules.

$$R(x) \leftarrow S(x), N(x) \qquad\qquad \text{Out}(x) \leftarrow T(y), N(y), R(x)$$

The extensional symbols of P are S, T and N. Therefore, P is not frontier-guarded because the rule for the output symbol Out is not.

The set modest Σ of data-moving distribution constraints consists of the single constraint

$$R(x')@\kappa, N(y')@\lambda \rightarrow R(x')@\lambda.$$

We prove in Section 4.2.4 that Q and Σ belong to classes which enjoy the polynomial communication property, because, in a nutshell, the atoms $R(x')@\kappa$ and $N(y')@\lambda$ do *not* share any variable. In this case, it also follows easily from the observation that all atoms in the body, except for the atom $R(x')@\kappa$, which witnesses the constraint being data-moving, are extensional.

Finally, the hash policy scheme is just $Z = \{(S, 1, (1)), (T, 1, (1)), (N, 1, (1))\}$.

Following the construction in the proof for Lemma 4.2.22, the Datalog program P''' has the following rules.

$$\begin{aligned} S^1(x, x) &\leftarrow S(x) & R^1(z, x) &\leftarrow S^1(z, x), N^1(z, x) \\ T^1(x, x) &\leftarrow T(x) & \text{Out}^1(z, x) &\leftarrow T^1(z, y), N^1(z, y), R^1(z, x) \\ N^1(x, x) &\leftarrow N(x, x) & \text{Out}(x) &\leftarrow \text{Out}^1(z, x) \end{aligned}$$

The Datalog program P'' has the additional rule

$$\tau\colon \text{Out}^1(z, x') \leftarrow T^1(z, y), N^1(z, y), N^1(z, y'), R^1(z', x')$$

that results from replacing $R^1(z, x)$ in $\text{Out}^1(z, x) \leftarrow T^1(z, y), N^1(z, y), R^1(z, x)$ with the (translated) body of the constraint.[8] We claim that the last step of the construction does not yield a frontier-guarded Datalog query equivalent to $Q'' = (P'', \text{Out})$. For this purpose, consider the global database $G = \{S(1), T(2), N(1), N(2)\}$. We have that $\text{Out}^1(2, 1) \in P''(G)$. A proof tree for $\text{Out}^1(2, 1)$ is illustrated in Figure 4.1. Its root node

[8] It is not hard to see that this additional rule suffices to obtain the desired query that simulates the distributed evaluation. In fact, to derive an output fact it suffices to send the corresponding R-fact once to a server with a T- and a matching N-fact.

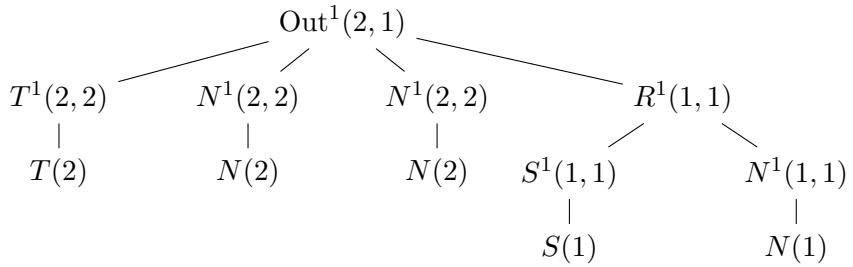

Figure 4.1: Proof tree for $\text{Out}^1(2,1)$ with respect to the Datalog program P'' and the database G from Example 4.2.25.

is witnessed by τ. No matter which extensional atom is added to $\mathsf{body}(\tau)$, the resulting rule cannot derive $\text{Out}^1(2,1)$ because there is *no* extensional fact in G containing the values 2 and 1.

Lastly, it is not hard to see that just removing τ is not an option either. Indeed, the proof tree in Figure 4.1 can easily be extended to a proof tree for $\text{Out}(1)$, thanks to the rule $\text{Out}(x) \leftarrow \text{Out}^1(z,x)$; and this is the only possible proof tree for $\text{Out}(1)$ with respect to P'' and G. ◁

4.2.4 Modest Communication Policies

In this section we define a syntactical restriction of data-moving distribution constraints that guarantees the polynomial communication property, even in combination with the class DL of all Datalog queries. The restriction is twofold.

First, we require that to test whether a fact $R(\bar{a})$ should be communicated from a server to another server, no other domain values than those in $\bar{a}$ need to be communicated.[9] In particular, for a constraint with only two server variables κ and λ, for the sending and the receiving server, we want to enable κ and λ to test "their" atoms independently, only communicating the domain values in $\bar{a}$. This leads to the following definition.

Definition 4.2.26 (Guarded Communication)**.** A data-moving distribution constraint σ has *guarded communication* if all distributed atoms $A@\kappa, B@\lambda \in \mathsf{body}(\sigma)$ with $\kappa \neq \lambda$, satisfy $\mathsf{vars}(A) \cap \mathsf{vars}(B) \subseteq \mathsf{vars}(\mathsf{head}(\sigma))$.

Secondly, we require that to test whether a fact $R(\bar{a})$ should be communicated, every server can only consult facts it computed itself; except for $R(\bar{a})$, of course. As a consequence, computation-free proof trees will adhere to a kind of linearity conditions. Formally, this part of the restriction is not imposed on single distribution constraints but rather on *sets* of data-moving distribution constraints.

Definition 4.2.27 (Modest)**.** A set Σ of data-moving distribution constraints is *modest* if all constraints in Σ have guarded communication and, for every $\sigma \in \Sigma$, there is exactly

[9] It may still be required to send bits indicating which servers can send the fact or contains facts required to satisfy the body of a constraint.

one distributed atom[10] in $\mathsf{body}(\sigma)$ whose relation symbol occurs in the head of some constraint in Σ.

By Hash-MConstraints we denote the class of families $\mathcal{F}(Z, \Sigma)$, where Z is a hash policy scheme and Σ is a modest set of data-moving distribution constraints.

We can now state the main result of this section.

Theorem 4.2.28. PC(FGDL, Hash-MConstraints) *is* 2ExpTime*-complete. The lower bound even holds for instances with a primitive hash policy scheme.*

The lower bound will be implied by (the proof for) Proposition 4.2.16. The first step towards a proof for the upper bound is to show that modest sets of data-moving distribution constraints indeed have the polynomial communication property.

Lemma 4.2.29. *The class DL of Datalog queries and the class of modest sets of data-moving distribution constraints have the polynomial communication property.*

Proof. Let $Q = (P, \mathrm{Out})$ be some Datalog query, Σ be a modest set of data-moving distribution constraints, and $\mathcal{D}$ be a distributed database.

Let now $R(\bar{a})@k$ be a fact that has a (partial) computation-free proof tree T with respect to P Σ, and $\mathcal{D}$. It suffices to show that there is such a partial proof tree for $R(\bar{a})@k$ that has size polynomial in $\|Q\|$ and $\|\Sigma\|$.

Thanks to Σ being modest, the branching structure of T is almost linear: Each inner node has at most one child node that is an inner node, every other child node is a leaf. Let $v_0, \ldots, v_m$ be the nodes on the unique path from a leaf v_0 to the root $v_m = \mathsf{root}(T)$. Since all constraints in Σ are data-moving, all nodes v_i are labelled with some fact $R(\bar{a})@\ell_i$ for some server ℓ_i. That is, the labels of $v_0, \ldots, v_m$ form a sequence of head facts $R(\bar{a})@\ell_1, \ldots, R(\bar{a})@\ell_m$ with $\ell_m = k$. Furthermore, let, for each $i > 0$, σ_i be the constraint and ϑ_i be the network aware valuation which are the witnesses for the node v_i.

We argue that if $m > |\Sigma|$, then there is a partial proof tree for $R(\bar{a})@k$ of smaller depth. To this end, suppose $m > |\Sigma|$ holds. Then there are i, j with $0 < i < j$ such that $\sigma_i = \sigma_j$. Let $R(\bar{x})@\kappa$ be the head of σ_i and let $R(\bar{x})@\lambda$ be the distributed atom in $\mathsf{body}(\sigma_i)$ witnessing that σ_i is data-moving. Note that we have $\vartheta_i(R(\bar{x})@\kappa) = R(\bar{a})@\ell_i$ and $\vartheta_j(R(\bar{x})@\kappa) = R(\bar{a})@\ell_j$. That is, ϑ_i and ϑ_j agree on all head variables $\bar{x}$. Further, we can assume that $\kappa \neq \lambda$, because otherwise, v_j could just be replaced with v_{j-1}.

Let ϑ be the network aware valuation that is defined as ϑ_j for the variables occurring in $\mathsf{head}(\sigma_i)$ or in a distributed atom in $\mathsf{body}(\sigma_i)$ with server variable κ. In particular, $\vartheta(\kappa) = \vartheta_j(\kappa)$. For any other (server) variable ϑ is defined as ϑ_i. In particular, $\vartheta(\lambda) = \vartheta_i(\lambda)$. The valuation ϑ is well-defined because **(1)** every variable that occurs in an atom with server variable κ and in another atom with another server variable is a head variable since σ_i has guarded communication; and **(2)** ϑ_i and ϑ_j agree on all head variables $\bar{x}$.

Given the valuation ϑ the nodes $v_i, \ldots, v_j$ can be merged together into a new node v as follows. The new node v is labelled with $\vartheta(R(\bar{x})@\kappa) = R(\bar{a})@\ell_j$. It has the children of v_j with labels $S(\bar{b})@\ell_j$ induced by distributed atoms with server variable κ and the

[10] Clearly, this is the atom $A@\lambda$ required by the definition of *data-moving*.

children of v_i labelled with facts which are *not* induced by an atom with server variable κ. Since ϑ agrees with ϑ_j on the former atoms and with ϑ_i on the latter by construction, σ_i and ϑ are witnesses for the proof tree property of v. Thus, the new tree is again a partial proof tree.

Thus, if a fact has a partial computation-free proof tree, it has one of depth at most $|\Sigma| + 1$. Clearly, such a tree has size polynomial in $\|Q\|$ and $\|\Sigma\|$. □

Together, Lemmas 4.2.22 and 4.2.29 allow us to reduce the parallel-correctness problem to the containment problem for frontier-guarded Datalog queries.

Proof of Theorem 4.2.28. The proof is almost identical to the proof of Proposition 4.2.16. We first argue that PC(FGDL, Hash-MConstraints) ∈ 2ExpTime. Let Q be a frontier-guarded Datalog query, Z be a hash policy scheme, and Σ be a modest set of data-moving distribution constraints.

Thanks to Lemmas 4.2.22 and 4.2.29 there is a frontier-guarded Datalog query Q' such that, for every global database G, we have that $Q'(G) = [Q, \gamma](\delta(G))$ for some $(\delta, \gamma) \in \mathcal{F}(Z, \Sigma)$ with δ scattering G.

We claim that Q is parallel-correct w.r.t. $\mathcal{F}(Z, \Sigma)$ if and only if $Q \sqsubseteq Q'$ holds. Suppose $Q \sqsubseteq Q'$ holds and let G be a global database. Let (δ, γ) be the policy pair guaranteed by Lemma 4.2.22 such that δ scatters G and $[Q, \gamma](\delta(G)) = Q'(G)$ holds. By assumption we have $Q(G) \subseteq Q'(G)$, and, thus, $Q(G) \subseteq [Q, \gamma](\delta(G))$. Thanks to Lemma 4.2.6 we can conclude that Q is parallel-correct w.r.t. $\mathcal{F}(Z, \Sigma)$.

Conversely, suppose Q is parallel-correct w.r.t. $\mathcal{F}(Z, \Sigma)$. Let G be a global database and (δ, γ) be the policy pair satisfying $[Q, \gamma](\delta(G)) = Q'(G)$ due to Lemma 4.2.22. Since Q is parallel-correct we have $Q(G) = [Q, \gamma](\delta(G))$. Together we have, in particular, $Q(G) \subseteq Q'(G)$. We can conclude that $Q \sqsubseteq Q'$ holds.

All in all, parallel-correctness of Q can thus be decided by testing $Q \sqsubseteq Q'$.

The upper bound for the complexity of the parallel-correctness now follows in the same way as in the proof for Proposition 4.2.16 with the help of Theorem 4.2.20.

The 2ExpTime-hardness of PC(FGDL, Hash-MConstraints) is implied by the lower bound proof of Proposition 4.2.16. Indeed, the set Σ of data-moving distribution constraints constructed in the proof consists of a single constraint, namely

$$\mathrm{Out}_2(\bar{x})@\kappa, E(\bar{x})@\lambda \to \mathrm{Out}_2(\bar{x})@\lambda.$$

The constraint has guarded communication because all variables from its body also appear in its head. And, since Out_2 is the only relation symbol occurring in the head of a constraint, the set Σ is modest. □

4.2.5 The Non-Transitive Communication Setting

In this section we consider a variant of our setting where the polynomial communication property is ensured by the (distributed) evaluation semantics. In the *non-transitive communication* setting each server can only consult facts derived in the computation phase on the server itself and extensional facts to determine whether a fact is communicated. In

particular, a server can only send facts it derived itself through a local computation. For hash-based communication policies this means that, in the non-transitive communication setting, the intensional atoms in the body of a distribution constraint have to be satisfied by facts derived locally on the server their server variable is mapped to.

For frontier-guarded Datalog queries and sets of data-moving distribution constraints, we will obtain the expected result, namely that parallel-correctness for these classes is 2ExpTime-complete in the non-transitive communication setting. Interestingly, however, we will see that parallel-correctness for monadic Datalog queries remains undecidable in the non-transitive communication setting.

Before we state this formally, we first define the semantics of the non-transitive communication setting. We use *adorned* relation symbols to do so within our framework and *standard semantics* defined in Section 4.1. In general, an *adorned* relation symbol is a relation symbol augmented by labels [cf., e.g., GMT13]. For our purposes, we only require one label that indicates whether a fact has been derived in a local computation. Consequently, a fact whose relation symbol does *not* have this label has then been received from another server. For a relation symbol R we write $R^\bullet$ for the adorned symbol augmented by this label. Furthermore, for an atom $A = R(\bar{x})$, we write $A^\bullet$ for $R^\bullet(\bar{x})$. We call $R^\bullet$-facts and $R^\bullet$-atoms *adorned facts and atoms*, respectively. The semantics of the non-transitive communication setting are defined as follows.

Definition* 4.2.30. Let $Q = (P, \text{Out})$ be a Datalog query and Σ a set of data-moving distribution constraints. The *non-transitive translation* of the query Q is the Datalog query $Q^\bullet = (P^\bullet, \text{Out})$ where $P^\bullet$ is the Datalog program that consists of all rules of P, and, for each rule $\tau \in P$, has an additional *adorned rule* $\mathsf{head}(\tau)^\bullet \leftarrow \mathsf{body}(\tau)$.

The *non-transitive translation* of a data-moving distribution constraint $\sigma \in \Sigma$ of the form

$$R(\bar{x})@\kappa, R_1(\bar{y}_1)@\kappa_1, \ldots, R_m(\bar{y}_m)@\kappa_m, E_1(\bar{z}_1)@\mu_1, \ldots, E_n(\bar{z}_n)@\mu_n \to R(\bar{x})@\lambda$$

with intensional symbols $R, R_1, \ldots, R_m$, and extensional symbols $E_1, \ldots, E_n$ is the data-moving distribution constraint $\sigma^\bullet$ defined by

$$R(\bar{x})@\kappa, R^\bullet(\bar{x})@\kappa, R_1^\bullet(\bar{y}_1)@\kappa_1, \ldots, R_m^\bullet(\bar{y}_m)@\kappa_m, E_1(\bar{z}_1)@\mu_1, \ldots, E_n(\bar{z}_n)@\mu_n \to R(\bar{x})@\lambda.$$

The *non-transitive translation* of the set Σ is then $\Sigma^\bullet = \{\sigma^\bullet \mid \sigma \in \sigma\}$.

Note that a non-transitive translation $P^\bullet$ can derive every fact that P can derive, since it has all rules of P. Additionally, it derives, for every fact that P derives, the corresponding adorned fact. Over a global database a query Q is thus equivalent to its non-transitive translation $Q^\bullet$.

Thanks to the adorned atoms in the body of a non-transitive translation $\sigma^\bullet$ and the head of $\sigma^\bullet$ *not* being adorned, it is ensured that the body is indeed satisfied by facts derived locally. Note that $\sigma^\bullet$ still has the atom $R(\bar{x})@\kappa$ to comply with the definition of data-moving. However, since $R^\bullet(\bar{x})@\kappa$ is also present, it is ensured that these R-facts can be derived locally, too.

Let Q be a Datalog query, Z be a hash policy scheme, Σ be a set of data-moving distribution constraints, and $\gamma_{\Sigma,\mathcal{N}}$ be a communication policy for some network $\mathcal{N}$. We define the *parallel query result of a Datalog query Q on a distributed database $\mathcal{D}$ according to $\gamma_{\Sigma,\mathcal{N}}$ in the non-transitive communication setting* as $[Q^\bullet, \gamma_{\Sigma^\bullet,\mathcal{N}}](\mathcal{D})$.

Accordingly, we say that Q is *parallel-correct in the non-transitive communication setting* w.r.t. a policy pair $(\delta_{Z,H}, \gamma_{\Sigma,\mathcal{N}}) \in \mathcal{F}(Z,\Sigma)$ if $[Q^\bullet, \gamma_{\Sigma^\bullet,\mathcal{N}}](\mathcal{D}) = Q(G)$ holds for every distributed database $\mathcal{D} = (G, \mathcal{I})$ that complies with $\delta_{Z,H}$. Observe that $[Q^\bullet, \gamma_{\Sigma^\bullet,\mathcal{N}}](\mathcal{D}) = Q(G)$ holds if and only if $[Q^\bullet, \gamma_{\Sigma^\bullet,\mathcal{N}}](\mathcal{D}) = Q^\bullet(G)$ holds, because Q and $Q^\bullet$ are equivalent (over global databases). Hence, Q is parallel-correct w.r.t. a family $\mathcal{F}(Z,\Sigma)$ in the non-transitive communication setting if $Q^\bullet$ is parallel-correct w.r.t. $\mathcal{F}(Z,\Sigma^\bullet)$.

We slightly abuse notation and write "PC($\mathbb{Q}$, Hash-Constraints) in the non-transitive communication setting" for the decision problem that asks, given a query $Q \in \mathbb{Q}$, a hash policy scheme Z, and a set Σ of data-moving distribution constraints, whether Q is parallel-correct w.r.t. $\mathcal{F}(Z,\Sigma)$ in the non-transitive communication setting.

As for Datalog queries and policies in general, we always assume that non-transitive translations of queries Q and sets Σ of data-moving distribution constraint "match". In particular, adorned symbols in $Q^\bullet$ do *not* occur in Σ and adorned symbols in $\Sigma^\bullet$ *not* in Q.

The main result of this section concerning frontier-guarded Datalog queries is the following.

Theorem 4.2.31. *In the non-transitive communication setting the parallel-correctness problem* PC(FGDL, Hash-Constraints) *is* 2ExpTime*-complete. The lower bound even holds for instances with a primitive hash policy scheme.*

The proof approach is the same as for modest sets of data-moving distribution constraints. We first show that non-transitive translation $Q^\bullet$ and $\Sigma^\bullet$ have the polynomial communication property.

Lemma 4.2.32. *The class of Datalog queries $Q^\bullet$ and sets $\Sigma^\bullet$ of data-moving distribution constraints which are non-transitive translations have the polynomial communication property.*

Proof. Let $Q^\bullet = (P^\bullet, \text{Out})$ be a Datalog query, and $\Sigma^\bullet$ a set of data-moving distribution constraints which are the non-transitive translations of a Datalog query Q and set Σ. Furthermore, let T be a proof tree over some network $\mathcal{N}$ for some fact with respect to $P^\bullet$, and $\Sigma^\bullet$.

Consider a computation-free subtree of T whose root node v is labelled with some fact $R(\bar{a})@k$ and witnessed by a constraint $\sigma^\bullet$. By Definition 4.2.30 $\mathsf{body}(\sigma^\bullet)$ consists of two atoms of the form $R(\bar{x})@\lambda$ and $R^\bullet(\bar{x})@\lambda$, and (possibly) further atoms with extensional or adorned relations symbols. Consequently, all but one child node of v are labelled with extensional or adorned facts. Since *no* constraint in $\Sigma^\bullet$ has a head with an adorned relation symbol, all child nodes labelled with adorned facts are witnessed by Datalog rules.

The only exception is possibly the child node w for $R(\bar{x})@\lambda$ labelled with $R(\bar{a})@\ell$ for some $\ell \in \mathcal{N}$. Observe that there also is a node w' labelled with $R^\bullet(\bar{x})@\lambda$ because $R^\bullet(\bar{x})@\lambda$

occurs in the body of the constraint $\sigma^\bullet$. Then the subtree below w can be replaced with the subtree below w'. Furthermore, w can then be witnessed by a Datalog rule (and the same valuation). This is true since w' is witnessed by a Datalog rule with a head of the form $R^\bullet(\bar{y})$, and by Definition 4.2.30 there then also is a Datalog rule with the same body and head $R(\bar{y})$.

Applying this procedure iteratively yields a proof tree whose computation-free subtrees have depth at most 1. In particular, the size of such a computation-free subtree is polynomial in $\|Q^\bullet\|$, and $\|\Sigma^\bullet\|$. □

Given Lemma 4.2.32, Theorem 4.2.31 can now be proved analogously to Theorem 4.2.28.

Proof of Theorem 4.2.31. As usual, we first argue that, in the non-transitive communication setting, the parallel-correctness problem PC(FGDL, Hash-Constraints) is in 2ExpTime. To this end, let Q be a frontier-guarded Datalog query, Z be a hash policy scheme, and Σ be a set of data-moving distribution constraints.

By definition, deciding whether Q is parallel-correct w.r.t. $\mathcal{F}(Z, \Sigma)$ in the non-transitive communication setting boils down to testing whether the non-transitive translation $Q^\bullet$ is parallel-correct w.r.t. $\mathcal{F}(Z, \Sigma^\bullet)$ (in our standard setting). Due to Lemma 4.2.32 non-transitive translations have the polynomial communication property. Moreover, $Q^\bullet$ is frontier-guarded because the body and the head variables of any rule of $Q^\bullet$ are the same as for some rule of Q. Thus, there is a frontier-guarded Datalog query Q' such that, for every global database G, we have that $Q'(G) = [Q, \gamma](\delta(G))$ for some $(\delta, \gamma) \in \mathcal{F}(Z, \Sigma^\bullet)$ with δ scattering G.

We then have that $Q^\bullet$ is parallel-correct w.r.t. $\mathcal{F}(Z, \Sigma^\bullet)$ if and only if $Q^\bullet \sqsubseteq Q'$ holds. The proof is exactly the same as in Theorem 4.2.28 (and Proposition 4.2.16), as is the complexity analysis for testing $Q^\bullet \sqsubseteq Q'$ in doubly exponential time utilizing Theorem 4.2.20.

The 2ExpTime-hardness is also implied by the reduction from the containment problem for the lower bound proof for Proposition 4.2.16. To see this, recall that the constructed Datalog query is $Q = (P, \mathrm{Out})$ where

$$P = P_1' \cup P_2' \cup \big\{\mathrm{Out}(\bar{x}) \leftarrow \mathrm{Out}_1(\bar{x}), E(\bar{x})\big\} \cup \big\{\mathrm{Out}(\bar{x}) \leftarrow \mathrm{Out}_2(\bar{x}), E(\bar{x})\big\},$$

and neither the output symbol Out nor the extensional symbol E occur in the Datalog programs P_1' and P_2'. Furthermore, Out_1 occurs only in P_1', Out_2 only in P_2', and the sets $\mathsf{idb}(P_1')$ and $\mathsf{idb}(P_2')$ of intensional symbols are disjoint. The constructed set of data-moving distribution constraints consists of a single distribution constraint σ, namely

$$\sigma\colon \mathrm{Out}_2(\bar{x})@\kappa, E(\bar{x})@\lambda \to \mathrm{Out}_2(\bar{x})@\lambda.$$

Intuitively, the non-transitive translations $Q^\bullet$ and $\sigma^\bullet$ of Q and σ do *not* change the outcome of distributed evaluations. More precisely, we claim that $[Q^\bullet, \gamma_{\{\sigma^\bullet\},\mathcal{N}}](\mathcal{D}) = [Q, \gamma_{\{\sigma\},\mathcal{N}}](\mathcal{D})$ holds for every network $\mathcal{N}$ and distributed database $\mathcal{D}$ over $\mathcal{N}$. Then Q and σ can be replaced with $Q^\bullet$ and $\sigma^\bullet$ in the lower bound proof for Proposition 4.2.16, yielding a reduction from the containment problem for frontier-guarded Datalog queries,

which is 2ExpTime-hard, to the parallel-correctness problem in the non-transitive communication setting.

It suffices to show that, if there is a proof tree for a fact $R(\bar{a})@\ell$ with respect to P and $\{\sigma\}$, then there is also a proof tree for $R(\bar{a})@\ell$ with respect to $P^\bullet$ and $\{\sigma^\bullet\}$. The proof is by induction on the structure of a proof tree T with respect to P and $\{\sigma\}$ with a slightly stronger induction hypothesis: We require that if the root node of the tree with respect to P and $\{\sigma\}$ is witnessed by a Datalog rule, then so is the root node of the tree with respect to $P^\bullet$ and $\{\sigma^\bullet\}$.

For leaves there is nothing to show. For nodes witnessed by a Datalog rule the claim is immediate by the induction hypothesis, and because we have $P \subseteq P^\bullet$ by Definition 4.2.30.

It remains the case that a node v is witnessed by σ. Then v is labelled with $\mathrm{Out}_2(\bar{a})@\ell$, and has two children, one of which is labelled with $E(\bar{a})@\ell$. Moreover, since σ is data-moving, there is a node below v which is labelled with $\mathrm{Out}_2(\bar{a})@k$ for some k, and witnessed by a Datalog rule. By the induction hypothesis, there is then also a proof tree for $\mathrm{Out}_2(\bar{a})@k$ with respect to $P^\bullet$ and $\{\sigma^\bullet\}$. The same is trivially true for $\mathrm{Out}_2(\bar{a})@k$ since it is extensional. Furthermore, there then is a proof tree for $\mathrm{Out}_2^\bullet(\bar{a})@k$ because $P^\bullet$ has a rule $\mathsf{head}(\tau)^\bullet \leftarrow \mathsf{body}(\tau)$ for every rule $\tau \in P$, and we can assume that the root node of the tree for $\mathrm{Out}_2(\bar{a})@k$ is indeed witnessed by a rule, thanks to the induction hypothesis.

It is now straightforward to combine the proof trees for $\mathrm{Out}_2(\bar{a})@k$, $\mathrm{Out}_2^\bullet(\bar{a})@k$, and $E(\bar{a})@\ell$ into a proof tree for $\mathrm{Out}_2(\bar{a})@\ell$ whose root node is witnessed by

$$\sigma^\bullet\colon \mathrm{Out}_2(\bar{x})@\kappa, \mathrm{Out}_2^\bullet(\bar{x})@\kappa, E(\bar{x})@\lambda \to \mathrm{Out}_2(\bar{x})@\lambda.$$

Overall, we can conclude that, in the non-transitive communication setting, the parallel-correctness problem PC(FGDL, Hash-Constraints) is 2ExpTime-complete. □

We now turn to parallel-correctness for monadic Datalog queries. As mentioned earlier, the parallel-correctness problem for monadic Datalog query in the non-transitive communication setting is undecidable.

Theorem* 4.2.33. *In the non-transitive communication setting the parallel-correctness problem* PC(MDL, Hash-Constraints) *is undecidable.*

Let us emphasize that Theorem 4.2.31 and Theorem 4.2.33 together imply that it is, in general, not possible to transform a monadic Datalog query into a frontier-guarded Datalog query that yields the same parallel query results.

The reason for this drastic difference to the parallel-correctness problem for frontier-guarded Datalog queries is actually quite simple. It is due to "copy rules" that have the form $R(x) \leftarrow R(x)$ and allow to circumvent the restricted semantics of the non-transitive communication setting (cf. Example 4.1.2). In particular, such copy rules can be added to the monadic Datalog query constructed in the undecidability proof for Theorem 4.2.12.

The following result makes precise how copy rules can be used to circumvent the restricted semantics of the non-transitive communication setting.

Lemma 4.2.34. *For every monadic Datalog query Q' there is an equivalent monadic Datalog query Q such that $[Q^\bullet, \gamma_{\Sigma^\bullet,\mathcal{N}}](\mathcal{D}) = [Q', \gamma_{\Sigma,\mathcal{N}}](\mathcal{D})$ holds for every set Σ of data-moving distribution constraints, network $\mathcal{N}$, and distributed database $\mathcal{D}$ over $\mathcal{N}$.*

Proof. Let $Q' = (P', \text{Out})$ be a monadic Datalog query. The Datalog program P of the monadic Datalog query $Q = (P, \text{Out})$ consists of all rules in P' and rules $R(x) \leftarrow R(x)$ for all $R \in \mathsf{idb}(P')$.

We first observe that Q' and Q are equivalent. In fact, every proof tree with respect to Q' is a proof tree with respect to Q, since we have $P' \subseteq P$. Conversely, nodes witnessed by a rule $R(x) \leftarrow R(x)$ in a proof tree with respect to Q, can be iteratively replaced by their (single) child node to yield a proof tree with respect to Q'.

Now, let Σ be a set of data-moving distribution constraints, $\mathcal{N}$ be a network, and $\mathcal{D}$ a distributed database over $\mathcal{N}$. Using the same reasoning as above we obtain that $[Q, \gamma_{\Sigma,\mathcal{N}}](\mathcal{D}) = [Q', \gamma_{\Sigma,\mathcal{N}}](\mathcal{D})$ holds. Thus, it suffices to prove

$$[Q^\bullet, \gamma_{\Sigma^\bullet,\mathcal{N}}](\mathcal{D}) = [Q, \gamma_{\Sigma,\mathcal{N}}](\mathcal{D}).$$

For $[Q^\bullet, \gamma_{\Sigma^\bullet,\mathcal{N}}](\mathcal{D}) \subseteq [Q, \gamma_{\Sigma,\mathcal{N}}](\mathcal{D})$, we observe that every proof tree with respect to $P^\bullet$ and $\Sigma^\bullet$ can be turned into a proof tree with respect to P and Σ by removing the nodes for the atoms $R(\bar{x})@\kappa$ occurring in the body of constraints $\sigma^\bullet$ to comply with the definition of data-moving, and then replacing every adorned symbol $R^\bullet$ with R.

For the inclusion $[Q^\bullet, \gamma_{\Sigma^\bullet,\mathcal{N}}](\mathcal{D}) \supseteq [Q, \gamma_{\Sigma,\mathcal{N}}](\mathcal{D})$, consider a proof tree with respect to P and Σ. We show by structural induction over the tree structure how to obtain a proof tree with respect to $P^\bullet$ and $\Sigma^\bullet$. For leaves there is nothing to show, since they are just labelled with extensional facts. Similarly, nodes whose proof tree property is witnessed by a Datalog rule can just be inherited, because we have $P \subseteq P^\bullet$ and there are proof trees for all children thanks to the induction hypothesis.

Lastly, consider the case that the proof tree property of a node v is witnessed by a constraint σ from Σ. Let $w_1, \ldots, w_m$ be the children of v, and $R_1(\bar{a}_1)@k_1, \ldots, R_n(\bar{a}_n)@k_n$ their labels. By the induction hypothesis there are then proof trees for the facts $R_1(\bar{a}_1)@k_1, \ldots, R_n(\bar{a}_n)@k_n$ with respect to $P^\bullet$ and $\Sigma^\bullet$. Without loss of generality, we can assume that $R_1, \ldots, R_m$ for some $m \leq n$ are intensional and $R_{m+1}, \ldots, R_n$ are extensional symbols. Moreover, we can assume that R_1 is the symbol occurring in the head of the constraint σ.

Since P contains a rule $R_i(x) \leftarrow R_i(x)$ for every $i \in [1, m]$, the non-transitive translation $P^\bullet$ contains rules $R_i^\bullet(x) \leftarrow R_i(x)$ for all $i \in [1, m]$. Thus, it is straightforward to obtain proof trees for $R_1^\bullet(\bar{a}_1)@k_1, \ldots, R_m^\bullet(\bar{a}_m)@k_m$.

A proof tree for $\mathsf{fact}(v)$ with respect to $P^\bullet$ and $\Sigma^\bullet$ can now be assembled as follows. The root is labelled with $\mathsf{fact}(v)$. Its children are the root nodes for the trees for the facts

$$R_1(\bar{a}_1)@k_1, R_1^\bullet(\bar{a}_1)@k_1, \ldots, R_m^\bullet(\bar{a}_m)@k_m, R_{m+1}(\bar{a}_{m+1})@k_{m+1}, \ldots, R_n(\bar{a}_n)@k_n.$$

Since the proof tree property of v is witnessed by σ, it follows by Definition 4.2.30 that the proof tree property of the new root node is witnessed by $\sigma^\bullet$ (and the same valuation). □

It remains to conclude that the parallel-correctness problem for monadic Datalog queries in the non-transitive communication setting is undecidable.

Proof of Theorem 4.2.33. Lemma 4.2.34 yields a straightforward reduction from the parallel-correctness problem PC(MDL, Hash-Constraints) to its counterpart in the non-transitive communication setting. Since the former is undecidable due to Theorem 4.2.12, the same is true for PC(MDL, Hash-Constraints) in the non-transitive communication setting. □

4.3 The Containment Problem for Frontier-Guarded Datalog

In this section we prove Theorem 4.2.20 which states an upper bound for the complexity of the containment problem Cont(DL, FGDL). Recall that we require the upper bound to be at most singly exponential in the number of rules of the two given Datalog queries. This bound is, however, only implicit in the proof[11] given by Bourhis et al. [BKR15a, Theorem 7]. Another reason why we provide a proof here is, that we will build upon the construction used in the proof to obtain an upper bound for the parallel-boundedness problem in Section 4.4.

We start with an outline of the proof approach. Each Datalog query $Q = (P, \text{Out})$ induces a (possibly infinite) set $U(Q)$ of conjunctive queries in a natural way by finite "unwindings" of P. It is well-known that for two Datalog queries Q, Q' it holds $Q \sqsubseteq Q'$ if and only if each conjunctive query from $U(Q)$ is contained in some conjunctive query from $U(Q')$.[12] This is in turn equivalent to the existence of a homomorphisms from every query in $U(Q')$ to some query from $U(Q)$ [cf. CM77, Proof of Lemma 13].

The proof idea from Bourhis et al. [BKR15a], going back to Cosmadakis et al. [Cos+88], is to use alternating two-way tree automata to test the existence of such homomorphisms. This approach requires to encode conjunctive queries by trees over a finite alphabet, even though the number of variables in the queries from $U(Q)$ and $U(Q')$ can be unlimited.

In the following, we define symbolic proof trees to represent (and formally define) the queries in $U(Q)$ and $U(Q')$. We note that our definition does not immediately yield a finite alphabet; we will discuss that this can be achieved afterwards.

Symbolic Proof Trees. A *rule application* for a Datalog rule τ is a variable substitution $\alpha\colon \mathsf{vars}(\tau) \to \mathsf{vars}$. The *rule instantiation* $\alpha(\tau)$ of τ induced by α is the (query) rule $\alpha(\mathsf{head}(\tau)) \leftarrow \alpha(\mathsf{body}(\tau))$. Clearly, $\alpha(\tau)$ is frontier-guarded, if τ is frontier-guarded.

Definition 4.3.1 (Symbolic Proof Tree)**.** A *symbolic proof tree* for a Datalog program P is a rooted tree T in which every node v is labelled with a rule instantiation, denoted $\mathsf{head}(v) \leftarrow \mathsf{body}(v)$, of some rule of P, and which has the following properties for every node v.

[11] A more detailed proof is also available in an extended technical report [BKR15b, Theorem 9].

[12] This is, for instance, stated for finite sets of conjunctive queries by Sagiv and Yannakakis [SY80, Theorem 3], but their proof also applies to countable unions.

(a) The children of v are labelled with rules for the intensional atoms of $\mathsf{body}(v)$:

$$\big\{R(\bar{x}) \in \mathsf{body}(v) \mid R \in \mathsf{idb}(P)\big\} = \big\{\mathsf{head}(w) \mid w \in \mathsf{children}_T(v)\big\}.$$

(b) All variables that occur in $\mathsf{body}(v)$ but not in $\mathsf{head}(v)$, do *not* occur in the label of the parent of v.

We note that, in particular, for every leaf v of T, $\mathsf{body}(v)$ only contains extensional atoms. A *symbolic proof tree for a query* $Q = (P, \mathrm{Out})$ is a symbolic proof tree for P whose root node is labelled with a rule for Out.

Let v_1, v_2 be two nodes of a symbolic proof tree for some Datalog program P, and v their lowest common ancestor. Then v_1, v_2 are *x-connected* if the variable x occurs in $\mathsf{head}(w)$ for every node v on the shortest path between v_1 and v_2, except possibly v.

With each symbolic proof tree T we associate a conjunctive query $\mathsf{q}(T)$ which is obtained from T as follows. For all variables x and maximal x-connected components V, all occurrences of x in V are replaced by a distinct fresh variable x_V. Then $\mathsf{q}(T)$ is the conjunctive query with head $\mathsf{head}(\mathsf{root}(T))$ and whose body consists of all extensional atoms occurring in T, after the variable replacement. Observe that different, unconnected occurrences of the same variable in a symbolic proof tree hence represent different variables in the associated conjunctive query.

Example 4.3.2. Consider the Datalog query $Q = (P, \mathrm{Out})$ where P consists of the following rules.

$$R(x_1, x_2) \leftarrow E(x_1, x_2) \qquad \mathrm{Out}(x_2) \leftarrow S(x_1), R(x_1, x_2)$$
$$R(x_1, x_2) \leftarrow R(x_1, x_3), R(x_3, x_2)$$

It asks for all nodes reachable from a *starting node* in S via edges in E. Figure 4.2 shows a symbolic proof tree T for Q. To obtain an associated conjunctive query $\mathsf{q}(T)$ it suffices to replace all occurrences of x_4 in the right subtree with a fresh variable x_4'. The rule defining the conjunctive query is then

$$\mathrm{Out}(x_2) \leftarrow S(x_1), E(x_1, x_4), E(x_4, x_3), E(x_3, x_4'), E(x_4', x_2). \qquad \triangleleft$$

Further on, we associate with a set $\mathcal{T}$ of symbolic proof trees a (possibly infinite) set $\mathsf{q}(\mathcal{T})$ of conjunctive queries via $\mathsf{q}(\mathcal{T}) = \{\mathsf{q}(T) \mid T \in \mathcal{T}\}$.

For a Datalog query Q we denote by $\mathcal{T}_Q^\star$ the set of all symbolic proof trees for Q. Each subset $\mathcal{T} \subseteq \mathcal{T}_Q^\star$ induces a query that maps each global database G to the query result $\bigcup_{T \in \mathcal{T}} \mathsf{q}(T)(G)$.[13] Slightly abusing notation we denote this query also by $\mathsf{q}(\mathcal{T})$.

We believe the following to be folklore. Nevertheless we will provide a proof for the sake of completeness.

[13] This definition indeed yields a well-defined query, because, for all $T \in \mathcal{T}_Q^\star$, $\mathsf{q}(T)(G)$ is a set of Out-facts. Hence, a union of such results forms a relation.

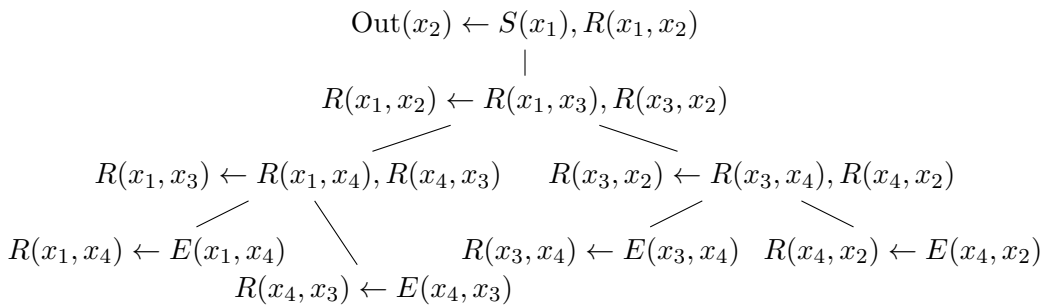

Figure 4.2: Symbolic proof tree for the Datalog query defined in Example 4.3.2. There are two maximal x_4-connected components, namely the left and the right subtree, respectively.

Lemma 4.3.3 (Folklore)**.** *Every Datalog query Q is equivalent to the query $\mathsf{q}(\mathcal{T}^\star_Q)$.*

Proof. Let $Q = (P, \mathrm{Out})$ be a Datalog query. We first show $Q \sqsubseteq \mathsf{q}(\mathcal{T}^\star_Q)$. To this end, let G be a global database, and $\mathrm{Out}(\bar{a}) \in Q(G)$. Then there is a proof tree for $\mathrm{Out}(\bar{a})$ with respect to P and G. This proof tree can be transformed into a symbolic proof tree in two steps as follows. We introduce, for every domain value a that occurs in G, a variables x_a. For every inner node v, let τ_v and ϑ_v be the Datalog rule and the valuation witnessing v. Let α_v be the application that maps each variable $y \in \mathsf{vars}(\tau_v)$ to $x_{\vartheta(y)}$. Changing the label of every inner node to $\alpha_v(\tau_v)$ and removing all leaves (originally labelled with extensional facts), yields a tree that satisfies Property (a) of Definition 4.3.1, but not necessarily Property (b). Property (b) can be established in a second step by replacing variables in a top-down fashion: If a variable x_a occurs in some $\mathsf{body}(v)$ and the label of the parent of v, but not in $\mathsf{head}(v)$, it is replaced with a fresh variable in $\mathsf{body}(v)$ and in the labels of all nodes below v.

Let now T be the symbolic proof tree obtained this way, and ϑ be the valuation that maps a variable x to a, if $x = x_a$ or x is a variable introduced as a replacement for x_a in the second step. Then every atom in $\mathsf{q}(T)$ is mapped to an extensional fact in the original proof tree for $\mathrm{Out}(\bar{a})$ and the head of $\mathsf{q}(T)$ is mapped to $\mathrm{Out}(\bar{a})$. Thus, $\mathrm{Out}(\bar{a}) \in \mathsf{q}(\mathcal{T}^\star_Q)$.

For the inclusion $\mathsf{q}(\mathcal{T}^\star_Q) \sqsubseteq Q$, consider a fact $\mathrm{Out}(\bar{a}) \in \mathsf{q}(T)(G)$ for some $T \in \mathcal{T}^\star_Q$ and global database G. Then there is a valuation ϑ that maps the head of $\mathsf{q}(T)(G)$ to $\mathrm{Out}(\bar{a})$ and its body into G. Let T' be the tree obtained by replacing variables as required by the definition of $\mathsf{q}(T)$. Then every variable of $\mathsf{q}(T)$ appears in some label of T'. The converse is true as well: Thanks to Property (a) of symbolic proof trees and Datalog rules being safe, every variable occurs in some extensional atom in some label of T'; and, hence, in $\mathsf{q}(T)$.

A proof tree for $\mathrm{Out}(\bar{x})$ can then be obtained by replacing the label of every node v of T' with the fact $\vartheta(\mathsf{head}(v))$, and adding a leaf labelled $\vartheta(A)$ as child to v, for every extensional atom in $\mathsf{body}(v)$. A node v is then witnessed by τ_v and $\vartheta \circ \alpha_v$ where τ_v and α_v are the Datalog rule and the application that induced the original label $\mathsf{head}(v) \leftarrow \mathsf{body}(v)$. We can conclude $\mathrm{Out}(\bar{a}) \in Q(G)$. □

Query Containment and Automata. As mentioned earlier, $\mathcal{T}_Q^\star$ cannot be understood as a tree language over a finite alphabet, because the number of variables that can occur in a symbolic proof tree is unbounded. However, thanks to the following result by Chaudhuri and Vardi [CV97], we can restrict attention to symbolic proof trees over a finite pool of variables.

Lemma 4.3.4 [CV97, Section 5.1]**.** *Let Q be a Datalog query with n variables. For every $T \in \mathcal{T}_Q^\star$ there is a symbolic proof tree $T' \in \mathcal{T}_Q^\star$ with at most $2n$ variables and $\mathsf{q}(T') \equiv \mathsf{q}(T)$.*

For every Datalog query $Q = (P, \mathrm{Out})$ we fix a set $\{x_1, \dots, x_{2n}\}$ of $2n$ variables, where n is the number of variables occurring in Q. By $\mathcal{T}_Q$ we denote the set of all symbolic proof trees for Q over the set $\{x_1, \dots, x_{2n}\}$ of variables. Thanks to Lemma 4.3.3 and Lemma 4.3.4 we then have $Q \equiv \mathsf{q}(\mathcal{T}_Q)$. Furthermore, $\mathcal{T}_Q$ is a tree language over a (finite) alphabet. More precisely, the alphabet Γ_P consists of all rule instantiations $\alpha(\mathsf{head}(\tau)) \leftarrow \alpha(\mathsf{body}(\tau))$ with $\tau \in P$ and $\alpha \colon \mathsf{vars}(\tau) \to \{x_1, \dots, x_{2n}\}$. The rank of such a rule instantiation is the number of intensional atoms in its body. Its size is at most polynomial in the number and length of rules in P and exponential in the number of variables occurring in P.

The main step of the proof for Theorem 4.2.20 involves the construction of two automata for the given Datalog queries Q and Q'. The first automaton $\mathbb{A}_Q$ is simply for $\mathcal{T}_Q$. The second automaton $\mathbb{A}_{Q \sqsubseteq Q'}$ is for the tree language

$$\mathcal{T}_{Q \sqsubseteq Q'} = \big\{ T \in \mathcal{T}_Q \mid \text{ there is a tree } T' \in \mathcal{T}_{Q'} \text{ such that } \mathsf{q}(T) \sqsubseteq \mathsf{q}(T') \big\}.$$

To decide whether $Q \sqsubseteq Q'$ holds it then suffices to test for $\mathcal{T}_Q \subseteq \mathcal{T}_{Q \sqsubseteq Q'}$. The latter can be done, as usual, by testing whether the intersection of $\mathcal{T}_Q$ with the complement of $\mathcal{T}_{Q \sqsubseteq Q'}$ is empty. Chaudhuri and Vardi [CV97, Proposition 5.9] proved that a *non-deterministic bottom-up tree automaton*[14] $\mathbb{A}_Q$ for $\mathcal{T}_Q$ can be constructed in time and has size at most (singly) exponential in $\|P\|$. Moreover, Bourhis et al. [BKR15a] proved that an alternating two-way tree automaton $\mathbb{A}_{Q \sqsubseteq Q'}$ for $\mathcal{T}_{Q \sqsubseteq Q'}$ can be constructed in exponential time, if Q' is frontier-guarded. Thanks to Cosmadakis et al. [Cos+88, Theorem A.1], a non-deterministic bottom-up tree automaton for the complement of $\mathcal{T}_{Q \sqsubseteq Q'}$ can then be obtained in time (and has size) exponential in the size of $\mathbb{A}_{Q \sqsubseteq Q'}$; that is, doubly exponential in $\|P\| + \|P'\|$. Given the two non-deterministic automata, constructing an automaton for the intersection and checking for emptiness can be done in polynomial time [cf., e.g., Com+08, Proof of Theorem 1.3.1 and Theorem 1.7.4]. Altogether this then implies that $Q \sqsubseteq Q'$ can be tested in doubly exponential time.

To obtain the refined complexity bounds stated in Theorem 4.2.20, it suffices to argue that the size of $\mathbb{A}_{Q \sqsubseteq Q'}$ is only polynomial in the number of rules of P and P' (and at most singly exponential in the number of variables and size of rules). The automaton for the complement has then size doubly exponential in the number of variables and maximal size of a rules in P and P', but only singly exponential in the number of rules. Thus, yielding Theorem 4.2.20.

[14] For a formal definition of this well-known automata model we refer to [Com+08, Section 1.1].

Since $\mathcal{T}_{Q \sqsubseteq Q'}$ is a subset of $\mathcal{T}_Q$, the construction of $\mathbb{A}_{Q \sqsubseteq Q'}$ naturally builds upon $\mathbb{A}_Q$. Therefore, we require that $\mathbb{A}_Q$ also has polynomial size in the number of Datalog rules.

As mentioned earlier, the following result has been proven by Chaudhuri and Vardi [CV97], but we state it here with the more refined complexity bounds we need, and, for convenience, in terms of alternating two-way tree automata.[15]

Proposition 4.3.5 [CV97, Proposition 5.9]**.** *For every Datalog query $Q = (P, \mathrm{Out})$ there is an alternating two-way tree automaton $\mathbb{A}_Q$ recognizing $\mathcal{T}_Q$. It has size (and can be constructed in time)*

- *exponential in the number of variables in P;*
- *polynomial in the maximal size of a rule of P; and*
- *polynomial in the number of rules of P.*

For a formal definition of alternating two-way tree automata we refer to Section 2.5.2, and just recall here that the semantics are defined in terms of a two-player games between the existential player, Morgana, and the universal player, Arthur, on the input trees.

Proof of Proposition 4.3.5. First, we note that the alphabet Γ_P satisfies the size bounds, because its size is at most polynomial in the number and size of rules in P and exponential in the number of variables occurring in P.

We describe how the automaton $\mathbb{A}_Q$ verifies Properties (a) and (b) of Definition 4.3.1 in a top-down fashion by means of the underlying game semantics. At every node v Arthur either ascertains that one of the properties does not hold, or chooses an intensional atom from $\mathsf{body}(v)$. Morgana then chooses a child w of v and the game continues with the next round and w being the new node. For Arthur to make his next move, the rule instantiation of v and the atom he chose are "stored" in the state upon descending from v to w.

More precisely, $\mathbb{A}_Q$ has states (τ, A) for every $\tau \in \Gamma_P$ and intensional atom A from $\mathsf{body}(\tau)$. Intuitively, τ is the rule instantiation of the parent node and A the atom Arthur chose in the previous round. To verify the root node it also has states $(\varepsilon, \mathrm{Out}(\bar{x}))$ for all atoms $\mathrm{Out}(\bar{x})$ occurring as head atom of some rule instantiation in Γ_P. Here, ε is just a placeholder indicates that there is no parent node. Finally, it has an initial state s_0, and a failure state s_f.

Initially, Morgana selects a state $(\varepsilon, \mathrm{Out}(\bar{x}))$ and the game remains at the root node. That is, the transition function ρ maps (s_0, τ) to the disjunction of all $(0, (\varepsilon, \mathrm{Out}(\bar{x})))$, for all τ. In a state $(\varepsilon, \mathrm{Out}(\bar{x}))$ Arthur then checks whether the head of the rule instantiation of the root node is indeed $\mathrm{Out}(\bar{x})$. If this is *not* the case, the automaton enters the failure state. Formally, $\rho((\varepsilon, \mathrm{Out}(\bar{x})), \tau) = (0, s_f)$, for all τ with $\mathsf{head}(\tau) \neq \mathrm{Out}(\bar{x})$. Similarly, $\rho((\tau', A), \tau) = (0, s_f)$ if $\mathsf{head}(\tau) \neq A$ or τ and τ' witness a violation of Property (b).

[15] It is not hard to see that the construction can easily be modified to yield a non-deterministic bottom-up tree automaton. For proving Theorem 4.2.20 it would even be acceptable to convert it into a non-deterministic bottom-up automaton using the black-box method of Cosmadakis et al. [Cos+88, Theorem A.2]. This leads to an exponential blow-up, but the resulting automaton is not (asymptotically) larger than $\mathbb{A}_{Q \sqsubseteq Q'}$.

For any other (τ', A) and τ, Arthur selects an intensional atom $B \in \mathsf{body}(\tau)$. As described above, Morgana then chooses a child node. Her choices can be expressed as $\varphi_B = \bigvee_{1 \leq i \leq |\mathsf{body}(\tau)|}(i, (\tau, B))$. Overall, $\rho((\tau', A), \tau)$ is the conjunction of all the φ_B. We note that, if $\mathsf{body}(\tau)$ does not contain any intensional atoms, then the game arrived at a leaf and we have $\rho((\tau', B), \tau) = \top$. Hence, Morgana wins in this case, as intended.

