Marcelo Arenas, Pablo Barceló, Leonid Libkin,
Wim Martens, Andreas Pieris

Principles of Databases

(Preliminary Version)

May 2, 2021

Santiago Paris
Bayreuth Edinburgh

Contents

1

Introduction

This is a release of parts 1, 2, and 4 of the upcoming book "Principles of Databases", which will be about the foundational and mathematical principles of databases in its various forms. The first two parts focus on an overview of the relational model, and on processing some of the most commonly occurring relational queries. Forthcoming parts will focus on additional aspects of the relational model and will cover tree-structured and graph-structured data as well.

The general philosophy of the book is the following:

- We planned the book such that large parts of it are suitable for teaching. A chapter roughly corresponds to the contents of a single lecture.
- For the ease of teaching and understanding the material, we may sometimes cut corners. If we want to give the reader a relatively quick insight of a particular result, this sometimes means that we present a weaker form of the result than the most general result known in the literature.

We have been teaching from this book ourselves, but the present version will undoubtedly still have errors. If you find any errors in the book, or places that you find particularly unclear, please let us know through the repository: `https://github.com/pdm-book/community`. The new versions of the book, including corrections, will be published in this repository.

What is Planned

The finished book will consist of the following parts:

(I) The Relational Model: The Classics
(II) Conjunctive Queries
(III) Fast Conjunctive Query Evaluation

- Includes material on acyclic queries, treewidth and hypertreewidth, and worst-case optimal join algorithms.

(IV) Expressive Languages

- Includes material on adding features found in most commonly used query languages: union, negation, aggregates, and recursion.

(V) Uncertainty

- Includes material on incomplete information, probabilistic databases, consistent query answering, and query answering in the presence of ontologies.

(VI) Query Answering Paradigms

- Includes material on bag semantics, incremental maintenance, provenance, top-k queries, distributed evaluation, and constant delay query evaluation.

(VII) Mappings and Views

- Includes material on determinacy, data exchange, and ontology-based data access.

(VIII) Tree-Structured Data

- Includes material on tree pattern queries, XPath, MSO, tree automata, monadic datalog, schema languages, and their static analysis.

(IX) Graph-Structured Data

- Includes material on various types of graph queries, their evaluation and containment, property graphs, RDF, and SPARQL.

We will continue to release parts, not necessarily in the order presented here. Furthermore, the ordering and contents of the chapters is preliminary and may change in future versions.

What is Still Missing, Even From Released Parts

Let's start by saying what is mainly there: in every chapter that we release, we believe that the technical content is relatively stable. For every part that we have released, two things still need work though:

Exercises. We have generated some initial ideas for exercises, but we are aware that the exercises for the currently released parts still need work. In fact, we are open to exercise suggestions.

Bibliography. We plan to accompany each part with references and a bibliographic discussion. These are not implemented yet, even for Parts I–IV.

2

Background

In this chapter, we introduce the mathematical concepts and terminology that will be used throughout the book. These include:

- the relational model,
- queries and query languages, and
- computational problems central in the study of principles of databases.

Basic Notions and Notation

We begin with a brief discussion of the very basic mathematical notions and notation that we are going to use in this book.

Sets

A *set* contains a finite or infinite number of elements (e.g., numbers, symbols, other sets), without repetition or respect to order. The elements in a set S are the *members* of S. We use the symbols $\in$ and $\notin$ to denote set membership and nonmemberhip, respectively. For a finite set S, we write $|S|$ for its *cardinality*, that is, the number of elements in it. The set without elements is called the *empty set*, written as $\emptyset$.

Given two (finite or infinite) sets S and T, we write:

- $S \cup T$ for their *union* $\{a \mid a \in S \text{ or } a \in T\}$,
- $S \cap T$ for their *intersection* $\{a \mid a \in S \text{ and } a \in T\}$, and
- $S - T$ for their *difference* $\{a \mid a \in S \text{ and } a \notin T\}$.

We further say that

- S is *equal* to T, written $S = T$, when $x \in S$ if and only if $x \in T$,

- S is a *subset* of T, written $S \subseteq T$, when $x \in S$ implies $x \in T$, and
- S is a *proper* (or *strict*) subset of T, written $S \subsetneq T$, if $S \subseteq T$ and $S \neq T$.

We write $\mathcal{P}(S)$ for the *powerset* of S, that is, the set consisting of all the subsets of S. Analogously, we write $\mathcal{P}_{\text{fin}}(S)$ for the *finite powerset* of S, namely the set consisting of all the finite subsets of S.

We write $\mathbb{N}$ for the set $\{0, 1, 2, \ldots\}$ of natural numbers. For $i, j \in \mathbb{N}$, we denote by $[i, j]$ the set $\{k \in \mathbb{N} \mid i \leq k \text{ and } k \leq j\}$. We simply write $[i]$ for $[1, i]$.

Sequences and Tuples

A *sequence* of elements is a list of these elements in some order. We typically identify a sequence by writing the list within parentheses. Recall that in a set the order does not matter, but in a sequence it does. Hence, the sequence $(1, 2, 3)$ is not the same as $(3, 2, 1)$. Similarly, repetition does not matter in a set, but is does matter in a sequence. Thus, the sequence $(1, 1, 2, 3)$ is different than $(1, 2, 3)$, while the set $\{1, 1, 2, 3\}$ is the same as $\{1, 2, 3\}$. Finite sequences are called *tuples*. A sequence with $k \in \mathbb{N}$ elements is a tuple of *arity* k, called *k-ary tuple* (or simply *k-tuple*). Note that when $k = 0$ we get the empty tuple $()$. We often abbreviate a k-ary tuple $(a_1, \ldots, a_k)$ as $\bar{a}$. Moreover, for a k-ary tuple $\bar{a}$, we usually assume that its elements are $(a_1, \ldots, a_k)$. We say that $\bar{a} = (a_1, \ldots, a_k)$ has the *positions* $1, \ldots, k$ and that an element b *occurs* at position i if $b = a_i$. For example, 1 occurs at positions 1 and 3 in the tuple $(1, 2, 1, 4)$. Conversely, $\bar{a}$ *mentions* a if $a \in \{a_1, \ldots, a_k\}$.

For two sets S, T, we write $S \times T$ for the set of all pairs (a, b), where $a \in S$ and $b \in T$, called the *Cartesian product* or *cross product* of S and T. We can also define the Cartesian product of $k \geq 1$ sets $S_1, \ldots, S_k$, known as the *k-fold Cartesian product*, which is the set of all tuples $(a_1, \ldots, a_k)$, where $a_i \in S_i$ for each $i \in [k]$. For the k-fold Cartesian product of a set S with itself we write

$$S^k \;=\; \underbrace{S \times \cdots \times S}_{k}.$$

Functions

Consider two (finite or infinite) sets S and T. A *function* f from S to T, written $f : S \to T$, is a mapping from (all or some) elements of S to elements of T, i.e., for every $a \in S$, either $f(a) \in T$, in which case we say f is defined on a, or $f(a)$ is undefined, such that the following holds: for every $a, b \in S$ on which f is defined, $a = b$ implies $f(a) = f(b)$. We call f *total* if it is defined on every element of S; otherwise, it is called *partial*. By default, we assume functions to be total. When a function f is partial, we explicitly say this, and write $\text{Dom}(f)$ for the set of elements from S on which f is defined.

We say that a function f is

- *injective* (or *one-to-one*) if $a \neq b$ implies $f(a) \neq f(b)$ for every $a, b \in S$;

- *surjective* (or *onto*) if, for every $b \in T$, there is $a \in S$ such that $f(a) = b$,
- *bijective* (or *one-to-one correspondence*) if it is injective and surjective.

A useful notion is that of composition of functions. Given two functions $f : S \to T$ and $g : T \to U$, the *composition* of f and g, denoted $g \circ f$, is the function from S to U defined as follows: $g \circ f(a) = g(f(a))$ for every $a \in S$. Another useful notion is that of union of functions. Given two functions $f : S \to T$ and $g : S' \to T'$ with $f(a) = g(a)$ for every $a \in S \cap S'$, the *union* of f and g, denoted $f \cup g$, is the function from $S \cup S'$ to $T \cup T'$ defined as follows: $f \cup g(a) = f(a)$ for every $a \in S$, and $f \cup g(a) = g(a)$ for every $a \in S'$.

Given a function $f : S \to T$, for brevity, we will use the same letter f to denote extensions of f on more complex objects (such as tuples of elements of S, sets of elements of S, etc.). More precisely, if $\bar{a} = (a_1, \ldots, a_k) \in S^k$, then $f(\bar{a}) = (f(a_1), \ldots, f(a_k))$. If $R \subseteq S$, then $f(R) = \{f(a) \mid a \in R\}$. Notice that this convention also extends further, e.g., to sets of sets of tuples.

Graphs and Trees

A *graph* is a tuple

$$G = (V, E)$$

where V is a finite set of *nodes*. We distinguish between *directed* and *undirected* graphs, which only differ in the way how we define their set E of *edges*.

- In *directed graphs*, we have that $E \subseteq V \times V$.
- In *undirected graphs*, we have that $E \subseteq \{S \subseteq V \mid 1 \leq |S| \leq 2\}$.

An *undirected path* in a (directed or undirected) graph G is a non-empty sequence of nodes

$$\pi = n_0 n_1 \cdots n_k$$

where

- if G is a directed graph, then $(n_{i-1}, n_i) \in E$ or $(n_i, n_{i-1}) \in E$ for every $i \in [k]$ and
- if G is an undirected graph, then $\{n_{i-1}, n_i\} \in E$ for every $i \in [k]$.

We say that π is *from* n_0 *to* n_n and has *length* n. (The path of length zero is from n_0 to n_0.) A graph is *connected* if, for every pair of nodes $n, m \in V$, there is an undirected path from n to m.

A connected directed graph $T = (V, E)$ is a *tree* if

- for every node n, there is at most one node m with $(m, n) \in E$ (called the *parent* of n) and
- there is exactly one node n (called the *root* of T) without a parent.

As is common in Computer Science (and as opposed to Biology), we graphically depict trees with their root on top, such that all edges are directed downward. As such, even if trees are directed, we do not put arrows on their edges.

The Relational Model

To define tables in real-life databases, for example, by the `create table` statements of SQL, one needs to specify their names and names of their attributes. Therefore, to model databases, we need two disjoint sets

$$\mathsf{Rel} \text{ of } \textit{relation names} \qquad \text{and} \qquad \mathsf{Att} \text{ of } \textit{attribute names}.$$

We assume that these sets are countably infinite in order to ensure that we never run out of ways to name new tables and their attributes. In practice, of course, these sets are finite but extremely large: they are strings that can be so large that one never really runs out of names. Theoretically, we model this by assuming that these sets are countably infinite.

In `create table` declarations, one specifies types of attributes as well, for example, integer, Boolean, string. In the study of the theoretical foundations of databases, one typically does not make this distinction, and assumes that all elements populating databases come from another countably infinite set

$$\mathsf{Const} \text{ of } \textit{values}.$$

This simplifying assumption does not affect the various results on the complexity of query evaluation, expressiveness of languages, equivalence of queries, and many other subjects studied in this book. At the same time, it brings the setting closer to that of mathematical logic, allowing us to borrow many tools from it. It also allows us to significantly streamline notations.

The Named and Unnamed Perspective

There exist two standard perspectives from which databases can be defined, called the *named* and the *unnamed* perspectives. While the named perspective is closer to how databases appear in database management systems, and therefore more natural when giving examples, the unnamed perspective provides a clean mathematical model that is easier to use for studying the principles of databases. Importantly, the modeling power of those two perspectives is exactly the same, which allows us to go back and forth between the two.

Named Perspective. Under the *named perspective*, attribute names are viewed as an explicit part of a database. More precisely, a *database tuple* is a function $t : U \to \mathsf{Const}$, where $U = \{A_1, \ldots, A_k\}$ is a finite subset of Att. The *sort* of t is U, and its *arity* is the cardinality $|U|$ of U; we

say that t is k-ary if $|U| = k$. We usually do not use the function notation for database tuples in the named perspective, and denote them as $t = (A_1 \colon a_1, \ldots, A_k \colon a_k)$, meaning that $t(A_i) = a_i$ for every $i \in [k]$. Notice that, according to this notation, $(A_1 \colon a_1, A_2 \colon a_2)$ and $(A_2 \colon a_2, A_1 \colon a_1)$ represent the same function t. A *relation instance* in the named perspective is a *finite* set S of database tuples of the same sort U, which we also call the sort of the relation instance S and denote by $\mathrm{sort}(S)$. By nRI (for *named* relational instances) we denote the set of all such relation instances. A *possibly infinite relation instance* in the named perspective is defined as the notion of relation instance, but without forcing it to be finite. We write nRI^∞ for the set of all possibly infinite relation instances in the named perspective.

Database systems usually use a *database schema* that associates attribute names to relation names. This can be formalized as follows.

Definition 2.1: Named Database Schema

A *named (database) schema* is a partial function

$$\mathbf{S} \ : \ \mathsf{Rel} \to \mathcal{P}_{\mathrm{fin}}(\mathsf{Att})$$

such that $\mathrm{Dom}(\mathbf{S})$ is finite. For $R \in \mathrm{Dom}(\mathbf{S})$, the *sort of R under $\boldsymbol{S}$* is the set $\mathbf{S}(R)$. The *arity of R under $\boldsymbol{S}$*, denoted $\mathrm{ar}_{\mathbf{S}}(R)$, is $|\mathbf{S}(R)|$.

In other words, a named database schema $\mathbf{S}$ provides a finite set of relation names, together with their (finitely many) attribute names. These attribute names form the sort of the relation names under $\mathbf{S}$, and their number specifies the arity of the relation names under $\mathbf{S}$. For arities 1, 2, and 3, we speak of unary, binary, and ternary relation names, respectively. We now introduce the notion of database instance of a named schema.

Definition 2.2: Database Instance (The Named Case)

A *database instance* D of a named schema $\mathbf{S}$ is a function

$$D \ : \ \mathrm{Dom}(\mathbf{S}) \to \mathsf{nRI}$$

such that $\mathrm{sort}(D(R)) = \mathbf{S}(R)$, for every $R \in \mathrm{Dom}(\mathbf{S})$.

We can also talk about possibly infinite database instances. Formally, a *possibly infinite database instance* D of a named schema $\mathbf{S}$ is a function

$$D \ : \ \mathrm{Dom}(\mathbf{S}) \to \mathsf{nRI}^\infty$$

such that $\mathrm{sort}(D(R)) = \mathbf{S}(R)$, for every $R \in \mathrm{Dom}(\mathbf{S})$. This means that D is either finite as in Definition 2.2, where each relation name of $\mathrm{Dom}(\mathbf{S})$ is

mapped to a finite relation instance, or infinite in the sense that at least one relation name of $\mathrm{Dom}(\mathbf{S})$ is mapped to an infinite relation instance. Infinite database instances are obviously not a real-life concept, and we are not interested in studying them per se. Having said that, they are a very useful mathematical tool as they allow us to prove some results in a more elegant way. In other words, infinite database instances are considered for purely technical reasons, which will be revealed later in the book.

To avoid heavy notation, and because the name $\mathbf{S}$ of a schema is often not important, we usually provide schema information without explicitly using the symbol $\mathbf{S}$. We write $R[A_1, \ldots, A_k]$ instead of $\mathbf{S}(R) = \{A_1, \ldots, A_k\}$ for the schema $\mathbf{S}$ in question. For example, we write

```
City[city_id, name, country]
```

to refer to a relation name `City` with attribute names `city_id`, `name`, and `country`. Likewise, we write $\mathrm{ar}(R)$ instead of $\mathrm{ar}_{\mathbf{S}}(R)$. We may even write $R[k]$ to indicate that the arity of R under the schema in question is k

Unnamed Perspective. Under the *unnamed perspective*, a *database tuple* is an element of Const^k for some $k \in \mathbb{N}$. We denote such tuples using lowercase letters from the beginning of the alphabet, that is, as $(a_1, \ldots, a_k)$, $(b_1, \ldots, b_k)$, etc., or even more succinctly as $\bar{a}, \bar{b}$, etc. A *relation instance* in the unnamed perspective is a *finite* set S of database tuples of the same arity k. We say that k is the *arity* of S, denoted by $\mathrm{ar}(S)$. By uRI (for *unnamed* relation instances) we denote the set of all such relation instances. A *possibly infinite relation instance* in the unnamed perspective is defined as the notion of relation instance, but without forcing it to be finite. We write uRI^∞ for the set of all possibly infinite relation instances in the unnamed perspective. The notion of unnamed database schema follows.

Definition 2.3: Unnamed Database Schema

An *unnamed (database) schema* is a partial function

$$\mathbf{S} \ : \ \mathsf{Rel} \to \mathbb{N}$$

such that $\mathrm{Dom}(\mathbf{S})$ is finite. For a relation name $R \in \mathrm{Dom}(\mathbf{S})$, the *arity of R under $\mathbf{S}$*, denoted $\mathrm{ar}_{\mathbf{S}}(R)$, is defined as $\mathbf{S}(R)$.

In simple words, an unnamed databases schema $\mathbf{S}$ provides a finite set of relation names from Rel, together with their arity. We proceed to introduce the notion of database instance of an unnamed database schema.

Definition 2.4: Database Instance (The Unnamed Case)

A *database instance* D of an unnamed schema $\mathbf{S}$ is a function

$$D \ : \ \mathrm{Dom}(\mathbf{S}) \to \mathsf{uRI}$$

such that $\mathrm{ar}(D(R)) = \mathrm{ar}_{\mathbf{S}}(R)$, for every $R \in \mathrm{Dom}(\mathbf{S})$.

Analogously, a *possibly infinite database instance* D of an unnamed schema $\mathbf{S}$ is defined as a function of the form

$$D \ : \ \mathrm{Dom}(\mathbf{S}) \to \mathsf{uRI}^{\infty}$$

such that $\mathrm{ar}(D(R)) = \mathrm{ar}_{\mathbf{S}}(R)$, for every $R \in \mathrm{Dom}(\mathbf{S})$. Recall that infinite database instances are considered for purely technical reasons. As in the named perspective, in order to avoid heavy notation, we write $\mathrm{ar}(R)$ instead of $\mathrm{ar}_{\mathbf{S}}(R)$ for the arity of R under $\mathbf{S}$. We may even write $R[k]$ to indicate that the arity of R under the schema in question is k.

For a (named or unnamed) schema $\mathbf{S}$, we write $\mathrm{Inst}(\mathbf{S})$ for the set of all database instances of $\mathbf{S}$. Notice that $\mathrm{Inst}(\mathbf{S})$ does not contain infinite database instances. We also need the crucial notion of the active domain of a (possibly infinite) database instance, which is, roughly speaking, the set of constants that occur in it. Under the named perspective, we say that a database tuple $t : U \to \mathsf{Const}$ *mentions* a constant $a \in \mathsf{Const}$ if there exists $A \in U$ such that $t(A) = a$. Under the unnamed perspective, a database tuple $(a_1, \ldots, a_k) \in \mathsf{Const}^k$ *mentions* $a \in \mathsf{Const}$ if there exists $i \in [k]$ such that $a_i = a$. The *active domain* of a (possibly infinite) database instance D of $\mathbf{S}$ is defined as the set

$$\{a \in \mathsf{Const} \mid \text{there exists } R \in \mathrm{Dom}(\mathbf{S}) \text{ such that } D(R) \text{ contains a database tuple that mentions } a\}.$$

Henceforth, for brevity, we simply refer to the domain instead of the active domain of D, and denote it $\mathrm{Dom}(D)$. We will *never* use the term domain, and the notation $\mathrm{Dom}(D)$, to refer to the domain of the function D, i.e., $\mathrm{Dom}(\mathbf{S})$.

Simplified Terminology and Notation

We will refer to a (possibly infinite) database instance as a *(possibly infinite) database*, to a relation instance as a *relation*, and to a database tuple as a *tuple*. In both the named and the unnamed perspectives, we will write R_i^D instead of $D(R_i)$. When it is clear from the context, we shall omit the superscript D, and simply write R_i instead of R_i^D. This means that we will effectively use the same notation for relation names and for relation instances. This is a common practice that is used to simplify notation, and it will never lead to confusion; when the instance is important, we will make it explicit.

Although database schemas are formally defined as partial functions, with their domain being a finite subset of Rel, it is often convenient to tread them as sets of relation names. Thus, we will usually tread a schema $\mathbf{S}$ as the finite set

$\mathrm{Dom}(\mathbf{S})$. This means that whenever we write $\mathbf{S} = \{R_1, \ldots, R_n\}$, we actually mean that $\mathrm{Dom}(\mathbf{S}) = \{R_1, \ldots, R_n\}$. In the unnamed case, we may also write

$$\mathbf{S} \;=\; \{R_1[k_1], \ldots, R_n[k_n]\}$$

for the fact that $\mathrm{Dom}(\mathbf{S}) = \{R_1, \ldots, R_n\}$ and $\mathbf{S}(R_i) = k_i$, for each $i \in [n]$. Having this notation for schemas, we can then take, e.g., the union $\mathbf{S}_1 \cup \mathbf{S}_2$ of two schemas $\mathbf{S}_1$ and $\mathbf{S}_2$ (providing that $\mathrm{Dom}(\mathbf{S}_1)$ and $\mathrm{Dom}(\mathbf{S}_2)$ are disjoint).

Analogously, databases of unnamed schemas can be seen as sets, in particular, as sets of facts. For a k-ary relation name R, and a tuple $\bar{a} \in \mathsf{Const}^k$, we call $R(\bar{a})$ a *fact*. Since a fact is always a statement about a single tuple, we simplify the notation $R((a_1, \ldots, a_k))$ to $R(a_1, \ldots, a_k)$. We will usually tread a (possibly infinite) database D of an unnamed schema $\mathbf{S}$ as the set of facts

$$\left\{R(\bar{a}) \mid R \in \mathbf{S} \;\text{ and }\; \bar{a} \in R^D\right\}.$$

For example, we can write $D = \{R_1(a, b), R_1(b, c), R_2(a, c, d)\}$ as a shorthand for $R_1^D = \{(a, b), (b, c)\}$ and $R_2^D = \{(a, c, d)\}$. Note that the active domain of D is precisely the set of constants occurring in $\{R(\bar{a}) \mid R \in \mathbf{S} \text{ and } \bar{a} \in R^D\}$.

Named versus Unnamed Perspective

There is clearly a close connection between the two perspectives, which is not surprising since both are mathematical abstractions of the same concept. A (possibly infinite) database of a named schema can be transformed into a semantically equivalent one of an unnamed schema, and vice versa. By semantically equivalent, we mean databases that are essentially the same modulo representation details. It is instructive to properly formalize this connection, which will be used throughout the book. We do this for databases, but the exact same constructions work also for possibly infinite databases.

From Named to Unnamed. Consider a named schema $\mathbf{S}$, and assume that there is an ordering $\lessdot$ on the set of relation-attribute pairs $\{(R, A) \mid R \in \mathrm{Dom}(\mathbf{S})$ and $A \in \mathbf{S}(R)\}$. We define the unnamed schema $\mathbf{S}' : \mathsf{Rel} \to \mathbb{N}$ as follows: $\mathrm{Dom}(\mathbf{S}') = \mathrm{Dom}(\mathbf{S})$, and $\mathbf{S}'(R) = \mathrm{ar}_{\mathbf{S}}(R)$ for every $R \in \mathrm{Dom}(\mathbf{S})$. Moreover, for every database D of $\mathbf{S}$, a semantically equivalent database $D' : \mathrm{Dom}(\mathbf{S}') \to \mathsf{uRl}$ of $\mathbf{S}'$ is defined as follows: for every $R \in \mathrm{Dom}(\mathbf{S}')$,

$$\begin{aligned} D'(R) \;=\; \{(a_1, \ldots, a_k) \mid (A_1 \colon a_1, \ldots, A_k \colon a_k) \in D(R) \\ \text{such that } (R, A_1) \lessdot (R, A_2) \lessdot \cdots \lessdot (R, A_k)\}. \end{aligned}$$

From Unnamed to Named. Consider an unnamed database schema $\mathbf{S}$. We assume that Att contains an attribute name $\#_i$ for each $i \geq 1$. We define the named schema $\mathbf{S}' : \mathsf{Rel} \to \mathcal{P}_{\mathrm{fin}}(\mathsf{Att})$ as follows: $\mathrm{Dom}(\mathbf{S}') = \mathrm{Dom}(\mathbf{S})$, and $\mathbf{S}'(R) = \{\#_1, \ldots, \#_{\mathrm{ar}_{\mathbf{S}}(R)}\}$ for every $R \in \mathrm{Dom}(\mathbf{S})$. Moreover, for every database D of $\mathbf{S}$, a semantically equivalent database $D' : \mathrm{Dom}(\mathbf{S}') \to \mathsf{nRl}$ of $\mathbf{S}'$ is defined as follows: for every $R \in \mathrm{Dom}(\mathbf{S}')$,

$$D'(R) \;=\; \{(\#_1 \colon a_1, \ldots, \#_k \colon a_k) \mid (a_1, \ldots, a_k) \in D(R)\}\,.^{1}$$

Since the above connection between the two perspectives is useful in many places in the book, we assume from now on that, whenever a named database schema is used, the ordering $\lessdot$ on relation-attribute pairs is available.

The unnamed perspective is usually mathematically more elegant, while the named perspective is closer to practice. Therefore, we often define notions in the book using the unnamed perspective, but illustrate them with examples using the named perspective. When we do so, we use the following convention. When we denote a relation name as $R[A, B, \ldots]$ of a named database schema $\mathbf{S}$ in an example, we assume that the ordering of attributes in $\mathbf{S}$ is consistent with how we write it in the example, that is, $(R, A) \lessdot (R, B)$, etc. This allows us to easily switch between the named and unnamed perspective in examples, e.g., by being able to say that the "first" attribute of R is A.

Queries and Query Languages

Queries will appear throughout the book as both *semantic* and *syntactic* objects. As a semantic object, a query q over a schema $\mathbf{S}$ is a function that maps databases of $\mathbf{S}$ to *finite* sets of tuples of the same arity over Const.

Definition 2.5: Queries and Query Languages

Consider a database schema $\mathbf{S}$. A *query of arity* $k \geq 0$ (or simply a *k-ary query*) over $\mathbf{S}$ is a function of the form

$$q \;:\; \mathrm{Inst}(\mathbf{S}) \to \mathcal{P}_{\mathrm{fin}}(\mathsf{Const}^k).$$

A *query language* is a set of queries.

An important subject, which will be considered in the book, is to classify query languages according to their expressive power. Two query languages $\mathcal{L}_1$ and $\mathcal{L}_2$ are *equally expressive* if $\mathcal{L}_1 = \mathcal{L}_2$. Furthermore, $\mathcal{L}_1$ is *more expressive* than $\mathcal{L}_2$ if $\mathcal{L}_2 \subseteq \mathcal{L}_1$, and $\mathcal{L}_1$ is *strictly more expressive* than $\mathcal{L}_2$ if $\mathcal{L}_2 \subsetneq \mathcal{L}_1$.

Of course, queries as semantic objects must be given in some syntax. The syntax of queries could be SQL, relational algebra, first-order logic, and Datalog, to name a few. We proceed to explain some of our notational conventions for queries. For the sake of the discussion, we focus on query languages that are based on logic. To this end, we assume a countably infinite set

$$\mathsf{Var} \text{ of } \textit{variables},$$

[1] Notice that under the assumption that $(R, \#_i) \lessdot (R, \#_{i+1})$ for every relation name $R \in \mathbf{S}$ and $i \in [\mathbf{S}(R) - 1]$, one can translate a database D from the unnamed perspective to the named perspective and back, and obtain D again.

disjoint from Const, Rel, and Att. If φ is a logical formula and $\bar{x} = (x_1, \ldots, x_k) \in \mathsf{Var}^k$ is a tuple of variables, we will denote queries as $\varphi(\bar{x})$. We will also use a letter such as q to refer to the entire query, that is, $q = \varphi(\bar{x})$. The purpose of $\bar{x}$ is to make clear what is the *output* of the query; we will also write $q(\bar{x})$ to emphasise that q has the output tuple $\bar{x}$. More precisely, we will always define for a database D and tuple $\bar{a} = (a_1, \ldots, a_k) \in \mathsf{Const}^k$ whether D *satisfies* φ *using the values* $\bar{a}$, denoted by $D \models \varphi(\bar{a})$. Then, with the syntactic object $q = \varphi(\bar{x})$, we associate a semantic object that produces an *output*, i.e., a set of k-ary tuples over Const, for each database D, defined as:

$$q(D) \;=\; \{\bar{a} \in \mathsf{Const}^k \mid D \models \varphi(\bar{a})\}\,.$$

This semantic object will always be a query in the sense of Definition 2.5. In other words, we will use the letter q to refer to both

- the syntactic object denoting a query (for example, a logical formula together with an output tuple), and
- the query itself (i.e., the function that maps databases to finite sets of tuples of the same arity over Const).

A query of arity 0 is called *Boolean*. In this case, there are only two possible outputs: either the singleton set $\{()\}$ containing the empty tuple, or the empty set $\{\}$. We interpret $\{()\}$ as the Boolean value `true`, and $\{\}$ as `false`. For readability, we write $q(D) = \texttt{true}$ in place of $q(D) = \{()\}$, and $q(D) = \texttt{false}$ in place of $q(D) = \{\}$. When denoting Boolean queries, we will often omit the empty tuple () from the notation, i.e., write $q = \varphi$ instead of $q = \varphi()$.

A useful notion that we will use throughout the book, and, in particular, for defining the syntax of query languages that are based on logic, is that of relational atom. When R is a k-ary relation symbol and $\bar{u} \in (\mathsf{Const} \cup \mathsf{Var})^k$, $R(\bar{u})$ is a *relational atom*. Observe that the only difference between a fact and a relational atom is that the former mentions only constants, whereas the latter can mention both constants and variables. As for facts, since a relational atom is always a statement about a single tuple, we simply write $R(u_1, \ldots, u_k)$ instead of $R((u_1, \ldots, u_k))$. Given a set of atoms S, we write $\mathrm{Dom}(S)$ for the set of constants and variables in S. For example, $\mathrm{Dom}(\{R(a, x, b), R(x, a, y)\}) = \{a, b, x, y\}$. We also write R^S for the set of tuples $\{\bar{u} \mid R(\bar{u}) \in S\}$.

Key Problems: Query Evaluation and Query Analysis

Much of what we do in databases boils down to running queries on a database, or statically analyzing queries. The latter is the basis of query optimization: we need to be able to reason about queries, and to be able to replace a query with a better behaved one that has the same output. We proceed to introduce the main algorithmic problems associated with the above tasks. In their most common form, they are parameterized by a query language $\mathcal{L}$.

Query Evaluation

We start with the *query evaluation problem*, or simply the *evaluation problem*, that has the following form:

Problem: $\mathcal{L}$-Evaluation

Input: A query q from $\mathcal{L}$, a database D, a tuple $\bar{a}$ over Const
Output: `true` if $\bar{a} \in q(D)$, and `false` otherwise

Note that the evaluation problem is presented as a *decision* problem, that is, a problem whose output is either `true` or `false`. Although in practice the goal is to compute the output of q on D, in the study of the principles of databases we are mainly interested in understanding the inherent complexity of a query language. This can be achieved by studying the complexity of the decision version of the evaluation problem, which in turn allows us to employ well established tools from complexity theory such as the standard complexity classes that can be found in Appendix B.

The complexity of the problem as stated above is referred to as *combined complexity* of query evaluation. The term combined reflects the fact that both the query q and the database D are part of the input.

Very often we shall deal with a different kind of complexity of query evaluation, where the query q is *fixed*. This is referred to as *data complexity* since we measure the complexity only in terms of the size of the database D, which in practice, almost invariably, is much bigger than the size of the query q. More precisely, when we talk about data complexity, we are actually interested in the complexity of the problem q-Evaluation for some query q:

Problem: q-Evaluation

Input: A database D, and a tuple $\bar{a}$ over Const
Output: `true` if $\bar{a} \in q(D)$, and `false` otherwise

Thus, when we talk about the data complexity of $\mathcal{L}$-Evaluation, we actually refer to a family of problems, one for each query q from $\mathcal{L}$. Nonetheless, we shall apply the standard notions of complexity theory, such as membership in a complexity class, or hardness and completeness for a class, to data complexity. We proceed to precisely explain what we mean by that.

Definition 2.6: Data Complexity

Let $\mathcal{L}$ be a query language, and $\mathcal{C}$ a complexity class. $\mathcal{L}$-Evaluation is

- *in $\mathcal{C}$ in data complexity* if, for every q from $\mathcal{L}$, q-Evaluation is in $\mathcal{C}$,

- $\mathcal{C}$-*hard in data complexity* if there exists a query q from $\mathcal{L}$ such that q-Evaluation is $\mathcal{C}$-hard, and
- $\mathcal{C}$-*complete in data complexity* if $\mathcal{L}$-Evaluation is in $\mathcal{C}$ in data complexity, and $\mathcal{C}$-hard in data complexity.

To reiterate, as we shall use these concepts many times in this book:

Combined Complexity of query evaluation refers to the complexity of the $\mathcal{L}$-Evaluation problem when all of q, D, and $\bar{a}$ are inputs, and

Data Complexity refers to the complexity of $\mathcal{L}$-Evaluation when its input consists only of D and $\bar{a}$, whereas q is fixed. In other words, it refers to the complexity of the family of problems $\{q\text{-Evaluation} \mid q \text{ is a query from } \mathcal{L}\}$ in the sense of Definition 2.6.

Query Containment and Equivalence

The basis of static analysis of queries is the *containment* problem. We say that a query q is *contained* in a query q', written as $q \subseteq q'$, if $q(D) \subseteq q'(D)$ for every database D; note that since queries return sets of tuples, the notion of subset is applicable to query outputs. This is the most basic task of reasoning about queries; note that containment is one part of equivalence. Indeed, q is *equivalent* to q', denoted $q \equiv q'$, if $q \subseteq q'$ and $q' \subseteq q$. The equivalence problem is the most basic one in query optimization, whose goal is to transform a query q into an equivalent, and more efficient, query q'.

In relation to containment and equivalence, we consider the following decision problems, again parameterized by a query language $\mathcal{L}$.

Problem: $\mathcal{L}$-Containment

Input: Two queries q and q' from $\mathcal{L}$
Output: `true` if $q \subseteq q'$, and `false` otherwise

Problem: $\mathcal{L}$-Equivalence

Input: Two queries q and q' from $\mathcal{L}$
Output: `true` if $q \equiv q'$, and `false` otherwise

Observe that for the previous problems, the input consists of two queries. Typically, queries are much smaller objects than databases. Therefore, for the containment and equivalence problems, we shall in general tolerate higher complexity than for query evaluation; even intractable complexity will often be reasonable, given the small size of the input.

Analyzing Computational Complexity

We will use two different cost models for analyzing the computational complexity of problems.

Turing Machine models are typically associated with complexity classes such as PTime, NP, PSpace, Space($\log n$), Π_2^p, etc.

Random-Access Machine models are usually used for analyzing the runtime of efficient algorithms. For instance, if we say that n numbers can be sorted in time $O(n \log n)$, then the intended underlying computational model is a random-access machine model.

Throughout the book, we will assume the Turing Machine model for analyzing the complexity of problems in terms of complexity classes, whereas we will assume random-access models when it comes to proving that algorithms have low complexity. The difference between the two will always be clear in the formal statement, where we will always either

- refer to a concrete complexity class, such as PTime, NLogSpace, or DLogSpace,[2] in which case we assume Turing Machines; or
- we say that the problem "is solvable in" time $O(f(n))$ or space $O(f(n))$ for some function f, in which case we assume a random access machine model.

Size of the Input

The complexity of algorithms is always analyzed in terms of the size of their input. To this end we will define, throughout the book the size $\|o\|$ of objects o that we will consider for complexity analysis. We define the following.

- $\|\emptyset\| = \|()\| = 1$.
- $\|u\| = 1$ for each $u \in \mathsf{Const} \cup \mathsf{Var}$.
- For a nonempty set $S = \{e_1, \ldots, e_n\}$, we define $\|S\| = \sum_{i=1}^{n} \|e_i\|$.
- For a tuple $\bar{u} = (u_1, \ldots, u_k)$ or a fact $R(\bar{u}) = R(u_1, \ldots, u_k)$ with $k \geq 1$, we define $\|\bar{u}\| = \|R(\bar{u})\| = \sum_{i=1}^{k} \|u_i\|$.

Therefore, if D is a nonempty database instance of schema $\mathbf{S} = \{R_1, \ldots, R_n\}$, then

$$\|D\| = \sum_{i=1}^{n} \left(|D(R_i)| \cdot (\mathrm{ar}(R_i)) \right) ,$$

[2] In this book, we usually denote complexity classes in small caps, see Appendix B. Exceptions to this rule are well-known complexity classes for which fonts are irrelevant, such as Π_2^p and #P.

assuming that the arities of $R_1, \ldots, R_n$ are nonzero.

Turing Machines and random-access machines perceive their inputs differently. Whereas a random-access machine can store a natural number $n \in \mathbb{N}$ in a single register, a Turing Machine will store n as a word of $O(\log n)$ symbols from its finite alphabet. In Appendix C, we discuss how databases and queries are encoded for Turing Machines. For instance, storing a database D on a Turing Machine costs space $O(\|D\| \cdot \log \|D\|)$. Intuitively, the encoding uses $O(\|D\|)$ many constants, and we need $O(\log \|D\|)$ space to encode each such constant using the Turing Machine's finite alphabet.

Since it is well known that the random access model and the Turing Machine model are equally efficient as long as polynomial differences do not matter, we sometimes do a random-access-style analysis for easier presentation, even if the underlying computational model is a Turing Machine.

Further Background Reading

Should the reader find herself/himself in a situation "that she/he does not have the prerequisites for reading the prerequisites" [7], rather than being discouraged she/he is advised to continue with the main material, as it is still very likely to be understood completely or almost completely. Should the latter happen, the prerequisites can be supplemented by information from many standard sources, some of which are listed below.

The book [1] covers the basics of database theory, while many database systems texts cover design, querying, and building real-life databases, for example, [5, 13, 15]. The basic mathematical background needed is covered in a standard undergraduate "discrete mathematics for computer science" course; moreover, a good source for this material is the book [14]. For additional information about computability theory, we provide a primer in Appendix B. Furthermore, we refer the reader to [8, 10, 16]; standard texts on complexity theory are [2, 12, 18]. For the foundations of finite model theory and descriptive complexity, the reader is referred to [6, 9, 11].

Part I

The Relational Model: The Classics

3

First-Order Logic

Database query languages are either *declarative* or *procedural*. In a declarative language, one provides a specification of what a query result should be, typically by means of logical formulae (sometimes presented in a specialized programming syntax). In the case of relational databases, such languages are usually based on *first-order logic*, which often appears in the literature under the name *relational calculus*. In a procedural language, on the other hand, one specifies how the data is manipulated to produce the desired result. The most commonly used one for relational databases is *relational algebra*. We present these languages next, starting with first-order logic.

Syntax of First-Order Logic

Recall that a schema $\mathbf{S}$ can be seen as a finite set of relation names, and each relation name of $\mathbf{S}$ has an arity under $\mathbf{S}$. Recall also that we assume a countably infinite set of values Const called constants, and a countably infinite set of variables Var. Constants will be typically denoted by $a, b, c, \ldots$, and variables by $x, y, z, \ldots$ (possibly with subscripts and superscripts). Constants and variables are called *terms*. Formulae of first-order logic are inductively defined using terms, conjunction ($\wedge$), disjunction ($\vee$), negation ($\neg$), existential quantification ($\exists$), and universal quantification ($\forall$).

Definition 3.1: Syntax of First-Order Logic

We define *formulae of first-order logic* (FO) over a schema $\mathbf{S}$ as follows:

- If a is a constant from Const, and x, y are variables from Var, then $x = a$ and $x = y$ are atomic formulae.
- If $u_1, \ldots, u_k$ are terms (not necessarily distinct), and R is a k-ary relation name from $\mathbf{S}$, then $R(u_1, \ldots, u_k)$ is an atomic formula.

- If φ_1 and φ_2 are formulae, then $(\varphi_1 \wedge \varphi_2)$, $(\varphi_1 \vee \varphi_2)$, and $(\neg\varphi_1)$ are formulae.
- If φ is a formula and $x \in \mathsf{Var}$, then $(\exists x\, \varphi)$ and $(\forall x\, \varphi)$ are formulae.

The *size* $\|\varphi\|$ of φ is defined to be the total number of constants, variables, and symbols from $\{\wedge, \vee, \neg, =, \exists, \forall\}$ occurring in φ. For example, the size of $(x = a \ \vee \ x = b)$ is seven.

Formulae of the form $x = a$ and $x = y$ are called *equational atoms*. Furthermore, as already mentioned in Chapter 2, formulae of the form $R(\bar{u})$ are called *relational atoms*. Note that we allow repetition of variables in relational atoms, for example, we may write $R(x, x, y)$. We shall use the standard shorthand $(\varphi \to \psi)$ for $((\neg\varphi) \vee \psi)$ and $(\varphi \leftrightarrow \psi)$ for $((\varphi \to \psi) \wedge (\psi \to \varphi))$. To reduce notational clutter, we will often omit the outermost brackets of formulae.

A crucial notion is that of free variables of a formula, which are essentially the variables in a formula that are not quantified. Given an FO formula φ, the set of *free* variables of φ, denoted $\mathrm{FV}(\varphi)$, is inductively defined as follows:

- $\mathrm{FV}(x = y) = \{x, y\}$.
- $\mathrm{FV}(x = a) = \{x\}$.
- $\mathrm{FV}(R(u_1, \ldots, u_k)) = \{u_1, \ldots, u_k\} \cap \mathsf{Var}$.
- $\mathrm{FV}(\varphi_1 \vee \varphi_2) = \mathrm{FV}(\varphi_1 \wedge \varphi_2) = \mathrm{FV}(\varphi_1) \cup \mathrm{FV}(\varphi_2)$.
- $\mathrm{FV}(\neg\varphi) = \mathrm{FV}(\varphi)$.
- $\mathrm{FV}(\exists x\, \varphi) = \mathrm{FV}(\forall x\, \varphi) = \mathrm{FV}(\varphi) - \{x\}$.

If $x \in \mathrm{FV}(\varphi)$, we call it a *free variable (of φ)*; otherwise, x is called *bound*. An FO formula φ without free variables is called a *sentence*.

Example 3.2: First-Order Formulae

Consider the following (named) database schema:

```
Person [ pid, pname, cid ]
Profession [ pid, prname ]
City [ cid, cname, country ]
```

The Person relation stores internal person IDs (`pid`), names (`pname`), and the ID of their city of birth (`cid`). The Profession relation contains the professions of persons by storing their person ID (`pid`) and profession name (`prname`). Finally, City contains a bit of geographic information by storing IDs (`cid`) and names (`cname`) of cities, together with the country they are located in (`country`). In what follows, we give some examples of FO formulae over this schema. Consider first the FO formula:

$$\exists y \exists z \exists u_1 \exists u_2 \big(\text{Person}(x, y, z) \wedge \text{Profession}(x, u_1) \wedge \text{Profession}(x, u_2) \wedge \neg(u_1 = u_2)\big). \quad (3.1)$$

This formula has one free variable, that is, x. Consider now the formula

$$\exists z \big(\text{Person}(x, y, z) \wedge \forall r \forall s \, (\neg \text{City}(z, r, s))\big). \quad (3.2)$$

The free variables of this formula are x, y. Finally, consider the formula

$$\exists x \exists z \big(\text{Person}(x, y, z) \wedge (\text{Profession}(x, \text{'author'}) \vee \text{Profession}(x, \text{'actor'}))\big). \quad (3.3)$$

This formula has one free variable, that is, y.