Of course, $\rho(s_f, \tau) = (0, s_f)$, for all τ, and consequently, Morgana cannot win any more, once Arthur asserts a violation.

We observe that the size of $\mathbb{A}_Q$ is within the claimed size bounds. Indeed, the number of states is polynomial in the size of the alphabet Γ_P and the maximal size of a rule in P. This is also the case for the size of any formula in the image of ρ, and hence, the transition function. □

In the remainder of this section we argue that the following result, which states the refined complexity bounds for $\mathbb{A}_{Q \sqsubseteq Q'}$, does indeed hold.

Proposition 4.3.6 [BKR15a, Implicit in the proof of Theorem 6]**.** *For each Datalog query $Q = (P, \mathrm{Out})$ and frontier-guarded Datalog query $Q' = (P', \mathrm{Out})$ one can construct an alternating two-way tree automaton $\mathbb{A}_{Q \sqsubseteq Q'}$ for $\mathcal{T}_{Q \sqsubseteq Q'}$. It has size (and can be constructed in time)*

- *exponential in the number of variables in P and P';*
- *exponential in the maximal size of a rule of P and P'; and*
- *polynomial in the number of rules of P and P'.*

Bourhis et al. [BKR15a] actually derive Proposition 4.3.6 from their proof of an analogous result for the more general *Guarded Queries* (which yields a triple exponential time upper bound for the containment problem for these queries). We present here a direct construction of the automaton $\mathbb{A}_{Q \sqsubseteq Q'}$ for frontier-guarded Datalog queries, thereby avoiding some technical complications caused by the more general queries.

In the following, we first provide a high-level description of the semantics of $\mathbb{A}_{Q \sqsubseteq Q'}$ in terms of the underlying two-player games on symbolic proof trees T (for P).

The Game. Recall that Morgana is the existential player, and Arthur the universal player. Morgana's task is to show that T is in $\mathcal{T}_{Q \sqsubseteq Q'}$, while Arthur's task is to challenge her proof. Initially, Arthur can choose to play the game $\mathbb{A}_Q \times T$, which he wins if and only if $T \notin \mathcal{T}_Q$. For this purpose, $\mathbb{A}_{Q \sqsubseteq Q'}$ has all the states (and transitions) of $\mathbb{A}_Q$. If Arthur decides not to play $\mathbb{A}_Q \times T$, it is safe to assume that indeed $T \in \mathcal{T}_Q$ holds. In that case, Morgana builds up a tree $T' \in \mathcal{T}_{Q'}$ with the intention to prove $\mathsf{q}(T) \sqsubseteq \mathsf{q}(T')$. During the game, both players can traverse the tree T in a two-way fashion but T' is only "traversed" top-down along a single path. To prove $\mathsf{q}(T) \sqsubseteq \mathsf{q}(T')$ Morgana attempts to establish the existence of a homomorphism from $\mathsf{q}(T')$ to $\mathsf{q}(T)$.[16]

After each round, there is a current node v and a pair $(h, \mathcal{A})$ where h is a partial mapping $h\colon \mathsf{vars}(\tau') \rightharpoonup \{x_1, \ldots, x_{2|\mathsf{vars}(P)|}\}$ and $\mathcal{A} \subseteq \mathsf{body}(\tau')$, for some rule instantiation τ'

[16] Recall that $\mathsf{q}(T) \sqsubseteq \mathsf{q}(T')$ holds if and only if there is homomorphism from $\mathsf{q}(T')$ to $\mathsf{q}(T)$ [cf. CM77, Proof of Lemma 13].

of P' with $\mathsf{vars}(\tau') \subseteq \{y_1, \ldots, y_{2|\mathsf{vars}(P')|}\}$. Intuitively, τ' is the label of the (implicit) current node v' of T', $\mathcal{A}$ consists of the atoms of $\mathsf{q}(T')$ that are induced by τ' and still have to be mapped into $\mathsf{q}(T)$, and h is a partial homomorphism that has to be extended by Morgana to do so. Although, let us emphasize that $\mathcal{A}$ might contain intensional atoms. If Arthur demands from Morgana to prove that such an intensional atom $A \in \mathcal{A}$ can be "mapped" into $\mathsf{q}(T)$, Morgana replaces $\mathcal{A}$ with the body of a rule instantiation for A. In other words, she appends a node for A to v' and the game descends to this new node in T'.

To be exact, the game is played as follows. In the first round, Morgana chooses a rule instantiation τ' of a rule in P' and a (partial) mapping h which maps the head of τ' to the head of $\mathsf{root}(T)$, i.e. the head of $\mathsf{q}(T)$. Consequently, $\mathcal{A} = \mathsf{body}(\tau')$. In all further rounds, there are three possible cases.

Case 1: $|\mathcal{A}| > 1$. If there is a subset $\mathcal{B} \subsetneq \mathcal{A}$ such that $\mathsf{vars}(\mathcal{B}) \cap \mathsf{vars}(\mathcal{A} \setminus \mathcal{B}) \subseteq \mathsf{dom}(h)$, i.e. if h is defined for all variables occurring in $\mathcal{B}$ and its complement, Arthur chooses $\mathcal{B}$ or $\mathcal{A} \setminus \mathcal{B}$ and the new state is $(h', \mathcal{B})$ or $(h', \mathcal{A} \setminus \mathcal{B})$ where h' is the restriction of h to $\mathsf{vars}(\mathcal{B})$ or $\mathsf{vars}(\mathcal{A} \setminus \mathcal{B})$, respectively.

Otherwise, Morgana has to extend the mapping h to another variable x in $\mathcal{A}$ as follows. She has to choose a (possibly new) current node w of the input tree T such that all nodes on the path from v to w contain all variables[17] of $\mathsf{dom}(h) \cap \mathsf{vars}(\mathcal{A})$. Then she has to choose a variable appearing in the label of w for $h(x)$.

Case 2: $|\mathcal{A}| = 1$ and $\mathsf{vars}(\mathcal{A}) \not\subseteq \mathsf{dom}(h)$. Morgana needs to extend the mapping h to all variables in the remaining atom $A \in \mathcal{A}$. To this end, she has to choose a node w such that all nodes on the path from v to w contain all variables of $\mathsf{dom}(h) \cap \mathsf{vars}(A)$. Then she has to extend h such that it maps all remaining variables of $\mathcal{A}$ to variables at w.

Case 3: $|\mathcal{A}| = 1$ and $\mathsf{vars}(\mathcal{A}) \subseteq \mathsf{dom}(h)$. Morgana chooses a node w such that all nodes on the path from v to w contain all variables of $h(A)$ where A is the remaining atom in $\mathcal{A}$.

Case 3.1: A is extensional. Morgana wins if the image $h(A)$ is in the label of w, otherwise she loses.

Case 3.2: A is intensional. Morgana has to choose a rule instantiation τ' of a rule in P' whose head equals A and a guard atom B of τ'. Then she extends, if necessary, h to all variables occurring in B (but not in A). If $h(B)$ is *not* at w, Morgana loses. Otherwise, the game continues at w with the homomorphism h' that is the restriction of h to all variables in B, and the set $\mathsf{body}(\tau')$ of atoms.

Note that Morgana constructs the symbolic proof tree T' by choosing new rule instantiations and Arthur determines a path through T' by, eventually, choosing an atom of a rule instantiation. Before we move on to the correctness and formal construction, we illustrate the game by means of an example.

[17] Recall that non-connected occurrences of a variable in T correspond to different variables in $\mathsf{q}(T)$.

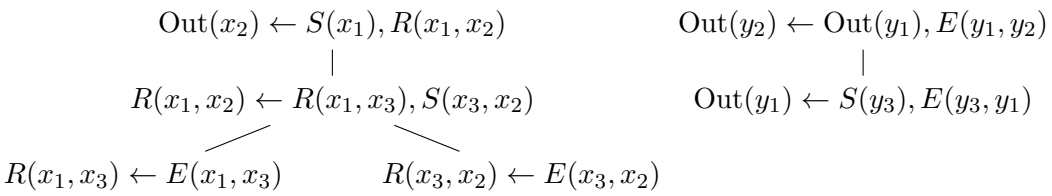

Figure 4.3: Symbolic proof trees T (on the left) and T' (on the right) discussed in Example 4.3.7.

Example 4.3.7. Consider the Datalog query $Q = (P, \mathrm{Out})$ from Example 4.3.2 and the frontier-guarded Datalog query $Q' = (P', \mathrm{Out})$ where P' consists of the following rules.

$$\mathrm{Out}(y_2) \leftarrow S(y_1), E(y_1, y_2) \qquad \mathrm{Out}(y_2) \leftarrow \mathrm{Out}(y_1), E(y_1, y_2)$$

Furthermore, consider the symbolic proof trees T and T' for P and P' depicted in Figure 4.3. Observe that $\mathsf{q}(T)$ is $\mathrm{Out}(x_2) \leftarrow S(x_1), E(x_1, x_3), E(x_3, x_2)$, and $\mathsf{q}(T')$ is $\mathrm{Out}(y_2) \leftarrow E(y_1, y_2), S(y_3), E(y_3, y_1)$. Clearly these queries are equivalent and Morgana should win the game $\mathbb{A}_{Q \sqsubseteq Q'} \times T$, if she plays with the tree T' in mind, thanks to $\mathsf{q}(T) \sqsubseteq \mathsf{q}(T')$.

Indeed, she has a winning strategy, which we describe in the following. Initially, Morgana chooses the rule instantiation $\mathrm{Out}(y_2) \leftarrow \mathrm{Out}(y_1), E(y_1, y_2)$ and the mapping $h = \{y_2 \mapsto x_2\}$. Thus, $\mathcal{A} = \{\mathrm{Out}(y_1), E(y_1, y_2)\}$.

In the second round, Morgana has to extend h, since Case 1 applies and there is no atom in $\mathcal{A}$ such that h is defined for all variables in that atom. To this end, she can move to the rightmost leaf of T and extend h by $y_1 \mapsto x_3$.

In the next round, Case 1 applies again but this time Arthur has to choose a (strict) subset of $\mathcal{A}$. If he chooses $\{E(y_1, y_2)\}$ he looses because then Case 3.1 applies and $h(E(y_1, y_2)) = E(x_3, x_2)$ is in the label of the current node, i.e. the rightmost leaf of T. The other choice is the set $\{\mathrm{Out}(y_1)\}$. The new mapping is then $\{y_1 \mapsto x_3\}$.

Now Case 3.2 applies and Morgana can move to the leftmost child, choose $\mathrm{Out}(y_1) \leftarrow S(y_3), E(y_3, y_1)$ as τ', guard atom $E(y_3, y_1)$, and extend the mapping by $y_3 \mapsto x_1$. The new atom set is $\{S(y_3), E(y_3, y_1)\}$. She does not loose because $E(y_3, y_1)$ is mapped to $E(x_1, x_3)$ which occurs in the label of the leftmost child.

Arthur can now choose either $\{S(y_3)\}$ or $\{E(y_3, y_1)\}$ according to Case 1. In both cases Morgana wins because Case 3.1 applies: If Arthur chose $\{E(y_3, y_1)\}$ she wins immediately. Otherwise, she can move to the root node of T, since $S(y_3)$ is mapped to $S(x_1)$. ◁

Correctness. Let us briefly discuss the correctness. Suppose Morgana has a winning strategy for the game $\mathbb{A}_{Q \sqsubseteq Q'} \times T$. Then there is a symbolic proof tree T' for Q' such that Morgana can map every atom in any label of T' to an atom occurring in T, because Arthur alone determines the path "traversed" in T'. Moreover, if a variable x occurs in two atoms A, B in T', and these atoms occur in labels of x-connected nodes, Morgana has to map x for these occurrences of A and B to the same occurrence of a variable in T.

This is because Arthur will separate those two atoms only, if the mapping is defined for x, according to Case 1. But then the images of A and B are also connected by the image of x in T. Thus, by renaming variables in T and T' in the same manner as in the definition of $\mathsf{q}(T)$ and $\mathsf{q}(T')$ yields a homomorphism from $\mathsf{q}(T')$ to $\mathsf{q}(T)$. Therefore, we have $\mathsf{q}(T) \sqsubseteq \mathsf{q}(T')$.

The other way around, any homomorphism from $\mathsf{q}(T')$ to $\mathsf{q}(T)$ for some T' induces a winning strategy for Morgana. A bit more precisely, Morgana maps an occurrence of a variable x in T' to an occurrence of a variable y in T, if x' is mapped to y' by the homomorphism, where x' and y' are the variables in $\mathsf{q}(T')$ and $\mathsf{q}(T)$ corresponding to the occurrences of x and y, respectively.

For Cases 1 and 2 it is crucial that Morgana can map all variables occurring in a rule instantiation of T' to variables in T without leaving the connected component spanned by the images of already mapped variables. This is indeed ensured because Q' is frontier-guarded: Every variable in such a rule instantiation τ that occurs in an intensional atom $R(\bar{x})$ in $\mathsf{body}(\tau)$, also occurs in the guard atom of the label for the child node for $R(\bar{x})$. Therefore, all variables of τ occur in extensional atoms connected with respect to all of these variables in T'. This is then also the case for the corresponding atoms (and variables) in $\mathsf{q}(T')$. The homomorphism guarantees that they can be mapped to $\mathsf{q}(T)$, and thus, to connected occurrences in T.

The Automaton. It remains to show that the game defines indeed the semantics of an alternating two-way tree automaton $\mathbb{A}_{Q \sqsubseteq Q'}$ whose size is polynomial in the number of rules and at most exponential in the maximal size of a rule and the number of variables.

Every pair $(h, \mathcal{A})$ constitutes a state of $\mathbb{A}_{Q \sqsubseteq Q'}$ where h is a partial mapping $h \colon \mathsf{vars}(\tau') \rightharpoonup \{x_1, \ldots, x_{2|\mathsf{vars}(P)|}\}$ and $\mathcal{A} \subseteq \mathsf{body}(\tau')$, for some rule instantiation τ' of P' with $\mathsf{vars}(\tau') \subseteq \{y_1, \ldots, y_{2|\mathsf{vars}(P')|}\}$. Furthermore, $\mathbb{A}_{Q \sqsubseteq Q'}$ has a distinct initial state s_0, a distinct "failure" state s_f, and, as mentioned earlier, all states (and transitions) of $\mathbb{A}_Q$.

The number of partial mapping h is exponential in the number of variables occurring in Q and Q'. The number of possible sets $\mathcal{A}$ is exponential in the maximal size of a rule in Q', and polynomial in the number of rules of Q'. Therefore, the number of possible pairs $(h, \mathcal{A})$ is polynomial in the number of rules and at most exponential in the number of variables and the maximal size of a rule. The same upper bound applies to the states inherited from $\mathbb{A}_Q$, thanks to Proposition 4.3.5.

In the following we define the transition function ρ step by step, following the game semantics. For a set Φ of formulas we write $\bigwedge \Phi$ and $\bigvee \Phi$ for the conjunction and disjunction of all formulas in Φ, respectively. Given a mapping h and set $\mathcal{B}$ we write $h_{|\mathcal{B}}$ for the restriction of h to $\mathsf{vars}(\mathcal{B})$. Further, we write $h[x \mapsto y]$ for the mapping with domain $\mathsf{dom}(h) \cup \{x\}$, $h[x \mapsto y](x) = y$, and $h[x \mapsto y](x') = h(x')$ for all $x' \in \mathsf{dom}(h)$.

Recall that, in the first round, Arthur can decide to play the game $\mathbb{A}_Q \times T$ to prove that the input tree T is not in $\mathcal{T}_Q$, and hence, not in $\mathcal{T}_{Q \sqsubseteq Q'}$. If he does not, Morgana has to choose a rule instantiation τ' of a rule in P' and a partial mapping such that the image of $\mathsf{head}(\tau')$ is $\mathsf{head}(\mathsf{root}(T))$. This is modelled by the transitions

$$\rho(s_0, \tau) = (0, s_0') \wedge \bigvee \big\{(0, (h, \mathsf{body}(\tau'))) \mid h(\mathsf{head}(\tau')) = \mathsf{head}(\tau)\big\}$$

for all τ. Here s_0' is the initial state of $\mathbb{A}_Q$, and it is implicit that the τ' are rule instantiations, the h are defined for all variables in $\mathsf{head}(\tau')$, and the pairs $(h, \mathsf{body}(\tau'))$ constitute states.[18]

Next, we model Morgana's ability to "move" from a node v to another node w, as long as all variables in $h(\mathsf{dom}(h) \cap \mathcal{A})$ occur in all nodes along the path. To this end, we observe the following. Arthur cannot "traverse" T himself, nor can he extend h. He can only force Morgana to do so. Thus, a situation where *not* all variables in $h(\mathsf{dom}(h) \cap \mathsf{vars}(\mathcal{A}))$ occur at the current node, can only arise if Morgana "cheats" – by violating here movement restriction or extending h such that $h(x)$ does not occur at the current node, for some variable x. Therefore, by setting $\rho((h, \mathcal{A}), \tau) = (0, s_f)$ for every state $(h, \mathcal{A})$ and rule instantiation τ with $h(\mathsf{dom}(h) \cap \mathsf{vars}(\mathcal{A})) \not\subseteq \mathsf{vars}(\tau)$ ensures that Morgana will not "cheat", because then she will lose: Recall that s_f is the "failure" state and, of course, we have $\rho(s_f, \tau) = (0, s_f)$ for all τ. With this in mind, we can simply use the following kind of auxiliary formulas to model Morgana's movements.

$$\varphi_{\texttt{move}}((h, \mathcal{A}), \tau) = \bigvee \big\{(i, (h, \mathcal{A})) \mid i \in \{-1, 1, \dots, |\mathsf{body}(\tau)|\}\big\}$$

Of course, these transitions are only defined for states $(h, \mathcal{A})$ and rule instantiations τ with $h(\mathsf{dom}(h) \cap \mathsf{vars}(\mathcal{A})) \subseteq \mathsf{vars}(\tau)$. We will call such combinations of states $(h, \mathcal{A})$ and rule instantiations τ *legal*.

Further on, the moves of Arthur in Case 1 are modelled by the transitions

$$\rho((h, \mathcal{A}), \tau) = \bigwedge \big\{(0, (h_{|\mathcal{B}}, \mathcal{B})) \mid \mathcal{B} \subsetneq \mathcal{A}, \mathsf{vars}(\mathcal{B}) \cap \mathsf{vars}(\mathcal{A} \setminus \mathcal{B}) \subseteq \mathsf{dom}(h)\big\}$$

which are defined for all legal $(h, \mathcal{A})$ and τ such that there is at least one set $\mathcal{B} \subsetneq \mathcal{A}$ with $\mathsf{vars}(\mathcal{B}) \cap \mathsf{vars}(\mathcal{A} \setminus \mathcal{B}) \subseteq \mathsf{dom}(h)$. Note that these transitions also allow Arthur to pick $\mathcal{A} \setminus \mathcal{B}$ instead of $\mathcal{B}$.

Morgana's moves in Case 1 are defined for all remaining legal $(h, \mathcal{A})$ and τ, namely by

$$\rho((h, \mathcal{A}), \tau) = \varphi_{\texttt{move}}((h, \mathcal{A}), \tau) \vee \bigvee \big\{(0, (h[x \mapsto y], \mathcal{A})) \mid x \in \mathsf{vars}(\mathcal{A}) \setminus \mathsf{dom}(h), y \in \mathsf{vars}(\tau)\big\}.$$

The transitions for Case 2 are similar. That is,

$$\rho((h, \{A\}), \tau) = \varphi_{\texttt{move}}((h, \{A\}), \tau) \vee \bigvee \big\{(0, h', \{A\})) \mid \mathsf{dom}(h') = \mathsf{vars}(A), h'(x) = h(x) \text{ for all } x \in \mathsf{dom}(h)\big\}$$

for all legal $(h, \{A\})$ and τ with $\mathsf{vars}(A) \not\subseteq \mathsf{dom}(h)$.

It remains to provide transitions for Cases 3.1 and 3.2. That is, for legal $(h, \{A\})$ and τ with $\mathsf{dom}(h) = \mathsf{vars}(A)$. We first consider the case that A is extensional, i.e. Case 3.1 applies. If $h(A) \in \mathsf{body}(\tau)$ then Morgana wins. Consequently, $\rho((h, \{A\}), \tau) = \top$ for such $(h, \{A\})$ and τ. Otherwise, Morgana can move to another node:

$$\rho((h, \{A\}), \tau) = \varphi_{\texttt{move}}((h, \{A\}), \tau).$$

[18] The intention is to avoid clutter and improve readability.

Finally, we model Case 3.2 as follows.

$$\rho((h,\{A\}),\tau) = \varphi_{\texttt{move}}((h,\{A\}),\tau) \vee \bigvee \big\{(0,(h',\mathsf{body}(\tau'))) \mid \mathsf{head}(\tau') = A, h'(x) = h(x) \text{ for all } x \in \mathsf{dom}(h), \text{ and there is a guard atom } B \text{ of } \tau' \text{ such that } \mathsf{dom}(h') = \mathsf{vars}(B), h'(B) \in \mathsf{body}(\tau)\big\}$$

Clearly, all transition formulas have size polynomial in the number of states. Thus, altogether, the size of $\mathbb{A}_{Q \sqsubseteq Q'}$ is bounded as stated in Proposition 4.3.6. We can conclude that Proposition 4.3.6, and hence, Theorem 4.2.20 hold.

4.4 Parallel-Boundedness

In this section, we study parallel-boundedness of Datalog queries and our distributed evaluation strategies.

Definition 4.4.1 (Parallel-Boundedness)**.** A Datalog query Q is *parallel-bounded* w.r.t. a family $\mathcal{F}$ of policy pairs if there is an integer $r > 0$ such that, for all policy pairs $(\delta, \gamma) \in \mathcal{F}$ and all distributed databases $\mathcal{D}$ complying with δ, *no* new *output facts* are derived on any server after r rounds in the distributed evaluation of Q over $\mathcal{D}$ induced by γ.

We note that our definition of parallel-boundedness w.r.t. families of policy pairs embraces a uniform bound r for all policy pairs. At the same time, our definition is more generous than the one of Ketsman et al. [KAK20, Definition 5] regarding the derivation of facts: Our notion only demands that no new output facts are derived after r rounds, while their notion requires that no new facts whatsoever are derived any more.

We study parallel-boundedness only for queries Q and families $\mathcal{F}$ for which Q is parallel-correct w.r.t. $\mathcal{F}$. Parallel-boundedness for queries that are *not* parallel-correct does not appear meaningful. For classes $\mathbb{Q}$ of Datalog queries, and classes $\mathbb{F}$ of families of policy pairs, we write PBound($\mathbb{Q}$, $\mathbb{F}$) for the following decision problem.

— PBound($\mathbb{Q}$, $\mathbb{F}$) —

Given: Datalog query $Q \in \mathbb{Q}$ and family $\mathcal{F} \in \mathbb{F}$ of policy pairs such that Q is parallel-correct w.r.t. $\mathcal{F}$

Question: Is Q parallel-bounded w.r.t. $\mathcal{F}$?

Our objective in this section is to obtain decision procedures for parallel-boundedness. As for parallel-correctness, we focus on classes of frontier-guarded Datalog queries and sets of data-moving distribution constraints that enjoy the polynomial communication property. In particular, we will obtain the following result as well as a similar result for parallel-boundedness in the non-transitive communication setting.

Theorem 4.4.2. PBound(FGDL, Hash-MConstraints) *is* 2ExpTime*-complete.*

Similarly to the lower bound proof for parallel-correctness, cf. Theorem 4.2.28, we will prove the lower bound of Theorem 4.4.2 by a reduction from the containment problem.

Our proof for the upper bound builds upon the ingredients for deciding parallel-correctness. This involves, in particular, the Datalog queries from Lemma 4.2.22 which simulate distributed evaluations over scattered databases, and the construction of the automaton $\mathbb{A}_{Q \sqsubseteq Q'}$ in Section 4.3 for testing containment.

The first step towards the upper bound is therefore to show that, for deciding parallel-boundedness, it suffices to consider scattered databases.

Lemma 4.4.3. *Let Q be a Datalog query, Z be a hash policy scheme, and Σ be a set of data-moving distribution constraints such that Q is parallel-correct w.r.t. $\mathcal{F}(Z, \Sigma)$. Then Q is parallel-bounded w.r.t. $\mathcal{F}(Z, \Sigma)$ if and only if there is an integer $r > 0$ such that for all global databases G there is a policy pair $(\delta, \gamma) \in \mathcal{F}$ such that δ scatters G and* no *new* output facts *are derived on any server after r rounds in the distributed evaluation of Q over $\delta(G)$ induced by γ.*

Proof. For the only-if direction suppose that Q is parallel-correct w.r.t. $\mathcal{F}(Z, \Sigma)$ and let G be a global database. Thanks to Lemma 4.2.7 there is a tuple H of hash functions over a network $\mathcal{N}$ such that $\delta_{Z,H}$ scatters G. Since Q is parallel-bounded w.r.t. $\mathcal{F}(Z, \Sigma)$ there is an integer r such that, for all $(\delta, \gamma) \in \mathcal{F}(Z, \Sigma)$ and all distributed databases $\mathcal{D}$ that comply with δ, no new output facts are derived on any server after r rounds in the distributed evaluation of Q over $\mathcal{D}$ induced by γ. In particular, this also applies to $(\delta_{Z,H}, \gamma_{\Sigma,\mathcal{N}})$ and $\delta_{Z,H}(G)$.

For the converse, let $r > 0$ be an integer such that for all global databases G there is a policy pair $(\delta, \gamma) \in \mathcal{F}$ such that δ scatters G and *no* new *output facts* are derived on any server after r rounds in the distributed evaluation of Q over $\delta(G)$ induced by γ.

Consider now any policy pair $(\delta', \gamma') \in \mathcal{F}(Z, \Sigma)$ and distributed database $\mathcal{D} = (G, \mathcal{I})$ that complies with δ'. We show that r is an upper bound for the number of rounds required to derive all output facts in the distributed evaluation of Q over $\mathcal{D}$ induced by γ'.

By assumption there is a policy pair $(\delta, \gamma) \in \mathcal{F}(Z, \Sigma)$ such that δ scatters G, and *no* new *output facts* are derived on any server after r rounds in the distributed evaluation of Q over $\delta(G)$ induced by γ. Since Q is parallel-correct w.r.t. $\mathcal{F}(Z, \Sigma)$, we have that $[Q, \gamma](\delta(G)) = [Q, \gamma'](\mathcal{D})$.

Let $\mathrm{Out}(\bar{a}) \in [Q, \gamma'](\mathcal{D})$. Then $\mathrm{Out}(\bar{a}) \in [Q, \gamma](\delta(G))$, and by assumption, $\mathrm{Out}(\bar{a})$ is derived in at most r rounds in the distributed evaluation of Q over $\delta(G)$ induced by γ. Due to Lemma 4.2.10, $\mathcal{D}$ covers $\delta(G)$. It follows that $\mathrm{Out}(\bar{a})$ is derived in at most r rounds in the distributed evaluation of Q over $\mathcal{D}$ induced by γ', thanks to Lemma 4.2.11. □

In the following, let Q be a frontier-guarded Datalog query, Z be a hash policy scheme, and Σ be a set of data-moving distribution constraints, where Q and Σ originate from classes enjoying the polynomial communication property. Thanks to Lemma 4.4.3 it suffices to consider scattered databases for testing parallel-boundedness. Recall that we can construct a frontier-guarded Datalog query Q' which simulates the distributed evaluation of Q over scattered databases, cf. Lemma 4.2.22. Deciding parallel-correctness

then boils down to testing whether $Q \sqsubseteq Q'$ holds. This is in turn done by testing whether each symbolic proof tree in the tree language $\mathcal{T}_Q$ can be "captured" by a tree in $\mathcal{T}_{Q'}$ using the automata $\mathbb{A}_Q$ and $\mathbb{A}_{Q \sqsubseteq Q'}$ constructed in Section 4.3.

For deciding parallel-boundedness we will extend and integrate these steps more tightly. In a nutshell, the idea is to assign a "communication cost" to each tree T' in $\mathcal{T}_{Q'}$ that reflects how many rounds are required to derive facts "covered" by $\mathfrak{q}(T')$. The question then becomes whether all trees in $\mathcal{T}_Q$ can be "captured" by a subset of trees in $\mathcal{T}_{Q'}$ whose costs are bounded.

To this end, we first discuss how "communication costs" can be assigned to proof trees over a network $\mathcal{N}$ for a fact $R(\bar{a})@k$ with respect to Q and Σ. Indeed, this is straightforward, thanks to Lemma 4.2.18. We assign a cost of 1 to nodes witnessed by distribution constraints and 0 to nodes witnessed by rules (and leaves). The cost of a path is then the sum of all costs (of nodes) along this path, and the cost assigned to the tree is the maximal cost over all root-to-leaf paths.

We now turn to symbolic proof trees. Recall that some rules of Q' incorporate distribution constraints. Specifically, a body of such a rule might contain both, atoms that originate from a distribution constraint, and atoms originating from a Datalog rule. The former represent communication and the latter computation. Thus, assigning cost to a symbolic proof tree for Q' node-wise is not as straightforward as for proof trees with respect to Q. In fact, the following example suggests that assigning costs to edges is more suitable for symbolic proof trees.

Example 4.4.4. Consider the frontier-guarded Datalog query $Q = (P, \mathrm{Out})$ where P consists of the following three rules.

$$\mathrm{Out}(x) \leftarrow \mathrm{Out}(y), E(x,y), N(x) \quad \mathrm{Out}(x) \leftarrow E(z,x), T(x), N(x) \quad N(x) \leftarrow M(x)$$

We can understand an extensional relation E as the edge relation of a directed graph. The query Q then asks for all nodes from which there is a path along M-labelled nodes to a T-labelled node. Note that N just serves as a copy of M which can be communicated.

Let $Z = \{(E, 1, ()), (T, 1, ()), (M, 2, ())\}$ and Σ be the set consisting of the following two data-moving distribution constraints.

$$E(x,y)@\lambda, N(x)@\kappa \to N(x)@\lambda \quad E(y,x)@\lambda, N(x)@\kappa \to N(x)@\lambda$$

Then Q is parallel-correct and parallel-bounded w.r.t. $\mathcal{F}(Z, \Sigma)$. Indeed, Z asserts that there is a server on which all E- and T-facts reside, and all M-facts are sent to this server in the first communication phase due to the distribution constraints. In particular, every fact, that can be derived at all, can be derived after at most two rounds.

This is also reflected by the proof trees with respect to P, Σ, and a scattered database. Their shape is illustrated in Figure 4.4. Clearly, the cumulative cost along each root-to-leaf path is at most 1, which corresponds to the number of rounds required in the distributed evaluation.

Figure 4.5 depicts the shape of symbolic proof trees with respect to the query Q' which simulates the distributed evaluation of Q induced by Σ. Here costs are assigned to

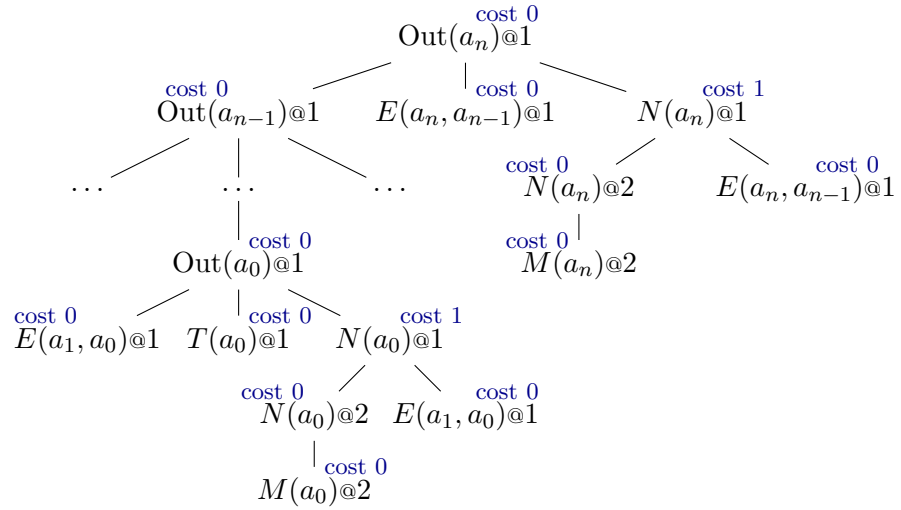

Figure 4.4: Shape of a proof tree with respect to the query and the set of data-moving distribution constraints from Example 4.4.4. Nodes which are witnessed by a distribution constraint have cost 1, and every other node has cost 0.

edges: An edge (v, w) has cost 1 if the occurrence of $\mathsf{head}(w)$ in $\mathsf{body}(v)$ originates from a distribution constraint. Otherwise, it has cost 0. Observe that this cost assignment yields the same property as before: The cumulative cost of each root-to-leaf path is at most 1, and hence, corresponds to the number of rounds. Moreover, if we were to assign a cost of 1 to each node labelled with a rule obtained by inlining constraints, then the cost of a path would not be bounded by the number of rounds. In fact, up to a constant, the cost would be equal the height of the tree. ◁

We now make our cost assignments more precise. A 0-1-*cost function* for a symbolic proof tree T is a function c which maps edges (v, w) of T into $\{0, 1\}$. Such a 0-1-cost function c extends naturally to paths, symbolic proof trees T, and sets of symbolic proof trees $\mathcal{T}$ as follows. The cost of a path $\pi = v_0 v_1 \dots v_n$ is $\mathsf{c}(\pi) = \sum_{i=1}^{n} \mathsf{c}(v_{i-1}, v_i)$. The cost $\mathsf{c}(T)$ of a tree T is the maximal cost $\mathsf{c}(\pi)$, where π ranges over all root-to-leaf paths of T. Finally, for a set $\mathcal{T}$ of symbolic proof trees, we set $\mathsf{c}(\mathcal{T}) = \sup\{\mathsf{c}(T) \mid T \in \mathcal{T}\}$.

Recall that $\mathcal{T}_Q$ is the set of all symbolic proof trees for $Q = (P, \mathrm{Out})$ with variables from $\{x_1, \dots, x_{2|\mathsf{vars}(P)|}\}$. Given a 0-1-cost function c, we write

$$\mathcal{T}^{\min}_{Q,\mathsf{c}} = \{T \in \mathcal{T}_Q \mid \text{ there is no } T' \in \mathcal{T}_Q \text{ such that } \mathsf{q}(T) \sqsubseteq \mathsf{q}(T') \text{ and } \mathsf{c}(T') < \mathsf{c}(T)\}$$

for the set of symbolic proof trees with minimal costs (among the trees in $\mathcal{T}_Q$). Thanks to the following refinement of Lemma 4.3.3 and Lemma 4.3.4, we can restrict our attention to trees in $\mathcal{T}^{\min}_{Q,\mathsf{c}}$.

Lemma 4.4.5. *For each Datalog query Q and 0-1-cost function c defined on the set $\mathcal{T}_Q$, the equivalence $Q \equiv \mathsf{q}(\mathcal{T}^{\min}_{Q,\mathsf{c}})$ holds.*

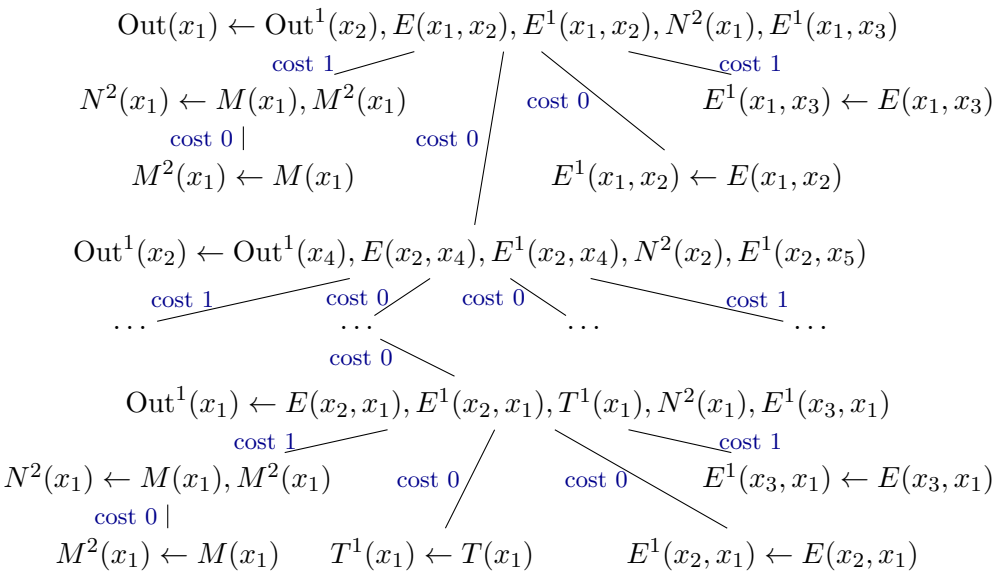

Figure 4.5: Shape of symbolic proof trees for the query Q' from Example 4.4.4. An edge (v, w) has cost 1 if the occurrence of $\mathsf{head}(w)$ in $\mathsf{body}(v)$ originates from a distribution constraint. Otherwise, it has cost 0.

Proof. Thanks to Lemma 4.3.3 and Lemma 4.3.4 we have that $Q \equiv \mathsf{q}(\mathcal{T}_Q)$. Hence, it suffices to show $\mathsf{q}(\mathcal{T}_Q) \equiv \mathsf{q}(\mathcal{T}^{\min}_{Q,\mathsf{c}})$. The inclusion from right to left is immediate by definition.

Let $T_0 \in \mathcal{T}_Q \setminus \mathcal{T}^{\min}_{Q,\mathsf{c}}$. With $T_0 \notin \mathcal{T}^{\min}_{Q,\mathsf{c}}$ it follows by definition of $\mathcal{T}^{\min}_{Q,\mathsf{c}}$ that there is a tree $T_1 \in \mathcal{T}_Q$ such that $\mathsf{q}(T_0) \sqsubseteq \mathsf{q}(T_1)$ and $\mathsf{c}(T_1) < \mathsf{c}(T_0)$. Thus, it suffices to show that $\mathsf{q}(T_1) \sqsubseteq \mathsf{q}(\mathcal{T}^{\min}_{Q,\mathsf{c}})$, since then $\mathsf{q}(T_0) \sqsubseteq \mathsf{q}(T_1) \sqsubseteq \mathsf{q}(\mathcal{T}^{\min}_{Q,\mathsf{c}})$.

If $T_1 \in \mathcal{T}^{\min}_{Q,\mathsf{c}}$, we are done. Otherwise, we can apply the same reasoning iteratively on T_1, yielding a sequence $T_0, T_1, T_2, \ldots$ with $\mathsf{q}(T_i) \sqsubseteq \mathsf{q}(T_{i+1})$ and $\mathsf{c}(T_{i+1}) < \mathsf{c}(T_i)$ for all $i \geq 0$. Since the natural numbers are well-ordered, this sequence is finite (the costs are strictly decreasing). We conclude that there is a j such that $\mathsf{q}(T_j) \sqsubseteq \mathsf{q}(\mathcal{T}^{\min}_{Q,\mathsf{c}})$ holds, and thus, $\mathsf{q}(T_0) \sqsubseteq \mathsf{q}(\mathcal{T}^{\min}_{Q,\mathsf{c}})$. □

We are now ready to prove that the cost assignment discussed in Example 4.4.4 indeed leads to the desired outcome. That is, the cost $\mathsf{c}(\mathcal{T}^{\min}_{Q',\mathsf{c}})$ is bounded by some d if and only if Q is parallel-bounded. We note that our construction will, in general, *not* yield a bound d for $\mathsf{c}(\mathcal{T}^{\min}_{Q',\mathsf{c}})$ which is also a bound for the number of rounds in the distributed evaluation.

Lemma 4.4.6. *Let $\mathbb{Q}$ be a class of Datalog queries and $\mathbb{C}$ be a class of sets of data-moving distribution constraints that have the polynomial communication property. For every frontier-guarded Datalog query $Q \in \mathbb{Q}$, hash policy scheme Z, and set $\Sigma \in \mathbb{C}$ of data-moving distribution constraints, such that Q is parallel-correct w.r.t. $\mathcal{F}(Z, \Sigma)$, a frontier-guarded Datalog query Q' can be constructed in exponential time and there is a*

0-1-*cost function* c *such that* Q *is parallel-bounded w.r.t.* $\mathcal{F}(Z, \Sigma)$ *if and only if there is a constant* d *such that* $\mathsf{c}(\mathcal{T}^{\min}_{Q',\mathsf{c}}) \leq d$ *holds.*

The number of variables and the length of rules in Q' *is polynomial in the sizes of* Q*,* Z*, and* Σ*; and the number of rules is at most exponential. Furthermore, for each pair* (v, w) *of nodes with* w *being a child of* v*,* $\mathsf{c}(v, w)$ *does solely depend on the labels of* v *and* w*.*

Proof. Let p be the polynomial such that the size of computation-free proof trees with respect to Q and Σ can be bounded by $p(\|Q\|, \|\Sigma\|)$. The frontier-guarded Datalog query $Q' = (P', \mathrm{Out})$ is the same as constructed in the proof for Lemma 4.2.22. That is, Q' simulates the distributed evaluation of Q induced by some $\gamma_{\Sigma,\mathcal{N}}$ over a scattered database.

We distinguish two kinds of rules in P': Those that resulted by the replacement of an intensional atom by the (translated) body of a distribution constraint from Σ^*, which we call *communication prone rules*, and those which did not. Recall that the idea of communication prone rules was precisely to simulate the communication of facts.

We define the 0-1-cost function c on symbolic proof trees T' for Q' as follows. Let v be an inner node labelled $\mathsf{head}(v) \leftarrow \mathsf{body}(v)$ and $w \in \mathsf{children}_{T'}(v)$ be one of its children labelled $\mathsf{head}(w) \leftarrow \mathsf{body}(w)$. Note that, by definition, we have $\mathsf{head}(w) \in \mathsf{body}(v)$. If, in the construction of P', $\mathsf{head}(w) \in \mathsf{body}(v)$ resulted from the replacement of an intensional atom by the (translated) body of a constraint from Σ^*, we define the cost for the edge (v, w) as $\mathsf{c}(v, w) = 1$. Otherwise, if $\mathsf{head}(w) \in \mathsf{body}(v)$ did *not* result from the replacement of intensional atoms by constraint bodies, we set $\mathsf{c}(v, w) = 0$. Note that the cost $\mathsf{c}(v, w)$ depends only on (the labels of) v and w.

It remains to show that Q is parallel-bounded w.r.t. $\mathcal{F}(Z, \Sigma)$ if and only if there is a d such that $\mathsf{c}(\mathcal{T}^{\min}_{Q',\mathsf{c}}) \leq d$.

Suppose that $\mathsf{c}(\mathcal{T}^{\min}_{Q',\mathsf{c}}) \leq d$ holds for some $d \geq 0$. We have to show that Q is parallel-bounded w.r.t. $\mathcal{F}(Z, \Sigma)$.

For this purpose, let G be a global database, and $(\delta, \gamma) \in \mathcal{F}(Z, \Sigma)$ be the policy pair guaranteed by Lemma 4.2.22 such that δ scatters G and $Q'(G) = [Q, \gamma](\delta(G))$. Consider an output fact $\mathrm{Out}(\bar{a}) \in [Q, \gamma](\delta(G))$. We prove that $\mathrm{Out}(\bar{a})$ can be derived within $r = k \cdot p(\|Q\|, \|\Sigma\|)$ rounds. Since $\mathrm{Out}(\bar{a}) \in Q'(G)$, it also is in $\mathsf{q}(\mathcal{T}^{\min}_{Q',\mathsf{c}})(G)$, thanks to Lemma 4.4.5. Hence, there is a symbolic proof tree $T \in \mathcal{T}^{\min}_{Q',\mathsf{c}}$ such that $\mathrm{Out}(\bar{a}) \in \mathsf{q}(T)$.

Following the proof of Lemma 4.3.3, T can be translated into a proof tree for $\mathrm{Out}(\bar{a})$ with respect to Q'. This proof tree has essentially the same structure as T, the only deviation being new leaves labelled with extensional facts. Furthermore, the original label $\mathsf{head}(v) \leftarrow \mathsf{body}(v)$ of a node v becomes the witness for that node in the proof tree for $\mathrm{Out}(\bar{a})$. Proof trees with respect to Q' can in turn be translated into proof trees with respect to Q and Σ^* as in the proof of Lemma 4.2.22. Recall that this translation inserts new inner nodes witnessed by distribution constraints from Σ^*. More precisely, if a node v is witnessed by a rule $\mathsf{head}(v) \leftarrow \mathsf{body}(v)$ and atoms $B_1, \ldots, B_m \in \mathsf{body}(v)$ resulted from replacing an atom A with the (translated) body of a constraint $\sigma^* \in \Sigma^*$, then this replacement is reverted and a new node witnessed by σ^* is inserted as child of v. The children of this new node are the root nodes of (translated) proof trees for $B_1, \ldots, B_m$. As a consequence, on every root-to-leaf path there are at most $\mathsf{c}(T)$ many

nodes witnessed by distribution constraints. Finally, since a proof tree with respect to Q and Σ can be obtained by replacing nodes witnessed by constraints from Σ^* with (partial) computation-free proof trees, there is such a proof tree for some $\mathrm{Out}(\bar{a})@\ell$ where every root-to-leaf path has at most $p(\|Q\|, \|\Sigma\|) \cdot \mathsf{c}(T)$ many nodes witnessed by constraints. We can conclude that $\mathrm{Out}(\bar{a})$ can be derived in at most $p(\|Q\|, \|\Sigma\|) \cdot \mathsf{c}(T) \leq p(\|Q\|, \|\Sigma\|) \cdot d$ many rounds in the distributed evaluation of Q induced by γ over $\delta(G)$, thanks to Lemma 4.2.18. Since δ scatters G and Q is parallel-correct w.r.t. $\mathcal{F}(Z, \Sigma)$, Lemma 4.4.3 implies that Q is parallel-bounded w.r.t. $\mathcal{F}(Z, \Sigma)$.

For the converse, suppose Q is parallel-bounded w.r.t. $\mathcal{F}(Z, \Sigma)$. Let $r > 0$ be the upper bound on the number of rounds required to derive all output facts in distributed evaluations of Q over any distributed database. That is, for all $(\delta, \gamma) \in \mathcal{F}(Z, \Sigma)$ and every distributed database $\mathcal{D}$ that complies with δ, all facts in $[Q, \gamma](\mathcal{D})$ can be derived in at most r rounds. We claim that $\mathsf{c}(\mathcal{T}^{\min}_{Q',\mathsf{c}}) \leq r$ holds.

Let again G be a global database, and $(\delta, \gamma) \in \mathcal{F}(Z, \Sigma)$ be the policy pair guaranteed by Lemma 4.2.22 such that δ scatters G and $Q'(G) = [Q, \gamma](\delta(G))$. For every output fact $\mathrm{Out}(\bar{a}) \in Q'(G)$ there thus is a proof tree for $\mathrm{Out}(\bar{a})@\ell$, where ℓ is some server, with respect to Q and Σ. Since Q is parallel-bounded we can assume that, on each root-to-leaf path of this tree, there are at most r nodes witnessed by a distribution constraint from Σ, again thanks to Lemma 4.2.18. Analogously to the above, following the constructions in Lemmas 4.2.22 and 4.3.3, we can obtain a symbolic proof tree T such that $\mathsf{c}(T) \leq r$ and $\mathrm{Out}(\bar{a}) \in \mathsf{q}(T)$. Particularly, note that computation-free subtrees are essentially merged into a node v witnessed by a rule, the children of the subtree's leaves become children of v, and the respective edges are assigned cost 1.

Let $\mathcal{T}_G$ be the set of symbolic proof trees obtained in this fashion. We have $Q'(G) = \mathsf{q}(\mathcal{T}_G)(G)$. Furthermore, we can assume $\mathcal{T}_G \subseteq \mathcal{T}_Q$ thanks to Lemma 4.3.4. Consider the infinite union $\mathcal{T}$ of all $\mathcal{T}_G$ where G ranges over all global databases. Clearly, $\mathsf{q}(\mathcal{T}) \equiv Q'$ because $\mathcal{T} \subseteq \mathcal{T}_Q$, $\mathsf{q}(\mathcal{T}_Q) \equiv Q'$, and $\mathcal{T}_G(G) = Q'(G)$ for every G. Moreover, we have $\mathsf{c}(\mathcal{T}) \leq r$.

Now, consider any $T \in \mathcal{T}^{\min}_{Q',\mathsf{c}}$. If $T \in \mathcal{T}$ then $\mathsf{c}(T) \leq r$ is immediate. Hence, we consider the case $T \notin \mathcal{T}$. Due to Lemma 4.4.5 and since we have $\mathsf{q}(\mathcal{T}) \equiv Q'$, we also have $\mathcal{T}^{\min}_{Q',\mathsf{c}} \equiv \mathsf{q}(\mathcal{T})$. Thus, there is a $T' \in \mathcal{T}$ such that $\mathsf{q}(T) \sqsubseteq \mathsf{q}(T')$. Since $T \in \mathcal{T}^{\min}_{Q',\mathsf{c}}$ and $T' \in \mathcal{T} \subseteq \mathcal{T}_Q$, we have that $\mathsf{c}(T) \leq \mathsf{c}(T')$. Therefore, we can conclude that $\mathsf{c}(T) \leq r$ holds. □

With Lemma 4.4.6 at hand, deciding parallel-boundedness boils down to testing whether $\mathsf{c}(\mathcal{T}^{\min}_{Q',\mathsf{c}})$ is bounded by some constant d. The starting point for our test procedure is the alternating two-way tree automaton $\mathbb{A}_{Q' \sqsubseteq Q'}$ (*not* $\mathbb{A}_{Q \sqsubseteq Q'}$) from Section 4.3. Recall that an automaton $\mathbb{A}_{Q_1 \sqsubseteq Q_2}$ for queries Q_1 and Q_2 accepts precisely those trees $T \in \mathcal{T}_{Q_1}$ for which there is a tree $T' \in Q_2$ such that $\mathsf{q}(T) \sqsubseteq \mathsf{q}(T')$ holds. Hence, $\mathbb{A}_{Q' \sqsubseteq Q'}$ clearly recognizes just $\mathcal{T}_{Q'}$. Furthermore, T' can always be picked from $\mathcal{T}^{\min}_{Q',\mathsf{c}}$, thanks to Lemma 4.4.5. The idea is to modify $\mathbb{A}_{Q' \sqsubseteq Q'}$ to keep track of the cost $\mathsf{c}(T')$, and then test whether all trees $T \in \mathcal{T}_{Q'}$ can be accepted by picking trees T' whose costs are bounded by some constant d.

Since d is not known in advance, a "cost counter" cannot be incorporated into the

states of $\mathbb{A}_{Q' \sqsubseteq Q'}$. However, there is an extension of tree automata that was invented for exactly this kind of situation: cost automata. For our purposes, we need a cost automaton that has one counter which can be incremented but neither be decremented nor reset to zero. We refer to Section 2.5.2 for the precise definition of the cost automata model, and just recall here that the semantics (including the cost assignment) are defined in terms of a two-player game. The existential player, Morgana, has the objective to minimize the final counter value while the universal player, Arthur, aims to maximize the value. Furthermore, recall that an alternating two-way tree cost automaton $\mathbb{A}$ is *limited* if there is some constant d such that Morgana can prevent the counter value from exceeding d, for all accepted trees.

Proposition 4.4.7. *Let $\mathbb{Q}$ be a class of Datalog queries and $\mathbb{C}$ be a class of sets of data-moving distribution constraints that have the polynomial communication property. For each frontier-guarded Datalog query $Q \in \mathbb{Q}$, hash policy scheme Z, and set $\Sigma \in \mathbb{C}$ of data-moving distribution constraints, such that Q is parallel-correct w.r.t. $\mathcal{F}(Z, \Sigma)$, one can construct an alternating two-way tree cost automaton $\mathbb{A}$ that uses only a single counter, which is never reset, such that $\mathbb{A}$ is limited if and only if Q is parallel-bounded w.r.t. $\mathcal{F}(Z, \Sigma)$. Furthermore, $\mathbb{A}$ has size (and can be constructed in time) exponential in $\|Q\|$, $\|Z\|$, and $\|\Sigma\|$.*

Proof. Let $Q' = (P', \mathrm{Out})$ be the frontier-guarded Datalog query for Q, Z, and Σ guaranteed by Lemma 4.4.6. Furthermore, let c be the associated 0-1-cost function. That is, Q is parallel-bounded w.r.t. $\mathcal{F}(Z, \Sigma)$ if and only if there is a constant d such that $\mathsf{c}(\mathcal{T}^{\min}_{Q', \mathsf{c}}) \leq d$.

To obtain the alternating two-way tree cost automaton $\mathbb{A}$ we modify the automaton $\mathbb{A}_{Q' \sqsubseteq Q'}$ constructed for Proposition 4.3.6 in Section 4.3. Recall that the semantics of $\mathbb{A}_{Q' \sqsubseteq Q'}$ are defined in terms of a two-player game. The task of the existential player, Morgana, in a game on a tree $T \in \mathcal{T}_{Q'}$ is to prove that there is a tree $T' \in \mathcal{T}_{Q'}$ such that $\mathsf{q}(T) \sqsubseteq \mathsf{q}(T')$. The universal player, Arthur, challenges Morgana's proof. Clearly, Morgana can always win by choosing $T' = T$.

We extend the game by assigning a value to each play, namely the cost $\mathsf{c}(T')$ of the tree Morgana chooses. This can be done by increasing the counter of $\mathbb{A}$ each time the game moves from a node v' in T' to a child w' of v with $\mathsf{c}(v, w) = 1$. Since T' is traversed along a single root-to-leaf path determined by Arthur (cf. the construction for Proposition 4.3.6), and Arthur's goal is to maximize the value, this does indeed yield the desired value. These counter increments can easily be added to the transition function because $\mathsf{c}(v, w)$ only depends on the labels of v and w. This may involve that Morgana decides whether to increase the counter or not, and Arthur verifying her choice (this requires encoding the label of the most recent parent node in the states, but this results only in a polynomial enlargement of the state space).

The size bounds for $\mathbb{A}$ are – asymptotically – the same as for $\mathbb{A}_{Q' \sqsubseteq Q'}$. That is, the size of $\mathbb{A}$ is polynomial in the number of rules in P', and at most exponential in the number of variables occurring in P' and the maximal size of a rule in P'. Thus, combined with the size bounds for $Q' = (P', \mathrm{Out})$ guaranteed by Lemma 4.4.6, $\mathbb{A}$ has size exponential in $\|Q\|$, $\|Z\|$, and $\|\Sigma\|$.

The correctness of our construction is almost immediate, given Lemmas 4.4.5 and 4.4.6. Indeed, if there is a constant d such that $\mathsf{c}(\mathcal{T}^{\min}_{Q',\mathsf{c}}) \leq d$, then Morgana can always pick a tree $T' \in \mathcal{T}^{\min}_{Q',\mathsf{c}}$ to win and keep the value of the counter below $d+1$. Thus, $\mathbb{A}$ is limited.

For the converse, consider the game on a tree $T \in \mathcal{T}^{\min}_{Q',\mathsf{c}}$. Since $\mathbb{A}$ is limited, Morgana can win by choosing a tree $T' \in \mathcal{T}_{Q'}$ such that $\mathsf{q}(T) \sqsubseteq \mathsf{q}(T')$ and $\mathsf{c}(T') \leq d$, for some constant d. But then, by definition of $\mathcal{T}^{\min}_{Q',\mathsf{c}}$, we also have that $\mathsf{c}(T) \leq d$. □

The final step for deciding parallel-boundedness is now to test whether $\mathbb{A}$ is limited. Thanks to Proposition 2.5.4, which has been proved by Benedikt et al. [Ben+15], this is possible in exponential time. We are now ready to prove that the parallel-boundedness problem for frontier-guarded Datalog queries and families in Hash-MConstraints is 2ExpTime-complete.

Proof of Theorem 4.4.2. The upper bound is implied by Proposition 4.4.7 and Proposition 2.5.4, which apply because frontier-guarded Datalog queries and modest sets of data-moving distribution constraints have the polynomial communication property according to Lemma 4.2.29.

For the lower bound, we show that the containment problem for monadic Datalog queries, which is 2ExpTime-hard [BBS12, Theorem 2], can be reduced to the parallel-boundedness problem PBound(FGDL, Hash-MConstraints), in polynomial time. To this end, let $Q_1 = (P_1, \text{Out})$ and $Q_2 = (P_2, \text{Out})$ be monadic Datalog queries. Without loss of generality, we assume that $\mathsf{idb}(P_1) \cap \mathsf{idb}(P_2) = \{\text{Out}\}$, and that Out does not occur in the body of any rule in P_1 and P_2. Furthermore, we can assume Q_1 and Q_2 are frontier-guarded, thanks to Lemma 2.4.9.

We construct a monadic, frontier-guarded Datalog query Q, a hash policy scheme Z and a modest set Σ of data-moving distribution constraints such that Q is parallel-correct w.r.t. $\mathcal{F}(Z, \Sigma)$, and Q is parallel-bounded w.r.t. $\mathcal{F}(Z, \Sigma)$ if and only if $Q_1 \sqsubseteq Q_2$ holds.

Let P_1' and P_2' be the Datalog programs obtained by replacing Out in P_1 and P_2 with fresh symbols Out_1 and Out_2, respectively. The Datalog program P is defined as $P_1' \cup P_2' \cup P_{\texttt{reach}}$ where $P_{\texttt{reach}}$ is the Datalog program consisting of the following rules.

$$
\begin{aligned}
S'(x) &\leftarrow \text{Start}(x), L() \\
S'(x) &\leftarrow S(y), E(y,x), L() \\
S(x) &\leftarrow S'(x), \text{Start}(x), K() \\
S(x) &\leftarrow S'(y), E(y,x), K() \\
\text{Out}(x) &\leftarrow \mathsf{body}(\tau), && \text{for all rules } \tau \in P_2' \text{ with } \mathsf{head}(\tau) = \text{Out}_2(x) \\
\text{Out}(x) &\leftarrow \mathsf{body}(\tau), S(y), \text{Target}(y) && \text{for all rules } \tau \in P_1' \text{ with } \mathsf{head}(\tau) = \text{Out}_1(x)
\end{aligned}
$$

In the rules for Out, the variable y is a fresh variable that does *not* appear in $\mathsf{body}(\tau)$. Further, E, K, L, Start, and Target are new extensional relation symbols, and S, S' are new intensional relation symbols, all of which do *not* occur in P_1' and P_2'.

We observe that Q is indeed frontier-guarded (and monadic), if Q_1 and Q_2 are. Evaluating Q over a global database G always yields all facts in $Q_2(G)$, thanks to the rules $\text{Out}(x) \leftarrow \mathsf{body}(\tau)$ constructed for all rules $\tau \in P_2'$ with $\mathsf{head}(\tau) = \text{Out}_2(x)$ in P_2'.

If G **(1)** contains the facts $K()$ and $L()$, and, **(2)** in the directed graph represented by the relation E, a Target-labelled node is reachable from a Start-labelled node, then $Q(G)$ also entails $Q_1(G)$. The only other case where this is true is if $Q_1(G) \subseteq Q_2(G)$ holds. In summary, we have $Q(G) = Q_2(G)$, or $Q(G) = Q_1(G) \cup Q_2(G)$. If (and only if) $Q_1 \sqsubseteq Q_2$ holds, these two cases collapse for all global databases.

We next define the hash policy scheme Z.

$$\begin{aligned} Z = &\big\{(R, 1, ()) \mid R \in \mathsf{edb}(P_1') \cup \mathsf{edb}(P_2')\big\} \\ &\cup \big\{(K, 1, ()), (E, 1, ()), (\mathrm{Start}, 1, ()), (\mathrm{Target}, 1, ())\big\} \\ &\cup \big\{(L, 2, ()), (E, 2, ()), (\mathrm{Start}, 2, ())\big\} \end{aligned}$$

For every distributed database $\mathcal{D} = (G, \mathcal{I})$ that complies with any $\delta_{Z,H}$, there are two servers k and ℓ such that I_k contains all extensional facts over $\mathsf{edb}(P_1') \cup \mathsf{edb}(P_2')$, the fact $K()$, if it is in G, and all E-, Start-, and Target-facts. Furthermore, the local database I_ℓ contains the fact $L()$, if it is in G, and all E- and Start-facts. If $\delta_{Z,H}$ scatters G then $k \neq \ell$ and neither server contains any other facts than those described above.

We observe that, in the first computation phase in any distributed evaluation over $\mathcal{D}$, all facts in $P_1(G)$ and $P_2(G)$ can be derived on server k. Next, consider the rules for S and S' in $P_{\texttt{reach}}$. They form a reachability subquery. An evaluation of this subquery alternates between deriving S and S'-facts. Since deriving S- and S'-facts requires the presence of $K()$ and $L()$, respectively, S-facts can only be derived on server k, and S'-facts can only be derived on server ℓ.

Thus, to correctly derive all S and S'-facts in a distributed evaluation over a scattered database, it is required (and sufficient) that all S'-facts can be sent from ℓ to k, and all S-facts can be sent from k to ℓ. This is indeed ensured by the modest set Σ of data-moving distribution constraints that consists of the following two rules.

$$K()@\kappa, L()@\lambda, S(x)@\kappa \to S(x)@\lambda \qquad K()@\kappa, L()@\lambda, S'(x)@\lambda \to S'(x)@\kappa$$

From the above it follows that, for every $(\delta, \gamma) \in \mathcal{F}(Z, \Sigma)$, in the distributed evaluation induced by γ over any distributed database $\mathcal{D} = (G, \mathcal{I})$ that complies with δ, all Out-facts in $Q_1(G)$ and $Q_2(G)$ can be derived on server k. We can conclude that Q is parallel-correct w.r.t. $\mathcal{F}(Z, \Sigma)$, and, hence, Q, Z, and Σ form a valid instance for the parallel-boundedness problem.

It remains to argue that Q is parallel-bounded w.r.t. $\mathcal{F}(Z, \Sigma)$ if and only if $Q_1 \sqsubseteq Q_2$ holds. Suppose $Q_1 \sqsubseteq Q_2$ holds. As argued above, we then have $Q \equiv Q_2$, i.e. $Q(G) = Q_2(G)$ for all global databases G. Also recall that all facts from $P_2'(G)$ can be derived on a single server in the first computation phase. Thanks to the rules $\mathrm{Out}(x) \leftarrow \mathsf{body}(\tau)$ constructed for all rules $\tau \in P_2'$ with $\mathsf{head}(\tau) = \mathrm{Out}_2(x)$ in P_2', the same is true for all Out-facts in $P_2(G)$. Therefore, all output facts can be derived within the first round. In other words, Q is parallel-bounded w.r.t. $\mathcal{F}(Z, \Sigma)$.

For the converse, suppose $Q_1 \sqsubseteq Q_2$ does *not* hold. Then there is a global database G and a fact $\mathrm{Out}(a)$ such that $\mathrm{Out}(a) \in Q_1(G)$ but $\mathrm{Out}(a) \notin Q_2(G)$.

We show that Q is *not* parallel-bounded w.r.t. $\mathcal{F}(Z, \Sigma)$ by providing an infinite sequence of global databases G_n, $n > 0$, such that in the distributed evaluation of Q induced by γ over $\delta(G_n)$, for some $(\delta, \gamma) \in \mathcal{F}(Z, \Sigma)$, at least $\Omega(n)$ rounds are required to derive $\text{Out}(a)$.

For every $n > 0$, G_n is obtained from G by adding the facts

- $K(), L(), \text{Start}(1), \text{Target}(n)$, and
- $E(i, i+1)$ for all $1 \leq i < n$.

To see that indeed at least $\Omega(n)$ rounds are required to derive $\text{Out}(a)$, we fix some $n > 0$. Let $(\delta, \gamma) \in \mathcal{F}(Z, \Sigma)$ such that δ scatters G_n. Recall that such a pair always exists due to Lemma 4.2.7. We consider the distributed evaluation induced by γ over $\delta(G_n)$. Since $\text{Out}(a)$ is not in $Q_2(G)$ – and hence not in $Q_2(G_n)$ – it can only be derived by a rule $\text{Out}(x) \leftarrow \mathsf{body}(\tau), S(y), \text{Target}(y)$ obtained from some rule τ of P_1'. Since $\text{Target}(n)$ is the only Target-fact in G, $\text{Start}(1)$ is the only Start-fact in G, and due to the choice of E, deriving $\text{Out}(a)$ requires the derivation of the sequence $S'(1), S(2), \ldots, S'(n-1), S(n)$ or $S(1), S'(2), \ldots, S(n-1), S(n)$ of S' and S-facts. Deriving a fact $S(i)$, for some i, requires $S'(i-1)$ and $K()$. Analogously, deriving $S'(j)$ require $S(j-1)$ and $L()$. Since δ scatters G_n, $K()$ and $L()$ do *not* reside on the same servers. But then, in every round, only one of the required S- or S'-facts can be derived, because the required S'- or S-fact, respectively, has to be communicated first. We conclude that the derivation of $\text{Out}(a)$ takes indeed at least $\Omega(n)$ rounds. □

We conclude this section with the assessment that the parallel-boundedness problem is also 2ExpTime-complete in the non-transitive communication setting. Recall that a Datalog query Q is parallel-correct w.r.t. a family $\mathcal{F}(Z, \Sigma)$ in the non-transitive communication setting, if its non-transitive translation $Q^\bullet$ is parallel-correct w.r.t. $\mathcal{F}(Z, \Sigma^\bullet)$, cf. Section 4.2.5. Correspondingly, we say that a *Datalog query Q is parallel-bounded w.r.t. a family $\mathcal{F}(Z, \Sigma)$ in the non-transitive communication setting*, if its non-transitive translation $Q^\bullet$ is parallel-bounded w.r.t. $\mathcal{F}(Z, \Sigma^\bullet)$. We have the following result.

Theorem 4.4.8. PBound(FGDL, Hash-Constraints) *is* 2ExpTime*-complete in the non-transitive communication setting.*

The proof is essentially the same as for Theorem 4.4.2. The upper bound is implied by Proposition 4.4.7 and Proposition 2.5.4, since non-transitive translations have the polynomial communication property thanks to Lemma 4.2.32.

The lower bound is implied by the lower bound proof for Theorem 4.4.2. Indeed, observe that in every distributed evaluation of the constructed query Q induced by a communication policy $\gamma_{\Sigma,\mathcal{N}}$, only S- and S'-facts are communicated, and the bodies of the distribution constraints in Σ contain only extensional atoms, except for the atoms witnessing the data-moving property, of course. Hence, every (partial) computation-free proof tree can always be replaced with a such a tree with only a single inner node (cf. the proof of Lemma 4.2.32). Thus, every proof tree with respect to Q and $\gamma_{\Sigma,\mathcal{N}}$ can be directly translated into a proof tree with respect to $Q^\bullet$ and $\gamma_{\Sigma^\bullet,\mathcal{N}}$. In particular, if a fact can be derived within r rounds in a distributed evaluation of Q induced by $\gamma_{\Sigma,\mathcal{N}}$, it can be

derived within r in a distributed evaluation of $Q^\bullet$ induced by $\gamma_{\Sigma^\bullet,\mathcal{N}}$. Clearly, the converse is true as well. Finally, recall that Q and $Q^\bullet$ are equivalent (over global databases). Therefore, Q and Σ can be substituted by $Q^\bullet$ and $\Sigma^\bullet$ in the correctness proof for the lower bound in the proof for Theorem 4.4.2. This results in a proof for the 2ExpTime-hardness of PBound(FGDL, Hash-Constraints) in the non-transitive communication setting.

4.5 Discussion and Related Work

We conclude this chapter with a discussion of related work. Particularly, we discuss the origins of our framework as well as similar frameworks used in the literature.

The MPC Model. As mentioned in the introduction, the MPC model has been introduced by Beame et al. [BKS17a]. In contrast to the adaption utilized in this chapter, the original definition allows more freedom for distributed evaluation strategies: In the communication phase servers can communicate any kind of data, and in the computation phase servers are only restricted by the data available locally. In particular, they are not limited to just evaluating the query or, in our case, the Datalog program, locally. Beame et al. studied the *load* – that is, the maximal number of bits (or facts) – and the number of rounds required to evaluate conjunctive queries within their MPC model. Here the more general model allows for stronger lower bound results. They also studied the *tuple-based* MPC model that restricts, up to some initial auxiliary data, communication to facts. While our communication policies effectively yield (only) a communication of facts, determining which facts are to be sent to which servers, might require additional data that cannot be computed upfront. For communication policies induced by distribution constraints, in particular, metadata about co-locations of facts is required at a minimum. To avoid the unnecessary communication of facts which are only used to determine whether (other) facts are to be communicated, the communication of further auxiliary data might be desired; notably for distribution constraints that have guarded communication.

The Hypercube Algorithm. The *Hypercube algorithm* introduced by Afrati and Ullman [AU11] can compute any multiway join in a single round. A concise description of it is given by Beame et al. [BKS17a, Algorithm 1]. Interestingly, the underlying ideas for the Hypercube algorithm originate from a parallel evaluation algorithm for Datalog programs by Ganguly et al. [GST90]. Hypercube-like algorithms played a fundamental role in several research contributions [e.g., BKS14; Afr+17; KS17; BKS17a] on parallel query evaluation – in particular, but not limited to, evaluation strategies following the MPC model. The distribution of facts for the Hypercube algorithm is guided by hash functions – one of our motivations for using hash-based distribution policies in our setting.

Parallel-Correctness. Ameloot et al. [Ame+17] introduced a framework to reason about Hypercube-like single-round algorithms for conjunctive queries. In this framework the initial distribution of facts is determined by a distribution policy, which are not necessarily

hash-based; and, in the (sole) computation phase every server evaluates the given Q on its local database. Ameloot et al. introduced and studied – among other related notions – parallel-correctness for conjunctive queries within this framework. Parallel-correctness and related problems were subsequently studied for (unions of) conjunctive queries (with and without negation) and under set as well as bag semantics by Ameloot et al. [Ame+17], Ketsman et al. [KNV18], Geck et al. [Gec+19] and Geck [Gec19]. Sundarmurthy et al. [SKN21] studied limitations of fact-based distribution policies (sometimes referred to as *oblivious* in the literature). Consequently, they also introduced a class of distribution policies, namely *co-hash schemes*, and studied parallel-correctness as well as a stronger variant, which requires that all output facts are disjointly partitioned among the servers, for conjunctive queries and these policies.

Ketsman et al. [KAK20] extended the framework of Ameloot et al. to study parallel-correctness and parallel-boundedness for Datalog in a multi-round MPC setting. As discussed in Sections 4.2.1 and 4.2.2, they proved that the parallel-correctness problem is undecidable for Datalog queries in general, even for value-independent distribution policies [KAK20, Theorem 1]. Their proof is by reduction from the containment problem for Datalog queries, which is well-known to be undecidable [Shm93, Theorem 1]. We continued from there and proved that it does *not* necessarily suffice to consider fragments of Datalog with a decidable containment problem to obtain a decidable parallel-correctness problem (Theorem 4.2.12). But restricting distribution policies (Proposition 4.2.16) or communication policies (Theorems 4.2.28 and 4.2.31) as well does.

A positive result proved by Ketsman et al. is that it is possible, given a Datalog query Q, to construct policies with respect to which Q is parallel-correct [KAK20, Proposition 7]. These policies are hash-based and extend the distribution pattern of the Hypercube algorithm.

Our framework is based on the framework of Ketsman et al. We provide a more detailed comparison of these frameworks next.

Economic Policies. Ketsman et al. [KAK20, Definition 2] defined *economic policies* to govern distributed evaluation strategies. An *economic policy* consists of a pair (π, χ) of fact-based distribution policies. Here, π is referred to as the *production policy* determining which servers are allowed to derive and send which intensional facts, whereas χ is the *consumption policy* determining which servers can receive and "consume" which intensional and extensional facts for the local computation. Initially, the facts of the input database are assumed to be partitioned arbitrarily over all servers. Thus, an economic policy not only guides the communication of intensional facts, it also reshuffles extensional facts in the first communication phase, and restricts the computation phases. Our framework extends the framework of Ketsman et al. in the sense that it allows to specify initial distributions and communication of facts independently and in different ways. Furthermore, our framework allows for policies which are *not* fact-based, as evidenced by our constraint-based communication policies.

An economic policy (π, χ) induces a policy pair (δ, γ) applicable to our setting. The distribution policy δ can be obtained by restricting χ to extensional facts. For the

communication policy γ, each fact $R(\bar{a})$ induces the set

$$\{R(\bar{a})@k \triangleright \ell \mid k \in \pi(R(\bar{a})), \ell \in \chi(R(\bar{a}))\}$$

of communicated facts. We note that the hash-based communication policies presented in Appendix B are also of this form, and, in combination with hash-based distribution policies, they can be understood as instantiation of economic policies in our framework.

Our evaluation strategies differ slightly from the ones of Ketsman et al. because the local fixpoint computations are *not* restricted by some kind of production policy. This kind of restriction can be meaningful in settings where, for instance, servers have access to all extensional facts. In such a case, the workload of individual servers can be kept under control.[19] In the conference paper [Nev+19, Section 7], which this chapter is based on, we therefore considered a variant of our setting – called the *locally restrained* setting – where every server is only allowed to derive facts that it can communicate to another server according to the communication policies. We showed that our 2ExpTime-completeness results on parallel-correctness are not affected by this difference in the setting. In a nutshell, the locally restrained semantics can be incorporated into the simulation of distributed evaluations over scattered database, that is, by adapting the construction for Lemma 4.2.22. Recall that the rules for the query simulating the distributed evaluation are obtained by replacing atoms in the body of an original rule with (translated) bodies of distribution constraints. In the same fashion, for each Datalog rule τ with head atom $R^i(\bar{z}, \bar{x})$, the (translated) body of a constraint with a head atom of the form $R(\bar{y})@\kappa$ (excluding the atom witnessing the constraint being data-moving) can be added to the body of τ. This ensures that every derived fact can be communicated. The construction of Lemma B.5 for the hash-based communication policies presented in Appendix B can be adapted similarly.

To conclude the comparison, let us point out that, in contrast to Ketsman et al., we consider parallel-correctness for *families* of policy pairs. This makes it, for example, possible to adapt hash functions and increase (or decrease) the number of servers on demand and automatically, without testing for parallel-correctness every time.

Parallel-Boundedness. Ketsman et al. [KAK20, Theorem 2] also studied parallel-boundedness within their framework. As for parallel-correctness they obtained undecidability results, in this case for Datalog queries without constants and variable repetitions in atoms, even if a bound for the number of rounds is given as parameter. A notable exception is the positive result that deciding whether such a Datalog queries can be evaluated within a single round with respect to families of their generalized hypercube policies is possible in polynomial time. As mentioned in Section 4.4 our definition of parallel-boundedness diverges slightly from that of Ketsman et al. because we only demand that no further *output* facts can be derived.

Our decision procedure for parallel-boundedness makes use of cost automata. This was inspired by the work of Benedikt et al. [Ben+15] who studied *boundedness* for guarded queries – including monadic Datalog. They proved, in particular, that boundedness can

[19] Recall that distribution policies can only demand the presence of facts, not their absence.

be decided in doubly exponential time, using cost automata. For this purpose they showed that the limitedness problem for alternating two-way tree cost automaton is decidable in exponential time. A result we also rely on and which is stated as Proposition 2.5.4 in this thesis.

Let us point out that boundedness is somewhat orthogonal to parallel-boundedness: The former asks, for a Datalog query, whether there is a d such that d applications of the immediate consequence operator suffice to evaluate the query, over any database. In contrast, parallel-boundedness does *not* restrict the local fixpoint computation, but instead the number of rounds. In fact, the local fixpoint computations may be unbounded, even if a Datalog query is parallel-bounded with respect to a family of policy pairs.

Distribution Constraints. The formalism for our data-moving distribution constraints is borrowed from Geck et al. [GNS20]. We emphasize that the notion of distribution constraints of Geck et al. is far more general (and powerful) than the constraints studied in this thesis. They encompass, in particular, tuple-, equality-, and server-generating constraints. Moreover, they can refer to the global database. This allows, for instance, to mimic hash policy schemes, cf. [GNS20, Section 3.3.1], and to define co-partitionings. Our notion of data-moving corresponds to *data-collecting* constraints that do not refer to the global database.

Geck et al. [GNS20, Corollary 27] proved that parallel-correctness for conjunctive queries w.r.t. families of distribution policies induced by sets Σ of their distribution constraints is decidable. The complexity ranges from NP to exponential time, depending on the distribution constraints allowed.

Abiteboul et al. [Abi+11] introduced the language *Webdamlog* (or *VWL* for short) which, roughly speaking, can be seen as an integration of Datalog queries (possibly with negation and equality) and distribution constraints. Similarly to the MPC model, evaluations proceed in rounds which consist of computation phases and communication steps. However, in a run of such a system, the servers do not operate in parallel but *fire* one after another. The evaluation result depends on the order, in general. Moreover, facts sent to a server are not persistent, i.e. they are deleted if not explicitly specified otherwise. Abiteboul et al. prove that *positive* Webdamlog systems always *converge*, and that this is a sufficient condition for parallel-correctness [Abi+11, Theorems 5 and 7]. We are not aware of any further research on parallel-correctness for Webdamlog. Interestingly, however, Webdamlog systems have been implemented [see e.g. AAS13; Mof+15].

Chapter 5

Structurally Simple Rewritings

This chapter is essentially concerned with two questions: When is it guaranteed that a structurally simple rewriting exists, if any rewriting exists. And what is the complexity to decide whether such a rewriting exists, and if so to compute one? Table 5.1 summarizes the answers we give to the second question in this chapter.

We note that, as in Chapter 4, the problems we study in this chapter are static analysis problems.

Outline. We will start by introducing terminology and some essential, known results for rewritings and related notions in Section 5.1. Based on this, we present our characterization of rewritability in Section 5.2.

In Section 5.3 we will then focus on *acyclic* rewritings: We study the existence of acyclic rewritings as well as the complexity of the acyclic rewriting problem. In Section 5.3.3 we will also briefly discuss the consequences of our NP-hardness results for multi-query evaluation settings.

Section 5.4 is dedicated to our tractability results for free-connex acyclic conjunctive queries over fixed database schemas (and a slight generalization thereof). In Section 5.5 we present our results on the existence of hierarchical and q-hierarchical rewritings as well as the associated decision problems.

Finally, we will discuss related work (not already covered in Section 5.1) and, in particular, the relationship of our characterization of rewritability with similar notions from the literature.

Publication and Contributions. This chapter is closely based on a journal article [Gec+23] authored by my advisor Prof. Dr Thomas Schwentick, my colleagues Dr Gaetano Geck and Dr Jens Keppeler, and – of course – myself. It has been accepted for publication in the *Logical Methods in Computer Science (LMCS)* journal, and is itself based on a conference paper [Gec+22] written by the same authors. A video presentation from me has also been published along with the conference paper [Spi+22].

The results (and definitions) presented in the journal article have been refined and extended multiple times by all authors and in joint research sessions. It is therefore fair to say that all authors contributed equally.

As mentioned above, this chapter closely follows the journal article [Gec+23], and can hence be considered a revision of it. In particular, I revised all proofs (to various degrees). Notable changes in comparison with the journal article are that

$\mathbb{V}$ Views	$\mathbb{Q}$ Query	$\mathbb{R}$ Rewriting	Restriction of views	$\textsc{Rewr}^k(\mathbb{V}, \mathbb{Q}, \mathbb{R})$ for every $k \in \mathbb{N}_0$ (bounded arity db-schema)	$\textsc{Rewr}(\mathbb{V}, \mathbb{Q}, \mathbb{R})$ (unbounded arity db-schema)
CQ	ACQ	CQ	Boolean views	NP-complete for $k \geq 3$ (Proposition 5.1.15)	
ACQ	CQ	CQ	Boolean views	NP-complete for $k \geq 3$ (Proposition 5.1.15)	
ACQ	ACQ	ACQ	no restriction	NP-complete for $k \geq 3$ (Theorem 5.3.8)	
ACQ	ACQ	ACQ	head arity $\leq \ell$ $\ell \in \mathbb{N}_0$	in P (Corollary 5.3.6)	
ACQ	ACQ	ACQ	weak head arity $\leq \ell$ $\ell \in \mathbb{N}_0$	in P (Proposition 5.4.8)	
CCQ	ACQ	ACQ	no restriction	in P (Theorem 5.4.2)	open
HCQ	HCQ	HCQ	no restriction	NP-complete for $k \geq 3$ (Corollary 5.5.5)	
QHCQ	HCQ	HCQ	no restriction	in P (Corollary 5.5.4)	open
QHCQ	QHCQ	QHCQ	no restriction	in P (Corollary 5.5.4)	open

Table 5.1: Complexity results for the (acyclic) rewriting problem. Note that the inclusions QHCQ $\subseteq$ CCQ $\subseteq$ ACQ and QHCQ $\subseteq$ HCQ $\subseteq$ ACQ hold. The head arity of a view V is the arity of its head atom $\mathsf{head}(V)$. The weak head arity of a view is defined in Definition 5.4.3. In a nutshell, a view with weak head arity ℓ can be "split" into (multiple) views with head arity at most ℓ.

(i) I proved NP-hardness for the cover description problem for the class QHCQ instead of "just" HCQ;

(ii) reworked the proofs of Theorem 5.1.3 and Corollary 5.3.13 to *no* longer rely on the construction from the proof of Theorem 5.3.9; and

(iii) highlighted whether (and under which circumstances) constructions can be computed efficiently by extending proofs and result statements.

The latter also allowed stating that q-hierarchical rewritings can be computed in polynomial time, if one exists and input query and views are q-hierarchical themselves (cf. Corollary 5.5.4). I believe that (i) and (ii) helped to flesh out the difference (in the complexity) of the acyclic rewriting problem and the cover description problem, when it comes to q-hierarchical queries.

5.1 Views, Rewritings, and the Problem

At the beginning of this section we introduce the rewriting problem and related notions. Of course, this includes a formal definition of views and rewritings. We will then proceed by discussing some known results, concepts, and techniques regarding the (classical) rewriting problem for conjunctive queries. At the end, we obtain our first NP-hardness result which, intuitively, states that the rewriting problem does not become "easier" if

only some of the input queries have a simple structure. This will serve as a baseline for the remainder of this chapter.

Views and Rewritings. A *view* V over a schema $\mathcal{S}$ is just a query over the schema $\mathcal{S}$.[1] A finite set $\mathcal{V}$ of views induces, for each database D over schema $\mathcal{S}$, the *$\mathcal{V}$-defined database*

$$\mathcal{V}(D) = \bigcup_{V \in \mathcal{V}} V(D).$$

Sometimes it will be useful to understand $\mathcal{V}$ as a query whose query result is defined as above, for all databases D. For convenience, we will make no distinction in the notation.

In this thesis, we only consider *finite* sets $\mathcal{V}$ of views and only views that are conjunctive queries. Furthermore, we require that all views in $\mathcal{V}$ have pairwise distinct relation symbols in their heads. In other words, each view in $\mathcal{V}$ contributes its own relation to $\mathcal{V}(D)$. Lastly, we assume that the head variables of each view are pairwise distinct as well. Note that this is no restriction, because, if there are multiple occurrences of a variable in the head of a view, all but one can just be removed – they only lead to duplicated columns in the $\mathcal{V}$-defined database. We will not state these assumptions explicitly every time.

We denote the *schema induced by the heads of the views* in $\mathcal{V}$ by $\mathcal{S}_\mathcal{V}$. Let us point out that, for a set $\mathcal{V}$ of views over a schema $\mathcal{S}$, the schemas $\mathcal{S}_\mathcal{V}$ and $\mathcal{S}$ are disjoint. This is because, by definition, the relation symbol in the head of a conjunctive query does *not* occur in the underlying database schema. For convenience, we will identify name and head symbol of views. That is, for a view $V \in \mathcal{V}$, we denote its distinguished head symbol also by V. If we want to emphasize that the relation symbol of an atom is from $\mathcal{S}_\mathcal{V}$, we call it a *view atom*.

In a nutshell, a $\mathcal{V}$-rewriting of a query Q is a query Q' over $\mathcal{S}_\mathcal{V}$ that is meant to yield, for every database D, the same query result over $\mathcal{V}(D)$ as Q does over D. Recall that $Q' \circ \mathcal{V}$ denotes the query over the same schema as $\mathcal{V}$ with query result $Q'(\mathcal{V}(D))$, for every database D.

Definition 5.1.1 (Rewriting)**.** Let Q be a query, and $\mathcal{V}$ be a set of views. A query Q' over $\mathcal{S}_\mathcal{V}$ is a *$\mathcal{V}$-rewriting of Q* if $Q' \circ \mathcal{V}$ and Q are equivalent.

We say that Q is *$\mathcal{V}$-rewritable* if there is a $\mathcal{V}$-rewriting of Q.

In the literature, rewritings are also often defined as queries over the schema $\mathcal{S}_\mathcal{V} \cup \mathcal{S}$, where $\mathcal{S}$ is the schema of the original query Q [see, e.g. Lev+95]. Rewritings which only refer to $\mathcal{S}_\mathcal{V}$ are then called *complete*. Another common variant are rewritings which are only meant to approximate the original query. In that case, they only have to satisfy a containment condition [see, e.g., PH01, Definitions 1 and 2]. In this thesis, however, we are only interested in complete and *equivalent* rewritings; and this is what Definition 5.1.1 yields. Let us point out, though, that requiring completeness is not a restriction, if one is – like us – interested in deciding whether a rewriting exists: Every $\mathcal{S}$-relation can be replicated as a $\mathcal{S}_\mathcal{V}$-relation by adding a (trivial) view. On the other hand, the answer to

[1] They are called views due to their special role which distinguishes them from "normal" queries.

whether an incomplete rewriting exists, is always trivially yes, because the original query itself is a rewriting in this case.[2]

We illustrate the notions introduced so far with an example.

Example 5.1.2. Let $\mathcal{V}$ be the set consisting of the following two views over the schema $\mathcal{S} = \{P, R, S, T\}$.

$$V_1(x_1, w_1) \leftarrow P(v_1, v_1', x_1), R(x_1, w_1), S(w_1)$$
$$V_2(y_2, z_2) \leftarrow S(y_2), T(y_2, z_2)$$

Furthermore, let Q be the conjunctive query

$$H(x, y, y') \leftarrow P(u, u', x), R(x, w), S(w), T(w, y), T(w, y')$$

over schema $\mathcal{S}$ and Q' be the conjunctive query

$$H(x, y, y') \leftarrow V_1(x, w), V_2(w, y), V_2(w, y')$$

over schema $\mathcal{S}_\mathcal{V} = \{V_1, V_2\}$. For each database D, the query Q yields the same result over D as the query Q' over $\mathcal{V}(D)$. Therefore, Q' is a $\mathcal{V}$-rewriting of Q. ◁

For classes $\mathbb{V}$, $\mathbb{Q}$, and $\mathbb{R}$ of conjunctive queries, we write Rewr($\mathbb{V}$, $\mathbb{Q}$, $\mathbb{R}$) for the *rewriting problem for* $\mathbb{V}$, $\mathbb{Q}$, *and* $\mathbb{R}$ which is defined as follows.

— Rewr($\mathbb{V}$, $\mathbb{Q}$, $\mathbb{R}$) —

Given: Set $\mathcal{V} \subseteq \mathbb{V}$ of views and a conjunctive query $Q \in \mathbb{Q}$

Question: Is there a $\mathcal{V}$-rewriting of Q in the class $\mathbb{R}$?

We write Rewr^k($\mathbb{V}$, $\mathbb{Q}$, $\mathbb{R}$) for the restriction of Rewr($\mathbb{V}$, $\mathbb{Q}$, $\mathbb{R}$), where the arity of each relation symbol in the database schema is bounded by k.

Levy et al. [Lev+95] proved that the most general instantiation of the rewriting problem for conjunctive queries is NP-complete.

Theorem 5.1.3 [Lev+95, Theorem 3.10]**.** Rewr(CQ, CQ, CQ) *is* NP*-complete.*

The Canonical Candidate. There is a straightforward, albeit in general inefficient, algorithm to determine whether a rewriting in the class CQ of all conjunctive queries exists (and if it does, to output one). It proceeds in two steps. In the first step it computes a *candidate*, and in the second step it tests whether this candidate is a rewriting. The correctness of this algorithm relies on the existence of a candidate that is guaranteed to be a rewriting, if a rewriting exists at all. Indeed, for every query Q and set $\mathcal{V}$ of views, there is such a candidate query over $\mathcal{S}_\mathcal{V}$.

The *canonical candidate* canon$(Q, \mathcal{V})$ can be computed as follows.

[2] For incomplete rewritings, one usually asks for rewritings that satisfy an additional requirement – for instance, one might ask for a rewriting with less atoms than the original query.

(1) The *canonical database* $\mathsf{canon}(Q)$ is obtained from the input query Q by interpreting the atoms in $\mathsf{body}(Q)$ as facts, in which variables are considered as fresh constants. More precisely, for each variable x in $\mathsf{vars}(Q)$, the active domain of $\mathsf{canon}(Q)$ contains a distinguished value a_x, and for every atom $R(\bar{x}) \in \mathsf{body}(Q)$, $\mathsf{canon}(Q)$ contains the fact $R(\bar{a}) = \vartheta(R(\bar{x}))$ where ϑ is the mapping with $\vartheta(x) = a_x$, for every $x \in \mathsf{vars}(Q)$.

(2) The given set $\mathcal{V}$ of views is evaluated over $\mathsf{canon}(Q)$. The result $\mathcal{V}(\mathsf{canon}(Q))$ is a database over the schema $\mathcal{S}_\mathcal{V}$.

(3) The canonical candidate $\mathsf{canon}(Q, \mathcal{V})$ is the query over $\mathcal{S}_\mathcal{V}$ with the same head as Q and the body obtained from $\mathcal{V}(\mathsf{canon}(Q))$ by interpreting the domain values as variables again. More formally, the head is $\mathsf{head}(Q)$ and the body is $\vartheta^{-1}(\mathcal{V}(\mathsf{canon}(Q)))$ where ϑ^{-1} is the inverse of the mapping used in the first step.

An important detail should be mentioned here: The canonical candidate does *not* always exist, for example, if $\vartheta^{-1}(\mathcal{V}(\mathsf{canon}(Q)))$ does *not* contain all head variables of Q, or it is outright empty. In that case, there is no rewriting. If the canonical candidate turns out to be a rewriting, then we often call it the *canonical rewriting*.

Nash et al. [NSV10] showed the following, fundamental result. We note that, for conjunctive queries without self-joins, this had already been shown by Chekuri and Rajaraman [CR00, Lemma 7], and implicitly been utilized by Levy et al. [Lev+95, Proof of Lemma 3.3].

Proposition 5.1.4 [NSV10, Proposition 5.1]**.** *Let Q be a conjunctive query and $\mathcal{V}$ a set of views. If there is a $\mathcal{V}$-rewriting of Q, then the canonical candidate $\mathsf{canon}(Q, \mathcal{V})$ is such a rewriting.*

Before we discuss how to decide whether the canonical candidate is a rewriting, we have a look at an example.

Example 5.1.5. Let us consider the views

$$V_1(u_1, v_1, w_1) \leftarrow C(u_1, v_1, w_1), \quad \text{and}$$
$$V_2(x_2, y_2, z_2, u_2) \leftarrow R(x_2, y_2), S(y_2, z_2), T(z_2, u_2);$$

and the conjunctive query Q defined by

$$H(x, y, z) \leftarrow C(x, y, z), R(x, y), S(y, z), T(z, x).$$

The canonical database of Q is

$$\mathsf{canon}(Q) = \{C(a_x, a_y, a_z), R(a_x, a_y), S(a_y, a_z), T(a_z, a_x)\}.$$

Evaluating the views V_1 and V_2 over $\mathsf{canon}(Q)$ of Q yields the result

$$\{V_1(a_x, a_y, a_z), V_2(a_x, a_y, a_z, a_x)\}.$$

Thus, the query Q' defined by

$$H(x, y, z) \leftarrow V_1(x, y, z), V_2(x, y, z, x)$$

is the canonical candidate.

In this example it not hard to see that the canonical candidate Q' is a $\mathcal{V}$-rewriting. Even without considering any (let alone all) databases. In fact, in this particular case, inlining the bodies of the views into Q' yields the original query Q. For instance, the view atom $V_1(x, y, z)$ in $\mathsf{body}(Q')$ yields, together with $\mathsf{head}(V_1) = V_1(u_1, v_1, w_1)$, the variable mapping $\alpha = \{u_1 \mapsto x, v_1 \mapsto y, w_1 \mapsto z\}$. It is therefore replaced with

$$\alpha(\mathsf{body}(V_1)) = \{C(x, y, z)\}. \qquad \triangleleft$$

Expansions. We discuss next, how the approach of inlining view definitions, as illustrated in Example 5.1.5, can be generalized. The goal is to obtain a conjunctive query over the same schema as the input query Q, which can then be directly compared to Q by, for instance, testing containment.

The idea is similar to the approach taken for checking containment of frontier-guarded Datalog queries in Section 4.3. Recall that every Datalog query is equivalent to a (possibly infinite) set of conjunctive queries derived from symbolic proof trees, that is, trees labelled with rule instantiations of the Datalog program, cf. Lemma 4.3.3. We can understand a query Q' over $\mathcal{S}_\mathcal{V}$ and $\mathcal{V}$ as a Datalog query with the Datalog program $\{Q'\} \cup \mathcal{V}$. The output symbol is the symbol from $\mathsf{head}(Q')$. Note that this Datalog query is already a query over the same schema as the original query Q, and is equivalent to $Q' \circ \mathcal{V}$. It also has a very simple structure: The body of the rule Q', which is the only rule for the output symbol, contains only intensional atoms, and the bodies of every other rule contain only extensional atoms. Thus, the symbolic proof trees have exactly two levels. The root node is labelled with a rule instantiation of Q' and the nodes below the root node are labelled with rule instantiations of rules in $\mathcal{V}$. Since it always suffices to consider symbolic proof trees over a finite set of variables, thanks to Lemma 4.3.4, the Datalog query induced by Q' and $\mathcal{V}$ is equivalent to a *finite* set of conjunctive queries over the same schema as the original query Q.

We argue next that, by carefully choosing the rule instantiations, it suffices to consider a single proof tree, and hence, a single conjunctive query, which we will call an *expansion*. To make this precise we employ the following kind of rule applications.

Definition 5.1.6 (View Application)**.** A *view application* of a view V is a substitution $\alpha \colon \mathsf{vars}(V) \to \mathsf{var}$ that does *not* unify any quantified variable of V with another variable of V.

In contrast to rule applications in general, view applications can thus not unify quantified variables. This reflects the fact that rewritings determine the variables in view atoms, and hence all head variables of the views upon inlining, but *not* quantified variables that only occur in the bodies of the views.

Example 5.1.7. Consider the view $V(x') \leftarrow R(x'), S(y')$ and the conjunctive query Q defined by $H(x) \leftarrow R(x), S(x)$. The canonical candidate Q' is $H(x) \leftarrow V(x)$. Inlining the definition of V according to the (rule) application α with $\alpha = \{x' \mapsto x, y' \mapsto x\}$ yields a query equivalent (and, in fact, identical) to Q. However, α is not a view application, and indeed, there is *no* view application that yields a query equivalent to Q, because this requires unifying the quantified variable y' with x'. This corresponds to the fact that Q' is *not* a $\{V\}$-rewriting – and due to Proposition 5.1.4 there is hence no $\{V\}$-rewriting at all. Indeed, evaluating V yields a copy of the relation R, if S is non-empty, and the empty relation, otherwise. The same is true for evaluating Q' over any database defined by $\{V\}$. The query Q, on the other hand, asks for the intersection of R and S. ◁

Let us point out that, in difference to our rewriting setting here, the use of the more general notion of rule applications in the context of checking containment of Datalog queries is not problematic, because the applications are for the rules of the original query itself, and the set of all symbolic proof trees encompasses all possible combinations of applications.[3]

In the remainder of this chapter we will only consider view applications even though we sometimes just write "application" for brevity.

Further on, we require one more ingredient to define expansions. The following corresponds to renaming unconnected occurrences of variables in a symbolic proof tree upon deriving the associated conjunctive query.

Let $\alpha_1, \ldots, \alpha_m$ be a sequence of view applications for views $V_1, \ldots, V_m$ from a set $\mathcal{V}$. Note that the sequence $V_1, \ldots, V_m$ might contain duplicates. This corresponds to a query over $\mathcal{S}_\mathcal{V}$ having self-joins. The view applications $\alpha_1, \ldots, \alpha_m$ fulfil *quantified variable disjointness*, if for all $i, j \in [1, m]$, each quantified variable x from $\mathsf{vars}(V_i)$, and each variable y from $\mathsf{vars}(V_j)$ with $i \neq j$, it holds $\alpha_i(x) \neq \alpha_j(y)$. That is, beyond what is already required by the definition of a view application, view applications fulfilling quantified variable disjointness do *not* unify quantified variables (with respect to their view) with any other variable occurring in any of the views. This ensures that, for all $i, j \in \{1, \ldots, m\}$ with $i \neq j$, the bodies $\alpha_i(\mathsf{body}(V_i))$ and $\alpha_j(\mathsf{body}(V_j))$ only share variables from $\mathsf{vars}(\alpha_i(\mathsf{head}(V_i))) \cap \mathsf{vars}(\alpha_j(\mathsf{head}(V_j)))$.

An expansion of a $\mathcal{V}$-rewriting Q' is, intuitively, obtained by inlining the bodies of the views from $\mathcal{V}$ in Q' according to view applications fulfilling quantified variable disjointness.

Definition 5.1.8 (Expansion)**.** Let $\mathcal{V}$ be a set of views over a schema $\mathcal{S}$ and let Q' be a conjunctive query with $\mathsf{body}(Q') = \{A'_1, \ldots, A'_m\}$ over the schema $\mathcal{S}_\mathcal{V}$. Furthermore, for each $i \in [1, m]$, let V_i be the view in $\mathcal{V}$ and α_i be a view application for V_i such that

(a) $A'_i = \alpha_i(\mathsf{head}(V_i))$, for each $i \in [1, m]$; and

(b) the sequence $\alpha_1, \ldots, \alpha_m$ fulfils quantified variable disjointness.

The *expansion of Q' with respect to $\mathcal{V}$ and $\alpha_1, \ldots, \alpha_m$* is the conjunctive query that has the same head as Q' and body $\bigcup_{i=1}^{m} \alpha_i(\mathsf{body}(V_i))$.

[3] In fact, the more general notion is required because, to properly model the iterative inlining necessary for Datalog queries in general, quantified variables might have to be unified.

Since the view applications $\alpha_1, \ldots, \alpha_m$ in Definition 5.1.8 are uniquely determined up to renaming of quantified variables, we usually do not mention them explicitly and just speak of an expansion of a query Q' w.r.t. $\mathcal{V}$.

Example 5.1.9 (Continuation of Example 5.1.2)**.** Recall the query Q' defined by $H(x, y, y') \leftarrow V_1(x, w), V_2(w, y), V_2(w, y')$ from Example 5.1.2. For convenience, let us also recall the rules for the views V_1 and V_2.

$$V_1(x_1, w_1) \leftarrow P(v_1, v_1', x_1), R(x_1, w_1), S(w_1) \qquad V_2(y_2, z_2) \leftarrow S(y_2), T(y_2, z_2)$$

Now, consider the view application α_1 for V_1 which maps x_1 to x, w_1 to w, and is the identity on every other variable. Furthermore, let α_2 and α_3 be view applications for V_2 with $\alpha_2(y_2) = w$, $\alpha_2(z_2) = y$, $\alpha_3(y_2) = w$, and $\alpha_3(z_2) = y'$. We have that α_1, α_2, and α_3 fulfil quantified variable disjointness, and the atoms $\alpha_1(\mathsf{head}(V_1))$, $\alpha_2(\mathsf{head}(V_2))$, $\alpha_3(\mathsf{head}(V_2))$ form the body of Q'. Therefore, the conjunctive query

$$H(x, y, y') \leftarrow \underbrace{P(v_1, v_1', x), R(x, w), S(w)}_{\alpha_1(\mathsf{body}(V_1))}, \underbrace{S(w), T(w, y)}_{\alpha_2(\mathsf{body}(V_2))}, \underbrace{S(w), T(w, y')}_{\alpha_3(\mathsf{body}(V_2))}$$

is an expansion of Q' w.r.t. $\{V_1, V_2\}$. ◁

It is not hard to see that the expansion of Q' in Example 5.1.9 is equivalent to the original query Q defined in Example 5.1.2. We discuss next that this implies that Q' is a rewriting of Q. In fact, this will follow readily from the following result.

Lemma 5.1.10 [see, e.g., AC19]**.** *Let $\mathcal{V}$ be a set of views, Q' be a query over the schema $\mathcal{S}_\mathcal{V}$, and Q'' be an expansion of Q' w.r.t. $\mathcal{V}$. Then Q'' is equivalent to $Q' \circ \mathcal{V}$.*

Lemma 5.1.10 has been proved, for instance, as part of the proof of [AC19, Theorem 3.5]. For the sake of convenience, we provide a proof using our notation in the following.

Proof of Lemma 5.1.10. Following Definition 5.1.8, let $\{A_1', \ldots, A_m'\}$ be the body of Q', and let $V_1, \ldots, V_m \in \mathcal{V}$ be the views and $\alpha_1, \ldots, \alpha_m$ be the applications fulfilling quantified variable disjointness for these views, and such that, for all $i \in \{1, \ldots, m\}$, $A_i' = \alpha_i(\mathsf{head}(V_i))$ holds and $\mathsf{body}(Q'') = \bigcup_{i=1}^m \alpha_i(\mathsf{body}(V_i))$.

We first show $Q'' \sqsubseteq Q' \circ \mathcal{V}$. For this purpose, let D be a database and ϑ be a valuation such that the fact $\vartheta(\mathsf{head}(Q''))$ is in the query result $Q''(D)$. By the semantics of conjunctive queries, we then have that D satisfies $\vartheta(\mathsf{body}(Q'')) = \vartheta(\bigcup_{i=1}^m \alpha_i(\mathsf{body}(V_i)))$ under ϑ. By the definition of an expansion, we also have $\vartheta(\mathsf{head}(Q'')) = \vartheta(\mathsf{head}(Q'))$. Thus, it suffices to show that $\vartheta(\mathsf{body}(Q')) \subseteq \mathcal{V}(D)$ holds, since that implies

$$\vartheta(\mathsf{head}(Q')) \in Q'(\mathcal{V}(D)) = (Q' \circ \mathcal{V})(D).$$

Since $\vartheta(\mathsf{body}(Q'') \subseteq D$, we have $\vartheta(\alpha_i(\mathsf{body}(V_i))) \subseteq D$, for all $i \in \{1, \ldots, m\}$. Interpreting $\vartheta \circ \alpha_i$ as a valuation, it follows that $\vartheta(\alpha_i(\mathsf{head}(V_i))) \in V_i(D)$. But $\alpha_i(\mathsf{head}(V_i)) = A_i'$ for all $i \in [1, m]$ and, therefore, we can conclude that

$$\vartheta(\mathsf{body}(Q')) = \vartheta(\{A_1', \ldots, A_m'\}) \subseteq \mathcal{V}(D).$$

For the proof of $Q' \circ \mathcal{V} \sqsubseteq Q''$ let ϑ be a valuation such that $\mathcal{V}(D)$ satisfies $\mathsf{body}(Q')$ under ϑ. Thus, $\vartheta(A'_i) \in \mathcal{V}(D)$ and therefore $\vartheta(\alpha_i(\mathsf{head}(V_i))) \in V_i(D)$ holds, for all $i \in [1, m]$. The latter implies that there are valuations ϑ_i such that $\vartheta(\alpha_i(\mathsf{head}(V_i))) = \vartheta_i(\mathsf{head}(V_i))$ and $\vartheta_i(\mathsf{body}(V_i)) \subseteq D$. That is, the ϑ_i map $\mathsf{body}(V_i)$ into D and agree with $\vartheta \circ \alpha_i$ on all head variables of V_i. Moreover, since the view application α_i does not unify any quantified variables with other variables, the valuation ϑ can be extended to a valuation ϑ_i^+ such that $\vartheta_i^+(\alpha_i(\mathsf{body}(V_i))) \subseteq D$ holds.