Semantics of First-Order Logic

Given a database D of a schema $\mathbf{S}$, we inductively define the notion of satisfaction of a formula φ over $\mathbf{S}$ in D with respect to an *assignment η for φ over D*. Such an assignment is a function from $\text{FV}(\varphi)$ to $\text{Dom}(D) \cup \text{Dom}(\varphi) \subseteq \mathsf{Const}$, where $\text{Dom}(\varphi)$ is the set of constants mentioned in φ. For example, for the formula $R(x, y, a)$, η is the function $\{x, y\} \to \text{Dom}(D) \cup \{a\}$. In the following definition (and also later in the book), we write $\eta[x/u]$, for a variable x and term u, for the assignment that modifies η by setting $\eta(x) = u$. Furthermore, to avoid heavy notation, we extend η to be the identity on Const.

Definition 3.3: Semantics of First-Order Logic

Given a database D of a schema $\mathbf{S}$, a formula φ over $\mathbf{S}$, and an assignment η for φ over D, we inductively define when φ is *satisfied* in D under η, written $(D, \eta) \models \varphi$, as follows:

- If φ is $x = y$, then $(D, \eta) \models \varphi$ if $\eta(x) = \eta(y)$.
- If φ is $x = a$, then $(D, \eta) \models \varphi$ if $\eta(x) = a$.
- If φ is $R(u_1, \ldots, u_k)$, then $(D, \eta) \models \varphi$ if $R(\eta(u_1), \ldots, \eta(u_k)) \in D$.
- If $\varphi = \varphi_1 \wedge \varphi_2$, then $(D, \eta) \models \varphi$ if $(D, \eta) \models \varphi_1$ and $(D, \eta) \models \varphi_2$.
- If $\varphi = \varphi_1 \vee \varphi_2$, then $(D, \eta) \models \varphi$ if $(D, \eta) \models \varphi_1$ or $(D, \eta) \models \varphi_2$.
- If $\varphi = \neg\psi$, then $(D, \eta) \models \varphi$ if $(D, \eta) \models \psi$ does not hold.
- If $\varphi = \exists x \, \psi$, then $(D, \eta) \models \varphi$ if $(D, \eta[x/a]) \models \psi$ for *some* constant $a \in \text{Dom}(D) \cup \text{Dom}(\varphi)$.
- If $\varphi = \forall x \, \psi$, then $(D, \eta) \models \varphi$ if $(D, \eta[x/a]) \models \psi$ for *each* constant $a \in \text{Dom}(D) \cup \text{Dom}(\varphi)$.

An assignment η for a sentence φ has an empty domain (since the domain of η is $\mathrm{FV}(\varphi)$), and thus it is unique. For this unique η, it is either the case that $(D, \eta) \models \varphi$ or not. If the former is true, then we simply write $D \models \varphi$ and say that D *satisfies* φ.

Example 3.4: Semantics of First-Order Formulae

We provide an intuitive description of the semantic meaning of the formulae given in Example 3.2:

- Formula (3.1) is satisfied by all x such that x is the ID of a person with two different professions.
- Formula (3.2) is satisfied by all x, y such that x and y are the ID and name of persons for which their city of birth is not in the database.
- Formula (3.3) is satisfied by all y such that y is the name of a person who is an author or an actor.

It is crucial to say that the semantics of FO are defined in a way that is well-suited for database applications, but slightly departs from the logic literature. In particular, the range of quantifiers is the set of constants $\mathrm{Dom}(D) \cup \mathrm{Dom}(\varphi)$ (see the last two items of Definition 3.3), whereas in the standard definition is the set of values Const. This is why η associates elements of $\mathrm{Dom}(D) \cup \mathrm{Dom}(\varphi)$ to variables, while in the standard definition one would allow η to associate arbitrary elements of Const to variables. The set $\mathrm{Dom}(D) \cup \mathrm{Dom}(\varphi)$ is called the *active domain of* D *and* φ. Therefore, Definition 3.3 actually defines the so-called *active domain semantics*, which is standard in the database literature. The importance of the active domain semantics is revealed below where we use FO to define database queries.

Notational Conventions

We introduce some notational conventions concerning FO formulae that would significantly improve readability:

- Since conjunction is associative, we will omit brackets in long conjunctions and write, for example, $x_1 \wedge x_2 \wedge x_3 \wedge x_4$ instead of $((x_1 \wedge x_2) \wedge x_3) \wedge x_4$. We follow the same convention for disjunction. We also omit brackets within sequences of quantifiers.
- We often write $\exists \bar{x}\, \varphi$ for $\exists x_1 \exists x_2 \ldots \exists x_m\, \varphi$, where $\bar{x} = (x_1, \ldots, x_m)$, and likewise for universal quantifiers $\forall \bar{x}$.
- We assume that $\neg$ binds the strongest, followed by $\wedge$, then $\vee$, and finally quantifiers. For example, by $\exists x\, \neg R(x) \wedge S(x)$ we mean the formula $\exists x\, ((\neg R(x)) \wedge S(x))$. We will, however, add brackets to formulae when

we feel that it improves their readability. Notice that this precedence of operators also influences the range of variables; e.g., by $\forall x\, R(x) \wedge S(x)$ we mean the formula $\forall x\, (R(x) \wedge S(x))$, as opposed to $(\forall x\, R(x)) \wedge S(x)$.

- Finally, we write $x \neq y$ instead of $\neg(x = y)$, and likewise for $(x = a)$.

Equivalences

In the way FO is defined in Definition 3.1, some constructors are redundant. For instance, we know by De Morgan's laws that $\neg(\varphi \vee \psi)$ is equivalent to $\neg\varphi \wedge \neg\psi$, and $\neg(\varphi \wedge \psi)$ is equivalent to $\neg\varphi \vee \neg\psi$. Furthermore, the formula $\neg\forall x\, \varphi$ is equivalent to $\exists x\, \neg\varphi$ and $\neg\exists x\, \varphi$ is equivalent to $\forall x\, \neg\varphi$. These equivalences mean that the full set of Boolean connectives and quantifiers is not necessary to define all of FO. For example, one can just use $\vee, \neg$, and $\exists$, or $\wedge, \neg$, and $\exists$, and this will capture the full expressive power of FO. This is useful for proofs that proceed by induction on the structure of FO formulae.

For some proofs in Part I of the book it will be convenient to assume that constants do not appear in relational atoms. We can always rewrite FO formulae to such a form via equalities, at the expense of a linear blow-up. For instance, we can write $R(x, a, b)$ as $\exists x_a \exists x_b\, R(x, x_a, x_b) \wedge (x_a = a) \wedge (x_b = b)$.

First-Order Queries

Recall that a k-ary query q produces a finite set of k-ary tuples $q(D) \subseteq \mathsf{Const}^k$, for every database D. FO formulae can be used to define database queries. In order to do this, we specify together with the formula φ a tuple $\bar{x}$ of variables that indicates how the output of the query is formed. As a simple example, consider an atomic formula $\varphi = R(x, y)$ and the tuple (x, y). Then the query $\varphi(x, y)$ would return the entire relation R from the database. Notice that the query is actually $R(x, y)(x, y)$, where the first occurrence of (x, y) is part of the relational atom $R(x, y)$, and the second occurrence specifies how the output of the query is formed. To consider a few other examples, if $\varphi = R(x, y)$, then the query $\varphi(x, x, y)$ returns all tuples (a, a, b) such that (a, b) is in the relation R. Finally, if $\varphi = R(x, x)$, then the query $\varphi(x)$ returns all tuples (a) such that (a, a) is in the relation R. The definition of FO queries follows.

Definition 3.5: First-Order Queries

A *first-order query* over a schema $\mathbf{S}$ is an expression of the form $\varphi(\bar{x})$, where φ is an FO formula over $\mathbf{S}$, and $\bar{x}$ is a tuple of free variables of φ such that each free variable of φ occurs in $\bar{x}$ at least once.

We define *size* $\|\varphi(\bar{x})\|$ of a first-order query $\varphi(\bar{x})$ as $\|\varphi\| + \|\bar{x}\|$.

Let $\varphi(\bar{x})$ be an FO query over $\mathbf{S}$. Given a database D of $\mathbf{S}$, and a tuple $\bar{a}$ of elements from Const, we say that D *satisfies the query* $\varphi(\bar{x})$ *using the values* $\bar{a}$, denoted by $D \models \varphi(\bar{a})$, if there exists an assignment η for φ over D such that $\eta(\bar{x}) = \bar{a}$ and $(D, \eta) \models \varphi$. Having this notion in place, we can now define what is the output of an FO query on a database.

Definition 3.6: Evaluation of First-Order Queries

Given a database D of a schema $\mathbf{S}$, and an FO query $q = \varphi(x_1, \ldots, x_k)$ over $\mathbf{S}$, where $k \geq 0$, the *output* of q on D is defined as the set of tuples

$$q(D) \;=\; \{\bar{a} \in \mathsf{Const}^k \mid D \models \varphi(\bar{a})\}.$$

It is clear that $q(D) \in \mathcal{P}(\mathsf{Const}^k)$. However, to be able to say that q defines a k-ary query over $\mathbf{S}$ in the sense of Definition 2.5, we need to ensure that $q(D) \in \mathcal{P}_{\text{fin}}(\mathsf{Const}^k)$, i.e., the output of q on D is finite. This is guaranteed by the following result, which is an immediate consequence of the active domain semantics of FO (see Definition 3.3).

Proposition 3.7

Given a database D of a schema $\mathbf{S}$, and an FO query $q = \varphi(x_1, \ldots, x_k)$ over $\mathbf{S}$, where $k \geq 0$, it holds that

$$q(D) \;=\; \{\bar{a} \in (\mathrm{Dom}(D) \cup \mathrm{Dom}(\varphi))^k \mid D \models \varphi(\bar{a})\}.$$

Since, by definition, the set of values $\mathrm{Dom}(D) \cup \mathrm{Dom}(\varphi)$ is finite, Proposition 3.7 implies that $q(D) \in \mathcal{P}_{\text{fin}}(\mathsf{Const}^k)$, and thus, q defines a k-ary query over $\mathbf{S}$ in the sense of Definition 2.5.

Before we proceed further, let us stress that if we adopt the standard semantics of FO from logic textbooks, which uses assignments η that associate arbitrary elements of Const to variables, then there is no guarantee that $q(D)$ is finite. Consider, for example, the query $q = \varphi(x)$ with $\varphi = \neg R(x)$, and the database $D = \{R(a), P(b)\}$. Under the standard FO semantics, the output of q on D would be the set $\{(c) \mid c \in \mathsf{Const} - \{a\}\}$, and thus infinite. On the other hand, under the active domain semantics we have that $q(D) = \{(b)\}$.

Example 3.8: Evaluation of First Order Queries

A database D of the schema in Example 3.2 is depicted in Figure 3.1. We proceed to evaluate the FO queries obtained from the FO formulae given in Example 3.2 on D:

- Let q_1 be the query $\varphi_1(x)$, where φ_1 is the formula (3.1). Then

$$q_1(D) \;=\; \{(\text{'1'}), (\text{'3'}), (\text{'4'})\}.$$

Person

pid	pname	cid
1	Aretha	MPH
2	Billie	BLT
3	Bob	DLT
4	Freddie	ST

Profession

pid	prname
1	singer
1	songwriter
1	actor
2	singer
3	singer
3	songwriter
3	author
4	singer
4	songwriter

City

cid	cname	country
MPH	Memphis	United States
DLT	Duluth	United States
ST	Stone Town	Tanzania

Fig. 3.1: A database of the schema in Example 3.2.

- Let q_2 be the query $\varphi_2(x, y)$, where φ_2 is the formula (3.2). Then

$$q_2(D) \;=\; \{(\text{`2'}, \text{`Billie'})\}.$$

- Let q_3 be the query $\varphi_3(y)$, where φ_3 is the formula (3.3). Then

$$q_3(D) \;=\; \{(\text{`Aretha'}), (\text{`Bob'})\}.$$

Boolean First-Order Queries

FO sentences, that is, FO formulae without free variables, are used to define Boolean queries, i.e., queries that return `true` or `false`, and hence the name *Boolean FO queries*. By definition, the output of a query q on a database D corresponds to a set of tuples, and thus, Boolean FO queries will be no exception to this. We consider such queries to be of the form $q = \varphi()$, where φ is an FO sentence, and $()$ denotes the empty tuple. There are two cases:

- either $q(D)$ consists of the empty tuple, that is, $q(D) = \{()\}$, which happens precisely when $D \models \varphi$, or
- $q(D)$ is the empty set, which happens precisely when $D \models \neg\varphi$.

By convention, we write $q(D) = \texttt{true}$ if $D \models \varphi$, and $q(D) = \texttt{false}$ otherwise.

4

Relational Algebra

Queries expressed in FO are *declarative* and tell us *what* the output of a query should be. In this chapter, we introduce *relational algebra*, abbreviated RA, which contrasts itself with FO because it is *procedural*, i.e., it specifies *how* the output of queries can be obtained via a sequence of operations on the data. Relational algebra is of significant practical importance in databases, since database systems typically use relational-algebra-like representations of queries to do query optimization, that is, to discover methods in which a given query can be evaluated efficiently.

We present relational algebra in its most elementary form, in both the unnamed and the named perspective. The following table gives a quick overview of the operators in the unnamed and named relational algebra.

Operator Name	*(Unnamed) RA Symbol*	*Named RA Symbol*
selection	σ_θ	σ_θ
projection	π_α	π_α
Cartesian product	$\times$	
rename		ρ
union	$\cup$	$\cup$
difference	$-$	$-$
join	$\bowtie_\theta$	$\bowtie$

We explain these operators and their semantics next, in the definitions of the *unnamed* and *named RA*. Since we will usually be working with the unnamed perspective in this book, we will often abbreviate "unnamed RA" as "RA".

Syntax of the Unnamed Relational Algebra

Under the unnamed perspective, RA consists of five primitive operations: selection, projection, Cartesian product, union, and difference. Before giving the

formal definitions of those operations, we first introduce the notion of condition over a set of integers that is needed for defining the selection operation. A *condition* θ over $\{1, \ldots, k\}$, for some $k \geq 0$, is a Boolean combination of statements of the form $i \doteq j$, $i \doteq a$, $i \not\doteq j$, and $i \not\doteq a$, where for $a \in \mathsf{Const}$ and $i, j \in [k]$. Intuitively, a condition $i \doteq j$ is used to indicate that in a tuple the values of the i-th attribute and the j-th attribute must be the same, while $i \not\doteq j$ is used to indicate that these values must be different. Moreover, a condition $i \doteq a$ is used to indicate that in a tuple the value of the i-th attribute must be the constant a, while $i \not\doteq a$ is used to indicate that this value must be different than a. Let us clarify that we use the symbols $\doteq$ and $\not\doteq$, instead of $=$ and $\neq$, to avoid writing statements such as "$1 = 2$", which are likely to confuse the reader. Notice that by using De Morgan's laws to propagate negation, we can define conditions as *positive* Boolean combinations of statements $i \doteq j$ and $i \not\doteq j$, i.e., Boolean combinations using only conjunction $\wedge$ and disjunction $\vee$. For example, $\neg\big((1 \doteq 2) \vee (2 \not\doteq 3)\big)$ is equivalent to $(1 \not\doteq 2) \wedge (2 \doteq 3)$.

Definition 4.1: Syntax of Unnamed Relational Algebra

We inductively define *RA expressions* over a schema $\mathbf{S}$, and their associated arities, as follows:

Base Expressions. If R is a k-ary relation name from $\mathbf{S}$, then R is an atomic RA expression over $\mathbf{S}$ of arity k. If $a \in \mathsf{Const}$, then $\{a\}$ is an RA expression over $\mathbf{S}$ of arity 1.

Selection. If e is an RA expression over $\mathbf{S}$ of arity $k \geq 0$ and θ is a condition over $[k]$, then $\sigma_\theta(e)$ is an RA expression over $\mathbf{S}$ of arity k.

Projection. If e is an RA expression over $\mathbf{S}$ of arity $k \geq 0$ and $\alpha = (i_1, \ldots, i_m)$, for $m \geq 0$, is a list of numbers from $[k]$, then $\pi_\alpha(e)$ is an RA expression over $\mathbf{S}$ of arity m.

Cartesian Product. If e_1, e_2 are RA expressions over $\mathbf{S}$ of arity $k \geq 0$ and $m \geq 0$, respectively, then their Cartesian product $(e_1 \times e_2)$ is an RA expression over $\mathbf{S}$ of arity $k + m$.

Union. If e_1, e_2 are RA expressions over $\mathbf{S}$ of the same arity $k \geq 0$, then their union $(e_1 \cup e_2)$ is an RA expression over $\mathbf{S}$ of arity k.

Difference. If e_1, e_2 are RA expressions over $\mathbf{S}$ of the same arity $k \geq 0$, then their difference $(e_1 - e_2)$ is an RA expression over $\mathbf{S}$ of arity k.

The *size* $\|e\|$ of RA expression e is the total number of occurrences of relation names, constants, natural numbers, and symbols from $\{\sigma, \pi, \times, \cup, -, \wedge, \vee, \neg, \doteq, \not\doteq\}$ in e. For instance, the size of $\sigma_{1 \doteq 2}(\pi_{(1,2)}(R))$ is eight.

Notice that in the definition of the projection operation, we allow m to be 0, in which case the list of integers $\alpha = (i_1, \ldots, i_m)$ is the empty list $()$. This is useful for expressing Boolean queries.

Semantics of Unnamed Relational Algebra

We proceed to define the semantics of RA expressions. We first need to define the operation of projection over tuples. For a tuple $\bar{a} = (a_1, \ldots, a_k) \in \mathsf{Const}^k$, and a list $\alpha = (i_1, \ldots, i_m)$ of numbers from $[k]$, the projection $\pi_\alpha(\bar{a})$ is defined as the tuple $(a_{i_1}, a_{i_2}, \ldots, a_{i_m})$.[1] Here are some simple examples:

$$\pi_{(1,3)}(a,b,c,d) = (a,c) \qquad \pi_{(1,3,3)}(a,b,c,d) = (a,c,c) \qquad \pi_{()}(a,b,c,d) = ().$$

We also need the notion of satisfaction of conditions over tuples. We inductively define when a tuple $\bar{a}$ *satisfies the condition* θ, denoted $\bar{a} \models \theta$:

$$\begin{array}{ll} \bar{a} \models i \doteq j & \text{if } a_i = a_j \\ \bar{a} \models i \not\doteq j & \text{if } a_i \neq a_j \\ \bar{a} \models \theta \wedge \theta' & \text{if } \bar{a} \models \theta \text{ and } \bar{a} \models \theta' \\ \bar{a} \models \neg\theta & \text{if } \bar{a} \models \theta \text{ does not hold} \end{array} \qquad \begin{array}{ll} \bar{a} \models i \doteq a & \text{if } a_i = a \\ \bar{a} \models i \not\doteq a & \text{if } a_i \neq a \\ \bar{a} \models \theta \vee \theta' & \text{if } \bar{a} \models \theta \text{ or } \bar{a} \models \theta' \end{array}$$

We are now ready to define the semantics of RA expressions.

Definition 4.2: Semantics of Unnamed RA Expressions

Let D be a database of a schema $\mathbf{S}$, and e an RA expression over $\mathbf{S}$. We inductively define the *output* $e(D)$ of e on D as follows:

- If $e = R$, where R is a relation name from $\mathbf{S}$, then $e(D) = R^D$.
- If $e = \{a\}$, for $a \in \mathsf{Const}$, then $e(D) = \{a\}$.
- If $e = \sigma_\theta(e_1)$, where e_1 is an RA expression of arity $k \geq 0$ and θ is a condition over $[k]$, then $e(D) = \{\bar{a} \mid \bar{a} \in e_1(D) \text{ and } \bar{a} \models \theta\}$.
- If $e = \pi_\alpha(e_1)$, where e_1 is an RA expression of arity $k \geq 0$ and $\alpha = (i_1, \ldots, i_m)$, for $m \geq 0$, is a list of numbers from $[k]$, then $e(D)$ is the m-ary relation $\{\pi_\alpha(\bar{a}) \mid \bar{a} \in e_1(D)\}$.
- If $e = (e_1 \times e_2)$, where e_1 and e_2 are RA expressions of arity $k \geq 0$ and $\ell \geq 0$, respectively, then $e(D) = e_1(D) \times e_2(D)$.
- If $e = (e_1 \cup e_2)$, where e_1 and e_2 are RA expressions of the same arity $k \geq 0$, then $e(D) = e_1(D) \cup e_2(D)$.
- If $e = (e_1 - e_2)$, where e_1 and e_2 are RA expressions of the same arity $k \geq 0$, then $e(D) = e_1(D) - e_2(D)$.

We sometimes use derived operations, one of them of special importance:

[1] The projection $\pi_\alpha(\bar{u})$, where $\bar{u}$ is tuple from $(\mathsf{Const} \cup \mathsf{Var})^k$, is defined in the same way. For example, $\pi_{(1,3)}(a,x,y,d) = (a,y)$ and $\pi_{(1,3,3)}(a,x,y,d) = (a,y,y)$. We are going to apply the projection operator over tuples of constants and variables in subsequent chapters such as Chapters 10 and 11.

Join. Given a k-ary RA expression e_1, an m-ary RA expression e_2, and a condition θ over $\{1, \ldots, k+m\}$, the *θ-join* of e_1 and e_2 is denoted $e_1 \bowtie_\theta e_2$. Its output on a database D is defined as

$$(e_1 \bowtie_\theta e_2)(D) \;=\; \sigma_\theta(e_1(D) \times e_2(D))\,.$$

We note that RA expressions readily define queries on databases. Indeed, if e is a RA expression, then the *output* of e on a database D is $e(D)$. In the remainder of the book, we will therefore sometimes also refer to e as a *query*.

Example 4.3: Unnamed RA Queries

Consider again the (named) database schema:

```
Person [ pid, pname, cid ]
Profession [ pid, prname ]
City [ cid, cname, country ]
```

The RA expression

$$\pi_{(1)}\big(\sigma_{5\neq 7}\big((\text{Person} \bowtie_{1\doteq 4} \text{Profession}) \bowtie_{1\doteq 6} \text{Profession}\big)\big)$$

returns the IDs of persons with at least two professions. The expression

$$\pi_{(1,2)}(\text{Person}) - \pi_{(1,2)}\big(\text{Person} \bowtie_{3\doteq 4} \text{City}\big)$$

returns the ID and name of persons whose city of birth does not appear in the database. Finally, the expression

$$\pi_{(2)}\big(\sigma_{(5\doteq\text{`author'})\vee(5\doteq\text{`actor'})}(\text{Person} \bowtie_{1\doteq 4} \text{Profession})\big)$$

returns the names of persons that are author or actors.

Syntax of the Named Relational Algebra

Under the named perspective, the presentation changes a bit. Before giving the formal definition, let us first note that the notion of condition, needed for defining the selection operation, is now over a set of attributes, and not a set of integers as in the case of unnamed RA. More precisely, a *condition* θ over a set of attributes $U \subseteq \mathsf{Att}$ is a Boolean combination of statements of the form $A \doteq B$, $A \doteq a$, $A \not\doteq B$, and $A \not\doteq a$, where $a \in \mathsf{Const}$ and $A, B \in U$.

Definition 4.4: Syntax of Named Relational Algebra

We inductively define *named RA expressions* over a schema $\mathbf{S}$, and their associated sorts, as follows:

Base Expressions. If $R \in \mathbf{S}$, then R is an atomic named RA expression over $\mathbf{S}$ of sort $\mathbf{S}(R)$. If $a \in \mathsf{Const}$ and $A \in \mathsf{Att}$, then $\{(A\colon a)\}$ is a named RA expression of sort $\{A\}$.

Selection. If e is a named RA expression of sort U and θ is a *condition over* U, then $\sigma_\theta(e)$ is a named RA expression of sort U.

Projection. If e is a named RA expression of sort U and $\alpha \subseteq U$, then $\pi_\alpha(e)$ is a named RA expression of sort α.

Join. If e_1, e_2 are named RA expressions of sort U_1 and U_2, respectively, then their join $(e_1 \bowtie e_2)$ is a named RA expression of sort $U_1 \cup U_2$.

Rename. If e is a named RA expression of sort U, then $\rho_{A\to B}(e)$, where $A \in U$ and $B \in \mathsf{Att} - U$, is a named RA expression of sort $(U - \{A\}) \cup \{B\}$.

Union. If e_1, e_2 are named RA expressions of the same sort U, then their union $(e_1 \cup e_2)$ is a named RA expression of sort U.

Difference. If e_1, e_2 are named RA expressions of the same sort U, then their difference $(e_1 - e_2)$ is a named RA expression of sort U.

The *size* $\|e\|$ of a named RA expression e is the total number of occurrences of relation names, constants, attribute names, and symbols from $\{\sigma, \pi, \bowtie, \rho, \cup, -, \wedge, \vee, \neg, \doteq, \not\doteq\}$ in e. For instance, the size of $\sigma_{A\doteq B}(\pi_{(A,B)}(R))$ is eight.

Notice in the definition of the projection operation the contrast with the unnamed perspective, where α is a list of numbers with repetitions.

Semantics of the Named Relational Algebra

The semantics of named RA expressions is defined similarly to the unnamed case, with the main difference that $e(D)$ is now a named relation instance. Therefore, we only discuss rename and join, and leave the others as exercises.

Rename. If $e = \rho_{A\to B}(e_1)$, where e_1 is a named RA expression of sort U, $A \in U$, and $B \in \mathsf{Att} - U$, then $e(D)$ is the relation

$$\{t \mid t(B) = t_1(A) \text{ and } t(C) = t_1(C) \text{ for } t_1 \in e_1(D) \text{ and } C \in U - \{A\}\}.$$

Note that renaming does not change the data at all, it only changes names of attributes. Nonetheless, this operation is necessary under the named perspective. For instance, consider two relations, R and S, the former with a single attribute A and the latter with a single attribute B. Suppose we want to find their union in relational algebra. The problem is that the union is only defined if the sorts of R and S are the same, which is not the case. To take their union, we can therefore rename the attribute of S to be A, and complete the task by writing the expression $\big(R \cup \rho_{B\to A}(S)\big)$.

Join. The other new primitive operator in the named perpective is *join* (also known in the literature as *natural join*). It is simply a join of two relations on the condition that their common attributes are the same. Formally, if $e = e_1 \bowtie e_2$, where e_1 and e_2 are named RA expressions of sorts U_1 and U_2, then $e(D)$ is the set of tuples t such that

$$t(A) = \begin{cases} t_1(A) & \text{if } A \in U_1, \\ t_2(A) & \text{if } A \in U_2 - U_1, \end{cases}$$

where $t_1 \in e_1(D)$, $t_2 \in e_2(D)$, and $t_1(A) = t_2(A)$ for all $A \in U_1 \cap U_2$. To give an example, consider the relations $R[A, B]$ and $S[B, C]$. Their join $R \bowtie S$ has attributes A, B, C, and consists of triples (a, b, c) such that $R(a, b)$ and $S(b, c)$ are both facts in the database. Notice that, if R and S have no common attributes, their join is their Cartesian product. For this reason, we do not have the operator $\times$ in the named RA.

Similarly to the unnamed perspective, we can interpret named RA expressions e as queries over databases D. However, since queries return tuples in Const^k according to Definition 2.5, and since $e(D)$ is a named relation instance, we still need to explain how we go from $e(D)$ to a finite set of tuples over Const. To this end, we will assume that the order $\lessdot$ that we introduced in Chapter 2 for translating between databases from the named to the unnamed perspective, is also an order on Att, i.e., we assume that it is an oder on the set $\mathsf{Att} \cup (\mathsf{Rel} \times \mathsf{Att})$.[2] We can now associate to e a query q_e by defining that, on database D, the *output of* q_e *on* D is the set

$$q_e(D) \;=\; \{(a_1, \ldots, a_k) \mid (A_1 \colon a_1, \ldots, A_k \colon a_k) \in e(D) \text{ such that } A_1 \lessdot A_2 \lessdot \cdots \lessdot A_k\}.$$

In the remainder of the book, we will usually not formally distinguish between the RA expression e and the query q_e. In particular, if we talk about the *query* e, then we mean the query q_e that we just defined.

Example 4.5: Named RA Queries

We provide named RA versions for the expressions given in Example 4.3. The expression

$$\pi_{\{\text{pid}\}}\big(\sigma_{\text{prname} \neq \text{prname2}}\big((\text{Person} \bowtie \text{Profession}) \bowtie \rho_{\text{prname} \to \text{prname2}}(\text{Profession})\big)\big)$$

returns the IDs of persons with at least two professions. The expression

[2] We can assume that $A \lessdot (R, B)$ for all $A, B \in \mathsf{Att}$ and $R \in \mathsf{Rel}$, although this is inconsequential.

$$\pi_{\{\text{pid,pname}\}}(\text{Person}) - \pi_{\{\text{pid,pname}\}}(\text{Person} \bowtie \text{City})$$

returns the ID and name of persons whose city of birth does not appear in the database. Finally, the expression

$$\pi_{\{\text{pname}\}}(\sigma_{(\text{prname}\doteq\text{'author'})\vee(\text{prname}\doteq\text{'actor'})}(\text{Person} \bowtie \text{Profession}))$$

returns the names of persons that are authors or actors.

Expressiveness of Named and Unnamed RA

We often use named RA in examples since it is closer to how we think about real-life databases. On the other hand, many results are easier to state and prove in unnamed RA. This comes at no cost since, as we discuss below, every named RA query can be expressed in unnamed RA, and vice versa.

Let f be the function that converts a database D from the named to the unnamed perspective, as presented in Chapter 2. Recall that this converts each tuple $t = (A_1 \colon a_1, \ldots, A_k \colon a_k)$ in $D(R)$, for a relation name R of sort $\{A_1, \ldots, A_k\}$ (with $(R, A_1) \lessdot \cdots \lessdot (R, A_k)$), into a tuple $t' = (a_1, \ldots, a_k)$ in $f(D)(R)$. Let q_{n} be a named RA query, q_{u} an unnamed RA query, and $\mathbf{S}$ a named database schema. We say that q_{n} *is equivalent to* q_{u} *under* $\boldsymbol{S}$ if, for every database D of $\mathbf{S}$, we have that $q_{\mathsf{n}}(D) = q_{\mathsf{u}}(f(D))$.

Note that two queries can be equivalent under one schema but not equivalent under another one. This is unavoidable since the order inside an unnamed tuple depends on the names of the attributes (the order is defined by $\lessdot$). For instance, if R is a binary relation name, then $\pi_{(1)}(R)$ is equivalent to $\pi_{\{B\}}(R)$ over $\mathbf{S}_1 = \{R(B, C)\}$ but not over $\mathbf{S}_2 = \{R(A, B)\}$.

The following theorem establishes that each named RA query can be translated into an equivalent unnamed RA query. We leave the statement of the reverse direction and its proof as an exercise.

Theorem 4.6

Consider a named database schema $\mathbf{S}$, and a named RA query q_{n}. There exists an unnamed RA query q_{u} that is equivalent to q_{n} under $\mathbf{S}$.

Proof. We prove this by induction on the structure of q_{n}. Assume that q_{n} has sort $\{A_1, \ldots, A_k\}$ with $A_1 \lessdot \cdots \lessdot A_k$, where $\lessdot$ is the ordering we used in the definition of named RA queries. We proceed to explain how to obtain an unnamed RA query q_{u} that is equivalent to q_{n}, which means that the i-th attribute in the output of q_{u} corresponds to the A_i-attribute in the output of q_{n}. In the remainder of the proof, whenever we write a set of attributes as a set $\{A_1, \ldots, A_k\}$, we assume that $A_1 \lessdot \cdots \lessdot A_k$. The base cases are:

- If $q_{\mathsf{n}} = R$, for a relation name $R \in \mathbf{S}$ of sort $\{A_1, \ldots, A_k\}$, then $q_{\mathsf{u}} = R$.
- If $q_{\mathsf{n}} = \{(A : a)\}$, then $q_{\mathsf{u}} = \{a\}$.

For the inductive step, assume that q'_{n} and q''_{n} are named RA expressions of sort $U' = \{A'_1, \ldots, A'_k\}$ and $U'' = \{A''_1, \ldots, A''_\ell\}$, respectively, and assume that they are equivalent to the unnamed RA expressions q'_{u} and q''_{u}, respectively.

- Let $q_{\mathsf{n}} = \sigma_\theta(q'_{\mathsf{n}})$. Then $q_{\mathsf{u}} = \sigma_{\theta'}(q'_{\mathsf{u}})$, where θ' is the condition that is obtained from θ by replacing each occurrence of attribute A'_i with i, for every $i \in [k]$. For example, if θ is the condition $(A'_1 \doteq A'_3) \wedge (A'_2 \not\doteq b)$, then θ' is the condition $(1 \doteq 3) \wedge (2 \not\doteq b)$.
- Let $q_{\mathsf{n}} = \pi_\alpha(q'_{\mathsf{n}})$ and $\alpha \subseteq U'$. Then $q_{\mathsf{u}} = \pi_{\alpha'}(q'_{\mathsf{u}})$, where α' is the list of all $i \in [k]$ with $A'_i \in \alpha$.
- Let $q_{\mathsf{n}} = (q'_{\mathsf{n}} \bowtie q''_{\mathsf{n}})$. Then $q_{\mathsf{u}} = \pi_\alpha\,(q'_{\mathsf{u}} \bowtie_\theta q''_{\mathsf{u}})$, where θ is the conjunction of all conditions $i = j$ such that $A'_i = A''_j$, for $i \in [k]$ and $j \in [\ell]$. To define α, let $\{A_1, \ldots, A_m\} = U' \cup U''$ and let $g : [m] \to [k + \ell]$ be such that

$$g(i) = \begin{cases} j & \text{if } A_i = A'_j, \\ \\ k + j & \text{if } A_i = A''_j \text{ and } A''_j \in U'' - U' \ . \end{cases}$$

  We now define $\alpha = (g(1), \ldots, g(m))$. Therefore, θ allows us to mimic the natural join on q'_{n} and q''_{n}, while π_α is used for getting rid of redundant attributes and putting the attributes in an ordering that conforms to $\lessdot$.
- Let $q_{\mathsf{n}} = \rho_{A \to B}(q'_{\mathsf{n}})$, where $A = A'_i$ for some $i \in [k]$. Let $j = |\{i \mid A'_i \lessdot B\}|$. Then $q_{\mathsf{u}} = \pi_\alpha(q'_{\mathsf{u}})$, where α is obtained from $(1, \ldots, k)$ by deleting i and reinserting it right after j if $j > 0$, and at the beginning of the list if $j = 0$.
- Finally, if $q_{\mathsf{n}} = q'_{\mathsf{n}} \cup q''_{\mathsf{n}}$, then $q_{\mathsf{u}} = q'_{\mathsf{u}} \cup q''_{\mathsf{u}}$, where q'_{u} and q''_{u} are the unnamed RA expressions that are obtained by the induction hypothesis for q'_{n} and q''_{n}, respectively. The case when $q_{\mathsf{n}} = q'_{\mathsf{n}} - q''_{\mathsf{n}}$ is analogous. □

5

Relational Algebra and SQL

In this chapter, we shed light on the relationship between relational algebra and SQL, the dominant query language in the relational database world. It is a complex language (the full descriptions takes many hundreds of pages), and thus here we focus our attention on its core fragment.

A Core of SQL

We assume that the reader by virtue of being interested in the principles of databases has some basic familiarity with relational databases and thus, by necessity, with SQL. For now, we concentrate on the part of the language that corresponds to relational algebra. Its expressions are basic queries of the form

```
SELECT [DISTINCT] <list of attributes>
FROM <list of relations>
WHERE <condition>
```

and we can form more complex queries by using expressions

$$Q_1 \texttt{ UNION } Q_2 \quad \text{and} \quad Q_1 \texttt{ EXCEPT } Q_2 \ .$$

If the queries Q_1 and Q_2 return tables over the same set of attributes, these correspond to union and difference.

The *list of relations* provides relation names used in the query, and also their *aliases*; we either put a name `R` in the list, or `R AS R1`, in which case `R1` is used as a new name for `R`. This could be used to shorten the name, e.g.,

```
RelationWithAVeryLongName AS ShortName
```

or to use the same relation more than once, in which case different aliases are needed. We shall do both in the examples very soon.

The *list of attributes* containts attributes of relation names mentioned in `FROM` or constants. For example, if we had `R AS R1` in `FROM` and `R` has an attribute `A`, we can have a reference to `R1.A` in that list. The list of attributes specifies the attributes that will compose the output of the query. For a constant, one needs to provide the name of attribute: for example, `5 AS B` will output the constant 5 as value of attribute B.

The keyword `DISTINCT` is to instruct the query to perform duplicate elimination. In general, SQL tables and query results are allowed to contain duplicates. For example, in a database containing two facts, $R(a, b)$ and $R(a, c)$, projecting on the first column would result in *two* copies of a. We shall discuss duplicates in Chapter 46. In this Chapter, we will always assume that SQL queries only return sets, and omit `DISTINCT` from queries used in examples.

As *conditions* in this basic fragment we shall consider:

- equalities between attributes, e.g., `R.A = S.B`,
- equalities between attributes and constants, e.g., `Person.name = 'John'`,
- complex conditions built from these basic ones by using `AND`, `OR`, and `NOT`.

Example 5.1: SQL Queries

Consider the FO query $\varphi_1(x)$, where φ_1 is the FO formula (3.1). This can be written as the SQL query

```
SELECT P.pid
FROM Person AS P, Profession AS Pr1, Profession AS Pr2
WHERE P.pid = Pr1.pid
  AND P.pid = Pr1.pid
  AND NOT (Pr1.prname = Pr2.prname)
```

The formula φ_1 mentions the relation name `Person` once, and the relation name `Profession` twice, and so does the above SQL query in the `FROM` clause (assigning different names to different occurrences, to avoid ambiguity). The first two conditions in the `WHERE` clause capture the use of the same variable x in three atomic subformulae of φ_1, whereas the last condition corresponds to the subformula $\neg(u_1 = u_2)$.

Consider now the query $\varphi_2(x, y)$, where φ_2 is the FO formula (3.2), which asks for IDs and names of people whose cities of birth were not recorded in the `City` relation. This can be expressed as the SQL query:

```
SELECT Person.pid, Person.pname
FROM Person
  EXCEPT
SELECT Person.pid, Person.pname
FROM Person, City
```

```
WHERE Person.cid = City.cid
```

The first subquery asks for all people, the second subquery for those that have a city of birth recorded, and `EXCEPT` is their difference. This query returns people as pairs, consisting of their ID and their name.

Relational Algebra to Core SQL

We now show that (named) relational algebra queries can always be written as Core SQL queries. Let e be a named RA expression. We inductively translate e into an equivalent SQL query Q_e as follows.

Base Expressions. If $e = R$, and R has attributes $A_1, \ldots, A_n$, then Q_e is

```
SELECT A1, ..., An
FROM R
```

In fact, SQL has a shorthand `*` for listing all attributes of a relation name, and the above query can be written as `SELECT * FROM R`.

If $e = \{(A : a)\}$, then Q_e is simply

```
SELECT a AS A
```

Selection and Projection. Assume that e is translated into

```
SELECT A1, ..., An
FROM R1, ..., Rm
WHERE condition
```

- Then, $\sigma_\theta(e)$ is translated into

  ```
  SELECT A1, ..., An
  FROM R1, ..., Rm
  WHERE condition AND Cθ
  ```

  where C_θ expresses the condition θ in SQL syntax. For instance, if θ is $(A \doteq B) \wedge \neg(C \doteq 1)$ then C_θ is `(A = B) AND NOT (C = 1)`.
- Furthermore, $\pi_\alpha(e)$ is translated into

  ```
  SELECT Ai1, ..., Aik
  FROM R1, ..., Rm
  WHERE condition
  ```

  where $A_{i_1}, \ldots, A_{i_k}$ are the elements from the set α.

Rename. Assume now that e is translated into

```
SELECT ..., R_i.A_j AS A, ...
FROM R_1, ..., R_m
WHERE condition
```

Then, $\rho_{A\to B}(e)$ is translated into

```
SELECT ..., R_i.A_j AS B, ...
FROM R_1, ..., R_m
WHERE condition
```

Join, Union, and Difference. Assume now that e_1 is translated into

```
SELECT A_1, ..., A_k, B_1, ..., B_p,
FROM R_1, ..., R_m
WHERE condition
```

and that e_2 is translated into

```
SELECT A_1, ..., A_k, C_1, ..., C_s,
FROM S_1, ..., S_ℓ
WHERE condition'
```

where all the aliases $\mathtt{R}_1, \ldots, \mathtt{R}_m, \mathtt{S}_1, \ldots, \mathtt{S}_\ell$ are (renamed to be) distinct.

- The expression $e_1 \bowtie e_2$ is translated into

```
SELECT n_{e1}(A_1) AS A_1, ..., n_{e1}(A_k) AS A_k,
       n_{e1}(B_1) AS B_1, ..., n_{e1}(B_p) AS B_p,
       n_{e2}(C_1) AS C_1, ..., n_{e2}(C_s) AS C_s,
FROM   R_1, ..., R_m, S_1, ..., S_ℓ
WHERE  condition AND condition'
       AND n_{e1}(A_1) = n_{e2}(A_1) AND ··· AND n_{e1}(A_k) = n_{e2}(A_k)
```

  where $n_{e_1}(A_i)$ is the name of the attribute that was renamed as A_i in the translation of e_1. In other words, if we had `R.A AS` $\mathtt{A}_i$ in that query, then $n_{e_1}(\mathtt{A}_i) = \mathtt{R.A}$, and the definition is similar for e_2.

  This can be easily illustrated via an example. Consider the relation names $R[A, B, D]$, $S[B, C]$, $T[A, C, D]$, and the two queries

```
SELECT A, C, D                SELECT A, B, D
FROM R, S AS S1      and      FROM S AS S2, T
WHERE R.B = S1.B              WHERE S2.C = T.C
```

  Then, their join, having attributes A, B, C, D, is given by

```
SELECT R.A AS A, S2.B AS B, S1.C AS C, R.D AS D
FROM R, S AS S1, S AS S2, T
WHERE R.B = S1.B AND S2.C = T.C AND R.A = T.A AND R.D = T.D
```

- If $e = e_1 - e_2$, then Q_e is

$$(Q_{e_1})\ \texttt{EXCEPT}\ (Q_{e_2})$$

- Finally, if $e = e_1 \cup e_2$, then Q_e is

$$(Q_{e_1})\ \texttt{UNION}\ (Q_{e_2})$$

This completes the translation from (named) RA to Core SQL.

Core SQL to Relational Algebra

While the previous section explained how to write RA queries in SQL, this section gives an intuition as to what happens when an SQL query is executed on a DBMS. A declarative query is translated into a procedural query to be executed. The real translation of SQL into RA is significantly more complex and, of course, captures many more features of SQL (and thus, the algebra implemented in DBMSs goes beyond the algebra we consider here). Nonetheless, the translation we outline presents the key ideas of the real-life translation.

Assume that we start with the query

```
SELECT α1 AS B1, ..., αn AS Bn
FROM R1 AS S1, ..., Rm AS Sm
WHERE condition
```

where all relation names in `FROM` have been renamed so they are different, and each α_i is of the form $\texttt{S}_j.\texttt{A}_p$, that is, one of the attributes of the relation names in the `FROM` clause. Let $\boldsymbol{\rho}_i$ be the sequence of renaming operators that rename each attribute `A` of $\texttt{R}_i$ to $\texttt{S}_i.\texttt{A}$. Let $\boldsymbol{\rho}_{\text{out}}$ be the sequence of renaming operators that forms the output, i.e., it renames each α_i as $\texttt{B}_i$. Then, the translated query in relational algebra follows:

$$\boldsymbol{\rho}_{\text{out}}\left(\pi_{\{\alpha_1,\ldots,\alpha_n\}}\left(\sigma_{\text{condition}}\left(\boldsymbol{\rho}_1(R_1) \bowtie \cdots \bowtie \boldsymbol{\rho}_m(R_m)\right)\right)\right) .$$

Essentially the `FROM` defines the join, `WHERE` provides the condition for selection, and `SELECT` is the final projection (hence, some clash of the naming conventions in SQL and RA).

The translation is then supplemented by translating `UNION` to RA's union $\cup$ and `EXCEPT` to RA's difference $-$.

Other SQL Features Captured by RA

A very important feature of SQL is using *subqueries*. In the fragment we are considering, they are very convenient for a declarative presentation of queries (although from the point of view of expressiveness of the language, they can be omitted). Consider, for example, the query that computes the difference of two relations R and S with one attribute A. We could use `EXCEPT`, but using subqueries we can also write

```
SELECT R.A
FROM R
WHERE R.A NOT IN (SELECT S.A FROM S)
```

saying that we need to return elements of R that are not present in S, or

```
SELECT R.A
FROM R
WHERE NOT EXISTS (SELECT S.A FROM S WHERE S.A = R.A)
```

which asks for elements a of R such that there is no b in S satisfying $a = b$. Both queries express the difference.

Example 5.2: Subqueries in SQL

Consider the query $\varphi(x, y)$, where φ is the FO formula (3.2), which asks for IDs and names of people whose cities of birth were not recorded in the `City` relation. This can also be written as the SQL query:

```
SELECT P.pid, P.pname
FROM Person AS P
WHERE P.cid NOT IN (SELECT City.cid FROM City)
```

The above two forms of subqueries, using `NOT IN` and `NOT EXISTS`, correspond to adding the following two types of selection conditions to RA, which, nevertheless, do not increase the expressiveness of RA; see Exercise 1.5:

- $\bar{a} \in e$, where $\bar{a}$ is a tuple of terms and e is an expression, checking whether $\bar{a}$ belongs to the result of the evaluation of e, and
- $\text{empty}(e)$, checking if the result of the evaluation of e is empty.

In general, subqueries can be used in other clauses, and in fact they are very commonly used in `FROM`. A simple example follows.

Example 5.3: Subqueries in FROM

Consider the query $\varphi(x)$, where φ is the FO formula (3.1), which asks for people who have two different professions. This can be written as

```
SELECT PProfs.id
FROM
  (SELECT P.pid AS id, Pr1.prname as pf1, Pr2.prname as pf2
   FROM Person AS P, Profession AS Pr1, Profession AS Pr2
   WHERE P.pid = Pr1.pid AND P.pid = Pr1.pid)
  AS PProfs
WHERE NOT (PProfs.pf1 = PProfs.pf2)
```

In Example 5.3, the join of `Person` and `Profession` occurs in the subquery in `FROM`, and the condition that two professions are different is applied to the result of the join, which is given the name of `PProfs`. Again such addition does not increase expressiveness (Exercise 1.6) but makes writing queries easier.

Other SQL Features Not Captured by RA

Bag Semantics. As mentioned already, SQL's data model is based on bags, i.e., the same tuple may occur multiple times in a database or output of a query. Here we tacitly assumed that all relations are sets and each `SELECT` is followed by `DISTINCT` to ensure that duplicates are eliminated. To see how RA operations change in the presence of duplicates, see Chapter 46.

Grouping and Aggregation. An extremely common feature of SQL queries is the use of aggregation and grouping. Aggregation allows numerical functions to be applied to entire columns, for example, to find the total salary of all employees in a company. Grouping allows such columns to be split according to a value of some attribute; an example of this is a query that returns the total salary of each department in a company. These features will be discussed in more detail in Chapter 34.

Nulls. SQL databases permit missing values in tuples. To handle this, they allow a special element `null` to be placed as a value. The handling of nulls is very different though from the handling of values from Const, and even the notion of query output changes in this case. These issues are discussed in detail in Chapters 41 and 42.

Types. In SQL databases, attributes must be typed, i.e., all values in a column must have the same type. There are standard types such as numbers (integers, floats), strings of various length, fixed or varying, date, time, and many others. With the exception of the consideration of arithmetic operations (Chapter 34), this is a subject that we do study in this book.

6

Equivalence of Logic and Algebra

In this chapter, we prove that the declarative query language based on FO, and the procedural query language RA have the same expressive power, which is a fundamental result of relational database theory. Recall that we focus on the unnamed version of RA for reasons that we explained earlier.

Theorem 6.1

The languages of RA queries and of FO queries are equally expressive.

The proof of Theorem 6.1 boils down to showing that, for a schema $\mathbf{S}$, the following statements hold:

(a) For every RA expression e over $\mathbf{S}$, there exists an FO query q_e such that $q_e(D) = e(D)$, for every database D of $\mathbf{S}$.

(b) For every FO query q over $\mathbf{S}$, there exists an RA expression e_q such that $e_q(D) = q(D)$, for every database D of $\mathbf{S}$.

In the proof of the above, we need a mechanism that allows us to substitute variables in formulae. For an FO formula φ and variables $\{x_1, \ldots, x_n\}$, we denote by $\varphi[x_1/y_1, \ldots, x_n/y_n]$ the formula obtained from φ by simultaneously replacing each x_i with y_i. We also use the notation $\exists\{x_1, \ldots, x_n\}\varphi$ for a set of variables $\{x_1, \ldots, x_n\}$ as an abbreviation for $\exists x_1 \cdots \exists x_n \varphi$. Notice that the ordering of quantification is irrelevant for the semantics of this formula.

From RA to FO

We first show (a) by induction on the structure of e. The base cases are:

- If $e = R$ for $R \in \mathrm{Dom}(\mathbf{S})$, then the FO query is $\varphi_e(x_1, \ldots, x_{\mathrm{ar}(R)})$, where

$$\varphi_e \;=\; R(x_1, \ldots, x_{\mathrm{ar}(R)})$$

with all the variables $x_1, \ldots, x_{\mathrm{ar}(R)}$ being different.

- If e is $\{a\}$ with $a \in \mathsf{Const}$, then the FO query is $\varphi_e(x)$, where

$$\varphi_e \;=\; (x = a).$$

We now proceed with the induction step. Assume that e and e' are RA expressions over $\mathbf{S}$ for which we have equivalent FO queries $\varphi_e(x_1, \ldots, x_k)$ and $\varphi_{e'}(y_1, \ldots, y_\ell)$, respectively. By renaming variables, we can assume, without loss of generality, that $\{x_1, \ldots, x_k\}$ and $\{y_1, \ldots, y_\ell\}$ are disjoint.

- Let θ be a condition over $\{1, \ldots, k\}$. Taking $\bar{x} = (x_1, \ldots, x_k)$, we inductively define the formula $\theta[\bar{x}]$ as follows:
  - if θ is $i \doteq j$, $i \doteq a$, $i \neq j$, or $i \neq a$, then $\theta[\bar{x}]$ is $x_i = x_j$, $x_i = a$, $x_i \neq x_j$, or $x_i \neq a$, respectively,
  - if $\theta = \theta_1 \wedge \theta_2$, then $\theta[\bar{x}] = \theta_1[\bar{x}] \wedge \theta_2[\bar{x}]$,
  - if $\theta = \theta_1 \vee \theta_2$, then $\theta[\bar{x}] = \theta_1[\bar{x}] \vee \theta_2[\bar{x}]$, and
  - if $\theta = \neg\theta_1$, then $\theta[\bar{x}] = \neg\theta_1[\bar{x}]$.

  Then, the FO query equivalent to $\sigma_\theta(e)$ is $\varphi_{\sigma_\theta(e)}(\bar{x}) = \varphi_e(\bar{x}) \wedge \theta[\bar{x}]$.
- Let $\alpha = (i_1, \ldots, i_p)$ be a list of numbers from $\{1, \ldots, k\}$. The FO query equivalent to $\pi_\alpha(e)$ is $\varphi_{\pi_\alpha(e)}(x_{i_1}, \ldots, x_{i_p})$, where $\varphi_{\pi_\alpha(e)}$ is the formula

$$\exists(\{x_1, \ldots, x_n\} - \{x_{i_1}, \ldots, x_{i_p}\})\ \varphi_e.$$

  Notice that, if α has repetitions, then $(x_{i_1}, \ldots, x_{i_p})$ has repeated variables. For example, if $e = R$, where R is binary, and $\alpha = (1, 1)$, then the FO query is $\varphi_e(x_1, x_1)$ with $\varphi_e = \exists x_2\, R(x_1, x_2)$.
- The FO query equivalent to $e \times e'$ is $\varphi_{e \times e'}(x_1, \ldots, x_k, y_1, \ldots, y_\ell)$, where $\varphi_{e \times e'}$ is the formula

$$\varphi_e \wedge \varphi_{e'}.$$

- Let $e \cup e'$ be an RA expression, which is only well-defined if $k = \ell$. The equivalent FO query is $\varphi_{e \cup e'}(x_1, \ldots, x_k)$, where $\varphi_{e \cup e'}$ is

$$\varphi_e \vee (\varphi_{e'}[y_1/x_1, \ldots, y_k/x_k]).$$

- Let $e - e'$ be an RA expression, which is only well-defined if $k = \ell$. The equivalent FO query is $\varphi_{e-e'}(x_1, \ldots, x_k)$, where $\varphi_{e-e'}$ is

$$\varphi_e \wedge \neg(\varphi_{e'}[y_1/x_1, \ldots, y_k/x_k]).$$

We leave the verification of the construction, that is, the inductive proof of the equivalence of e and $\varphi_e(\bar{x})$, to the reader. This concludes part (a).

From FO to RA

For proving (b), we assume that relational atoms do not mention constants, which we observed in Chapter 3 is always possible. We also consider a slight generalization of FO queries that will simplify the induction: $\varphi(x_1, \ldots, x_n)$ is an FO query even if the free variables of φ are a subset of $\{x_1, \ldots, x_n\}$. The semantics of such a query $\varphi(x_1, \ldots, x_n)$ is the usual semantics of the FO query $\varphi'(x_1, \ldots, x_n)$, where φ' is the formula $\varphi \wedge (x_1 = x_1) \wedge \cdots \wedge (x_n = x_n)$.

Let q be an FO query of the form $\varphi(x_1, \ldots, x_n)$. We can assume, without loss of generality, that φ is in *prenex normal form*, that is, of the form

$$Q_k \cdots Q_1 \, \varphi_{\mathsf{qf}} \ ,$$

where

- each Q_j is of the form $\exists y_j$ or $\neg\exists y_j$,
- φ_{qf} is quantifier-free and has (free) variables $y_1, \ldots, y_m$,
- $\{x_1, \ldots, x_n\} = \{y_{k+1}, \ldots, y_m\}$, and
- φ_{qf} only uses the Boolean operators $\vee$ and $\neg$.

Let $\mathrm{Dom}(\varphi) = \{a_1, \ldots, a_\ell\}$. First, we build an RA expression Adom for the active domain, that is,

$$\mathrm{Adom} \ = \ \bigcup_{i=1}^{\ell} \{a_i\} \ \cup \ \bigcup_{R[n] \in \mathbf{S}} \big(\pi_1(R) \cup \cdots \cup \pi_n(R)\big) \ .$$

In the following, we denote by Adom^i, for $i \in \mathbb{N}$, the i-fold Cartesian product

$$\underbrace{\mathrm{Adom} \times \cdots \times \mathrm{Adom}}_{i} .$$

We construct for each subformula ψ of φ an RA query e_ψ. The induction hypothesis consists of two parts.

(1) For each subformula ψ of φ_{qf}, the expression e_ψ has arity m and is equivalent to the FO query $\psi(y_1, \ldots, y_m)$.

(2) For all the other subformulae ψ of φ, it holds that $\psi = Q_j \cdots Q_1 \, \varphi_{\mathsf{qf}}$, for $j \in [k]$, $\mathrm{FV}(\psi) = \{y_{j+1}, \ldots, y_m\}$, and the expression e_ψ, which has arity $m - j$, is equivalent to the FO query $Q_j \cdots Q_1 \, \varphi_{\mathsf{qf}}(y_{j+1}, \ldots, y_m)$.

The inductive construction defines the expression

$$e_\psi = \begin{cases} \pi_{1,\ldots,m}(\sigma_{i_1=m+1,\ldots,i_j=m+j}(\mathrm{Adom}^m \times R)) & \text{if } \psi \text{ is } R(y_{i_1},\ldots,y_{i_j}) \\ \sigma_{i \doteq j}(\mathrm{Adom}^m) & \text{if } \psi \text{ is } y_i = y_j \\ \sigma_{i \doteq a}(\mathrm{Adom}^m) & \text{if } \psi \text{ is } y_i = a \\ e_{\psi_1} \cup e_{\psi_2} & \text{if } \psi \text{ is } (\psi_1 \vee \psi_2) \\ \mathrm{Adom}^m - e_{\psi'} & \text{if } \psi \text{ is } \neg\psi', \text{ and} \\ & \psi \text{ is a subformula of } \varphi_{\mathsf{qf}} \\ \pi_{2,\ldots,m-j+1}(e_{\psi'}) & \text{if } \psi \text{ is } \exists y_j\ \psi' \text{ and } Q_j = \exists y_j \\ \mathrm{Adom}^{m-j} - \pi_{2,\ldots,m-j+1}(e_{\psi'}) & \text{if } \psi \text{ is } \neg\exists y_j\ \psi' \end{cases}$$

We leave the proof that the inductive construction gives an expression that is equivalent to $\varphi(y_{k+1},\ldots,y_m)$ to the reader. To obtain an expression equivalent to $\varphi(x_1,\ldots,x_n)$, observe that $x_i \in \{y_{k+1},\ldots,y_m\}$ for every $i \in [n]$. Therefore, there exists a function $f : [n] \to [m-k]$ such that $x_i = y_{f(i)}$ for every $i \in [n]$. This means that the expression $\pi_{(f(1),\ldots,f(n))}e_\varphi$ is equivalent to $\varphi(x_1,\ldots,x_n)$.

7

First-Order Query Evaluation

In this chapter, we study the complexity of evaluating first-order queries, that is, FO-Evaluation. Recall that this is the problem of checking whether $\bar{a} \in q(D)$ for an FO query q, a database D, and a tuple $\bar{a}$ over Const.

Combined Complexity

We first concentrate on the combined complexity of the problem, that is, when the input consists of the query q, the database D, and the tuple $\bar{a}$.

Theorem 7.1

FO-Evaluation is PSPACE-complete.

Proof. We start with the upper bound. We prove the result for the case where $\bar{a}$ is over $\mathrm{Dom}(D)$, and leave the extension to arbitrary tuples over Const as an exercise. Consider an FO query $q = \varphi(\bar{x})$, a database D, and a tuple $\bar{a}$ over $\mathrm{Dom}(D)$. We can assume, as discussed in Chapter 3, that the relational atoms in φ do not contain constant values. We can also assume that the tuples $\bar{x} = (x_1, \ldots, x_n)$ and $\bar{a} = (a_1, \ldots, a_m)$ are *compatible*, that is, they have the same length (i.e., $n = m$), and $x_i = x_j$ implies $a_i = a_j$ for every $i, j \in [n]$. Indeed, if $\bar{x}$ and $\bar{a}$ are not compatible, which can be easily checked using logarithmic space, then $\bar{a} \notin q(D)$ holds trivially. We can also assume that φ uses only $\neg$, $\vee$, and $\exists$ (see Exercise 1.1).

By Definition 3.6, $\bar{a} \in q(D)$ if and only if $(D, \eta) \models \varphi$ with η being the assignment for φ over D such that $\eta(\bar{x}) = \bar{a}$. Therefore, to establish Theorem 7.1, it suffices to show that the problem of checking whether $(D, \eta) \models \varphi$ is in PSPACE. This is done by exploiting the recursive procedure EVALUATION, depicted in Algorithm 1. Notice that the algorithm performs simple Boolean tests for determining its output values, like testing if $R(\eta(\bar{x}))$ is an element of D in line 1 or whether $\eta(x_i) = \eta(x_j)$ in line 2. It is not difficult to verify that

$(D, \eta) \models \varphi$ if and only if $\textsc{Evaluation}(\varphi, D, \eta) = \texttt{true}$. It remains to argue that $\textsc{Evaluation}(\varphi, D, \eta)$ uses polynomial space.

Algorithm 1 $\textsc{Evaluation}(\varphi, D, \eta)$

Input: An FO formula φ, a database D, and an assignment η for φ over D.
Output: `true` if $(D, \eta) \models \varphi$, and `false` otherwise.

if φ is of the form $R(\bar{x})$ **then return** $R(\eta(\bar{x})) \in D$
else if φ is of the form $(x_i = x_j)$ **then return** $\eta(x_i) = \eta(x_j)$
else if φ is of the form $(x_i = a)$ **then return** $\eta(x_i) = a$
else if φ is of the form $\neg\varphi'$ **then return** $\neg\textsc{Evaluation}(\varphi', D, \eta)$
else if φ is of the form $\varphi' \vee \varphi''$ **then**
    return $\textsc{Evaluation}(\varphi', D, \eta) \vee \textsc{Evaluation}(\varphi'', D, \eta)$
else if φ is of the form $\exists x\, \varphi'$ **then**
    return $\bigvee_{a \in \mathrm{Dom}(D)} \textsc{Evaluation}(\varphi', D, \eta[x/a])$
▷ $\eta[x/a]$ extends η by setting $\eta(x) = a$.

Lemma 7.2. $\textsc{Evaluation}(\varphi, D, \eta)$ *runs in* $\textsc{Space}(O(\|\varphi\|^2 \cdot \log \|D\|))$.

Proof. Observe that the total space used by $\textsc{Evaluation}(\varphi, D, \eta)$ is its recursion depth times the space needed by each recursive call. It is clear that the recursion depth is $O(\|\varphi\|)$. We proceed to argue, by induction on the structure of φ, that each recursive call uses $O(\|\varphi\| \cdot \log \|D\|)$ space on a Turing Machine, which in turn implies that the total space used by $\textsc{Evaluation}(\varphi, D, \eta)$ is $O(\|\varphi\|^2 \cdot \log \|D\|)$.

- Assume first that $\varphi = R(\bar{x})$. In this case, the algorithm checks whether $R(\eta(\bar{x})) \in D$. The space needed to store $\eta(\bar{x})$ on the work tape (adopting the encoding discussed in Appendix C) is $O(\|\varphi\| \cdot \log \|D\|)$. Furthermore, as shown in Appendix C (see Lemma C.1), for a tuple $\bar{t}$ over $\mathrm{Dom}(D)$, we can check whether $R(\bar{t}) \in D$ using $O(\mathrm{ar}(R) \cdot \log \|D\|)$ space if $\mathrm{ar}(R) > 0$, and $O(\log \|D\|)$ space if $\mathrm{ar}(R) = 0$. Therefore, in the worst-case where $\mathrm{ar}(R) > 0$, we can check whether $R(\eta(\bar{x})) \in D$ using space

$$O(\|\varphi\| \cdot \log \|D\|) \;+\; O(\mathrm{ar}(R) \cdot \log \|D\|).$$

  Since $\mathrm{ar}(R) \leq \|\varphi\|$, the total space used is $O(\|\varphi\| \cdot \log \|D\|)$.
- When $\varphi = (x_i = x_j)$, the algorithm checks whether $\eta(x_i) = \eta(x_j)$, which can be done using $O(\|\varphi\| \cdot \log \|D\|)$ space by simply storing the tuples $\eta(x_i)$ and $\eta(x_j)$ (adopting the encoding from Appendix C) on the work tape, and then check that they are equal. The case $\varphi = (x_i = a)$ is analogous.
- When $\varphi = \neg\varphi'$, the algorithm computes the value $\neg\textsc{Evaluation}(\varphi', D, \eta)$, which, by induction hypothesis, can be done using $O(\|\varphi\| \cdot \log \|D\|)$ space.

- When $\varphi = \varphi' \vee \varphi''$, the algorithm computes $\textsc{Evaluation}(\varphi', D, \eta) \vee \textsc{Evaluation}(\varphi'', D, \eta)$, which, by induction hypothesis, can be done using $O(\|\varphi\| \cdot \log \|D\|)$ space.
- Finally, assume that $\varphi = \exists x\, \varphi'$. In this case, the algorithm computes $\bigvee_{a \in \mathrm{Dom}(D)} \textsc{Evaluation}(\varphi', D, \eta[x/a])$. This is done by iterating over the constants of $\mathrm{Dom}(D)$ in the order provided by the encoding of D (see Appendix C), and reusing the space used by the previous iteration. Thus, it suffices to argue that computing the value $\textsc{Evaluation}(\varphi', D, \eta[x/a])$, for some value $a \in \mathrm{Dom}(D)$, can be done using $O(\|\varphi\| \cdot \log \|D\|)$ space. The latter clearly holds by induction hypothesis, and the claim follows. □

For the lower bound, we provide a reduction from QSAT, which we know is PSPACE-complete (see Appendix B). Consider an input to QSAT given by

$$\psi \;=\; \exists \bar{x}_1 \forall \bar{x}_2 \exists \bar{x}_3 \ldots Q_n \bar{x}_n\, \psi'\langle \bar{x}_1, \ldots, \bar{x}_n \rangle,$$

where $Q_n = \forall$ if n is even, and $Q_n = \exists$ if n is odd. We assume that ψ' is in negation normal form, which means that negation is only applied to variables, since QSAT remains PSPACE-hard. We construct the database

$$D \;=\; \{\mathrm{Zero}(0), \mathrm{One}(1)\}$$

and the Boolean FO query

$$q_\psi \;=\; \exists \bar{x}_1 \forall \bar{x}_2 \exists \bar{x}_3 \ldots Q_n \bar{x}_n\, \psi'',$$

where ψ'' is obtained from ψ' by replacing each occurrence of the literal x by $\mathrm{One}(x)$, and each occurrence of the literal $\neg x$ by $\neg\mathrm{One}(x)$. For example, if $\psi'(x_1, x_2, x_3) = (x_1 \wedge x_2) \vee (\neg x_1 \wedge x_3)$, then $\psi'' = (\mathrm{One}(x_1) \wedge \mathrm{One}(x_2)) \vee (\neg\mathrm{One}(x_1) \wedge \mathrm{One}(x_3))$. It is not hard to verify that ψ is satisfiable if and only if $D \models q_\psi$ (we leave the proof as an exercise). □

Note that $q(D)$, for an FO query $q = \varphi(\bar{x})$ and a database D, can also be computed in polynomial space as follows: iterate over all tuples $\bar{a}$ over $\mathrm{Dom}(D)$ that are compatible with $\bar{x}$, and output $\bar{a}$ if and only if $\textsc{Evaluation}(\varphi, D, \eta)$ = `true` with η being the assignment for φ over D such that $\eta(\bar{x}) = \bar{a}$. It is easy to show that this procedure runs in polynomial space. This, of course, relies on the fact that the running space of a Turing Machine with output is defined without considering the output tape; see Appendix B for details.

Data Complexity

How can it be that databases are so successful in practice, even though Theorem 7.1 proves that the most essential database problem is PSPACE-complete, a complexity class that we consider to be intractable? If we take a closer look

at the lower bound proof of Theorem 7.1, we see that the entire difficulty of the problem is encoded in the query. In fact, the database $D = \{\text{Zero}(0), \text{One}(1)\}$ consists of only two atoms, whereas the query q can be arbitrarily large. This is in contrast to what we typically experience in practice, where databases are orders of magnitude larger than queries, which means that databases and queries contribute in different ways to the complexity of evaluation. This brings us to the data complexity of FO query evaluation.

As discussed in Chapter 2, when we study the data complexity of query evaluation, we essentially consider the query to be fixed, and only the database and the candidate output are considered as input. Formally, we are interested in the complexity of the problem q-Evaluation for an FO query q, which takes as input a database D and a tuple $\bar{a}$ over $\text{Dom}(D)$, and asks whether $\bar{a} \in q(D)$. Recall that, by convention, we say that FO-Evaluation is in a complexity class $\mathcal{C}$ in data complexity if q-Evaluation is in $\mathcal{C}$ for every FO query q.

Theorem 7.3

FO-Evaluation is in DLogSpace in data complexity.

Proof. Fix an FO query $q = \varphi(\bar{x})$. Our goal is to show that q-Evaluation is in DLogSpace. As for Theorem 7.1, we prove the result for the case where $\bar{a}$ is over $\text{Dom}(D)$, and leave the extension to tuples over Const as an exercise.

Consider a database D, and a tuple $\bar{a}$ over $\text{Dom}(D)$. Observe that the input word encoding D on a Turing Machine has length $O(\|D\| \log \|D\|)$. We therefore need to prove that FO-Evaluation can be solved in $\text{Space}(O(\log(\|D\| \log \|D\|))) = \text{Space}(O(\log \|D\|))$. As explained in the proof of Theorem 7.1, we can assume that the relational atoms in φ do not contain constants, the tuples $\bar{x} = (x_1, \ldots, x_n)$ and $\bar{a} = (a_1, \ldots, a_m)$ are compatible, and that φ uses only $\neg$, $\vee$, and $\exists$. To prove our claim it suffices to show that checking whether $(D, \eta) \models \varphi$ with η being the assignment for φ over D such that $\eta(\bar{x}) = \bar{a}$ is in DLogSpace. This is done by exploiting the procedure $\text{Evaluation}_\varphi$, which takes as input D and η, and is defined in exactly the same way as the procedure Evaluation given in Algorithm 1. It is straightforward to see that $(D, \eta) \models \varphi$ if and only if $\text{Evaluation}_\varphi(D, \eta) = \texttt{true}$. Moreover, from the complexity analysis of Evaluation performed in the proof of Theorem 7.1, and the fact that φ is fixed, we conclude that $\text{Evaluation}_\varphi(D, \eta)$ runs in space $O(\log \|D\|)$, and the claim follows. □

Theorem 7.3 essentially tells us that fixing the query indeed has a big impact to the complexity of evaluation, which goes from PSpace to DLogSpace. Actually, FO-Evaluation is in AC_0 in data complexity, a class that is properly contained in DLogSpace. The class AC_0 consists of those languages that are accepted by polynomial-size circuits of constant depth and unbounded fan-in (the number of inputs to their gates). This is the reason why FO-Evaluation is often regarded as an "embarrassingly parallel" task.

8

Static Analysis

We now study central static analysis tasks for FO queries. We focus on satisfiability, containment, and equivalence, which are key ingredients for query optimization. As we shall see, these problems are undecidable for FO queries. This in turn implies that, given an FO query, computing an optimal equivalent FO query is, in general, algorithmically impossible.

Satisfiability

A query q is *satisfiable* if there is a database D such that $q(D)$ is non-empty. It is clear that a query that is not satisfiable it is not a useful query since its output on a database is always empty. In relation to satisfiability, we consider the following problem, parameterized by a query language $\mathcal{L}$.

Problem: $\mathcal{L}$-Satisfiability

Input: A query q from $\mathcal{L}$

Output: `true` if there is a database D such that $q(D) \neq \emptyset$, and `false` otherwise

Notice that satisfiability is, in a sense, the most elementary static analysis question one can ask about a query: "does there exist a database *at all* for which the query returns an answer?" Indeed, if there does not, then optimizing the query is extremely simple: one can just always return the empty set of answers, independently of the input database.

We are asking the satisfiability question focussing on finite databases. In the case of possibly infinite databases, we know from a classical result in logic that goes back in the 1930s, known as Church's Theorem (sometimes called Church-Turing Theorem), that checking for satisfiability is undecidable. The problem remains undecidable even for finite databases, a result proved by

Trakhtenbrot in the 1950s, i.e., several years after Church's Theorem. In what follows we present Trakhtenbrot's Theorem.

Theorem 8.1: Trakhtenbrot's Theorem

FO-Satisfiability is undecidable.

Proof. The proof is by reduction from the halting problem for Turing Machines; details on Turing Machines can be found in Appendix B. It is well-known that the problem of deciding whether a (deterministic) Turing Machine $M = (Q, \Sigma, \delta, s)$ halts on the empty word is undecidable. Our goal is to construct a Boolean FO query q_M such that the following are equivalent:

1. M halts on the empty word.
2. There exists a database D such that $q_M(D) = \texttt{true}$.

The Boolean FO query q_M will be over the schema

$$\{\prec[2], \text{First}[1], \text{Succ}[2]\} \ \cup \ \{\text{Symbol}_a[2] \mid a \in \Sigma\} \ \cup \ \{\text{Head}[2], \text{State}[2]\}.$$

The intuitive meaning of the above relation names is the following:

- $\prec(\cdot, \cdot)$ encodes a strict linear order over the underlying domain, which will be used to simulate the time steps of the computation of M on the empty word, and the tape cells of M.
- $\text{First}(\cdot)$ contains the first element from the linear order $\prec$.
- $\text{Succ}(\cdot, \cdot)$ encodes the successor relation over the linear order $\prec$.
- $\text{Symbol}_a(t, c)$: at time instant t, the tape cell c contains the symbol a.
- $\text{Head}(t, c)$: at time instant t, the head points at cell c.
- $\text{State}(t, p)$: at time instant t, the machine M is in state p.

Having the above schema in place, we can now proceed with the definition of the Boolean FO query q_M, which is of the form

$$\varphi_{\prec} \ \wedge \ \varphi_{\text{first}} \ \wedge \ \varphi_{\text{succ}} \ \wedge \ \varphi_{\text{comp}},$$

where $\varphi_{\prec}$, φ_{first} and φ_{succ} are FO sentences that are responsible for defining the relations $\prec$, First and Succ, respectively, while φ_{comp} is an FO sentence responsible for mimicking the computation of M on the empty word. The definitions of the above FO sentences follow. For the sake of readability, we write $x \prec y$ instead of the formal $\prec(x, y)$.

The Sentence $\varphi_{\prec}$

This sentence simply expresses that the binary relation $\prec$ over the underlying domain is total, irreflexive, and transitive:

$$\forall x \forall y \left(\neg(x = y) \rightarrow (x \prec y \vee y \prec x)\right) \wedge \forall x \neg(x \prec x) \wedge \forall x \forall y \forall z \left((x \prec y \wedge y \prec z) \rightarrow x \prec z\right).$$

Note that irreflexivity and transitivity together imply that the relation $\prec$ is also asymmetric, i.e., $\forall x \forall y \neg(x \prec y \wedge y \prec x)$.

The Sentence φ_{first}

This sentence expresses that $\mathrm{First}(\cdot)$ contains the smallest element over $\prec$:

$$\forall x \forall y \left(\mathrm{First}(x) \;\leftrightarrow\; (x = y \vee x \prec y)\right)$$

The Sentence φ_{succ}

It simply defines the successor relation over $\prec$ as expected:

$$\forall x \forall y \left(\mathrm{Succ}(x, y) \;\leftrightarrow\; \left(x \prec y \;\wedge\; \neg \exists z \left(x \prec z \wedge z \prec y\right)\right)\right).$$

The Sentence φ_{comp}

Assume that the set of states of M is $Q = \{p_1, \ldots, p_k\}$, where $p_1 = s$ is the start state, $p_2 =$ "yes" is the accepting state, and $p_3 =$ "no" is the rejecting state. The key idea is to associate to each state of M a distinct element of the underlying domain, which in turn will allow us to refer to the states of M. Thus, φ_{comp} is defined as the following FO sentence; for a subformula ψ of φ_{comp}, we write $\psi\langle \bar{x} \rangle$ to indicate that $\mathrm{FV}(\psi)$ consists of the variables in $\bar{x}$:

$$\exists x_1 \cdots \exists x_k \Bigg(\bigwedge\nolimits_{i,j \in [k] \,:\, i<j} \neg(x_i = x_j) \;\wedge\; \varphi_{\mathrm{start}}\langle x_1 \rangle \;\wedge\; \varphi_{\mathrm{consistent}}\langle x_1, \ldots, x_k \rangle \wedge \varphi_{\delta}\langle x_1, \ldots, x_k \rangle \;\wedge\; \varphi_{\mathrm{halt}}\langle x_2, x_3 \rangle \Bigg),$$

where

- φ_{start} defines the start configuration $sc(\varepsilon)$,
- $\varphi_{\mathrm{consistent}}$ performs several consistency checks to ensure that the computation of M on the empty word is faithfully described,
- φ_{δ} encodes the transition function of M, and
- φ_{halt} checks whether M halts.

The definitions of the subformulae of φ_{comp} follow.

The Formula φ_{start}. It is defined as the conjunction of the following FO formulae, expressing that the first tape cell contains the left marker

$$\forall x \left(\text{First}(x) \;\rightarrow\; \text{Symbol}_{\triangleright}(x,x)\right),$$

the rest of tape cells contain the blank symbol

$$\forall x \forall y \left((\text{First}(x) \wedge \neg\text{First}(y)) \;\rightarrow\; \text{Symbol}_{\sqcup}(x,y)\right),$$

the head points to the first cell

$$\forall x \left(\text{First}(x) \;\rightarrow\; \text{Head}(x,x)\right),$$

and the machine M is in state s

$$\forall x \left(\text{First}(x) \;\rightarrow\; \text{State}(x,x_1)\right).$$

Note that we refer to the start state $s = p_1$ via the variable x_1.

The Formula $\varphi_{\text{consistent}}$. It is defined as the conjunction of the following FO formulae, expressing that, at any time instant x, M is in exactly one state

$$\forall x \left(\left(\bigvee_{i=1}^{k} \text{State}(x,x_i) \right) \;\wedge \bigwedge_{i,j \in [k] \,:\, i<j} \neg\big(\text{State}(x,x_i) \wedge \text{State}(x,x_j)\big) \right),$$

each tape cell y contains exactly one symbol

$$\forall x \forall y \left(\left(\bigvee_{a \in \Sigma} \text{Symbol}_a(x,y) \right) \;\wedge \bigwedge_{a,b \in \Sigma \,:\, a \neq b} \neg\big(\text{Symbol}_a(x,y) \wedge \text{Symbol}_b(x,y)\big) \right),$$

and the head points at exactly one cell

$$\forall x \left(\exists y \, \text{Head}(x,y) \;\wedge\; \forall y \forall z \left(\big(\text{Head}(x,y) \wedge \text{Head}(x,z)\big) \rightarrow y = z \right) \right).$$

The Formula φ_{δ}. It is defined as the conjunction of the following FO formulae: for each pair $(p_i, a) \in (Q - \{\text{“yes”}, \text{“no”}\}) \times \Sigma$ with $\delta(p_i, a) = (p_j, b, \text{dir})$,

$$\forall x \forall y \Bigg(\big(\text{State}(x,x_i) \wedge \text{Head}(x,y) \wedge \text{Symbol}_a(x,y) \wedge \exists t \, (x \prec t)\big) \rightarrow$$
$$\exists z \exists w \Bigg(\text{Succ}(x,z) \wedge \text{Move}(y,w) \wedge \text{Head}(z,w) \wedge \text{Symbol}_b(z,y) \wedge \text{State}(z,x_j) \wedge$$
$$\forall u \Big(\neg(y=u) \rightarrow \bigwedge_{c \in \Sigma} \big(\text{Symbol}_c(x,u) \rightarrow \text{Symbol}_c(z,u)\big) \Big) \Bigg) \Bigg),$$

where

$$\text{Move}(y, w) \;=\; \begin{cases} \text{Succ}(y, w) & \text{if dir} = \rightarrow, \\ \text{Succ}(w, y) & \text{if dir} = \leftarrow, \text{ and} \\ y = w & \text{if dir} = -. \end{cases}$$

The Formula $\varphi_{\mathbf{halt}}$. Finally, this formula checks whether M has reached an accepting or a rejecting configuration

$$\exists x \, (\text{State}(x, x_2) \;\vee\; \text{State}(x, x_3)).$$

Recall that, by assumption, $p_2 =$ "yes" and $p_3 =$ "no". Thus, the states "yes" and "no" can be accessed via the variables x_2 and x_3, respectively.

This completes the construction of the Boolean FO query q_{comp}, and thus of q_M. It is not hard to verify that M halts on the empty word if and only if there exists a database D such that $q(D) = \texttt{true}$, and the claim follows. □

The proof of Theorem 8.1 relies on the *finiteness* of databases; it does not work for possibly infinite databases. Assuming that the Turing Machine M does not halt on the empty word, we can construct an infinite database D such that $q_M(D) = \texttt{true}$ (we leave this as an exercise).[1] As mentioned earlier, Church's Theorem shows the undecidability of the satisfiability problem for FO queries over possibly infinite databases (see also Exercise 1.11).

We have seen in Chapter 6 that FO and RA have the same expressive power (Theorem 6.1). This fact and Theorem 8.1 immediately imply the following.

Corollary 8.2

RA-Satisfiability is undecidable.

Containment and Equivalence

We now focus on the problems of containment and equivalence for FO queries: given two FO queries q and q', is it the case that $q \subseteq q'$ and $q \equiv q'$, respectively. By exploiting Theorem 8.1, it is easy to show the following.

Theorem 8.3

FO-Containment and FO-Equivalence are undecidable.

[1] The output of an FO query on an infinite database D is defined in the same way as for databases (see Definition 3.6).

Proof. The proof is by an easy reduction from FO-Satisfiability. Consider an FO query q. From the proof of Theorem 8.1, we know that FO-Satisfiability is undecidable even for Boolean FO queries. Consider the Boolean FO query

$$q' \;=\; \exists x\,(R(x) \wedge \neg R(x)),$$

which is trivially unsatisfiable. It is easy to verify that q is unsatisfiable if and only if $q \equiv q'$ (or even $q \subseteq q'$), and the claim follows. □

The following is an easy consequence of the fact that FO and RA have the same expressive power, and Theorem 8.3.

Corollary 8.4

RA-Containment and RA-Equivalence are undecidable.

9

Homomorphisms

Homomorphisms are a fundamental tool that plays a very prominent role in various aspects of relational databases. We introduce them here, because we will use them in Chapter 10 to reason about functional dependencies. In this chapter, we define homomorphisms and provide some simple examples.

Definition of Homomorphism

Homomorphisms are structure-preserving functions between two objects of the same type. In our setting, the objects that we are interested in are (possibly infinite) databases and queries. To talk about them as one we define homomorphisms among (possibly infinite) sets of relational atoms. Recall that relational atoms are of the form $R(\bar{u})$, where $\bar{u}$ is a tuple that can mix variables and constants, e.g., $R(a, x, 2, b)$. Recall also that we write $\mathrm{Dom}(S)$ for the set of constants and variables occurring in a set of relational atoms S; for example, $\mathrm{Dom}(\{R(a, x, b), R(x, a, y)\}) = \{a, b, x, y\}$.

The way that the notion of homomorphism is defined between sets of atoms is slightly different from the standard notion of mathematical homomorphism, namely constant values of Const should be mapped to themselves. The reason for this is that, in general, a value $a \in \mathsf{Const}$ represents an object different from the one represented by $b \in \mathsf{Const}$ with $a \neq b$, and homomorphisms, as structure preserving functions, should preserve this information as well.

Definition 9.1: Homomorphism

Let S, S' be sets of relational atoms over the same schema. A *homomorphism from S to S'* is a function $h : \mathrm{Dom}(S) \to \mathrm{Dom}(S')$ such that:

1. $h(a) = a$ for every $a \in \mathrm{Dom}(S) \cap \mathsf{Const}$, and
2. if $R(\bar{u})$ is an atom in S, then $R(h(\bar{u}))$ is an atom in S'.

If $h(\bar{u}) = \bar{v}$, where $\bar{u}, \bar{v}$ are tuples of the same length over $\mathrm{Dom}(S)$ and $\mathrm{Dom}(S')$, respectively, then h is a *homomorphism from* $(S, \bar{u})$ *to* $(S', \bar{v})$. We write $S \to S'$ if there exists a homomorphism from S to S', and $(S, \bar{u}) \to (S', \bar{v})$ if there exists a homomorphism from $(S, \bar{u})$ to $(S', \bar{v})$.

Example 9.2: Homomorphism

Assume that S and S' are sets of relational atoms over a schema with a single binary relation name R. In this way, we can view both S and S' as a graph: the set of nodes is the set of constants and variables occurring in the relational atoms, and $R(u, v)$ means that there exists an edge from u to v. Unless stated otherwise, the elements in S and S' are variables.

A homomorphism always exists. Let $S' = \{R(z, z)\}$. The function $h : \mathrm{Dom}(S) \to \mathrm{Dom}(S')$ such that $h(x) = z$, for each $x \in \mathrm{Dom}(S)$, is a homomorphism from S to S' since $R(h(x), h(y)) = R(z, z)$ is an atom of S', for every $x, y \in \mathrm{Dom}(S)$.

A homomorphism does not exist. Let $S = \{R(a, x)\}$ and $S' = \{R(z, z)\}$, where $a \in \mathsf{Const}$. In contrast to the previous example, there is no homomorphism h from S to S' since, by definition, $h(a)$ must be equal to a, while $a \notin \mathrm{Dom}(S')$.

A homomorphism is easy to find. Let now $S' = \{R(x, y), R(y, x)\}$. Assume that a homomorphism h from S to S' exists. As usual, h^{-1} stands for the inverse, i.e., $h^{-1}(x) = \{z \in \mathrm{Dom}(S) \mid h(z) = x\}$, and likewise for $h^{-1}(y)$. The sets $h^{-1}(x)$ and $h^{-1}(y)$ are disjoint since $x \neq y$. If we have an edge (z, w) in S, we know that the variables z and w cannot belong to the same set $h^{-1}(x)$ or $h^{-1}(y)$; otherwise, either $R(x, x)$ or $R(y, y)$ would be an atom in S by the definition of the homomorphism. This means that S, viewed as a graph, is *bipartite*: its nodes are partitioned into two sets such that edges can only connect vertices in different sets. In other words, the nodes of the graph given by S can be colored with two colors x and y. Thus, in this case, checking for the existence of a homomorphism witnessing $S \to S'$ is the same as checking for the existence of a 2-coloring of S, which can be done in polynomial time (by using, for example, a coloring version of depth-first search).