Lastly, thanks to the quantified variable disjointness of the α_i, the extended mappings ϑ_i^+ can be combined into a valuation ϑ^+ which maps all $\alpha_i(\mathsf{body}(V_i))$ into D. □

As illustrated in Example 5.1.9, the expansion of a rewriting Q' can be directly compared with a query Q since it is over the same schema. In fact, we will frequently use the following result which follows readily from Lemma 5.1.10.

Proposition 5.1.11 [AC19, Theorem 3.5]**.** *Let Q be a conjunctive query over some schema $\mathcal{S}$, $\mathcal{V}$ be a set of views over $\mathcal{S}$, and Q' be a conjunctive query over the schema $\mathcal{S}_\mathcal{V}$.*

An expansion Q'' of Q' w.r.t. $\mathcal{V}$ is equivalent to Q if and only if Q' is a $\mathcal{V}$-rewriting of Q.

In Example 5.1.2, the query Q' is thus a $\mathcal{V}$-rewriting of Q because the expansion of Q' considered in Example 5.1.9 is equivalent to Q. Up to variable renamings (and duplicated atoms in the presentation), they are even identical. This applies similarly to the queries considered in Example 5.1.5. Here the views are even full queries, and, thus, there is only one (unique) expansion which, in this case, coincides with the original query.

Deciding Rewritability. In general, canonical candidates have one noteworthy disadvantage when it comes to algorithmic applications. If the arity of views is not restricted, then the size (and even the length) of the canonical candidate can be exponential in $\|Q\| + \|\mathcal{V}\|$. However, Levy et al. [Lev+95] proved that there always is a rewriting of polynomial size, if there is one at all. Recall that the length of a conjunctive query is the number of its atoms.

Lemma 5.1.12 [Lev+95, Lemma 3.5]**.** *Let Q be a conjunctive query and $\mathcal{V}$ be a set of views. If there is a $\mathcal{V}$-rewriting of Q then there is one of length $|Q|$ and size polynomial in $\|Q\| + \|\mathcal{V}\|$.*

Example 5.1.13. Consider the conjunctive query $H() \leftarrow S(x, y), R(x), R(y)$ and the views

$$V_1(x_1, y_1) \leftarrow S(x_1, y_1) \text{ and } V_2(x_{2,1}, \ldots, x_{2,n}) \leftarrow R(x_{2,1}), \ldots, R(x_{2,n}).$$

Evaluating the views on the canonical database $\mathsf{canon}(Q)$ yields the result

$$\{V_1(a_x, a_y)\} \cup \{V_2(a_1, \ldots, a_n) \mid a_1, \ldots, a_n \in \{a_x, a_y\}\}.$$

This query result, and thus, the body of the canonical candidate, has exponential size. There is, however, a simple $\{V_1, V_2\}$-rewriting which has the same length as Q, namely,

$$H() \leftarrow V_1(x, y), V_2(x, \ldots, x), V_2(y, \ldots, y).$$ ◁

Lemma 5.1.12 in combination with the well-known result from Chandra and Merlin [CM77] that the containment problem for conjunctive queries is in NP, yields that Rewr(CQ, CQ, CQ) is in NP, as stated in Theorem 5.1.3. An NP-algorithm for Rewr(CQ, CQ, CQ) can "guess" a rewriting of polynomial size, compute an expansion, and then test whether the expansion is equivalent to the given query.

Canonical candidates have the potential advantage that they can be computed instead of having to be "guessed". In cases where they are guaranteed to have polynomial size, this can sometimes lead to efficient algorithms. For instance, this guarantee is given if the views have bounded arity. However, note that the algorithm suggested by Proposition 5.1.4 of computing the canonical candidate, and then testing whether an expansion of it is equivalent to the given query, is only efficient, if evaluation and testing equivalence can be performed efficiently as well. For acyclic views of bounded arity and acyclic input queries all these constraints are satisfied, which lets us derive the following result. For a class $\mathbb{V}$ of views, we write $\mathbb{V}^k$ for the restriction to views of arity at most k.

Proposition 5.1.14. *For every $k \geq 0$,* Rewr(ACQ^k, ACQ, CQ) *is in* P. *Moreover, if a rewriting exists, one of polynomial size can be computed in polynomial time.*

Proof. A polynomial time algorithm for Rewr(ACQ^k, ACQ, CQ) can proceed in three steps. Given a set $\mathcal{V} \subseteq \mathsf{ACQ}^k$ of views and a query $Q \in \mathsf{ACQ}$, it first computes the canonical candidate $\mathsf{canon}(Q, \mathcal{V})$. The second step is to obtain an expansion Q_E of $\mathsf{canon}(Q, \mathcal{V})$. In the third and final step, the algorithm tests whether $Q_E \sqsubseteq Q$ holds, and returns the answer. Of course, if the canonical candidate does not exist, the algorithm aborts after the first step and outputs *no*.

We first argue that this algorithm runs in polynomial time. Afterwards, we will prove the correctness, and, particularly, why a containment test suffices.

Thanks to the primordial results on acyclic conjunctive queries of Yannakakis [Yan81, Theorem 4.1], evaluating the acyclic views in $\mathcal{V}$ over the canonical database of Q takes polynomial time, in the input and the output size [see also Are+21, Theorem 20.4]. Observe that the output size, and hence the size of the canonical candidate $\mathsf{canon}(Q, \mathcal{V})$, is polynomial because the arity of the views is bounded. Obtaining an expansion Q_E is mainly a matter of renaming (quantified) variables according to Definition 5.1.8, and can clearly be done in polynomial time. Lastly, the containment test can be done in polynomial time because the containment problem Cont(CQ, ACQ) is in P and we have $Q \in \mathsf{ACQ}$. The former is again thanks to Yannakakis [Yan81, Corollary 4.1].[4]

For the correctness we will show that an expansion Q_E of $\mathsf{canon}(Q, \mathcal{V})$ always obeys $Q \sqsubseteq Q_E$, if the canonical candidate exists. Therefore, the algorithm outputs *yes* if and only if Q_E and Q are equivalent. The correctness of the algorithm then follows from Propositions 5.1.4 and 5.1.11.

It remains to prove that, for an expansion Q_E of the canonical candidate, we have $Q \sqsubseteq Q_E$. To this end, we fix an expansion Q_E of $\mathsf{canon}(Q, \mathcal{V})$ in the following. To

[4] Testing for $Q_E \sqsubseteq Q$ can be done by deciding whether there is a homomorphism from Q to Q_E. This can in turn be determined by testing whether $\mathsf{head}(Q_E)$ is in the query result of Q over the canonical database of the query Q_E.

improve readability and avoid technical clutter we identify the variables in Q with their corresponding constants in $\mathsf{canon}(Q)$ in the remainder. Notably, this means that we identify $\mathsf{canon}(Q)$ with $\mathsf{body}(Q)$, i.e. we write $\mathsf{canon}(Q) = \mathsf{body}(Q)$.

Following Definition 5.1.8, let $A'_1, \ldots, A'_m$ be the atoms of $\mathsf{canon}(Q, \mathcal{V})$ and, for each $i \in [1, m]$, let α_i be a view application and V_i be the view such that $A'_i = \alpha_i(\mathsf{head}(V_i))$, and the $\alpha_1, \ldots, \alpha_m$ fulfil quantified variable disjointness. Furthermore, since the A'_i are also contained in $\mathcal{V}(\mathsf{canon}(Q))$ – by definition of $\mathsf{canon}(Q, \mathcal{V})$ – there are valuations ϑ_i such that $A'_i = \vartheta_i(\mathsf{head}(V_i))$ and $\vartheta_i(\mathsf{body}(V_i)) \subseteq \mathsf{canon}(Q)$ holds. In particular, we have that $\vartheta_i(\mathsf{head}(V_i)) = \alpha_i(\mathsf{head}(V_i))$ for all $i \in [1, m]$. We fix Q_E to be the expansion whose body is the union of all $\alpha_i(\mathsf{body}(V_i))$.

To prove $Q \sqsubseteq Q_E$ we construct a homomorphism h from Q_E to Q. Let y be a variable occurring in $\alpha_i(\mathsf{body}(V_i))$ and x such that $y = \alpha_i(x)$ for some $i \in [1, m]$. Then the desired homomorphism h maps y to $\vartheta_i(x)$. Note that h is the identity on variables occurring in $\alpha_i(\mathsf{head}(V_i))$ because $\alpha_i(\mathsf{head}(V_i)) = \vartheta_i(\mathsf{head}(V_i))$ and, thus, $y = \alpha_i(x) = \vartheta_i(x) = h(y)$. Hence, h is well-defined, since, if a variable y occurs in $\alpha_i(\mathsf{body}(V_i))$ and $\alpha_j(\mathsf{body}(V_j))$ with $i \neq j$ then it also occurs in $\alpha_i(\mathsf{head}(V_i))$ and $\alpha_j(\mathsf{head}(V_j))$ thanks to quantified variable disjointness. Furthermore, for variables y occurring in a set $\alpha_i(\mathsf{body}(V_i))$ but not in $\alpha_i(\mathsf{head}(V_i))$, the preimage x is a quantified variable, and hence, unique because view applications cannot unify quantified variables.

We have that h is indeed a homomorphism from the expansion Q_E of $\mathsf{canon}(Q, \mathcal{V})$ to Q, because

$$h(\alpha_i(\mathsf{body}(V_i))) = \vartheta_i(\mathsf{body}(V_i)) \subseteq \mathsf{canon}(Q)$$

holds for all $i \in [1, m]$ and h is the identity on all variables occurring in the head of $\mathsf{canon}(Q, \mathcal{V})$. □

Chekuri and Rajaraman [CR00, Theorem 5] have shown that the rewriting problem for arbitrary views and acyclic queries without self-joins is in P using a similar idea.

However, even for Boolean views and databases over a small fixed schema, it does not suffice to restrict only the views or only the query to the class of acyclic queries.

Proposition 5.1.15. $\textsc{Rewr}^k(\mathsf{ACQ}^0, \mathsf{CQ}, \mathsf{CQ})$ *and* $\textsc{Rewr}^k(\mathsf{CQ}^0, \mathsf{ACQ}, \mathsf{CQ})$ *are* NP*-complete, for every* $k \geq 3$*. This even holds for Boolean conjunctive queries, sets of views with only one Boolean view and a schema with two relation symbols of maximum arity* 3*.*

Proof. The upper bound holds thanks to Theorem 5.1.3. For the lower bound, we first show, by a reduction from the well-known three-colouring problem, that it is NP-hard to test, for an *acyclic* Boolean conjunctive query Q_1 and a Boolean conjunctive query Q_2, whether $Q_1 \sqsubseteq Q_2$ holds.[5]

To this end, let G be an undirected graph without isolated nodes and self-loops, i.e. edges of the form (v, v). With each node v of G we associate a variable x_v. We define Q_2 to be the Boolean conjunctive query with head $H()$ whose body consists of all atoms $E(x_v, x_w)$ such that (v, w) is an edge of G. The Boolean conjunctive query Q_1 is defined

[5] We think that this is folklore but could not find a reference for it.

by the rule

$$H() \leftarrow E(b,r), E(r,y), E(y,b), E(r,b), E(y,r), E(b,y), C(b,r,y)$$

where b, r, y are variables representing the colours blue, red, and yellow, respectively. Observe that Q_1 is acyclic, because all variables are contained in the "cover-atom" $C(b,r,y)$. Hence, there is a simple join tree for Q_1 that has only two levels, the root node is labelled with $C(b,r,y)$.

Towards the correctness of the reduction, if $Q_1 \sqsubseteq Q_2$ holds, there is a homomorphism h from Q_2 to Q_1. This homomorphism represents a valid colouring, since two variables x_v and x_w cannot be mapped to the same variable in Q_1 if there is an edge (v,w) in G. Conversely, every valid colouring of G gives rise to a homomorphism from Q_2 to Q_1 and therefore witnesses $Q_1 \sqsubseteq Q_2$.

Let $Q_1 \wedge Q_2$ denote the Boolean conjunctive query whose body is $\mathsf{body}(Q_1) \cup \mathsf{body}(Q_2)$. Since it is well-known that $Q_1 \sqsubseteq Q_2$ if and only if $Q_1 \equiv Q_2 \wedge Q_1$, we conclude that the equivalence problem for Boolean conjunctive queries Q_1, Q_2 is NP-hard, even if Q_1 is acyclic.

Finally, we reduce the equivalence problem with acyclic Q_1 to the rewriting problem. For this purpose, let Q_1 and Q_2 be Boolean conjunctive queries. The input instance (Q_1, Q_2) for the equivalence problem is mapped to an input instance for the rewriting problem by assigning Q_1 the role of the query and Q_2 the role of (the sole) view.

The canonical rewriting, if it exists, consists only of the single atom $Q_2()$, since Q_2 is a Boolean query. Thus, there is a $\{Q_2\}$-rewriting of Q_1 if and only if Q_1 is equivalent to Q_2. This reduction establishes that $\textsc{Rewr}^3(\mathsf{CQ}^0, \mathsf{ACQ}, \mathsf{CQ})$ is NP-hard. For the NP-hardness of $\textsc{Rewr}^3(\mathsf{ACQ}^0, \mathsf{CQ}, \mathsf{CQ})$, the roles of Q_1 and Q_2 are simply swapped in the last reduction. □

5.2 A Characterization

For studying the structure of rewritings, and, in particular, asserting the existence of structurally simple rewritings, we will employ a characterization of rewritability. The objective for this section is to introduce this characterization. We note that our characterization is very similar to other such characterizations utilized in the literature [GKC06; AC19], in particular to the "MiniCon Descriptions" studied by Pottinger and Halevy [PH01]. However, in its specific form and notation it is tailored for our needs in the subsequent sections. We will provide a detailed comparison of our characterization with others in Section 5.6.

Before we start with the formal definitions, let us explain the idea by means of an example.

Example 5.2.1. Consider the set $\mathcal{V}$ of the following views.

$$\begin{aligned} V_1(x_1, y_1) &\leftarrow R(x_1, y_1), S(y_1) \\ V_2(x_2) &\leftarrow R(x_2, y_2), S(y_2) \\ V_3(y_3, z_3, w_3) &\leftarrow S(y_3), T(y_3, z_3, w_3) \end{aligned}$$

Furthermore, let Q be the conjunctive query defined by

$$H(y) \leftarrow R(x, y), S(y), T(y, z, z).$$

Our characterization is in terms of a partition of $\mathsf{body}(Q)$, where each set in the partition can be "covered" by a view from $\mathcal{V}$. We will see that, in this example, a suitable partition consists of the two sets $\mathcal{A} = \{R(x, y), S(y)\}$, and $\mathcal{B} = \{T(y, z, z)\}$. The set $\mathcal{A}$ can be "covered" by V_1 quite naturally, since $\mathcal{A}$ coincides, up to variable names, with $\mathsf{body}(V_1)$. Furthermore, the set $\mathcal{B}$ can be "covered" by V_3. However, for the T-atoms to match, it is required to unify the variables z_3 and w_3. This can be achieved using a view application because both variables occur in $\mathsf{head}(V_3)$. We note that the extra atom $S(y_3)$ in $\mathsf{body}(V_3)$ is not problematic here, because it can be mapped into $\mathsf{body}(Q)$, albeit "outside" of $\mathcal{B}$. This informal description of how the sets $\mathcal{A}$ and $\mathcal{B}$ can be "covered" translates directly into the query with body $\{V_1(x, y), V_3(y, z, z)\}$ and head $H(y)$. It is not hard to see that this query is indeed a $\mathcal{V}$-rewriting of Q. Of course, the alternative partition $\{R(x, y)\}, \{S(y), T(y, z, z)\}$ leads to a similar result.

So far we have silently ignored an important aspect: It is crucial that the variable y occurs in both view atoms (and corresponds to y_1 and y_3, respectively), for two reasons. For once it occurs in the head of Q and, hence, contributes to the query result. Secondly, it is necessary to ensure that the relations R, S, and T are properly "joined". In other words, the query result should only contain values that occur in the second component of relation R, in the set S, and the first component of relation T. For this reason, the set $\mathcal{A}$ cannot be "covered" by V_2 instead of V_1. ◁

Formally, the partitions in Example 5.2.1 correspond to *cover partitions*, and the relationship between sets in such partitions and views is represented by *cover descriptions*. The variable y is (a special case of) a *bridge variable*.

We define these notions next. Let Q be a conjunctive query. For Q and a set $\mathcal{A} \subseteq \mathsf{body}(Q)$ we define $\mathsf{bvars}_Q(\mathcal{A})$ as the set of *bridge variables* of $\mathcal{A}$, that occur in $\mathcal{A}$ as well as "outside of $\mathcal{A}$". Here "outside of $\mathcal{A}$" means in the head of Q or in some atom of Q not in $\mathcal{A}$. More formally,

$$\mathsf{bvars}_Q(\mathcal{A}) = \mathsf{vars}(\mathcal{A}) \cap \big(\mathsf{vars}(\mathsf{head}(Q)) \cup \mathsf{vars}(\mathsf{body}(Q) \setminus \mathcal{A})\big).$$

Example 5.2.2. Consider the query Q defined by

$$H(x, y, z) \leftarrow R(x, u), S(u, y, w), T(y, w, z).$$

For the set $\mathcal{A} = \{R(x, u), S(u, y, w)\}$ of atoms from the body of Q, the set of bridge variables is $\mathsf{bvars}_Q(\mathcal{A}) = \{x, y, w\}$ because x and y are head variables of Q and because w, and also y, occurs in the atom $T(y, w, z)$ that does not belong to $\mathcal{A}$. The variable u in $\mathcal{A}$ is not a bridge variable since it is quantified and does not occur in any atom outside of $\mathcal{A}$. ◁

Definition 5.2.3 (Cover Description)**.** A *cover description* d for a query Q is a tuple $\langle \mathcal{A}, V, \alpha, \psi \rangle$ where

- $\mathcal{A}$ is a subset of $\mathsf{body}(Q)$,
- V is a view,
- α is a view application of V, and
- ψ is a mapping from $\mathsf{vars}(\alpha(V))$ to $\mathsf{vars}(Q)$,

such that the following four conditions hold.

(1) $\mathcal{A} \subseteq \alpha(\mathsf{body}(V))$

(2) $\mathsf{bvars}_Q(\mathcal{A}) \subseteq \alpha(\mathsf{vars}(\mathsf{head}(V)))$

(3) ψ is a body homomorphism from $\alpha(V)$ to Q

(4) ψ is the identity on $\mathsf{vars}(\mathcal{A})$

Intuitively, the conditions of Definition 5.2.3 reflect the aspects for V "covering" $\mathcal{A}$ discussed in Example 5.2.1: Condition (1) means that every atom in $\mathcal{A}$ has a matching atom in the body of V. Condition (3) ensures that all atoms in an expansion can be mapped into the query – in particular, this concerns extra atoms such as $S(y_3)$ in Example 5.2.1. Lastly, Conditions (2) and (4) establish an "interface" to combine multiple cover descriptions in a straightforward and compatible manner such that, overall, a rewriting is described. In the following we discuss a slightly more involved example.

Example 5.2.4. Consider the three views

$$\begin{aligned} V_1(x_1, y_1, w_1) &\leftarrow R(x_1, y_1, u_1), T(x_1, v_1), F(v_1), E(w_1), S(w_1, u_1), \\ V_2(x_2, y_2, z_2) &\leftarrow R(x_2, y_2, z_2), F(v_2), \\ V_3(w_3, z_3) &\leftarrow S(w_3, z_3), E(w_3), \end{aligned}$$

and the conjunctive query Q given by the rule

$$H(x, y, z) \leftarrow R(x, y, z), T(x, v), F(v), E(w), S(w, z).$$

The tuple $d_1 = \langle \mathcal{A}_1, V_1, \alpha_1, \psi_1 \rangle$ with $\mathcal{A}_1 = \{T(x, v), F(v)\}$,

$$\begin{aligned} \alpha_1 &= \{x_1 \mapsto x, y_1 \mapsto y', u_1 \mapsto u', v_1 \mapsto v, w_1 \mapsto w'\}, \text{ and} \\ \psi_1 &= \{x \mapsto x, y' \mapsto y, u' \mapsto z, v \mapsto v, w' \mapsto w\} \end{aligned}$$

is a cover description for Q with $\mathsf{bvars}_Q(\mathcal{A}_1) = \{x\}$. Although ψ_1 could be replaced with id (if α_1 is adapted accordingly), we will see in Example 5.2.7, that this is not always desirable. ◁

Now we can characterize rewritability of a conjunctive query Q by the existence of a partition of $\mathsf{body}(Q)$ whose subsets have cover descriptions with views from $\mathcal{V}$.

Definition 5.2.5 (Cover Partition)**.** Let Q be a conjunctive query and $\mathcal{V}$ be a set of views. A collection $\mathcal{P} = \{d_1, \ldots, d_m\}$ of cover descriptions $d_i = \langle \mathcal{A}_i, V_i, \alpha_i, \psi_i \rangle$ for Q with $V_i \in \mathcal{V}$ is a *cover partition* for Q over $\mathcal{V}$ if the sets $\mathcal{A}_1, \ldots, \mathcal{A}_m$ constitute a partition of $\mathsf{body}(Q)$.

Moreover, a cover partition is *consistent* if, for all $i, j \in [1, m]$ with $i \neq j$,

$$\mathsf{vars}(\alpha_i(V_i)) \cap \mathsf{vars}(\alpha_j(V_j)) \subseteq \mathsf{bvars}_Q(\mathcal{A}_i).$$

In other words, a cover partition is consistent if variables of any $\alpha_i(V_i)$ are in the range of any other α_j only if they are bridge variables. This implies quantified variable disjointness.

Lemma 5.2.6. *For every consistent cover partition $\mathcal{P} = \{d_1, \ldots, d_m\}$ of cover descriptions $d_i = \langle \mathcal{A}_i, V_i, \alpha_i, \psi_i \rangle$ for a conjunctive query Q, the view applications $\alpha_1, \ldots, \alpha_m$ fulfil quantified variable disjointness.*

Proof. For the sake of contradiction, assume that $\alpha_1, \ldots, \alpha_m$ do *not* obey quantified variable disjointness. Then there are $i, j \in [1, m]$ with $i \neq j$ such that there is a quantified variable $x \in \mathsf{vars}(V_i)$ and a variable $y \in \mathsf{vars}(V_j)$ with $\alpha_i(x) = \alpha_j(y)$. Let $z = \alpha_i(x) = \alpha_j(y)$. Since $z \in \mathsf{vars}(\alpha_i(V_i)) \cap \mathsf{vars}(\alpha_j(V_j))$ and $\mathcal{P}$ is consistent, z is in $\mathsf{bvars}_Q(\mathcal{A}_i)$. Therefore, it is also in $\alpha_i(\mathsf{head}(V_i))$ due to Condition (2) of Definition 5.2.3. But this means that α_i unifies x with a head variable, which is a contradiction to α_i being a view application. □

Let us point out that, while consistency implies quantified variable disjointness, it asks for more, namely to unify variables only if necessary. In particular, this also concerns head variables. We will see that this leads to, intuitively, rewritings whose atoms are as disconnected as possible – which in turn tends to promote a simple query structure.

Example 5.2.7 (Continuation of Example 5.2.4)**.** Let $d_1 = \langle \mathcal{A}_1, V_1, \alpha_1, \psi_1 \rangle$ be the cover description defined in Example 5.2.4. In addition, we consider the cover descriptions d_2 and d_3 with $d_i = \langle \mathcal{A}_i, V_i, \alpha_i, \psi_i \rangle$ for $i \in \{2, 3\}$ where

$$\mathcal{A}_2 = \{R(x, y, z)\}, \qquad \mathcal{A}_3 = \{E(w), S(w, z)\},$$
$$\alpha_2 = \{x_2 \mapsto x, y_2 \mapsto y, z_2 \mapsto z, v_2 \mapsto v\}, \qquad \alpha_3 = \{w_3 \mapsto w, z_3 \mapsto z\},$$

and $\psi_2 = \psi_3 = \mathsf{id}$.

The cover descriptions d_1, d_2, d_3 constitute a cover partition for Q over $\{V_1, V_2, V_3\}$. It is, however, not consistent, since v is in the range of α_1 and α_2, but *not* in $\mathsf{bvars}_Q(\mathcal{A}_1)$. However, replacing α_2 and ψ_2 by mappings α_2' and ψ_2' with $\alpha_2'(v_2) = v'$ and $\psi_2'(v') = v$ that agree with α_2 and ψ_2 on all other variables, respectively, yields a consistent cover partition. Note that there is no consistent cover partition with a cover description of the form $\langle \mathcal{A}_1, V_1, \alpha_1', \mathsf{id} \rangle$ because necessarily $\alpha_1'(w_1) = w$ would hold, and thus, w would be in the range of α_1' and α_3. ◁

A cover partition $\mathcal{P}$ of cover descriptions $d_i = \langle \mathcal{A}_i, V_i, \alpha_i, \psi_i \rangle$ for a conjunctive query Q over a set $\mathcal{V}$ of views induces a query $\mathsf{q}(\mathcal{P})$ over $\mathcal{S}_\mathcal{V}$ as follows. We note first that each variable in $\mathsf{head}(Q)$ occurs in at least one of the sets $\mathcal{A}_i$ and thus in some set $\mathsf{bvars}_Q(\mathcal{A}_i)$. Therefore, Condition (2) of Definition 5.2.3 guarantees that each head variable of Q occurs in some set $\alpha_i(\mathsf{head}(V_i))$, and thus $\mathcal{P}$ induces the query $\mathsf{q}(\mathcal{P})$ with

$$\mathsf{head}(\mathsf{q}(\mathcal{P})) = \mathsf{head}(Q) \text{ and } \mathsf{body}(\mathsf{q}(\mathcal{P})) = \{\alpha_i(\mathsf{head}(V_i)) \mid 1 \leq i \leq m\}.$$

If $\mathcal{P}$ is consistent it also induces an expansion of $\mathsf{q}(\mathcal{P})$, thanks to quantified variable disjointness being guaranteed, cf. Lemma 5.2.6. The expansion induced by $\mathcal{P}$ is the query $\mathsf{exp}(\mathcal{P})$ with

$$\mathsf{head}(\mathsf{exp}(\mathcal{P})) = \mathsf{head}(Q) \text{ and } \mathsf{body}(\mathsf{exp}(\mathcal{P})) = \bigcup_{i=1}^{m} \alpha_i(\mathsf{body}(V_i)).$$

We stress that there is a one-to-one correspondence between the cover descriptions in $\mathcal{P}$ and the atoms in $\mathsf{body}(\mathsf{q}(\mathcal{P}))$. A fact which we will often employ. We are now ready to state and prove the main result of this section: The existence of a cover partition indeed characterizes rewritability.

Theorem 5.2.8. *Let Q be a minimal conjunctive query and $\mathcal{V}$ be a set of views. The following three statements are equivalent.*

(a) *There is a $\mathcal{V}$-rewriting of Q.*

(b) *There is a cover partition $\mathcal{P}$ for Q over $\mathcal{V}$.*

(c) *There is a consistent cover partition $\mathcal{P}$ for Q over $\mathcal{V}$.*

We prove Theorem 5.2.8 by establishing somewhat stronger results which will be useful in their own right, in particular for designing algorithms. More precisely, Theorem 5.2.8 is a direct consequence of Lemmas 5.2.9 to 5.2.11 stated below.

The following result states that every cover partition can be transformed into a consistent cover partition. This establishes that Statement (b) implies Statement (c).

Lemma 5.2.9. *Let $\mathcal{V}$ be a set of views and Q be a conjunctive query. If there is a cover partition $\mathcal{P}$ for Q over $\mathcal{V}$, then there is a consistent cover partition $\mathcal{P}'$ with the same underlying partition of $\mathsf{body}(Q)$ as $\mathcal{P}$. Moreover, given $\mathcal{P}$, the consistent cover partition $\mathcal{P}'$ can be computed in polynomial time.*

Proof. Let $\mathcal{P} = \{d_1, \ldots, d_m\}$ be a cover partition for Q over $\mathcal{V}$, where, for each i, $d_i = \langle \mathcal{A}_i, V_i, \alpha_i, \psi_i \rangle$. Let z be a variable violating the consistency condition. That is, z occurs in some $\alpha_i(V_i)$, but *not* in $\mathsf{bvars}_Q(\mathcal{A}_i)$, and also in some $\alpha_j(V_j)$ with $j \neq i$. Since $z \notin \mathsf{bvars}_Q(\mathcal{A}_i)$ we have $z \notin \mathsf{vars}(\mathcal{A}_j)$ as well. We define α'_j like α_j but, for some fresh variable z', we set $\alpha'_j(x) = z'$ whenever $\alpha_j(x) = z$. Accordingly, we define ψ'_j like ψ_j but with $\psi'_j(z') = \psi_j(z)$. It is easy to verify that $d'_j = \langle \mathcal{A}_j, V_j, \alpha'_j, \psi'_j \rangle$ is a cover description, which can replace d_j. Repeating this process yields a consistent cover partition over $\mathcal{V}$ for Q. Lastly, note that the number of iterations is bounded polynomially in the number of variables and cover descriptions in $\mathcal{P}$. □

The next result establishes that Statement (c) implies Statement (a).

Lemma 5.2.10. *Let $\mathcal{P}$ be a consistent cover partition for a conjunctive query Q over a set $\mathcal{V}$ of views. Then $\mathsf{q}(\mathcal{P})$ is a $\mathcal{V}$-rewriting of Q. Moreover, $\mathsf{q}(\mathcal{P})$ can be obtained from $\mathcal{P}$ in polynomial time.*

Proof. Let $\mathcal{P} = \{d_1, \ldots, d_m\}$ be a consistent cover partition for Q over $\mathcal{V}$, where, for each i, $d_i = \langle \mathcal{A}_i, V_i, \alpha_i, \psi_i \rangle$.

We prove that the query $\mathsf{q}(\mathcal{P})$ induced by $\mathcal{P}$ is a $\mathcal{V}$-rewriting of Q by establishing that the expansion $\mathsf{exp}(\mathcal{P})$ is equivalent to Q. Proposition 5.1.11 then implies that $\mathsf{q}(\mathcal{P})$ is a rewriting. First, since $\mathcal{A}_1, \ldots, \mathcal{A}_m$ constitute a partition of $\mathsf{body}(Q)$, and thanks to Condition (1) of Definition 5.2.3, the identity mapping id is a homomorphism from Q to $\mathsf{exp}(\mathcal{P})$. Therefore, we have $\mathsf{exp}(\mathcal{P}) \sqsubseteq Q$, and it suffices to show $Q \sqsubseteq \mathsf{exp}(\mathcal{P})$. To this end, we prove that the union of the mappings $\psi_1, \ldots, \psi_m$ is a homomorphism h' from $\mathsf{exp}(\mathcal{P})$ into Q.

We first show that h' is well-defined: Let us assume that a variable z occurs in $\alpha_i(V_i)$ and $\alpha_j(V_j)$ for some $i, j \in [1, m]$ with $i \neq j$. Thanks to consistency, z is in $\mathsf{bvars}_Q(\mathcal{A}_i)$ and $\mathsf{bvars}_Q(\mathcal{A}_j)$. In particular, it occurs in $\mathcal{A}_i$ and $\mathcal{A}_j$. We can conclude that $\psi_i(z) = \psi_j(z)$ holds, thanks to Condition (4) of Definition 5.2.3.

That h' is a homomorphism follows easily, because each ψ_i is a body homomorphism by Condition (3) and h' is the identity on $\mathsf{head}(Q)$ by Conditions (1) and (4).

Given $\mathcal{P}$, computing $\mathsf{q}(\mathcal{P})$ is trivial since it suffices to apply the α_i to the $\mathsf{head}(V_i)$. □

The last step for proving Theorem 5.2.8 is the following result, which states that a cover partition can be derived from a rewriting. This establishes that Statement (a) of Theorem 5.2.8 implies Statement (b). We note that the proof of this result relies on Lemma 5.2.12 which we state after the proof to improve readability.

Lemma 5.2.11. *Let $\mathcal{V}$ be a set of views and Q be a minimal conjunctive query. If there is a $\mathcal{V}$-rewriting Q_R of Q, there also is a cover partition over $\mathcal{V}$ for Q. The number of cover descriptions in the cover partition is bounded by $|\mathsf{body}(Q_R)|$.*

Proof. Let us assume that Q has a $\mathcal{V}$-rewriting Q_R. Let further Q_E be an expansion of Q_R. Thanks to Proposition 5.1.11, we have that Q_E and Q are equivalent. Let the equivalence of Q_E and Q be witnessed by homomorphisms h from Q to Q_E and h' from Q_E to Q.

Since Q is minimal, we can assume, thanks to Lemma 5.2.12, that h is injective and h' is the inverse of h on the atoms of $h(\mathsf{body}(Q))$. Since h is injective, we can further assume, without loss of generality, that h is the identity mapping on the variables in Q and that we have $h'(x) = x$ for every such variable.[6]

That is, we have $\mathsf{head}(Q) = \mathsf{head}(Q_E) = \mathsf{head}(Q_R)$ as well as $\mathsf{body}(Q) \subseteq \mathsf{body}(Q_E)$.

Let $\alpha_1, \ldots, \alpha_m$ be view applications of views $V_1, \ldots, V_m \in \mathcal{V}$ that yield the expansion Q_E. That is, $\alpha_1, \ldots, \alpha_m$ fulfil quantified variable disjointness, and we have

$$\mathsf{body}(Q_E) = \bigcup_{i=1}^{m} \alpha_i(\mathsf{body}(V_i)).$$

The sequence $\alpha_1, \ldots, \alpha_m$ of view applications induces a partition $\mathcal{A}_1, \ldots, \mathcal{A}_m$ of $\mathsf{body}(Q_E)$ as follows. For every $i \in [1, m]$ we define

$$\mathcal{A}_i = \{A \in \mathsf{body}(Q) \mid i \text{ is minimal such that } A \in \alpha_i(\mathsf{body}(V_i))\}.$$

[6] This can easily be achieved by renaming the variables of Q_R and Q_E appropriately.

We note that, in general, some of the $\mathcal{A}_i$ might be empty. In that case, they can just be removed. For convenience, we assume in the following that all $\mathcal{A}_i$ are non-empty.

Let now $d_i = \langle \mathcal{A}_i, V_i, \alpha_i, h'_i \rangle$, where h'_i is the restriction of h' to the variables in $\alpha_i(V_i)$, for every $i \in [1, m]$. To show that $d_1, \ldots, d_m$ constitute a cover partition, it only remains to show that each d_i is a cover description.

To this end, we show that all four conditions of Definition 5.2.3 are satisfied for d_i, for every $i \in [1, m]$. Condition (1) holds by the definition of $\mathcal{A}_i$. Condition (3) is true because each h'_i is a restriction of the body homomorphism h'. Condition (4) follows since h' is the identity on all variables in $\mathsf{body}(Q)$ and thus also on *all* variables of $\mathcal{A}_i \subseteq \mathsf{body}(Q)$.

Hence, it only remains to show that Condition (2) is satisfied. For this purpose, let x be an arbitrary bridge variable in some $\mathcal{A}_i$. If x occurs in a subset $\mathcal{A}_j$ where $j \neq i$, then it is a head variable in both $\alpha_i(V_i)$ and $\alpha_j(V_j)$ because $\alpha_1, \ldots, \alpha_m$ fulfil quantified variable disjointness. Otherwise, $\mathcal{A}_i$ is the only subset containing variable x, which thus has to be a head variable of Q in order to be a bridge variable in $\mathcal{A}_i$. Because of $\mathsf{head}(Q) = \mathsf{head}(Q_R)$, it is then also a head variable of Q_R. This, in turn, implies that x is a head variable of $\alpha_i(V_i)$ because the quantified variables of $\alpha_i(V_i)$ do *not* occur in Q_R. □

The previous proof used the following lemma. It is similarly stated by Chen et al. [Che+20a, Proof of Lemma 2], the authors also provide more details in a full version of their paper [Che+20b, Proof of Lemma 9]. For convenience, we provide a short proof.

Lemma 5.2.12. *Let Q_1 be a minimal conjunctive query, Q_2 be a conjunctive query equivalent to Q_1, and $h_1 \colon Q_1 \to Q_2$ a homomorphism. Then, there is a homomorphism $h_2 \colon Q_2 \to Q_1$ that is the inverse of h_1 on $h_1(\mathsf{body}(Q_1))$.*

Proof. Since Q_1 and Q_2 are equivalent, there is a homomorphism $h'_2 \colon Q_2 \to Q_1$.

The mapping $h'_2 \circ h_1$ is an automorphism on Q_1, because Q_1 is minimal. Since the automorphisms of Q_1 constitute a group, there is some $k > 0$ such that $(h'_2 \circ h_1)^k$ is the identity on Q_1. We choose $h_2 = (h'_2 \circ h_1)^{k-1} \circ h'_2$. Clearly, h_2 is a homomorphism from Q_2 to Q_1 and h_2 is the inverse of h_1 on $h_1(\mathsf{body}(Q_1))$. □

We observe that the proof for Lemma 5.2.11 is partially constructive. Given homomorphisms h and h' as well as view applications $\alpha_1, \ldots, \alpha_m$ that – together – fulfil the assumptions made in the proof, it is straightforward to derive a cover partition from the given rewriting. By carefully combining (and somewhat extending) the proofs for Proposition 5.1.14 and Lemma 5.2.11 we obtain the following result, which is the core of all tractability results in the remainder of this chapter.

Lemma 5.2.13. *For every $k \geq 0$, there is a polynomial time algorithm that, given a set $\mathcal{V} \subseteq \mathsf{ACQ}^k$ of views and a minimal conjunctive query $Q \in \mathsf{ACQ}$ that is $\mathcal{V}$-rewritable, computes a cover partition over $\mathcal{V}$ for Q.*

Proof. The algorithm proceeds in four steps.

The first step is to compute a $\mathcal{V}$-rewriting Q_R of Q. Thanks to Proposition 5.1.14 this can be done in polynomial time (and Q_R is of polynomial size). Let $B_1, \ldots, B_m$ be the view atoms constituting the body of Q_R.

In the second step, the algorithm determines view applications $\alpha_1, \ldots, \alpha_m$ such that the identity mapping is a homomorphism from Q to the expansion of a rewriting (not necessarily Q_R) with respect to $\alpha_1, \ldots, \alpha_m$. For that purpose, it first fixes view applications $\alpha'_1, \ldots, \alpha'_m$ for Q_R according to Definition 5.1.8. That is, each α'_j is a view application for a view $V_j \in \mathcal{V}$ such that $\alpha_j(\mathsf{head}(V_j)) = B_j$. Furthermore, $\alpha'_1, \ldots, \alpha'_m$ fulfil quantified variable disjointness. Since, for each $j \in [1, m]$, the image of head variables from V_j under α'_j is uniquely determined by B_j, computing α'_j boils down to simply renaming quantified variables in $\mathsf{body}(V_j)$. Thus, this can be done in polynomial time.

Next, the algorithm determines a homomorphism h from Q to the expansion Q'_E of Q_R with respect to $\alpha'_1, \ldots, \alpha'_m$. As detailed in the proof of Proposition 5.1.14 testing for the existence of a homomorphism can be done in polynomial time, thanks to Q being acyclic [Yan81]. Furthermore, there is a homomorphism from Q to Q'_E, since Q_R is a rewriting of Q. Such a homomorphism h can thus be determined as follows. Pick a variable $x \in \mathsf{vars}(Q)$ and let R_x be a fresh, unary relation symbol for x. For each variable $y \in \mathsf{vars}(Q'_E)$, the algorithm adds the atom $R_x(x)$ to $\mathsf{body}(Q)$ and the atom $R_x(y)$ to $\mathsf{body}(Q'_E)$, and then tests whether there still is a homomorphism. Observe that adding unary atoms to an acyclic query always results in an acyclic query again. If the test succeeds for a variable y, there is a homomorphism that maps x to y. The algorithm fixes such a variable y as the image $h(x)$ by keeping the respective R_x-atoms. Repeating this procedure for every variable in $\mathsf{vars}(Q)$ yields the desired homomorphism h. We note that, overall, at most $|\mathsf{vars}(Q)| \cdot |\mathsf{vars}(Q'_E)|$ tests are required. Thus, this step can be performed in polynomial time.

Since Q is minimal, the homomorphism h is injective. Thus, renaming variables in the ranges of the α'_j appropriately yields view applications $\alpha_1, \ldots, \alpha_m$ such that the query with body $\alpha_1(\mathsf{body}(V_1)), \ldots, \alpha_m(\mathsf{body}(V_m))$ and head $\mathsf{head}(Q)$ is a rewriting of Q, and the identity mapping is a homomorphism from Q to the expansion Q_E of this rewriting with respect to $\alpha_1, \ldots, \alpha_m$.

The third step is to compute a homomorphism h' from Q_E to Q that is the inverse of h on $\mathsf{body}(Q)$ – that is, the identity on $\mathsf{body}(Q)$ because we have $h = \mathsf{id}$. Thanks to Lemma 5.2.12 we know that such a homomorphism exists. In principle, it can thus be computed analogously to the homomorphism h from Q to Q'_E in the second step. Here the identity mappings for variables x in $h(\mathsf{body}(Q)) = \mathsf{body}(Q)$ are fixed a priori by adding the atom $R_x(x)$ to both queries. We note that, while Q_E is *not* necessarily acyclic, it is *semantically acyclic*. That is, it is equivalent to an acyclic query, namely Q, since it is the expansion of a rewriting of Q. Barceló et al. [BRV16, Theorem 2.2] proved that semantically acyclic conjunctive queries can be evaluated in polynomial time. Thus, the existence of a homomorphism from Q_E to Q can be asserted in polynomial time as well.[7]

Finally, the last step is to obtain the cover partition using the homomorphisms h and h', and the view applications $\alpha_1, \ldots, \alpha_m$. This step can be done completely analogous as in the proof for Lemma 5.2.11. It is not hard to see that this procedure, and in particular

[7] We note that this approach does *not* yield an efficient procedure to test for the existence of a rewriting, because in this case it is *not* guaranteed that the expansion is semantically acyclic.

the partitioning of $\mathsf{body}(Q)$, runs in polynomial time. □

Remark 5.2.14. We note that the requirement in Theorem 5.2.8 and Lemmas 5.2.11 and 5.2.13 for Q to be minimal is only seemingly a restriction in our setting, since we apply it only to acyclic queries. If Q is acyclic but not minimal, an equivalent minimal query Q' can be computed in polynomial time by iteratively removing atoms from its body and testing containment [CM77; CR00, Corollary 1]. Moreover, it is guaranteed that the minimal query Q' is also acyclic. We believe this to be folklore, it also follows readily from more general results [cf., for instance, BPR17, Proposition 3].

The same is true for free-connex acyclic queries: Every homomorphism from Q to Q' is also a homomorphism from $\mathsf{body}(Q) \cup \{\mathsf{head}(Q)\}$ to $\mathsf{body}(Q') \cup \{\mathsf{head}(Q')\}$ and vice versa, since the relation symbol of $\mathsf{head}(Q) = \mathsf{head}(Q')$ does not occur in $\mathsf{body}(Q')$. Thus, Q' is minimal if and only if the Boolean query whose body is $\mathsf{body}(Q') \cup \{\mathsf{head}(Q')\}$ is minimal. Therefore, if $\mathsf{body}(Q) \cup \{\mathsf{head}(Q)\}$ is acyclic, so is $\mathsf{body}(Q') \cup \{\mathsf{head}(Q')\}$. In other words, if Q is free-connex acyclic, then Q' is free-connex acyclic as well.

For hierarchical and q-hierarchical queries the same holds: It is easy to see that removing atoms does not change the conditions in their respective definitions (cf. Definition 2.4.2).

5.3 Towards Acyclic Rewritings

In this section, we turn our focus to the main topic of this chapter: Acyclic rewritings and the *acyclic rewriting problem* – the decision problem that asks whether an acyclic rewriting exists. We will study the complexity of $\textsc{Rewr}(\mathbb{V}, \mathbb{Q}, \mathsf{ACQ})$ in Section 5.3.2, for the case that $\mathbb{V}$ and $\mathbb{Q}$ are the class of conjunctive queries as well as for various cases where $\mathbb{V}$ and $\mathbb{Q}$ are subclasses of acyclic conjunctive queries. For that purpose, we first study the *existence* of acyclic rewritings in Section 5.3.1, specifically for the case that the given query is acyclic, i.e. in case $\mathbb{Q} = \mathsf{ACQ}$ and $\mathbb{V} = \mathsf{CQ}$.

5.3.1 On the Existence of Acyclic Rewritings for Acyclic Queries

We start by discussing some examples to gain some insights into the existence of acyclic rewritings and their relationship with canonical candidates. The following example illustrates that, even if an acyclic rewriting exists, the canonical rewriting is *not* necessarily acyclic. Furthermore, it may be that an acyclic rewriting cannot be obtained from the canonical rewriting by removing atoms, because each "subquery" of the canonical rewriting is cyclic or *not* a rewriting, and thus none of them is an acyclic rewriting.

Example 5.3.1. Consider the set $\mathcal{V}$ consisting of the following three views.

$$V_1(u_1, v_1) \leftarrow R_1(u_1), R_2(v_1) \qquad V_2(u_2, v_2) \leftarrow S(u_2, v_2) \qquad V_3(u_3, v_3) \leftarrow T_1(u_3), T_2(v_3)$$

Let further Q be the acyclic conjunctive query given by the rule

$$H() \leftarrow R_1(x), R_2(y), S(x, z), T_1(z), T_2(y).$$

The canonical candidate Q_R is $H() \leftarrow V_1(x,y), V_2(x,z), V_3(z,y)$. It is a $\mathcal{V}$-rewriting of Q, but it is *cyclic*. Each query whose body is a proper subset of $\mathsf{body}(Q_R)$ is not a $\mathcal{V}$-rewriting of Q. However, there is an *acyclic* $\mathcal{V}$-rewriting of Q, namely the query Q'_R defined by

$$H() \leftarrow V_1(x,y), V_2(x,z), V_3(z,y'), V_3(z',y). \qquad \triangleleft$$

Example 5.3.1 suggests that an acyclic rewriting may have more atoms than a "general" rewriting. In particular, there may *not* be an acyclic rewriting which is "minimal" – in the sense that there is another rewriting with less atoms. Furthermore, an acyclic rewriting may have more variables than an arbitrary rewriting *and* the original query.[8]

Despite these differences, it turns out that it is always possible to obtain an acyclic rewriting from an arbitrary rewriting. Furthermore, our proof implies that Lemma 5.1.12 transfers to acyclic rewritings: If there is one at all, there is an acyclic rewriting whose length is bounded by the length $|Q|$ of the original query Q.

Towards a proof of these claims, we have a closer look at the structure of cover descriptions, with the aim to determine the circumstances under which a decomposition of view atoms as in Example 5.3.1 is possible.

Example 5.3.2 (Continuation of Example 5.3.1)**.** We first have a closer look at the V_3-atoms in the rewritings of Example 5.3.1. The V_3-atom $V_3(z,y)$ in the cyclic rewriting Q_R corresponds to a cover description $d = \langle \mathcal{A}, V_3, \alpha, \mathsf{id} \rangle$ with $\mathcal{A} = \{T_1(z), T_2(y)\}$, $\alpha(u_3) = z$, and $\alpha(v_3) = y$. We observe that the set $\mathcal{A}$ is *not* connected, the two atoms in $\mathcal{A}$ do *not* share any variable. This allows us to "split" the cover description into the two cover descriptions

$$d_1 = \langle \{T_1(z)\}, V_3, \{u_3 \mapsto z, v_3 \mapsto y'\}, \{z \mapsto z, y' \mapsto y\} \rangle, \text{ and}$$
$$d_2 = \langle \{T_2(y)\}, V_3, \{u_3 \mapsto z', v_3 \mapsto y\}, \{z' \mapsto z, y \mapsto y\} \rangle.$$

Indeed, these cover descriptions correspond to the two V_3-atoms $V_3(z,y')$ and $V_3(z',y)$ of the acyclic rewriting Q'_R. It is not hard to see that replacing d with d_1 and d_2 in any cover partition for Q yields again a cover partition for Q, as long as y' and z' are fresh variables that do *not* occur in the original partition.

Such a replacement can also be understood as replacing a view atom in a rewriting with a set of view atoms which is "equivalent with respect to expansions". Indeed, the two V_3-atoms in the acyclic rewriting Q'_R constitute a (sub-)query with body $\{V_3(z,y'), V_3(z',y)\}$ and head variables z, y, whereas the V_3-atom in the cyclic rewriting Q_R constitutes a (sub-)query with body $\{V(z,y)\}$ and head variables z, y. The expansions of these two (sub-)queries are equivalent.

We note that one could also replace the (cover description corresponding to the) V_1-atom instead of the V_3-atom in Q_R to obtain an acyclic rewriting. $\triangleleft$

In Example 5.3.1 the acyclic rewriting was obtained by "splitting" cover descriptions, and thereby effectively refining a cover partition. The refinement in the example is

[8] We note that, if a conjunctive query Q is rewritable, then it has a rewriting that uses only variables from Q, namely the canonical rewriting.

encouraged by the non-connectedness of the two atoms of Q "covered" by the original view atom $V_3(z, y)$. We note that, given a rewriting alone, it may *not* always be obvious which atoms are "covered" by which view atom. A cover partition, on the other hand, readily provides this information.

One might suspect that it is also important that the body of the view is *not* connected. The next example illustrates that this is *not* the case, and indeed, we will see that, for the existence of an acyclic rewriting, the structure of the views is not important at all.[9]

Example 5.3.3. Consider the two views

$$V_1(x_1, y_1) \leftarrow E(x_1, y_1) \quad \text{and} \quad V_2(x_2, y_2) \leftarrow R(x_2), E(x_2, z_2), E(z_2, y_2), S(y_2),$$

and the acyclic conjunctive query Q defined by

$$H(z) \leftarrow R(x), E(x, z), E(z, y), S(y).$$

The canonical candidate is the *cyclic* rewriting

$$H(z) \leftarrow V_1(x, z), V_1(z, y), V_2(x, y).$$

The view atom $V_2(x, y)$ corresponds to a cover description $d = \langle \mathcal{A}, V_2, \alpha, \psi \rangle$ where $\mathcal{A} = \{R(x_2), S(y_2)\}$. Note that V_2 cannot cover the E-atoms because z is a head variable. Although $\mathsf{body}(V_2)$ is connected, the cover description d can be "split" because $\mathcal{A}$ is *not* connected. This yields an acyclic rewriting, for instance

$$H(z) \leftarrow V_1(x, z), V_1(z, y), V_2(x, y'), V_2(x', y). \qquad \triangleleft$$

The next – and final – example illustrates that a refinement of the cover partition as done in Examples 5.3.2 and 5.3.3 is on its own *not* always sufficient to obtain an acyclic rewriting. It is also important that the corresponding cover partition is consistent.

Example 5.3.4. Consider the views

$$\begin{aligned} V_1(x_1, y_1, w_1) &\leftarrow R(x_1, y_1, v_1), E_1(x_1), S(w_1, v_1), \\ V_2(x_2, y_2, z_2) &\leftarrow R(x_2, y_2, z_2), E_2(y_2), \\ V_3(w_3, z_3) &\leftarrow S(w_3, z_3), E_3(w_3), \end{aligned}$$

and the conjunctive query Q defined by

$$H(x, y, z) \leftarrow R(x, y, z), E_1(x), E_2(y), E_3(w), S(w, z).$$

The canonical candidate $H(x, y, z) \leftarrow V_1(x, y, w), V_2(x, y, z), V_3(w, z)$ is a cyclic rewriting of Q. It corresponds to a cover partition $\mathcal{P} = \{d_1, d_2, d_3\}$ where the cover descriptions have the shape $d_i = \langle \mathcal{A}_i, V_i, \alpha_i, \alpha_i \rangle$, and the sets $\mathcal{A}_1$, $\mathcal{A}_2$, $\mathcal{A}_3$ are as follows.

$$\mathcal{A}_1 = \{E_1(x)\} \quad \mathcal{A}_2 = \{R(x, y, z), E_2(y)\} \quad \mathcal{A}_3 = \{S(w, z), E_3(w)\}$$

[9] It plays, however, an important role for our tractability results.