A homomorphism is hard to find. We now add z to $\mathrm{Dom}(S')$, and let $S' = \{R(x, y), R(y, x), R(x, z), R(z, x), R(y, z), R(z, y)\}$. Then, as before, if $h : \mathrm{Dom}(S) \to \mathrm{Dom}(S')$ is a homomorphism from S to S', and $R(z, w)$ is an edge in S, then $h(z) \neq h(w)$. In other words, the nodes of the graph given by S can be colored with three colors x, y and z. Therefore, in this case, checking for the existence of a

homomorphism witnessing $S \to S'$ is the same as checking for the existence of a 3-coloring of S, which is an NP-complete problem.

Grounding Sets of Atoms

In several chapters, it will be convenient to have a mechanism viewing sets of atoms as databases. This is done by converting a set of atoms S into a possibly infinite database by replacing the variables occurring in S by new constants not already in S.[1] This process is called *grounding*, and can be easily defined via homomorphisms.

Definition 9.3: Grounding

Let S be a set of relational atoms over a schema $\mathbf{S}$. A possibly infinite database D of $\mathbf{S}$ is called a *grounding of* S if there exists a homomorphism from S to D that is a bijection.

Note that, in general, there is no unique grounding for a set of atoms. Consider, for example, the set of atoms

$$S \;=\; \{R(x, a, y), P(y, b, x, z)\},$$

where a, b are constants and x, y, z are variables. The databases

$$D_1 \;=\; \{R(c_1, a, d_1), R(d_1, b, c_1, e_1)\} \text{ and } D_2 \;=\; \{R(c_2, a, d_2), R(d_2, b, c_2, e_2)\}$$

with $c_1 \neq c_2$, $d_1 \neq d_2$, and $e_1 \neq e_2$, are both groundings of S. On the other hand, D_1 and D_2 are isomorphic databases, that is, they are the same up to renaming of constants. This simple observation can be generalized to any set of atoms. In particular, for a set of atoms S, it is straightforward to show that, for every two groundings D_1 and D_2 of S, there is a bijection $\rho : \mathsf{Const} \to \mathsf{Const}$ such that $\rho(D_1) = D_2$. Therefore, we can refer to:

- *the* grounding of S, denoted $S^{\downarrow}$, and
- *the* unique bijective homomorphism G_S from S to $S^{\downarrow}$.

We conclude the chapter with a note on the difference between $\mathrm{Dom}(S)$ and $\mathrm{Dom}(S^{\downarrow})$ to avoid confusion later in the book. If S is a set of atoms, then $\mathrm{Dom}(S) \subseteq \mathsf{Const} \cup \mathsf{Var}$, that is, it may contain *both* constants and variables. On the other hand, by definition, $\mathrm{Dom}(S^{\downarrow})$ contains only constants. Similarly, R^S is a set of tuples that may mention constants and variables, while $R^{S^{\downarrow}}$ is a set of tuples that mention only constants.

[1] Converting a database into a set of atoms by replacing constants with variables is needed less often; this is discussed in Chapter 14.

10

Functional Dependencies

In a relational databse system, it is possible to specify semantic properties that should be satisfied by all databases of a certain schema, such as *"every person should have at most one social security number"*. Such properties are crucial in the development of transparent and usable database schemas for complex applications, as well as for optimizing the evaluation of queries. However, the relational model as presented in Chapter 2 is not powerful enough to express such semantic properties. This can be achieved by incorporating *integrity constraints*, also known as *dependencies*.

One of the most important classes of dependencies supported by relational systems is the class of *functional dependencies*, which can express that the values of some attributes of a tuple uniquely (or functionally) determine the values of other attributes of that tuple.

Example 10.1: Functional Dependencies

Consider the (named) database schema

```
Person [ pid, name, cid ]
```

We can express that the id of a person uniquely determines that person via the functional dependency

$$\text{Person} : \{1\} \to \{1, 2, 3\},$$

which states that whenever two tuples of the relation Person agree on the first attribute, the id, they should also agree on all the other attributes.

Note that the form of dependency used in Example 10.1, where a set of attributes determines the *entire tuple*, is of particular interest and is called a *key dependency*. We may also say that the id attribute is a *key* of Person.

Syntax and Semantics

We start with the syntax of functional dependencies.

Definition 10.2: Syntax of Functional Dependencies

A *functional dependency* (FD) σ over a schema $\mathbf{S}$ is an expression

$$R : U \to V$$

where $R \in \mathbf{S}$ and $U, V \subseteq \{1, \ldots, \mathrm{ar}(R)\}$. If $V = \{1, \ldots, \mathrm{ar}(R)\}$, then σ is called a *key dependency*, and we simply write $\mathit{key}(R) = U$.

Intuitively, an FD $R : U \to V$ expresses that the values of the attributes U of R functionally determine the values of the attributes V of R, while a key dependency $\mathit{key}(R) = U$ states that the values of the attributes U of R functionally determine the values of *all* the attributes of R. We proceed to formally define the semantics of FDs. Note that in the following definition, by abuse of notation, we write U and V in the projection expressions $\pi_U(\cdot)$ and $\pi_V(\cdot)$ for the lists consisting of the elements of U and V in ascending order.

Definition 10.3: Semantics of FDs

A database D of a schema $\mathbf{S}$ *satisfies* an FD σ of the form $R : U \to V$ over $\mathbf{S}$, denoted $D \models \sigma$, if for each pair of tuples $\bar{a}, \bar{b} \in R^D$,

$$\pi_U(\bar{a}) = \pi_U(\bar{b}) \quad \text{implies} \quad \pi_V(\bar{a}) = \pi_V(\bar{b}).$$

D *satisfies* a set Σ of FDs, written $D \models \Sigma$, if $D \models \sigma$ for each $\sigma \in \Sigma$.

Note that the notion of satisfaction for FDs can be easily transferred to finite sets of atoms by exploiting the notion of grounding of sets of atoms. In particular, a finite set of atoms S satisfies an FD σ, denoted $S \models \sigma$, if $S^{\downarrow} \models \sigma$, while S satisfies a set Σ of FDs, written $S \models \Sigma$, if $S^{\downarrow} \models \Sigma$.

Satisfaction of Functional Dependencies

A central task is checking whether a database D satisfies a set Σ of FDs.

Problem: FD-Satisfaction

Input: A database D of a schema $\mathbf{S}$, and a set Σ of FDs over $\mathbf{S}$
Output: `true` if $D \models \Sigma$, and `false` otherwise

It is not difficult to show the following result:

Theorem 10.4

FD-Satisfaction is in PTIME.

Proof. Consider a database D of a schema $\mathbf{S}$, and a set Σ of FDs over $\mathbf{S}$. Let σ be an FD from Σ of the form $R : U \to V$. To check whether $D \models \sigma$ we need to check that, for every $\bar{a}, \bar{b} \in R^D$, $\pi_U(\bar{a}) = \pi_U(\bar{b})$ implies $\pi_V(\bar{a}) = \pi_V(\bar{b})$. It is easy to verify that this can be done in time $O(\|D\|^2)$. Therefore, we can check whether $D \models \Sigma$ in time $O(\|\Sigma\| \cdot \|D\|^2)$, and the claim follows. □

The Chase for Functional Dependencies

Another crucial task in connection with dependencies is that of (logical) implication, which allows us to discover new dependencies from existing ones. A natural problem that arises in this context is, given a set of dependencies Σ and a dependency σ, to determine whether Σ implies σ. This means checking if, for every database D such that $D \models \Sigma$, it holds that $D \models \sigma$. Before formalizing and studying this problem, we first introduce a fundamental algorithmic tool for reasoning about dependencies known as *the chase*. Actually, the chase should be understood as a family of algorithms since, depending on the class of dependencies in question, we may get a different variant. However, all the chase variants have the same objective, that is, given a finite set of relational atoms S, and a set Σ of dependencies, to transform S as dictated by Σ into a set of relational atoms that satisfies Σ.

Consider a finite set S of relational atoms over a schema $\mathbf{S}$, and an FD $\sigma = R : U \to V$ over $\mathbf{S}$. We say that σ is *applicable to S with* $(\bar{u}, \bar{v})$, where $\bar{u}, \bar{v} \in R^S$,[1] if $\pi_U(\bar{u}) = \pi_U(\bar{v})$ and $\pi_V(\bar{u}) \neq \pi_V(\bar{v})$. Let $\pi_V(\bar{u}) = (u_1, \ldots, u_k)$ and $\pi_V(\bar{v}) = (v_1, \ldots, v_k)$. For technical convenience, we assume that there is a strict total order $<$ on the elements of the set $\mathsf{Const} \cup \mathsf{Var}$ such that $a < x$, for each $a \in \mathsf{Const}$ and $x \in \mathsf{Var}$, i.e., constants are smaller than variables according to $<$. Let $h_{\bar{u},\bar{v}} : \mathrm{Dom}(S) \to \mathrm{Dom}(S)$ be a function such that

$$h_{\bar{u},\bar{v}}(w) = \begin{cases} u_i & \text{if } w = v_i \text{ and } u_i < v_i, \text{ for some } i \in [k], \\ v_i & \text{if } w = u_i \text{ and } v_i < u_i, \text{ for some } i \in [k], \\ w & \text{otherwise.} \end{cases}$$

The *result of applying σ to S with* $(\bar{u}, \bar{v})$ is defined as

$$S' = \begin{cases} \bot & \text{if there is an } i \in [k] \text{ with } u_i \neq v_i \text{ and } u_i, v_i \in \mathsf{Const}, \\ h_{\bar{u},\bar{v}}(S) & \text{otherwise.} \end{cases}$$

[1] Recall that tuples in R^S can contain both constants and variables.

Intuitively, the application of σ to S with $(\bar{u}, \bar{v})$ fails, indicated by $\perp$, whenever we have two distinct constants from Const that are supposed to be equal to satisfy σ. In case of non-failure, S' is obtained from S by simply replacing u_i and v_i by the smallest of the two, for every $i \in [k]$. Recall that, by our assumption on $<$, if one of u_i, v_i is a variable and the other one is a constant, then the variable is always replaced by the constant. The application of σ to S with $(\bar{u}, \bar{v})$, which results to S', is denoted by $S \xrightarrow{\sigma,(\bar{u},\bar{v})} S'$.

We are now ready to introduce the notion of chase sequence of a finite set S of relational atoms under a set Σ of FDs, which formalizes the objective of transforming S as dictated by Σ into a set of atoms that satisfies Σ.

Definition 10.5: The Chase for FDs

Consider a finite set S of relational atoms over a schema $\mathbf{S}$, and a set Σ of FDs over $\mathbf{S}$.

- A *finite chase sequence* of S under Σ is a finite sequence $s = S_0, \ldots, S_n$ of sets of relational atoms, where $S_0 = S$, and
  - for each $i \in [0, n-1]$, there is an FD $\sigma = R : U \to V$ in Σ and atoms $R(\bar{u}), R(\bar{v}) \in S_i$ such that $S_i \xrightarrow{\sigma,(\bar{u},\bar{v})} S_{i+1}$, and
  - either $S_n = \perp$, in which case we say that s is *failing*, or, for every FD $\sigma = R : U \to V$ in Σ and atoms $R(\bar{u}), R(\bar{v}) \in S_n$, σ is not applicable to S_n with $(\bar{u}, \bar{v})$, in which case s is called *successful*.
- An *infinite chase sequence* of S under Σ is an infinite sequence $S_0, S_1, \ldots$ of sets of relational atoms, where $S_0 = S$, and for each $i \geq 0$, there is an FD $\sigma = R : U \to V$ in Σ and atoms $R(\bar{u}), R(\bar{v}) \in S_i$ such that $S_i \xrightarrow{\sigma,(\bar{u},\bar{v})} S_{i+1}$.

We proceed to present some fundamental properties of the chase for FDs.[2] In what follows, let S be a finite set of relational atoms, and Σ a finite set of FDs, both over the same schema $\mathbf{S}$. It is not hard to see that there are no infinite chase sequences under FDs.[3] This is a consequence of the fact that each non-failing chase application does not introduce new terms but only equalizes them. Therefore, in the worst-case, the chase either will fail, or will produce after finitely many steps a set of relational atoms with only one term, which trivially satisfies every functional dependency.

Lemma 10.6. *There is no infinite chase sequence of S under Σ.*

[2] Formal proofs are omitted since in Chapter 54 we are going to present the chase for a more general class of dependencies than FDs, known as equality-generating dependencies, and provide proofs there for all the desired properties.

[3] As we discuss in Chapter 11, this is not the case for other types of dependencies, in particular, inclusion dependencies.

Although there could be several finite chase sequences of S under Σ, depending on the application order of the FDs in Σ, we can show that all those sequences either fail or end in exactly the same set of relational atoms.

Lemma 10.7. *Let $S_0, \ldots, S_n$ and $S'_0, \ldots, S'_m$ be two finite chase sequences of S under Σ. Then it holds that $S_n = S'_m$.*

The above lemma allows us to refer to *the* result of the chase of S under Σ, denoted by $\text{Chase}(S, \Sigma)$, which is defined as S_n for some (any) finite chase sequence $S_0, \ldots, S_n$ of S under Σ. Notice that we do not need to define the result of infinite chase sequences under FDs since, by Lemma 10.6, they do not exist. Hence, $\text{Chase}(S, \Sigma)$ is either the symbol $\bot$, or a finite set of relational atoms. It is not difficult to verify that in the latter case, $\text{Chase}(S, \Sigma)$ satisfies Σ. Actually, this follows from the definition of successful chase sequences.

Lemma 10.8. *If* $\text{Chase}(S, \Sigma) \neq \bot$, *then* $\text{Chase}(S, \Sigma) \models \Sigma$.

A central notion is that of chase homomorphism, which essentially computes the result of a successful finite chase sequence of S under Σ. Consider such a chase sequence $s = S_0, S_1, \ldots, S_n$ of S under Σ such that

$$S_0 \xrightarrow{\sigma_0, (\bar{u}_0, \bar{v}_0)} S_1 \xrightarrow{\sigma_1, (\bar{u}_1, \bar{v}_1)} S_2 \cdots S_{n-1} \xrightarrow{\sigma_{n-1}, (\bar{u}_{n-1}, \bar{v}_{n-1})} S_n.$$

Recall that $S_i = h_{\bar{u}_{i-1}, \bar{v}_{i-1}}(S_{i-1})$, for each $i \in [n]$. The *chase homomorphism of s*, denoted h_s, is defined as the composition of functions

$$h_{\bar{u}_{n-1}, \bar{v}_{n-1}} \circ h_{\bar{u}_{n-2}, \bar{v}_{n-2}} \circ \cdots \circ h_{\bar{u}_0, \bar{v}_0}.$$

It is clear that $h_s(S_0) = h_s(S) = S_n$. Since, by Lemma 10.7, different finite chase sequences have the same result, we get the following.

Lemma 10.9. *Let s and s' be successful finite chase sequences of S under Σ. It holds that $h_s(S) = h_{s'}(S)$.*

Therefore, assuming that $\text{Chase}(S, \Sigma) \neq \bot$, we can refer to *the* chase homomorphism of S under Σ, denoted $h_{S,\Sigma}$. It should be clear that $\text{Chase}(S, \Sigma) \neq \bot$ implies $h_{S,\Sigma}(S) = \text{Chase}(S, \Sigma)$.

By Lemma 10.6, $\text{Chase}(S, \Sigma)$ can be computed after finitely many steps. Furthermore, assuming that $\text{Chase}(S, \Sigma) \neq \bot$, also the chase homomorphism $h_{S,\Sigma}$ can be computed after finitely many steps. In fact, as the next lemma states, this is even possible after polynomially many steps.

Lemma 10.10. $\text{Chase}(S, \Sigma)$ *can be computed in polynomial time. Furthermore, if* $\text{Chase}(S, \Sigma) \neq \bot$, *then $h_{S,\Sigma}$ can be computed in polynomial time.*

The last main property of the chase states that, if $\text{Chase}(S, \Sigma) \neq \bot$, then it acts as a representative of all the sets of atoms S' that satisfy Σ and $S \to S'$, that is, there exists a homomorphism from S to S'.

Lemma 10.11. *Let S' be a set of atoms over $\boldsymbol{S}$ such that $(S, \bar{u}) \to (S', \bar{v})$ and $S' \models \Sigma$. If* $\text{Chase}(S, \Sigma) \neq \bot$*, then* $(\text{Chase}(S, \Sigma), h_{S,\Sigma}(\bar{u})) \to (S', \bar{v})$.

Note that the definition of the chase for FDs, as well as its main properties, would be technically simpler if we focus on sets of constant-free atoms since in this case there are no failing chase sequences. As we shall see, this suffices for studying the implication problem for FDs. Nevertheless, we consider sets of atoms with constants since the chase is also used in Chapter 18 for studying a different problem for which the proper treatment of constants is vital.

Implication of Functional Dependencies

We now proceed to study the implication problem for FDs, which we define next. Given a set Σ of FDs over a schema $\mathbf{S}$ and a single FD σ over $\mathbf{S}$, we say that Σ *implies* σ, denoted $\Sigma \models \sigma$, if, for every database D of $\mathbf{S}$, we have that $D \models \Sigma$ implies $D \models \sigma$. The main problem of concern is the following:

Problem: FD-Implication

Input: A set Σ of FDs over a schema $\mathbf{S}$, and an FD σ over $\mathbf{S}$
Output: `true` if $\Sigma \models \sigma$, and `false` otherwise

We proceed to show the following result:

Theorem 10.12

FD-Implication is in PTIME.

To show Theorem 10.12, we first show how implication of FDs can be characterized via the chase for FDs. This is done by showing that checking whether a set of FDs Σ implies an FD σ boils down to checking whether the result of the chase of the prototypical set of relational atoms S_σ that violates σ is a set of atoms that satisfies σ. Given an FD σ of the form $R : U \to V$, the set S_σ is defined as $\{R(x_1, \ldots, x_{\text{ar}(R)}), R(y_1, \ldots, y_{\text{ar}(R)})\}$, where

- $x_1, \ldots, x_{\text{ar}(R)}, y_1, \ldots, y_{\text{ar}(R)}$ are variables,
- for each $i, j \in \{1, \ldots, \text{ar}(R)\}$ with $i \neq j$, $x_i \neq x_j$ and $y_i \neq y_j$, and
- for each $i \in \{1, \ldots, \text{ar}(R)\}$, $x_i = y_i$ if and only if $i \in U$.

We can now show the following useful characterization:

Proposition 10.13

Consider a set Σ of FDs over as schema $\mathbf{S}$, and an FD σ over $\mathbf{S}$. Then:

$$\Sigma \models \sigma \quad \text{if and only if} \quad \text{Chase}(S_\sigma, \Sigma) \models \sigma.$$

Proof. ($\Rightarrow$) By hypothesis, for every finite set of relational atoms S, it holds that $S \models \Sigma$ implies $S \models \sigma$. Observe that $\text{Chase}(S_\sigma, \Sigma) \neq \bot$ since S_σ contains only variables. Therefore, by Lemma 10.8, we have that $\text{Chase}(S_\sigma, \Sigma) \models \Sigma$. Since, by Lemma 10.6, $\text{Chase}(S_\sigma, \Sigma)$ is finite, we get that $\text{Chase}(S_\sigma, \Sigma) \models \sigma$.

($\Leftarrow$) Consider now a database D of $\mathbf{S}$ such that $D \models \Sigma$, and with σ being of the form $R : \{i_1, \ldots, i_k\} \to \{j_1, \ldots, j_\ell\}$, assume that there are tuples $(a_1, \ldots, a_{\text{ar}(R)}), (b_1, \ldots, b_{\text{ar}(R)}) \in R^D$ such that $(a_{i_1}, \ldots, a_{i_k}) = (b_{i_1}, \ldots, b_{i_k})$. Recall also that S_σ is of the form $\{R(x_1, \ldots, x_{\text{ar}(R)}), R(y_1, \ldots, y_{\text{ar}(R)})\}$. Let $\bar{z} = (x_{j_1}, \ldots, x_{j_\ell}, y_{j_1}, \ldots, y_{j_\ell})$ and $\bar{c} = (a_{j_1}, \ldots, a_{j_\ell}, b_{j_1}, \ldots, b_{j_\ell})$. It is clear that $(S_\sigma, \bar{z}) \to (D, \bar{c})$. Since $D \models \Sigma$ and $\text{Chase}(S_\sigma, \Sigma) \neq \bot$, by Lemma 10.11

$$(\text{Chase}(S_\sigma, \Sigma), h_{S_\sigma, \Sigma}(\bar{z})) \;\to\; (D, \bar{c}).$$

Since, by hypothesis, $\text{Chase}(S_\sigma, \Sigma) \models \Sigma$, we can conclude that

$$(h_{S_\sigma, \Sigma}(x_{j_1}), \ldots, h_{S_\sigma, \Sigma}(x_{j_\ell})) \;=\; (h_{S_\sigma, \Sigma}(y_{j_1}), \ldots, h_{S_\sigma, \Sigma}(y_{j_\ell})),$$

which in turn implies that

$$(a_{j_1}, \ldots, a_{j_\ell}) \;=\; (b_{j_1}, \ldots, b_{j_\ell}).$$

Therefore, $D \models \sigma$, and the claim follows. □

By Proposition 10.13, we get a simple procedure for checking whether a set Σ of FDs implies an FD σ that runs in polynomial time:

$$\text{if } \text{Chase}(S_\sigma, \Sigma) \models \sigma, \text{then return } \texttt{true}\text{; otherwise, return } \texttt{false}.$$

We know that the set of atoms $\text{Chase}(S_\sigma, \Sigma)$ can be constructed in polynomial time (Lemma 10.10), and we also know that $\text{Chase}(S_\sigma, \Sigma) \models \sigma$ can be checked in polynomial time (Theorem 10.4), and Theorem 10.12 follows.

11

Inclusion Dependencies

In this chapter, we concentrate on another central class of constraints supported by relational database systems, called *inclusion dependencies* (also known as *referential constraints*). With this type of constraints we can express relationships among attributes of different relations, which is not possible using functional dependencies that can talk only about one relation.

Example 11.1: Inclusion Dependencies

Having the (named) database schema

```
Person [ pid, pname, cid ]
Profession [ pid, prname ]
```

we would like to express that the values occurring in the first attribute of Profession are person ids. This can be done via the inclusion dependency

$$\text{Profession}[1] \subseteq \text{Person}[1].$$

This dependency simply states that the set of values occurring in the first attribute of the relation Profession should be a subset of the set of values appearing in the first attribute of the relation Person.

Syntax and Semantics

We start with the syntax of inclusion dependencies.

Definition 11.2: Syntax of Inclusion Dependencies

An *inclusion dependency* (IND) σ over a schema $\mathbf{S}$ is an expression

$$R[i_1, \ldots, i_k] \subseteq P[j_1, \ldots, j_k]$$

where $k \geq 1$, R, P belong to $\mathbf{S}$, and $(i_1, \ldots, i_k)$ and $(j_1, \ldots, j_k)$ are lists of distinct integers from $\{1, \ldots, \mathrm{ar}(R)\}$ and $\{1, \ldots, \mathrm{ar}(P)\}$, respectively.

Intuitively, an IND $R[i_1, \ldots, i_k] \subseteq P[j_1, \ldots, j_k]$ states that if $R(\bar{a})$ belongs to a database D, then in the same database an atom $P(\bar{b})$ should exist such that the i_ℓ-th element of $\bar{a}$ coincides with the j_ℓ-th element of $\bar{b}$, for $\ell \in [k]$. The formal definition of the semantic meaning of INDs follows.

Definition 11.3: Semantics of INDs

A database D of a schema $\mathbf{S}$ *satisfies* an IND σ of the form $R[i_1, \ldots, i_k] \subseteq P[j_1, \ldots, j_k]$ over $\mathbf{S}$, denoted $D \models \sigma$, if for every tuple $\bar{a} \in R^D$, there exists a tuple $\bar{b} \in P^D$ such that

$$\pi_{(i_1,\ldots,i_k)}(\bar{a}) = \pi_{(j_1,\ldots,j_k)}(\bar{b}).$$

D *satisfies* a set Σ of INDs, denoted $D \models \Sigma$, if $D \models \sigma$ for each $\sigma \in \Sigma$.

Satisfaction of Inclusion Dependencies

A central task is checking whether a database D satisfies a set Σ of INDs.

Problem: IND-Satisfaction

Input: A database D over a schema $\mathbf{S}$, and a set Σ of INDs over $\mathbf{S}$
Output: `true` if $D \models \Sigma$, and `false` otherwise

It is not difficult to show the following result:

Theorem 11.4

IND-Satisfaction is in PTIME.

Proof. Consider a database D of a schema $\mathbf{S}$, and a set Σ of INDs over $\mathbf{S}$. Let σ be an IND from Σ of the form $R[i_1, \ldots, i_k] \subseteq P[j_1, \ldots, j_k]$. To check whether $D \models \sigma$ we need to check that, for every tuple $(a_1, \ldots, a_{\mathrm{ar}(R)}) \in R^D$, there exists a tuple $(b_1, \ldots, b_{\mathrm{ar}(P)}) \in P^D$ such that $(a_{i_1}, \ldots, a_{i_k}) = (b_{j_1}, \ldots, b_{j_k})$. It is not difficult to verify that this can be done in time $O(\|D\|^2)$. Therefore, we can check whether $D \models \Sigma$ in time $O(\|\Sigma\| \cdot \|D\|^2)$, and the claim follows. □

The Chase for Inclusion Dependencies

As for FDs, the other crucial task of interest in connection with INDs is (logical) implication. Unsurprisingly, the main tool for studying the implication problem for INDs is the chase for INDs, which we introduce next.

Consider a finite set S of atoms over $\mathbf{S}$, and an IND $\sigma = R[i_1, \ldots, i_m] \subseteq P[j_1, \ldots, j_m]$ over $\mathbf{S}$. We say that σ is *applicable to S with $\bar{u} = (u_1, \ldots, u_{ar(R)})$* if $\bar{u} \in R^S$. Let $\mathsf{new}(\sigma, \bar{u}) = P(v_1, \ldots, v_{\mathrm{ar}(P)})$, where, for each $\ell \in [\mathrm{ar}(P)]$,

$$v_\ell = \begin{cases} u_{i_k} & \text{if } \ell = j_k, \text{ for } k \in [m], \\ x_\ell^{\sigma, \pi_{(i_1, \ldots, i_m)}(\bar{u})} & \text{otherwise,} \end{cases}$$

with $x_\ell^{\sigma, \pi_{(i_1, \ldots, i_m)}(\bar{u})} \in \mathsf{Var} - \mathrm{Dom}(S)$.[1] The *result of applying σ to S with $\bar{u}$* is the set of atoms $S' = S \cup \{\mathsf{new}(\sigma, \bar{u})\}$. In simple words, S' is obtained from S by adding the new atom $\mathsf{new}(\sigma, \bar{u})$, which is uniquely determined by σ and $\bar{u}$. The application of σ to S with $\bar{u}$, which results in S', is denoted $S \xrightarrow{\sigma, \bar{u}} S'$.

We are now ready to introduce the notion of chase sequence of a finite set S of relational atoms under a set Σ of INDs, which formalizes the objective of transforming S as dictated by Σ into a set of atoms that satisfies Σ.

Definition 11.5: The Chase for INDs

Consider a finite set S of relational atoms over a schema $\mathbf{S}$, and a set Σ of INDs over $\mathbf{S}$.

- A *finite chase sequence* of S under Σ is a finite sequence $s = S_0, \ldots, S_n$ of sets of relational atoms, where $S_0 = S$, and
  1. for each $i \in [0, n-1]$, there is $\sigma = R[\alpha] \subseteq P[\beta]$ in Σ and $\bar{u} \in R^{S_i}$ such that $\mathsf{new}(\sigma, \bar{u}) \notin S_i$ and $S_i \xrightarrow{\sigma, \bar{u}} S_{i+1}$, and
  2. for each IND $\sigma = R[\alpha] \subseteq P[\beta]$ in Σ and $\bar{u} \in R^{S_n}$, $\mathsf{new}(\sigma, \bar{u}) \in S_n$.

  The *result* of s is defined as the set of atoms S_n.
- An *infinite chase sequence* of S under Σ is an infinite sequence $s = S_0, S_1, \ldots$ of sets of atoms, where $S_0 = S$, and
  1. for each $i \geq 0$, there is $\sigma = R[\alpha] \subseteq P[\beta]$ in Σ and $\bar{u} \in R^{S_i}$ such that $\mathsf{new}(\sigma, \bar{u}) \notin S_i$ and $S_i \xrightarrow{\sigma, \bar{u}} S_{i+1}$, and
  2. for each $i \geq 0$, and for each $\sigma = R[\alpha] \subseteq P[\beta]$ in Σ and $\bar{u} \in R^{S_i}$ such that σ is applicable to S_i with $\bar{u}$, there exists $j > i$ such that $\mathsf{new}(\sigma, \bar{u}) \in S_j$.

[1] One could adopt a simpler naming scheme for these newly introduced variables. For example, for each $\ell \in [\mathrm{ar}(P)] - \{j_1, \ldots, j_m\}$, we could simply name the new variable $x_\ell^{\sigma, \bar{u}}$. For further details on this matter see the comments for Part I.

The *result* of s is defined as the set infinite set of atoms $\bigcup_{i \geq 0} S_i$.

In the case of finite chase sequences, the first condition in Definition 11.5 simply says that S_{i+1} is obtained from S_i by applying σ to S_i with $\bar{u}$, while σ has not been already applied to some S_j, for $j < i$, with $\bar{u}$. The second condition states that no new atom, which is not already in S_n, can be derived by applying an IND of Σ to S_n. Now, in the case of infinite chase sequences, the first condition in Definition 11.5, as in the finite case, says that S_{i+1} is obtained from S_i by applying σ to S_i with $\bar{u}$, while σ has not been already applied before. The second condition is known as the *fairness condition*, and it ensures that all the INDs that are applicable eventually will be applied.

We proceed to show some fundamental properties of the chase for INDs.[2] In what follows, let S be a finite set of relational atoms, and Σ a finite set of INDs, both over the same schema $\mathbf{S}$. Recall that in the case of FDs we know that there are no infinite chase sequences since a chase application does not introduce new terms, but only equalizes terms. However, in the case of INDs, a chase step may introduce new variables not occurring in the given set of atoms, which may lead to infinite chase sequences. Indeed, this can happen even for simple sets of atoms and INDs. For example, it is not hard to verify that the single chase sequence of $\{R(a, b)\}$ under $\{R[2] \subseteq R[1]\}$ is infinite.

Although we may have infinite chase sequences, we can still establish some favourable properties. It is clear that there are several chase sequences of S under Σ depending on the order that we apply the INDs of Σ. However, the adopted naming scheme of new variables ensures that, no matter when we apply an IND σ with a tuple $\bar{u}$, the newly generated atom $\mathsf{new}(\sigma, \bar{u})$ is always the same, which in turn allows us to show that all those chase sequences have the same result. At this point, let us stress that the result of an infinite chase sequence $s = S_0, S_1, \ldots$ of S under Σ always exists.[3] This can be shown by exploiting classical results of fixpoint theory. By using Kleene's Theorem, we can show that $\bigcup_{i \geq 0} S_i$ coincides with the least fixpoint of a continuous operator (which corresponds to a single chase step) on the complete lattice $(\mathrm{Inst}(\mathbf{S}), \subseteq)$, which we know that always exists by Knaster-Tarski's Theorem (we leave the proof as an exercise). We can now state the announced result.

Lemma 11.6. *The following hold:*

1. *There exists a finite chase sequence of S under Σ if and only if there is no infinite chase sequence of S under Σ.*
2. *Let $S_0, \ldots, S_n$ and $S'_0, \ldots, S'_m$ be two finite chase sequences of S under Σ. Then, it holds that $S_n = S'_m$.*

[2] Formal proofs are omitted since in Chapter 45 we are going to present the chase for a more general class of dependencies than INDs, known as tuple-generating dependencies, and provide proofs there for all the desired properties.

[3] This statement trivially holds for finite chase sequences.

3. *Let* $S_0, S_1, \ldots$ *and* $S'_0, S'_1, \ldots$ *be two infinite chase sequences of* S *under* Σ*. Then, it holds that* $\bigcup_{i \geq 0} S_i = \bigcup_{i \geq 0} S'_i$*.*

Lemma 11.6 allows us to refer to *the* unique result of the chase of S under Σ, denoted $\text{Chase}(S, \Sigma)$, which is defined as the result of some (any) finite or infinite chase sequence of S under Σ. At this point, the reader may expect that the next key property is that $\text{Chase}(S, \Sigma)$ satisfies Σ. However, it should not be overlooked that $\text{Chase}(S, \Sigma)$ is a possibly infinite set of atoms, and thus, we cannot directly apply the notion of satisfaction from Definition 11.3. Nevertheless, Definition 11.3 can be readily applied to possibly infinite databases, which in turn allows us to transfer the notion of satisfaction for INDs to sets of atoms via the notion of grounding. In particular, a set of atoms S satisfies an IND σ, denoted $S \models \sigma$, if $S^{\downarrow} \models \sigma$, while S satisfies a set Σ of INDs, written $S \models \Sigma$, if $S^{\downarrow} \models \Sigma$. We can now formally state that $\text{Chase}(S, \Sigma)$ satisfies Σ. Let us clarify, though, that in the case where only infinite chase sequences exist, this result heavily relies on the fairness condition.

Lemma 11.7. *It holds that* $\text{Chase}(S, \Sigma) \models \Sigma$*.*

The last crucial property states that $\text{Chase}(S, \Sigma)$ acts as a representative of all the finite or infinite sets of atoms S' that satisfy Σ, and such that there exists a homomorphism from S to S', that is, $S \to S'$.

Lemma 11.8. *Let* S' *be a set of atoms over* $\boldsymbol{S}$ *such that* $(S, \bar{u}) \to (S', \bar{v})$ *and* $S' \models \Sigma$*. It holds that* $(\text{Chase}(S, \Sigma), \bar{u}) \to (S', \bar{v})$*.*

Implication of Inclusion Dependencies

We now proceed to study the implication problem for INDs. The notion of implication for INDs is defined in the same way as for functional dependencies. More precisely, given a set Σ of INDs over a schema $\mathbf{S}$ and a single IND σ over $\mathbf{S}$, we say that Σ *implies* σ, denoted $\Sigma \models \sigma$, if, for every database D of $\mathbf{S}$, we have that $D \models \Sigma$ implies $D \models \sigma$. This leads to the following problem:

Problem: IND-Implication

Input: A set Σ of INDs over a schema $\mathbf{S}$, and an IND σ over $\mathbf{S}$
Output: `true` if $\Sigma \models \sigma$, and `false` otherwise

Although for FDs the implication problem is tractable (Theorem 10.12), for INDs it turns out to be an intractable problem:

Theorem 11.9

IND-Implication is PSPACE-complete.

We first concentrate on the upper bound. We are going to establish a result, analogous to Proposition 10.13 for FDs, that characterizes implication of INDs via the chase. However, since the chase for INDs may build an infinite set of atoms, we can only characterize implication under possibly infinite databases. Given a set Σ of INDs over a schema $\mathbf{S}$ and a single IND σ over $\mathbf{S}$, we say that Σ *implies without restriction* σ, denoted $\Sigma \models^{\infty} \sigma$, if, for every possibly infinite database D of $\mathbf{S}$, we have that $D \models \Sigma$ implies $D \models \sigma$.

Given an IND σ of the form $R[i_1, \ldots, i_k] \subseteq P[j_1, \ldots, j_k]$, the set S_σ is defined as the singleton $\{R(x_1, \ldots, x_{\text{ar}(R)})\}$, where $x_1, \ldots, x_{\text{ar}(R)}$ are distinct variables. We can now show the following auxiliary lemma.

Lemma 11.10. *Consider a set Σ of INDs over schema $\boldsymbol{S}$, and an IND σ over $\boldsymbol{S}$. It holds that $\Sigma \models^{\infty} \sigma$ if and only if* $\text{Chase}(S_\sigma, \Sigma) \models \sigma$.

Proof. ($\Rightarrow$) By hypothesis, for every possibly infinite set of relational atoms S, it holds that $S \models \Sigma$ implies $S \models \sigma$. By Lemma 11.7, $\text{Chase}(S_\sigma, \Sigma) \models \Sigma$, and therefore, $\text{Chase}(S_\sigma, \Sigma) \models \sigma$.

($\Leftarrow$) Consider now a possibly infinite database D of $\mathbf{S}$ such that $D \models \Sigma$, and with σ being of the form $R[i_1, \ldots, i_k] \subseteq P[j_1, \ldots, j_k]$, assume that there exists a tuple $(a_1, \ldots, a_{\text{ar}(R)}) \in R^D$. Recall also that S_σ is of the form $\{R(x_1, \ldots, x_{\text{ar}(R)})\}$. Let $\bar{y} = (x_{i_1}, \ldots, x_{i_k})$ and $\bar{b} = (a_{i_1}, \ldots, a_{i_k})$. It is clear that $(S_\sigma, \bar{y}) \to (D, \bar{b})$. Since $D \models \Sigma$, by Lemma 11.8

$$(\text{Chase}(S_\sigma, \Sigma), \bar{y}) \;\to\; (D, \bar{b}).$$

Since, by hypothesis, $\text{Chase}(S_\sigma, \Sigma) \models \sigma$, we can conclude that there exists a tuple $(z_1, \ldots, z_{\text{ar}(P)}) \in P^{\text{Chase}(S_\sigma, \Sigma)}$ such that

$$(x_{i_1}, \ldots, x_{i_k}) \;=\; (z_{j_1}, \ldots, z_{j_k}),$$

which in turn implies that there exists $(c_1, \ldots, c_{\text{ar}(P)}) \in P^D$ such that

$$(a_{i_1}, \ldots, a_{i_k}) \;=\; (c_{j_1}, \ldots, c_{j_k}).$$

Therefore, $D \models \sigma$, and the claim follows. □

Lemma 11.10 alone is of little use since it characterizes implication of INDs under possibly infinite databases, whereas we are interested only in (finite) databases. However, we can show that implication of INDs is *finitely controllable*, which means that implication under finite databases ($\models$) and implication under possibly infinite databases ($\models^{\infty}$) coincide.

Theorem 11.11: Finite Controllability of Implication

Consider a set Σ of INDs over as schema $\mathbf{S}$, and an IND σ over $\mathbf{S}$. Then:

$$\Sigma \models \sigma \quad \text{if and only if} \quad \Sigma \models^{\infty} \sigma.$$

Although the above theorem is crucial for our analysis, we do not discuss its proof here (see Exercise 1.15). An immediate consequence of Lemma 11.10 and Theorem 11.11 is the following:

Corollary 11.12

Consider a set Σ of INDs over a schema $\mathbf{S}$, and an IND σ over $\mathbf{S}$. Then:

$$\Sigma \models \sigma \quad \text{if and only if} \quad \text{Chase}(S_\sigma, \Sigma) \models \sigma.$$

Due to Corollary 11.12, the reader may think that the procedure for checking whether $\Sigma \models \sigma$, which will lead to the PSPACE upper bound claimed in Theorem 11.9, is simply to construct the set of atoms $\text{Chase}(S_\sigma, \Sigma)$, and then check whether it satisfies σ, which can be achieved due to Theorem 11.4. However, it should not be forgotten that $\text{Chase}(S_\sigma, \Sigma)$ may be infinite. Therefore, we need to rely on a finer procedure that avoids the explicit construction of $\text{Chase}(S_\sigma, \Sigma)$. We proceed to present a technical lemma that is the building block of this refined procedure, but first we need some terminology.

Given an IND $\sigma = R[i_1, \ldots, i_m] \subseteq P[j_1, \ldots, j_m]$ and a tuple of variables $\bar{x} = (x_1, \ldots, x_{\text{ar}(R)})$, we define the atom $\mathsf{new}^\star(\sigma, \bar{x})$ as the atom obtained from $\mathsf{new}(\sigma, \bar{x})$ after replacing the newly introduced variables with the special variable $\star \notin \{x_1, \ldots, x_{\text{ar}(R)}\}$, which should be understood as a placeholder for new variables. Formally, $\mathsf{new}^\star(\sigma, \bar{x}) = P(y_1, \ldots, y_{\text{ar}(P)})$, where, for $\ell \in [\text{ar}(P)]$,

$$y_\ell = \begin{cases} x_{i_k} & \text{if } \ell = j_k, \text{ for } k \in [m], \\ \star & \text{otherwise.} \end{cases}$$

Given a set Σ of INDs, a *witness of σ relative to Σ* is a sequence of atoms $R_1(\bar{x}_1), \ldots, R_n(\bar{x}_n)$, for $n \geq 1$, such that:

- $S_\sigma = \{R_1(\bar{x}_1)\}$,
- for each $i \in [2, n]$, there is $\sigma_i = R_{i-1}[\alpha_{i-1}] \subseteq R_i[\alpha_i]$ in Σ that is applicable to $\{R_{i-1}(\bar{x}_{i-1})\}$ with $\bar{x}_{i-1}$ such that $R_i(\bar{x}_i) = \mathsf{new}^\star(\sigma_i, \bar{x}_{i-1})$,
- $R_n = P$, and
- $\pi_{(i_1,\ldots,i_m)}(\bar{x}_1) = \pi_{(j_1,\ldots,j_m)}(\bar{x}_n)$.

A witness of σ relative to Σ is essentially a compact representation, which uses only $\text{ar}(R) + 1$ variables, of a sequence of atoms of $\text{Chase}(S_\sigma, \Sigma)$ that

Algorithm 2 ImplicationWitness(Σ, σ)

Input: A set Σ of INDs over $\mathbf{S}$ and $\sigma = R[i_1, \ldots, i_k] \subseteq P[j_1, \ldots, j_k]$ over $\mathbf{S}$.
Output: `true` if there is a witness of σ relative Σ, and `false` otherwise.

1: **if** $R = P$ and $(i_1, \ldots, i_k) = (j_1, \ldots, j_k)$ **then**
2: **return** `true`
3: $S_\triangledown := \{R(\bar{x})\}$, where $\bar{x} = (x_1, \ldots, x_{\mathrm{ar}(R)})$ consists of distinct variables
4: $S_\triangleright := \emptyset$
5: **repeat**
6: **if** $\sigma' = T[\alpha] \subseteq T'[\beta] \in \Sigma$ is applicable to $S_\triangledown$ with $\bar{y} \in \mathrm{Dom}(S_\triangledown)^{\mathrm{ar}(T)}$ **then**
7: $S_\triangleright := \{\mathsf{new}^\star(\sigma', \bar{y})\}$
8: **if** $S_\triangleright = \emptyset$ **then**
9: **return** `false`
10: $S_\triangledown := S_\triangleright$
11: $S_\triangleright := \emptyset$
12: *Check* $:= b$, where $b \in \{0, 1\}$
13: **until** *Check* $= 1$
14: **return** $(T' = P \wedge \pi_{(i_1,\ldots,i_k)}(\bar{x}) = \pi_{(j_1,\ldots,j_k)}(\bar{z}))$

witnesses the following: starting from $S_\sigma = \{R(x_1, \ldots, x_{\mathrm{ar}(R)})\}$, an atom $P(y_1, \ldots, y_{\mathrm{ar}(P)})$ with $\pi_{(i_1,\ldots,i_m)}(x_1, \ldots, x_{\mathrm{ar}(R)}) = \pi_{(j_1,\ldots,j_m)}(y_1, \ldots, y_{\mathrm{ar}(P)})$ can be obtained via chase applications, which means that $\mathrm{Chase}(S_\sigma, \Sigma) \models \sigma$. It is also not difficult to see that if $\mathrm{Chase}(S_\sigma, \Sigma) \models \sigma$, then a witness of σ relative to Σ can be extracted from $\mathrm{Chase}(S_\sigma, \Sigma)$. This discussion is summarized in the following technical lemma, whose proof is left as an exercise.

Lemma 11.13. *Consider a set Σ of INDs over a schema $\boldsymbol{S}$, and an IND σ over $\boldsymbol{S}$. Then,* $\mathrm{Chase}(S_\sigma, \Sigma) \models \sigma$ *iff there is a witness of σ relative to Σ.*

By Corollary 11.12 and Lemma 11.13, we have that the problem of checking whether a set Σ of INDs over a schema $\mathbf{S}$ implies a single IND σ over $\mathbf{S}$, boils down to checking whether a witness of σ relative to Σ exists. This is done via the nondeterministic procedure depicted in Algorithm 2. Assume that σ is of the form $R[i_1, \ldots, i_k] \subseteq P[j_1, \ldots, j_k]$. The algorithm first checks whether $R[i_1, \ldots, i_k] = P[j_1, \ldots, j_k]$, in which case a witness of σ relative to Σ trivially exists, and returns `true`. Otherwise, it proceeds to nondeterministically construct a witness of σ relative to Σ (if one exists). This is done by constructing one atom after the other via chase steps, without having to store more than two consecutive atoms. The algorithm starts from $S_\triangledown = \{R(x_1, \ldots, R_{\mathrm{ar}(R)})\}$; $S_\triangledown$ should be understood as the "current atom", which at the beginning is S_σ, from which we construct the "next atom" $S_\triangleright$ in the sequence. The repeat-until loop is responsible for constructing $S_\triangleright$ from $S_\triangledown$. This is done by guessing an IND $\sigma' \in \Sigma$, and adding to $S_\triangleright$ the atom $\mathsf{new}^\star(\sigma', \bar{y})$ if σ' is applicable to $S_\triangledown$ with $\bar{y}$; note that $\bar{y}$ is the single tuple occurring in $S_\triangledown$. This is repeated until the algorithm chooses to exit the loop by setting *Check* to 1, and check whether

$S_\triangleright$ consists of an atom $T'(\bar{z})$ with $T' = P$ and $\pi_{(i_1,\ldots,i_k)}(\bar{x}) = \pi_{(j_1,\ldots,j_k)}(\bar{z})$, in which case it returns `true`; otherwise, it returns `false`.

It is easy to verify that Algorithm 2 uses polynomial space. This heavily relies on the fact that the atoms generated during its computation contain only variables from $\{x_1, \ldots, x_{\mathrm{ar}(R)}\}$ and the special variable $\star$, which in turn implies that $S_\triangledown$ and $S_\triangleright$ can be represented using polynomial space. It also takes polynomial space to check if $R[i_1, \ldots, i_k] = P[j_1, \ldots, j_k]$ (see line 1), to check if an IND is applicable to $S_\triangledown$ with $\bar{y}$ (see line 6), and to check if $T' = P$ and $\pi_{(i_1,\ldots,i_k)}(\bar{x}) = \pi_{(j_1,\ldots,j_k)}(\bar{z})$ (see line 14). Therefore, IND-Implication is in NPSpace, and thus in PSpace since NPSpace = PSpace.

A PSpace lower bound for IND-Implication can be shown via a reduction from the following PSpace-hard problem: given 2-TM M that runs in linear space, and a word w over the alphabet of M, decide whether M accepts input w. The formal proof is left as Exercise 1.17.

12

Exercises for Part I

Exercise 1.1. Let q be an FO query. Prove that one can compute in polynomial time an FO query q' that uses only $\neg$, $\vee$, and $\exists$ such that $q \equiv q'$.

Exercise 1.2. We say that a query q from a database schema $\mathbf{S}$ to a relation schema $\mathbf{S}'$ is C-*generic*, for some $C \subseteq \mathsf{Const}$, if for every database D of $\mathbf{S}$, and for every bijection $\rho : \mathsf{Const} \to \mathsf{Const}$ that is the identity on C, $q(\rho(D)) = \rho(q(D))$. Show that an FO query $\varphi(\bar{x})$ over a schema $\mathbf{S}$ is $\mathrm{Dom}(\varphi)$-generic.

Exercise 1.3. The semantics of the rename and join operations in the named RA has been defined in Chapter 4. Provide formal definitions for the semantics of the other operations, i.e., selection, projection, union, and difference.

Exercise 1.4. State and prove the converse of Theorem 4.6.

Exercise 1.5. Prove that allowing conditions of the form $\bar{a} \in e$ and $\mathrm{empty}(e)$ in selection conditions of RA does not increase its expressiveness. In particular, show that selections with these new conditions can be expressed using standard operations of RA.

Exercise 1.6. Prove that adding nested subqueries in the `FROM` clause does not increase expressiveness. In particular, extend the translation from basic SQL to RA that handles nested subqueries in `FROM`.

Exercise 1.7. The proofs of Theorems 7.1 and 7.3 only consider the special case of FO-Evaluation where the input tuple $\bar{a}$ is over $\mathrm{Dom}(D)$. In this case, the complexity analysis is easier, because the size of $\bar{a}$ is subsumed by the size of the database. How can the proof be extended to FO-Evaluation in general, i.e., allowing arbitrary tuples $\bar{a}$ over Const?

Hint: If $\bar{a}$ contains a value not in the active domain, what should FO-Evaluation return?

Exercise 1.8. For showing that FO-Evaluation is PSPACE-hard, we provided a reduction from QSAT. In particular, for an input to QSAT given by ψ, we constructed a database D and an FO query q_ψ (see the proof of Theorem 7.1). Show that ψ is satisfiable if and only if $D \models q_\psi$.

Exercise 1.9. For an integer $k > 0$, we write FO_k for the class of FO queries that can mention at most k variables. The evaluation problem for the class of FO_k queries, for some fixed $k > 0$, is defined as expected: given an FO_k query q, a database D, and a tuple $\bar{a}$, decide whether $\bar{a} \in q(D)$. Show that the evaluation problem for FO_k queries, for a fixed $k > 0$, is in PTime.

Exercise 1.10. Let q_M be the Boolean FO query constructed in the proof of Theorem 8.1. Prove that if the Turing machine M on the empty word does not halt, then there exists an infinite database D such that $q(D) = \texttt{true}$.

Exercise 1.11. Let FO-Unrestricted-Satisfiability be the unrestricted version of FO-Satisfiability where we consider possibly infinite databases. In other words, FO-Unrestricted-Satisfiability is defined as follows: given an FO query q, is there a possibly infinite database D such that $q(D) \neq \emptyset$? Show that FO-Unrestricted-Satisfiability is undecidable by adapting the proof of Theorem 8.1.

Exercise 1.12. Prove that FO-Containment remains undecidable even if the left hand-side query is a Boolean query $q = \exists \bar{x}\, \varphi$, where φ is a conjunction of relational atoms or the negation of relational atoms.

Exercise 1.13. The algorithms underlying Theorems 10.4 and 11.4 for checking whether a database satisfies a set of FDs and INDs, respectively, were designed with simplicity instead of efficiency in mind. Provide more efficient algorithms for the problems FD-Satisfaction and IND-Satisfaction.

Exercise 1.14. Prove that the result of an infinite chase sequence of a finite set of relational atoms under a set of INDs always exists.

Exercise 1.15. Prove Theorem 11.11. The non-trivial task is to show that if $\Sigma \models^{\infty} \sigma$ does not hold, then also $\Sigma \models \sigma$ does not hold. One can exploit Lemma 11.10, which states that if $\Sigma \models^{\infty} \sigma$ does not hold, then $\mathrm{Chase}(S_\sigma, \Sigma)$ does not satisfy σ. If $\mathrm{Chase}(S_\sigma, \Sigma)$ is finite, then we have that $\Sigma \models \sigma$ does not hold. The main task is, when $\mathrm{Chase}(S_\sigma, \Sigma)$ is infinite, to convert $\mathrm{Chase}(S_\sigma, \Sigma)$ into a finite set S such that $S \models \Sigma$, but S does not satisfy σ.

Exercise 1.16. Prove Lemma 11.13.

Exercise 1.17. Prove that IND-Implication is PSpace-hard. To this end, provide a reduction from the following PSpace-hard problem: given 2-TM M that runs in linear space, and a word w over the alphabet of M, decide whether M accepts input w.

Bibliographic Comments for Part I

To be done.

Part II

Conjunctive Queries

13

Syntax and Semantics

Conjunctive queries are of special importance to databases. They express relational joins, which correspond to the operation that is most commonly performed by relational database engines. This is because data is typically spread over multiple relations, and thus, to answer queries, one needs to join such relations. Actually, conjunctive queries have the power of select-project-join RA queries, which means that they correspond to a very common type of queries written in Core SQL. The goal of this chapter is to introduce the syntax and semantics of conjunctive queries.

Syntax of Conjunctive Queries

We start with the syntax of conjunctive queries.

Definition 13.1: Syntax of Conjunctive Queries

A *conjunctive query* (CQ) over a schema $\mathbf{S}$ is an FO query $\varphi(\bar{x})$ over $\mathbf{S}$ with φ being a formula of the form

$$\exists \bar{y} \left(R_1(\bar{u}_1) \wedge \cdots \wedge R_n(\bar{u}_n) \right)$$

for $n \geq 1$, where $R_i(\bar{u}_i)$ is a relational atom, and $\bar{u}_i$ a tuple of constants and variables mentioned in $\bar{x}$ and $\bar{y}$, for every $i \in [n]$.

It is very common to represent CQs via a rule-like syntax, which is reminiscent of the syntax of logic programming rules. In particular, the CQ $\varphi(\bar{x})$ given in Definition 13.1 can be written as the *rule*

$$\text{Answer}(\bar{x}) \ \text{:-} \ R_1(\bar{u}_1), \ldots, R_n(\bar{u}_n) \,,$$

where Answer is a relation name not in $\mathbf{S}$, and its arity (under the singleton schema $\{\text{Answer}\}$) is equal to the arity of q. The relational atom $\text{Answer}(\bar{x})$

that appears on the left of the :– symbol is called the *head* of the rule, while the expression $R_1(\bar{u}_1), \ldots, R_n(\bar{u}_n)$ that appears on the right of the :– symbol is called the *body* of the rule. In general, throughout the book, we use the rule-like syntax for CQs. Nevertheless, for convenience, we will freely interpret a CQ as a first-order query or as a rule.

Example 13.2: Conjunctive Queries

Consider again the relational schema from Example 3.2:

```
Person [ pid, pname, cid ]
Profession [ pid, prname ]
City [ cid, cname, country ]
```

The following CQ can be used to retrieve the list of names of computer scientists that were born in the city of Athens in Greece:

$$\exists x \exists z \big(\text{Person}(x, y, z) \ \wedge \ \text{Profession}(x, \text{`computer scientist'}) \wedge \text{City}(z, \text{`Athens'}, \text{`Greece'})\big).$$

In rule-like representation, this query is expressed as follows:

$$\begin{aligned} \text{Answer}(y) \ \text{:–} \ & \text{Person}(x, y, z), \text{Profession}(x, \text{`computer scientist'}), \\ & \text{City}(z, \text{`Athens'}, \text{`Greece'}). \end{aligned}$$

A CQ q is *Boolean* if it has no output variables, i.e., $\bar{x}$ is the empty tuple. When we write a Boolean CQ as a rule, we simply write Answer as the head, instead of Answer(). For example, the following Boolean CQ checks whether there exists a computer scientist that was born in the city of Athens in Greece:

$$\begin{aligned} \text{Answer} \ \text{:–} \ & \text{Person}(x, y, z), \text{Profession}(x, \text{`computer scientist'}), \\ & \text{City}(z, \text{`Athens'}, \text{`Greece'}). \end{aligned}$$

Semantics of Conjunctive Queries

Since CQs are FO queries, the definition of their output on a database can be inherited from Definition 3.6. More precisely, given a database D of a schema $\mathbf{S}$, and a k-ary CQ $q = \varphi(\bar{x})$ over $\mathbf{S}$, where $k \geq 0$, the *output* of q on D is

$$q(D) \ = \ \{\bar{a} \in \text{Dom}(D)^k \mid D \models \varphi(\bar{a})\}.$$

Notice that $q(D)$ consists of tuples over $\text{Dom}(D)$, not over $\text{Dom}(D) \cup \text{Dom}(\varphi)$. This is because CQs do not allow for equational atoms, and thus, there is no way for a constant of $\text{Dom}(\varphi) - \text{Dom}(D)$ to appear in the output.

Interestingly, there is a more intuitive (and equivalent) way of defining the semantics of CQs when they are viewed as rules. The body of a CQ q of the form $\text{Answer}(\bar{x})$:– body can be seen as a pattern that must be matched with the database D via an assignment η that maps the variables in q to $\text{Dom}(D)$. For each such assignment η, if η applied to this pattern produces only facts of D, it means that the pattern matches with D via η, and the tuple $\eta(\bar{x})$ is an output of q on D. We proceed to formalize this informal description.

Consider a database D and a CQ q of the form

$$\text{Answer}(\bar{x}) \;\text{ :– }\; R_1(\bar{u}_1), \ldots, R_n(\bar{u}_n)\,.$$

An *assignment* for q over D is a function η from the set of variables in q to $\text{Dom}(D)$. We say that η is *consistent* with D if

$$\{R_1(\eta(\bar{u}_1)), \ldots, R_n(\eta(\bar{u}_n))\} \;\subseteq\; D\,,$$

where, for $i \in [n]$, the fact $R_i(\eta(\bar{u}_i))$ is obtained by replacing each variable x in $\bar{u}_i$ with $\eta(x)$, and leaving the constants in $\bar{u}_i$ untouched. The consistency of η with D essentially means that the body of q matches with D via η. Having this notion in place, we can define what is the output of a CQ on a database.

Definition 13.3: Evaluation of CQs

Given a database D of a schema $\mathbf{S}$, and a CQ $q(\bar{x})$ over $\mathbf{S}$, the *output* of q on D is defined as the set of tuples

$$q(D) \;=\; \{\eta(\bar{x}) \mid \eta \text{ is an assignment for } q \text{ over } D \text{ consistent with } D\}\,.$$

It is an easy exercise to show that the semantics of CQs inherited from the semantics of FO queries in Definition 3.6, and the semantics of CQs given in Definition 13.3, are equivalent, i.e., for a CQ $q = \varphi(\bar{x})$ and a database D,

$$\{\bar{a} \in \text{Dom}(D)^k \mid D \models \varphi(\bar{a})\} = \{\eta(\bar{x}) \mid \eta \text{ is an assignment for } q \text{ over } D \text{ consistent with } D\}\,.$$

Example 13.4: Evaluation of CQs

Let $\mathbf{S}$ be the schema from Example 3.2, which has been also used in Example 13.2. Let D be the database of $\mathbf{S}$ shown in Figure 3.1; we recall the relations Person and Profession in Figure 13.1. The following CQ q can be used to retrieve the ids and names of actors:

$$\text{Answer}(x, y) \;\text{ :– }\; \text{Person}(x, y, z), \text{Profession}(x, \text{`actor'}).$$

Observe that the assignment η for q over D such that

$$\eta(x) = \text{`1'} \qquad \eta(y) = \text{`Aretha'} \qquad \eta(z) = \text{`MPH'}$$

Person

pid	pname	cid
1	Aretha	MPH
2	Billie	BLT
3	Bob	DLT
4	Freddie	ST

Profession

pid	prname
1	singer
1	songwriter
1	actor
2	singer
3	singer
3	songwriter
3	author
4	singer
4	songwriter

Fig. 13.1: The relations Person and Profession for Example 13.4.

is consistent with D. Indeed, when applied to the body of q it produces the facts Person('1', 'Aretha', 'MPH') and Profession('1', 'actor'), both of which are facts of D. On the other hand, the assignment η' such that

$$\eta'(x) = \text{'2'} \qquad \eta'(y) = \text{'Billie'} \qquad \eta'(z) = \text{'BLT'}$$

is not consistent with D. When applied to the body of q, it generates the fact Profession('2', 'actor') that is not in D. It is straightforward to verify that η is the only assignment for q over D that is consistent with D, which in turn implies that the output of q on D is

$$q(D) \;=\; \{(\text{'1'}, \text{'Aretha'})\}.$$

If q is a Boolean CQ, then $q(D) = \texttt{true}$ if and only if there is an assignment for q over D that is consistent with D. In other words, $q(D) = \texttt{true}$ if and only if the body of the CQ matches with D via at least one assignment for q over D. For instance, if in Example 13.4 we consider also the Boolean CQ q'

$$\text{Answer} \;\text{ :– }\; \text{Person}(x, y, z), \text{Profession}(x, \text{'actor'}),$$

which is the Boolean version of q in Example 13.4, then $q'(D) = \texttt{true}$ since the assignment η is consistent with D. On the other hand, given the CQ q''

$$\text{Answer} \;\text{ :– }\; \text{Person}(x, y, z), \text{Profession}(x, \text{'nurse'}),$$

$q''(D) = \texttt{false}$ since there is no assignment η such that $\text{Person}(\eta(x), \eta(y), \eta(z))$ and $\text{Profession}(\eta(x), \text{'nurse'})$ are both facts of D.

Conjunctive Queries as a Fragment of FO

When CQs are seen as FO queries they use only relational atoms, conjunction ($\wedge$), and existential quantification ($\exists$). Thus, every CQ can be expressed using

formulae from the fragment of FO that corresponds to the closure of relational atoms under $\exists$ and $\wedge$; we refer to this fragment of FO as $\mathrm{FO}^{\mathrm{rel}}[\wedge, \exists]$. Actually, the converse is also true. Consider a query $\varphi(\bar{x})$ with φ being an $\mathrm{FO}^{\mathrm{rel}}[\wedge, \exists]$ formula. It is easy to show that $\varphi(\bar{x})$ is equivalent to a CQ. We first rename variables in order to ensure that bound variables do not repeat (which leads to an equivalent query), and then push the existential quantifiers outside. This conversion can be easily illustrated via a simple example.

Example 13.5: From $\mathrm{FO}^{\mathrm{rel}}[\wedge, \exists]$ Queries to CQs

Consider the $\mathrm{FO}^{\mathrm{rel}}[\wedge, \exists]$ query $\varphi(x)$ with

$$\varphi \;=\; (\exists y\; R(x, a, y)) \wedge (\exists y\; S(y, x, b)).$$

We first rename the second occurrence of y, and get the query $\varphi'(x)$ with

$$\varphi' \;=\; (\exists y\; R(x, a, y)) \wedge (\exists z\; S(z, x, b)).$$

We then push all the quantifiers outside, and get the CQ $\varphi''(x)$ with

$$\varphi'' \;=\; \exists y \exists z\; \big(R(x, a, y) \wedge S(z, x, b)\big).$$

From the above discussion, we immediately get that:

Theorem 13.6

The languages of CQs and of $\mathrm{FO}^{\mathrm{rel}}[\wedge, \exists]$ queries are equally expressive.

Notice that $\mathrm{FO}^{\mathrm{rel}}[\wedge, \exists]$ is *not* the same as $\mathrm{FO}[\wedge, \exists]$, that is, the fragment of FO that allows only for conjunction ($\wedge$) and existential quantification ($\exists$). Fragments defined by listing a set of features of FO are assumed to be the closure of *all* atomic formulae (including equational atoms) under those features. Therefore, the fragment $\mathrm{FO}[\wedge, \exists]$ allows also for equational atoms, which means that the query $\varphi(x, y)$ with $\varphi = (x = y)$ is an $\mathrm{FO}[\wedge, \exists]$ query. As we shall see in the next chapter, though, $\varphi(x, y)$ is not equivalent to a CQ.

Conjunctive Queries as a Fragment of RA

The class of CQs has the same expressive power as the fragment of RA that is built from base expressions $R \in \mathsf{Rel}$ and allows for selection, projection, and Cartesian product. Furthermore, conditions in selections are conjunctions of equalities. Note that base expressions of the form $\{a\}$ with $a \in \mathsf{Const}$ are not included. This fragment of RA is called the *select-project-join* (SPJ) fragment. Henceforth, we simply refer to the associated queries as SPJ queries. Recall

that the join operation is actually a selection from the Cartesian product on a condition that is a conjunction of equalities. We proceed to show that:

Theorem 13.7

The languages of CQs and of SPJ queries are equally expressive.

Proof. We first show how to translate a CQ q of the form

$$\text{Answer}(\bar{x}) \ :- \ R_1(\bar{u}_1), \ldots, R_n(\bar{u}_n)$$

into an SPJ query. In fact, q can be expressed as the query

$$\pi_\alpha\big(\sigma_{\theta(q)}\big(\sigma_{\theta(\bar{u}_1)}(R_1) \times \sigma_{\theta(\bar{u}_2)}(R_2) \times \cdots \times \sigma_{\theta(\bar{u}_n)}(R_n)\big)\big),$$

where conditions in selections, as well as the list of positions in the projections are defined as follows:

- For each $i \in [n]$, $\theta(\bar{u}_i)$ is a conjunction of statements $j \doteq a$ and $j \doteq k$, where $a \in \mathsf{Const}$ and $j, k \in [\text{ar}(R_i)]$, such that $j \doteq a$ is a conjunct of $\theta(\bar{u}_i)$ if and only if the j-th component of $\bar{u}_i$ is the constant a, and $j \doteq k$ is a conjunct of $\theta(\bar{u}_i)$ if and only if the j-th and the k-th components of $\bar{u}_i$ are the same variable. If no constant occurs in $\bar{u}_i$, and $\bar{u}_i$ consists of distinct variables, then the selection is omitted; we have R_i instead of $\sigma_{\theta(\bar{u}_i)}(R_i)$.
- The condition $\theta(q)$ is a conjunction of statements of the form $j \doteq k$, where $j, k \in [\text{ar}(R_1) + \cdots + \text{ar}(R_n)]$, such that $j \doteq k$ is a conjunct of $\theta(q)$ if and only if the following hold:
  - (i) if $j = \text{ar}(R_1) + \cdots + \text{ar}(R_\ell) + \ell'$, for some $\ell \in [0, n-1]$ and $\ell' \in [\text{ar}(R_{\ell+1})]$, then $k > \text{ar}(R_1) + \cdots + \text{ar}(R_{\ell+1})$, and
  - (ii) the j-th and the k-th components of $\bar{u}_1\bar{u}_2\ldots\bar{u}_n$ are the same variable.

  Item (i) states that j and k should be positions from different $\bar{u}_i$ tuples.
- Finally, α is a list of positions among $\bar{u}_1\bar{u}_2\ldots\bar{u}_n$ that form the output tuple of variables $\bar{x}$.

The correctness of the above translation is left as an exercise. Note that instead of using the condition $\theta(q)$, one can replace the Cartesian products by θ-joins (recall that the θ-join of relations R and S is defined as $R \bowtie_\theta S = \sigma_\theta(R \times S)$). Here is a simple example that illustrates the above translation.

Example 13.8: From CQs to SPJ Queries

Consider the CQ q defined as

$$\text{Answer}(x, x, y) \ :- \ R_1(\underbrace{x, z, z, a, x}_{\bar{u}_1}), R_2(\underbrace{a, y, z, a, b}_{\bar{u}_2}), R_3(\underbrace{x, y, z}_{\bar{u}_3}).$$

It is easy to verify that

$$\theta(\bar{u}_1) = (4 \doteq a) \wedge (1 \doteq 5) \wedge (2 \doteq 3)$$
$$\theta(\bar{u}_2) = (1 \doteq a) \wedge (4 \doteq a) \wedge (5 \doteq b)\,,$$

while the selection operation $\sigma_{\theta(\bar{u}_3)}$ is omitted since neither a constant nor a repetition of variables occurs in $\bar{u}_3$.

The condition $\theta(q)$ essentially has to specify that in

$$\bar{u}_1\bar{u}_2\bar{u}_3 \;=\; (x, z, z, a, x, a, y, z, a, b, x, y, z)$$

the variable x in $\bar{u}_1$ and the variable x in $\bar{u}_3$ are the same, the variable z in $\bar{u}_1$ and the variable z in both $\bar{u}_2$ and $\bar{u}_3$ are the same, and that the variable y in $\bar{u}_2$ and the variable y in $\bar{u}_3$ are the same. This results in

$$\theta(q) \;=\; (1 \doteq 11) \wedge (5 \doteq 11) \wedge (2 \doteq 8) \wedge (2 \doteq 13) \wedge (3 \doteq 8) \wedge (3 \doteq 13) \wedge (8 \doteq 13) \wedge (7 \doteq 12).$$

Finally, α corresponds to variable x repeated twice and variable y, i.e., $\alpha = (1, 1, 7)$. Summing up, the CQ q is expressed as

$$\pi_{(1,1,7)}\Big(\sigma_{(1\doteq 11)\wedge(2\doteq 8)\wedge(2\doteq 13)\wedge(7\doteq 12)}\big(\sigma_{(4\doteq a)\wedge(1\doteq 5)\wedge(2\doteq 3)}(R_1) \times \sigma_{(1\doteq a)\wedge(4\doteq a)\wedge(5\doteq b)}(R_2) \times R_3\big)\Big).$$

For the sake of readability, we have eliminated $(5 \doteq 11)$ from $\theta(q)$ since it can be derived from $(1 \doteq 11)$ in $\theta(q)$ and $(1 \doteq 5)$ in $\theta(\bar{u}_1)$, and likewise for conditions $(3 \doteq 8)$, $(3 \doteq 13)$ and $(8 \doteq 13)$ in $\theta(q)$.

We now proceed with the other direction, and show that every SPJ query e can be expressed as a CQ q_e. The proof is by induction on the structure of e. We can assume that in e all selections are either of the form $\sigma_{i\doteq a}$ or $\sigma_{i\doteq j}$ (because more complex selections can be obtained by applying a sequence of simple selections). We also assume that all projections are of the form $\pi_{\bar{i}}$ that exclude the i-th component; for instance, $\pi_{\bar{2}}(R)$ applied to a ternary relation R will transform each tuple (a, b, c) into (a, c) by excluding the second component (again, more complex projections are simply sequences of these simple ones).

- If $e = R$, where R is a k-ary relation, then q_e is the CQ $\varphi(\bar{x}) = R(\bar{x})$, where $\bar{x}$ is a k-ary tuple of pairwise distinct and fresh variables.
- If e is of arity k with $q_e = \varphi(x_1, \ldots, x_k)$, where the x_i's are not necessarily distinct, then
  - $q_{\sigma_{i\doteq a}(e)}$ is the CQ obtained from q_e by replacing each occurrence of the variable x_i by the constant a,
  - $q_{\sigma_{i\doteq j}(e)}$ is the CQ obtained from q_e by replacing each occurrence of the variable x_j with the variable x_i, and

  - $q_{\pi_{\bar{i}}(e)}$ is the CQ $\varphi(x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_k)$ if x_i occurs among the x_j's with $j \neq i$, and $\exists x_i \, \varphi(x_1, \ldots, x_k)$ otherwise.

- If e_1 is k-ary with $q_{e_1} = \varphi_1(x_1, \ldots, x_k)$ and $\varphi_1 = \exists \bar{z} \, \psi_1$, and e_2 is m-ary with $q_{e_2} = \varphi_2(y_1, \ldots, y_m)$ and $\varphi_2 = \exists \bar{w} \, \psi_2$, then $q_{(e_1 \times e_2)}$ is the CQ $\varphi(x_1, \ldots, x_k, y_1, \ldots, y_m)$ with $\varphi = \exists \bar{z} \exists \bar{w} \, \psi_1 \wedge \psi_2$; we assume that ψ_1 and ψ_2 do not share variables.

This completes the construction of the CQ q_e. The correctness of the above translation is left as an exercise to the reader.

We conclude by explaining further the difference between the two cases of handling projection. Consider the unary relations U, V and an RA expression $e = \pi_{\bar{1}}(\sigma_{1 \doteq 2}(U \times V))$. First, notice that $U \times V$ is translated as $\varphi(x, y) = U(x) \wedge V(y)$, since the expression U has to be translated as a relational atom of the form $U(z)$ where the variable z is fresh, and likewise for the expression V; thus, the occurrences of U and V in e have to be translated considering distinct variables, in this case x and y. Then $\sigma_{1 \doteq 2}(U \times V)$ is translated as $\varphi(x, x) = U(x) \wedge V(x)$, since y is replaced with x. Finally, $\pi_{\bar{1}}(\sigma_{1 \doteq 2}(U \times V))$ is obtained by eliminating the first occurrence of x as an output variable: the CQ defining e is $\psi(x) = U(x) \wedge V(x)$. On the other hand, the correct way to define $e' = \pi_{\bar{2}}(U \times V)$ as a CQ is to existentially quantify over y in $\varphi(x, y)$ that defines $U \times V$, that is, the CQ $\psi'(x)$ with $\psi = \exists y \, (U(x) \wedge V(y))$. □

The following is an immediate corollary of Theorems 13.6 and 13.7 that relates the languages of $\mathrm{FO}^{\mathrm{rel}}[\wedge, \exists]$ and SPJ queries.

Corollary 13.9

The language of $\mathrm{FO}^{\mathrm{rel}}[\wedge, \exists]$ queries and the language of SPJ queries are equally expressive.

14

Homomorphisms and Expressiveness

As already discussed in Chapter 9, homomorphisms are a fundamental tool that plays a key role in various aspects of relational databases. In this chapter, we discuss how homomorphisms emerge in the context of CQs. In particular, we show that they provide an alternative way to describe the evaluation of CQs, and also use them as a tool to understand the expressiveness of CQs.

CQ Evaluation and Homomorphisms

We can recast the semantics of CQs using the notion of homomorphism. The key observation is that the body of a CQ, written as a rule, can be viewed as a set of atoms. More precisely, given a CQ q of the form

$$\text{Answer}(\bar{x}) \text{ :- } R_1(\bar{u}_1), \ldots, R_n(\bar{u}_n)$$

we define the set of relational atoms

$$A_q \;=\; \{R_1(\bar{u}_1), \ldots, R_n(\bar{u}_n)\}.$$

Thus, we can naturally talk about homomorphisms from CQs to databases.

Definition 14.1: Homomorphisms from CQs to Databases

Consider a CQ $q(\bar{x})$ over a schema $\mathbf{S}$, and a database D of $\mathbf{S}$. We say that there is a *homomorphism from q to D*, written as $q \to D$, if $A_q \to D$. We also say that there is a *homomorphism from $(q, \bar{x})$ to $(D, \bar{a})$*, written as $(q, \bar{x}) \to (D, \bar{a})$, if $(A_q, \bar{x}) \to (D, \bar{a})$.

To define the output of a CQ $q(\bar{x})$ on a database D (see Definition 13.3), we used the notion of assignment for q over D, which is a function from the set of variables in q to $\text{Dom}(D)$. The output of q on D consists of all the tuples $\eta(\bar{x})$, where η is an assignment for q over D that is consistent with D, i.e.,

$$\{R_1(\eta(\bar{u}_1)), \ldots, R_n(\eta(\bar{u}_n))\} \subseteq D.$$

Since, for $i \in [n]$, $R_i(\eta(\bar{u}_i))$ is the fact obtained after replacing each variable x in $\bar{u}_i$ with $\eta(x)$, and leaving the constants in $\bar{u}_i$ untouched, such an assignment η corresponds to a function $h : \mathrm{Dom}(A_q) \to \mathrm{Dom}(D)$, which is the identity on the constants occurring in q, such that $R(h(\bar{u}_i)) = R(\eta(\bar{u}_i))$. But, of course, this is the same as saying that h is a homomorphism from q to D. Therefore, $q(D)$ is the set of all tuples $h(\bar{x})$, where h is a homomorphism from q to D, i.e., the set of all tuples $\bar{a}$ over $\mathrm{Dom}(D)$ with $(q, \bar{x}) \to (D, \bar{a})$. This leads to an alternative characterization of CQ evaluation in terms of homomorphisms.

Theorem 14.2

Given a database D of a schema $\mathbf{S}$, and a CQ $q(\bar{x})$ of arity $k \geq 0$ over $\mathbf{S}$,

$$q(D) = \{\bar{a} \in \mathrm{Dom}(D)^k \mid (q, \bar{x}) \to (D, \bar{a})\}.$$

Here is a simple example that illustrates the above characterization.

Example 14.3: CQ Evaluation via Homomorphisms

Let D and q be the database and the CQ, respectively, that have been considered in Example 13.4. We know that $q(D) = \{(\text{'1'}, \text{'Aretha'})\}$. By the characterization given in Theorem 14.2, we conclude that $\big(q, (x, y)\big) \to \big(D, (\text{'1'}, \text{'Aretha'})\big)$. To verify that this is the case, recall that we need to check whether $\big(A_q, (x, y)\big) \to \big(D, (\text{'1'}, \text{'Aretha'})\big)$, where

$$A_q = \{\mathrm{Person}(x, y, z), \mathrm{Profession}(x, \text{'actor'})\}.$$

Consider the function $h : \mathrm{Dom}(A_q) \to \mathrm{Dom}(D)$ such that

$$h(x) = \text{'1'} \qquad h(y) = \text{'Aretha'} \qquad h(z) = \text{'MPH'} \qquad h(\text{'actor'}) = \text{'actor'}.$$

It is clear that the following facts belong to D:

$$\mathrm{Person}(h(x), h(y), h(z)) = \mathrm{Person}(\text{'1'}, \text{'Aretha'}, \text{'MPH'})$$
$$\mathrm{Profession}(h(x), h(\text{'actor'})) = \mathrm{Profession}(\text{'1'}, \text{'actor'})$$

Moreover, $h\big((x, y)\big) = (\text{'1'}, \text{'Aretha'})$. Thus, h is a homomorphism from $\big(A_q, (x, y)\big)$ to $\big(D, (\text{'11'}, \text{'Aretha'})\big)$, witnessing that

$$\big(A_q, (x, y)\big) \to \big(D, (\text{'1'}, \text{'Aretha'})\big).$$

Preservation Results for CQs

Some particularly useful properties of CQs are their preservation under various operations, such as application of homomorphisms, or taking direct products. These properties will provide a precise explanation of the expressiveness of CQs as a subclass of FO queries.

Preservation Under Homomorphisms

By saying that a query q is preserved under homomorphisms, we essentially mean the following: if a tuple $\bar{a}$ belongs to the output of q on a database D, and $(D, \bar{a}) \to (D', \bar{b})$, then $\bar{b}$ should belong to the output of q on D'. Although we can naturally talk about homomorphisms among databases (since databases are sets of relational atoms), there is a caveat that is related to the fact that homomorphisms are the identity on constant values. Since $\mathrm{Dom}(D) \subseteq \mathsf{Const}$ for every database D, it follows that $D \to D'$ if and only if $D \subseteq D'$. Thus, the notion of homomorphism among databases is actually subset inclusion. However, the intention underlying the notion of homomorphism is to preserve the structure, possibly by leaving some constants unchanged.

To overcome this mismatch, we need a mechanism that allows us to convert a database into a set of relational atoms by replacing constant values with variables.[1] To this end, for a finite set of constants $C \subseteq \mathsf{Const}$, we define an injective function $\mathsf{V}_C : \mathsf{Const} \to \mathsf{Const} \cup \mathsf{Var}$ that is the identity on C. We then write $(D, \bar{a}) \to_C (D', \bar{b})$ if $(\mathsf{V}_C(D), \mathsf{V}_C(\bar{a})) \to (D', \bar{b})$. Note that in $\mathsf{V}_C(D)$ and $\mathsf{V}_C(\bar{a})$ all constants, except for those in C, have been replaced by variables, so the definition of homomorphism no longer trivializes to being a subset.

Example 14.4: Homomorphisms Among Databases

Consider the databases

$$D_1 = \{R(a,b), R(b,a)\} \qquad D_2 = \{R(c,c)\}.$$

If $C_1 = \emptyset$, then we have that $\mathsf{V}_{C_1}(a)$ and $\mathsf{V}_{C_1}(b)$ are distinct elements of Var, let say $\mathsf{V}_{C_1}(a) = x$ and $\mathsf{V}_{C_1}(b) = y$. Hence,

$$\mathsf{V}_{C_1}(D_1) = \{R(x,y), R(y,x)\} \qquad \mathsf{V}_{C_1}((a,b)) = (x,y),$$

from which we conclude that $(D_1, (a,b)) \to_{C_1} (D_2, (c,c))$ since

$$\big(\mathsf{V}_{C_1}(D_1), \mathsf{V}_{C_1}((a,b))\big) \to \big(D_2, (c,c)\big).$$

On the other hand, if $C_2 = \{a, b\}$, then

$$\mathsf{V}_{C_2}(D_1) = \{R(a,b), R(b,a)\} \qquad \mathsf{V}_{C_2}((a,b)) = (a,b).$$

[1] This is essentially the opposite of grounding a set of atoms discussed in Chapter 9.

Therefore, it does not hold that $(D_1, (a, b)) \to_{C_2} (D_2, (c, c))$, since it does not hold that $\big(\mathsf{V}_{C_2}(D_1), \mathsf{V}_{C_2}((a, b))\big) \to \big(D_2, (c, c)\big)$.

We can now define the notion of preservation under homomorphisms.

Definition 14.5: Preservation Under Homomorphisms

Consider a k-ary FO query $q = \varphi(\bar{x})$ over a schema $\mathbf{S}$. We say that q is *preserved under homomorphisms* if, for every two databases D and D' of $\mathbf{S}$, and tuples $\bar{a} \in \mathrm{Dom}(D)^k$ and $\bar{b} \in \mathrm{Dom}(D')^k$, it holds that

$$(D, \bar{a}) \to_{\mathrm{Dom}(\varphi)} (D', \bar{b}) \text{ and } \bar{a} \in q(D) \text{ implies } \bar{b} \in q(D').$$

We then show the following for CQs.

Proposition 14.6

Every CQ is preserved under homomorphisms.

Proof. Consider a k-ary CQ $q(\bar{x})$ over a schema $\mathbf{S}$, and let C be the set of constants in q. Assume that $(D, \bar{a}) \to_C (D', \bar{b})$ for some databases D, D' of $\mathbf{S}$, and tuples $\bar{a} \in \mathrm{Dom}(D)^k$ and $\bar{b} \in \mathrm{Dom}(D')^k$. Assume also that $\bar{a} \in q(D)$. Let h be a homomorphism witnessing $(\mathsf{V}_C(D), \mathsf{V}_C(\bar{a})) \to (D', \bar{b})$. By Theorem 14.2, $(q, \bar{x}) \to (D, \bar{a})$ via some h'. It holds that $h_q = \mathsf{V}_C \circ h'$ is a homomorphism witnessing $(q, \bar{x}) \to (\mathsf{V}_C(D), \mathsf{V}_C(\bar{a}))$ since h_q is the identity on C; indeed, for $a \in C$, $\mathsf{V}_C(h'(a)) = a$ by definition. Observe that $h \circ h_q$ is a homomorphism from $(q, \bar{x})$ to $(D', \bar{b})$, and thus, by Theorem 14.2, $\bar{b} \in q(D')$, as needed. □

Another key property is that of monotonicity. A query q over a schema $\mathbf{S}$ is *monotone* if, for every two databases D and D' of $\mathbf{S}$, we have that

$$D \subseteq D' \text{ implies } q(D) \subseteq q(D').$$

We show that homomorphism preservation implies monotonicity of CQs.

Corollary 14.7

Every CQ is monotone.

Proof. Let q be a CQ over $\mathbf{S}$, and C be the set of constants occurring in q. Consider the databases D, D' of $\mathbf{S}$ such that $D \subseteq D'$, and assume that $\bar{a} \in q(D)$. It is clear that V_C^{-1} is a homomorphism from $(\mathsf{V}_C(D), \mathsf{V}_C(\bar{a}))$ to $(D', \bar{a})$ and thus, $(D, \bar{a}) \to_C (D', \bar{a})$. By Proposition 14.6, we get that $\bar{a} \in q(D')$. □

Preservation under Direct Products

The second preservation result stated here concerns *direct products*. We first recall what a direct product of graphs is. Given two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, their direct product $G_1 \otimes G_2$ has $V_1 \times V_2$ as the set of vertices, i.e., each vertex is a pair (v_1, v_2) with $v_1 \in V_1$ and $v_2 \in V_2$. In $G_1 \otimes G_2$ there is an edge between (v_1, v_2) and (v_1', v_2') if there is an edge from v_1 to v_1' in E_1 and from v_2 to v_2' in E_2. Note that the notion of direct product is different from that of Cartesian product. Indeed, the Cartesian product of two binary relations is a 4-ary relation, while their direct product is still binary.

The definition of direct products for databases is essentially the same, modulo one small technical detail. Elements of databases come from Const. For two constants a_1 and a_2, the pair (a_1, a_2) is not an element of Const, but we can think of it as such. Indeed, since Const is countably infinite, there is a *pairing* function, i.e., a bijection $\tau : \mathsf{Const} \times \mathsf{Const} \to \mathsf{Const}$. A typical example, assuming that Const is enumerated as $c_0, c_1, c_2, \ldots$, is to define $\tau(c_n, c_m) = c_k$ for $k = (n+m)(n+m+1)/2 + m$. Given a pairing function, we can think of (a_1, a_2) as being in Const, represented by $\tau(a_1, a_2)$, and then simply extend the previous definition to arbitrary databases as follows. Given two databases D and D' of a schema $\mathbf{S}$, their direct product $D \otimes D'$ is a database of $\mathbf{S}$ that, for each n-ary relation name R in $\mathbf{S}$, contains the following facts:

$$R\big(\tau(a_1, a_1'), \ldots, \tau(a_n, a_n')\big) \text{ where } R(a_1, \ldots, a_n) \in D \text{ and } R(a_1', \ldots, a_n') \in D' .$$

Technically speaking, this definition depends on the choice of a pairing function, but this choice is irrelevant for FO queries (see Exercise 2.4).

We proceed to define the notion of preservation under direct products. We do this for Boolean queries without constants, as this suffices to understand the limitations of CQs. Exercises 2.6 and 2.7 explain how these results can be extended to queries with constants and free variables, respectively.

Definition 14.8: Preservation under Direct Products

A Boolean FO query q over a schema $\mathbf{S}$ is *preserved under direct products* if, for every two databases D and D' of $\mathbf{S}$, it holds that

$$D \models q \text{ and } D' \models q \text{ implies } D \otimes D' \models q.$$

We then show the following for CQs.

Proposition 14.9

Every Boolean CQ is preserved under direct products.