We note that neither the R-atom nor the S-atom can be in $\mathcal{A}_1$ because z is a head variable but $v_1 \notin \mathsf{vars}(\mathsf{head}(V_1))$. In contrast to Examples 5.3.2 and 5.3.3, the sets $\mathcal{A}_1$, $\mathcal{A}_2$, and $\mathcal{A}_3$ are all connected.

We observe that to yield the view atom $V_1(x, y, z)$ the variables y and z have to be in the range of the view application α_1. However, they are also in the range of α_2, but *not* bridge variables of $\mathcal{A}_1$, since they do *not* even occur in $\mathcal{A}_1$. Thus, $\mathcal{P}$ is *not* consistent, because α_1 and α_2 violate the condition.

The procedure guaranteed by Lemma 5.2.9 to obtain a consistent cover partition yields the following acyclic rewriting of Q.

$$H(x, y, z) \leftarrow V_1(x, y', w'), V_2(x, y, z), V_3(w, z) \qquad \triangleleft$$

We conclude the discussion with the main result of this section.

Theorem 5.3.5. *Let $\mathcal{V}$ be a set of views and Q be a conjunctive query.*

(a) *If Q is acyclic and $\mathcal{V}$-rewritable, then there is an acyclic $\mathcal{V}$-rewriting of Q.*

(b) *If Q is free-connex acyclic and $\mathcal{V}$-rewritable, then there is a free-connex acyclic $\mathcal{V}$-rewriting of Q.*

The rewritings guaranteed by Statements (a) and (b) have length at most $|Q|$. Moreover, given a cover partition over $\mathcal{V}$ for Q, an acyclic (or free-connex acyclic, respectively) rewriting of Q can be computed in polynomial time.

Proof. Since Q is acyclic, we can assume that it is minimal, thanks to Remark 5.2.14. Moreover, it has a join tree T_Q, and, thanks to Theorem 5.2.8 and Q being $\mathcal{V}$-rewritable, there is a consistent cover partition $\mathcal{P} = \{d_1, \ldots, d_m\}$ for Q over $\mathcal{V}$, and the query $\mathsf{q}(\mathcal{P})$ is a $\mathcal{V}$-rewriting of Q. For each i, let $d_i = \langle \mathcal{A}_i, V_i, \alpha_i, \psi_i \rangle$.

We show first that $\mathsf{q}(\mathcal{P})$ is acyclic if each set $\mathcal{A}_i$ is connected in T_Q. Afterwards we show that a consistent cover partition with that property can always be constructed, given $\mathcal{P}$.

To this end, we construct a join tree $T_{\mathcal{P}}$ for $\mathsf{q}(\mathcal{P})$: We first cluster, for each j, the nodes for $\mathcal{A}_j$ in T_Q together into one node that is labelled with $\alpha_j(\mathsf{head}(V_j))$. Since the $\mathcal{A}_j$ are connected in T_Q, the resulting graph $T_{\mathcal{P}}$ is a tree.

To verify that $T_{\mathcal{P}}$ is a join tree, let us consider two nodes v, w of $T_{\mathcal{P}}$ labelled by $\alpha_i(\mathsf{head}(V_i))$ and $\alpha_j(\mathsf{head}(V_j))$, respectively, and x be a variable that appears in $\alpha_i(\mathsf{head}(V_i))$ *and* $\alpha_j(\mathsf{head}(V_j))$. Thanks to $\mathcal{P}$ being consistent, the variable x appears in $\mathsf{bvars}_Q(\mathcal{A}_i)$ and $\mathsf{bvars}_Q(\mathcal{A}_j)$. In particular, this means that x appears in two atoms $A \in \mathcal{A}_i$ and $A' \in \mathcal{A}_j$. Moreover, since T_Q is a join tree, x appears in every node on the (shortest) path from A to A' in T_Q. But that means x is in $\mathsf{bvars}_Q(\mathcal{A}_\ell)$ for every $\alpha_\ell(\mathsf{head}(V_\ell))$ along the corresponding contracted path in $T_{\mathcal{P}}$ from v to w. Thanks to Condition (2) of Definition 5.2.3, x thus appears in all labels $\alpha_\ell(\mathsf{head}(V_\ell))$ on the path from v to w. Thus, v and w are x-connected. We conclude that $T_{\mathcal{P}}$ is a join tree for $\mathsf{q}(\mathcal{P})$. Hence, $\mathsf{q}(\mathcal{P})$ is acyclic, and we have thus established Statement (a).

If Q is also free-connex acyclic, there is a join tree T_Q^+ for $\mathsf{body}(Q) \cup \{\mathsf{head}(Q)\}$. As we show later, we can assume that the sets $\mathcal{A}_i$ of the cover partition $\mathcal{P}$ are also connected

in T_Q^+. Clustering the nodes[10] of T_Q^+ analogously as for T_Q yields a join tree for the Boolean query with body $\mathsf{body}(\mathsf{q}(\mathcal{P})) \cup \{\mathsf{head}(\mathsf{q}(\mathcal{P}))\}$ since $\mathsf{head}(\mathsf{q}(\mathcal{P})) = \mathsf{head}(Q)$ and all head variables in a set $\mathcal{A}_i$ also occur in the new label $\alpha_i(\mathsf{head}(V_i))$, thanks to Condition (2) of Definition 5.2.3. We can conclude that Statement (b) holds.

It remains to show how, from a consistent cover partition $\mathcal{P} = \{d_1, \ldots, d_m\}$ with cover descriptions $d_i = \langle \mathcal{A}_i, V_i, \alpha_i, \psi_i \rangle$, for all $i \in [1, m]$, we can construct a consistent cover partition $\mathcal{P}' = \{d'_1, \ldots, d'_n\}$ such that each set $\mathcal{A}'_i$ from $\mathcal{P}'$ is connected in T_Q. To this end, let us assume that some set $\mathcal{A}_j$ is not connected in T_Q. Let $\mathcal{B} \subseteq \mathcal{A}_j$ be a maximally connected subset and let $\mathcal{A}'_j = \mathcal{A}_j \setminus \mathcal{B}$. We observe that each variable x that appears in $\mathcal{B}$ and $\mathcal{A}'_j$, is also in $\mathsf{bvars}_Q(\mathcal{A}_j)$, since x has to occur in at least one other atom in T_Q (otherwise, $\mathcal{B}_j$ and $\mathcal{A}'_j$ would be connected in T_Q). Thus, $\mathsf{bvars}_Q(\mathcal{B}_j) \subseteq \mathsf{bvars}_Q(\mathcal{A}_j)$. We conclude that $d_\mathcal{B} = \langle \mathcal{B}, V_j, \alpha_j, \psi_j \rangle$ is a cover description. Likewise, $d'_j = \langle \mathcal{A}'_j, V_j, \alpha_j, \psi_j \rangle$ is a cover description. The cover description d_j is then replaced by $d_\mathcal{B}$ and d'_j. Repeatedly applying this modification step, eventually yields a cover partition $\mathcal{P}' = \{d'_1, \ldots, d'_n\}$ in which each set $\mathcal{A}'_i$ is connected. If Q is free-connex, then $\mathcal{P}'$ can be further refined as described above but along T_Q^+ instead of T_Q. Indeed, refining the cover partition iteratively along T_Q and T_Q^+ yields a cover partition $\mathcal{P}'' = \{d''_1, \ldots, C''_p\}$ in which each set $\mathcal{A}''_i$ is connected in T_Q and T_Q^+. The number of iterations is bounded by the number of atoms of Q, and thus polynomial.

Thanks to Lemma 5.2.9, $\mathcal{P}''$ can be turned into a consistent cover partition without changing the underlying partition of $\mathsf{body}(Q)$ in polynomial time (and the required renaming of variables does not affect the connectedness). Note that the same can be done for the original cover partition, if necessary. Finally, observe that $\mathcal{P}''$ consists of at most $|\mathsf{body}(Q)|$ cover descriptions, and thus, $\mathsf{q}(\mathcal{P}'')$ has length at most $|Q|$. □

Theorem 5.3.5 delivers good news as well as bad news. The good news is that, in combination with Proposition 5.1.14, it yields that Rewr(ACQ^k, ACQ, ACQ) is in P, for every $k \geq 0$. Since it is possible to efficiently compute a cover partition in this case, thanks to Lemma 5.2.13, we get the following corollary.

Corollary 5.3.6. *For every $k \geq 0$,* Rewr(ACQ^k, ACQ, ACQ) *is in* P*, and an acyclic rewriting can be computed in polynomial time, if it exists.*

The bad news is that, because Rewr($\mathbb{V}$, ACQ, CQ) and Rewr($\mathbb{V}$, ACQ, ACQ) are essentially the same problem, lower bounds for Rewr($\mathbb{V}$, ACQ, CQ) also apply to Rewr($\mathbb{V}$, ACQ, ACQ). Particularly, we get the following, since Rewr^k(CQ, ACQ, CQ) is NP-hard due to Proposition 5.1.15.[11]

Corollary 5.3.7. *For every $k \geq 3$, the problem* Rewr^k(CQ, ACQ, ACQ) *and, therefore, also* Rewr^k(CQ, CQ, ACQ) *is* NP*-hard.*

[10] The node labelled $\mathsf{head}(Q)$ is not clustered with any other node, since it does not occur in any $\mathcal{A}_i$.

[11] We note that Corollary 5.3.7 also follows from the proof of Proposition 5.1.15, since the canonical candidate constructed in that proof is trivially acyclic. Indeed, NP-hardness of Rewr(ACQ, CQ, ACQ) is also implied.

Corollaries 5.3.6 and 5.3.7 leave the complexity of the problems Rewr(ACQ, ACQ, CQ) and Rewr(ACQ, ACQ, ACQ) as well as their restrictions to fixed database schemas open. In the next subsection we will resolve this.

5.3.2 The Complexity of the Acyclic Rewriting Problem

It may be tempting to assume that, since acyclic queries are so well-behaved in general, it should be tractable to decide whether for an acyclic query and a set of acyclic views there exists an acyclic rewriting. However, as we show next, this is (probably) not the case, and this surprising finding even holds for the even better behaved hierarchical conjunctive queries as well.

Theorem 5.3.8. $\text{Rewr}^k(\mathsf{ACQ}, \mathsf{ACQ}, \mathsf{CQ})$ *and* $\text{Rewr}^k(\mathsf{ACQ}, \mathsf{ACQ}, \mathsf{ACQ})$*, as well as* $\text{Rewr}^k(\mathsf{HCQ}, \mathsf{HCQ}, \mathsf{CQ})$ *and* $\text{Rewr}^k(\mathsf{HCQ}, \mathsf{HCQ}, \mathsf{ACQ})$ *are* NP*-complete, for* $k \geq 3$*. The lower bounds even hold for instances with a single view.*

Of course, Theorem 5.3.8 immediately implies NP-hardness of $\text{Rewr}^k(\mathbb{V}, \mathbb{Q}, \mathsf{ACQ})$ and $\text{Rewr}(\mathbb{V}, \mathbb{Q}, \mathsf{ACQ})$, for all pairs $\mathbb{V}, \mathbb{Q}$ of classes with $\mathsf{HCQ} \subseteq \mathbb{V} \subseteq \mathsf{CQ}$ and $\mathsf{HCQ} \subseteq \mathbb{Q} \subseteq \mathsf{CQ}$. All of these problems are also in NP, thanks to Theorem 5.1.3 and Theorem 5.3.5.

In Section 5.5 we will prove an analogue of Theorem 5.3.5 for hierarchical conjunctive queries. Consequently, this will let us conclude that $\text{Rewr}^k(\mathbb{V}, \mathbb{Q}, \mathsf{HCQ})$ and $\text{Rewr}(\mathbb{V}, \mathbb{Q}, \mathsf{HCQ})$, for $\mathbb{V}$, $\mathbb{Q}$ as above, are also NP-complete, cf. Corollary 5.5.5.

Towards a proof for Theorem 5.3.8, recall that deciding whether a rewriting exists is the same as deciding whether a cover partition exists, thanks to our characterization, stated as Theorem 5.2.8. We show next that even deciding whether a single cover description exists is NP-hard, given a query, a set of atoms, and a single view as input. The lower bound proof of Theorem 5.3.8 is then by reduction from the *cover description problem* $\text{CovDesc}(\mathbb{V}, \mathbb{Q})$ which is defined as follows, for classes $\mathbb{V} \subseteq \mathsf{CQ}$ and $\mathbb{Q} \subseteq \mathsf{CQ}$ of conjunctive queries.

— $\text{CovDesc}(\mathbb{V}, \mathbb{Q})$ —

Given: View $V \in \mathbb{V}$, conjunctive query $Q \in \mathbb{Q}$, subset $\mathcal{A} \subseteq \mathsf{body}(Q)$

Question: Are there mappings α and ψ such that $\langle \mathcal{A}, V, \alpha, \psi \rangle$ is a cover description?

Again, we denote by $\text{CovDesc}^k(\mathbb{V}, \mathbb{Q})$ the restriction of $\text{CovDesc}(\mathbb{V}, \mathbb{Q})$, where the arity of each relation symbol in the database schema is bounded by k.

At the first glance $\text{CovDesc}^k(\mathbb{V}, \mathbb{Q})$ may appear simpler than the corresponding acyclic rewriting problem $\text{Rewr}^k(\mathbb{V}, \mathbb{Q}, \mathsf{ACQ})$. However, as it turns out the cover description problem is NP-hard not only for hierarchical queries but even for q-hierarchical conjunctive queries. In fact, our reduction from the cover description problem to the rewriting problem will not preserve q-hierarchical queries, and we will prove that the acyclic rewriting problem for q-hierarchical queries is in P, if the arity of the relations in the database schema is bounded, cf. Corollary 5.5.4.

Theorem 5.3.9. CovDesc^k(QHCQ, QHCQ) *is* NP*-hard, for* $k \geq 2$*. This holds even for instances with minimal Boolean queries* Q *and* $\mathcal{A} = \mathsf{body}(Q)$*.*

Proof. We reduce the satisfiability problem Sat for propositional formulas[12] to the cover description problem CovDesc^k(QHCQ, QHCQ).

For a propositional formula f in conjunctive normal form with at least one literal per clause, we describe how a query Q and a view V can be derived in polynomial time such that

- Q and V are q-hierarchical, and
- f is satisfiable if and only if there are mappings α and ψ such that $\langle \mathsf{body}(Q), V, \alpha, \psi \rangle$ is a cover description for Q.

In this proof we will use the term *proposition* for propositional variables of a formula f in order to easily distinguish them from variables occurring in queries and views.

Construction. Let $f = c_1 \wedge \cdots \wedge c_m$ be a propositional formula in conjunctive normal form over propositions $X_1, \ldots, X_n$ with clauses $c_1, \ldots, c_m$, for some $m \geq 1$. Without loss of generality we assume that *no* constants occur in f, i.e. each clause is a (non-empty) disjunction of literals. The database schema consists of a relation symbol C_j, for each clause c_j, and a "special" relation symbol TruthMap, all of which have arity 2.

The Query. We start with the construction of the q-hierarchical conjunctive query Q. This query is Boolean, say with head $H()$, and is defined over only two variables w_0 and w_1, which are intended to represent the truth values `false` and `true`, respectively.

The body of Q is defined as the union

$$\big\{\mathrm{TruthMap}(w_0, w_1)\big\} \cup \mathcal{C}_0 \cup \mathcal{C}_1$$

of the following sets of atoms.

- The set $\mathcal{C}_0$ consists of the atoms $C_j(w_0, w_1)$, for each clause c_j of the formula f.

  Intuitively, these atoms represent *unsatisfied* clauses as opposed to the next atoms that represent *satisfied* clauses. Note that the variables w_0 and w_1 are swapped in these atoms (and this is the only difference).

- The set $\mathcal{C}_1$ consists of the atoms $C_j(w_1, w_0)$, for each clause c_j of f.

The purpose of the TruthMap-atom is to "fix" the truth values represented by w_0 and w_1.

We observe that Q is minimal, because unifying w_0 and w_1 does *not* yield an equivalent query. Moreover, Q is indeed q-hierarchical: It is hierarchical because both variables w_0 and w_1 occur in all atoms, and hence we have $\mathsf{atoms}(w_0) = \mathsf{atoms}(w_1)$. The additional condition imposed on q-hierarchical queries is trivially satisfied for Boolean conjunctive queries.

[12] As one of the problems originally considered by Karp [Kar72, Main Theorem, Problem 11] it is NP-hard.

The View. We now proceed with the construction of the view V. Like for the query Q, we use the variables w_0 and w_1, with the same intended meaning as before, but also additional variables for the propositions $X_1, \ldots, X_n$ of the formula f. More precisely, for each proposition X_i, we use two variables x_i and $\bar{x}_i$, intended to represent the positive literal X_i, and the negated literal $\neg X_i$, respectively.

In contrast to the Boolean query Q, the view V is a full query. That is, its head is $\mathsf{head}(V) = V(w_0, w_1, x_1, \bar{x}_1, \ldots, x_n, \bar{x}_n)$. The body of V is defined as the union

$$\{\mathrm{TruthMap}(w_0, w_1)\} \cup \mathcal{C}_0 \cup (\mathcal{X}_1^+ \cup \mathcal{X}_1^-) \cup \cdots \cup (\mathcal{X}_n^+ \cup \mathcal{X}_n^-)$$

of sets of atoms. The sets $\{\mathrm{TruthMap}(w_0, w_1)\}$ and $\mathcal{C}_0$ are the same as for Q. Notably, the set $\mathcal{C}_1$ is *not* part of $\mathsf{body}(V)$.

The new sets are defined as follows.

- For each positive literal $L = X_i$ and each clause c_j of f that contains L, the set $\mathcal{X}_i^+$ contains the atom $C_j(x_i, \bar{x}_i)$.
- For each negated literal $L = \neg X_i$ and each clause c_j of f that contains L, the set $\mathcal{X}_i^-$ contains the atom $C_j(\bar{x}_i, x_i)$.

  Note that, in comparison with $\mathcal{X}_i^+$, the variables x_i and $\bar{x}$ are swapped.

The intention for $\mathcal{X}_i^+$ and $\mathcal{X}_i^+$ is to be mapped to $\mathcal{C}_0 \cup \mathcal{C}_1$ in $\mathsf{body}(Q)$, effectively yielding a truth assignment that satisfies every clause. Note that, by construction, such a mapping will always be consistent in the sense that each $(x_i, \bar{x}_i)$ is mapped to either (w_0, w_1) or (w_1, w_0), that is, $\bar{x}_i$ is mapped to the complementary truth value "assigned to" x_i. Further, since each proposition X_i occurs – positively or negated – in f, all x_i and $\bar{x}_i$ occur in V.

Before we prove the correctness of our construction, we show that V is q-hierarchical. Every atom in $\mathsf{body}(V)$ contains either w_0 and w_1 or x_i and $\bar{x}_i$, for some $i \in [1, n]$, but *no* other variable. We thus have $\mathsf{atoms}(w_0) = \mathsf{atoms}(w_1)$, and $\mathsf{atoms}(x_i) = \mathsf{atoms}(\bar{x}_i)$, for all $i \in [1, n]$, as well as $\mathsf{atoms}(y) \cap \mathsf{atoms}(z) = \emptyset$ for every other pair y, z of variables with $y \neq z$ in $\mathsf{vars}(V)$. Therefore, V is hierarchical because every pair of variables satisfies (at least) one condition of Definition 2.4.2. Furthermore, V is q-hierarchical because full conjunctive queries satisfy the additional condition trivially, since every variable is a head variable.

Clearly, Q and V can be computed in polynomial time, given a propositional formula f in conjunctive normal form.

Correctness. We now prove that the formula f is satisfiable if and only if there are mappings α and ψ such that $\langle \mathsf{body}(Q), V, \alpha, \psi \rangle$ is a cover description for Q.

First, we show that, if f is satisfiable, then there are mappings α and ψ such that $\langle \mathsf{body}(Q), V, \alpha, \psi \rangle$ is a cover description for Q. To this end, suppose that f is satisfiable, and let $\beta \colon \{X_1, \ldots, X_n\} \to \{0, 1\}$ be a truth assignment satisfying f. We define the view application α of V as follows. We set $\alpha(w_0) = w_0$, $\alpha(w_1) = w_1$, and, for every $i \in [1, m]$,

- $\alpha(x_i) = w_0$ and $\alpha(\bar{x}_i) = w_1$, if $\beta(X_i) = 0$, or

▶ $\alpha(x_i) = w_1$ and $\alpha(\bar{x}_i) = w_0$, if $\beta(X_i) = 1$.

Furthermore, we define ψ to be the identity on the variables of $\alpha(V)$.

We prove that $\langle \mathsf{body}(Q), V, \alpha, \psi \rangle$ is a cover description for Q by establishing that the conditions of Definition 5.2.3 hold. Since Q is a Boolean query, the set $\mathcal{A} = \mathsf{body}(Q)$ has *no* bridge variables. Therefore, Condition (2) holds trivially. For Condition (4) it suffices to observe that $\mathsf{vars}(\mathcal{A}) = \{w_0, w_1\}$, and, since α is the identity on these variables, so is ψ, because it is the identity on all variables in the range of α. To establish that Conditions (1) and (3) hold, we prove next that $\mathsf{body}(Q) = \alpha(\mathsf{body}(V))$. Since ψ is the identity mapping, this indeed implies Conditions (1) and (3). Overall we can then conclude that $\langle \mathsf{body}(Q), V, \alpha, \psi \rangle$ is a cover description for Q.

To prove $\mathsf{body}(Q) = \alpha(\mathsf{body}(V))$, we show that α sets up the following relationships between the atoms in $\mathsf{body}(V)$ (on the left-hand side) and the atoms in $\mathsf{body}(Q)$ (on the right-hand side).

(i) $\alpha(\mathrm{TruthMap}(w_0, w_1)) = \mathrm{TruthMap}(w_0, w_1)$

(ii) $\alpha(\mathcal{C}_0) = \mathcal{C}_0$

(iii) $\alpha(\mathcal{X}_1^+ \cup \mathcal{X}_1^- \cup \cdots \cup \mathcal{X}_n^+ \cup \mathcal{X}_n^-) \subseteq \mathcal{C}_0 \cup \mathcal{C}_1$

(iv) $\alpha(\mathcal{X}_1^+ \cup \mathcal{X}_1^- \cup \cdots \cup \mathcal{X}_n^+ \cup \mathcal{X}_n^-) \supseteq \mathcal{C}_1$

Properties (i) and (ii) hold because α is the identity on $\mathsf{vars}(\mathcal{C}_0) = \{w_0, w_1\}$. Next, let us consider the set $\mathcal{X}_i^+$, for some $i \in [1, n]$. By definition, view application α maps $(x_i, \bar{x}_i)$ either to (w_0, w_1) or to (w_1, w_0). The atoms in $\mathcal{X}_i^+$, which are of the form $C_j(x_i, \bar{x}_i)$, are thus mapped to $C_j(w_0, w_1)$ or $C_j(w_1, w_0)$, which are contained in $\mathcal{C}_0$ and $\mathcal{C}_1$, respectively. Analogously, we get $\mathcal{X}_i^- \subseteq \mathcal{C}_0 \cup \mathcal{C}_1$, for every $i \in [1, n]$. Thus, Property (iii) holds.

Finally, we show that Property (iv) holds. To this end, let $C_j(w_1, w_0)$ be an arbitrary atom from $\mathcal{C}_1$. Since the truth assignment β satisfies f, it also satisfies the clause c_j. Hence, there is a literal L in c such that β assigns the truth value 1 to L. Let us assume $L = X_i$, for some $i \in [1, n]$ is a positive literal; the proof for a negated literal is analogous. By definition of α, we then have $\alpha(x_i) = w_1$ and $\alpha(\bar{x}_i) = w_0$. Since $L = X_i$ is a literal in clause c_j, we also have that $C_j(x_i, \bar{x}_i) \in \mathcal{X}_i^+$. We can conclude that $C_j(w_1, w_0) = \alpha(C_j(x_i, \bar{x}_i))$ is indeed contained in $\alpha(\mathcal{X}_i^+)$.

Correctness, Part II. Now, we show that the formula f is satisfiable if there are mappings α and ψ such that $\langle \mathsf{body}(Q), V, \alpha, \psi \rangle$ is a cover description for Q.

Thanks to Definition 5.2.3, we know that

$$\mathcal{A} = \mathsf{body}(Q) \subseteq \alpha(\mathsf{body}(V)) \quad \text{and} \quad \psi(\alpha(\mathsf{body}(V))) \subseteq \mathsf{body}(Q)$$

hold, due to Condition (1) and Condition (3), respectively. Since both bodies $\mathsf{body}(Q)$ and $\mathsf{body}(V)$ contain exactly one TruthMap-atom, namely $B = \mathrm{TruthMap}(w_0, w_1)$, we

have $\alpha(B) = B$. Therefore, the mappings α and ψ are the identity on $\{w_0, w_1\}$. Hence, the view application α induces unambiguously a truth assignment β with

$$\beta(X_i) = \begin{cases} 1, & \text{if } \alpha(x_i) = w_1 \\ 0, & \text{otherwise} \end{cases} \qquad \text{for every } i \in [1, n].$$

It suffices to show that β satisfies the formula f. To this end, let c_j be an arbitrary clause of f. From $\mathsf{body}(Q) \subseteq \alpha(\mathsf{body}(V))$, we know that the atom $C_j(w_1, w_0)$ is in $\alpha(\mathsf{body}(V))$ because it is in $\mathcal{C}_1$ and, therefore, in $\mathsf{body}(Q)$. Since $\mathsf{body}(V)$ does not contain atoms from $\mathcal{C}_1$, it has to contain a different C_j-atom which is mapped to $C_j(w_1, w_0)$ by α. This atom cannot be $C_j(w_0, w_1)$ from $\mathcal{C}_0$ because α is the identity on $\{w_0, w_1\}$. Hence, there has to be an atom $C_j(x_i, \bar{x}_i) \in \mathcal{X}_i^+$ or $C_j(\bar{x}_i, x_i) \in \mathcal{X}_i^-$, for some $i \in [1, n]$, in $\mathsf{body}(V)$ that is mapped to $C_j(w_1, w_0)$ by α. Suppose that an atom $C_j(x_i, \bar{x}_i) \in \mathcal{X}_i^+$ is in $\mathsf{body}(V)$ and is mapped to $C_j(w_1, w_0)$. By construction $C_j(x_i, \bar{x}_i)$ is only in $\mathcal{X}_i^+$, and thus in $\mathsf{body}(V)$, if the literal $L = X_i$ occurs in clause c_j. Furthermore, from $\alpha(C_j(x_i, \bar{x}_i)) = C_j(w_1, w_0)$, we can infer that $\alpha(x_i) = w_1$ holds. But then, the truth assignment β assigns the truth value 1 to the literal X_i. Finally, since X_i is a literal of the clause c_j, we can conclude that β satisfies c_j. The case that an atom $C_j(\bar{x}_i, x_i) \in \mathcal{X}_i^-$ is mapped to $C_j(w_1, w_0)$ is analogous: In this case, we have that $\alpha(C_j(\bar{x}_i, x_i)) = C_j(w_1, w_0)$, and $\neg X_i$ occurs in c_j. From the former we can derive that $\alpha(x_i) = w_0 \neq w_0$, and, hence $\beta(X_i) = 0$.

Overall the truth assignment β satisfies every clause of the formula f, and thus, f is satisfiable. □

The proof of Theorem 5.3.9 does *not directly* imply NP-hardness of the acyclic rewriting problems, as illustrated by the next example. In a nutshell, the reason is that, to obtain a cover partition, one is free to choose the underlying partition of $\mathsf{body}(Q)$.

Example 5.3.10. Consider the propositional formula $f = X_1 \wedge \neg X_1$ with the two clauses $c_1 = X_1$ and $c_2 = \neg X_1$, which is obviously *not* satisfiable. The construction from the proof of Theorem 5.3.9 yields the query Q defined by

$$H() \leftarrow \mathrm{TruthMap}(w_0, w_1), \underbrace{C_1(w_0, w_1), C_2(w_0, w_1)}_{\mathcal{C}_0}, \underbrace{C_1(w_1, w_0), C_2(w_1, w_0)}_{\mathcal{C}_1}$$

and the view V with

$$H(w_0, w_1, x_1, \bar{x}_1) \leftarrow \mathrm{TruthMap}(w_0, w_1), \underbrace{C_1(w_0, w_1), C_2(w_0, w_1)}_{\mathcal{C}_0}, \underbrace{C_1(x_i, \bar{x}_i)}_{\mathcal{X}_1^+}, \underbrace{C_2(\bar{x}_i, x_i)}_{\mathcal{X}_2^-}.$$

Note that, in this example, $\mathcal{X}_1^- = \mathcal{X}_2^+ = \emptyset$. As expected there is *no* cover partition $\langle \mathcal{A}, V, \alpha, \psi \rangle$ with $\mathcal{A} = \mathsf{body}(Q)$ for Q, because, no matter how α is chosen, $C_1(w_1, w_0)$ or $C_2(w_1, w_0)$ is *not* in $\alpha(\mathsf{body}(V))$. Thus, Condition (1) of Definition 5.2.3 cannot be satisfied.

There is, however, a cover partition over V for Q. It consists of the two cover descriptions $\langle \mathsf{body}(Q) \setminus \{C_2(w_1, w_0)\}, V, \alpha, \mathsf{id} \rangle$ and $\langle \{C_2(w_1, w_0)\}, V, \alpha', \mathsf{id} \rangle$ where α and α' are the

identity on $\{w_0, w_1\}$, and $\alpha(x_i) = w_1$, $\alpha(\bar{x}_i) = w_0$, $\alpha'(x_i) = w_0$, and $\alpha'(\bar{x}_i) = w_1$. Note that the bridge variables w_0 and w_1 occur in both heads, $\alpha(\mathsf{head}(V))$ and $\alpha'(\mathsf{head}(V))$.

In terms of propositional formulas, it is possible to choose a different truth assignment for each clause. ◁

Despite Example 5.3.10 the cover description problem can be reduced to the acyclic rewriting problem, using a straightforward reduction. The idea is to "bind" the atoms in $\mathsf{body}(Q)$ together by a quantified variable which makes a "split", as endorsed in the example, impossible. This leads us to the proof that $\textsc{Rewr}^3(\mathsf{HCQ}, \mathsf{HCQ}, \mathsf{CQ})$, and hence all problems stated in Theorem 5.3.8, are NP-complete.

Proof of Theorem 5.3.8. The upper bound follows from Theorem 5.1.3 and Theorem 5.3.5. We prove the lower bound by reduction from $\textsc{CovDesc}^2(\mathsf{QHCQ}, \mathsf{QHCQ})$, which is NP-hard due to Theorem 5.3.9, to $\textsc{Rewr}^3(\mathsf{HCQ}, \mathsf{HCQ}, \mathsf{CQ})$. Since we have $\mathsf{HCQ} \subseteq \mathsf{ACQ}$, cf. Proposition 2.4.3, and thanks to Theorem 5.3.5, this immediately implies NP-hardness of the other problems stated in Theorem 5.3.8.

Construction. For convenience, we introduce the following notation. For an atom $A = R(x_1, \ldots, x_r)$ and a variable u we denote by A^u the atom $R^u(u, x_1, \ldots, x_r)$ resulting from extending A by the variable u. We lift this notation to sets $\mathcal{A}$ of atoms in the natural way, that is

$$\mathcal{A}^u = \{R^u(u, x_1, \ldots, x_r) \mid R(x_1, \ldots, x_r) \in \mathcal{A}\}.$$

Let now Q be a q-hierarchical conjunctive query and V be a q-hierarchical view such that $(V, Q, \mathsf{body}(Q))$ is an instance for $\textsc{CovDesc}^2(\mathsf{QHCQ}, \mathsf{QHCQ})$. Recall that, thanks to Theorem 5.3.9, it suffices to consider instances of this form for our purpose. Moreover, let u be a fresh variable that does not occur in Q or V. We define Q^+ as the conjunctive query with head $\mathsf{head}(Q)$ and body $\mathsf{body}(Q)^u$. Analogously, we define V^+ as the view with the same head variables as V and body $\mathsf{body}(V)^u$. Observe that u is a quantified variable, since it does *not* occur in the heads of V^+ and Q^+.

We argue next that V^+ and Q^+ are hierarchical, and thus $\{V^+\}$ and Q^+ form an instance for $\textsc{Rewr}^3(\mathsf{HCQ}, \mathsf{HCQ}, \mathsf{CQ})$. Indeed, for pairs of variables $y, z \in \mathsf{vars}(V^+) \setminus \{u\}$ with $y \neq z$, at least one the conditions of Definition 2.4.2 is satisfied because V is q-hierarchical, and hence hierarchical in particular. More precisely, we have that $\mathsf{atoms}_{V^+}(y) = \mathsf{atoms}_V(y)^u$, for all variables $y \neq u$, and hence, the construction preserves the relationships – with respect to containment and disjointness – of these sets. It remains to consider the relationships of atom sets for the new variable u and a variable $y \in \mathsf{vars}(V)$. And indeed, since u occurs in every atom from $\mathsf{body}(V^+)$, we have $\mathsf{atoms}_{V^+}(y) \subseteq \mathsf{atoms}_{V^+}(u)$. We conclude that V^+ is hierarchical.[13] The proof for Q^+ being hierarchical is completely

[13] We note that V^+ is *not* necessarily q-hierarchical even though V is. This is, in particular, true for the view V constructed in the proof of Theorem 5.3.9. Recall that, for instance, w_0 is a head variable of V. Since it does not occur together with x_1 in an atom but u occurs in every atom, we have $\mathsf{atoms}_{V^+}(w_0) \subsetneq \mathsf{atoms}_{V^+}(u)$. This means that V^+ is *not* q-hierarchical because u is a quantified variable. Moreover, it is also *not* in CCQ because u connects atoms with variables w_0, w_1 and atoms with variables $x_i, \bar{x}_i$. Together with the head atom they then form a cycle.

analogous.

Obtaining Q^+ and V^+ from Q and V is straightforward and can easily be done in polynomial time.

Correctness. We prove that there are mappings α and ψ such that $\langle \mathsf{body}(Q), V, \alpha, \psi \rangle$ is a cover description for Q if and only if there is a cover partition over $\{V^+\}$ for Q^+. Since Theorem 5.3.9 states NP-hardness even for minimal queries and Q^+ is minimal if Q is, this indeed suffices to prove Theorem 5.3.8, thanks to Theorem 5.2.8.

The direction from left to right is trivial. If $\langle \mathsf{body}(Q), V, \alpha, \psi \rangle$ is a cover description for Q, for some mappings α and ψ, then $\langle \mathsf{body}(Q^+), V^+, \alpha^u, \psi^u \rangle$ is a cover description which constitutes a cover partition for Q^+. Here α^u and ψ^u are the mappings which are the identity on u and agree with α and ψ on every other variable, respectively. Note that, in particular, the bridge variables remain the same.

For the converse, let $\mathcal{P}$ be a cover partition for Q^+ over $\{V^+\}$. If $\mathcal{P}$ consists of only one cover description, then this cover description has the form $\langle \mathsf{body}(Q^+), V^+, \alpha', \psi' \rangle$ for some mappings α' and ψ'. But then $\langle \mathsf{body}(Q), V, \alpha, \psi \rangle$ is a cover description for Q; the mappings α and ψ are the same as α' and ψ', except that they are *not* defined for u. This is again trivial, because the first position (and thus u) is just deleted from every atom.

It remains to show that this is the only possible case. That is, we show that $\mathcal{P}$ cannot consist of more than one cover description. For the sake of contradiction, suppose that $\mathcal{P}$ consists of at least two cover descriptions. Let d be a cover description from $\mathcal{P}$. Since V^+ is the only view and d is *not* the only cover description in $\mathcal{P}$, d has the shape $d = \langle \mathcal{A}^u, V, \alpha, \psi \rangle$ with $\mathcal{A}^u \subsetneq \mathsf{body}(Q^+)$. We observe that the variable u is a bridge variable of $\mathcal{A}^u$ since it occurs in every atom of Q^+, and, thus, in- and outside $\mathcal{A}^u$.

Since u occurs precisely at the first position of every atom in Q^+ and V^+, it follows from Condition (1) of Definition 5.2.3 that $\alpha(u) = u$. Furthermore, α maps *no* other variable to u, because u is a quantified variable and as such cannot be unified with any other variable by α. But then $\mathsf{bvars}_Q(\mathcal{A}) \not\subseteq \alpha(\mathsf{vars}(\mathsf{head}(V^+)))$, since u does not occur in the head of V^+. This contradicts Condition (2) of Definition 5.2.3. Therefore, d cannot be a cover description.

We can conclude that $\mathcal{P}$ consists of a single cover description. □

5.3.3 An Implication for Multi-Query Evaluation

We next discuss the cover description problem from the perspective of another setting – the *multi-query evaluation setting*. Theorem 5.3.9 will let us conclude NP-hardness of a very closely related problem in this setting. Aside from that this perspective will also allow us to further highlight the underlying reason for the (supposed) intractability of the cover description problem.

In the multi-query evaluation setting, multiple given queries $Q_1, \ldots, Q_m$ are to be evaluated. In this setting it may be possible to evaluate a query Q_i more efficiently, if the query results of some of the other given queries have already been computed. The goal is

thus to obtain an order in which the given queries can be evaluated most efficiently.[14]

In this section we consider a very simple variant, where only two queries are given, one of which is a Boolean query. The question is then whether the Boolean query can be evaluated using only some basic operators, if the query result for the other query is available. In terms of the relational algebra we permit the select operator as well as a *full project* operator, i.e. a projection to the empty set of attributes, as basic operators. For consistency and convenience, we state the formal definition in terms of conjunctive queries. For a class $\mathbb{Q}$ of conjunctive queries, the *select-full-project equivalence* problem for $\mathbb{Q}$, denoted SelProjEquiv($\mathbb{Q}$) is defined as follows.

— SelProjEquiv($\mathbb{Q}$) —

Given: Query $Q' \in \mathbb{Q}$, Boolean query $Q \in \mathbb{Q}$

Question: Is there a Boolean conjunctive query Q'' which is equivalent to Q and whose body can be obtained from $\mathsf{body}(Q')$ by unifying head variables of Q' within $\mathsf{body}(Q')$.

Example 5.3.11. Consider the conjunctive query Q' defined by

$$H'(y, z, u, v) \leftarrow R(x), S(y, x, v), S(y, x, w), T(y, z), T(u, x)$$

and the Boolean query Q defined by

$$H() \leftarrow R(x), S(y, x, y), T(y, z), T(z, x).$$

Unifying the head variables y and v as well as z and u in $\mathsf{body}(Q')$ yields the Boolean query

$$H() \leftarrow R(x), S(y, x, y), S(y, x, w), T(y, z), T(z, x).$$

It is equivalent to Q, as its body is the same as $\mathsf{body}(Q)$ except for a redundant S-atom. Thus, we have $(Q', Q) \in$ SelProjEquiv(CQ). ◁

Interpreting the query Q' in Example 5.3.11 as a view, the unification of head variables can be realized by a view application α which does not unnecessarily rename variables. Whether the desired Boolean query Q'' exists then boils down to whether there is a $\{Q'\}$-rewriting for Q whose body consists of a single view atom, i.e. $\alpha(\mathsf{head}(Q'))$. But this is just the same as asking whether there is a cover description with atom set $\mathsf{body}(Q)$ for Q.

Example 5.3.12 (Continuation of Example 5.3.11)**.** Recall that the body of the Boolean query Q'' in Example 5.3.11 is obtained from $\mathsf{body}(Q')$ by unifying the head variables y and v as well as z and u. Let α be the view application of Q' which maps v to y, u to z, and is the identity on every other variable. Furthermore, let ψ be the variable mapping

[14] The acyclic rewriting problem can be understood as a subproblem in this setting. If a query Q_i has a (structurally simple) $\mathcal{V}$-rewriting, where $\mathcal{V}$ consists of some of the other queries in the sequence $Q_1, \ldots, Q_m$, then it might be favourable to evaluate the queries in $\mathcal{V}$ before Q_i.

that maps w to y and is the identity on every other variable. Then $\langle \mathsf{body}(Q), Q', \alpha, \psi \rangle$ is a cover description for Q. In other words, the Boolean query with body

$$\{\alpha(\mathsf{head}(Q'))\} = \{H'(y, z, z, y)\}$$

is a $\{Q'\}$-rewriting of Q. ◁

As suggested by Examples 5.3.11 and 5.3.12, the select-full-project equivalence problem and the cover description problem are closely related. This lets us conclude the following.

Corollary 5.3.13. SelProjEquiv(QHCQ) *is* NP*-hard, even if all database relations have arity at most* 2*.*

Proof. The proof is by reduction from CovDesc^k(QHCQ, QHCQ), which is NP-hard due to Theorem 5.3.9, even for instances with minimal Boolean queries Q and $\mathcal{A} = \mathsf{body}(Q)$. To be exact, the reduction is thus from the restriction of CovDesc^k(QHCQ, QHCQ) to such instances.

Let $V \in \mathsf{QHCQ}$ be a view and $Q \in \mathsf{QHCQ}$ be a Boolean query such that $(V, Q, \mathsf{body}(Q))$ stipulates an instance for CovDesc^k(QHCQ, QHCQ). The instance for the select-full-project equivalence problem is then just (V, Q). That is, V takes the role of the second query Q'.

If there are mappings α an ψ such that $d = \langle \mathsf{body}(Q), V, \alpha, \psi \rangle$ is a cover description for Q, then it constitutes, in particular, a consistent cover partition for Q. Thus, the Boolean query with body $\{\alpha(\mathsf{head}(V))\}$ is a rewriting of Q, thanks to Lemma 5.2.10. We conclude that Q is equivalent to the Boolean query Q'' whose body is obtained from $\mathsf{body}(V)$ by unifying exactly the (head) variables in $\mathsf{body}(V)$ which are unified by the view application α. Indeed, Q'' coincides with the expansion $\mathsf{exp}(\{d\})$, up to renaming variables.

Conversely, if Q is equivalent to a Boolean query Q'' whose body can be obtained from $\mathsf{body}(V)$ by unifying head variables of V in $\mathsf{body}(V)$, then there is a view application α with $\alpha(\mathsf{body}(V)) = \mathsf{body}(Q'')$. Since Q is a Boolean query, the Boolean query with body $\{\alpha(\mathsf{head}(V))\}$ is a $\{V\}$-rewriting of Q. Thanks to Lemma 5.2.11 there is a cover partition over $\{V\}$ for Q that consists of a single cover description. This cover description has necessarily the form $\langle \mathsf{body}(Q), V, \alpha', \psi' \rangle$. □

Of course, Corollary 5.3.13 implies NP-hardness of SelProjEquiv($\mathbb{Q}$) for any class $\mathbb{Q}$ with $\mathsf{QHCQ} \subseteq \mathbb{Q} \subseteq \mathsf{CQ}$. This includes, in particular, SelProjEquiv(ACQ) and SelProjEquiv(HCQ).

The relationship of the cover description problem and the select-full-project equivalence problem highlights that one of the difficulties of the former is to decide which head variables to unify. Given that – as discussed in Remark 5.2.14 – the containment problem and deciding whether homomorphisms exists is in P for acyclic conjunctive queries (and hence hierarchical and q-hierarchical queries), it *might* be the only one. This is further supported by the fact that restricting the select-full-project equivalence problem to instances (Q', Q) where both queries Q', Q are Boolean, yields the equivalence problem for Boolean conjunctive queries, which is in P for acyclic queries [cf., e.g., CR00, Theorem 2].

5.4 A Tractable Case: Mind your Head!

Roughly speaking, Theorem 5.3.5 states that the *existence* of a structurally simple rewriting is guaranteed, if the given query has a simple structure itself.[15] However, we have also proved that, even if queries *and* views have a simple structure, the acyclic rewriting problem is *not* necessarily tractable.

If, in addition to a simple structure, the heads of the views have bounded arity, the acyclic rewriting problem is in P, thanks to Corollary 5.3.6. Together with our observations in Sections 5.3.2 and 5.3.3 this suggests that the heads of the views play a crucial role – when it comes to tractability. This motivates the study of view classes that impose restriction on the heads of views.

In this section, we primarily study the acyclic rewriting problem for free-connex acyclic views. And indeed, it turns out the restriction imposed on the heads of free-connex acyclic views suffices for tractability, at least if the arity of all databases relations is bounded. Our proof of this statement will reveal that it actually holds for a slightly larger class of views, and that the bound on the arity of database relations is not always required.

We prove the tractability result for free-connex acyclic views by reduction from $\textsc{Rewr}^k(\mathsf{CCQ}, \mathsf{ACQ}, \mathsf{ACQ})$ to $\textsc{Rewr}^k(\mathsf{ACQ}^k, \mathsf{ACQ}, \mathsf{ACQ})$. The following result is the core of this reduction.

Proposition 5.4.1. *There is a polynomial time algorithm that computes, given a set $\mathcal{V}$ of free-connex acyclic views over a schema $\mathcal{S}$, a set $\mathcal{W}$ of acyclic views over $\mathcal{S}$ such that*

(A) *the arity of the views in $\mathcal{W}$ is bounded by the arity of $\mathcal{S}$, and*

(B) *every conjunctive query Q is $\mathcal{V}$-rewritable if and only if it is $\mathcal{W}$-rewritable.*

Moreover, given a cover partition over $\mathcal{W}$ for Q, a cover partition over $\mathcal{V}$ for Q can be computed in polynomial time.

Similarly to the proof of Theorem 5.3.5, the idea is to refine a cover partition. However, instead of refining the partition along a join tree for the query, here the refinement is guided by join trees for $\mathsf{body}(V) \cup \{\mathsf{head}(V)\}$, for each view V. A bit more precisely, each view V is "split" into views of smaller arity, and each of these new views then "covers" a subset of the atoms covered by the original view.

Proof of Proposition 5.4.1. Let $\mathcal{V} \subseteq \mathsf{CCQ}$ be a set of free-connex acyclic views over a schema $\mathcal{S}$, and k be the arity of $\mathcal{S}$. We first describe how the procedure replaces a single view $V \in \mathcal{V}$ by views of smaller arity. It consists of two phases – the partitioning and the splitting phase.

[15] Concretely, Theorem 5.3.5 states this for acyclic and free-connex acyclic queries, but we will see that this is also true for hierarchical and q-hierarchical queries.

Partitioning. Let T_V be a join tree for V including its head atom $\mathsf{head}(\mathrm{V})$. Such a join tree exists because V is free-connex acyclic. Since a join tree is undirected, we can assume that $\mathsf{root}(T_V)$ is the unique node labelled with $\mathsf{head}(V)$. Let $v_1, \ldots, v_n$ be the children of the root node, and $B_1, \ldots, B_n$ their respective labels. Furthermore, let, for each $i \in [1, n]$, $\mathcal{B}_i$ be the set of atoms in the subtree of T_V with root v_i.

Since T_V is a join tree, each variable that occurs in a set $\mathcal{B}_i$ and in $\mathsf{head}(V)$ also occurs in B_i. Furthermore, variables that occur in two sets $\mathcal{B}_i$, $\mathcal{B}_j$, $i \neq j$, necessarily also occur in $\mathsf{head}(V)$. Thus, the following two properties hold, for all $i, j \in [1, n]$ with $j < i$.

(a) $|\mathsf{vars}(\mathcal{B}_i) \cap \mathsf{vars}(\mathsf{head}(V))| \leq k$ **(b)** $\mathsf{vars}(\mathcal{B}_i) \cap \mathsf{vars}(\mathcal{B}_j) \subseteq \mathsf{vars}(\mathsf{head}(V))$

Splitting. For each $i \in [1, n]$ we define the view V_i as the projection of V to the variables that occur in $\mathcal{B}_i$ and in $\mathsf{head}(V)$. That is, the body of V_i is just the body of V and $\mathsf{vars}(\mathsf{head}(V_i)) = \mathsf{vars}(\mathsf{head}(V)) \cap \mathsf{vars}(\mathcal{B}_i)$.

The desired set $\mathcal{W}$ of views is obtained by replacing each view $V \in \mathcal{V}$ by views $V_1, \ldots, V_n$ as constructed above. It is not hard to see that $\mathcal{W}$ can be computed in polynomial time.

Correctness. The constructed views V_i are acyclic since they have the same body as the view V they were derived from.[16] The latter is acyclic because it is even free-connex acyclic. Furthermore, the constructed views V_i have arity at most k, thanks to Property (a). Thus, Statement (A) holds.

To establish Statement (B), we prove that, for every conjunctive query Q, there is a cover partition over $\mathcal{V}$ for Q if and only if there is a cover partition over $\mathcal{W}$ for Q. Then Statement (B) follows, thanks to Theorem 5.2.8.

For the direction from right to left, let $\mathcal{P}_\mathcal{W}$ be a cover partition witnessing that Q is $\mathcal{W}$-rewritable. Consider a cover description $\langle \mathcal{A}, V_i, \alpha, \psi \rangle$ where V_i is a view constructed as above, originating from a view $V \in \mathcal{V}$. Since the only difference between V and V_i is that V has more head variables, replacing V_i with V yields a cover description $\langle \mathcal{A}, V, \alpha, \psi \rangle$. Analogous replacements in all cover descriptions from $\mathcal{P}_\mathcal{W}$ yield a cover partition over $\mathcal{V}$ for Q. Clearly, this transformation can be done in polynomial time.

For the direction from left to right, let $\mathcal{P}_\mathcal{V}$ be a cover partition over $\mathcal{V}$ for Q. Thanks to Theorem 5.2.8, we can assume that $\mathcal{P}_\mathcal{V}$ is consistent. Again, we replace the cover descriptions in $\mathcal{P}_\mathcal{V}$ to obtain a cover partition over $\mathcal{W}$. To this end, let $d = \langle \mathcal{A}, V, \alpha, \psi \rangle$ be a cover description from $\mathcal{P}_\mathcal{V}$. Furthermore, let $\mathcal{B}_1, \ldots, \mathcal{B}_n$ and $V_1, \ldots, V_n$ be the partition and the views in $\mathcal{W}$ for V. For each $i \in [1, n]$, let $\mathcal{A}_i$ be the set of all atoms in $\mathcal{A}$ which are in $\alpha(\mathcal{B}_i)$ but in no $\alpha(\mathcal{B}_j)$, for any $j < i$. Since we have $\mathcal{A} \subseteq \alpha(\mathsf{body}(V))$, thanks to Condition (1) of Definition 5.2.3, this yields a partition of $\mathcal{A}$. We claim that, for each $i \in [1, n]$, $d_i = \langle \mathcal{A}_i, V_i, \alpha, \psi \rangle$ is a cover description. Indeed, since V_i and V have the same body, Conditions (1) and (3) of Definition 5.2.3 hold. Condition (4) of Definition 5.2.3 holds since $\mathsf{vars}(\mathcal{A}_i) \subseteq \mathsf{vars}(\mathcal{A})$.

[16] We observe that, since all head variables of V_i occur in B_i, each V_i is even free-connex acyclic. However, we do *not* require this property and – as we will discuss – the construction can be applied to a slightly more general class of views as is, but will *not* guarantee free-connex acyclicity.

In the remainder, we prove that Condition (2) of Definition 5.2.3 holds. To this end, let $x \in \mathsf{bvars}_Q(\mathcal{A}_i)$. Then x is either in $\mathsf{bvars}_Q(A)$ or it is a "new" bridge variable that also occurs in some $\mathcal{A}_j \subseteq \mathcal{A}$, for some $j \neq i$. In the former case, we have $x \in \alpha(\mathsf{head}(V))$ since Condition (2) holds for the original cover description d. Thus, there are variables $y \in \mathsf{vars}(\mathsf{head}(V))$ and y' in $\mathsf{vars}(\mathcal{B}_i)$ such that $\alpha(y) = x = \alpha(y')$. Since, thanks to quantified variable disjointness, α maps quantified and head variables disjointly, it follows that y' must be from $\mathsf{head}(V)$ as well. But then y' occurs in $\mathsf{head}(V_i)$ since it occurs in $\mathsf{head}(V)$ and $\mathcal{B}_i$. Therefore, $x = \alpha(y') \in \alpha(\mathsf{vars}(\mathsf{head}(V_i)))$.

In the other case, let $j \neq i$ be such that x occurs in $\mathcal{A}_j$ and, hence, in $\alpha(\mathcal{B}_j)$. Let y, z be variables from $\mathcal{B}_i$ and $\mathcal{B}_j$, respectively, such that $\alpha(y) = x = \alpha(z)$. If $y = z$, then y is a head variable of V, since the only variables $\mathcal{B}_i$ and $\mathcal{B}_j$ have in common are head variable of V, due to Property (b). If $y \neq z$, then both y and z occur in $\mathsf{head}(V)$ thanks to quantified variable disjointness. In both cases, we can conclude that $x = \alpha(y)$ occurs in $\alpha(\mathsf{head}(V_i))$, because x occurs in $\mathcal{B}_i$ and $\mathsf{head}(V)$. Thus, d_i satisfies Condition (2) of Definition 5.2.3.

Since the sets $\mathcal{A}_i$ form a partition of $\mathcal{A}$, replacing d in $\mathcal{P}_\mathcal{V}$ with $d_1, \ldots, d_n$ yields a cover partition over $\mathcal{V} \cup \mathcal{W}$. Repeating this process for all cover description from the original cover partition $\mathcal{P}_\mathcal{V}$ yields a cover partition over $\mathcal{W}$ for Q. □

Since Proposition 5.4.1 effectively provides a reduction from $\textsc{Rewr}^k(\mathsf{CCQ}, \mathsf{ACQ}, \mathsf{ACQ})$ to $\textsc{Rewr}^k(\mathsf{ACQ}^k, \mathsf{ACQ}, \mathsf{ACQ})$, and the latter is in P due to Corollary 5.3.6, we get the main result of this section.

Theorem 5.4.2. *For every $k \geq 0$, the following two statements hold.*

(1) $\textsc{Rewr}^k(\mathsf{CCQ}, \mathsf{ACQ}, \mathsf{ACQ})$ *is in* P*, and an acyclic rewriting can be computed in polynomial time, if it exists.*

(2) $\textsc{Rewr}^k(\mathsf{CCQ}, \mathsf{CCQ}, \mathsf{CCQ})$ *is in* P*, and a free-connex acyclic rewriting can be computed in polynomial time, if it exists.*

Proof. Let $\mathcal{V} \subseteq \mathsf{CCQ}$ be a set of free-connex acyclic views over a schema $\mathcal{S}$ with arity at most k, and $Q \in \mathsf{ACQ}$ be an acyclic conjunctive query over $\mathcal{S}$. Given $\mathcal{V}$, an "equivalent" set $\mathcal{W}$ of acyclic views of arity at most k can be computed in polynomial time, thanks to Proposition 5.4.1. Thus, the claim that $\textsc{Rewr}^k(\mathsf{CCQ}, \mathsf{ACQ}, \mathsf{ACQ})$ is in P follows immediately from Corollary 5.3.6. $\textsc{Rewr}^k(\mathsf{CCQ}, \mathsf{ACQ}, \mathsf{ACQ})$ is then also in P since $\mathsf{CCQ} \subseteq \mathsf{ACQ}$, and thanks to Theorem 5.3.5.

Moreover, a cover partition over $\mathcal{W}$ for Q can be computed in polynomial time, thanks to Lemma 5.2.13. Again thanks to Proposition 5.4.1 this cover partition can be transformed into a cover partition over $\mathcal{V}$ for Q in polynomial time. This cover partition can be turned into an acyclic rewriting (or even a free-connex acyclic rewriting, if Q is free-connex acyclic), thanks to Theorem 5.3.5. □

We leave the complexity of $\textsc{Rewr}(\mathsf{CCQ}, \mathsf{ACQ}, \mathsf{ACQ})$ and $\textsc{Rewr}(\mathsf{CCQ}, \mathsf{CCQ}, \mathsf{CCQ})$ as an open problem.

A closer inspection of the proof for Proposition 5.4.1 reveals that only the partitioning phase actually requires the views to be free-connex acyclic. For the splitting phase, and, more importantly, the correctness it suffices that the body of each view V can be partitioned into sets $\mathcal{B}_1, \ldots, \mathcal{B}_m$ which obey Properties (a) and (b) from the proof of Proposition 5.4.1. This leads to the following definition. We emphasize that, while we state and study it for conjunctive queries in general, we will actually only use it for views.

Definition 5.4.3 (Weak Head Arity)**.** The *weak head arity* of a conjunctive query Q is the smallest integer $k \geq 0$ for which $\mathsf{body}(V)$ can be partitioned into sets $\mathcal{B}_1, \ldots, \mathcal{B}_n$ that have, for all $i, j \in [1, n]$ with $j < i$, the following two properties.

(a) $|\mathsf{vars}(\mathcal{B}_i) \cap \mathsf{vars}(\mathsf{head}(Q))| \leq k$ **(b)** $\mathsf{vars}(\mathcal{B}_i) \cap \mathsf{vars}(\mathcal{B}_j) \subseteq \mathsf{head}(Q)$

Observe that, thanks to the partitioning phase in the proof for Proposition 5.4.1, we know that free-connex acyclic conjunctive queries over a fixed database schema have bounded weak head arity. The same is obviously true for views with bounded head arity, i.e. queries from ACQ^k, for some fixed k. The following example illustrates, in particular, that bounded weak head arity extends both classes. More precisely, it shows that there are indeed views over a fixed schema that have bounded weak head arity but are neither free-connex acyclic nor have bounded head arity.

Example 5.4.4. Let us consider the family $\big(V_n\big)_{n \in \mathbb{N}_0}$ of views with

- $\mathsf{head}(V_n) = V_n(x, y_1, \ldots, y_n, z_1, \ldots, z_n)$, and
- $\mathsf{body}(V_n) = \big\{R(x, u_i, y_i), S(x, u_i, z_i), T(y_i) \mid i \in [1, n]\big\}$.

For $n \geq 1$ the view V_n is acyclic but *not* free-connex acyclic. Furthermore, its arity is $2n + 1$, and thus, the arity of the family is unbounded. It has, however, weak head arity 3. This is witnessed by the sets $\mathcal{B}_i = \{R(x, u_i, y_i), S(x, u_i, z_i), T(y_i)\}$ for $i \in [1, n]$ which form a partition of $\mathsf{body}(V_n)$ and have Properties (a) and (b) of Definition 5.4.3.

Replacing the atoms $T(y_i)$ with atoms $T_n(y_i, w_1, \ldots, w_n)$ preserves the weak head arity and acyclicity. Hence, the notion of bounded weak head arity also captures views over schemas whose arity cannot be unbounded. If, in addition to replacing the T-atoms, the R-atoms are removed, the resulting views are free-connex acyclic. Still, the weak head arity is 3. Thus, some free-connex acyclic queries over unbounded database schemas (and unbounded head arity) are captured. However, the family of trivial views $V'_n(x_1, \ldots, x_n) \leftarrow R'(x_1, \ldots, x_n)$ witnesses that *not* all free-connex acyclic conjunctive queries are captured. ◁

In the remainder of this section, we prove that Theorem 5.4.2 can be generalized to views of bounded weak head arity. For this purpose, we generalize the partitioning phase in the proof of Proposition 5.4.1. Recall that, in the partitioning phase, the algorithm uses a join tree (comprising the head atom) to compute a partition that has Properties (a) and (b) of Definition 5.4.3. Here we cannot assume the existence of such join trees. Instead, we will employ *cover graphs*, which are defined as follows.

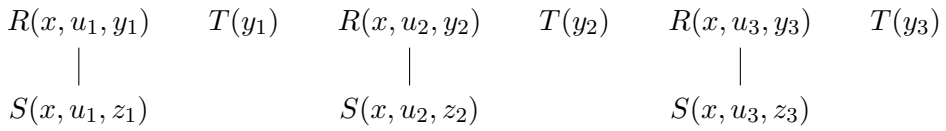

Figure 5.1: The cover graph $G[V_3]$ of the query V_n for $n = 3$ defined in Example 5.4.4.

Definition 5.4.5 (Cover Graph)**.** The *cover graph* $G[Q]$ of a conjunctive query Q is the undirected graph with node set $\mathsf{body}(Q)$ and edge relation

$$\big\{\{A, A'\} \mid A, A' \in \mathsf{body}(Q), A \neq A', \mathsf{vars}(A) \cap \mathsf{vars}(A') \not\subseteq \mathsf{vars}(\mathsf{head}(Q))\big\}.$$

That is, there is an edge between two atoms A and A' if they share a quantified variable.

Example 5.4.6. The cover graph of the (unmodified) view V_3 from Example 5.4.4 is depicted in Figure 5.1. ◁

The following result provides a proper generalization of the partitioning phase in the proof of Proposition 5.4.1.

Lemma 5.4.7. *Let Q be a conjunctive query and $\mathcal{C}_1, \ldots, \mathcal{C}_n$ be the connected components of its cover graph. The weak head arity of Q is the maximal number ℓ of head variables from Q that occur in a set $\mathcal{C}_i$.*

Moreover, the connected components $\mathcal{C}_1, \ldots, \mathcal{C}_n$ witness that Q has weak head arity ℓ, and can be computed in polynomial time.

Proof. Let Q be a conjunctive query, and let $\mathcal{B}_1, \ldots, \mathcal{B}_m$ be sets forming a partition of $\mathsf{body}(Q)$ and witnessing that Q has weak head arity k. Furthermore, let $\mathcal{C}_1, \ldots, \mathcal{C}_n$ be the connected components of the cover graph $G[Q]$, and ℓ be the maximal number of head variables from Q that occur in a set $\mathcal{C}_i$.

We prove that $\ell = k$ holds. We first observe that, for $i, j \in [1, n]$ with i, j, the two components $\mathcal{C}_i$ and $\mathcal{C}_j$ only share head variables, because if they were to share a quantified variable, there would be an edge between them. Thus, $\mathcal{C}_1, \ldots, \mathcal{C}_n$ have Property (b) of Definition 5.4.3. Furthermore, we have $|\mathsf{vars}(\mathcal{C}_i) \cap \mathsf{vars}(\mathsf{head}(Q))| \leq \ell$. We can conclude that $k \leq \ell$, thanks to Definition 5.4.3.

To show that $\ell \leq k$, it suffices to prove that every connected component $\mathcal{C}_i$ in $G[Q]$ is contained in a set $\mathcal{B}_j$, for some $j \in [1, m]$. Towards a contradiction we assume that there is a connected component $\mathcal{C}_i$ which contains atoms from some $\mathcal{B}_j$ and another set. That is, there are atoms $A, A' \in \mathcal{C}_i$ with $A \in \mathcal{B}_j$ and $A' \notin \mathcal{B}_j$. Since $\mathcal{C}_i$ is connected there is a sequence $A = A_1, \ldots, A_p = A'$ of (pairwise different) atoms such that $\{A_s, A_s{+}1\}$ is an edge in $G[Q]$, for all $s \in [1, p-1]$. Let now s be the smallest index such that $A_s \in \mathcal{B}_j$ but $A_{s+1} \notin \mathcal{B}_j$. Furthermore, let $p \neq j$ be such that $A_{s+1} \in \mathcal{B}_p$.

By definition of the edge relation of $G[Q]$, the atoms A_s and A_{s+1} share a quantified variables x, which is then also in $\mathsf{vars}(\mathcal{B}_j) \cap \mathsf{vars}(\mathcal{B}_p)$. But, since $x \notin \mathsf{vars}(\mathsf{head}(Q))$, the

sets $\mathcal{B}_j$ and $\mathcal{B}_p$ do *not* obey Property (b) of Definition 5.4.3. This is a contradiction to $\mathcal{B}_1, \ldots, \mathcal{B}_m$ witnessing that Q has weak head arity k.

We can conclude that every component $\mathcal{C}_i$ is contained in a set $\mathcal{B}_j$, and that thus $\ell \leq k$ holds. Moreover, since the $\mathcal{C}_1, \ldots, \mathcal{C}_n$ comprise a partition of $\mathsf{body}(Q)$ and posses Property (b) of Definition 5.4.3, they indeed witness Q having weak head arity $\ell = k$.

Obtaining $G[Q]$ from Q is straightforward, and it is well-known that connected components can be computed in polynomial time. Consequently, $\mathcal{C}_1, \ldots, \mathcal{C}_n$, and hence ℓ, can be computed in polynomial time. □

By replacing the partitioning phase in the proof of Proposition 5.4.1 with the algorithm guaranteed by Lemma 5.4.7, and in combination with the proof of Theorem 5.4.2, we obtain the following generalization of Theorem 5.4.2.

Proposition 5.4.8. *For every $k \geq 0$, the acyclic rewriting problem for the classes of acyclic conjunctive queries and acyclic views with weak head arity k is in* P*.*

5.5 Hierarchical and Quantified-Hierarchical Rewritings

Thanks to Theorem 5.3.5, we know that every acyclic conjunctive query has an acyclic rewriting, if it has a rewriting at all. The same is true for free-connex acyclic conjunctive queries. In other words, the structural properties of these queries, which guarantee – among other benefits – good complexity bounds for query evaluation, transfer to rewritings. It is natural to ask whether other, stronger structural properties transfer in the same fashion. In this section, we study this question for hierarchical and q-hierarchical conjunctive queries.

In Section 5.3.2 we also proved that the cover description problem is NP-hard for q-hierarchical conjunctive queries. For the acyclic rewriting problem we established NP-hardness "only" for hierarchical conjunctive queries. Towards the end of this section, we will address this discrepancy, and conclude some complexity results for deciding the existence of hierarchical and q-hierarchical rewritings.

On the Existence of Hierarchical and Quantified-Hierarchical Rewritings. As for acyclic queries and rewritings, the canonical rewriting is *not* necessarily hierarchical, even if a hierarchical rewriting exists. The same applies for q-hierarchical queries. We illustrate this fact by means of an example.

Example 5.5.1. Consider the two views

$$V_1(x_1, y_1) \leftarrow R(x_1), S(y_1) \quad \text{and} \quad V_2(z_2) \leftarrow T(z_2)$$

and the q-hierarchical conjunctive query defined by $H(x, y) \leftarrow R(x), S(y), T(x), T(y)$.

The canonical rewriting

$$H(x, y) \leftarrow V_1(x, y), V_2(x), V_2(y)$$

is *not* q-hierarchical, since $\mathsf{atoms}(x)$ and $\mathsf{atoms}(y)$ are neither disjoint nor subsets of one another. Thus, it is *not* even hierarchical. However, there is a q-hierarchical rewriting, namely

$$H(x, y) \leftarrow V_1(x, y'), V_1(x', y), V_2(x), V_2(y). \qquad \triangleleft$$

The main result of this section is an analogue of Theorem 5.3.5 for hierarchical and q-hierarchical queries.

Theorem 5.5.2. *Let Q be a conjunctive query and $\mathcal{V}$ be a set of views.*

(a) *If Q is hierarchical and $\mathcal{V}$-rewritable, then there is a hierarchical $\mathcal{V}$-rewriting of Q.*

(b) *If Q is q-hierarchical and $\mathcal{V}$-rewritable, then there is a q-hierarchical $\mathcal{V}$-rewriting of Q.*

The rewritings guaranteed by Statements (a) and (b) have length at most $|Q|$. Moreover, given a cover partition over $\mathcal{V}$ for Q, a hierarchical (or q-hierarchical, respectively) rewriting of Q can be computed in polynomial time.

Similarly, as for Theorem 5.3.5, the proof idea for Theorem 5.5.2 is to refine cover partitions by splitting cover descriptions with respect to the structure of the query Q. However, Example 5.5.1 suggests that it does *not* suffice to do so with respect to the join tree of Q: The canonical rewriting in this example is already acyclic, but *not* hierarchical.

Thus, we employ another strategy for splitting cover descriptions. More concretely, a cover description $\langle \mathcal{A}, V, \alpha, \psi \rangle$ is split according to the partitioning of $\mathcal{A}$ guaranteed by the following result.

Lemma 5.5.3. *Let Q be a hierarchical conjunctive query and $\langle \mathcal{A}, V, \alpha, \psi \rangle$ be a cover description for Q. There is a polynomial time algorithm that partitions $\mathcal{A}$ into sets $\mathcal{A}_1, \ldots, \mathcal{A}_n$ that have the following properties.*

(A) *Each variable $y \notin \alpha(\mathsf{vars}(\mathsf{head}(V)))$ appears in at most one set $\mathcal{A}_i$.*

(B) *Each set $\mathcal{A}_i$ is a singleton set, or there is a variable $x \notin \alpha(\mathsf{vars}(\mathsf{head}(V)))$ that appears in every atom in $\mathcal{A}_i$.*

Proof. We employ an undirected graph G similar to the cover graphs utilized in Section 5.4. Let G be the graph whose nodes are the atoms from $\mathcal{A}$, and which has an edge between two atoms $A, A' \in \mathcal{A}$, if x is a variable that occurs in A and A' but *not* in $\alpha(\mathsf{vars}(\mathsf{head}(V)))$.[17]

Let $\mathcal{A}_1, \ldots, \mathcal{A}_n$ be the connected components of G. We prove that $\mathcal{A}_1, \ldots, \mathcal{A}_n$ have Properties (A) and (B). Thus, it suffices to compute the graph G and then its connected components to obtain the desired partition. Clearly, this can be done in polynomial time.

The sets $\mathcal{A}_1, \ldots, \mathcal{A}_n$ have Property (A) because, otherwise, there would be an edge between two different sets $\mathcal{A}_i$ and $\mathcal{A}_j$ – a contradiction to $\mathcal{A}_i$ and $\mathcal{A}_j$ being connected components.

[17] We note that, in contrast to a cover graph, the edge relation is defined with respect to *another* query, namely V, instead of $\mathsf{head}(Q)$.

To establish Property (B) we make a case distinction. In case a set $\mathcal{A}_i$ contains an atom A with $\mathsf{vars}(A) \subseteq \alpha(\mathsf{vars}(\mathsf{head}(V)))$, we have that $\mathcal{A}_i$ is the singleton set $\mathcal{A}_i = \{A\}$, because there are no edges for A.

The other case is that every atom A in $\mathcal{A}_i$ contains at least one variable $y \notin \mathsf{vars}(\mathsf{head}(\alpha(V)))$. We prove that there is a variable $x \notin \mathsf{vars}(\mathsf{head}(\alpha(V)))$ such that $\mathsf{atoms}(x) = \mathcal{A}_i$. This then implies Property (B) because $\mathsf{atoms}(x)$ is by definition the set of precisely those atoms in which x occurs.

To this end, observe that, for every variable $y \notin \alpha(\mathsf{vars}(\mathsf{head}(V)))$, the set $\mathcal{C}_y$ of atoms from $\mathcal{A}$ in which y occurs constitutes a (not necessarily maximal) clique in G. Moreover, y cannot be a bridge variable, because that would violate Condition (2) of Definition 5.2.3. Thus, we have $\mathsf{atoms}(y) \subseteq \mathcal{A}_i$ and $\mathcal{C}_y = \mathsf{atoms}(y)$.

Recall that, since Q is hierarchical, two sets $\mathsf{atoms}(x)$ and $\mathsf{atoms}(y)$, for two different variables x, y, are either disjoint or one contains another, due to Definition 2.4.2. Since $\mathcal{A}_i$ is connected, it readily follows, that there is a variable $x \notin \mathsf{vars}(\mathsf{head}(\alpha(V)))$ in $\mathsf{vars}(\mathcal{A}_i)$ such that $\mathsf{atoms}(y) \subseteq \mathsf{atoms}(x)$ holds, for every variable y that occurs in $\mathcal{A}_i$ but *not* in $\mathsf{vars}(\mathsf{head}(\alpha(V)))$. Finally, since every atom in $\mathcal{A}_i$ contains at least one variable $y \notin \mathsf{vars}(\mathsf{head}(\alpha(V)))$, we can conclude that $\mathcal{A}_i = \mathsf{atoms}(x)$ holds. □

Now we are prepared to prove Theorem 5.5.2.

Proof of Theorem 5.5.2. Towards Statement (a), let Q be a $\mathcal{V}$-rewritable hierarchical conjunctive query. We can assume that Q is minimal, thanks to Remark 5.2.14. Since Q is $\mathcal{V}$-rewritable, there is a cover partition for Q, thanks to Theorem 5.2.8.

We describe how this cover partition can be transformed into a cover partition that corresponds to a hierarchical rewriting. Let $d = \langle \mathcal{A}, V, \alpha, \psi \rangle$ be a cover description of the cover partition for Q. Furthermore, let $\mathcal{A}_1, \ldots, \mathcal{A}_n$ be sets constituting the partition of $\mathcal{A}$ guaranteed by Lemma 5.5.3. For each $i \in [1, n]$, the tuple $d_i = \langle \mathcal{A}, V, \alpha, \psi \rangle$ is a cover description. Indeed, it is easy to verify that d_i satisfies Conditions (1), (3) and (4) of Definition 5.2.3 because the original cover description d does. The same is true for Condition (2) because the sets $\mathcal{A}_1, \ldots, \mathcal{A}_n$ have Property (A) as stated by Lemma 5.5.3.

Therefore, and since the $\mathcal{A}_1, \ldots, \mathcal{A}_n$ form a partition of $\mathcal{A}$, the cover description d can be replaced with $d_1, \ldots, d_n$. Let $\mathcal{P}$ be the cover partition obtained by replacing every cover description in the original partition as described above. We can assume that $\mathcal{P}$ is consistent because every cover partition can be transformed into a consistent cover partition with the same underlying partition of $\mathsf{body}(Q)$, thanks to Lemma 5.2.9. All of these transformations can be done in polynomial time thanks to Lemmas 5.2.9 and 5.5.3.

For readability, let R denote the rewriting obtained from $\mathcal{P}$, i.e., $R = \mathsf{q}(\mathcal{P})$. In the remainder of the proof for Statement (a), we show that R is hierarchical; that is, we prove that every pair x, y of variables from $\mathsf{vars}(R)$ satisfies (at least) one of the conditions of Definition 2.4.2. To this end, let x and y be two variables in $\mathsf{vars}(R)$.

If x occurs in only one atom of $\mathsf{body}(R)$ then there are two cases: **(i)** If in the only atom in which x occurs, y occurs as well, then $\mathsf{atoms}_R(x) \subseteq \mathsf{atoms}_R(y)$, and **(ii)** otherwise, $\mathsf{atoms}_R(x) \cap \mathsf{atoms}_R(y) = \emptyset$ holds. Analogously, if y occurs in only one atom of $\mathsf{body}(R)$ then either $\mathsf{atoms}_R(y) \subseteq \mathsf{atoms}_R(x)$ or $\mathsf{atoms}_R(x) \cap \mathsf{atoms}_R(y) = \emptyset$ holds.

Let us finally assume that both x and y occur in at least two atoms of $\mathsf{body}(R)$ each. Thanks to $\mathcal{P}$ being a consistent cover partition and Condition (2) of Definition 5.2.3, x and y are bridge variables, and occur in atom sets of at least two different cover descriptions of $\mathcal{P}$ each. Since Q is hierarchical we have the following three cases.

Case 1: $\mathsf{atoms}_Q(x) \subseteq \mathsf{atoms}_Q(y)$. Let $\langle \mathcal{A}, V, \alpha, \psi \rangle$ from $\mathcal{P}$ be a cover description such that $x \in \alpha(\mathsf{head}(V))$. In particular, x thus occurs in an atom $A \in \mathcal{A}$. Therefore, y also occurs in A, since $\mathsf{atoms}_Q(x) \subseteq \mathsf{atoms}_Q(y)$. Since y is a bridge variable by assumption, it follows that $y \in \alpha(\mathsf{head}(V))$ thanks to Condition (2) of Definition 5.2.3. We conclude that $\mathsf{atoms}_R(x) \subseteq \mathsf{atoms}_R(y)$ holds.

Case 2: $\mathsf{atoms}_Q(y) \subseteq \mathsf{atoms}_Q(x)$. This case is analogous to the first case.

Case 3: $\mathsf{atoms}_Q(x) \cap \mathsf{atoms}_Q(y) = \emptyset$. If there is *no* cover description in $\mathcal{P}$, in which x and y occur together, then $\mathsf{atoms}_R(x) \cap \mathsf{atoms}_R(y) = \emptyset$ holds. Let us thus assume, for the sake of a contradiction, that there is a cover description $\langle \mathcal{A}, V, \alpha, \psi \rangle$ in $\mathcal{P}$ in which both x and y occur. Thanks to $\mathsf{atoms}_Q(x) \cap \mathsf{atoms}_Q(y) = \emptyset$, set $\mathcal{A}$ contains at least two atoms. Then there is a variable $z \notin \mathsf{vars}(\mathsf{head}(\alpha(V)))$ that appears in all atoms of $\mathcal{A}$, because $\mathcal{A}$ is *not* a singleton and has Property (B) as stated in Lemma 5.5.3. Since Q is hierarchical, we must have $\mathsf{atoms}_Q(x) \subsetneq \mathsf{atoms}_Q(z)$. However, this yields a contradiction, because x occurs in at least two cover descriptions, and therefore outside $\mathcal{A}$, whereas z does *not*.

This concludes the proof of Statement (a).

Towards Statement (b), let us assume that Q is q-hierarchical. Our goal is to show that, for all variables $x, y \in \mathsf{vars}(R)$, if $\mathsf{atoms}_R(x) \subsetneq \mathsf{atoms}_R(y)$ holds and x occurs in the head of R, then y occurs in the head of R as well. Note that x and y are both bridge variables because x occurs in the head of Q – which is the same as the head of R – and y occurs in at least two atoms of $\mathsf{body}(R)$ due to $\mathsf{atoms}_R(x) \subsetneq \mathsf{atoms}_R(y)$. In particular, x and y both occur in Q. Since the heads of Q and R are the same and Q is q-hierarchical, whenever $\mathsf{atoms}_Q(x) \subsetneq \mathsf{atoms}_Q(y)$ holds, we can conclude that if x is in the head of R, then y is also in the head of R.

In the remainder we assume, for the sake of a contradiction, that $\mathsf{atoms}_Q(x) \subsetneq \mathsf{atoms}_Q(y)$ does *not* hold. Since Q is hierarchical this means that either $\mathsf{atoms}_Q(x) \supseteq \mathsf{atoms}_Q(y)$ or $\mathsf{atoms}_Q(x) \cap \mathsf{atoms}_Q(y) = \emptyset$ holds.[18]

If x also occurs in at least two atoms of $\mathsf{body}(R)$, then the preconditions for Cases 2 and 3 in the proof for Statement (a) above are met. Thus, $\mathsf{atoms}_Q(x) \supseteq \mathsf{atoms}_Q(y)$ and $\mathsf{atoms}_Q(x) \cap \mathsf{atoms}_Q(y) = \emptyset$ imply $\mathsf{atoms}_R(x) \supseteq \mathsf{atoms}_R(y)$ and $\mathsf{atoms}_R(x) \cap \mathsf{atoms}_R(y) = \emptyset$, respectively. But this is a contradiction to $\mathsf{atoms}_R(x) \subsetneq \mathsf{atoms}_R(y)$.

It remains to consider the case that x occurs in exactly one atom B of $\mathsf{body}(R)$. The variable y then occurs in B and outside B, because $\mathsf{atoms}_R(x) \subsetneq \mathsf{atoms}_R(y)$. Since x and y are bridge variables, it follows that x occurs in exactly one atom set $\mathcal{A}$ of $\mathcal{P}$ and y occurs in and outside $\mathcal{A}$. Hence, $\mathsf{atoms}_Q(x) \cap \mathsf{atoms}_Q(y) = \emptyset$ holds because $\mathsf{atoms}_Q(x) \supseteq \mathsf{atoms}_Q(y)$ cannot. This implies that $\mathcal{A}$ consists of at least two atoms, since x and y co-occur

[18] We note that the case $\mathsf{atoms}_Q(x) = \mathsf{atoms}_Q(y)$ is a special case of $\mathsf{atoms}_Q(x) \supseteq \mathsf{atoms}_Q(y)$.

in $\mathcal{A}$. Therefore, there is a non-bridge variable z that occurs in all atoms of $\mathcal{A}$ thanks to Property (B) as stated in Lemma 5.5.3. But then $\mathsf{atoms}_Q(x) \subsetneq \mathsf{atoms}_Q(z)$ holds, and, since z is a quantified variable, x is a quantified variable as well, because Q is q-hierarchical. This is a contradiction to x being a head variable.

All in all, we can conclude that $\mathsf{atoms}_Q(x) \subsetneq \mathsf{atoms}_Q(y)$ holds, and therefore, that y is a head variable. Thus, R is q-hierarchical. □

Complexity Results. Similarly to Theorem 5.3.5, Theorem 5.5.2 delivers good news as well as bad news. The good news is that, since the inclusions $\mathsf{QHCQ} \subseteq \mathsf{CCQ}$ and $\mathsf{QHCQ} \subseteq \mathsf{HCQ} \subseteq \mathsf{ACQ}$ hold (cf. Proposition 2.4.3), the rewriting problem for q-hierarchical views and hierarchical conjunctive queries over a fixed schema is tractable thanks to Theorem 5.4.2 and Theorem 5.5.2.

Corollary 5.5.4. *For every $k \geq 0$, the following two statements hold.*

(1) $\textsc{Rewr}^k(\mathsf{QHCQ}, \mathsf{HCQ}, \mathsf{HCQ})$ *is in* P*, and a hierarchical rewriting can be computed in polynomial time, if it exists.*

(2) $\textsc{Rewr}^k(\mathsf{QHCQ}, \mathsf{QHCQ}, \mathsf{QHCQ})$ *is in* P*, and a q-hierarchical rewriting can be computed in polynomial time, if it exists.*

The computability of a hierarchical (or q-hierarchical) in polynomial time can be proven analogously to Theorem 5.4.2, the only difference is that Theorem 5.3.5 is exchanged for Theorem 5.5.2.

We emphasize that Corollary 5.5.4 contrasts $\textsc{CovDesc}^k(\mathsf{QHCQ}, \mathsf{QHCQ})$ being NP-hard for $k \geq 2$, as stated by Theorem 5.3.9.

The bad news delivered by Theorem 5.5.2 is that Theorem 5.3.8 and Theorem 5.5.2 imply NP-completeness for hierarchical conjunctive queries and views.

Corollary 5.5.5. $\textsc{Rewr}^k(\mathsf{HCQ}, \mathsf{HCQ}, \mathsf{HCQ})$ *is* NP*-complete for every $k \geq 3$.*

Of course, Corollary 5.5.5 implies NP-hardness of $\textsc{Rewr}^k(\mathbb{V}, \mathbb{Q}, \mathsf{ACQ})$, for all pairs $\mathbb{V}, \mathbb{Q}$ of classes with $\mathsf{HCQ} \subseteq \mathbb{V} \subseteq \mathsf{CQ}$ and $\mathsf{HCQ} \subseteq \mathbb{Q} \subseteq \mathsf{CQ}$.

5.6 Discussion and Related Work

In this section we discuss literature related to our results and techniques. A large part of this discussion is actually dedicated to a comparison of our characterization (Theorem 5.2.8) with similar notions utilized in prior work. Notably, this will lead to another result concerning "minimal" cover descriptions. For (some) applications of structurally simple queries – which are, of course, also applicable to structurally simple rewritings – we refer to Section 1.1.3.

The literature on rewritings is vast, even if restricted to conjunctive queries. The notion of rewritings has been considered for many query languages, database models, and variations of semantics; including languages relevant in practise, like `XML` and SQL. We refer to the surveys of Chirkova and Yang [CY12] and Halevy [Hal01] for an overview

on rewritings in general. For rewritings of conjunctive queries, we already covered the most essential results in Section 5.1. For more details (and results), the book of Afrati and Chirkova [AC19] is worth reading, if accessible.

Determinacy. A notion closely related to rewritability is *determinacy*. A set $\mathcal{V}$ of views *determines* a query Q if, for every database D, the query result $Q(D)$ can be obtained from $\mathcal{V}(D)$. Of course, if Q is $\mathcal{V}$-rewritable, then $\mathcal{V}$ determines Q. The converse, however, does *not* hold in general – it might be possible to obtain the query result $Q(D)$ from $\mathcal{V}(D)$ by other means than evaluating a rewriting. Nash et al. [NSV10] considered the notion of *completeness* for classes of rewritings. More precisely, for classes $\mathbb{Q}$, $\mathbb{V}$, and $\mathbb{R}$ of queries, the class $\mathbb{R}$ is called *complete for* $\mathbb{V}$*-to-*$\mathbb{Q}$ *rewritings*, if, for each $Q \in \mathbb{Q}$ and $\mathcal{V} \subseteq \mathbb{V}$, the query Q is $\mathcal{V}$-rewritable, whenever $\mathcal{V}$ determines Q. Nash et al. [NSV10, Theorem 5.2] proved that CQ is *not* complete for CQ-to-CQ rewritings. In fact, the query and views they used to prove this result are all acyclic (but neither free-connex acyclic nor hierarchical). Thus, they actually proved, that CQ is *not* even complete for ACQ-to-ACQ rewritings. We emphasize that this obviously implies that *no* subclass of CQ is complete for ACQ-to-ACQ rewritings, in particular ACQ. Consequently, the prerequisite of Theorem 5.3.5 cannot be lifted from rewritability to the more general notion of determinacy.

Characterizations. We already mentioned that our notion of cover partitions is similar to various notions from the literature. We briefly discuss three of them here.

- Afrati and Chirkova [AC19] present algorithms for finding exact (or complete) rewritings with a minimal number of atoms and (maximally) contained rewritings. For this purpose they consider triples of the form (S, S', h) which are comparable to cover descriptions [AC19, Definition 3.12]. Namely, S' corresponds to the set $\mathcal{A}$ in a cover description, $S = \mathsf{body}(Q)$, and h is a homomorphism from S' into some (implicit) view V. In our characterization we can always assume $h = \mathsf{id}$ (and, thus, omit it in the specification of a cover description), thanks to the view application α. We note that h can be assumed to be one-to-one for (equivalent) $\mathcal{V}$-rewritings [AC19, Theorem 3.15]. Furthermore, for (equivalent) rewritings, only candidates whose body is a proper subset of the canonical rewriting's body are considered [AC19, Section 3.2.3]. The existence of a body homomorphism ψ which maps $\alpha(V)$ into Q is guaranteed for such candidates, and ψ determines the view application α which includes, in particular, the unification of variables in $\mathsf{head}(V)$. However, it does *not* suffice to consider such candidates, if one wants to obtain an acyclic rewriting, as we have seen in Example 5.3.1. This is why α and ψ are part of a cover description.

  Let us point out that Condition (2) of Definition 5.2.3 is equivalent to the *shared-variable property* [AC19, Definition 3.12]. An analogue of the implication (a) ⇒ (b) of Theorem 5.2.8 is also proven [AC19, Theorem 3.15] and, based on that, an algorithm CoreCover for finding (equivalent) rewritings is derived. This algorithm, however, considers only triples with maximal sets S' (called *tuple cores*) and they are allowed to overlap, i.e. they do not have to form a partition. In contrast, we consider non-maximal

sets. And indeed, in many cases, our proofs rely on the ability to split sets, and thus on non-maximal sets.

Afrati and Chirkova also provide a proof for an analogue of (b) ⇒ (a) for (maximally) contained rewritings [AC19, implied by the proof of Theorem 4.19]. Interestingly, the proof of this result (and the associated algorithm) exploits triples with minimal sets S' (that still satisfy the shared-variable property) and a partition property. We discuss the relation of this minimality constraint with our results below in more detail. An analogue of (b) ⇒ (a) for (equivalent) $\mathcal{V}$-rewritings, and, therefore, a characterization for $\mathcal{V}$-rewritability, is not stated (nor implied).

- Gou et al. [GKC06] employ a characterization for $\mathcal{V}$-rewritability to find rewritings efficiently. It is in terms of *tuple coverages* and a *partition condition* corresponding to cover descriptions and partitions, respectively [GKC06, Theorem 5, Theorem 6]. A tuple coverage, denoted $s(t_V, Q)$, is a (non-empty) set G, where t_V corresponds to $\alpha(V)$ and G to the set $\mathcal{A}$ of a cover description. Similarly to the notion of Afrati and Chirkova [AC19], the most crucial difference in comparison with our characterization is that only rewritings whose body is contained in the canonical rewriting – that is, for which $\mathsf{vars}(\alpha(V)) \subseteq \mathsf{vars}(Q)$ holds for every tuple coverage – are considered. Besides, the mappings ψ and α and their associated conditions are not denoted explicitly; instead it is required that G is isomorphic to a subset of $\alpha(V)$ and that the analogue of our Condition (2) holds. This is equivalent to the definitions of Definition 5.2.3, if restricted to the rewritings considered by Gou et al.

- Lastly, Pottinger and Halevy [PH01] use *MiniCon descriptions (MCDs)* to compute (maximally) contained rewritings – which are unions of conjunctive queries, in general. A MCD is a tuple of the form (h, V, φ, G) which relates to a cover description $\langle \mathcal{A}, V, \alpha, \psi \rangle$ as follows: **(i)** h is called a head homomorphism and is basically the restriction of a view application α to the head variables of the view V, **(ii)** G corresponds to the set $\mathcal{A}$, and **(iii)** φ is a mapping embedding G into $h(V)$. The component φ has no counterpart in a cover description because, thanks to α being able to rename any variable in V, we can always assume $\varphi = \mathsf{id}$. We note that the view applications α_i of a consistent cover partition $\mathcal{P}$ also allow us to conveniently denote the expansion of the associated rewriting $\mathsf{q}(\mathcal{P})$. On the other hand, there is no counterpart for the body homomorphism ψ in a MCD. However, since the ψ_i in a consistent cover partition ensure that the query is contained in the associated expansion of the rewriting, there is also no need for them if contained rewritings are considered. Condition (2) of Definition 5.2.3 is stated as [PH01, Property 1] and the idea of a cover partition in [PH01, Property 2].

Minimal Cover Descriptions. As mentioned above, Afrati and Chirkova [AC19, Definition 4.16] employ "minimal" triples to construct maximally contained rewritings. In terms of cover descriptions, an analogous definition of minimality is as follows.

Definition 5.6.1 (Minimal Cover Description)**.** A cover description $\langle \mathcal{A}, V, \alpha, \psi \rangle$ for a conjunctive query Q is *minimal* if $\mathcal{A}$ *cannot* be partitioned into $n \geq 2$ non-empty

sets $\mathcal{A}_1, \ldots, \mathcal{A}_n$ such that, for each $i \in [1, n]$, there is a cover description of the form $\langle \mathcal{A}_i, V, \alpha_i, \psi_i \rangle$ for Q.

Let us emphasize that Definition 5.6.1 allows for different mappings α_i, ψ_i, for every set $\mathcal{A}_i$. The sets of the partition guaranteed by Lemma 5.5.3 are thus *not* necessarily minimal, although, they are in some sense "minimal" with respect to fixed mappings α, and ψ.

Example 5.6.2. Consider the hierarchical view

$$V(x_1, v, x_2) \leftarrow R(u, x_1), S(u), R(v, x_2), S(v),$$

and the hierarchical conjunctive query Q defined by

$$H(x, x') \leftarrow R(u, x), S(u), R(u', x'), S(u').$$

Let further $\langle \mathcal{A}, V, \alpha, \mathsf{id} \rangle$ be the cover description with $\mathcal{A} = \{R(u, x), S(u)\}$ and

$$\alpha = \{u \mapsto u, x_1 \mapsto x, v \mapsto u', x_2 \mapsto x'\}.$$

The trivial partition of $\mathcal{A}$ that just consists of $\mathcal{A}$ itself has Properties (A) and (B) as stated in Lemma 5.5.3. It also cannot be further refined without changing α because the variable $u \notin \alpha(\mathsf{vars}(\mathsf{head}(V)))$ would then occur in two different sets, and, in particular, become a bridge variable.

However, $\langle \mathcal{A}, V, \alpha, \mathsf{id} \rangle$ is *not* minimal. This is witnessed by the cover descriptions $\langle \{R(u, x)\}, V, \alpha', \mathsf{id} \rangle$ and $\langle \{S(u)\}, V, \alpha', \mathsf{id} \rangle$ with

$$\alpha' = \{u \mapsto u, x_1 \mapsto x, v \mapsto u, x_2 \mapsto x\}. \qquad \triangleleft$$

It might be tempting to always make use of minimal cover descriptions. In particular, they could be exploited to simplify the constructions in the proofs for Theorem 5.3.5 and Theorem 5.5.2: Instead of partitioning a set $\mathcal{A}$ of a cover description "manually", minimal cover descriptions could be computed. However, there are potential drawbacks to this approach. For instance, it might result in larger rewritings. And, more importantly, even deciding whether a given cover description is minimal is coNP-hard. As a consequence, using minimal cover descriptions in our construction results in (probably) inefficient algorithms – in contrast to our "manual" constructions. Formally, for classes $\mathbb{V}$ and $\mathbb{Q}$ of views and conjunctive queries, respectively, we consider the following decision problem.

— MinCovDesc($\mathbb{V}$, $\mathbb{Q}$) —

Given: View $V \in \mathbb{V}$, query $Q \in \mathbb{Q}$, cover description $\langle \mathcal{A}, V, \alpha, \psi \rangle$ for Q

Question: Is $\langle \mathcal{A}, V, \alpha, \psi \rangle$ minimal?

Proposition 5.6.3. MinCovDesc(HCQ, HCQ) *is* coNP*-hard.*

Proof. We reduce the problem Rewr(HCQ, HCQ, CQ), which is NP-hard due to Theorem 5.3.8, to the *complement* of MinCovDesc(HCQ, HCQ). For a hierarchical conjunctive query Q and a set $\mathcal{V}$ of hierarchical views, we describe the construction of a hierarchical conjunctive query Q' and a cover description d' for Q' such that d' is *not* minimal if and only if there is a $\mathcal{V}$-rewriting for Q.

Similarly to our approach in the proof for Theorem 5.3.8, we use quantified variables to "bind" atoms together. Recall that, for an atom $A = R(x_1, \ldots, x_k)$ and a variable u, we write A^u for the atom $R^u(u, x_1, \ldots, x_k)$. This notation naturally extends to sets $\mathcal{A}$ of atoms. That is, we have $\mathcal{A}^u = \{A^u \mid A \in \mathcal{A}\}$. Furthermore, if a set of atoms or a query consists of atoms extended by (possibly different) variables as described above, we often signify this with a "$_+$"-symbol in the superscript, e.g. we write $\mathcal{A}^+$.

Further on, let $V_1, \ldots, V_n$ be the views in $\mathcal{V}$. We assume that the views in $\mathcal{V}$ and the query Q refer to distinct variables, which is no restriction since the variables can be renamed accordingly in polynomial time.

Construction. We start with the construction of the view for the cover description d'. To define this view (and eventually also the query Q'), we consider views $V_1^+, \ldots, V_n^+$ that are obtained from the input views $V_1, \ldots, V_n$ by a distinguished variable v_i to each atom of V_i, including the head atom. More precisely, we define $V_1^+, \ldots, V_n^+$ as the views with $\mathsf{head}(V_i^+) = \mathsf{head}(V_i)^{v_i}$ and $\mathsf{body}(V_i^+) = \mathsf{body}(V_i)^{v_i}$ where $v_1, \ldots, v_n$ are pairwise distinct variables that do not occur in the query Q or in any of the views in $\mathcal{V}$. In the same fashion, we derive a set of atoms from $\mathsf{body}(Q)$, but we add a fresh atom whose relation symbol does *not* occur in Q or in any of the views in $\mathcal{V}$. That is, for a fresh variable u and a fresh relation symbol S, we set $\mathcal{A}_Q^+ = \mathsf{body}(Q)^u \cup \{S(u)\}$.

We are now ready to define the view W for the cover description d'. Its body is

$$\mathsf{body}(W) = \mathcal{A}_Q^+ \cup \bigcup_{i=1}^{n} \mathsf{body}(V_i^+) \cup \{S(v_{n+1})\}$$

where v_{n+1} is again a fresh variable that does *not* occur anywhere else. The head of W contains all the head variables from Q and V_i, for all $i \in [1, n]$, as well as the variables $v_1, \ldots, v_n, v_{n+1}$, but *not* u. Hence, W is a "super view" that encompasses all the V_i^+ and the extended version of Q. In particular, for each view V_i^+, we have that the set $\mathsf{body}(V_i^+)$ is contained in $\mathsf{body}(W)$.

The query Q' is defined by the rule $\mathsf{head}(Q) \leftarrow \mathsf{body}(W)$, that is Q' has the same head as the original query Q and its body coincides with the body of W.

Finally, we define the cover description d' as $\langle \mathcal{A}_Q^+, W, \mathsf{id}, \mathsf{id} \rangle$. Note that the only bridge variables of $\mathcal{A}_Q^+$ are the variables in $\mathsf{head}(Q)$, since $\mathcal{A}_Q^+$ does *not* share any variable with any of the (extended) views. Thus, d' fulfils Condition (2) of Definition 5.2.3, because all head variables from Q also occur in $\mathsf{head}(W)$. The other conditions of Definition 5.2.3 are satisfied trivially. Thus, d' is indeed a cover description for Q'.

We observe that the views V_i^+ are hierarchical since, for the new variables v_i, we have that $\mathsf{atoms}_{V_i^+}(v_i) \supseteq \mathsf{atoms}_{V_i^+}(y)$ for all $y \in \mathsf{vars}(V_i)$, and the given views in $\mathcal{V}$ are hierarchical. The same is true for the set $\mathcal{A}_Q^+$ (interpreted as a Boolean query here).

We can conclude that W and Q' are hierarchical as well, since the views and the query, and, hence, the V_i^+ and $\mathcal{A}_Q^+$ do *not* share any variable. Clearly, W and Q' can be obtained from Q and $\mathcal{V}$ in polynomial time.

Correctness. In the following, we prove that Q has a $\mathcal{V}$-rewriting if and only if d' is *not* minimal.

For the only-if direction assume that Q has a $\mathcal{V}$-rewriting. Then there is a consistent cover partition $\mathcal{P}$ over $\mathcal{V}$ for Q, thanks to Theorem 5.2.8. By construction each cover description $\langle \mathcal{A}, V_j, \alpha, \psi \rangle$ in $\mathcal{P}$ can be turned into a cover description

$$\langle \mathcal{A}^u, V_j^+, \alpha \cup \{v_j \mapsto u\}, \psi \cup \{u \mapsto u\} \rangle$$

for Q' and, furthermore into a cover description $\langle \mathcal{A}^u, W, \alpha', \psi' \rangle$ for Q' where α' and ψ' coincide with $\alpha \cup \{v_j \mapsto u\}$ and $\psi \cup \{u \mapsto u\}$ on their domain, respectively, and are the identity on all other variables. Let $\mathcal{P}'$ be the collection of cover descriptions obtained by transforming every cover description in $\mathcal{P}$ as described above, and the additional cover description $\langle \{S(u)\}, W, \{v_{n+1} \mapsto u\}, \mathsf{id} \rangle$ for Q'.

Since $\mathcal{P}$ is a cover partition for Q and Q does *not* contain an S-atom, the atom sets of the cover descriptions in $\mathcal{P}'$ form a partition of $\mathcal{A}_Q^+ = \mathsf{body}(Q)^u \cup \{S(u)\}$. But then $\langle \mathcal{A}_Q^+, W, \mathsf{id}, \mathsf{id} \rangle$ is *not* minimal because $\mathcal{P}'$ consists of at least two cover descriptions.

For the other direction, suppose that d' is *not* minimal for Q'. Let

$$\mathcal{P}' = \{\langle \mathcal{A}_1^+, W, \alpha_1, \psi_1 \rangle, \ldots, \langle \mathcal{A}_k^+, W, \alpha_k, \psi_k \rangle\}$$

be a collection of cover descriptions witnessing that $d' = \langle \mathcal{A}_Q^+, W, \mathsf{id}, \mathsf{id} \rangle$ is *not* minimal. From $\mathcal{P}'$ we will derive a cover partition $\mathcal{P}$ for Q witnessing that Q is indeed $\mathcal{V}$-rewritable. For this purpose, we first analyse to which atoms in $\alpha_i(W)$ the atoms of a set $\mathcal{A}_i^+$ are mapped to and then associate views V_j^+ with (subsets of) the sets $\mathcal{A}_i^+$.

We can assume that the α_i fulfil quantified variable disjointness. Moreover, we note that $k \geq 2$, since d' is *not* minimal. Therefore, u is a bridge variable of each set $\mathcal{A}_i^+$ since it occurs in every atom of $\mathcal{A}_Q^+$. Hence, *no* atom from $\mathcal{A}_i^+$ can be mapped into the copy of $\alpha_i(\mathcal{A}_Q^+)$ in $\alpha_i(W)$ because the variable u is *not* a head variable of W. Hence, $\mathcal{A}_i^+$ is a subset of

$$\alpha_i(\mathsf{body}(V_1)^{v_1}) \cup \ldots \cup \alpha_i(\mathsf{body}(V_n)^{v_n}) \cup \{\alpha_i(S(v_{n+1}))\}.$$

Since the sets $\alpha_i(\mathsf{body}(V_j)^{v_j})$ do *not* share any variable that is *not* in $\mathsf{head}(\alpha_i(W))$, each cover description in $\mathcal{P}'$ can be partitioned into cover descriptions

$$\langle \mathcal{B}_{i,1}^+, W, \alpha_i, \psi_i \rangle, \ldots, \langle \mathcal{B}_{i,n}^+, W, \alpha_i, \psi_i \rangle$$

for Q' where $\mathcal{B}_{i,j}^+ = \mathcal{A}_i^+ \cap \alpha_i(\mathsf{body}(V_j)^{v_j})$ and, in case $S(u) \in \mathcal{A}_i^+$, a cover description $\langle \{S(u)\}, W, \alpha_i, \psi_i \rangle$. For the sake of readability we assume that $\mathcal{B}_{i,j}^+ \neq \emptyset$ holds for all i, j. If not, the respective cover descriptions can just be removed from the sequence.

We further observe that the view W in each cover description $\langle \mathcal{B}_{i,j}^+, W, \alpha_i, \psi_i \rangle$ can be replaced by V_j^+, because we have $\mathcal{B}_{i,j}^+ \subseteq \alpha_i(\mathsf{body}(V_j^+))$ by definition. More precisely, each cover description $\langle \mathcal{B}_{i,j}^+, W, \alpha_i, \psi_i \rangle$ can be transformed into a cover description

$\langle \mathcal{B}^+_{i,j}, V^+_j, \alpha_{i,j}, \psi_{i,j} \rangle$ where $\alpha_{i,j}$ and $\psi_{i,j}$ are the restrictions of α_i and ψ_i on $\mathsf{vars}(V^+_j)$, respectively.

Every cover description $\langle \mathcal{B}^+_{i,j}, V^+_j, \alpha_{i,j}, \psi_{i,j} \rangle$ can, in turn, be transformed into a cover description $\langle \mathcal{B}_{i,j}, V_j, \alpha'_{i,j}, \psi'_{i,j} \rangle$ for Q where $\mathcal{B}_{i,j}$ is the atom set with $\mathcal{B}^{v_j}_{i,j} = \mathcal{B}^+_{i,j}$, and the mappings $\alpha'_{i,j}$ and $\psi'_{i,j}$ are the restrictions of $\alpha_{i,j}$ and $\psi_{i,j}$ on $\mathsf{vars}(V_j)$, respectively.

Let now $\mathcal{P}$ be the collection of cover descriptions we obtain by applying the transformations described above to all cover descriptions in $\mathcal{P}'$ and removing cover descriptions with atom set $\{S(u)\}$. By construction, the atom sets in $\mathcal{P}'$ form a partition of $\mathsf{body}(Q)$. Thus, we can conclude that $\mathcal{P}'$ is a cover partition for Q, and hence, Q is $\mathcal{V}$-rewritable, thanks to Theorem 5.2.8. □

Chapter 6

Conclusion

We conclude this thesis by recalling our main results, pointing out open problems, and discussing further prospects.