Proof. As stated earlier, for technical clarity, we only consider CQs that do not mention constants, but the result holds even for CQs with constants (see

Exercise 2.6). Let q be a Boolean CQ without constants over a schema $\mathbf{S}$, and let D, D' be databases of $\mathbf{S}$ such that $D \models q$ and $D' \models q$. By Theorem 14.2, there are homomorphisms h, g witnessing $q \to D$ and $q \to D'$, respectively. Define now $f(x) = \tau\big(h(x), g(x)\big)$. Assume that $R(u_1, \ldots, u_n)$ is an atom in q. Then $R(h(u_1), \ldots, h(u_n)) \in D$ and $R(g(u_1), \ldots, g(u_n)) \in D'$. Hence,

$$R\big(f(u_1), \ldots, f(u_n)\big) \;=\; R\big(\tau(h(u_1), g(u_1)), \ldots, \tau(h(u_n), g(u_n))\big)$$

belongs to $D \otimes D'$, proving that f is a homomorphism from q to $D \otimes D'$. Thus, by Theorem 14.2, $D \otimes D' \models q$, as needed. □

Expressiveness of CQs

The above preservation results allow us to delineate the expressiveness boundaries of CQs. By Theorem 13.7, CQs and SPJ queries, that is, RA queries that do *not* have inequality in selections, union (and disjunction in selection conditions), and difference, are equally expressive. We prove that none of these is expressible as a CQ. Also notice that in the definition of CQs we disallow explicit equality: CQs correspond to $\mathrm{FO}^{\mathrm{rel}}[\wedge, \exists]$ queries, i.e., FO queries based on the fragment of FO that is the closure of relational atoms under $\exists$ and $\wedge$. Implicit equality is, of course, allowed by reusing variables. We show that by adding explicit equality one obtains queries that cannot be expressed as CQs.

CQs cannot express inequality. This is because CQs with inequality are not preserved under homomorphisms.[2] Consider, e.g., the FO query

$$q_1 \;=\; \exists x \exists y \big(R(x, y) \wedge x \neq y\big).$$

For $D = \{R(a, b)\}$ and $D' = \{R(c, c)\}$, we have that $D \to_\emptyset D'$. However, $D \models q_1$ while $D' \not\models q_1$. As a second example, consider the FO query

$$q_2 \;=\; \exists x \, (S(x) \wedge x \neq a),$$

where a is a constant. Given $D = \{S(b)\}$ and $D' = \{S(a)\}$, we have that $D \to_{\{a\}} D'$. However, $D \models q_2$ while $D' \not\models q_2$.

CQs cannot express negative relational atoms. The reason is because such queries are not monotone. Consider, for example, the FO query

$$q \;=\; \neg P(a),$$

where a is a constant. If we take $D = \emptyset$ and $D' = \{P(a)\}$, then $D \subseteq D'$ but $D \models q$ while $D' \not\models q$.

[2] Conjunctive queries with inequality are studied in-depth in Chapter 32.

CQs cannot express difference. This is because difference is not monotone. Consider, for example, the FO query

$$q \;=\; \exists x(P(x) \wedge \neg Q(x)).$$

For $D = \{P(a)\} \subseteq D' = \{P(a), Q(a)\}$, we have that $D \models q$ while $D' \not\models q$.

CQs cannot express union. This is because such queries are not preserved under direct products. Consider, for example, the FO query

$$q \;=\; \exists x\,(R(x) \vee S(x)).$$

Let $D = \{R(a)\}$ and $D' = \{S(a)\}$. Then, $D \models q$ and $D' \models q$, but $D \otimes D'$ is empty, and thus, $D \otimes D' \not\models q$.

CQs cannot express explicit equality. This is because such queries are not preserved under direct products. Consider, for example, the FO query

$$q \;=\; \exists x \exists y\,(x = y).$$

Let $D = \{R(a)\}$ and $D' = \{S(a)\}$. Observe that $D \models q$ and $D' \models q$, but $D \otimes D' \not\models q$ since $D \otimes D'$ is empty.

15

Query Evaluation

In this chapter, we study the complexity of evaluating conjunctive queries, that is, CQ-Evaluation. Recall that this is the problem of checking whether $\bar{a} \in q(D)$ for a CQ query q, a database D, and a tuple $\bar{a}$ over $\mathrm{Dom}(D)$. Recall that for FO queries the same problem is PSPACE-complete (Theorem 7.1). As we show next, the complexity for CQs lies in NP.

Theorem 15.1

CQ-Evaluation is NP-complete.

Proof. We start with the upper bound. Consider a CQ $q(\bar{x})$, a database D, and a tuple $\bar{a} \in \mathrm{Dom}(D)$. By Theorem 14.2, $\bar{a} \in q(D)$ if and only if $(q, \bar{x}) \to (D, \bar{a})$. Therefore, we need to show that checking whether there exists a homomorphism from $(q, \bar{x})$ to $(D, \bar{a})$ is in NP. This is done by guessing a function $h : \mathrm{Dom}(A_q) \to \mathrm{Dom}(D)$, and then verifying that h is a homomorphism from $(A_q, \bar{x})$ to $(D, \bar{a})$, i.e., h is the identity on $\mathrm{Dom}(A_q) \cap \mathsf{Const}$, and $R(\bar{u}) \in A_q$ implies $R(h(\bar{u})) \in D$. Since both steps are feasible in polynomial time, we conclude that checking whether $(q, \bar{x}) \to (D, \bar{a})$ is in NP, as needed.

For the lower bound, we provide a reduction from a graph-theoretic problem, called Clique, which is NP-complete. Recall that a *clique* in an undirected graph $G = (V, E)$ is a complete subgraph $G' = (V', E')$ of G, i.e., every two distinct nodes of V' are connected via an edge of E'. We say that such a clique is of size $k \geq 1$ if V' consists of k nodes. The problem Clique follows:

Problem: Clique

Input: An undirected graph G, and an integer $k \geq 1$
Output: `true` if G has a clique of size k, and `false` otherwise

Consider an input to Clique given by $G = (V, E)$ and $k \geq 1$. The goal is to construct in polynomial time a database D and a Boolean CQ q such that G

has a clique of size k if and only if $D \models q$. We construct the database

$$D \;=\; \{\text{Node}(v) \mid v \in V\} \cup \{\text{Edge}(v,u) \mid (v,u) \in E \text{ and } v \neq u\},$$

which essentially stores the graph G, but without loops of the form (v,v) that may occur in E. We can eliminate loops, which is crucial for the correctness of the CQ that we construct next, since they do not affect the existence of a clique of size k in G, i.e., G has a clique of size k if and only if G' obtained from G after eliminating the loops has a clique of size k. We also construct

$$q \;=\; \exists x_1 \cdots \exists x_k \left(\bigwedge_{i=1}^{k} \text{Node}(x_i) \wedge \bigwedge_{i,j \in [k] \,:\, i \neq j} \text{Edge}(x_i, x_j) \right),$$

which asks whether G has a clique of size k. It is clear that D and q can be constructed in polynomial time from G and k. Moreover, it is easy to see that G has a clique of size k if and only if $D \models q$, and the claim follows. □

The data complexity of CQ-Evaluation is immediately inherited from FO-Evaluation (see Theorem 7.3) since CQs are FO queries. Recall that, by convention, CQ-Evaluation is in a complexity class $\mathcal{C}$ in data complexity if, for every CQ query q, the problem q-Evaluation, which takes as input a database D and a tuple $\bar{a}$ over $\text{Dom}(D)$, and asks whether $\bar{a} \in q(D)$, is in $\mathcal{C}$.

Corollary 15.2

CQ-Evaluation is in DLogSpace in data complexity.

Actually, as discussed in Chapter 7, FO-Evaluation, and thus CQ-Evaluation, is in AC_0 in data complexity, a class that is properly contained in DLogSpace. Recall that AC_0 consists of those languages that are accepted by polynomial-size circuits of constant depth and unbounded fan-in.

Parameterized Complexity

As discussed in Chapter 2, queries are typically much smaller than databases in practice. This motivated the notion of data complexity, where the cost of evaluation is measured only in terms of the size of the database, while the query is considered to be fixed. However, an algorithm that runs, for example, in time $O(\|D\|^{\|q\|})$, although is tractable in terms of data complexity since $\|q\|$ is a constant, it cannot be considered to be really practical when the database D is very large, even if the query q is small. This suggests that we need to rely on a finer notion of complexity than data complexity for classifying query evaluation algorithms as practical or impractical.

This finer notion of complexity is *parameterized complexity*, which is relevant whenever we need to classify the complexity of a problem depending on

some central parameters. In the context of query evaluation, it is sensible to consider the size of the database and the size of the query as separate parameters when designing evaluation algorithms, and target algorithms that take less time on the former parameter. For example, a query evaluation algorithm that runs in time $O(\|D\| \cdot \|q\|^2)$ is expected to perform better in practice than an algorithm that runs in time $O(\|D\|^2 \cdot \|q\|)$. Moreover, if the difference between $\|D\|$ and $\|q\|$ is significant, as it usually happens in real-life, then even an algorithm that runs in time $O(\|D\| \cdot 2^{\|q\|})$ could perform better in practice than an algorithm that runs in time $O(\|D\|^2 \cdot \|q\|)$.

Background on Parameterized Complexity

Before studying the parameterized complexity of CQ-Evaluation when considering the size of the database and the size of the query as separate parameters, we first need to introduce some fundamental notions of parameterized complexity. We start with the notion of parameterized problem (or language).

Definition 15.3: Parameterized Problem

Consider a finite alphabet Σ. A *parameterization* of Σ^* is a polynomial time computable function $\kappa : \Sigma^* \to \mathbb{N}$. A *parameterized problem (over Σ)* is a pair (L, κ), where $L \subseteq \Sigma^*$, and κ is a parameterization of Σ^*.

A typical example of such a problem is the parameterized version of Clique.

Example 15.4: Parameterized Clique

Recall that Clique is the set of pairs (G, k), where G is an undirected graph that contains a clique of size $k \geq 1$. Assume that graph-integer pairs are encoded as words over some finite alphabet Σ. Let $\kappa : \Sigma^* \to \mathbb{N}$ be the parameterization of Σ^* defined by

$$\kappa(w) = \begin{cases} k & \text{if } w \text{ is the encoding of a graph-integer pair } (G, k), \\ 1 & \text{otherwise,} \end{cases}$$

for $w \in \Sigma^*$. We denote the parameterized problem (Clique, κ) as p-Clique.

The input to a parameterized problem (L, κ) over the alphabet Σ is a word $w \in \Sigma^*$, and the numbers $\kappa(w)$ are the corresponding *parameters*. Similarly to (non-parameterized) problems that are represented in the form input-output, we will represent parameterized problems in the form input-parameter-output. For example, p-Clique is represented as follows:

Problem: p-Clique

Input: An undirected graph G, and an integer $k \geq 1$
Parameter: k
Output: `true` if G has a clique of size k, and `false` otherwise

Analogously, we can talk about the parameterized version of CQ-Evaluation, where the parameter is the size of the query:

Problem: p-CQ-Evaluation

Input: A CQ $q(\bar{x})$, a database D, and a tuple $\bar{a}$ over $\mathrm{Dom}(D)$
Parameter: $\|q\|$
Output: `true` if $\bar{a} \in q(D)$, and `false` otherwise

Recall that the motivation underlying parameterized complexity is to have a finer notion of complexity that allows us to classify algorithms as practical or impractical. But when an algorithm in the realm of parameterized complexity is considered to be practical? This brings us to *fixed-parameter tractability*.

Definition 15.5: Fixed-Parameter Tractability

Consider a finite alphabet Σ, and a parametarization $\kappa : \Sigma^* \to \mathbb{N}$ of Σ^*. An algorithm A with input alphabet Σ is an *fpt-algorithm with respect to* κ if there exists a computable function $f : \mathbb{N} \to \mathbb{R}_0^+$, and a polynomial $p(\cdot)$ such that, for every $w \in \Sigma^*$, A on input w runs in time

$$O\big(p(|w|) \cdot f(\kappa(w))\big).$$

A parameterized problem (L, κ) is *fixed-parameter tractable* if there is an fpt-algorithm with respect to κ that decides L. We write FPT for the class of all fixed-parameter tractable problems.

In simple words, (L, κ) is fixed-parameter tractable if there is an algorithm that decides whether $w \in L$ in time arbitrarily large in the parameter $\kappa(w)$, but polynomial in the size of the input w. This reflects the assumption that $\kappa(w)$ is much smaller than $|w|$, and thus, an algorithm that runs, e.g., in time $O(|w| \cdot 2^{\kappa(w)})$ is preferable than one that runs in time $O(|w|^{\kappa(w)})$.

Whenever we deal with an intractable problem, e.g., the problem of concern of this chapter, i.e., CQ-Evaluation, it would be ideal to be able to show that its parameterized version is in FPT. The reader may be tempted to think that p-CQ-Evaluation is in FPT, and that this can be easily shown by exploiting the algorithm for proving that CQ-Evaluation is in NP. It turns out that this is not true. Consider a CQ $q(\bar{x})$, a database D, and a tuple $\bar{a}$ over $\mathrm{Dom}(D)$.

To check if $\bar{a} \in q(D)$, we can iterate over all functions $h : \mathrm{Dom}(A_q) \to \mathrm{Dom}(D)$ until we find one that is a homomorphism from $(A_q, \bar{x})$ to $(D, \bar{a})$, in which case we return `true`; otherwise, we return `false`. Since there are $|\mathrm{Dom}(D)|^{|\mathrm{Dom}(A_q)|}$ such functions, we conclude that this algorithm runs in time

$$O\big(\|D\|^{\|q\|} \cdot r(\|D\| + \|q\|)\big)$$

for some polynomial $r(\cdot)$; note that the size of $\bar{a}$ is not included in the bound since it is polynomially bounded by $\|D\|$ and $\|q\|$. Therefore, we cannot conclude that p-CQ-Evaluation is in FPT since the expression that describes the running time of the above algorithm is not of the form $O(p(\|D\|) \cdot f(\|q\|))$, for some polynomial $p(\cdot)$ and computable function $f : \mathbb{N} \to \mathbb{R}_0^+$, as required by fixed-parameter tractability in Definition 15.5.

It is widely believed that there is no fpt-algorithm that decides the parameterized version of CQ-Evaluation. But then the natural question that comes up is the following: how can we prove that a parameterized problem is not in FPT? Several complexity classes have been defined in the context of parameterized complexity in order to prove that a parameterized problem is not in FPT. Such classes are widely believed to properly contain FPT. This means that if a parameterized problem is complete for one of those classes, then this is a strong indication that the problem in question is not in FPT. Notice here the analogy with classes such as NP and PSpace: it is not known whether these classes properly contain PTime, but if a problem is complete for any of them, then this is considered as a strong evidence that the problem is not tractable. We proceed to define one of such classes, namely W[1], which will allow us to pinpoint the exact complexity of p-CQ-Evaluation.

To define the class W[1], we need to introduce some auxiliary terminology. Consider a schema $\mathbf{S}$. Let X be a relation name of arity $m \geq 0$ that does not belong to $\mathbf{S}$, and φ an FO sentence over $\mathbf{S} \cup \{X\}$. For a database D of $\mathbf{S}$, and a relation $S \subseteq \mathrm{Dom}(D)^m$, we write $D \models \varphi(S)$ to indicate that $D' \models \varphi$, where $D' = D \cup \{X(\bar{a}) \mid \bar{a} \in S\}$. We further define the problem p-WD$_\varphi$ as follows:

Problem: p-WD$_\varphi$

Input:	A database D of the schema $\mathbf{S}$, and $k \in \mathbb{N}$
Parameter:	k
Output:	`true` if there exists $S \subseteq \mathrm{Dom}(D)^m$ such that $\lvert S\rvert = k$ and $D \models \varphi(S)$, and `false` otherwise

Notice that the sentence φ is fixed in the definition of p-WD$_\varphi$. Therefore, a different FO sentence ψ of the form described above gives rise to a different parameterized problem, dubbed p-WD$_\psi$. The last notion that we need before introducing the class W[1] is that of FPT-reduction.

An FPT-*reduction* from a parameterized problem (L_1, κ_1) over Σ_1 to a parameterized problem (L_2, κ_2) over Σ_2 is a function $\Phi : \Sigma_1^* \to \Sigma_2^*$ such that

the following holds: there are computable functions $f, g : \mathbb{N} \to \mathbb{R}_0^+$, and a polynomial $p(\cdot)$, such that, for every word $w \in \Sigma_1^*$:

1. $w \in L_1$ if and only if $\Phi(w) \in L_2$,
2. $\Phi(w)$ can be computed in time $p(|w|) \cdot f(\kappa_1(w))$, and
3. $\kappa_2(\Phi(w)) \leq g(\kappa_1(w))$.

The first and the second conditions are natural. The third condition is needed to ensure the crucial property that FPT is closed under FPT-reductions: if there exists an FPT-reduction from (L_1, κ_1) to (L_2, κ_2), and $(L_2, \kappa_2) \in \text{FPT}$, then $(L_1, \kappa_1) \in \text{FPT}$; the proof is left as an exercise.

We now have all the ingredients needed for introducing the class W[1]. Recall that *universal* FO sentences are FO sentences of the form $\forall x_1 \cdots \forall x_n \, \psi$, where ψ is quantifier free and $\text{FV}(\psi) = \{x_1, \ldots, x_n\}$.

Definition 15.6: The Class W[1]

A parameterized problem (L, κ) is in W[1] if there exists a schema $\mathbf{S}$, a relation name X not in $\mathbf{S}$, and a universal FO sentence φ over $\mathbf{S} \cup \{X\}$, such that there exists an FPT-reduction from (L, κ) to p-WD_φ.

To give some intuition about the definition of W[1], we show that p-Clique is in W[1]. We first define a universal FO sentence φ, and then show that there exists an FPT-reduction from p-Clique to p-WD_φ. Assume that $\mathbf{S}$ consists of the relation names Node[1] and Edge[2]. Let also Elem[1] be a relation name not in $\mathbf{S}$. We define the universal FO sentence φ over $\mathbf{S} \cup \{\text{Elem}\}$

$$\forall x \forall y \big((\text{Elem}(x) \wedge \text{Elem}(y) \wedge x \neq y) \to \text{Edge}(x, y) \big).$$

We proceed to show that there is an FPT-reduction from p-Clique to p-WD_φ. Consider an input to p-Clique given by $G = (V, E)$ and $k \geq 1$. Let

$$D \;=\; \{\text{Node}(v) \mid v \in V\} \cup \{\text{Edge}(v, u) \mid (v, u) \in E \text{ and } v \neq u\}.$$

The sentence φ checks whether the nodes in the relation Elem form a clique. Thus, G has a clique of size k if and only if there exists $S \subseteq \text{Dom}(D)$ such that $|S| = k$ and $D \models \varphi(S)$. It is also clear that (D, k) can be computed in polynomial time. Therefore, the above reduction from p-Clique to p-WD_φ is an FPT-reduction, which in turn implies that p-Clique $\in$ W[1].

Before we proceed with the parameterized complexity of CQ-Evaluation, let us comment on the nomenclature of W[1]. The class W[1] is the first level of a hierarchy of complexity classes W[t], for each $t \geq 1$; hence the number 1. More specifically, the class W[t] is defined in the same way as the class W[1], but allowing the FO sentence φ in p-WD_φ to be of the form $\forall \bar{x}_1 \exists \bar{x}_2 \cdots Q \bar{x}_t \, \psi$, where ψ is quantifier free, $Q = \exists$ if t is even, and $Q = \forall$ if t is odd. The W-hierarchy is defined as the union of all the classes W[t], that is, $\bigcup_{t \geq 1} \text{W}[t]$.

Parameterized Complexity of CQ-Evaluation

We know that p-Clique is W[1]-complete, which means that p-Clique $\in$ W[1] (this has been shown above), and every parameterized problem in W[1] can be reduced via an FPT-reduction to p-Clique. We also known that FPT $\subseteq$ W[1], and it is widely believed that this inclusion is strict (the status of the question whether FPT $\neq$ W[1] is comparable to that of PTime $\neq$ NP). Thus, it is unlikely that p-Clique $\in$ FPT (as FPT is closed under FPT-reductions). We use this result to prove that the same holds for p-CQ-Evaluation, thus providing strong evidence that this problem is not fixed-parameter tractable.

Theorem 15.7

p-CQ-Evaluation is W[1]-complete.

Proof. For the lower bound, we show that there exists an FPT-reduction from p-Clique to p-CQ-Evaluation. We use the same reduction as for the lower bound in Theorem 15.1, which we recall here for the sake of readability. Consider an input to p-Clique given by $G = (V, E)$ and $k \geq 1$. The database is

$$D \;=\; \{\mathrm{Node}(v) \mid v \in V\} \cup \{\mathrm{Edge}(v,u) \mid (v,u) \in E \text{ and } v \neq u\},$$

and the Boolean CQ is

$$q \;=\; \exists x_1 \cdots \exists x_k \left(\bigwedge_{i=1}^{k} \mathrm{Node}(x_i) \wedge \bigwedge_{i,j \in [k] \,:\, i \neq j} \mathrm{Edge}(x_i, x_j) \right).$$

As discussed in the proof of Theorem 15.1, G has a clique of size k if and only if $D \models q$, and D and q can be constructed in polynomial time from G and k. To conclude that this is an FPT-reduction, it remains to show that the third condition in the definition of FPT-reductions holds, i.e., $\|q\| \leq g(k)$ for some computable function $g : \mathbb{N} \to \mathbb{R}_0^+$. It is easy to verify that $\|q\| \leq c \cdot \log k \cdot k^2$ for some constant $c \in \mathbb{R}^+$, and thus, p-CQ-Evaluation is W[1]-hard.

We now focus on the upper bound. For technical clarity, we consider only constant-free Boolean CQs over a schema consisting of a single binary relation name Edge. We leave the prove for the general case, where no restrictions are imposed to the query and its schema, as an exercise.

We first define a universal FO sentence φ, and then show that there exists an FPT-reduction from p-CQ-Evaluation to p-WD$_\varphi$. Consider the schema

$$\mathbf{S} \;=\; \{\mathrm{Const}[1],\, \mathrm{Var}[1],\, \mathrm{Edge}_1[2],\, \mathrm{Edge}_2[2]\}.$$

Consider also the relation name Hom[2] that does not belong to $\mathbf{S}$. We define the universal FO sentence φ over $\mathbf{S} \cup \{\mathrm{Hom}\}$ as follows:

$$\forall x \forall y \forall z \left((\mathrm{Hom}(x,y) \wedge \mathrm{Hom}(x,z)) \rightarrow y = z \right) \wedge$$
$$\forall x \forall y \left(\mathrm{Hom}(x,y) \rightarrow \left(\mathrm{Var}(x) \wedge \mathrm{Const}(y) \right) \right) \wedge$$
$$\forall x_1 \forall y_1 \forall x_2 \forall y_2 \left((\mathrm{Edge}_1(x_1,y_1) \wedge \mathrm{Hom}(x_1,x_2) \wedge \mathrm{Hom}(y_1,y_2)) \rightarrow \mathrm{Edge}_2(x_2,y_2) \right).$$

We show that there is an FPT-reduction from p-CQ-Evaluation to $\mathsf{p\text{-}WD}_\varphi$. Consider an input to p-CQ-Evaluation given by a constant-free Boolean CQ q over the schema $\{\mathrm{Edge}[2]\}$, and a database D of $\{\mathrm{Edge}[2]\}$. Assuming that $\{x_1, \ldots, x_n\}$ are the variables occurring in q, we define the database D' as

$$\begin{aligned} D \;\cup\; \{\mathrm{Const}(a) \mid a \in \mathrm{Dom}(D)\} \;\cup\; \{\mathrm{Var}(a_{x_1}), \ldots, \mathrm{Var}(a_{x_n})\} \\ \cup\; \{\mathrm{Edge}_1(a_{x_i}, a_{x_j}) \mid \mathrm{Edge}(x_i, x_j) \text{ is an atom occuring in } q\} \\ \cup\; \{\mathrm{Edge}_2(a,b) \mid \mathrm{Edge}(a,b) \in D\}. \end{aligned}$$

Roughly, the relation Const stores the constants occurring in D, the relation Var stores the variables occurring in q, the relation Edge_1 stores the atoms of q, and the relation Edge_2 stores the facts of D. We further define $n = k$, that is, k is the number of variables occurring in q.

With the definitions of D' and k in place, we can now explain the meaning of the FO sentence φ. The first conjunct $\forall x \forall y \forall z \left((\mathrm{Hom}(x,y) \wedge \mathrm{Hom}(x,z)) \rightarrow y = z \right)$ states that Hom represents a function, as only one value can be associated to x. The second conjunct states that Hom maps variables of q to constants of D. Finally, the third conjunct states that Hom represents a homomorphism from q to D. Notice, however, that φ does not impose the restriction that every variable occurring q has to be mapped to a constant of D, as this requires a non-universal FO sentence of the form

$$\forall x \left(\mathrm{Var}(x) \rightarrow \exists y \left(\mathrm{Const}(y) \wedge \mathrm{Hom}(x,y) \right) \right).$$

Instead, the parameter $k = n$ is used to force Hom to map every variable in q to a constant of D, as n is the number of variables occurring in q.

Summing up, $q(D) = \texttt{true}$ if and only if there is $S \subseteq \mathrm{Dom}(D')^2$ with $|S| = k$ and $D' \models \varphi(S)$. It is also clear that D' and k can be constructed from D and q in polynomial time, and $k \leq \|q\|$. Thus, we have provided an FPT-reduction from p-CQ-Evaluation to $\mathsf{p\text{-}WD}_\varphi$, which shows that p-CQ-Evaluation $\in \mathrm{W}[1]$ (for constant-free Boolean CQs over a single binary relation). □

16

Containment and Equivalence

We have seen in Chapter 8 that the satisfiability problem for FO and RA is undecidable. In terms of query optimization, satisfiability is arguably the most elementary task one can think of, since it simply asks whether a query has a non-empty output on at least one database. Indeed, if a query is not satisfiable, then we do not even need to access the database in order to compute its output, which is trivially empty. Furthermore, for FO and RA, undecidability of other static analysis tasks such as containment and equivalence immediately follow from the undecidability of satisfiability.

On the other hand, the satisfiability problem for CQs is trivial. Indeed, given a CQ q, there is always a database on which q has a non-empty output, that is, the grounding $A_q^{\downarrow}$ of A_q (see Definition 9.3). This means that static analysis for CQs is drastically different than for FO and RA, which in turn indicates that we need to revisit the problems of containment and equivalence in the case of CQs. This is the goal of this chapter.

Optimizing A Simple Query

We start by first illustrating the role of containment and equivalence for CQs in query optimization by means of a simple example.

Example 16.1: A CQ with Redundancy

Consider again the relational schema

```
Person [ pid, pname, cid ]
Profession [ pid, prname ]
City [ cid, cname, country ]
```

from Chapter 3, and the CQ

$$q \;=\; \text{Answer}(y) \text{ :– } \text{Person}(x, y, z), \text{Profession}(x, \text{`actor'}), \text{Profession}(x, w)$$

over this schema. The query q asks for names of persons who are actors and who have some profession. It is clear that q contains some redundancy since, if a person is an actor, then this person also has a profession (namely, being an actor). In fact, the CQ

$$q' \;=\; \text{Answer}(y) \text{ :– } \text{Person}(x, y, z), \text{Profession}(x, \text{`actor'})$$

asks the same query, but in smarter way in the sense that it mentions fewer relational atoms in its body. We make two observations:

(a) The query q' is a part of q, that is, all atoms in the body of q' belong also to the body of q.

(b) In order to test if q and q' are equivalent, we only need to test if $q' \subseteq q$. The other inclusion immediately follows from (a).

The above example suggests that the following simple strategy may be useful for optimizing a CQ q. We write $(q - R(\bar{u}))$ for the CQ obtained by deleting from the body of q the relational atom $R(\bar{u})$.

Algorithm 3 OPTIMIZE-BY-CONTAINMENT(q)

Input: A CQ $q(\bar{x})$
Output: A CQ $q^*(\bar{x})$ that is equivalent to $q(\bar{x})$, and may mention fewer atoms

1: **while** there exists an atom $R(\bar{u})$ in the body of q such that $(q - R(\bar{u})) \subseteq q$ **do**
2: $\quad q := (q - R(\bar{u}))$
3: **return** $q(\bar{x})$

The approach in Algorithm 3 captures a very natural idea for optimizing CQs: *keep removing atoms from the body of the CQ as long as the resulting CQ is equivalent to the original one.* In order to carry out this strategy (and numerous other, more intricate, optimization strategies), it is crucial that we are able to effectively test containment, and thus equivalence, between CQs. We therefore study in this chapter the closely related problems CQ-Containment and CQ-Equivalence. We will retake Algorithm 3 in Chapter 17.

Containment

We first concentrate on CQ-Containment. We start by illustrating the notion of containment for CQs via a simple example.

Example 16.2: CQ Containment

Consider the CQ

$$q_1 \;=\; \text{Answer}(y_1) \text{ :– } \text{Person}(x_1, y_1, z_1), \text{Profession}(x_1, \text{'actor'}), \\ \text{City}(z_1, \text{'Los Angeles'}, \text{'United States'})$$

asking for names of actors who live in Los Angeles, and the CQ

$$q_2 \;=\; \text{Answer}(y_2) \text{ :– } \text{Person}(x_2, y_2, z_2), \text{Profession}(x_2, w_2)$$

asking for persons who have a profession. It is easy to verify that $q_1 \subseteq q_2$ since q_1 imposes the extra conditions that the returned persons are actors who live in Los Angeles.

We proceed to show that checking for containment in the case of CQs is decidable, but an intractable problem.

Theorem 16.3

CQ-Containment is NP-complete.

The proof of Theorem 16.3 relies on a useful characterization of containment of CQs in terms of homomorphisms, which we present below. Given two CQs $q(\bar{x})$ and $q'(\bar{x}')$, we write $(q', \bar{x}') \to (q, \bar{x})$ for the fact that there exists a homomorphism from $(A_{q'}, \bar{x}')$ to $(A_q, \bar{x})$; we also write $q \to q'$ to indicate that $A_q \to A_{q'}$. Recall that A_q and $A_{q'}$ are the sets of atoms occurring in the body of q and q', respectively, when seen as rules.

We also remind the reader that for a set of atoms S, we write $S^{\downarrow}$ for the grounding of S, which allows us to view S as a database. Such a grounding is given by the bijective homomorphism G_S from S to $S^{\downarrow}$ that replaces variables in S by new constants; in particular, $\mathsf{G}_S(S) = S^{\downarrow}$.

Theorem 16.4: Homomorphism Theorem

Let $q(\bar{x})$ and $q'(\bar{x}')$ be CQs. Then:

$$q \subseteq q' \quad \text{if and only if} \quad (q', \bar{x}') \to (q, \bar{x}).$$

Proof. ($\Rightarrow$) Assume that $q \subseteq q'$. Since G_{A_q} is a homomorphism, Theorem 14.2 implies $\mathsf{G}_{A_q}(\bar{x}) \in q(\mathsf{G}_{A_q}(A_q))$. Since $q \subseteq q'$, we have $\mathsf{G}_{A_q}(\bar{x}) \in q'(\mathsf{G}_{A_q}(A_q))$. Applying Theorem 14.2 again, we conclude that there exists a homomorphism h from $(A_{q'}, \bar{x}')$ to $(\mathsf{G}_{A_q}(A_q), \mathsf{G}_{A_q}(\bar{x}))$. Since G_{A_q} is bijective, $\mathsf{G}_{A_q}^{-1} \circ h$ is a homomorphism from $(A_{q'}, \bar{x}')$ to $(A_q, \bar{x})$, as needed.

($\Leftarrow$) Conversely, assume that $(q', \bar{x}') \to (q, \bar{x})$, and let h be a homomorphism from $(A_{q'}, \bar{x}')$ to $(A_q, \bar{x})$. Given a database D, assume that $\bar{a} \in q(D)$. By Theorem 14.2, there exists a homomorphism g from $(A_q, \bar{x})$ to $(D, \bar{a})$. Since homomorphisms compose, $g \circ h$ is a homomorphism from $(A_{q'}, \bar{x}')$ to $(D, \bar{a})$ and, thus, $\bar{a} \in q'(D)$ by Theorem 14.2. Therefore, we have that $q(D) \subseteq q'(D)$, from which we conclude that $q \subseteq q'$. □

The next example shows the usefulness of the Homomorphism Theorem.

Example 16.5: Homomorphism Theorem

Consider again the CQs q_1 and q_2 from Example 16.2, and recall that $q_1 \subseteq q_2$. This is confirmed by the Homomorphism Theorem since

$$(q_2, y_2) \;\to\; (q_1, y_1).$$

This is the case since the function $h : \mathrm{Dom}(A_{q_2}) \to \mathrm{Dom}(A_{q_1})$ defined as

$$h(x_2) = x_1 \qquad h(y_2) = y_1 \qquad h(z_2) = z_1 \qquad h(w_2) = \text{`actor'}$$

is a homomorphism from (A_{q_2}, y_2) to (A_{q_1}, y_1).

An easy consequence of the Homomorphism Theorem is that the problem CQ-Containment can be reduced to CQ-Evaluation.

Corollary 16.6

Let $q(\bar{x})$ and $q'(\bar{x}')$ be CQs. Then:

$$q \subseteq q' \;\text{ if and only if }\; \mathsf{G}_{A_q}(\bar{x}) \in q'(\mathsf{G}_{A_q}(A_q)).$$

Proof. By Theorem 16.4, we conclude that

$$q \subseteq q' \;\text{ if and only if }\; (A_{q'}, \bar{x}') \to (A_q, \bar{x}).$$

We can also show that

$$(A_{q'}, \bar{x}') \to (A_q, \bar{x}) \;\text{ if and only if }\; (A_{q'}, \bar{x}') \;\to\; (\mathsf{G}_{A_q}(A_q), \mathsf{G}_{A_q}(\bar{x})).$$

Indeed, if $(A_{q'}, \bar{x}') \to (A_q, \bar{x})$ is witnessed via h, then we have that $\mathsf{G}_{A_q} \circ h$ is a homomorphism from $(A_{q'}, \bar{x}')$ to $(\mathsf{G}_{A_q}(A_q), \mathsf{G}_{A_q}(\bar{x}))$. Conversely, assuming that $(A_{q'}, \bar{x}') \to (\mathsf{G}_{A_q}(A_q), \mathsf{G}_{A_q}(\bar{x}))$ is witnessed via g, $\mathsf{G}_{A_q}^{-1} \circ g$ is a homomorphism from $(A_{q'}, \bar{x}')$ to $(A_q, \bar{x})$. By Theorem 14.2, we get that

$$(A_{q'}, \bar{x}') \to (\mathsf{G}_{A_q}(A_q), \mathsf{G}_{A_q}(\bar{x})) \;\text{ if and only if }\; \mathsf{G}_{A_q}(\bar{x}) \in q'(\mathsf{G}_{A_q}(A_q)).$$

Consequently, we get that $q \subseteq q'$ if and only if $\mathsf{G}_{A_q}(\bar{x}) \in q'(\mathsf{G}_{A_q}(A_q))$. □

By exploiting the Homomorphism Theorem, we can further show that the problem CQ-Evaluation can be reduced to CQ-Containment, i.e., the opposite of what Corollary 16.6 shows. In the proof of Corollary 16.6, we essentially convert the CQ q into a database via the bijective homomorphism G_{A_q}. Now we are going to do the opposite, i.e., convert a database into a CQ. As discussed in Chapter 14, we can convert a database D into a set of relational atoms via the injective function $\mathsf{V}_C : \mathsf{Const} \to \mathsf{Const} \cup \mathsf{Var}$, where C is a finite set of constants. Recall that $\mathsf{V}_C(D)$ is the set of relational atoms obtained from D by replacing constants, except for those in C, with variables. The following corollary, which establishes that CQ-Evaluation can be reduced to CQ-Containment, is stated for Boolean CQs, as this suffices for the purpose of pinpointing the complexity of CQ-Containment, but it can be easily generalized to arbitrary CQs.

Corollary 16.7

Let q be a Boolean CQ, D a database, and q_D the Boolean CQ such that $A_{q_D} = \mathsf{V}_C(D)$, where $C = \mathrm{Dom}(A_q) \cap \mathsf{Const}$. Then:

$$D \models q \text{ if and only if } q_D \subseteq q.$$

Proof. By Theorem 14.2, we conclude that

$$D \models q \text{ if and only if } q \to D.$$

It is easy to show that

$$q \to D \text{ if and only if } q \to q_D.$$

Indeed, if $q \to D$ is witnessed via h, then we get that $\mathsf{V}_C \circ h$ is a homomorphism from q to q_D. Conversely, assuming that $q \to q_D$ is witnessed via g, $\mathsf{V}_C^{-1} \circ g$ is a homomorphism from q to D. By Theorem 16.4, we conclude that

$$q \to q_D \text{ if and only if } q_D \subseteq q.$$

From the above equivalences, we get that $D \models q$ if and only if $q_D \subseteq q$. □

By Theorem 15.1, CQ-Evaluation is in NP, and thus, Corollary 16.6 implies that also CQ-Containment is in NP. Moreover, since CQ-Evaluation is NP-hard even for Boolean CQs (this is because the CQ that the reduction from Clique to CQ-Evaluation builds in the proof of Theorem 15.1 is Boolean), Corollary 16.7 implies that CQ-Containment is NP-hard. Therefore, CQ-Containment is NP-complete, which establishes Theorem 16.3.

Equivalence

We now focus on the equivalence problem: given two CQs q, q', check whether $q \equiv q'$, i.e., whether $q(D) = q'(D)$ for every database D. We show that:

Theorem 16.8

CQ-Equivalence is NP-complete.

Proof. Concerning the upper bound, it suffices to observe that

$$q \equiv q' \text{ if and only if } q \subseteq q' \text{ and } q' \subseteq q,$$

which implies that CQ-Equivalence is in NP since, by Theorem 16.3, the problem of deciding whether $q \subseteq q'$ and $q' \subseteq q$ is in NP.

Concerning the lower bound, we provide a reduction from CQ-Containment. In fact, CQ-Containment is NP-hard even if we consider Boolean CQs (this is a consequence of the proof of Theorem 16.3). Consider two Boolean CQs

$$q = \text{Answer} :\!- R_1(\bar{u}_1), \ldots, R_n(\bar{u}_n) \qquad q' = \text{Answer} :\!- R'_1(\bar{u}'_1), \ldots, R'_m(\bar{u}'_m),$$

We assume that q, q' do not share variables since we can always rename variables without affecting the semantics of a query. Let $q_\cap$ be the Boolean CQ

$$\text{Answer} :\!- R_1(\bar{u}_1), \ldots, R_n(\bar{u}_n), R'_1(\bar{u}'_1), \ldots, R'_m(\bar{u}'_m),$$

which essentially computes the intersection of q and q'. In other words, for every database D, $q(D) \cap q'(D) = q_\cap(D)$. It is straightforward to see that

$$q \subseteq q' \text{ if and only if } q \equiv q_\cap,$$

which in turn implies that CQ-Equivalence is NP-hard, as needed. □

17

Minimization

Query optimization is the task of transforming a query into an equivalent one that is easier to evaluate. Since joins are expensive operations, we typically consider an equivalent version of a CQ q with fewer atoms in its body, and thus, with fewer joins to perform. Ideally, we would like to compute a CQ q' that is equivalent to q, and is also minimal, i.e., it has the minimum number of atoms. This brings us to the notion of minimization of CQs.

Definition 17.1: Minimization of CQs

Consider a CQ q over a schema $\mathbf{S}$. A CQ q' over $\mathbf{S}$ is a *minimization* of q if the following hold:

1. $q \equiv q'$, and
2. for every CQ q'' over $\mathbf{S}$, $q' \equiv q''$ implies $|A_{q'}| \leq |A_{q''}|$.

In other words, q' is a minimization of q if it is equivalent to q and has the smallest number of atoms among all the CQs that are equivalent to q. It is straightforward to see that every CQ q over a schema $\mathbf{S}$ has a minimization, which is actually a query from the finite set (up to variable renaming)

$$M_q \;=\; \{q' \mid q' \text{ is a CQ over } \mathbf{S} \text{ and } |A_{q'}| \leq |A_q|\}$$

that collects all the CQs over $\mathbf{S}$ (up to variable renaming) with at most $|A_q|$ atoms. Hence, to compute a minimization of q, we could, e.g., iterate over all CQs of M_q in increasing order with respect to the number of body atoms, until we find one that is equivalent to q. But now the following questions arise:

1. Is there a smarter procedure for computing a minimization of q instead of naively iterating over the exponentially many CQs of M_q? In particular, does the strategy of removing atoms from q as long as the resulting query is equivalent to q (see Algorithm 3) lead to a minimization of q?

2. Which minimization of q should be computed? Is there one that stands out as the best?

The above questions have neat answers, which we discuss in detail in the rest of the chapter. In a nutshell, one can indeed find minimizations of a CQ q by removing atoms from its body. Moreover, although q may have several minimizations, they are all the same (up to variable renaming). This implies that no matter in which order we remove atoms from the body of q, we will always compute the same minimization of q (up to variable renaming).

Minimization via Atom Removals

Consider a CQ q of the form $\text{Answer}(\bar{x}) \coloneq R_1(\bar{u}_1), \ldots, R_n(\bar{u}_n)$. The CQ q' obtained from q by removing the atom $R_i(\bar{u}_i)$, for some $i \in [n]$, is

$$\text{Answer}(\bar{x}') \ \coloneq \ R_1(\bar{u}_1), \ldots, R_{i-1}(\bar{u}_{i-1}), R_{i+1}(\bar{u}_{i+1}), \ldots, R_n(\bar{u}_n),$$

where $\bar{x}'$ is obtained from $\bar{x}$ by removing every variable that is only mentioned in the atom $R_i(\bar{u}_i)$. For example, if we remove the atom $R(x)$ from the CQ $\text{Answer}(x, y) \coloneq R(x), S(y)$, then we obtain the CQ $\text{Answer}(y) \coloneq S(y)$ as the variable x is only mentioned in $R(x)$. On the other hand, if we remove the atom $R(x)$ from the CQ $\text{Answer}(x, y) \coloneq R(x), T(x, y)$, then we obtain the CQ $\text{Answer}(x, y) \coloneq T(x, y)$ since x occurs also in $T(x, y)$.

The building block of minimization via atom removals is as follows: given a CQ $q(\bar{x})$, construct a CQ $q'(\bar{x})$ by removing an atom $R(\bar{u})$ from the body of q such that $(q, \bar{x}) \to (q', \bar{x})$. Notice that the output tuple $\bar{x}$ remains the same, which means that the atom $R(\bar{u})$ either it does not contain a variable of $\bar{x}$, or it contains only variables of $\bar{x}$ that occur also in atoms of $A_q - \{R(\bar{u})\}$. In this way, we actually construct a CQ that is equivalent to q. Indeed, since $(q, \bar{x}) \to (q', \bar{x})$, we get that $q' \subseteq q$ (by Theorem 16.4). Moreover, $(q', \bar{x}) \to (q, \bar{x})$ holds trivially due to the identity homomorphism from $A_{q'}$ to A_q, and thus, $q \subseteq q'$ (again by Theorem 16.4). We then iteratively remove atoms as above until we reach a CQ $q''(\bar{x})$ that is minimal, i.e., any CQ $q'''(\bar{x})$ that can be obtained by removing an atom from the body of q'' is such that $(q'', \bar{x}) \to (q''', \bar{x})$ does not hold. The CQ q'' is typically called a *core* of q. The formal definition follows.