In Chapter 3 we designed $\mathcal{O}(1)$-time parallel algorithms for evaluating queries. For acyclic conjunctive queries we derived parallel versions of the Yannakakis algorithm with work bound $\mathcal{O}((\mathsf{IN} + \mathsf{IN} \cdot \mathsf{OUT})^{1+\varepsilon})$, for every fixed $\varepsilon > 0$. Since the original, sequential algorithm runs in time $\mathcal{O}(\mathsf{IN} + \mathsf{IN} \cdot \mathsf{OUT})$ these algorithms are *not* work-optimal. On the other hand, as we have discussed in Section 3.7, it seems reasonable to consider $\mathcal{O}(1)$-time parallel algorithms with a work bound of $\mathcal{O}(T^{1+\varepsilon})$, where T is the running time of the best sequential algorithm, as work-efficient. Of course, it would be preferable to sustain this with matching lower bounds – but this seems to be quite challenging, in particular, if we do *not* insist on a compact representation of the query result.

We obtained similar results for free-connex acyclic and semi-join algebra queries, as well as weakly worst-case optimal work algorithms for natural join queries. Furthermore, we could lift our algorithms for (free-connex) acyclic conjunctive query to conjunctive queries – recall that the work bound then depends on the (free-connex) generalized hypertree width. Interestingly, for queries with width at least 2, the work bounds for the general and the dictionary setting collapse.

We have already discussed in Section 3.7 that the motivation for studying work-efficient $\mathcal{O}(1)$-time parallel algorithms originated from the dynamic setting, where databases can be updated, and algorithms have to update query results accordingly, thus *maintaining* them. Pursuing work-efficient $\mathcal{O}(1)$-time parallel algorithms for this setting seems to be a sensible next step. Since it is also possible to maintain reachability queries [Dat+18; SVZ20], it might even be feasible to consider algorithms for maintaining Datalog queries in the dynamic setting.

It could also be worthwhile to study work-efficient constant-time versions of (the preprocessing phase of) enumeration algorithms [see, e.g., BGS20], and to consider succinct representation systems for relational databases, e.g., factorized databases [OZ15].

In Chapter 4 we proved that the parallel-correctness problem for frontier-guarded Datalog queries, hash-based distribution policies, and communication policies defined by modest sets of data-moving distribution constraints is 2ExpTime-complete. We also established an analogous result for the parallel-boundedness problem.

Whether similar results hold for monadic Datalog queries remains open. For the non-transitive communication setting we were able to draw a more complete picture: Here deciding parallel-correctness is 2ExpTime-complete for frontier-guarded Datalog

queries but undecidable for monadic Datalog queries.

Although we know that the parallel-correctness problem for frontier-guarded Datalog queries, hash-based distribution policies, and communication policies defined by sets of data-moving distribution constraints, which are *not* necessarily modest, is undecidable (in the "normal" setting), it remains open where the exact border of decidability lies, even for frontier-guarded Datalog queries. To this end, it might be worth considering other Datalog fragments, and formalism to specify policies. In fact, it would also be interesting to study whether there are sensible formalisms for distribution policies which are *not* hash-based, and yield a decidable parallel-correctness problem.

Regarding distribution constraints, it is open how and in what manner they can be efficiently evaluated. Since any meaningful constraint refers to at least two server variables – for the "sender" and the "receiver" – we expect that evaluating them involves communication. Indeed, it would be interesting to know how the *load* – that is, the number of bits a server can send and receive in each round – for evaluating constraints compares to the load required for sending the actual facts for the query evaluation.

Finally, in Chapter 5 we studied structurally simple rewritings for conjunctive queries. We proved that, if an acyclic conjunctive query has any rewriting, then it is guaranteed to have an acyclic rewriting as well. The same is true for free-connex acyclic, hierarchical, and q-hierarchical conjunctive queries. More precisely, for each of these classes, our characterization of rewritability allowed us to refine a given rewriting along the structure of the query to obtain a rewriting with the same structural properties.

Concerning the complexity of the associated rewriting problems we showed that it is NP-complete to decide whether an acyclic rewriting exists, even if the input query and the views are hierarchical. It turned out that the rewriting problem becomes tractable if the arity of the database schema is bounded, and the views are free-connex acyclic, or q-hierarchical – that is, if they are from one of the classes whose structural property takes the head atom into account. The same applies for acyclic, and hence hierarchical, views if the arity of their head atoms is bounded. In this case, it is *not* necessary that the arity of the database schema is bounded. The cases of free-connex acyclic and q-hierarchical views over unbounded database schemas remain open.

Another open question is the complexity of rewriting problems $\textsc{Rewr}(\mathbb{V}, \mathbb{Q}, \mathbb{R})$ with $\mathbb{R} \subsetneq \mathbb{Q}$, for instance $\textsc{Rewr}(\mathsf{ACQ}, \mathsf{ACQ}, \mathsf{QHCQ})$. So far we have only NP-hardness results for the variants $\textsc{Rewr}(\mathbb{V}, \mathbb{Q}, \mathsf{ACQ})$ and $\textsc{Rewr}(\mathbb{V}, \mathbb{Q}, \mathsf{HCQ})$ where $\mathbb{V}$ and $\mathbb{Q}$ encompass all acyclic, or all hierarchical queries, respectively.

It would also be interesting to study whether our results can be extended to the query classes we studied in Chapters 3 and 4: Conjunctive queries with bounded (free-connex) generalized hypertree width, but also semi-join algebra queries and Datalog queries.

Finally, our approach of refining rewritings yields, in general, structurally simple rewritings *with* self-joins. But for some applications queries without self-joins play an important role, for instance, hierarchical queries in the context of probabilistic databases [FO14; DS07]. Therefore, it seems worthwhile to investigate the existence of structurally simple rewritings without self-joins – and the complexity of the associated decision problem(s).

Bibliography

[AAS13] Serge Abiteboul, Émilien Antoine and Julia Stoyanovich. "The Webdamlog System Managing Distributed Knowledge on the Web". In: *CoRR* abs/1304.4187 (2013).
10.48550/ARXIV.1304.4187.
cs.DB/1304.4187.
Cited on page 146.

[Abi+11] Serge Abiteboul, Meghyn Bienvenu, Alban Galland and Émilien Antoine. "A rule-based language for web data management". In: *Proceedings of the thirtieth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. ACM, 2011, pp. 293–304.
10.1145/1989284.1989320.
Cited on page 146.

[AC19] Foto N. Afrati and Rada Chirkova. *Answering Queries Using Views, Second Edition*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2019.
10.2200/S00884ED2V01Y201811DTM054.
Cited on pages 154, 155, 158, 190, 191.

[Afr+17] Foto N. Afrati, Manas R. Joglekar, Christopher Ré, Semih Salihoglu and Jeffrey D. Ullman. "GYM: A Multiround Distributed Join Algorithm". en. In: *International Conference on Database Theory, ICDT 2017*. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik GmbH, Wadern/Saarbruecken, Germany, 2017, 4:1–4:18.
10.4230/LIPIcs.ICDT.2017.4.
Cited on page 143.

[AGM13] Albert Atserias, Martin Grohe and Dániel Marx. "Size Bounds and Query Plans for Relational Joins". In: *SIAM Journal on Computing* 42.4 (2013), pp. 1737–1767.
10.1137/110859440.
Cited on page 65.

[AHV95] Serge Abiteboul, Richard Hull and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995. ISBN: 0-201-53771-0.
http://webdam.inria.fr/Alice/.
Cited on pages 16, 20, 21.

[Ajt93] Miklós Ajtai. "Approximate Counting with Uniform Constant-Depth Circuits". In: *Advances In Computational Complexity Theory, Proceedings of a DIMACS Workshop, New Jersey, USA, December 3-7, 1990*. Ed. by Jin-Yi Cai. Vol. 13. DIMACS Series in Discrete Mathematics and Theoretical Computer Science. DIMACS/AMS, 1993, pp. 1–20.
10.1090/dimacs/013/01.
Cited on page 219.

[Ame+17] Tom J. Ameloot, Gaetano Geck, Bas Ketsman, Frank Neven and Thomas Schwentick. "Parallel-Correctness and Transferability for Conjunctive Queries". In: *Journal of the ACM* 64.5 (2017), 36:1–36:38.
10.1145/3106412.
Cited on pages 6, 143, 144.

[Apaa] Apache Software Foundation. *Apache Hadoop*.
https://hadoop.apache.org/ (visited on 01/02/2023).
Cited on page 5.

[Apab] Apache Software Foundation. *Apache Spark*.
https://spark.apache.org/ (visited on 01/02/2023).
Cited on page 5.

[Are+21] Marcelo Arenas, Pablo Barceló, Leonid Libkin, Wim Martens and Andreas Pieris. *Database Theory*. Preliminary Version, August 19, 2022. Open source at https://github.com/pdm-book/community, 2021.
Cited on pages 3, 11, 13, 14, 20, 21, 65, 66, 69, 156.

[AU11] Foto N. Afrati and Jeffrey D. Ullman. "Optimizing Multiway Joins in a Map-Reduce Environment". In: *IEEE Transactions on Knowledge and Data Engineering* 23.9 (2011), pp. 1282–1298.
10.1109/TKDE.2011.47.
Cited on page 143.

[AV87] Yossi Azar and Uzi Vishkin. "Tight Comparison Bounds on the Complexity of Parallel Sorting". In: *SIAM Journal on Computing* 16.3 (1987), pp. 458–464.
10.1137/0216032.
Cited on page 76.

[BBS12] Michael Benedikt, Pierre Bourhis and Pierre Senellart. "Monadic Datalog Containment". In: *International Colloquium on Automata, Languages, and Programming, ICALP 2012*. Springer Berlin Heidelberg, 2012, pp. 79–91.
10.1007/978-3-642-31585-5_11.
Cited on pages 103, 140, 234.

[BCO12] Vince Bárány, Balder ten Cate and Martin Otto. "Queries with Guarded

Negation". In: *Proceedings of the Very Large Database Endowment* 5.11 (2012), pp. 1328–1339.
10.14778/2350229.2350250.
Cited on page 6.

[BCS11] Vince Bárány, Balder ten Cate and Luc Segoufin. "Guarded Negation". In: *International Colloquium on Automata, Languages, and Programming, ICALP 2011*. Springer Berlin Heidelberg, 2011, pp. 356–367.
10.1007/978-3-642-22012-8_28.
Cited on page 6.

[BDG07] Guillaume Bagan, Arnaud Durand and Etienne Grandjean. "On Acyclic Conjunctive Queries and Constant Delay Enumeration". In: *Computer Science Logic, 21st International Workshop, CSL 2007, 16th Annual Conference of the EACSL, Lausanne, Switzerland, September 11-15, 2007, Proceedings*. Ed. by Jacques Duparc and Thomas A. Henzinger. Vol. 4646. Lecture Notes in Computer Science. Springer, 2007, pp. 208–222.
10.1007/978-3-540-74915-8_18.
Cited on pages 8, 17, 19.

[Ben+15] Michael Benedikt, Balder Ten Cate, Thomas Colcombet and Michael Vanden Boom. "The complexity of boundedness for guarded logics". In: *2015 30th Annual ACM/IEEE Symposium on Logic in Computer Science*. IEEE, 2015, pp. 293–304.
10.1109/LICS.2015.36.
Cited on pages 25, 26, 140, 145.

[BG81] Philip A. Bernstein and Nathan Goodman. "Power of Natural Semijoins". In: *SIAM Journal on Computing* 10.4 (1981), pp. 751–771.
10.1137/0210059.
Cited on page 61.

[BGS20] Christoph Berkholz, Fabian Gerhardt and Nicole Schweikardt. "Constant delay enumeration for conjunctive queries: a tutorial". In: *ACM SIGLOG News* 7.1 (2020), pp. 4–33.
10.1145/3385634.3385636.
Cited on pages 8, 19, 64, 65, 77, 197.

[BKR15a] Pierre Bourhis, Markus Krötzsch and Sebastian Rudolph. "Reasonable Highly Expressive Query Languages". In: *International Joint Conference on Artificial Intelligence, IJCAI 2015*. 2015, pp. 2826–2832.
Cited on pages 6, 22, 79, 102, 103, 122, 125, 127.

[BKR15b] Pierre Bourhis, Markus Krötzsch and Sebastian Rudolph. *Reasonable Highly Expressive Query Languages: Extended Technical Report*. Technical Report. TU Dresden, May 2015.
https://iccl.inf.tu-dresden.de/web/Techreport3020.
Cited on page 122.

[BKS14] Paul Beame, Paraschos Koutris and Dan Suciu. "Skew in parallel query processing". In: *Proceedings of the 33rd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS'14, Snowbird, UT, USA, June 22-27, 2014*. Ed. by Richard Hull and Martin Grohe. ACM, 2014, pp. 212–223.
10.1145/2594538.2594558.
Cited on page 143.

[BKS17a] Paul Beame, Paraschos Koutris and Dan Suciu. "Communication Steps for Parallel Query Processing". In: *Journal of the ACM* 64.6 (2017), 40:1–40:58.
10.1145/3125644.
Cited on pages 5, 80, 143.

[BKS17b] Christoph Berkholz, Jens Keppeler and Nicole Schweikardt. "Answering Conjunctive Queries under Updates". In: *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2017, Chicago, IL, USA, May 14-19, 2017*. Ed. by Emanuel Sallinger, Jan Van den Bussche and Floris Geerts. ACM, 2017, pp. 303–318.
10.1145/3034786.3034789.
Cited on pages 8, 17.

[BM72] Rudolf Bayer and Edward M. McCreight. "Organization and Maintenance of Large Ordered Indices". In: *Acta Informatica* 1 (1972), pp. 173–189.
10.1007/BF00288683.
Cited on page 77.

[BPR17] Pablo Barceló, Andreas Pieris and Miguel Romero. "Semantic Optimization in Tractable Classes of Conjunctive Queries". In: *ACM SIGMOD Record* 46.2 (2017), pp. 5–17.
10.1145/3137586.3137588.
Cited on page 166.

[Bra13] Johann Brault-Baron. "De la pertinence de l'énumération : complexité en logiques propositionnelle et du premier ordre". Theses. Université de Caen, Apr. 2013.
https://hal.archives-ouvertes.fr/tel-01081392.
Cited on pages 17, 20.

[BRV16] Pablo Barceló, Miguel Romero and Moshe Y. Vardi. "Semantic Acyclicity on Graph Databases". In: *SIAM Journal on Computing* 45.4 (2016), pp. 1339–1376.
10.1137/15M1034714.
Cited on page 165.

[CDR86] Stephen A. Cook, Cynthia Dwork and Rüdiger Reischuk. "Upper and Lower Time Bounds for Parallel Random Access Machines without Simultaneous Writes". In: *SIAM Journal on Computing* 15.1 (1986), pp. 87–97.
10.1137/0215006.
Cited on page 29.

[CGK01] Zhiyuan Chen, Johannes Gehrke and Flip Korn. “Query Optimization In Compressed Database Systems”. In: *Proceedings of the 2001 ACM SIGMOD international conference on Management of data, Santa Barbara, CA, USA, May 21-24, 2001*. Ed. by Sharad Mehrotra and Timos K. Sellis. ACM, 2001, pp. 271–282.
10.1145/375663.375692.
Cited on page 35.

[Cha96] Shiva Chaudhuri. “Sensitive Functions and Approximate Problems”. In: *Information and Computation* 126.2 (1996), pp. 161–168.
10.1006/inco.1996.0043.
Cited on pages 32, 33.

[Che+20a] Hubie Chen, Georg Gottlob, Matthias Lanzinger and Reinhard Pichler. “Semantic Width and the Fixed-Parameter Tractability of Constraint Satisfaction Problems”. In: *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI 2020*. Ed. by Christian Bessiere. ijcai.org, 2020, pp. 1726–1733.
10.24963/ijcai.2020/239.
Cited on page 164.

[Che+20b] Hubie Chen, Georg Gottlob, Matthias Lanzinger and Reinhard Pichler. “Semantic Width and the Fixed-Parameter Tractability of Constraint Satisfaction Problems”. In: *CoRR* abs/2007.14169 (2020).
cs.CC/2007.14169.
Cited on page 164.

[CL10] Thomas Colcombet and Christof Löding. “Regular cost functions over finite trees”. In: *2010 25th Annual IEEE Symposium on Logic in Computer Science*. IEEE, 2010, pp. 70–79.
10.1109/LICS.2010.36.
Cited on pages 24, 25.

[CM77] Ashok K. Chandra and Philip M. Merlin. “Optimal Implementation of Conjunctive Queries in Relational Data Bases”. In: *Proceedings of the 9th Annual ACM Symposium on Theory of Computing, May 4-6, 1977, Boulder, Colorado, USA*. Ed. by John E. Hopcroft, Emily P. Friedman and Michael A. Harrison. ACM, 1977, pp. 77–90.
10.1145/800105.803397.
Cited on pages 3, 8, 16, 122, 127, 156, 166.

[Cod70] E. F. Codd. “A Relational Model of Data for Large Shared Data Banks”. In: *Communications of the ACM* 13.6 (1970), pp. 377–387.
10.1145/362384.362685.
Cited on page 3.

[Cod72] E. F. Codd. “Relational Completeness of Data Base Sublanguages”. In: *Database Systems*. Ed. by R. Rustin. Prentice-Hall, 1972, pp. 33–64.
Cited on pages 3, 75.

[Com+08] Hubert Comon, Max Dauchet, Rémi Gilleron, Florent Jacquemard, Denis Lugiez, Christof Löding, Sophie Tison and Marc Tommasi. *Tree Automata Techniques and Applications*. 2008, p. 262.
https://inria.hal.science/hal-03367725.
Cited on page 125.

[Cos+88] Stavros S. Cosmadakis, Haim Gaifman, Paris C. Kanellakis and Moshe Y. Vardi. "Decidable Optimization Problems for Database Logic Programs (Preliminary Report)". In: *Proceedings of the 20th Annual ACM Symposium on Theory of Computing, May 2-4, 1988, Chicago, Illinois, USA*. 1988, pp. 477–490.
10.1145/62212.62259.
Cited on pages 122, 125, 126.

[CR00] Chandra Chekuri and Anand Rajaraman. "Conjunctive query containment revisited". In: *Theoretical Computer Science* 239.2 (2000), pp. 211–229.
10.1016/s0304-3975(99)00220-0.
Cited on pages 8, 151, 157, 166, 179.

[CR98] Ka Wong Chong and Edgar A. Ramos. "Improved Deterministic Parallel Padded Sorting". In: *Algorithms - ESA '98, 6th Annual European Symposium, Venice, Italy, August 24-26, 1998, Proceedings*. Ed. by Gianfranco Bilardi, Giuseppe F. Italiano, Andrea Pietracaprina and Geppino Pucci. Vol. 1461. Lecture Notes in Computer Science. Springer, 1998, pp. 405–416.
10.1007/3-540-68530-8_34.
Cited on page 76.

[CV97] Surajit Chaudhuri and Moshe Y. Vardi. "On the Equivalence of Recursive and Nonrecursive Datalog Programs". In: *Journal of Computer and System Sciences* 54.1 (1997), pp. 61–78.
10.1006/jcss.1997.1452.
Cited on pages 125, 126.

[CY12] Rada Chirkova and Jun Yang. "Materialized Views". In: *Foundations and Trends® in Databases* 4.4 (2012), pp. 295–405.
10.1561/1900000020.
Cited on pages 7, 189.

[Dat+18] Samir Datta, Raghav Kulkarni, Anish Mukherjee, Thomas Schwentick and Thomas Zeume. "Reachability Is in DynFO". In: *Journal of the ACM* 65.5 (2018), 33:1–33:24.
10.1145/3212685.
Cited on pages 76, 197.

[DS07] Nilesh N. Dalvi and Dan Suciu. "The dichotomy of conjunctive queries on probabilistic structures". In: *Proceedings of the Twenty-Sixth ACM SIGACT-*

SIGMOD-SIGART Symposium on Principles of Database Systems, June 11-13, 2007, Beijing, China. Ed. by Leonid Libkin. ACM, 2007, pp. 293–302.
10.1145/1265530.1265571.
Cited on pages 8, 17, 198.

[DS95] Guozhu Dong and Jianwen Su. "Incremental and Decremental Evaluation of Transitive Closure by First-Order Queries". In: *Information and Computation* 120.1 (1995), pp. 101–106.
10.1006/inco.1995.1102.
Cited on page 76.

[DV91] Karl Denninghoff and Victor Vianu. "The Power of Methods With Parallel Semantics". In: *17th International Conference on Very Large Data Bases, September 3-6, 1991, Barcelona, Catalonia, Spain, Proceedings*. Ed. by Guy M. Lohman, Amílcar Sernadas and Rafael Camps. Morgan Kaufmann, 1991, pp. 221–232.
http://www.vldb.org/conf/1991/P221.PDF.
Cited on page 76.

[Emd90] Peter van Emde Boas. "Machine Models and Simulation". In: *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity*. Ed. by Jan van Leeuwen. Elsevier and MIT Press, 1990, pp. 1–66.
Cited on page 30.

[FO14] Robert Fink and Dan Olteanu. "A dichotomy for non-repeating queries with negation in probabilistic databases". In: *Proceedings of the 33rd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS'14, Snowbird, UT, USA, June 22-27, 2014*. Ed. by Richard Hull and Martin Grohe. ACM, 2014, pp. 144–155.
10.1145/2594538.2594549.
Cited on pages 8, 198.

[FO16] Robert Fink and Dan Olteanu. "Dichotomies for Queries with Negation in Probabilistic Databases". In: *ACM Transactions on Database Systems* 41.1 (2016), 4:1–4:47.
10.1145/2877203.
Cited on page 8.

[Gec+16] Gaetano Geck, Bas Ketsman, Frank Neven and Thomas Schwentick. "Parallel-Correctness and Containment for Conjunctive Queries with Union and Negation". en. In: *International Conference on Database Theory, ICDT 2016*. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik GmbH, Wadern/Saarbruecken, Germany, 2016, 9:1–9:17.
10.4230/LIPIcs.ICDT.2016.9.
Cited on pages 6, 80.

[Gec+19] Gaetano Geck, Bas Ketsman, Frank Neven and Thomas Schwentick. "Parallel-Correctness and Containment for Conjunctive Queries with Union and Negation". In: *ACM Transactions on Computational Logic* 20.3 (2019), 18:1–18:24.
10.1145/3329120.
Cited on page 144.

[Gec+22] Gaetano Geck, Jens Keppeler, Thomas Schwentick and Christopher Spinrath. "Rewriting with Acyclic Queries: Mind Your Head". In: *25th International Conference on Database Theory, ICDT 2022, March 29 to April 1, 2022, Edinburgh, UK (Virtual Conference)*. Ed. by Dan Olteanu and Nils Vortmeier. Vol. 220. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022, 8:1–8:20.
10.4230/LIPIcs.ICDT.2022.8.
Cited on pages 10, 147.

[Gec+23] Gaetano Geck, Jens Keppeler, Thomas Schwentick and Christopher Spinrath. "Rewriting with Acyclic Queries: Mind Your Head". In: *Logical Methods in Computer Science* 19.4 (2023).
10.46298/LMCS-19(4:17)2023.
Cited on pages 10, 147.

[Gec19] Gaetano Geck. "Reasoning about distributed relational data and query evaluation". PhD thesis. Technical University of Dortmund, Germany, 2019.
10.17877/DE290R-21704.
https://hdl.handle.net/2003/39813.
Cited on pages 81, 144.

[GKC06] Gang Gou, Maxim Kormilitsin and Rada Chirkova. "Query evaluation using overlapping views: completeness and efficiency". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 27-29, 2006*. Ed. by Surajit Chaudhuri, Vagelis Hristidis and Neoklis Polyzotis. ACM, 2006, pp. 37–48.
10.1145/1142473.1142479.
Cited on pages 158, 191.

[GLS01] Georg Gottlob, Nicola Leone and Francesco Scarcello. "The complexity of acyclic conjunctive queries". In: *Journal of the ACM* 48.3 (2001), pp. 431–498.
10.1145/382780.382783.
Cited on page 8.

[GLS02] Georg Gottlob, Nicola Leone and Francesco Scarcello. "Hypertree Decompositions and Tractable Queries". In: *Journal of Computer and System Sciences* 64.3 (2002), pp. 579–627.
10.1006/jcss.2001.1809.
Cited on pages 19, 63.

[GMT13] Sergio Greco, Cristian Molinaro and Irina Trubitsyna. “Logic programming with function symbols: Checking termination of bottom-up evaluation through program adornments”. In: *Theory and Practice of Logic Programming* 13.4-5 (July 2013), pp. 737–752.
10.1017/S147106841300046X.
Cited on page 117.

[GMV93] Michael T. Goodrich, Yossi Matias and Uzi Vishkin. “Approximate Parallel Prefix Computation and its Applications”. In: *The Seventh International Parallel Processing Symposium, Proceedings, Newport Beach, California, USA, April 13-16, 1993*. IEEE Computer Society, 1993, pp. 318–325.
10.1109/IPPS.1993.262899.
Cited on pages 221, 222.

[GMV94] Michael T. Goodrich, Yossi Matias and Uzi Vishkin. “Optimal Parallel Approximation for Prefix Sums and Integer Sorting”. In: *Proceedings of the Fifth Annual ACM-SIAM Symposium on Discrete Algorithms. 23-25 January 1994, Arlington, Virginia, USA*. Ed. by Daniel Dominic Sleator. SODA ’94. Society for Industrial and Applied Mathematics, 1994, pp. 241–250. ISBN: 0898713293.
Cited on pages 221, 222.

[GNS20] Gaetano Geck, Frank Neven and Thomas Schwentick. “Distribution Constraints: The Chase for Distributed Data”. In: *23rd International Conference on Database Theory, ICDT 2020, March 30-April 2, 2020, Copenhagen, Denmark*. Ed. by Carsten Lutz and Jean Christoph Jung. Vol. 155. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020, 13:1–13:19.
10.4230/LIPIcs.ICDT.2020.13.
Cited on pages 86, 146.

[Got+16] Georg Gottlob, Gianluigi Greco, Nicola Leone and Francesco Scarcello. “Hypertree Decompositions: Questions and Answers”. In: *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2016, San Francisco, CA, USA, June 26 - July 01, 2016*. Ed. by Tova Milo and Wang-Chiew Tan. ACM, 2016, pp. 57–74.
10.1145/2902251.2902309.
Cited on page 19.

[GST90] Sumit Ganguly, Avi Silberschatz and Shalom Tsur. “A Framework for the Parallel Processing of Datalog Queries”. In: *Proceedings of the 1990 ACM SIGMOD international conference on Management of data*. ACM, 1990, pp. 143–152.
10.1145/93597.98724.
Cited on page 143.

[GZ95] Tal Goldberg and Uri Zwick. “Optimal deterministic approximate parallel prefix sums and their applications”. In: *Third Israel Symposium on Theory*

of Computing and Systems, ISTCS 1995, Tel Aviv, Israel, January 4-6, 1995, Proceedings. IEEE Computer Society, 1995, pp. 220–228.
10.1109/ISTCS.1995.377028.
Cited on pages 4, 28, 32, 33, 47, 219, 221.

[Hag92a] Torben Hagerup. "On a Compaction Theorem of Ragde". In: *Information Processing Letters* 43.6 (1992), pp. 335–340.
10.1016/0020-0190(92)90121-B.
Cited on page 75.

[Hag92b] Torben Hagerup. "The Log-Star Revolution". In: *STACS 92, 9th Annual Symposium on Theoretical Aspects of Computer Science, Cachan, France, February 13-15, 1992, Proceedings*. Ed. by Alain Finkel and Matthias Jantzen. Vol. 577. Lecture Notes in Computer Science. Springer, 1992, pp. 259–278.
10.1007/3-540-55210-3_189.
Cited on page 56.

[Hal01] Alon Y. Halevy. "Answering queries using views: A survey". In: *The International Journal on Very Large Data Bases* 10.4 (2001), pp. 270–294.
10.1007/s007780100054.
Cited on pages 7, 189.

[HR92] Torben Hagerup and Rajeev Raman. "Waste Makes Haste: Tight Bounds for Loose Parallel Sorting". In: *33rd Annual Symposium on Foundations of Computer Science, Pittsburgh, Pennsylvania, USA, 24-27 October 1992*. IEEE Computer Society, 1992, pp. 628–637.
10.1109/SFCS.1992.267788.
Cited on page 76.

[HY19] Xiao Hu and Ke Yi. "Instance and Output Optimal Parallel Algorithms for Acyclic Joins". In: *Proceedings of the 38th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*. Ed. by Dan Suciu, Sebastian Skritek and Christoph Koch. ACM, 2019, pp. 450–463.
10.1145/3294052.3319698.
Cited on page 17.

[Imm89] Neil Immerman. "Expressibility and Parallel Complexity". In: *SIAM Journal on Computing* 18.3 (1989), pp. 625–638.
10.1137/0218043.
Cited on pages 3, 75, 76.

[Imm99] Neil Immerman. *Descriptive Complexity*. Graduate texts in computer science. Springer, 1999. ISBN: 978-1-4612-6809-3.
10.1007/978-1-4612-0539-5.
Cited on pages 3, 75, 76, 220.

[IUV17] Muhammad Idris, Martín Ugarte and Stijn Vansummeren. "The Dynamic Yannakakis Algorithm: Compact and Efficient Query Processing Under Updates". In: *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*. Ed. by Semih Salihoglu, Wenchao Zhou, Rada Chirkova, Jun Yang and Dan Suciu. ACM, 2017, pp. 1259–1274.
10.1145/3035918.3064027.
Cited on page 17.

[JáJ92] Joseph F. JáJá. *An Introduction to Parallel Algorithms*. Addison-Wesley, 1992. ISBN: 0-201-54856-9.
Cited on pages 3, 30, 31, 221.

[KAK20] Bas Ketsman, Aws Albarghouthi and Paraschos Koutris. "Distribution Policies for Datalog". In: *Theory of Computing Systems* 64.5 (2020), pp. 965–998.
10.1007/s00224-019-09959-3.
Cited on pages 6, 80, 82, 85, 92, 98, 132, 144, 145.

[Kar+20] Ahmet Kara, Milos Nikolic, Dan Olteanu and Haozhe Zhang. "Trade-offs in Static and Dynamic Evaluation of Hierarchical Queries". In: *Proceedings of the 39th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2020, Portland, OR, USA, June 14-19, 2020*. Ed. by Dan Suciu, Yufei Tao and Zhewei Wei. ACM, 2020, pp. 375–392.
10.1145/3375395.3387646.
Cited on pages 8, 17.

[Kar72] Richard M. Karp. "Reducibility Among Combinatorial Problems". In: *Proceedings of a symposium on the Complexity of Computer Computations, held March 20-22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York, USA*. Ed. by Raymond E. Miller and James W. Thatcher. The IBM Research Symposia Series. Plenum Press, New York, 1972, pp. 85–103.
10.1007/978-1-4684-2001-2_9.
Cited on page 172.

[Kep20] Jens Keppeler. "Answering Conjunctive Queries and FO+MOD Queries under Updates". PhD thesis. Humboldt University of Berlin, Germany, 2020.
10.18452/21483.
http://edoc.hu-berlin.de/18452/22264.
Cited on page 8.

[KNV18] Bas Ketsman, Frank Neven and Brecht Vandevoort. "Parallel-Correctness and Transferability for Conjunctive Queries under Bag Semantics". In: *21st International Conference on Database Theory, ICDT 2018, March 26-29, 2018, Vienna, Austria*. Ed. by Benny Kimelfeld and Yael Amsterdamer.

Vol. 98. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik GmbH, Wadern/Saarbruecken, Germany, 2018, 18:1–18:16.
10.4230/LIPIcs.ICDT.2018.18.
Cited on pages 6, 144.

[KS11] Paraschos Koutris and Dan Suciu. "Parallel evaluation of conjunctive queries". In: *Proceedings of the 30th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2011, June 12-16, 2011, Athens, Greece*. Ed. by Maurizio Lenzerini and Thomas Schwentick. ACM, 2011, pp. 223–234.
10.1145/1989284.1989310.
Cited on page 8.

[KS17] Bas Ketsman and Dan Suciu. "A Worst-Case Optimal Multi-Round Algorithm for Parallel Computation of Conjunctive Queries". In: *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. ACM, 2017, pp. 417–428.
10.1145/3034786.3034788.
Cited on page 143.

[KSS23] Jens Keppeler, Thomas Schwentick and Christopher Spinrath. "Work-Efficient Query Evaluation with PRAMs". In: *26th International Conference on Database Theory, ICDT 2023, March 28-31, 2023, Ioannina, Greece*. Ed. by Floris Geerts and Brecht Vandevoort. Vol. 255. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023, 16:1–16:20.
10.4230/LIPIcs.ICDT.2023.16.
Cited on pages 9, 27, 75.

[Lei+05] Dirk Leinders, Maarten Marx, Jerzy Tyszkiewicz and Jan Van den Bussche. "The Semijoin Algebra and the Guarded Fragment". In: *Journal of Logic, Language and Information* 14.3 (2005), pp. 331–343.
10.1007/s10849-005-5789-8.
Cited on pages 60, 75.

[Lev+95] Alon Y. Levy, Alberto O. Mendelzon, Yehoshua Sagiv and Divesh Srivastava. "Answering Queries Using Views". In: *Proceedings of the Fourteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 22-25, 1995, San Jose, California, USA*. Ed. by Mihalis Yannakakis and Serge Abiteboul. ACM Press, 1995, pp. 95–104.
10.1145/212433.220198.
Cited on pages 149–151, 155.

[LV07] Dirk Leinders and Jan Van den Bussche. "On the complexity of division and set joins in the relational algebra". In: *Journal of Computer and System Sciences* 73.4 (2007), pp. 538–549.
10.1016/j.jcss.2006.10.011.
Cited on page 60.

[Min61] Marvin L. Minsky. “Recursive Unsolvability of Post’s Problem of ”Tag” and other Topics in Theory of Turing Machines”. In: *Annals of Mathematics* 74.3 (1961), pp. 437–455. ISSN: 0003-486X.
10.2307/1970290.
Cited on page 23.

[Mof+15] Vera Zaychik Moffitt, Julia Stoyanovich, Serge Abiteboul and Gerome Miklau. “Collaborative Access Control in WebdamLog”. In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*. Ed. by Timos K. Sellis, Susan B. Davidson and Zachary G. Ives. ACM, 2015, pp. 197–211.
10.1145/2723372.2749433.
Cited on page 146.

[Nev+19] Frank Neven, Thomas Schwentick, Christopher Spinrath and Brecht Vandevoort. “Parallel-Correctness and Parallel-Boundedness for Datalog Programs”. In: *22nd International Conference on Database Theory, ICDT 2019, March 26-28, 2019, Lisbon, Portugal*. Ed. by Pablo Barceló and Marco Calautti. Vol. 127. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019, 14:1–14:19.
10.4230/LIPIcs.ICDT.2019.14.
Cited on pages 10, 79, 145.

[Ngo+18] Hung Q. Ngo, Ely Porat, Christopher Ré and Atri Rudra. “Worst-case Optimal Join Algorithms”. In: *Journal of the ACM* 65.3 (2018), 16:1–16:40.
10.1145/3180143.
Cited on page 65.

[NSV10] Alan Nash, Luc Segoufin and Victor Vianu. “Views and queries: Determinacy and rewriting”. In: *ACM Transactions on Database Systems* 35.3 (2010), 21:1–21:41.
10.1145/1806907.1806913.
Cited on pages 9, 151, 190.

[OZ15] Dan Olteanu and Jakub Závodný. “Size Bounds for Factorised Representations of Query Results”. In: *ACM Transactions on Database Systems* 40.1 (2015), 2:1–2:44.
10.1145/2656335.
Cited on page 197.

[PH01] Rachel Pottinger and Alon Y. Halevy. “MiniCon: A scalable algorithm for answering queries using views”. In: *The International Journal on Very Large Data Bases* 10.2-3 (2001), pp. 182–198.
10.1007/s007780100048.
Cited on pages 149, 158, 191.

[PI97] Sushant Patnaik and Neil Immerman. “Dyn-FO: A Parallel, Dynamic Complexity Class”. In: *Journal of Computer and System Sciences* 55.2 (1997), pp. 199–209.
10.1006/jcss.1997.1520.
Cited on page 76.

[Rag92] Prabhakar Ragde. “Processor-Time Tradeoffs in PRAM Simulations”. In: *Journal of Computer and System Sciences* 44.1 (1992), pp. 103–113.
10.1016/0022-0000(92)90006-5.
Cited on page 29.

[Ros08] Benjamin Rossman. “On the constant-depth complexity of k-clique”. In: *Proceedings of the 40th Annual ACM Symposium on Theory of Computing, Victoria, British Columbia, Canada, May 17-20, 2008*. 2008, pp. 721–730.
10.1145/1374376.1374480.
Cited on page 59.

[Sap15] Mark V. Sapir. “Minsky Machines and Algorithmic Problems”. In: *Fields of Logic and Computation II - Essays Dedicated to Yuri Gurevich on the Occasion of His 75th Birthday*. Ed. by Lev D. Beklemishev, Andreas Blass, Nachum Dershowitz, Bernd Finkbeiner and Wolfram Schulte. Vol. 9300. Lecture Notes in Computer Science. Springer, 2015, pp. 273–292.
10.1007/978-3-319-23534-9_17.
Cited on page 23.

[Sch+21] Jonas Schmidt, Thomas Schwentick, Till Tantau, Nils Vortmeier and Thomas Zeume. “Work-sensitive Dynamic Complexity of Formal Languages”. In: *Foundations of Software Science and Computation Structures - 24th International Conference, FOSSACS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings*. Ed. by Stefan Kiefer and Christine Tasson. Vol. 12650. Lecture Notes in Computer Science. Springer, 2021, pp. 490–509.
10.1007/978-3-030-71995-1_25.
Cited on page 76.

[Shm93] Oded Shmueli. “Equivalence of DATALOG Queries is Undecidable”. In: *Journal of Logic Programming* 15.3 (1993), pp. 231–241.
10.1016/0743-1066(93)90040-N.
Cited on page 144.

[SKN21] Bruhathi Sundarmurthy, Paraschos Koutris and Jeffrey F. Naughton. “Locality-Aware Distribution Schemes”. In: *24th International Conference on Database Theory, ICDT 2021, March 23-26, 2021, Nicosia, Cyprus*. Ed. by Ke Yi and Zhewei Wei. Vol. 186. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, 22:1–22:25.
10.4230/LIPIcs.ICDT.2021.22.
Cited on page 144.

[Spi+22] Christopher Spinrath, Gaetano Geck, Jens Keppeler and Thomas Schwentick. *Rewriting with Acyclic Queries: Mind your Head*. International Conference on Database Theory (ICDT). Recorded video presentation. 2022.
10.5446/57486.
Cited on pages 10, 147.

[SS23] Jonas Schmidt and Thomas Schwentick. "Dynamic Constant Time Parallel Graph Algorithms with Sub-Linear Work". In: *48th International Symposium on Mathematical Foundations of Computer Science, MFCS 2023, August 28 to September 1, 2023, Bordeaux, France*. Ed. by Jérôme Leroux, Sylvain Lombardy and David Peleg. Vol. 272. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023, 80:1–80:15.
10.4230/LIPICS.MFCS.2023.80.
Cited on page 76.

[SST23] Jonas Schmidt, Thomas Schwentick and Jennifer Todtenhoefer. "On the Work of Dynamic Constant-Time Parallel Algorithms for Regular Tree Languages and Context-Free Languages". In: *48th International Symposium on Mathematical Foundations of Computer Science, MFCS 2023, August 28 to September 1, 2023, Bordeaux, France*. Ed. by Jérôme Leroux, Sylvain Lombardy and David Peleg. Vol. 272. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023, 81:1–81:15.
10.4230/LIPICS.MFCS.2023.81.
Cited on page 76.

[SVZ20] Thomas Schwentick, Nils Vortmeier and Thomas Zeume. "Sketches of Dynamic Complexity". In: *ACM SIGMOD Record* 49.2 (2020), pp. 18–29.
10.1145/3442322.3442325.
Cited on pages 76, 197.

[SY80] Yehoshua Sagiv and Mihalis Yannakakis. "Equivalences Among Relational Expressions with the Union and Difference Operators". In: *Journal of the ACM* 27.4 (1980), pp. 633–655.
10.1145/322217.322221.
Cited on page 122.

[Vel14] Todd L. Veldhuizen. "Triejoin: A Simple, Worst-Case Optimal Join Algorithm". In: *Proc. 17th International Conference on Database Theory (ICDT), Athens, Greece, March 24-28, 2014*. Ed. by Nicole Schweikardt, Vassilis Christophides and Vincent Leroy. OpenProceedings.org, 2014, pp. 96–106.
10.5441/002/icdt.2014.13.
Cited on page 69.

[Vol99] Heribert Vollmer. *Introduction to Circuit Complexity - A Uniform Approach*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 1999. ISBN: 978-3-540-64310-4.
10.1007/978-3-662-03927-4.
Cited on page 221.

[WY22] Yilei Wang and Ke Yi. "Query Evaluation by Circuits". In: *PODS '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*. Ed. by Leonid Libkin and Pablo Barceló. ACM, 2022, pp. 67–78.
10.1145/3517804.3524142.
Cited on page 75.

[Xin+13] Reynold S. Xin, Josh Rosen, Matei Zaharia, Michael J. Franklin, Scott Shenker and Ion Stoica. "Shark: SQL and Rich Analytics at Scale". In: *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. ACM, 2013, pp. 13–24.
10.1145/2463676.2465288.
Cited on page 5.

[Yan81] Mihalis Yannakakis. "Algorithms for Acyclic Database Schemes". In: *Very Large Data Bases, 7th International Conference, September 9-11, 1981, Cannes, France, Proceedings*. IEEE Computer Society, 1981, pp. 82–94.
Cited on pages 8, 61, 63, 156, 165.

Index of Definitions

Appendix A

Revisiting Consistent Approximate Prefix Sums

The purpose of this section is to revisit the proof (chain) for Proposition 3.1.8 by Goldberg and Zwick [GZ95] and analyse the space requirements of the algorithm.

Since we consider only consistent λ-approximate prefix sums we will omit the term "consistent" from here on out. We will also refrain from specifying the accuracy function λ if it is clear from the context or not important.

To prove Proposition 3.1.8, Goldberg and Zwick proceed in three major steps. Based on an algorithm for *approximate counting* by Ajtai [Ajt93], they first show that the sum of n integers can be approximated in constant time with polynomial work and space. In the second step this is used to compute approximate prefix sums with polynomial work and space. The third step is then to reduce the required work and space to $\mathcal{O}(n^{1+\varepsilon})$. We will proceed analogously.

For an array $\mathbf{A}$ whose cells contain values from $\{0, 1\}$, we write $\#_1(\mathbf{A})$ for the number of cells of $\mathbf{A}$ containing 1. The required result on approximate counting is the following.

Proposition A.1 [Ajt93, Theorem 2.1]**.** *For every integer $a > 0$ there is a $\mathcal{O}(1)$-time parallel algorithm that, given an array $\mathbf{A}$ whose cells contain either 1 or 0, computes a number k satisfying $\#_1(\mathbf{A}) \leq k \leq (1 + \lambda(|\mathbf{A}|))\#_1(\mathbf{A})$ where $\lambda(n) = (\log n)^{-a}$. The algorithm requires polynomial work and space on a common CRCW PRAM.*

Let us point out that Proposition A.1 is phrased with our application in mind. Ajtai actually stated it in terms of first-order formulas over the signature $(+, \times, \leq)$ where $+, \times, \leq$ are binary relation symbols with the usual intended meaning. For every n let $\mathcal{M}_n$ be the structure over $(+, \times, \leq)$ with universe $M_n = [0, n-1]$ and the usual interpretation of the arithmetic relations $+$,$\times$, and $\leq$. Ajtai proved the following.

For every $a' > 0$ there is a formula $\varphi(x, Y)$ over $(+, \times, \leq)$ with a free variable x and a free unary relation variable Y, such that for every integer $n > 0$, $m \in M_n$, and $A \subseteq M_n$ the following holds.

(a) If $|A| \leq (1 - \lambda'(n))m$ then $\mathcal{M}_n \not\models \varphi(m, A)$; and

(b) if $|A| \geq (1 + \lambda'(n))m$ then $\mathcal{M}_n \models \varphi(m, A)$.

Here $\lambda'(n)$ is of the same form as $\lambda(n)$ in Proposition A.1, that is, $\lambda'(n) = (\log n)^{-a'}$.

We will briefly discuss how this implies Proposition A.1 as stated above. Let $\mathbf{A}$ be an array with entries from $\{0,1\}$. We fix $n = |\mathbf{A}|$ and the interpretation

$$A = \{i \in M_n \mid \mathbf{A}[i+1] = 1\}$$

for Y. Further, we assume that $n > 2$. Otherwise, it is trivial to determine $\#_1(\mathbf{A})$ exactly.

It is well-known that first-order queries can be computed by a PRAM in constant time with a polynomial number of processors and space [cf., e.g., Imm99, Lemma 5.12, the space bound is stated in the proof]. Thus, it is possible to test, for all $0 \leq m \leq n-1$ in parallel, whether $\mathcal{M}_n \models \varphi(m, A)$ holds, and write the results into an array $\mathbf{C}$ of size n.

Observe that $\mathcal{M}_n \models \varphi(0, A)$ holds and let m_1 be the largest number from M_n such that $\mathcal{M}_n \models \varphi(m, A)$ holds for all $m \leq m_1$. Given the array $\mathbf{C}$, the number m_1 can easily be determined with quadratic work and linear space.

We claim that $k = (1+\lambda'(n))(m_1+1)$ satisfies $|A| \leq k \leq (1+3\lambda(n))|A|$, where $\lambda(n)$ is as in Proposition A.1. We note that the factor 3 in the term $(1+3\lambda(n))|A|$ is of *no* consequence for our purposes, since we can always choose $\lambda(n) = (\log n)^{-a}$ for a sufficiently large a. We first show that $|A| \leq k$ holds. If $m_1+1 = n$ then $|A| \leq n \leq k$ holds trivially. Otherwise, $m_1+1 \leq n-1$ and we have $\mathcal{M}_n \not\models \varphi(m_1+1, A)$ due to the choice of m_1. Thus, by contraposition of Statement (b) we know that $|A| < (1+\lambda'(n))(m_1+1) = k$.

Towards $k \leq (1+\lambda(n))|A|$, note that the contraposition of Statement (a) yields $|A| > (1-\lambda'(n))m_1$ since $\mathcal{M}_n \models \varphi(m_1, A)$ holds. Because $1-\lambda'(n) < 1$, this implies $|A| \geq (1-\lambda'(n))(m_1+1)$. Therefore, we can conclude

$$k = (1+\lambda'(n))(m_1+1) = \frac{(1+\lambda'(n))(1-\lambda'(n))(m_1+1)}{1-\lambda'(n)} \leq \frac{1+\lambda'(n)}{1-\lambda'(n)}|A|.$$

It remains to show that we can choose a' such that $\frac{1+\lambda'(n)}{1-\lambda'(n)} \leq 1+\lambda(n)$ holds (for all $n > 2$). We choose a' such that $\lambda'(n) \leq \frac{\lambda(n)}{2}$ holds for all $n > 2$. Note that this can be done by choosing a sufficiently large a'. Further note that $\lambda'(n) \leq \frac{1}{2}$ because $\lambda(n) \leq 1$ for all $n > 0$. We claim that then $\frac{1}{1-\lambda'(n)} \leq 1+\lambda(n)$ holds for every $n > 0$. Indeed, we have

$$\begin{aligned} &\frac{1}{1-\lambda'(n)} \leq 1+\lambda(n) \\ \Leftrightarrow\ &1 \leq (1+\lambda(n))(1-\lambda'(n)) \\ \Leftrightarrow\ &1 \leq 1+\lambda(n)-\lambda'(n)-\lambda(n)\lambda'(n) \end{aligned}$$

where the last inequality holds because $\lambda'(n) \leq \frac{\lambda(n)}{2}$ and $\lambda(n) \leq 1$. Overall, we thus have

$$k \leq (1+\lambda'(n))(1+\lambda(n))|A| \leq (1+3\lambda(n))|A|$$

where the last inequality holds because $\lambda'(n) \leq \lambda(n) \leq 1$.

This concludes the proof sketch for Proposition A.1.

We now turn towards the first step of the proof for Proposition 3.1.8. It is captured by the following result.

Proposition A.2 [GZ95, Theorem 3.1]**.** *For every fixed integer $a > 0$, there is a $\mathcal{O}(1)$-time parallel algorithm that, given an array $\mathbf{A}$ containing a sequence $a_1, \ldots, a_{|\mathbf{A}|}$ of natural numbers of size at most $\mathcal{O}(\log |\mathbf{A}|)$, computes an integer s satisfying*

$$\sum_{i=1}^{|\mathbf{A}|} a_i \leq s \leq (1 + \lambda(|\mathbf{A}|)) \sum_{i=1}^{|\mathbf{A}|} a_i$$

where $\lambda(n) = (\log n)^{-a}$. The algorithm requires polynomial work and space on a common CRCW PRAM.

For a number a_i, we denote by $a_{i,j}$ the bit at position j of the binary representation of a_i. The idea for Proposition A.2 is to approximate, for each position j, the number of a_i with $a_{i,j} = 1$. That is, for each j, the algorithm computes an approximation of $k_j = \sum_{i=1}^{|\mathbf{A}|} a_{i,j}$ using Proposition A.1. For this purpose, an algorithm can write the binary representations of the numbers $a_1, \ldots, a_n$ into a two-dimensional array of size $\mathcal{O}((\log n) \cdot n)$ where $n = |\mathbf{A}|$, filling up empty cells with 0. Assuming the numbers are written row-wise into the array, Proposition A.1 can be applied to each column in parallel. Overall this requires polynomial work and space.

The resulting numbers $k'_0, \ldots, k'_m$ with $m \in \mathcal{O}(\log n)$ satisfy $k_j \leq k'_j \leq (1 + \lambda(n))k_j$ and have size $\mathcal{O}(\log n)$. Note that $\sum_{j=1}^{n} k_j \cdot 2^j = \sum_{i=1}^{n} a_i$, and, consequently, for $s = \sum_{j=1}^{n} k'_j \cdot 2^j$ we have that $\sum_{i=1}^{n} a_i \leq s \leq (1+\lambda(n)) \sum_{i=1}^{n} a_i$. Since there are only $\mathcal{O}(\log n)$ many numbers k'_j their exact sum s can be computed with polynomial work and space on a common CRCW PRAM in constant time.[1]

The next step is to show that approximate prefix sums can be computed in constant time (with polynomial work and space). Note that this mainly involves establishing consistency.

Proposition A.3 [GZ95, Theorem 3.2]**.** *For every fixed $a > 0$, there is a $\mathcal{O}(1)$-time parallel algorithm that, given an array $\mathbf{A}$ containing a sequence $a_1, \ldots, a_{|\mathbf{A}|}$ of natural numbers of size at most $\mathcal{O}(\log |\mathbf{A}|)$, computes consistent λ-approximate prefix sums of $a_1, \ldots, a_{|\mathbf{A}|}$ where $\lambda(n) = (\log n)^{-a}$. The algorithm requires polynomial work and space on a common CRCW PRAM.*

The algorithm for Proposition A.3 uses *approximate summation trees* which were originally introduced by Goodrich et al. [GMV94; GMV93] for computing prefix sums using randomization.

Definition A.4 [GMV94; GMV93, Sections 2 and 3, respectively]**.** Let λ be an accuracy function. A *λ-approximate summation tree* T for a sequence $a_1, \ldots, a_n$ of integers is a complete, ordered binary tree with n leaves where every node v is labelled with an integer $\tilde{s}(v)$ such that

[1] This has been proved for polynomial-size, constant-depth circuits by, e.g., Vollmer [Vol99, Theorem 1.21] and these circuits can be simulated by a common CRCW PRAM in constant time with polynomial work and space [see, e.g., JáJ92, Theorem 10.9, the space bound is implicit in the proof].