Definition 17.2: Core of a CQ

Consider a CQ $q(\bar{x})$. A CQ $q'(\bar{x})$ is a *core* of q if the following hold:

1. $A_{q'} \subseteq A_q$,
2. $(q, \bar{x}) \to (q', \bar{x})$, and
3. for every CQ $q''(\bar{x})$ with $A_{q''} \subsetneq A_{q'}$, $(q', \bar{x}) \to (q'', \bar{x})$ does not hold.

The first condition in Definition 17.2 expresses that either $q = q'$, or q' is obtained by removing atoms from q but without altering the output tuple $\bar{x}$,

the second condition ensures that $q \equiv q'$, and the third condition states that q' is minimal. Here is an example that illustrates the notion of core of a CQ.

Example 17.3: Core of a CQ

Consider the Boolean CQ q_1 defined as

$$\text{Answer} \ :- \ R(x, y), R(x, z).$$

The function h defined as $h(x) = x$, $h(y) = y$ and $h(z) = y$ is a homomorphism from $\{R(x, y), R(x, z)\}$ to $\{R(x, y)\}$. Therefore, $q_1 \to q_1'$, where q_1' is the Boolean CQ defined as

$$\text{Answer} \ :- \ R(x, y).$$

Since, by definition, a CQ must have at least one atom in its body, we conclude that q_1' is a core of q_1. Observe that the Boolean CQ q_1''

$$\text{Answer} \ :- \ R(x, z)$$

is also a core of q_1 due to the homomorphism h' defined as $h(x) = x$, $h(y) = z$ and $h(z) = z$. Therefore, a CQ may have several cores that are syntactically different, depending on the order that atoms are removed.

Consider now the Boolean CQ q_2 defined as

$$\text{Answer} \ :- \ R(x, y), R(y, z).$$

Observe that there is neither a homomorphism from $\{R(x, y), R(y, z)\}$ to $\{R(x, y)\}$, nor a homomorphism from $\{R(x, y), R(y, z)\}$ to $\{R(y, z)\}$. This means that there is no way to remove an atom from q_2 and get an equivalent CQ. Therefore, we conclude that q_2 is its own core.

Finally, consider the CQ q_3 defined as

$$\text{Answer}(x, y, z) \ :- \ R(x, y), R(x, z),$$

which is actually q_1 with all the variables in the output tuple. By removing the atom $R(x, z)$ from q_3, we obtain the CQ q_3'

$$\text{Answer}(x, y) \ :- \ R(x, y).$$

In this case, there is no homomorphism from $(A_{q_3}, (x, y, z))$ to $(A_{q_3'}, (x, y))$ since there is no way to map the ternary tuple (x, y, z) to the binary tuple (x, y). Hence, q_3' is not equivalent to q_3. The case where we remove the atom $R(x, y)$ from q_3 is analogous. Therefore, q_3 is its own core.

We proceed to show that the notion of core captures our original intention, that is, the construction of a minimization of a CQ.

Proposition 17.4

Every CQ q has at least one core, and every core of q is a minimization of q.

Proof. We first show that a CQ $q(\bar{x})$ has a core. If q is a core of itself, then the claim follows. Assume now that this is not the case. This means that condition (3) in the definition of core (Definition 17.2) is violated, which in turn implies that there is a CQ $q'(\bar{x})$ with $A_{q'} \subsetneq A_q$ such that $(q, \bar{x}) \to (q', \bar{x})$. If q' is a core of itself, then it is clear that q' is a core of q. Otherwise, we iteratively apply the above argument until we reach a core of q.

We now proceed to show that a core of $q(\bar{x})$ is a minimization of it. We first show a useful technical lemma:

Lemma 17.5. *Consider a CQ $q_1(\bar{y}_1)$, and assume that there is a CQ $q_2(\bar{y}_2)$ such that $q_1 \equiv q_2$ and $|A_{q_2}| < |A_{q_1}|$. Then, there is a CQ $q_3(\bar{y}_1)$ such that*

$$(q_1, \bar{y}_1) \to (q_3, \bar{y}_1) \quad \text{and} \quad A_{q_3} \subsetneq A_{q_1}.$$

Proof. By Theorem 16.4, we conclude that

$$(q_1, \bar{y}_1) \to (q_2, \bar{y}_2) \quad \text{and} \quad (q_2, \bar{y}_2) \to (q_1, \bar{y}_1).$$

Assume that these statements are witnessed via the homomorphisms h_1 and h_2, respectively. Let $q_3(\bar{y}_3)$ be the CQ such that

$$A_{q_3} = h_2(A_{q_2}) \quad \text{and} \quad \bar{y}_3 = h_2(\bar{y}_2).$$

It is clear that $\bar{y}_3 = \bar{y}_1$ and $A_{q_3} \subseteq A_{q_1}$. Furthermore, since $|A_{q_3}| \leq |A_{q_2}|$ and $|A_{q_2}| < |A_{q_1}|$, we conclude that $|A_{q_3}| < |A_{q_1}|$, and thus, $A_{q_3} \subsetneq A_{q_1}$. It remains to show that $(q_1, \bar{y}_1) \to (q_3, \bar{y}_1)$. Since homomorphisms compose, the latter is witnessed via the homomorphism $h_2 \circ h_1$. □

Consider now a CQ $q'(\bar{x})$ that is a core of $q(\bar{x})$. Towards a contradiction, assume that q' is not a minimization of q. This implies that there exists a CQ q'' such that $q' \equiv q''$ and $|A_{q''}| < |A_{q'}|$. By Lemma 17.5, we conclude that there exists a CQ $q'''(\bar{x})$ such that $(q', \bar{x}) \to (q''', \bar{x})$ and $A_{q'''} \subsetneq A_{q'}$. This contradicts our hypothesis that q' is a core of q, and the claim follows. □

By Proposition 17.4, to compute a minimization of a CQ q, we simply need to compute a core of it. This can be done via the simple iterative procedure ComputeCore, given in Algorithm 4. Notice that this algorithm is a more detailed reformulation of Algorithm 3. It is straightforward to show that, for a CQ q, ComputeCore(q) terminates after finitely many steps. It is also not difficult to show that the procedure ComputeCore is correct.

Lemma 17.6. *Given a CQ q,* ComputeCore(q) *is a core of q.*

Algorithm 4 ComputeCore(q)

Input: A CQ $q(\bar{x})$
Output: A CQ $q^*(\bar{x})$ that is a core of $q(\bar{x})$

1: $S := A_q$
2: **while** there exists $R(\bar{u}) \in S$ such that each variable in $\bar{x}$
3: occurs in $\mathrm{Dom}(S - \{R(\bar{u})\})$ and $(S, \bar{x}) \to (S - \{R(\bar{u})\}, \bar{x})$ **do**
4: $S := S - \{R(\bar{u})\}$
5: **return** $q^*(\bar{x}) \coloneq R_1(\bar{u}_1), \ldots, R_n(\bar{u}_n)$, where $S = \{R_1(\bar{u}_1), \ldots, R_n(\bar{u}_n)\}$

Proof. At each iteration of the while-loop, the CQ $q'(\bar{x})$ with $A_{q'} = S$ (which is indeed a CQ since, by construction, every variable in $\bar{x}$ occurs in $A_{q'}$) is such that $A_{q'} \subseteq A_q$ and $(q, \bar{x}) \to (q', \bar{x})$. Therefore, the CQ $q^*(\bar{x})$ returned by the algorithm is such that $A_{q^*} \subseteq A_q$ and $(q, \bar{x}) \to (q^*, \bar{x})$. Furthermore, by construction, for every CQ $q''(\bar{x})$ with $A_{q''} \subsetneq A_{q^*}$, $(q^*, \bar{x}) \to (q'', \bar{x})$ does not hold. Therefore, q^* satisfies all the three conditions given in the definition of core (Definition 17.2), and thus, it is a core of q, as needed. □

Note that ComputeCore is a nondeterministic algorithm. Observe that there may be several atoms $R(\bar{y}) \in S$ satisfying the condition of the while loop (in particular, the condition $(S, \bar{x}) \to (S - \{R(\bar{y})\}, \bar{x})$), but we do not specify how such an atom is selected. In fact, the atom $R(\bar{y})$ of S that is eventually removed from S at step 4 is chosen nondeterministically. Therefore, the final result computed by the algorithm depends on how the atoms to be removed from S are chosen, and thus, different executions of ComputeCore(q) may compute cores of q that are syntactically different. This fact should not be surprising as it has been already illustrated in Example 17.3 (see the queries q_1' and q_1'' that are cores of q_1). This leads to the second main question raised above: is there a core of q that stands out as the best?

Uniqueness of Minimizations

It turns out that such a concept as the best core does not exist since a CQ has a *unique core* (up to variable renaming). This is a consequence of the fact that every CQ has a *unique minimization* (up to variable renaming). We proceed to show the latter statement.

We say that two CQs $q(\bar{x}), q'(\bar{x}')$ are *isomorphic* if one can be turned into the other via renaming of variables, i.e., if there is a bijection $\rho : \mathrm{Dom}(A_q) \to \mathrm{Dom}(A_{q'})$ that is a homomorphism from $(A_q, \bar{x})$ to $(A_{q'}, \bar{x}')$, and its inverse ρ^{-1} is a homomorphism from $(A_{q'}, \bar{x}')$ to $(A_q, \bar{x})$. (Recall from Chapter 9 that homomorphisms between sets of atoms are always the identity on constants.)

Proposition 17.7

Consider a CQ $q(\bar{x})$, and let $q'(\bar{x}')$ and $q''(\bar{x}'')$ be minimizations of q. Then q' and q'' are isomorphic.

Proof. We need to show that there is a bijection $\rho : \mathrm{Dom}(A_{q'}) \to \mathrm{Dom}(A_{q''})$ that is a homomorphism from $(A_{q'}, \bar{x}')$ to $(A_{q''}, \bar{x}'')$, and its inverse ρ^{-1} is a homomorphism from $(A_{q''}, \bar{x}'')$ to $(A_{q'}, \bar{x}')$. Since both q' and q'' are minimizations of q, we get that $q \equiv q'$ and $q \equiv q''$, and thus, $q' \equiv q''$. By Theorem 16.4,

$$(q', \bar{x}') \to (q'', \bar{x}'') \qquad \text{and} \qquad (q'', \bar{x}'') \to (q', \bar{x}').$$

Assume that these statements are witnessed via the homomorphisms h and g, respectively. We proceed to show a useful statement concerning h and g:

Lemma 17.8. *The functions h and g are bijections.*

Proof. We concentrate on h, and show that is both surjective and injective; the proof for g is analogous. We give a proof by contradiction:

- Assume first that h is not surjective. This implies that there is a variable $z \in \mathrm{Dom}(A_{q''})$ such that there is no variable $y \in \mathrm{Dom}(A_{q'})$ with $h(y) = z$. Let $R(\bar{u}) \in A_{q''}$ be an atom that mentions z. We have that $R(\bar{u}) \notin h(A_{q'})$. We define $q'''(\bar{x}'')$ as the CQ with $A_{q'''} = h(A_{q'})$. It is clear that $(q', \bar{x}') \to (q''', \bar{x}'')$ via h, and $(q''', \bar{x}'') \to (q', \bar{x}')$ via g. Therefore, by Theorem 16.4, $q' \equiv q'''$. Since $q' \equiv q''$, we conclude that $q'' \equiv q'''$. Observe also that $A_{q'''} \subsetneq A_{q''}$, which implies that $|A_{q'''}| < |A_{q''}|$. But this contradicts the fact that q'' is a minimization of q, and thus, h is surjective.
- Assume now that h is not injective. This implies that there are two distinct variables $y, z \in \mathrm{Dom}(A_{q'})$ such that $h(y) = h(z)$. Hence, $g(h(y)) = g(h(z))$, which implies that $g \circ h$ is a homomorphism from $(q', \bar{x}') \to (q', \bar{x}')$ that is not surjective. Therefore, there exists a variable $u \in \mathrm{Dom}(A_{q'})$ such that there is no variable $v \in \mathrm{Dom}(A_{q'})$ with $g(h(v)) = u$. Let $R(\bar{u}) \in A_{q'}$ be an atom that mentions u. We have that $R(\bar{u}) \notin g(h(A_{q'}))$. We define $q'''(\bar{x}')$ as the CQ with $A_{q'''} = g(h(A_{q'}))$. It is clear that $(q', \bar{x}') \to (q''', \bar{x}')$ via $g \circ h$. Observe also that $A_{q'''} \subsetneq A_{q'}$. Hence, $(q''', \bar{x}') \to (q', \bar{x}')$ via the identity homomorphism, which means that $q' \equiv q'''$ due to Theorem 16.4, and $|A_{q'''}| < |A_{q'}|$. But this contradicts the fact that q' is a minimization of q, which in turn implies that h is injective.

Since h is both surjective and injective, the claim follows. □

We are now ready to define the bijection $\rho : \mathrm{Dom}(A_{q'}) \to \mathrm{Dom}(A_{q''})$. Let $f = g \circ h$. It is clear that f is a homomorphism from $(A_{q'}, \bar{x}')$ to $(A_{q'}, \bar{x}')$. Since, by Lemma 17.8, both h and g are bijections, we can further conclude that f is a bijection. This implies that there exists $k \geq 0$ such that the function

$$f^k \;=\; \underbrace{f \circ \cdots \circ f}_{k}$$

is the identity homomorphism from $(A_{q'}, \bar{x}')$ to $(A_{q'}, \bar{x}')$. Let $\rho = h \circ f^{k-1}$. Since both h and f^{k-1} are bijections, we get that also ρ is a bijection. It is also clear that ρ is a homomorphism from $(A_{q'}, \bar{x}')$ to $(A_{q''}, \bar{x}'')$. Notice also that $g \circ \rho = f^k$ is the identity, which means that g is the inverse of ρ. Thus, the inverse of ρ is a homomorphism from $(A_{q''}, \bar{x}'')$ to $(A_{q'}, \bar{x}')$. Therefore, ρ witnesses the fact that q' and q'' are isomorphic, and the claim follows. □

From Proposition 17.4, which tells us that a core of a CQ q is a minimization of q, and Proposition 17.7, we immediately get the following corollary:

Corollary 17.9

Consider a CQ q, and let q' and q'' be cores of q. It holds that q' and q'' are isomorphic.

Recall that different executions of the nondeterministic procedure ComputeCore on some input CQ q, may compute cores of q that are syntactically different. However, Corollary 17.9 tells us that those cores differ only on the names of their variables. In other words, cores of q computed by different executions of ComputeCore(q) are actually the same up to variable renaming.

18

Containment Under Integrity Constraints

As discussed in Chapters 10 and 11, relational systems support the specification of semantic properties that should be satisfied by all databases of a certain schema. This is achieved via integrity constraints, also called dependencies. The question that arises is how static analysis, and in particular the notion of containment of CQs, studied in Chapter 17, is affected in the presence of constraints. In this chapter, we study this question concentrating on functional dependencies (FDs) and inclusion dependencies (INDs).

Functional Dependencies

We start with FDs, and illustrate via an example how containment of CQs is affected if we focus on databases that satisfy a given set of FDs.

Example 18.1: Containment of CQs Under FDs

Consider the CQs q_1 and q_2 defined as

$$\begin{aligned} \text{Answer}(x_1, y_1) &\text{ :- } R(x_1, y_1), R(y_1, z_1), R(x_1, z_1) \\ \text{Answer}(x_2, y_2) &\text{ :- } R(x_2, y_2), R(y_2, y_2), \end{aligned}$$

respectively. It is easy to verify that $(q_2, (x_2, y_2)) \to (q_1, (x_1, y_1))$ does not hold, and thus, we have that $q_1 \not\subseteq q_2$ by the Homomorphism Theorem. For example, if we consider the database

$$D \;=\; \{R(1,2), R(2,3), R(1,3)\},$$

then $q_1(D) = \{(1,2)\}$ and $q_2(D) = \emptyset$, so that $q_1(D) \not\subseteq q_2(D)$. Suppose now that q_1, q_2 will be evaluated only over databases that satisfy the FD

$$\sigma \;=\; R : \{1\} \to \{2\}.$$

In particular, q_1 and q_2 will not be evaluated over the database D since it does not satisfy σ. We can show that, for every database D',

$$D' \models \sigma \text{ implies } q_1(D') \subseteq q_2(D').$$

To see this, consider an arbitrary database D' that satisfies σ, and assume that $(a, b) \in q_1(D')$. By Theorem 14.2, we have that

$$(q_1, (x_1, y_1)) \rightarrow (D', (a, b))$$

via a homomorphism h_1. Since $D' \models \sigma$ and

$$\{R(h_1(x_1), h_1(y_1)), R(h_1(x_1), h_1(z_1))\} \subseteq D',$$

it holds that $h_1(y_1) = h_1(z_1)$. Since $R(h_1(y_1), h_1(z_1)) \in D'$, we get that

$$(q_2, (x_2, y_2)) \rightarrow (D', (a, b))$$

via h_2 such that $h_2(x_2) = h_1(x_1)$ and $h_2(y_2) = h_1(y_1) = h_1(z_1)$.

Our goal is to revisit the problem of containment for CQs in the presence of FDs. More precisely, given two CQs q and q', and a set Σ of FDs, we say that q *is contained in* q' *under* Σ, denoted by $q \subseteq_\Sigma q'$, if for every database D that satisfies Σ, it holds that $q(D) \subseteq q'(D)$. The problem of interest follows:

Problem: CQ-Containment-FD

Input: Two CQs q and q', and a set Σ of FDs
Output: `true` if $q \subseteq_\Sigma q'$, and `false` otherwise

We proceed to show the following result:

Theorem 18.2

CQ-Containment-FD is NP-complete.

It is clear that the NP-hardness is inherited from CQ containment without constraints (see Theorem 16.3). Recall that, by the Homomorphism Theorem, checking whether a CQ $q(\bar{x})$ is contained in a CQ $q'(\bar{x}')$ in the absence of constraints boils down to checking whether $(q', \bar{x}') \rightarrow (q, \bar{x})$. Even though this is not enough in the presence of FDs, we can adopt a similar approach providing that we first transform, by identifying terms as dictated by the FDs, the set of atoms A_q in q into a new set of atoms S that satisfies the FDs, and the tuple of variables $\bar{x}$ into a new tuple $\bar{u}$, which may contain also constants, and then check whether $(A_{q'}, \bar{x}') \rightarrow (S, \bar{u})$. This simple idea has

been already illustrated by Example 18.1. Unsurprisingly, the transformation of A_q and $\bar{x}$ into S and $\bar{u}$, respectively, can be done by exploiting the chase for FDs, which has been introduced in Chapter 10. For brevity, we simply write $\text{Chase}(q, \Sigma)$ instead of $\text{Chase}(A_q, \Sigma)$, and $h_{q,\Sigma}$ instead of $h_{A_q,\Sigma}$. We now show the following result by providing a proof similar to that of the Homomorphism Theorem:

Theorem 18.3

Let $q(\bar{x})$ and $q'(\bar{x}')$ be CQs over a schema $\mathbf{S}$, and Σ a set of FDs over $\mathbf{S}$. The following are equivalent:

1. $q \subseteq_\Sigma q'$.
2. $\text{Chase}(q, \Sigma) \neq \bot$ implies $(A_{q'}, \bar{x}') \to (\text{Chase}(q, \Sigma), h_{q,\Sigma}(\bar{x}))$.

Proof. For brevity, let $S = \text{Chase}(q, \Sigma)$ and $\bar{u} = h_{q,\Sigma}(\bar{x})$.

We first show that (1) implies (2). By hypothesis, $q \subseteq_\Sigma q'$. It is clear that, if $S \neq \bot$, then $\mathsf{G}_S(\bar{u}) \in q(\mathsf{G}_S(S))$. Since, by Lemma 10.8, $S \models \Sigma$, which means that $\mathsf{G}_S(S) \models \Sigma$, we have that $\mathsf{G}_S(\bar{u}) \in q'(\mathsf{G}_S(S))$. By Theorem 14.2, there exists a homomorphism h from $(A_{q'}, \bar{x}')$ to $(\mathsf{G}_S(S), \mathsf{G}_S(\bar{u}))$. Clearly, $\mathsf{G}_S^{-1} \circ h$ is a homomorphism from $(A_{q'}, \bar{x}')$ to $(S, \bar{u})$, as needed.

For showing that (2) implies (1) we proceed by case analysis:

- Assume first that $S = \bot$. This implies that, for every database D of $\mathbf{S}$ such that $D \models \Sigma$, there is no homomorphism from q to D; otherwise, there is a successful finite chase sequence of q under Σ, which contradicts the fact that $S = \bot$. Therefore, for every database D of $\mathbf{S}$ such that $D \models \Sigma$, $q(D) = \emptyset$, which in turn implies that $q \subseteq_\Sigma q'$.
- Assume now that $S \neq \bot$. By hypothesis, we get that $(A_{q'}, \bar{x}') \to (S, \bar{u})$ via a homomorphism h. Let D be an arbitrary database of $\mathbf{S}$ such that $D \models \Sigma$, and assume that $\bar{a} \in q(D)$. By Theorem 14.2, $(q, \bar{x}) \to (D, \bar{a})$. Since $D \models \Sigma$, Lemma 10.11 implies that $(S, \bar{u}) \to (D, \bar{a})$ via a homomorphism g. Since homomorphisms compose, $g \circ h$ is a homomorphism from $(q', \bar{x}')$ to $(D, \bar{a})$. By Theorem 14.2, $\bar{a} \in q'(D)$, which implies that $q \subseteq_\Sigma q'$.

Since in both cases we get that $q \subseteq_\Sigma q'$, the claim follows. □

The following is an easy consequence of Theorem 18.3 and Theorem 14.2.

Corollary 18.4

Let $q(\bar{x})$ and $q'(\bar{x}')$ be CQs over a schema $\mathbf{S}$, and Σ a set of FDs over $\mathbf{S}$. With $S = \text{Chase}(q, \Sigma)$, the following are equivalent:

1. $q \subseteq_\Sigma q'$.
2. $S \neq \bot$ implies $\mathsf{G}_S(h_{q,\Sigma}(\bar{x})) \in q'(\mathsf{G}_S(S))$.

By Lemma 10.10, $\text{Chase}(q, \Sigma)$ can be computed in polynomial time. Moreover, if $\text{Chase}(q, \Sigma) \neq \bot$, then the chase homomorphism $h_{q,\Sigma}$ can be also computed in polynomial time. Since CQ-Evaluation is in NP (see Theorem 15.1), we conclude that CQ-Containment-FD is also in NP, and Theorem 18.2 follows.

Inclusion Dependencies

We now focus on INDs. We first illustrate via an example how containment of CQs is affected if we focus on databases that satisfy a set of INDs.

Example 18.5: Containment of CQs Under INDs

Consider the CQs q_1 and q_2 defined as

$$\begin{aligned} \text{Answer}(x_1, y_1) &:- R(x_1, y_1), R(y_1, z_1), P(z_1, y_1) \\ \text{Answer}(x_2, y_2) &:- R(x_2, y_2), R(y_2, z_2), S(x_2, y_2, z_2), \end{aligned}$$

respectively. It is clear that $(q_2, (x_2, y_2)) \to (q_1, (x_1, y_1))$ does not hold, and thus, we have that $q_1 \not\subseteq q_2$ by the Homomorphism Theorem. Suppose now that q_1 and q_2 will be evaluated only over databases that satisfy

$$\sigma_1 = R[1,2] \subseteq S[1,2] \qquad \text{and} \qquad \sigma_2 = S[2,3] \subseteq R[1,2].$$

We can show that, for every database D,

$$D \models \{\sigma_1, \sigma_2\} \text{ implies } q_1(D) \subseteq q_2(D).$$

Consider an arbitrary database D that satisfies $\{\sigma_1, \sigma_2\}$, and assume that $(a, b) \in q_1(D)$, or, equivalently, $(q_1, (x_1, y_1)) \to (D, (a, b))$ via a homomorphism h_1. This implies that $R(h_1(x_1), h_1(y_1)) \in D$. Since $D \models \sigma_1$, we get that D contains an atom of the form $S(h_1(x_1), h(y_1), c)$. But since $D \models \sigma_2$, we also get that D contains the atom $R(h_1(y_1), c)$. Hence,

$$\{R(h_1(x_1), h_1(y_1)), R(h_1(y_1), c), S(h_1(x_1), h_1(y_1), c)\} \subseteq D.$$

This implies that $(q_1', (x_1, y_1)) \to (D, (a, b))$, where q_1' is obtained from q_1 by adding certain atoms according to σ_1 and σ_2, i.e., q_1' is defined as

$$\text{Answer}(x_1, y_1) :- R(x_1, y_1), R(y_1, z_1), P(z_1, y_1), S(x_1, y_1, w_1), R(y_1, w_1),$$

where w_1 is a new variable not in q_1. Now observe that $(q_2, (x_2, y_2)) \to (q_1', (x_1, y_1))$, which implies that $(q_2, (x_2, y_2)) \to (D, (a, b))$. By the Homomorphism Theorem, $(a, b) \in q_2(D)$, and thus, $q_1(D) \subseteq q_2(D)$.

Our goal is to revisit the problem of CQ containment in the presence of INDs. Given two CQs q and q', and a set Σ of INDs, q *is contained in* q' *under* Σ, denoted $q \subseteq_\Sigma q'$, if for every database D that satisfies Σ, $q(D) \subseteq q'(D)$. The problem of interest is defined as expected:

Problem: CQ-Containment-IND

Input: Two CQs q and q', and a set Σ of INDs
Output: `true` if $q \subseteq_\Sigma q'$, and `false` otherwise

Although the complexity of CQ containment in the presence of FDs remains NP-complete (Theorem 18.2), this is not true for INDs:

Theorem 18.6

CQ-Containment-IND is PSPACE-complete.

We first focus on the upper bound. Recall again that, by the Homomorphism Theorem, checking whether a CQ $q(\bar{x})$ is contained in a CQ $q'(\bar{x}')$ in the absence of constraints boils down to checking whether $(q', \bar{x}') \to (q, \bar{x})$. Although this is not enough in the presence of INDs, we can adopt a similar approach providing that we first transform, by adding atoms as dictated by the INDs, the set of atoms A_q occurring in q into a new set of atoms S that satisfies the INDs, and then check whether $(A_{q'}, \bar{x}') \to (S, \bar{x})$. This simple idea has been already illustrated by Example 18.5. As expected, the transformation of A_q into S can be achieved by exploiting the chase for INDs, which has been already introduced in Chapter 11.

We are going to establish a statement analogous to Theorem 18.3. However, since the chase for INDs may build an infinite set of atoms, we can only characterize CQ containment under possibly infinite databases. Notice that here we refer to the output of a CQ over a possibly infinite database. Although this is defined in the same way as for databases (Definition 13.3), we proceed to give the formal definition for the sake of completeness.

Consider a possibly infinite database D and a CQ q of the form

$$\text{Answer}(\bar{x}) \ \text{:-} \ R_1(\bar{u}_1), \ldots, R_n(\bar{u}_n) \,.$$

An *assignment* for q over D is a function η from the set of variables in q to $\text{Dom}(D)$. We say that η is *consistent* with D if

$$\{R_1(\eta(\bar{u}_1)), \ldots, R_n(\eta(\bar{u}_n))\} \ \subseteq \ D \,,$$

where, for $i \in [n]$, $R_i(\eta(\bar{u}_i))$ is the fact obtained after replacing each variable x in $\bar{u}_i$ with $\eta(x)$, and leave the constants in $\bar{u}_i$ untouched. Having this notion, we can define what is the output of a CQ on a possibly infinite database.

Definition 18.7: Evaluation on Possibly Infinite Databases

Given a possibly infinite database D of a schema $\mathbf{S}$, and a CQ $q(\bar{x})$ over $\mathbf{S}$, the *output* of q on D is defined as the set of tuples

$$q(D) \;=\; \{\eta(\bar{x}) \mid \eta \text{ is an assignment for } q \text{ over } D \text{ consistent with } D\}\,.$$

We can naturally talk about homomorphisms from CQs to possibly infinite databases. Actually, Definition 14.1 merely extends to possibly infinite databases, which allows us to state a result analogous to Theorem 14.2:

Theorem 18.8

Given a possibly infinite database D of a schema $\mathbf{S}$, and a CQ $q(\bar{x})$ of arity $k \geq 0$ over $\mathbf{S}$, it holds that

$$q(D) \;=\; \{\bar{a} \in \mathrm{Dom}(D)^k \;\mid\; (q, \bar{x}) \to (D, \bar{a})\}.$$

Consider two CQs q and q', and a set Σ of INDs. We say that q *is contained without restriction in* q' *under* Σ, denoted $q \subseteq^{\infty}_{\Sigma} q'$, if for every possibly infinite database D that satisfies Σ, $q(D) \subseteq q'(D)$. For brevity, we write $\mathrm{Chase}(q, \Sigma)$ instead of $\mathrm{Chase}(A_q, \Sigma)$. The next result is shown as Theorem 18.3.

Theorem 18.9

Let $q(\bar{x}), q'(\bar{x}')$ be CQs over schema $\mathbf{S}$, and Σ a set of INDs over $\mathbf{S}$. Then:

$$q \subseteq^{\infty}_{\Sigma} q' \quad \text{if and only if} \quad (A_{q'}, \bar{x}') \to (\mathrm{Chase}(q, \Sigma), \bar{x}).$$

The above statement alone is of little use since we are interested in finite databases. However, combined with the following result, known as the *finite controllability* of CQ containment under INDs, we get the desired characterization of CQ containment under finite databases via the chase.

Theorem 18.10: Finite Controllability of Containment

Let q and q' be CQs over a schema $\mathbf{S}$, and Σ a set of INDs over $\mathbf{S}$. Then:

$$q \subseteq_{\Sigma} q' \;\text{ if and only if }\; q \subseteq^{\infty}_{\Sigma} q'.$$

The above theorem is a deep result that is extremely useful for our analysis, but whose proof is out of the scope of this book. An easy consequence of Theorems 18.9 and 18.10, combined with Theorem 18.8, is the following:

Corollary 18.11

Let $q(\bar{x})$ and $q'(\bar{x}')$ be CQs over a schema $\mathbf{S}$, and Σ a set of INDs over $\mathbf{S}$. With $S = \text{Chase}(q, \Sigma)$, the following holds:

$$q \subseteq_\Sigma q' \text{ if and only if } \mathsf{G}_S(\bar{x}) \in q'(\mathsf{G}_S(S)).$$

Due to Corollary 18.11, the reader may be tempted to think that the procedure for checking whether $q \subseteq_\Sigma q'$, which in turn will lead to the PSPACE upper bound claimed in Theorem 18.6, is to check whether $\mathsf{G}_S(\bar{x})$ belongs to the evaluation of q' over $S^{\downarrow}$, where $S = \text{Chase}(q, \Sigma)$. However, it should not be forgotten that $\text{Chase}(q, \Sigma)$ may be infinite. Hence, we need a finer procedure that avoids the explicit construction of $\text{Chase}(q, \Sigma)$. We present a lemma that is the building block of this procedure, but first we need some terminology.

For an IND $\sigma = R[i_1, \ldots, i_m] \subseteq P[j_1, \ldots, j_m]$, a tuple $\bar{u} = (u_1, \ldots, u_{\text{ar}(R)})$, and a set of variables V, $\mathsf{new}^V(\sigma, \bar{u})$ is the atom obtained from $\mathsf{new}(\sigma, \bar{u})$ after replacing each newly introduced variable with a distinct variable from V. Formally, $\mathsf{new}^V(\sigma, \bar{u}) = P(v_1, \ldots, v_{\text{ar}(P)})$, where, for each $\ell \in [\text{ar}(P)]$,

$$v_\ell = \begin{cases} u_{i_k} & \text{if } \ell = j_k, \text{ for } k \in [m], \\ x \in V & \text{otherwise,} \end{cases}$$

such that, for each $i, j \in [\text{ar}(P)] - \{j_1, \ldots, j_m\}$, $i \neq j$ implies $v_i \neq v_j$.[1] Given two CQs $q(\bar{x}), q'(\bar{x}')$ over a schema $\mathbf{S}$, and a set Σ of INDs over $\mathbf{S}$, *a witness of q' relative to q and Σ* is a triple $(\mathcal{V}, \mathcal{S}, Q)$, where $\mathcal{V}$ is a sequence of (not necessarily disjoint) sets of variables $V_1, \ldots, V_n$, for $n \geq 0$, $\mathcal{S}$ is a sequence of disjoint sets of relational atoms $S_0, \ldots, S_n$, and $Q \subseteq \bigcup_{i \in [0,n]} S_i$, such that:

- $|\bigcup_{i \in [n]} V_i| \leq 3 \cdot |A_{q'}| \cdot \max_{R \in \mathbf{S}}\{\text{ar}(R)\}$,
- for each $i \in [n]$, $V_i \cap (\text{Dom}(S_{i-1}) \cup \text{Dom}(S)) = \emptyset$,
- for each $i \in [0, n]$, $|S_i| \leq |A_{q'}|$,
- $S_0 \subseteq A_q$,
- for each $i \in [n]$ and $P(\bar{v}) \in S_i$, there exists $\sigma = R[\alpha] \subseteq P[\beta]$ in Σ that is applicable on S_{i-1} with some $\bar{u} \in R^{S_{i-1}}$ such that $P(\bar{v}) = \mathsf{new}^{V_i}(\sigma, \bar{u})$,
- for each $i \in [n]$ and $x \in \text{Dom}(S_i) - \text{Dom}(S_{i-1})$, there is only one occurrence of x in S_i, i.e., it is mentioned only once by exactly one atom of S_i,
- $|Q| \leq |A_{q'}|$, and
- $\mathsf{G}_Q(\bar{x}) \in q'(\mathsf{G}_Q(Q))$.

[1] We assume some fixed mechanism that chooses the variable v_ℓ from the set V whenever $\ell \in [\text{ar}(P)] - \{j_1, \ldots, j_m\}$.

Let $S = \text{Chase}(q, \Sigma)$. Notice that $\mathsf{G}_S(\bar{x}) \in q'(\mathsf{G}_S(S))$ holds due to the existence of a set $A \subseteq \text{Chase}(q, \Sigma)$ such that $(A_{q'}, \bar{x}') \to (A, \bar{x})$. It is also not difficult to see that the construction of A can be witnessed via a sequence $A_0, A_1, \ldots, A_n$ of disjoint subsets of $\text{Chase}(q, \Sigma)$, where each such set consists of at most $|A_{q'}|$ atoms, $A_0 \subseteq A_{q'}$, $A_n = A$, and for each $i \in [n]$, the atoms of A_i are obtained from the atoms of A_{i-1} via chase applications using INDs of Σ. A witness of q' relative to q and Σ should be understood as a compact representation, which uses only polynomially many variables, of such a sequence $A_0, A_1, \ldots, A_n$ of disjoint subsets of $\text{Chase}(q, \Sigma)$. Therefore, the existence of a witness of q' relative to q essentially implies that $\mathsf{G}_S(\bar{x}) \in q'(\mathsf{G}_S(S))$. Furthermore, if $\mathsf{G}_S(\bar{x}) \in q'(\mathsf{G}_S(S))$, then a witness of q' relative to q and Σ can be extracted from $\text{Chase}(q, \Sigma)$. The above informal discussion is summarized in the following technical lemma, whose proof is left as an exercise.

Algorithm 5 ContainmentWitness(q, q', Σ)

Input: Two CQs $q(\bar{x})$ and $q'(\bar{x}')$ over $\mathbf{S}$, and a set Σ of INDs over $\mathbf{S}$.
Output: `true` if there is a witness for q' relative to q and Σ, and `false` otherwise.

$A_\triangledown := A$, where $A \subseteq A_q$ with $|A| \leq |A_{q'}|$
$A_\triangleright := \emptyset$
$Q := A$, where $A \subseteq A_\triangledown$
$V := \{y_1, \ldots, y_m\} \subset \mathsf{Var} - \text{Dom}(A_q)$ for some $m \in [3 \cdot |A_{q'}| \cdot \max_{R \in \mathbf{S}}\{\text{ar}(R)\}]$
repeat
  repeat
    if $\sigma = R[\alpha] \subseteq P[\beta] \in \Sigma$ is applicable on $A_\triangledown$ with $\bar{u} \in \text{Dom}(A_\triangledown)^{\text{ar}(R)}$ **then**
      $N := \mathsf{new}^V(\sigma, \bar{u})$
      $V := V - \text{Dom}(\{N\})$
      $A_\triangleright := A_\triangleright \cup \{N\}$
    if $|A_\triangleright| < |A_{q'}|$ **then**
      Next $:= b$, where $b \in \{0, 1\}$
    else
      Next $:= 1$
  until *Next* $= 1$
  if $A_\triangleright = \emptyset$ **then**
    return `false`
  $V := V \cup ((\text{Dom}(A_\triangledown) \cap \mathsf{Var}) - (\text{Dom}(A_\triangleright) \cup \text{Dom}(Q)))$
  $A_\triangledown := A_\triangleright$
  $A_\triangleright := \emptyset$
  $Q := Q \cup A$, where $A \subseteq A_\triangledown$
  if $|Q| < |A_{q'}|$ **then**
    Evaluate $:= b$, where $b \in \{0, 1\}$
  else
    Evaluate $:= 1$
until *Evaluate* $= 1$
return $\mathsf{G}_Q(\bar{x}) \in q'(\mathsf{G}_Q(Q))$

Lemma 18.12. *Let $q(\bar{x})$ and $q'(\bar{x}')$ be CQs over a schema $\boldsymbol{S}$, and Σ a set of INDs over $\boldsymbol{S}$. With $S = \text{Chase}(q, \Sigma)$, it holds that $\mathsf{G}_S(\bar{x}) \in q'(\mathsf{G}_S(S))$ if and only if there exists a witness of q' relative to q and Σ.*

By Corollary 18.11 and Lemma 18.12, we conclude that the problem of checking whether a CQ $q(\bar{x})$ is contained in a CQ $q'(\bar{x}')$ under a set Σ of INDs boils down to checking whether a witness of q' relative to q and Σ exists. This is done via the nondeterministic procedure shown in Algorithm 5. It essentially constructs the sequence of sets of variables $V_1, \ldots, V_n$, and the sequence of sets of atoms $S_0, \ldots, S_n$, required by a witness for q' relative to q and Σ, one after the other (if they exist), without storing more than two consecutive sets of a sequence during its computation. It also constructs on the fly the set of atoms Q. This is done by storing some of the atoms of a set S_i (possibly none) into Q before discarding it. Finally, the algorithm checks whether $\mathsf{G}_Q(\bar{x}) \in q'(\mathsf{G}_Q(Q))$, in which case it returns `true`; otherwise, it returns `false`. We proceed to give a bit more detailed description of Algorithm 5:

Initialization. The algorithm starts by guessing a subset of A_q with at most $|A_{q'}|$ atoms, which is stored in $A_\triangledown$ (see line 1); $A_\triangledown$ should be seen as the "current set" from which we construct the "next set" $A_\triangleright$ in the sequence. It also guesses a subset of $A_\triangledown$ that is stored in Q (see line 3); this step is part of the "on the fly" construction of the set Q. It also collects $3 \cdot |A_{q'}| \cdot \max_{R \in \mathbf{S}}\{\text{ar}(R)\}$ variables not occurring in A_q in the set V (see line 4).

Inner repeat-until loop. The inner repeat-until loop (see lines 6 - 15) is responsible for constructing the set $A_\triangleright$ from $A_\triangledown$. This is done by guessing an IND $\sigma \in \Sigma$ and a tuple $\bar{u}$ over $\text{Dom}(A_\triangledown)$, and adding to $A_\triangleright$ the atom $\mathsf{new}^V(\sigma, \bar{u})$ if σ is applicable on the current set $A_\triangledown$ with $\bar{u}$. It also removes from V the variables that has been used in $\mathsf{new}^V(\sigma, \bar{u})$ since they should not be reused in any other atom of $A_\triangleright$ that will be generated by a subsequent iteration. This is repeated until $A_\triangleright$ contains exactly $|A_{q'}|$ atoms, which means that its construction has been completed, or the algorithm nondeterministically chooses that its construction has been completed, even if it contains less than $|A_{q'}|$ atoms, by setting *Next* to 1. Once $A_\triangleright$ is in place, the algorithm updates V by adding to it the variables that occur in the current set $A_\triangledown$, but have not been propagated to $A_\triangleright$ and do not occur in Q (see line 18). This essentially gives rise to the next set of variables in the sequence of sets of variable under construction. Then $A_\triangledown$ is not needed further, and we can reuse the space that it occupies. The set $A_\triangleright$ becomes the current set $A_\triangledown$ (see line 19), while $A_\triangleright$ becomes empty (see line 20). Then the algorithm guesses a subset of $A_\triangledown$ that is stored in Q (see line 21); this step is part of the "on the fly" construction of Q.

Outer repeat-until loop. The above is repeated until Q contains more than $|A_{q'}|$ atoms (in the worst-case, $2 \cdot |A_{q'}|$ atoms), which means that its construction has been completed, or the algorithm nondeterministically chooses that its construction has been completed, even if it contains less

than $|A_{q'}|$ atoms, by setting *Evaluate* to 1. The algorithm returns `true` if $\mathsf{G}_{S'}(\bar{x}) \in q'(\mathsf{G}_{S'}(S'))$; otherwise, it returns `false`.

It is not difficult to verify that Algorithm 5 uses polynomial space, which is actually the space needed to represent the sets $A_\triangledown$, $A_\triangleright$, Q and V, as well as the space needed to check whether an IND is applicable on $A_\triangledown$ with some tuple $\bar{u} \in \mathrm{Dom}(A_\triangledown)^{\mathrm{ar}(R)}$ (see line 7), and the space needed to check whether $\mathsf{G}_Q(\bar{x}) \in q'(\mathsf{G}_Q(Q))$ (see line 27). This shows that CQ-Containment-IND is in NPSPACE, and thus in PSPACE since NPSPACE = PSPACE.

The PSPACE-hardness of CQ-Containment-IND is shown via a reduction from IND-Implication, which is PSPACE-hard (see Theorem 11.9). Recall that the IND-Implication problem takes as input a set Σ of INDs over a schema $\mathbf{S}$, and an IND σ over $\mathbf{S}$, and asks whether $\Sigma \models \sigma$, i.e., whether for every database over $\mathbf{S}$, $D \models \Sigma$ implies $D \models \sigma$. We are going to construct two CQs q and q' such that $\Sigma \models \sigma$ if and only if $q \subseteq_\Sigma q'$.

Assume that $\sigma = R[i_1, \ldots, i_k] \subseteq P[j_1, \ldots, j_k]$. The CQ q is defined as

$$\mathrm{Answer}(x_{i_1}, \ldots, x_{i_k}) \ \text{:-} \ R(x_1, \ldots, x_{\mathrm{ar}(R)}),$$

while the CQ q' is defined as

$$\mathrm{Answer}(x_{i_1}, \ldots, x_{i_k}) \ \text{:-} \ R(x_1, \ldots, x_{\mathrm{ar}(R)}), P(x_{f(1)}, \ldots, x_{f(\mathrm{ar}(R))}),$$

where, for each $m \in [\mathrm{ar}(P)]$,

$$f(m) \ = \ \begin{cases} i_\ell & \text{if } m = j_\ell, \text{ where } \ell \in [k], \\ \mathrm{ar}(R) + m & \text{otherwise.} \end{cases}$$

The function f ensures that the variable at position j_ℓ in the P-atom of q' is x_{i_ℓ}, i.e., the same as the one at position i_ℓ in the R-atom of q', while all the variables in the P-atom occurring at a position not in $\{j_1, \ldots, j_k\}$ are new variables occurring only once in the P-atom, and not occurring in the R-atom. It is an easy exercise to show that indeed $\Sigma \models \sigma$ if and only if $q \subseteq_\Sigma q'$.