- the i-th leaf is labelled with a_i, for all $1 \leq i \leq n$; and
- $s(v) \leq \tilde{s}(v) \leq (1 + \lambda(n))s(v)$ holds for each node v. Here $s(v)$ is the sum $\sum_{k=i}^{j} a_k$ where $a_i, \ldots, a_j$ are the leaf labels of the subtree rooted at v.

A λ-approximate summation tree is *consistent* if $\tilde{s}(v) \geq \tilde{s}(v_\ell) + \tilde{s}(v_r)$ where v_ℓ and v_r denote the children of v holds for all inner nodes v.

To compute λ-approximate prefix sums the algorithm first constructs a consistent λ-approximate summation tree and then derives the prefix sums from it.

Indeed, given a λ-approximate summation tree T for a sequence $a_1, \ldots, a_n$, consistent prefix sums can be derived as follows. For each i let V_i be the set of nodes in T that are left siblings of nodes on the path from the leaf for a_i to the root. That is, a node v is in V_i if and only if

(1) v is the left child of a node w;

(2) v does not occur on the path from the node for a_i to the root; but

(3) w does occur on the path from the node for a_i to the root.

Observe that the subtrees rooted at the nodes in V_i induce a partition of $a_1, \ldots, a_{i-1}$. Thus, for $b_i = \sum_{v \in V_i} \tilde{s}(v) + a_i$ we have

$$\sum_{j=1}^{i} a_j \leq b_i \leq (1 + \lambda(n)) \sum_{j=1}^{i} a_j.$$

Since V_i contains at most $\log n$ nodes, the exact sum b_i can be computed with polynomial work and space, cf. the proof of Proposition A.2. For a proof that the sequence $b_1, \ldots, b_n$ is consistent, we refer to [GMV94, Lemma 3.2].

It remains to revisit how a PRAM can construct a consistent λ-approximate summation tree. Thanks to Proposition A.2, for any integer $b > 0$, a λ'-approximate summation tree T' where $\lambda' = (\log n)^{-b}$ can be constructed with polynomial work and space, by computing the labels for each node in parallel. To obtain a consistent λ-approximate summation tree T the label of every inner node of T' is multiplied with $(1 + \lambda'(n))^h$ where h is the height of the node. More precisely, $\tilde{s}(v) = (1 + \lambda'(n))^h \tilde{s}(v')$ where v' is the corresponding node of v in T' and h is the height of v in T (or, equivalently, the height of v' in T').

We follow the proof given by Goodrich et al. [GMV93, Lemma 2.2][2] to show that T is a consistent λ-approximate summation tree. We start by proving that T is consistent.

Let v be an inner node of T, h its height, v_ℓ, v_r its children, and v', v'_ℓ, v'_r the corresponding nodes in T'. As in Definition A.4 we denote by $s(w)$ the exact sum of all leaf

[2] We note that the proof by Goodrich et al. [GMV93, Lemma 2.2] uses a slightly different factor since their summation trees assert $(1 - \lambda(n))s(v) \leq \tilde{s}(v)$ instead of $s(v) \leq \tilde{s}(v)$.

labels of the subtree rooted at w. We have that

$$\begin{aligned}\tilde{s}(v_\ell) + \tilde{s}(v_r) &= (1+\lambda'(n))^{h-1}\tilde{s}(v'_\ell) + (1+\lambda'(n))^{h-1}\tilde{s}(v'_r)\\ &\leq (1+\lambda'(n))^{h}s(v'_\ell) + (1+\lambda'(n))^{h}s(v'_r)\\ &= (1+\lambda'(n))^{h}s(v')\\ &\leq (1+\lambda'(n))^{h}\tilde{s}(v')\\ &= \tilde{s}(v)\end{aligned}$$

where the inequalities hold because T' is a λ'-approximate summation tree. Thus, T is consistent.

We now argue that T is a λ-approximate summation tree. Indeed, for any inner node v of T we have $s(v) \leq \tilde{s}(v) \leq (1+\lambda'(n))^{\log(n)+1}s(v)$ since T' is an approximate summation tree (and all nodes have height at most $\log n$). The claim then follows by the following observation, since for "small" n the algorithm can just compute the exact prefix sums.

Observation A.5. For all integers $a > 1$ there is an integer $b > 1$ such that for sufficiently large n the inequality

$$(1+\lambda'(n))^{\log n} \leq (1 + 2(\log n)\lambda'(n)) \leq 1 + \lambda(n)$$

holds where $\lambda(n) = (\log n)^{-a}$ and $\lambda'(n) = (\log n)^{-b}$.

The last step to prove Proposition 3.1.8 is to show that the work and space bounds can be improved to $\mathcal{O}(n^{1+\varepsilon})$.

Proof of Proposition 3.1.8. Due to Proposition A.3 there is a $\mathcal{O}(1)$-time parallel algorithm that computes approximate prefix sums with polynomial work and space. Let $n = |\mathbf{A}|$ and c be a constant such that this algorithm requires work and space $\mathcal{O}(n^c)$. Furthermore, let $\varepsilon > 0$ be arbitrary but fixed, and define $\delta = \frac{\varepsilon}{c}$.

Let $\lambda'(n)$ be an accuracy function of the form $\lambda'(n) = (\log n)^{-b}$ such that $(1+\lambda'(n))^2 \leq (1+\lambda(n))$ holds for all $n > 2$. Note that this can be done by choosing a large enough b which only depends on a.

The algorithm operates as follows in three steps. It first divides the given array $\mathbf{A}$ into $m = n^{1-\delta}$ subarrays $\mathbf{A}_1, \ldots, \mathbf{A}_m$ of length at most n^δ. Then, for each subarray $\mathbf{A}_k$ in parallel, arrays $\mathbf{B}_k$ containing consistent λ'-approximate prefix sums for $\mathbf{A}_k$ are computed using the algorithm guaranteed by Proposition 3.1.8. This step requires work and space $\mathcal{O}(n^{1+\varepsilon})$ because

$$\left(n^\delta\right)^c \cdot n^{1-\delta} = n^{\frac{\varepsilon}{c}c} \cdot n^{1-\frac{\varepsilon}{c}} = n^{1+\varepsilon-\frac{\varepsilon}{c}} \leq n^{1+\varepsilon}.$$

In the second step, the algorithm initializes an array $\mathbf{C}$ of length $n^{1-\delta}$ by setting $\mathbf{C}[1] = 0$ and, for every index $k > 1$, $\mathbf{C}[k]$ to the largest prefix sum computed for $\mathbf{A}_{k-1}$, i.e. $\mathbf{B}_{k-1}[n^\delta]$. It then invokes itself recursively to compute an array $\mathbf{D}$ containing consistent λ'-approximate prefix sums for $\mathbf{C}$.

Finally, in the third step, the prefix sums for the $\mathbf{A}_k$ and $\mathbf{C}$ are combined to prefix sums for $\mathbf{A}$. For an index $i \in [1, n]$ let $k_i \in [1, m]$ and $i' \in [1, n^\delta]$ be such that $n^\delta \cdot (k_i - 1) + i' = i$. That is, the i-th cell of $\mathbf{A}$ is the i'-th cell of subarray $\mathbf{A}_{k_i}$. Let $\mathbf{B}$ be the array of length n with $\mathbf{B}[i] = \mathbf{B}_{k_i}[i] + \mathbf{D}[k_i]$. The algorithm returns the array $\mathbf{B}$.

The last step can easily be performed with linear work and space. The recursive call in the second step thus requires $\mathcal{O}(n^{1+\varepsilon})$ work and space as well. Note that the recursion depth is at most $\log_{n^\delta}(n) \in \mathcal{O}(\frac{1}{\delta})$. Therefore, the algorithm runs in constant time and requires work and space $\mathcal{O}(n^{1+\varepsilon})$ as claimed.

In the following we prove that the algorithm is correct. That is, we prove that $\mathbf{B}$ contains consistent λ-approximate prefix sums for $\mathbf{A}$.

Let $i \in [1, n]$ and let i' and k_i be as before. We first show that $\mathbf{B}$ contains (not necessarily consistent) approximate prefix sums for $\mathbf{A}$. Indeed, we have

$$\begin{aligned}
\sum_{j=1}^{i} \mathbf{A}[j] &= \sum_{\ell=1}^{k_i-1} \sum_{j=1}^{n^\delta} \mathbf{A}_\ell[j] + \sum_{j=1}^{i'} \mathbf{A}_{k_i}[j] \\
&\leq \sum_{\ell=1}^{k_i-1} \mathbf{B}_\ell[n^\delta] + \mathbf{B}_{k_i}[i'] \\
&= \sum_{\ell=1}^{k_i-1} \mathbf{C}[\ell+1] + \mathbf{B}_{k_i}[i'] \\
&= \sum_{\ell=1}^{k_i} \mathbf{C}[\ell] + \mathbf{B}_{k_i}[i'] \\
&\leq \mathbf{D}[k_i] + \mathbf{B}_{k_i}[i'] \\
&= \mathbf{B}[i]
\end{aligned}$$

where the first inequality holds because the $\mathbf{B}_\ell$ contain prefix sums for the $\mathbf{A}_\ell$, the second and third equalities hold by definition of $\mathbf{C}$ (in particular, $\mathbf{C}[1] = 0$), and the last inequality holds because $\mathbf{D}$ contains prefix sums for $\mathbf{C}$.

Conversely, we have

$$\begin{aligned}
\mathbf{B}[i] &= \mathbf{D}[k_i] + \mathbf{B}_{k_i}[i'] \\
&\leq (1 + \lambda'(n)) \sum_{\ell=1}^{k_i} \mathbf{C}[\ell] + \mathbf{B}_{k_i}[i'] \\
&= (1 + \lambda'(n)) \sum_{\ell=1}^{k_i-1} \mathbf{B}_\ell[n^\delta] + \mathbf{B}_{k_i}[i'] \\
&\leq (1 + \lambda'(n)) \sum_{\ell=1}^{k_i-1} (1 + \lambda'(n)) \sum_{j=1}^{n^\delta} \mathbf{A}_\ell[j] + (1 + \lambda'(n)) \sum_{j=1}^{i'} \mathbf{A}_{k_i}[j] \\
&\leq (1 + \lambda'(n))^2 \Big[\sum_{\ell=1}^{k_i-1} \sum_{j=1}^{n^\delta} \mathbf{A}_\ell[j] + \sum_{j=1}^{i'} \mathbf{A}_{k_i}[j] \Big]
\end{aligned}$$

$$= (1 + \lambda'(n))^2 \sum_{j=1}^{i} \mathbf{A}[j]$$
$$\leq (1 + \lambda(n)) \sum_{j=1}^{i} \mathbf{A}[j]$$

where the first inequality holds because $\mathbf{D}$ contains λ'-approximate prefix sums for $\mathbf{C}$, the following equality holds by definition of $\mathbf{C}$, and the second inequality holds because the $\mathbf{B}_\ell$ contain λ'-approximate prefix sums for the $\mathbf{A}_\ell$. The final inequality holds by choice of λ'.

It remains to show consistency for $i \geq 2$. That is, we have to assert that $\mathbf{B}[i]-\mathbf{B}[i-1] \geq \mathbf{A}[i]$ holds. We make a case distinction. If i and $i-1$ both map to the same subarray, i.e. if $k_i = k_{i-1}$ holds, then

$$\mathbf{B}[i]-\mathbf{B}[i-1] = (\mathbf{B}_{k_i}[i']+\mathbf{D}[k_i])-(\mathbf{B}_{k_i}[i'-1]+\mathbf{D}[k_i]) = \mathbf{B}_{k_i}[i']-\mathbf{B}_{k_i}[i'-1] \geq \mathbf{A}_{k_i}[i'] = \mathbf{A}[i]$$

where the inequality holds because $\mathbf{B}_{k_i}$ contains consistent approximate prefix sums for $\mathbf{A}_{k_i}$.

Otherwise, we can conclude that i is mapped to the first cell of $\mathbf{A}_{k_i}$ and $i-1$ is mapped to the last cell of $\mathbf{A}_{k_i-1}$. Thus, we have

$$\begin{aligned}
\mathbf{B}[i] - \mathbf{B}[i-1] &= (\mathbf{B}_{k_i}[1] + \mathbf{D}[k_i]) - (\mathbf{B}_{k_i-1}[n^\delta] + \mathbf{D}[k_i - 1]) \\
&= \mathbf{D}[k_i] - \mathbf{D}[k_i - 1] - \mathbf{B}_{k_i-1}[n^\delta] + \mathbf{B}_{k_i}[1] \\
&\leq \mathbf{C}[k_i] - \mathbf{B}_{k_i-1}[n^\delta] + \mathbf{B}_{k_i}[1] \\
&= \mathbf{B}_{k_i}[1] \\
&\leq \mathbf{A}_{k_i}[1] = \mathbf{A}[i]
\end{aligned}$$

where the first inequality holds because $\mathbf{D}$ contains consistent approximate prefix sums for $\mathbf{C}$, the first equality holds because we have $\mathbf{C}[k_i] = \mathbf{B}_{k_i-1}[n^\delta]$ by definition, and the last inequality holds because $\mathbf{B}_{k_i}$ contains approximate prefix sums for $\mathbf{A}_{k-1}$. □

Appendix B

Parallel-Correctness for Hash-Based Distribution and Communication Policies

In this appendix we briefly discuss an alternative to constraint-based communication policies that fits within our framework presented in Section 4.1. The idea is to reuse the hash functions from the distribution policy for the communication policy. We make this concrete next.

Hash-Based Communication. Let Z_1, Z_2 be hash policy schemes that refer to intensional relation symbols of a Datalog query, and H be a tuple of hash functions over a network $\mathcal{N}$ such that $Z_1 \cup Z_2$ is consistent and compatible with H. Then Z_1, Z_2, and H induce the communication policy $\gamma_{Z_1,Z_2,H}$ which maps a distributed database $\mathcal{D} = (G, \mathcal{I})$ over $\mathcal{N}$ to the set of all communicated facts $R(\bar{a})@k \triangleright \ell$ with $R(\bar{a})@k \in \mathcal{D}$, $R(\bar{a})@k \in \delta_{Z_1,H}(\{R(\bar{a})\})$, and $R(\bar{a})@\ell \in \delta_{Z_2,H}(\{R(\bar{a})\})$. That is, if a fact $R(\bar{a})$ resides on server k, and $\delta_{Z_1,H}$ "permits" server k to communicate it, it is communicated to all servers determined by $\delta_{Z_2,H}$. We note that these hash-based communication policies are – like hash-based distribution policies and in contrast to constraint-based communication policies – *fact-based* in the sense that they map facts $R(\bar{a})$ to sets of communicated facts $R(\bar{a})@k \triangleright \ell$, independent of other facts.

Let now Z be a hash policy scheme that refers to the extensional relations symbols of a Datalog query Q, and Z_1, Z_2 be hash policy schemes that refer to intensional relation symbols of Q. If $Z \cup Z_1 \cup Z_2$ is consistent, then $\mathcal{F}(Z, Z_1, Z_2)$ denotes the set of policy pairs $(\delta_{Z,H}, \gamma_{Z_1,Z_2,H})$, where H is any tuple of hash functions with which Z,Z_1, and Z_2 are compatible. We denote by Hash-Hash the class of families that are defined in this way by triples Z, Z_1, Z_2 of hash policy schemes.

Example B.1. Consider, once more, the monadic Datalog query $Q = (P, \text{Out})$ from Example 2.4.6 whose rules we repeat in the following for convenience.

$$R(x) \leftarrow \text{Start}(x) \qquad S(x) \leftarrow \text{Start}(x) \qquad \text{Out}(x) \leftarrow R(x), S(x)$$
$$R(x) \leftarrow R(y), E_r(y, x) \qquad S(x) \leftarrow S(y), E_s(y, x)$$

Recall that Q asks for all nodes reachable from a starting node by a path containing only red as well as a path containing only sea blue edges.

Furthermore, let

$$Z = \{(\text{Start}, 1, ()), (\text{Start}, 2, ()), (E_r, 1, ()), (E_s, 2, ())\},$$
$$Z_1 = \{(R, 1, ()), (S, 2, ())\}, \text{ and}$$
$$Z_2 = \{(R, 3, (1)), (S, 3, (1))\}.$$

Note that $Z \cup Z_1 \cup Z_2$ is consistent. Further, consider the tuple $H = (h_1, h_2, h_3)$ of hash functions, where h_1 and h_2 have arity 0 and map all facts to $\{1\}$ and $\{2\}$, respectively, and h_3 is a unary hash function mapping each value a onto $\{((a-1) \bmod 4)+1\}$. Then Z, Z_1 and Z_2 are compatible with H, and $\gamma_{Z_1,Z_2,H} = \gamma$, where γ is as defined in Example 4.1.1. That is, in a nutshell, if a fact $R(a)$ is derived on server 1 and a matching fact $S(a)$ is derived on server 2, then both are sent to the same server, which can then derive $\text{Out}(a)$. As discussed in Example 4.1.4 the distribution policy $\delta_{Z,H}$ ensures that all Start-facts initially reside on servers 1 and 2, all E_r-facts on server 1, and all E_s-facts on server 2.

Clearly, we thus have that Q is parallel-correct w.r.t. $(\delta_{Z,H}, \gamma_{Z_1,Z_2,H})$. In fact, this even holds for every tuple H of hash functions with which Z, Z_1, Z_2 are compatible. Or equivalently, Q is parallel-correct w.r.t. $\mathcal{F}(Z, Z_1, Z_2)$. ◁

The main result of this appendix is the following.

Theorem⋆ B.2. PC(FGDL, Hash-Hash) *and* PC(MDL, Hash-Hash) *are* 2ExpTime*-complete. The lower bounds even hold for instances with primitive hash policy schemes.*

The proof approach is similar to the ones for Proposition 4.2.16 and Theorem 4.2.28.[1] The lower bound is, in both cases, by a reduction from the containment problem for monadic Datalog queries. For the upper bound, we show that it suffices to restrict attention to scattering policies to establish parallel-correctness, and then construct a Datalog program that simulates the distributed evaluation over a scattered database.

The following is the analogue of Lemma 4.2.6.

Lemma⋆ B.3. *Let Q be a Datalog query, and Z, Z_1, and Z_2 be hash policy schemes such that $Z \cup Z_1 \cup Z_2$ is consistent. Then Q is parallel-correct w.r.t. $\mathcal{F}(Z, Z_1, Z_2)$ if and only if for all global databases G there is a policy pair $(\delta, \gamma) \in \mathcal{F}(Z, Z_1, Z_2)$ such that δ scatters G and $[Q, \gamma](\delta(G)) \supseteq Q(G)$ holds.*

Recall that, for any global database, there always is a distribution policy that scatters it, and every distributed database covers a scattered database. This is thanks to Lemma 4.2.7 and Lemma 4.2.10, respectively. To prove Lemma B.3 we also require the following monotonicity condition, which is the analogue of Lemma 4.2.11.

Lemma⋆ B.4. *Let Q be a Datalog query, Z, Z_1, and Z_2 hash policy schemes such that $Z \cup Z_1 \cup Z_2$ is consistent, G be a global database, and $\mathcal{D}'$ be a distributed databases. For all policy pairs $(\delta, \gamma), (\delta', \gamma') \in \mathcal{F}(Z, Z_1, Z_2)$ such that δ scatters G, and $\mathcal{D}'$ complies with δ', we have*

$$[Q, \gamma](\delta(G))) \subseteq [Q, \gamma'](\mathcal{D}').$$

[1] Historically correct is that result and proof for hash-based communication policies were there first.

Proof. Let $(\delta, \gamma), (\delta', \gamma') \in \mathcal{F}(Z, Z_1, Z_2)$ such that δ scatters G, and $\mathcal{D}'$ complies with δ'. For each $r \geq 0$, we denote by $\mathcal{D}^r = (G^r, \mathcal{I}^r)$ and $(\mathcal{D}')^r = ((G')^r, (\mathcal{I}')^r)$ the distributed databases after the r-th communication phase of the distributed evaluation over $\mathcal{D} = \delta(G)$ and $\mathcal{D}'$, respectively. Similar to the proof for Lemma 4.2.11, it suffices to show by induction on the number of rounds r, that for each $r \geq 0$, $(\mathcal{D}')^r$ covers $\mathcal{D}^r$. The statement of the lemma then follows immediately.

However, here we require a stronger induction hypothesis. Let $H = (h_1, \ldots, h_m)$ and $H' = (h'_1, \ldots, h'_m)$ be the underlying tuples of hash functions of (δ, γ) and (δ', γ'), respectively. That is, we have $\delta = \delta_{Z,H}$, $\gamma = \gamma_{Z_1,Z_2,H}$, $\delta' = \delta_{Z,H'}$, and $\gamma = \gamma_{Z_1,Z_2,H'}$. Further, let $\mathcal{N}$ and $\mathcal{N}'$ be the networks over which H and H' are defined, respectively. Since δ scatters G, by definition so does H. We define a mapping $s \colon \mathcal{N} \to \mathcal{N}'$ of servers as follows. For a server $k \in \mathcal{N}$, let $i \in [1, m]$ and $\bar{c}$ be a tuple of domain values in $\mathsf{adom}(G)$ such that $k \in h_i(\bar{c})$. We can assume that such an i and $\bar{c}$ exist for every server k; otherwise the server does not play any role in the evaluation and can be removed. More importantly, note that i and $\bar{c}$ are unique, since H scatters G. Pick an arbitrary but fixed $\ell \in h'_i(\bar{c})$. We set $s(k) = \ell$.

We show by induction over $r \geq 0$ that $I^r_k \subseteq (I'_{s(k)})^r$ holds for every server $k \in \mathcal{N}$. This then implies that $(\mathcal{D}')^r$ covers $\mathcal{D}^r$.

For the induction base, recall that $\mathcal{D}^0 = \delta_{Z,H}(G)$. Let $k \in \mathcal{N}$ be a server, and i and $\bar{c}$ such that $k \in h_i(\bar{c})$. Since H scatters G, we have that for every fact $R(\bar{a}) \in I^0_k$ there is a hash directive $(R, i, \bar{u})$ such that $\bar{a}[\bar{u}] = \bar{c}$, because δ scatters G. Consequently, all these facts are in $(I'_{s(k)})^0$ because $(\mathcal{D}')^0 = \mathcal{D}$ complies with $\delta_{Z,H'}$ and $s(k) \in h_i(\bar{c})$ by definition of s.

For the induction step, we assume that $I^{r-1}_k \subseteq (I'_{s(k)})^{r-1}$ holds for all servers $k \in \mathcal{N}$. Let $\mathcal{E}$ and $\mathcal{E}'$ be the distributed databases with families of local databases $\mathcal{J}$ and $\mathcal{J}'$ obtained from $\mathcal{D}^{r-1}$ and $(\mathcal{D}')^{r-1}$, respectively, after the r-th computation phase. In particular, we have $J^r_k = P(I^{r-1}_k)$ and $(J'_{s(k)})^r = P((I'_{s(k)})^{r-1})$ for every $k \in \mathcal{N}$ where P is the Datalog program of the query Q. Since Datalog queries (and Datalog programs) are monotone, $J^r_k \subseteq (J'_{s(k)})^r$ holds for all servers k.

For the communication phase, it suffices to show that, for every communicated fact $R(\bar{a})@k \triangleright \ell \in \gamma(\mathcal{E})$, we have that $R(\bar{a})@s(k) \triangleright s(\ell) \in \gamma(\mathcal{E}')$. For this purpose, let $R(\bar{a})@k \triangleright \ell$ be a communicated fact in $\gamma(\mathcal{E})$. Then $R(\bar{a})@k \in \mathcal{E}$, $R(\bar{a})@k \in \delta_{Z_1,H}(\{R(\bar{a})\})$, and $R(\bar{a})@\ell \in \delta_{Z_2,H}(\{R(\bar{a})\})$. Then there are hash directives $(R, i, \bar{u}) \in Z_1$ and $(R, j, \bar{v}) \in Z_2$ such that $k \in h_i(\bar{a}[\bar{u}])$ and $\ell \in h_j(\bar{a}[\bar{v}])$. By our definition of s, we then also have $s(k) \in h'_i(\bar{a}[\bar{u}])$ and $s(\ell) \in h'_j(\bar{a}[\bar{v}])$. In other words, $R(\bar{a})@s(k) \in \delta_{Z_1,H'}(\{R(\bar{a})\})$, and $R(\bar{a})@s(\ell) \in \delta_{Z_2,H'}(\{R(\bar{a})\})$. Moreover, $R(\bar{a})@s(k) \in \mathcal{E}'$ because $J^r_k \subseteq (J'_{s(k)})^r$. We can conclude that $R(\bar{a})@s(k) \triangleright s(\ell)$ is in $\gamma_{Z_1,Z_2,H'}(\mathcal{E}') = \gamma'(\mathcal{E}')$. □

We are now ready to prove that it suffices to consider scattering distribution policies. The proof is analogous to the one for Lemma 4.2.6.

Proof of Lemma B.3. For the only-if direction suppose that Q is parallel-correct w.r.t. $\mathcal{F}(Z, Z_1, Z_2)$ and let G be a global database. Thanks to Lemma 4.2.7 there is a tuple H of

hash functions such that $\delta_{Z,H}$ scatters G. Since Q is parallel-correct w.r.t. $\mathcal{F}(Z, Z_1, Z_2)$, we can conclude that $[Q, \gamma_{Z_1,Z_2,H}](\delta_{Z,H}) = Q(G)$ holds.

For the converse, suppose that for every database G there is policy pair $(\delta, \gamma) \in \mathcal{F}(Z, Z_1, Z_2)$ such that δ scatters G and $[Q, \gamma](\delta(G)) \supseteq Q(G)$ holds. We have to show that Q is parallel-correct w.r.t. $\mathcal{F}(Z, \Sigma)$.

Let $(\delta, \gamma) \in \mathcal{F}(Z, Z_1, Z_2)$ and let $\mathcal{D}$ be an arbitrary distributed database that complies with δ. Thanks to Observation 4.2.3 we have $[Q, \gamma](\mathcal{D}) \subseteq Q(G)$. Hence, it suffices to establish parallel-completeness, by showing $[Q, \gamma](\mathcal{D}) \supseteq Q(G)$.

By assumption there is a policy pair $(\delta', \gamma') \in \mathcal{F}(Z, Z_1, Z_2)$ such that δ' scatters G and $[Q, \gamma'](\delta'(G)) \supseteq Q(G)$. Thanks to Lemma 4.2.10, $\mathcal{D}$ covers $\delta'(G)$ because δ' scatters G. Therefore, we have $[Q, \gamma'](\delta'(G)) \subseteq [Q, \gamma](\mathcal{D})$ thanks to Lemma B.4.

Altogether, we get $[Q, \gamma](\mathcal{D}) \supseteq [Q, \gamma'](\delta'(G)) \supseteq Q(G)$. □

Next we show that there always is a scattered database such that the distributed evaluation of monadic and frontier-guarded Datalog queries can be simulated by frontier-guarded Datalog queries over the global database. This is similar to Lemmas 4.2.19 and 4.2.22.

Lemma⋆ B.5. *For every monadic or frontier-guarded Datalog query Q, and hash policy schemes Z, Z_1, and Z_2 such that $Z \cup Z_1 \cup Z_2$ is consistent, a frontier-guarded Datalog query Q' can be constructed in exponential time such that the following holds: For every global database G there is a policy pair $(\delta, \gamma) \in \mathcal{F}(Z, Z_1, Z_2)$ such that δ scatters G and $Q'(G) = [Q, \gamma](\delta(G))$.*

The number of variables and the length of the rules of Q' is polynomial in $\|Q\|$, $\|Z\|$, $\|Z_1\|$, and $\|Z_2\|$; and the number of rules is at most exponential.

Proof. Let $Q = (P, \text{Out})$ be a monadic or frontier-guarded Datalog query, and Z, Z_1, and Z_2 be hash policy schemes such that $Z \cup Z_1 \cup Z_2$ is consistent. We first construct a Datalog query $Q'' = (P'', \text{Out})$ that has the claimed equivalence property and afterwards we "guard" it.

Construction. The Datalog program P'' uses one intensional relation symbol R^i of arity $|\bar{u}| + \mathsf{ar}(R)$, for every relation symbol $R \in \mathsf{edb}(P) \cup \mathsf{idb}(P)$ and hash directive $(R, i, \bar{u}) \in Z \cup Z_1 \cup Z_2$. It has four kinds of rules.[2]

- For every $(E, i, \bar{u}) \in Z$, P'' has a rule $E^i(\bar{z}, \bar{x}) \leftarrow E(\bar{x})$, where $\bar{x}$ is a tuple of pairwise different variables, and $\bar{z} = \bar{x}[\bar{u}]$.
- For every rule $R(\bar{x}) \leftarrow S_1(\bar{y}_1), \ldots, S_n(\bar{y}_n)$ of P and hash directive $(E, i, \bar{u})$, P''' has a rule
$$R^i(\bar{z}, \bar{x}) \leftarrow S_1^i(\bar{z}, \bar{y}_1), \ldots, S_n^i(\bar{z}, \bar{y}_n),$$
where $\bar{z}$ is a tuple of pairwise different variables *not* occurring in any of the $\bar{y}_i$ or $\bar{x}$.

[2] As in the construction for Lemma 4.2.19 all symbols R^i for which *no* rule is constructed are removed, along with all rules where R^i occurs in the body, and recursively.

- For every hash directive $(R, j, \bar{v}) \in Z_1$ and every hash directive $(R, i, \bar{u}) \in Z_2$, P'' has a rule
$$R^i(\bar{z}, \bar{x}) \leftarrow R^j(\bar{y}, \bar{x}),$$
where $\bar{x}$ is a tuple of pairwise different variables with $|\bar{x}| = \mathsf{ar}(R)$, $\bar{z} = \bar{x}[\bar{u}]$, and $\bar{y} = \bar{x}[\bar{v}]$.
- For every hash directive $(E, i, \bar{u})$, P'' has a rule $\mathrm{Out}(\bar{x}) \leftarrow \mathrm{Out}^i(\bar{z}, \bar{x})$ where $\bar{z}$ and $\bar{x}$ are tuples of pairwise different variables that share *no* variable, and $|\bar{z}| = |\bar{u}|$.

Correctness. Let G be a global database, and $\mathcal{N}$ be the network of all servers $(i, \bar{c})$ for which there is a triple $(R, i, \bar{u}) \in Z \cup Z_1 \cup Z_2$, and where $\bar{c} \in \mathsf{adom}(G)^{|\bar{u}|}$. Furthermore, let $H = (h_1, \ldots, h_m)$ be the tuple of hash functions, where h_i has arity $|\bar{u}|$ and maps every tuple $\bar{c}$ to $h_i(\bar{c}) = (i, \bar{c})$, if there is a hash directive $(R, i, \bar{u})$ in any of the hash policy schemes. Clearly, $Z \cup Z_1 \cup Z_2$ is compatible with H, and H scatters G.

It suffices to show that a fact $R(\bar{a})$ resides on server $(i, \bar{c})$ after the distributed evaluation of Q over $\delta_{Z,H}(G)$ induced by $\gamma_{Z_1,Z_2,H}$ if and only if $R^i(\bar{c}, \bar{a})$ can be derived in the evaluation of Q'' over G. Thanks to the rules $\mathrm{Out}(\bar{x}) \leftarrow \mathrm{Out}^i(\bar{z}, \bar{x})$ of the fourth kind we can then conclude that $Q''(G) = [Q, \gamma_{Z_1,Z_2,H}](\delta_{Z,H}(G))$ holds.

The direction from left to right can be shown by induction over the number $r \geq 0$ of rounds of the distributed evaluation. If an extensional fact $E(\bar{a})$ resides on server $(i, \bar{c})$ in the initial distributed database $\mathcal{D}^0 = \delta_{Z,H}(G)$, then there is a hash directive $(E, i, \bar{u}) \in Z$ such that $\bar{c} = \bar{a}[\bar{u}]$. Consequently, $E^i(\bar{c}, \bar{a})$ can be derived by the rule $E^i(\bar{z}, \bar{x}) \leftarrow E(\bar{x})$ constructed for $(E, i, \bar{u})$.

For the computation phase in round $r \geq 1$ on a server $(i, \bar{c})$, we have that, for all facts $S(\bar{b})$ residing on server $(i, \bar{c})$ after the previous round, facts $S^i(\bar{c}, \bar{b})$ can be derived in the evaluation of Q'' thanks to the induction hypothesis. Thus, if a fact $R(\bar{a})$ can be derived by a rule $R(\bar{x}) \leftarrow S_1(\bar{y}_1), \ldots, S_n(\bar{y}_n)$ and a valuation ϑ on server $(i, \bar{c})$, then $R^i(\bar{c}, \bar{a})$ can be derived by the rule $R^i(\bar{z}, \bar{x}) \leftarrow S_1(\bar{z}, \bar{y}_1), \ldots, S_n(\bar{z}, \bar{y}_n)$ and ϑ'; where ϑ' is the valuation with $\vartheta'(\bar{z}) = \bar{c}$ and $\vartheta'(x) = \vartheta(x)$ for all variables in $\bar{x}$ or any of the $\bar{y}_j$. Note that ϑ' is well-defined because $\bar{z}$ shares, by construction, no variables with $\bar{x}$ or any of the $\bar{y}_j$.

It remains to consider the communication phase. Observe that, for each communicated fact $R(\bar{a})@(j, \bar{c}')\triangleright(i, \bar{c})$, there are hash directives $(R, j, \bar{v}) \in Z_1$ and $(R, i, \bar{u}) \in Z_2$ such that $\bar{c}' = \bar{a}[\bar{v}]$ and $\bar{c} = \bar{a}[\bar{u}]$. Furthermore, $R(\bar{a})$ resides on server $(j, \bar{c}')$ after the computation phase. Therefore, $R^j(\bar{c}', \bar{a})$ can be derived thanks to the induction hypothesis or the previous reasoning on the computation phase. Then, $R^i(\bar{c}, \bar{a})$ can be derived by the rule $R^i(\bar{z}, \bar{x}) \leftarrow R^j(\bar{y}, \bar{x})$ and the valuation ϑ with $\vartheta(\bar{x}) = \bar{a}$ (recall that then also $\vartheta(\bar{z}) = \bar{c}$ and $\vartheta(\bar{y}) = \bar{c}'$, since we have $\bar{z} = \bar{x}[\bar{u}]$ and $\bar{y} = \bar{x}[\bar{v}]$ by construction).

The direction from right to left can be shown similarly by structural induction over a proof tree for $R^i(\bar{c}, \bar{a})$. In the base case the root node is labelled with a fact $E^i(\bar{c}, \bar{a})$ where E is an extensional symbol. Furthermore, it is witnessed by a rule $E^i(\bar{z}, \bar{x}) \leftarrow E(\bar{x})$ and a valuation ϑ with $\vartheta(\bar{x}) = \bar{a}$ and $\vartheta(\bar{z}) = \bar{c}$. By construction, there is a hash directive $(E, i, \bar{u}) \in Z$ and $\bar{z} = \bar{x}[\bar{u}]$. Thus, $\bar{a}[\bar{u}] = \bar{c}$ and we have $h_i(\bar{a}) = (i, \bar{c})$. We can conclude that $E(\bar{a})@(i, \bar{c}) \in \delta_{Z,H}(G)$.

For the induction step, let $R^i(\bar{c}, \bar{a})$ be the label of the root node. We consider two cases. The first case is that the root node is witnessed by a rule $R^i(\bar{z}, \bar{x}) \leftarrow S_1^i(\bar{z}, \bar{y}_1), \ldots, S_n^i(\bar{z}, \bar{y}_n)$ obtained from a rule $R(\bar{x}) \leftarrow S_1(\bar{y}_1), \ldots, S_n(\bar{y}_n)$ from P. Let ϑ be the matching valuation. In particular, that means we have $\vartheta(\bar{x}) = \bar{a}$ and $\vartheta(\bar{z}) = \bar{c}$. Thanks to the induction hypothesis, all facts $\vartheta(S_j(\bar{y}_j))$ then reside on server $(i, \bar{c})$. Hence, $R(\bar{a})$ can be derived on server $(i, \bar{c})$ by $R(\bar{x}) \leftarrow S_1(\bar{y}_1), \ldots, S_n(\bar{y}_n)$ and ϑ.

The second case is that the root node is witnessed by a rule $R^i(\bar{z}, \bar{x}) \leftarrow R^j(\bar{y}, \bar{x})$ constructed for hash directives $(R, j, \bar{v}) \in Z_1$ and $(R, i, \bar{u}) \in Z_2$. Again, let ϑ be the matching valuation with, in particular, $\vartheta(\bar{x}) = \bar{a}$ and $\vartheta(\bar{z}) = \bar{c}$. By construction, we also have $\bar{z} = \bar{x}[\bar{u}]$ and $\bar{y} = \bar{x}[\bar{v}]$. Therefore, $\bar{a}[\bar{u}] = \bar{c}$, and we have $h_i(\bar{a}) = (i, \bar{c})$. In other words, $R(\bar{a})@(i, \bar{c}) \in \delta_{Z_2,H}(\{R(\bar{a})\})$. Analogously, $R(\bar{a})@(j, \vartheta(\bar{y})) \in \delta_{Z_1,H}(\{R(\bar{a})\})$ because $\vartheta(\bar{x}) = \bar{a}$ and $\bar{y} = \bar{x}[\bar{v}]$.

Moreover, thanks to the induction hypothesis, we have that $R(\bar{a})$ resides, from some point on, at server $(j, \vartheta(\bar{y}))$ because the (only) child of the root node is labelled with $\vartheta(R^j(\bar{y}, \bar{x}))$. Altogether, we can conclude that the communication policy $\gamma_{Z_1,Z_2,H}$ yields, in some round, the communicated fact $R(\bar{a})@(i, \bar{c}) \triangleright (j, \vartheta(\bar{y}))$. Thus, $R(\bar{a})$ resides at server $(i, \bar{c})$ from some round onwards.

Guarding. It remains to explain how $Q'' = (P'', \text{Out})$ can be transformed into an equivalent frontier-guarded Datalog query $Q' = (P', \text{Out})$.

We first consider the case that Q is frontier-guarded. Rules $E^i(\bar{z}, \bar{x}) \leftarrow E(\bar{x})$ constructed for hash directives $(E, i, \bar{u})$ are already frontier-guarded, since we have $\bar{z} = \bar{x}[\bar{u}]$.

Next, consider a rule $\tau\colon R^i(\bar{z}, \bar{x}) \leftarrow S_1^i(\bar{z}, \bar{y}_1), \ldots, S_n^i(\bar{y}_n)$ obtained from a rule $R(\bar{x}) \leftarrow S_1^i(\bar{y}_1, \ldots, \bar{y}_n)$ in P. Since Q is frontier-guarded, there is a j such that S_j is extensional, and every variable in $\bar{x}$ occurs in $\bar{y}_j$. Thus, the only rules for S_j^i in P'' are of the form $S_j^i(\bar{z}', \bar{y}') \leftarrow S_j(\bar{y}')$ where every variable in $\bar{z}'$ occurs in $\bar{y}'$, i.e. rules constructed for Z. We describe how such a rule $S_j^i(\bar{z}', \bar{y}') \leftarrow S_j(\bar{y}')$ can be "inlined" to obtain a frontier-guarded rule. The original rule τ is then replaced with all rules obtained in this fashion. Let $\alpha\colon \mathsf{var} \to \mathsf{vars}(\tau)$ be a mapping such that $\alpha(\bar{y}') = \bar{y}_j$. Note that such a mapping always exists because, by construction, $\bar{y}'$ is a tuple of pairwise different variables. The new rule is then $R^i(\alpha(\bar{z}'), \bar{x}) \leftarrow S_j(\alpha(\bar{y}')), S_1^i(\alpha(\bar{z}'), \bar{y}_1), \ldots, S_n^i(\alpha(\bar{z}'), \bar{y}_n)$. Note that the new atom $S_j(\alpha(\bar{y}'))$ is indeed a guard atom because all variables in $\bar{x}$ occur in $\alpha(\bar{y}') = \bar{y}_j$, and all variables in $\alpha(\bar{z}')$ occur in $\alpha(\bar{y}')$. The number of rules obtained in this fashion is at most exponential. The number of variables and their size is, however, still polynomial in $\|P\|$, $\|Z\|$, $\|Z_1\|$, and $\|Z_2\|$.

To "guard" rules of the form $\tau\colon R^i(\bar{z}, \bar{x}) \leftarrow R^j(\bar{y}, \bar{x})$, we recall that every variable in $\bar{z}$ or $\bar{y}$ occurs in $\bar{x}$ by construction. Furthermore, we have already shown that, if a fact $R^i(\bar{c}, \bar{a})$ can be derived by τ, then a fact $R(\bar{a})$ can be derived by some rule $R(\bar{x}') \leftarrow S_1(\bar{y}_1), \ldots, S_n(\bar{y}_n)$ of Q. Since Q is frontier-guarded there is a guard atom $S_m(\bar{y}_m)$ such that all variables of $\bar{x}'$ occur in $\bar{y}_m$. Then, for every tuple $\bar{y}'$ that contains all variables in $\bar{x}$, a frontier-guarded variant of τ can be obtained by adding the guard atom $S_m(\bar{y}')$ to its body. Again, τ can then be replaced by all frontier-guarded rules obtained in this fashion. And, again, the number of rule is at most exponential, and

their size and the number of variables is polynomial.

Finally, rules $\mathrm{Out}(\bar{x}) \leftarrow \mathrm{Out}^i(\bar{z}, \bar{x})$ can again be replaced by rules $\mathrm{Out}(\bar{x}') \leftarrow \mathsf{body}(\tau)$ for all (frontier-guarded variants of) rules τ with head $\mathrm{Out}^i(\bar{z}', \bar{x}')$.

Let us now suppose that the original query Q is monadic. Again, Rules $E^i(\bar{z}, \bar{x}) \leftarrow E(\bar{x})$ constructed for hash directives $(E, i, \bar{u})$ are already frontier-guarded, since we have $\bar{z} = \bar{x}[\bar{u}]$.

To "guard" the other rules of P'' we exploit the following observation, which we will prove below.

Claim B.6. *Every proof tree for an intensional fact $R^i(\bar{c}, \bar{a})$ with respect to P'' contains a node which is labelled with an extensional fact $E(\bar{b})$ such that all values in $\bar{a}$ and all values in $\bar{c}$ occur in $\bar{b}$.*

Based on this observation, P' is constructed as follows. Let τ be a rule of P'' with a head $R^i(\bar{z}, \bar{x})$ for some intensional relation symbol R. For each extensional relation symbol E and for each tuple $\bar{y}$ of arity $\mathsf{ar}(E)$ which contains all variables of $\bar{x}$, we add a new rule to P' that results from τ by adding the guard atom $R(\bar{y})$. The observation guarantees that, for each fact that is derived by some rule of P'', there is a rule in P' that derives it as well, since a valuation can map the guard atom to the "covering" extensional atom.

As usual, the rules of the form $\mathrm{Out}(\bar{x}) \leftarrow \mathrm{Out}^i(\bar{z}, \bar{x})$ can then be substituted by rules $\mathrm{Out}(\bar{x}') \leftarrow \mathsf{body}(\tau)$ for all (frontier-guarded variants of) rules τ with head $\mathrm{Out}^i(\bar{z}', \bar{x}')$.

It remains to prove Claim B.6. We do so by induction over the structure of a proof tree with respect to P'' and any global database.

In the base case, the proof tree property is witnessed by a rule $E^i(\bar{z}, \bar{x}) \leftarrow E(\bar{x})$ where all variables in $\bar{z}$ occur in $\bar{x}$. Thus, our claim holds.

For the induction step, let $R^i(\bar{c}, a)$ be the label of the root node. We make a case distinction. First, we consider the case that the root node is witnessed by a rule of the form $R^i(\bar{z}, x) \leftarrow S_1^i(\bar{z}, \bar{y}_1), \ldots, S_n^i(\bar{z}, \bar{y}_n)$. Pick an atom $S_j^i(\bar{z}, \bar{y}_j)$ such that x occurs in $\bar{y}_j$. Since every Datalog rule is *safe*, such an atom always exists. Thanks to the induction hypothesis, there is an extensional fact $E(\bar{b})$ that contains all values that occur in $\vartheta(S_j^i(\bar{z}, \bar{y}_j))$, where ϑ is the valuation witnessing the root node. Since x occurs in $\bar{y}_j$, $\bar{c} = \vartheta(\bar{z})$, and $a = \vartheta(x)$, we can conclude that all values in $\bar{c}$ and a occur in $\bar{b}$.

The second case is that the root node is witnessed by a rule of the form $R^i(\bar{z}, x) \leftarrow R^j(\bar{y}, x)$. By construction, $\bar{z}$ and $\bar{y}$ consist of (zero or more) repetitions of x. Hence, $\bar{c}$ consists of repetitions of a as well. Thus, the claim follows by induction, because a has to occur in the label of the (only) child node. □

Altogether, we can now prove that the parallel-correctness problem for hash-based distribution policies and hash-based communication policies is 2ExpTime-complete.

Proof of Theorem B.2. The proof for the upper bound is again almost identical to the proofs of Proposition 4.2.16 and Theorem 4.2.28. Let Q be a monadic or frontier-guarded Datalog query, and Z, Z_1, and Z_2 be hash policy schemes such that $Z \cup Z_1 \cup Z_2$ is consistent.

Thanks to Lemma B.5 there is a frontier-guarded Datalog query Q' such that, for every global database G, we have that $Q'(G) = [Q, \gamma](\delta(G))$ for some $(\delta, \gamma) \in \mathcal{F}(Z, \Sigma)$ with δ scattering G.

We claim that Q is parallel-correct w.r.t. $\mathcal{F}(Z, Z_1, Z_2)$ if and only if $Q \sqsubseteq Q'$ holds. Suppose $Q \sqsubseteq Q'$ holds and let G be a global database. Let (δ, γ) be the policy pair guaranteed by Lemma B.5 such that δ scatters G and $[Q, \gamma](\delta(G)) = Q'(G)$ holds. By assumption we have $Q(G) \subseteq Q'(G)$, and, thus, $Q(G) \subseteq [Q, \gamma](\delta(G))$. Thanks to Lemma B.3 we can conclude that Q is parallel-correct w.r.t. $\mathcal{F}(Z, Z_1, Z_2)$.

Conversely, suppose Q is parallel-correct w.r.t. $\mathcal{F}(Z, Z_1, Z_2)$. Let G be a global database and (δ, γ) be the policy pair satisfying $[Q, \gamma](\delta(G)) = Q'(G)$ due to Lemma B.5. Since Q is parallel-correct we have $Q(G) = [Q, \gamma](\delta(G))$. Together we have, in particular, $Q(G) \subseteq Q'(G)$. We can conclude that $Q \sqsubseteq Q'$ holds.

All in all, parallel-correctness of Q can thus be decided by testing $Q \sqsubseteq Q'$.

The upper bound for the complexity of the parallel-correctness now follows in the same way as in the proofs for Proposition 4.2.16 and Theorem 4.2.28 with the help of Theorem 4.2.20 and the size constraints guaranteed by Lemma B.5.

It remains to prove 2ExpTime-hardness. The proof is by a reduction from the containment problem for monadic Datalog queries, which is known to be 2ExpTime-hard [BBS12, Theorem 2]. Let $Q_1 = (P_1, \mathrm{Out})$ and $Q_2 = (P_2, \mathrm{Out})$ be two monadic Datalog queries. Thanks to Lemma 2.4.9, we can assume that Q_1 and Q_2 are frontier-guarded as well. Moreover, we can assume that Out is the only intensional symbol P_1 and P_2 have in common; that is, $\mathsf{idb}(P_1) \cap \mathsf{idb}(P_2) = \{\mathrm{Out}\}$.

We construct a monadic frontier-guarded Datalog query $Q = (P, \mathrm{Out})$ and hash policy schemes Z, Z_1, and Z_2 such that Q is parallel-correct w.r.t. $\mathcal{F}(Z, Z_1, Z_2)$ if and only if $Q_1 \sqsubseteq Q_2$.

The query Q is the same as constructed in the lower bound proof for Proposition 4.2.16. For convenience, we recall the construction here. Let P_1' and P_2' be the Datalog programs obtained by replacing Out with Out_1 and Out_2 in P_1 and P_2, respectively. The output symbol of Q is Out, which does not occur in P_1' or P_2'; and the Datalog program is

$$P = P_1' \cup P_2' \cup \big\{\mathrm{Out}(x) \leftarrow \mathrm{Out}_1(x), E(x)\big\} \cup \big\{\mathrm{Out}(x) \leftarrow \mathrm{Out}_2(x), E(x)\big\},$$

where E is a fresh extensional symbol not appearing in $\mathsf{edb}(P_1') \cup \mathsf{edb}(P_2')$. Clearly, Q is monadic and frontier-guarded if Q_1 and Q_2 are.

On a global database G, the query result of Q is the union of $Q_1(G)$ and $Q_2(G)$ intersected with the relation E.

We next define the three primitive hash policy schemes as follows.

$$\begin{aligned} Z &= \big\{(R, 1, ()) \mid R \in \mathsf{edb}(P_1) \cup \mathsf{edb}(P_2)\big\} \cup \big\{(E, 2, ())\big\} \\ Z_1 &= \big\{(\mathrm{Out}_2, 2, ())\big\} \\ Z_2 &= \big\{(\mathrm{Out}_2, 1, ())\big\} \end{aligned}$$

Let G be a global database and $(\delta, \gamma) \in \mathcal{F}(Z, Z_1, Z_2)$ such that δ scatters G. Then, considering the distributed database $\delta(G)$, there are two distinct servers k and ℓ such

that all facts over $\mathsf{edb}(P) \setminus \{E\}$ reside on k and all E-facts reside on server ℓ. Notably, there is *no* local database of $\delta(G)$ that contains E-facts and facts over $\mathsf{edb}(P) \setminus \{E\}$.

Therefore, the outputs of P_1' and P_2' can be computed on server k because neither program refers to E. However, no Out-facts can be derived on k since no E-fact resides on k.

Due to the communication policy γ all Out_2-facts computed on k are send to ℓ. Hence, the intersection of Out_2 with E can be computed on ℓ thanks to the rule $\text{Out}(\bar{x}) \leftarrow \text{Out}_2(\bar{x}), E(\bar{x})$. On the other hand, the rule $\text{Out}(\bar{x}) \leftarrow \text{Out}_1(\bar{x}), E(\bar{x})$ is *never* used to derive a fact in the distributed evaluation, because only Out_2-facts can be communicated. Thus, the distributed evaluation yields the intersection of $Q_2(G)$ with the relation E.

It remains to argue that $Q_1 \sqsubseteq Q_2$ if and only if Q is parallel-correct w.r.t. $\mathcal{F}(Z, Z_1, Z_2)$. By Lemma B.3 it suffices to consider global databases G and distribution policies δ such that δ scatters G. As argued before $[Q, \gamma](\delta(G))$ is the intersection of $Q_2(G)$ with the relation E and $Q(G)$ is the intersection of $Q_1(G) \cup Q_2(G)$ with the relation E for such databases and distribution policies. Thus, if $Q_1 \sqsubseteq Q_2$ does *not* hold, then there is a G such that $[Q, \gamma](\delta(G)) \subsetneq Q(G)$ holds for any policy pair (δ, γ) such that δ scatters G. Since Lemma 4.2.7 guarantees that such a policy pair exists, we can conclude that Q is *not* parallel-correct w.r.t. $\mathcal{F}(Z, Z_1, Z_2)$.

For the converse suppose $Q_1 \sqsubseteq Q_2$ holds. Let G be a global database and δ be the distribution policy guaranteed by Lemma 4.2.7 that scatters G, and γ be the communication policy such that $(\delta, \gamma) \in \mathcal{F}(Z, Z_1, Z_2)$. By assumption we have that $Q_1(G) \subseteq Q_2(G)$ and, hence, $Q_2(G) = Q_1(G) \cup Q_2(G)$. Therefore, $[Q, \gamma](\delta(G)) = Q(G)$. Then Lemma B.3 allows us to conclude that Q is parallel-correct w.r.t. $\mathcal{F}(Z, Z_1, Z_2)$. □