19

Exercises for Part II

Exercise 2.1. Let q be the CQ given in Example 13.8. Express q as an RA query using θ-joins instead of Cartesian product.

Exercise 2.2. Prove the correctness of the translation of a CQ into an SPJ query, and the translation of an SPJ query into a CQ, given in the proof of Theorem 13.7, which establishes that the languages of CQs and of SPJ queries are equally expressive.

Exercise 2.3. For a CQ q, let e_q be the equivalent SPJ query obtained by applying the translation in the proof of Theorem 13.7. What is the size of e_q with respect to the size of q? Conversely, assuming that q_e is the CQ obtained after translating an SPJ query e into a CQ according to the translation in the proof of Theorem 13.7, what is the size of q_e with respect to the size of e?

Exercise 2.4. Prove that the choice of a pairing function in the definition of direct product does not matter. More precisely, let $\otimes_\tau$ be the direct product defined using a pairing function τ. Then, for every Boolean FO query q, every two databases D and D', and every two pairing functions τ and τ', show that $D \otimes_\tau D' \models q$ if and only if $D \otimes_{\tau'} D' \models q$.

Exercise 2.5. Let q be a Boolean FO query without constants over a schema $\mathbf{S}$. Prove that the following are equivalent:

1. There exists a CQ q' over $\mathbf{S}$ such that $q \equiv q'$, i.e., $q(D) = q'(D)$ for every database D of $\mathbf{S}$.
2. q is preserved under homomorphisms and direct products.

Exercise 2.6. The goal of this exercise is to extend the notion of preservation under direct products to queries with constants. To this end, we first refine the definition of a pairing function. Let $C \subseteq \mathsf{Const}$ be a finite set of constants, and τ_C a pairing function such that $\tau_C(a, a) = a$ for each $a \in C$. First, prove that such a pairing function exists. Then, prove that for any two databases D

and D' of the same schema $\mathbf{S}$, and for a Boolean CQ q over $\mathbf{S}$ that mentions only constants from C, if $D \models q$ and $D' \models q$, then $D \otimes D' \models q$, where the definition of $\otimes$ uses the pairing function τ_C.

Exercise 2.7. The goal is to extend further the notion of preservation under direct products to queries with constants that are not Boolean. For a finite set of constants $C \subseteq \mathsf{Const}$, let τ_C be a pairing function defined as in Exercise 2.6. Then, given two tuples $\bar{a} = (a_1, \ldots, a_n)$ and $\bar{b} = (b_1, \ldots, b_n)$, define the n-ary tuple $\bar{a} \otimes \bar{b}$ as $\big(\tau_C(a_1, b_1), \ldots, \tau_C(a_n, b_n)\big)$. Consider now an n-ary CQ $q(\bar{x})$ that mentions only constants from C. Show that if $\bar{a} \in q(D)$ and $\bar{b} \in q(D')$, then $\bar{a} \otimes \bar{b} \in q(D \otimes D')$, where $\otimes$ is defined with the pairing function τ_C.

Exercise 2.8. Use Exercise 2.6 to prove that the Boolean query $q = \exists x\,(x = a)$, where a is a constant, cannot be expressed as a CQ.

Exercise 2.9. Use Exercise 2.7 to prove that the query $q = \varphi(x, y)$, where φ is the equational atom $(x = y)$, cannot be expressed as a CQ.

Exercise 2.10. Consider a parameterized problem (L_1, κ_1) over Σ_1, and a parameterized problem (L_2, κ_2) over Σ_2. Show that if there is an FPT-reduction from (L_1, κ_1) to (L_2, κ_2), and $(L_2, \kappa_2) \in \mathrm{FPT}$, then $(L_1, \kappa_1) \in \mathrm{FPT}$.

Exercise 2.11. Recall that in the proof of the fact that p-CQ-Evaluation is in W[1] (see Theorem 15.7), for technical clarity, we consider only constant-free Boolean CQs over a schema consisting of a single binary relation name. Prove that p-CQ-Evaluation is in W[1] even for arbitrary CQs.

Exercise 2.12. Show Corollary 16.7 for arbitrary (non-Boolean) CQs.

Exercise 2.13. Show that the binary relation $\equiv$ over CQs is an equivalence relation, i.e., is reflexive, symmetric, and transitive. Show also that the binary relation $\subseteq$ over CQs is reflexive and transitive, but not necessarily symmetric.

Exercise 2.14. Answer the following questions about CQs and their cores.

(i) Consider the Boolean CQ q_1 over the schema $\{E[2]\}$ defined as

$$\text{Answer} \;:\!-\; E(x_1, y_1), E(y_1, z_1), E(z_1, w_1), E(w_1, x_1), E(x_2, y_2), E(y_2, x_2).$$

Assume that E is used to represent the edge relation of a graph G. What q_1 checks for G? Compute the core of q_1.

(ii) Consider the Boolean CQ q_2 over the schema $\{R[1], S[1]\}$ defined as

$$\text{Answer} \;:\!-\; R(x), S(x), R(y), S(y).$$

Compute the core of q_2.

(iii) Consider the CQ q_3 over the schema $\{R[1], S[1]\}$ defined as

$$\text{Answer}(x, y) \ :- \ R(x), S(x), R(y), S(y).$$

Prove that q_3 is a core of itself.

Exercise 2.15. Let $q(\bar{x})$ be a CQ, and $q'(\bar{x})$ a core of $q(\bar{x})$. Prove that there is a homomorphism from $(q, \bar{x})$ to $(q', \bar{x})$ that is the identity on $\text{Dom}(S_{q'})$.

Exercise 2.16. Recall that ComputeCore (see Algorithm 4) is nondeterministic. Devise a deterministic algorithm that computes the core of a CQ, and show that it runs in exponential time in the size of the input query.

Exercise 2.17. (a) Let CQ-Minimization be the problem where, given a Boolean CQ q and integer $k \in \mathbb{N}$, the question is if there exists a CQ q' such that $q' \equiv q$ and $|q'| \leq k$. Prove that CQ-Minimization is NP-complete.

(b) Let CQ-Minimality be the problem where, given a Boolean CQ q, the question is to answer `true` if q is minimal and `false` otherwise. Prove that CQ-Minimality is coNP-complete.

Exercise 2.18. Let D be a database, and $T = \{\bar{a}_1, \ldots, \bar{a}_n\}$ a set of m-ary tuples over $\text{Dom}(D)$, for $m > 0$. Show that there exists a CQ $q(\bar{x})$ such that $q(D) = T$ if and only if the following hold:

1. $\prod_{i \in [n]} \bar{a}_i$ appears in $\prod_{i \in [n]} D$, and
2. there is no tuple $\bar{b} \in \text{Dom}(D)^m - T$ such that $\prod_{i \in [n]} (D, \bar{a}_i) \to (D, \bar{b})$.

Exercise 2.19. The purpose of this exercise is to understand what happens if we allow equalities of the form $x = y$ or $x = a$ in CQs. We define a *conjunctive query with equalities* ($\text{CQ}^=$) similarly to a CQ, but we additionally allow equational atoms. Such queries can therefore be written as rules

$$\text{Answer}(\bar{x}) \ :- \ R_1(\bar{u}_1), \ldots, R_n(\bar{u}_n), y_1 = v_1, \ldots, y_k = v_k,$$

where $\{v_1, \ldots, v_k\} \subseteq \mathsf{Var} \cup \mathsf{Const}$.

1. Why are the queries $\text{Answer}(x) \ :- \ x = y$ and $\text{Answer}(x) \ :- \ x = a$ not expressible as CQs?
2. Prove that (i) Theorem 15.1, (ii) Theorem 16.3, and (iii) Theorem 16.8 also hold for $\text{CQ}^=$.
3. Prove that queries in $\text{CQ}^=$ can be minimized with a variant of Algorithm 4.

Exercise 2.20. Prove that the following problem is coNExpTime-complete: given a database D, and a set $T = \{\bar{a}_1, \ldots, \bar{a}_n\}$ of m-ary tuples over $\text{Dom}(D)$, for $m > 0$, check whether there exists a CQ q such that $q(D) = T$.

Exercise 2.21. Prove that FO-Containment remains undecidable even if one of the two input queries is a CQ.

Exercise 2.22. Prove Lemma 18.12.

Exercise 2.23. Prove that the reduction at the end of Chapter 18 from IND-Implication to CQ-Containment-IND, which establishes that the latter is PSPACE-hard, is correct.

Bibliographic Comments for Part II

To be done.

Part III

Fast Conjunctive Query Evaluation

Motivation

Here we are interested in understanding when CQ evaluation can be solved efficiently in combined complexity. In Theorem 15.1, we have shown that CQ evaluation is NP-complete by reducing from an NP-complete problem over graphs. It is known, on the other hand, that several NP-complete problems over graphs become tractable if they are restricted to be *nearly acyclic*. As we show in this part of the book, similar ideas can be applied to prove that CQ evaluation is tractable when CQs are nearly acyclic. This is highly relevant from a practical point of view, as such CQs appear often in real-world applications.

20

Acyclicity of Conjunctive Queries

We start by studying the notion of *acyclicity* for CQs, which has received considerable attention in the database literature since the early 1980s. In this chapter, we define acyclicity and present an algorithm to recognize it. In the next chapter, we will present two algorithms that show that acyclic CQs can be evaluated efficiently.

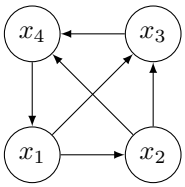
CQs and Hypergraphs

We have seen in Theorem 15.1 that the evaluation problem for CQs is NP-complete. So, the reader may wonder why it is possible that database systems are successful in practice even though its most fundamental problem on the most common class of queries is NP-complete.

The crux is that the *shape* of a CQ plays a very important role in how complex the query is to evaluate. For instance, assume that we have a database D over the schema $\{E[2]\}$. Hence, D can be understood as a directed graph where E is the edge relation. Evaluating the CQ

$$\text{Answer} :- E(x_1, x_2), E(x_2, x_3), E(x_3, x_4), E(x_4, x_1), E(x_1, x_3), E(x_2, x_4)$$

can be understood as matching a variant of the 4-clique, namely a graph of the form



in D. In fact, this correspondence between evaluation of CQs and graph matching is precisely what we used in Theorem 15.1 to encode the Clique problem into the CQ-Evaluation problem.

However, CQs in practice are usually not shaped as cliques. Instead, *tree-shaped* CQs are much more common. Since it is well-known that finding cliques in graphs is computationally difficult, whereas finding tree-like structures in graphs is much easier, it makes sense to study the evaluation problem of CQs for which the associated graph is acyclic.

To make this precise, however, we need to consider a generalization of (undirected) graphs that can deal with relations of arity three or more. Such graphs are called *hypergraphs*.

Definition 20.1: Hypergraph

A *hypergraph* is a pair $H = (V, E)$, where

- V is a finite set of *nodes* and
- E is a set of subsets of V, called *hyperedges*.

Acyclicity of Hypergraphs

Defining the notion of acyclicity for hypergraphs is not as simple as it is for graphs. In fact, several natural, non-equivalent notions of acyclicity for hypergraphs exist. We work here with one such a notion, often referred to as α-*acyclicity*, which has received considerable attention in database theory.

We will call a hypergraph H *acyclic* if it admits a *join tree*, that is, if its hyperedges can be arranged in the form of a tree, while preserving the connectivity of elements that occur in different hyperedges.

Definition 20.2: Join Tree and Acyclic Hypergraph

Given a hypergraph $H = (V, E)$, a tree T is a *join tree* of H if

- the nodes of T are precisely the hyperedges in E and,
- for each node $v \in V$, the set of nodes of T in which v is an element forms a connected subtree of T.

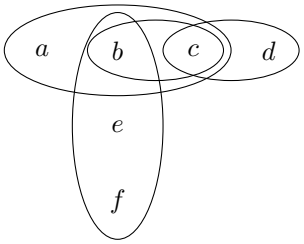
Moreover, H is *acyclic* if H admits a join tree.

We refrain from formally introducing trees at this point in the book, since we do not need them in a sophisticated manner at this point. That said, readers who would like to have a formal definition can look at Chapter 58.[1]

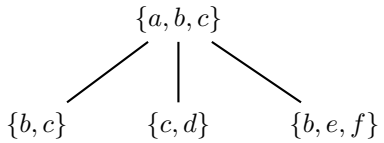
[1] In the terminology of Chapter 58, we use labeled unordered trees in which each node n is also labeled with n.

Example 20.3: Acyclic and Non-Acyclic Hypergraphs

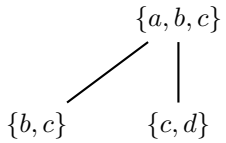
Consider the following hypergraph $H_1 = (V_1, E_1)$:



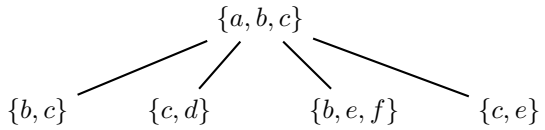
Thus, we have that $V_1 = \{a, b, c, d, e, f\}$ and $E_1 = \{\{a, b, c\}, \{b, c\}, \{c, d\}, \{b, e, f\}\}$. It holds that H_1 is an acyclic hypergraph, as the following tree T_1 is a join tree for H_1:



In fact, we have that T_1 is a join tree of H_1 as the nodes of T_1 are precisely the hyperedges in E_1, and for each $v \in V_1$, the set of nodes of T_1 in which v occurs defines a connected subtree of T_1. As an example of this latter condition, if we consider $v = c$, then we obtained the following subtree of T_1 that is connected:



On the other hand, consider a hypergraph H_2 that extends H_1 with the hyperedge $\{c, e\}$. We have that H_2 is not acyclic, as it is not possible to construct a join tree for it. For instance, consider the extension T_2 of T_1 that is obtained by adding a node $\{c, e\}$ and connecting it with the node $\{a, b, c\}$ of T_1:



Then we have that T_2 is not a join tree for H_2 as the set of nodes of T_2 in which e occurs do not define a connected subtree of T_2:

$$\{b, e, f\} \qquad \{c, e\}$$

It is not hard to see that for undirected graphs, the notion of α-acyclicity coincides with the usual notion of acyclicity that stems from graph theory (i.e., tree-shaped or forest-shaped graphs).

Acyclicity of Conjunctive Queries

The notion of acyclic hypergraph is the key concept in the definition of acyclic CQs. Each CQ q is naturally associated with a hypergraph H_q that represents the structure of joins among its variables. In particular, if q is of the form

$$q(\bar{x}) \coloneq R_1(\bar{u}_1), \ldots, R_n(\bar{u}_n) \ ,$$

where $\bar{u}_i$ is a tuple of constants and variables for every $i \in [n]$, then $H_q = (V, E)$ is a hypergraph such that

- its set V of vertices contains all variables mentioned in q and
- the hyperedges in E are precisely the sets of variables appearing in the atoms of q, i.e., $E = \{X_i \mid i \in [n]\}$, where X_i is the set of variables occurring in $\bar{u}_i$.

Definition 20.4: Acyclicity of CQs

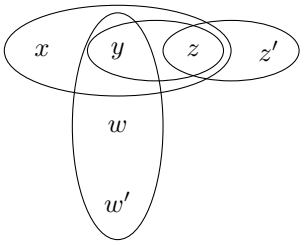
An *acyclic conjunctive query* (ACQ) is a conjunctive query q such that its associated hypergraph H_q is acyclic.

Example 20.5: Acyclic and Non-Acyclic CQs

Consider the following CQ q_1:

$$q_1(x, y) \coloneq R(x, y, z), T(y, z), S(y, w, w'), T(z, z').$$

Then we have that H_{q_1} is the following hypergraph:



We know from Example 20.3 that H_{q_1} is an acyclic hypergraph, as H_{q_1} can be obtained from the hypergraph H_1 in Example 20.3 by renaming the nodes of H_1. Therefore, we have that q_1 is an acyclic CQ. In the same way, we obtain that the following CQ q_2:

$$q_2(x, y) \text{ :- } R'(x, y, z, a), T'(y, z, a), S(y, w, w'), T'(z, z', b)$$

is acyclic as $H_{q_2} = H_{q_1}$. In particular, notice that constants a, b in q_2 do not play any role in the construction of H_{q_2}. On the other hand, the following CQ q_3:

$$q_3(x, y) \text{ :- } R(x, y, z), T(y, z), S(y, w, w'), T(z, z'), T(z, w)$$

is not acyclic as the hypergraph H_{q_3} is not acyclic. Notice that this latter fact is also obtained from Example 20.3, as H_{q_3} can be obtained from the hypergraph H_2 in Example 20.3 by renaming the nodes of H_2. Finally, consider the following CQ q_4:

$$q_4(x, y) \text{ :- } R(x, y, z), R(y, y, z), T(y, z), S(y, w, w'), T(z, z').$$

Then we have that q_3 is an acyclic CQ since $H_{q_3} = H_{q_1}$. Notice that this latter condition holds as the set of variable occurring in $R(y, y, z)$ is $\{y, z\}$, which is the same as the set of variables occurring in $T(y, z)$.

Acyclicity Recognition

In the following chapter, we will show that ACQs can be evaluated efficiently. But before doing so, it is important to explain why acyclicity itself can be efficiently recognized. This follows from the existence of an equivalent definition of acyclicity in terms of an iterative process described in the following proposition.

Proposition 20.6: GYO Algorithm

A hypergraph $H = (V, E)$ is acyclic if and only if all of its vertices can be deleted by repeatedly applying the following two operations (in no particular order):

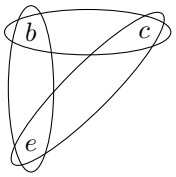
1. Delete a vertex that appears in at most one hyperedge.
2. Delete a hyperedge that is contained in another hyperedge.

This characterization leads directly to a polynomial-time algorithm for checking acyclicity of hypergraphs, and thus of CQs: Given a CQ q, we apply operations (1) and (2) in the statement of Proposition 20.6 over H_q until a fixpoint is reached. The query q is acyclic if and only if we are left with no vertices. Interestingly, a simple extension of this algorithm also constructs a join tree of H_q when q is acyclic (see Exercise 3.2). It is easy to see that the algorithm runs in quadratic time in the size of H_q and, therefore, in the size of the CQ q.

Example 20.7: Application of GYO Algorithm

Consider the hypergraph H_1 in Example 20.3. As expected, by using the previous algorithm we obtain that H_1 is acyclic. In fact, all vertices of H_1 are deleted by applying the following sequence of operations: delete vertex d (that appears only in hyperedge $\{c, d\}$), delete vertices e and f, delete hyperedges $\{b\}$ and $\{c\}$ (that are contained in hyperedge $\{b, c\}$), delete hyperedge $\{b, c\}$ (that is contained in hyperedge $\{a, b, c\}$), and delete vertices a, b and c.

On the other hand, and also as expected, by applying the previous algorithm on hypergraph H_2 from Example 20.3, we obtain that H_2 is not acyclic. In fact, no matter what order is used when applying the two operations of the algorithm, we reach the following fixed point:



Notice that no operation can be applied to reduce this hypergraph, which is intuitively correct as this hypergraph represents the canonical example of an undirected graph that is not acyclic.

It is important to mention that there are more sophisticated algorithms that check whether a CQ q is acyclic and construct a join tree of H_q if the latter is the case, in time $O(\|H_q\|)$, that is, linear time. We will exploit this fact later in the presentation of efficient evaluation algorithms for acyclic conjunctive queries.

21

Efficient Acyclic Conjunctive Query Evaluation

We will present two algorithms that show that acyclic CQs can be evaluated efficiently. The first one, known as Yannakakis's algorithm, makes use of the decomposition of an acyclic CQ as a join tree, which was defined in the previous chapter, while the second one is based on a simple *consistency* criterion. Yannakakis's algorithm achieves a relatively efficient running time of $O(\|D\| \cdot \log \|D\| \cdot \|q\|)$, where D is the database and q is the query. On the other hand, the algorithm based on the consistency criterion has the advantage that it does not require the CQ itself to be acyclic, only its *core* (as defined in Chapter 17).

Semijoins and Acyclic CQs

The evaluation of acyclic CQs is tightly related to a particular relational algebra operation, known as *semijoin*, which we describe next. Let D be a database. Given CQs $q(\bar{x})$ and $q'(\bar{x}')$ and tuples $\bar{a} \in q(D)$ and $\bar{b} \in q'(D)$, we call $\bar{a}$ and $\bar{b}$ *consistent* if they have the same value on each position that contains a common variable of $\bar{x}$ and $\bar{x}'$. We then define the semijoin of $q(D)$ and $q'(D)$, denoted by $q(D) \ltimes q'(D)$, as the set of tuples $\bar{a} \in q(D)$ that are consistent with some tuple $\bar{b} \in q'(D)$.

Example 21.1: Semijoin of Conjunctive Queries

Assume that $D = \{R(a,b,c), R(d,d,d), S(c,b,e), S(d,e,e)\}$, and that q and q' are the following CQs:

$$\begin{aligned} q(x,y,z) &:- R(x,y,z) \\ q'(z,y,w) &:- S(z,y,w). \end{aligned}$$

Then we have that $q(D) = \{(a,b,c), (d,d,d)\}$, $q'(D) = \{(c,b,e), (d,e,e)\}$, and $q(D) \ltimes q'(D) = \{(a,b,c)\}$. In particular, we have that (a,b,c) is in

$q(D) \ltimes q'(D)$ since (a, b, c) belongs to $q(D)$ and (a, b, c) is consistent with the tuple $(c, b, e) \in q'(D)$, as these two tuples have the same value b in the position that corresponds to the variable y shared by (x, y, z) and (z, y, w), and have the same value c in the position that corresponds to the variable z shared by (x, y, z) and (z, y, w). Moreover, we have that (d, d, d) is not in $q(D) \ltimes q'(D)$ as this tuple is not consistent with any tuple in $q'(D)$. Finally, notice that $q'(D) \ltimes q(D) = \{(c, b, e)\}$, which shows that, as opposed to the case of the join operator, $\ltimes$ is not a commutative operator.

We now explain the relationship between the CQs in ACQ and the semijoin operator. Let $q :\!- R_1(\bar{u}_1), \ldots, R_n(\bar{u}_n)$ be a Boolean ACQ, and consider an arbitrary join tree T of H_q. Recall that the set of nodes of T is $\{X_i \mid i \in [n]\}$, where each X_i is the set of variables occurring in $\bar{u}_i$. We see T as a rooted and directed tree by arbitrarily choosing a root r and directing all edges away from r. Therefore, we can naturally talk about the *leaves* of T and about the *children* or *descendants* of a node in T. For every node s of T, we define the following CQs for some $i \in [n]$, assuming that $s = X_i$ and that $\bar{y}_i$ is a tuple of pairwise distinct variables consisting exactly of the variables in X_i:

- A CQ $q_s(\bar{y}_i) :\!- R_{j_1}(\bar{u}_{j_1}), \ldots, R_{j_p}(\bar{u}_{j_p})$, where $\{R_{j_1}(\bar{u}_{j_1}), \ldots, R_{j_p}(\bar{u}_{j_p})\}$ is the set of of atoms of q such that $X_{j_\ell} = X_i$ for each $\ell \in [p]$.
- A CQ $Q_s(\bar{y}_i)$ whose set atoms is the union of those that appear in CQs $q_{s'}$, where s' is a descendant of s in T (including s itself).

Notice that $Q_s \subseteq q_s$ for each node s of T. Moreover, if s is a (non-leaf) node of T with children $s_1, \ldots, s_p$, then the set of atoms of Q_s is the union of the atoms of Q_{s_1}, …, Q_{s_p}, and the atoms of q_s.

In what follows, we present a fundamental connection between the evaluation of acyclic CQs and the semijoin operator. To understand this connection, assume that $q :\!- R_1(\bar{u}_1), \ldots, R_n(\bar{u}_n)$ is a Boolean ACQ, and suppose that T is a rooted and directed join tree of H_q with root $r = X_\ell$, for some $\ell \in [n]$. Then we have that Q_r is a CQ of the form $Q_r(\bar{y}_\ell) :\!- R_1(\bar{u}_1), \ldots, R_n(\bar{u}_n)$, which means that Q_r has the same body as q. Thus, for every database D, it holds that

$$q(D) = \texttt{true} \text{ if and only if } Q_r(D) \neq \emptyset$$

and, therefore, an efficient algorithm for the evaluation of Q_r can also be used to evaluate q. We show in the next section that the followin proposition gives us such an algorithm. The proposition tells us that Q_r can be inductively evaluated by computing semijoins while traversing T in a bottom-up manner, provided that the CQ q_s has been previously evaluated for every node s of T.

Proposition 21.2

Let q be a Boolean ACQ, T a rooted and directed join tree of H_q and D a database. Then for every node s of T,

- if s is a leaf of T, then $Q_s(D) = q_s(D)$ and
- otherwise, if the children of s in T are $s_1, \ldots, s_p$, then

$$Q_s(D) \;=\; \bigcap_{i=1}^{p} \big(q_s(D) \ltimes Q_{s_i}(D)\big) \,.$$

Proof. Assume that $q :- R_1(\bar{u}_1), \ldots, R_n(\bar{u}_n)$ is the Boolean ACQ. Therefore, the set of nodes of T is $\{X_i \mid i \in [n]\}$, where each X_i is the set of variables occurring in $\bar{u}_i$. If s is a leaf of T, then q_s and Q_s are the same CQ and, thus, $Q_s(D) = q_s(D)$.

Let us assume then that s is a non-leaf node of T with children $s_1, \ldots, s_p$. Moreover, assume $s = X_\ell$, $s_1 = X_{k_1}, \ldots s_p = X_{k_p}$, where $\ell, k_1, \ldots, k_p$ are pairwise distinct numbers in the set $[n]$. Then we have that $\bar{y}_\ell$, $\bar{y}_{k_1}$, …, $\bar{y}_{k_p}$ are the tuples of free variables of CQs Q_s, Q_{s_1}, …, Q_{s_p}, respectively. Let us consider first an arbitrary tuple in $Q_s(D)$. By definition, such a tuple is of the form $h(\bar{y}_\ell)$ for some homomorphism h from Q_s to D. It is not hard to see that $h(\bar{y}_\ell) \in q_s(D) \ltimes Q_{s_i}(D)$ for every $i \in [p]$. Indeed, $h(\bar{y}_\ell) \in q_s(D)$ since $Q_s \subseteq q_s$, and $h(\bar{y}_{k_i}) \in Q_{s_i}(D)$ since the atoms of Q_{s_i} are contained in those of Q_s. Moreover, $h(\bar{y}_\ell)$ and $h(\bar{y}_{k_i})$ are consistent by definition. We conclude that $h(\bar{y}_\ell) \in \bigcap_{i=1}^{p} \big(q_s(D) \ltimes Q_{s_i}(D)\big)$.

Let us consider now an arbitrary tuple in $\bigcap_{i=1}^{p} \big(q_s(D) \ltimes Q_{s_i}(D)\big)$. By definition, such a tuple is of the form $h(\bar{y}_\ell)$ for some homomorphism h from q_s to D. Moreover, for each $i \in [p]$, there is a homomorphism h_i from Q_{s_i} to D such that $h(\bar{y}_\ell)$ and $h_i(\bar{y}_{k_i})$ are consistent; i.e., they have the same values on positions where $\bar{y}_\ell$ and $\bar{y}_{k_i}$ have common variables. We claim that $h' = h \cup h_1 \cup \cdots \cup h_p$ is a well-defined homomorphism from Q_s to D. Since $h'(\bar{y}_\ell) = h(\bar{y}_\ell)$, this shows that $h(\bar{y}_\ell) \in Q_s(D)$ as desired.

We first show that h' is well-defined. Take an arbitrary variable y in Q_s. If y occurs only in q_s but not in any of the CQs Q_{s_i} (for $i \in [p]$), or if y occurs only in one of the CQs Q_{s_i} (for $i \in [p]$) but not in q_s, then clearly $h'(y)$ is well-defined. There are two other possibilities: y occurs in q_s and in Q_{s_i}, for some $i \in [p]$, or y occurs in Q_{s_i} and Q_{s_j}, for some $i, j \in [p]$ with $i \neq j$. We only consider the latter case since the former can be handled analogously. By definition of join trees, the nodes in T that contain y are connected, which means that $y \in s \cap s_i \cap s_j$. Therefore, we conclude that $h_i(y) = h_j(y) = h(y)$ since $h(\bar{y}_\ell)$ is consistent with both $h_i(\bar{y}_{k_i})$ and $h_j(\bar{y}_{k_j})$.

We now prove that h' is a homomorphism from Q_s to D. Take an arbitrary atom $R(\bar{z})$ in Q_s. Then, $R(h'(\bar{z})) = R(h(\bar{z}))$ or $R(h'(\bar{z})) = R(h_i(\bar{z}))$ for some $i \in [p]$. Thus, $R(h'(\bar{z})) \in D$ because h and h_i are homomorphisms. This concludes the proof of the proposition. □

Yannakakis's Algorithm

Yannakakis's algorithm uses the conditions in Proposition 21.2 to evaluate a Boolean ACQ, as shown in Algorithm 6. The soundness and completeness of the algorithm follows from Proposition 21.2—which justifies the correctness of the inductive computation carried out in the while loop—and the fact that the atoms of Q_r are precisely those of q, from which we conclude that $Q_r(D) \neq \emptyset$ if and only if $q(D) = \texttt{true}$.

Algorithm 6 YANNAKAKIS(q, D)

Input: A Boolean ACQ q and a database D
Output: $q(D)$

```
1: T := a rooted and directed join tree of H_q
2: N := the set of nodes of T
3: r := the root of T
4: while N ≠ ∅ do
5:     Choose s ∈ N such that no child of s is in N
6:     Compute q_s(D)
7:     if s is a leaf of T then
8:         Q_s(D) := q_s(D)
9:     else
10:        Let s_1, ..., s_p be the children of s in T
11:        Q_s(D) := ⋂_{i=1}^{p} (q_s(D) ⋉ Q_{s_i}(D))
12:    N := N − {s}
13: return ¬(Q_r(D) = ∅)
```

We now analyze the complexity of the algorithm. We remarked at the end of Chapter 20 that a join tree T of H_q can be computed in time $O(\|H_q\|)$, which is in time $O(\|q\|)$. We show next that the remainder of the algorithm can be implemented in time $O(\|D\| \cdot \log \|D\| \cdot \|q\|)$. To see why this is the case, we need the following observation (see Exercise 3.3): the cost of computing $q(D) \ltimes q'(D)$, given $q(D)$ and $q'(D)$, is time $O(N \log N)$ where $N = \|q(D)\| + \|q'(D)\|$. In particular, then, each $q_s(D)$, for a node s in T, can be computed in time $O(\|D\| \cdot \log \|D\| \cdot \|q_s\|)$. Therefore, the collection of all queries $q_s(D)$, for s a node in T, can be computed in time $O(\|D\| \cdot \log \|D\| \cdot \|q\|)$.

Now, if s is a node of T with children $s_1, \ldots, s_p$, we can compute $Q_s(D) = \bigcap_{1 \leq i \leq p} q_s(D) \ltimes Q_{s_i}(D)$ in time $O(\|D\| \cdot \log \|D\| \cdot p)$. This follows from the fact that $\|q_s(D)\| \leq \|D\|$ and $\|Q_{s_i}(D)\| \leq \|q_{s_i}(D)\| \leq \|D\|$, for each $i \in [p]$.

Therefore, we can inductively compute the collection of all queries Q_s, for s a node in T, in time $O(\|D\| \cdot \log \|D\| \cdot \|T\|) = O(\|D\| \cdot \log \|D\| \cdot \|q\|)$.

In summary, we obtain the following result:

Theorem 21.3

ACQ-Evaluation can be solved in time $O(\|D\| \cdot \log \|D\| \cdot \|q\|)$.

Proof. We already proved in the analysis preceding the theorem that the theorem holds for Boolean ACQs. Assume now that we are given a non-Boolean ACQ $q(\bar{x}) :- R_1(\bar{u}_1), \ldots, R_n(\bar{u}_n)$, a database D, and a tuple $\bar{a}$ over Const of the same arity than $\bar{x}$. We want to check whether $\bar{a} \in q(D)$. We start by turning $q(\bar{x})$ into a Boolean ACQ by simultaneously replacing in $q(\bar{x})$ each free variable x_i in $\bar{x} = (x_1, \ldots, x_k)$ by its corresponding value a_i in $\bar{a} = (a_1, \ldots, a_k)$. We denote this boolean CQ as $q_{\bar{a}}$. Clearly, $q_{\bar{a}}$ is acyclic and, in addition, $\bar{a} \in q(D)$ if and only if $q_{\bar{a}}(D) = \texttt{true}$. □

The Consistency Algorithm

While Yannakakis's algorithm uses a join tree of an acyclic CQ q in order to evaluate q over a database D in time $O(\|D\| \cdot \log \|D\| \cdot \|q\|)$, if we only aim for tractability then there is no need for such a join tree to be computed. In fact, below we present an algorithm that evaluates q on D in polynomial time, only by holding the *promise* that q is acyclic (i.e., that a join tree of q exists). The design of such an algorithm is based on a simple consistency criterion, established in the following proposition, which characterizes when $q(D) = \texttt{true}$ for a Boolean ACQ q and a database D.

Proposition 21.4: Consistency Property

Let $q :- R_1(\bar{u}_1), \ldots, R_n(\bar{u}_n)$ be a Boolean ACQ and D a database, and $q_i(\bar{x}_i) :- R_i(\bar{u}_i)$ be a CQ such that $\bar{x}_i$ is the tuple obtained from $\bar{u}_i$ by removing constants, for each $i \in [n]$. Then the following are equivalent:

1. $q(D) = \texttt{true}$.
2. There are nonempty sets $S_1 \subseteq q_1(D), \ldots, S_n \subseteq q_n(D)$ such that

$$S_i = S_i \ltimes S_j \text{ for all } i, j \in [n] \,.$$

That is, each tuple in S_i is consistent with some tuple in S_j for all $i, j \in [n]$.

Proof. Assume first that $q(D) = \texttt{true}$, i.e., there is a homomorphism h from q to D. In this case, we can choose S_i to be $\{h(\bar{x}_i)\}$, for each $1 \leq i \leq n$.

For the other direction, assume that nonempty sets $S_1, \ldots, S_n$ as described in item 2 exist. Let T be an arbitrary rooted and directed join tree of q. One can then prove by induction the following for each node s of T:

If s is the set X_i of variables occurring in $\bar{x}_i$, for $i \in [n]$, then $S_i \subseteq Q_s(D)$ (see Exercise 3.5).

In particular, if the root r of T is the set X_j of variables occurring in $\bar{x}_j$, for $j \in [n]$, then $S_j \subseteq Q_r(D)$. Therefore, since S_j is nonempty we conclude that $Q_r(D)$ is also nonempty. This implies that there is at least one homomorphism from Q_r to D. But the atoms of Q_r and q are the same by definition, and thus $q(D) = \texttt{true}$. □

We are ready to present the consistency algorithm, which can be understood as a greatest fixed-point computation that checks for the existence of nonempty sets $S_1, \ldots, S_n$ as described in item 2 of Proposition 21.4.

Algorithm 7 CONSISTENCY(q, D)

Input: A Boolean CQ $q :- R_1(\bar{u}_1), \ldots, R_n(\bar{u}_n)$ in ACQ and a database D
Output: $q(D)$

1: $S_i := q_i(D)$, for each $i \in [n]$
2: **while** $S_i \neq S_i \ltimes S_j$ for some $i, j \in [n]$ **do**
3: $\quad S_i := S_i \ltimes S_j$
4: **if** $S_i \neq \emptyset$ for every $i \in [n]$ **then return** `true`
5: **else return** `false`

The algorithm initializes S_i to be $q_i(D)$, for each $i \in [n]$. It then iteratively deletes every tuple in S_i that is not consistent with a tuple in S_j, for some $j \in [n]$. If some S_i becomes emtpy during this procedure, the algorithm declares $q(D) = \texttt{false}$. Otherwise, $q(D) = \texttt{true}$. The algorithm runs in polynomial time, but not in time $O(\|D\| \cdot \log \|D\| \cdot \|q\|)$ as Yannakakis's algorithm. Next, we establish that it is sound and complete.

Proposition 21.5

Given a boolean CQ q in ACQ and a database D, we have that $q(D) = \texttt{true}$ if and only if CONSISTENCY$(q, D) = \texttt{true}$.

We leave the proof of Proposition 21.5 as an exercise for the reader.

Acyclicity of the Core

It is known that there are boolean CQs that are not acyclic, yet their core is acyclic. (The reader is asked to prove this fact in Exercise 3.6). Interestingly,

Bibliographic Comments for Part III

To be done.

Part IV

Expressive Languages

30

Unions of Conjunctive Queries

The first, and simplest, addition to conjunctive queries is *union*, which leads to the language of union of conjunctive queries.

Definition 30.1: Union of Conjunctive Queries

A *union of conjunctive queries* (UCQ) over a schema $\mathbf{S}$ is an FO query $\varphi(\bar{x})$ over $\mathbf{S}$ where φ is a formula of the form

$$\varphi_1 \vee \cdots \vee \varphi_n$$

for $n \geq 1$, where $\mathrm{FV}(\varphi) = \mathrm{FV}(\varphi_i)$ and $\varphi_i(\bar{x})$ is a CQ, for every $i \in [n]$.

For notational convenience, we denote a UCQ $q = \varphi(\bar{x})$ with $\varphi = \varphi_1 \vee \cdots \vee \varphi_n$ as $q_1 \cup \cdots \cup q_n$, where q_i is the CQ $\varphi_i(\bar{x})$, for each $i \in [n]$. It is not difficult to verify that, given a database D of a schema $\mathbf{S}$, and a UCQ $q = q_1 \cup \cdots \cup q_n$ over $\mathbf{S}$, it holds that $q(D) = q_1(D) \cup \cdots \cup q_n(D)$.

Example 30.2: Union of Conjunctive Queries

Consider the relational schema from Example 3.2:

```
Person [ pid, pname, cid ]
Profession [ pid, prname ]
City [ cid, cname, country ]
```

The UCQ $\varphi(y)$, where $\varphi = \varphi_1 \vee \varphi_2$ with

$$\varphi_1 = \exists x \exists z \big(\mathrm{Person}(x, y, z) \wedge \mathrm{Profession}(x, \text{`computer scientist'}) \wedge \mathrm{City}(z, \text{`Athens'}, \text{`Greece'})\big).$$

and

$$\varphi_2 \;=\; \exists x \exists z \big(\text{Person}(x, y, z) \;\wedge\; \text{Profession}(x, \text{`computer scientist'}) \wedge \text{City}(z, \text{`Putú'}, \text{`Chile'})\big).$$

can be used to retrieve the list of names of computer scientists that were born in the city of Athens in Greece, or in the city of Putú in Chile.

Union of Conjunctive Queries as a Fragment of FO

By definition, UCQs use only relational atoms, conjunction ($\wedge$), disjunction ($\vee$), and existential quantification ($\exists$). Therefore, every UCQ can be expressed using formulae from the fragment of FO that corresponds to the closure of relational atoms under $\wedge$, $\vee$ and $\exists$; we refer to this fragment of FO as $\text{FO}^{\text{rel}}[\wedge, \vee, \exists]$. Interestingly, we can show that the converse is also true, which leads to the following expressive power result:

Theorem 30.3

The language of UCQs and the language of $\text{FO}^{\text{rel}}[\wedge, \vee, \exists]$ queries are equally expressive.

Proof. As discussed above, by definition, a UCQ is trivially an $\text{FO}^{\text{rel}}[\wedge, \vee, \exists]$ query. The interesting task is to show that an $\text{FO}^{\text{rel}}[\wedge, \vee, \exists]$ query $\varphi(\bar{x})$ can be equivalently expressed as a UCQ. This is done in three main steps:

- First, we propagate disjunction by using the following simple rules:

  $$\chi \wedge (\xi \vee \psi) \rightsquigarrow (\chi \wedge \xi) \vee (\chi \wedge \psi) \quad \text{and} \quad \exists x\, (\chi \vee \psi) \rightsquigarrow \exists x\, \chi \vee \exists x\, \psi,$$

  where χ, ξ and ψ are FO formulae. By applying the above rules, we can convert the formula φ into an equivalent formula of the form

  $$\varphi_1 \vee \cdots \vee \varphi_n$$

  for $n \geq 1$, where φ_i is a formula from $\text{FO}^{\text{rel}}[\wedge, \exists]$, for each $i \in [n]$.
- We then convert φ_i, for each $i \in [n]$, into a formula of the form $\exists \bar{x}_i\, \varphi_i'$, where φ_i' is a quantifier-free conjunction of relational atoms. This is done in the same way as the transformation of an $\text{FO}^{\text{rel}}[\wedge, \exists]$ query into a CQ (see Example 13.5): we first rename variables in order to ensure that bound variables do not repeat, and then push the existential quantifiers outside.
- After applying the above steps, we end up with a formula ψ of the form

  $$\psi_1 \vee \cdots \vee \psi_n$$

  where $\psi_i = \exists \bar{x}_1\, \varphi_i'$ and φ_i' is a quantifier-free conjunction of relational atoms, for each $i \in [n]$. Observe also that during the above two steps we

have not altered the set of free variable of φ, i.e., $\mathrm{FV}(\varphi) = \mathrm{FV}(\psi)$. Hence, $\psi(\bar{x})$ is a syntactically valid FO query that is equivalent to $\varphi(\bar{x})$. However, it should not be overlooked that $\psi(\bar{x})$ is not yet a UCQ since there is no guarantee that $\mathrm{FV}(\psi) = \mathrm{FV}(\psi_i)$, for each $i \in [n]$. In this final step, we explain how $\psi(\bar{x})$ can be converted into an equivalent UCQ.

Assume that we have access to a unary relation dom that stores all the values in the given database. In other words, we assume that every database D comes with a unary relation dom such that, for every $a \in \mathrm{Dom}(D)$, $\mathsf{dom}(a) \in D$. In this case, it is easy to see that $\psi'(\bar{x})$ with

$$\psi' \;=\; \bigvee_{i=1}^{n} \left(\psi_i \wedge \bigwedge_{y \in \mathrm{FV}(\psi) - \mathrm{FV}(\psi_i)} \mathsf{dom}(y) \right)$$

is a syntactically valid UCQ that is equivalent to $\psi(\bar{x})$. Indeed, $\mathrm{FV}(\psi) = \mathrm{FV}(\psi')$, each disjunct ψ'_i of ψ is such that $\mathrm{FV}(\psi') = \mathrm{FV}(\psi'_i)$, and $\psi'_i(\bar{x})$ is a CQ. Moreover, for every database D equipped with the unary relation dom, $\psi(\bar{x})(D) = \psi'(\bar{x})(D)$. It remains to show that the relational atoms of the form $\mathsf{dom}(\cdot)$ in ψ' can be eliminated with the help of disjunction.

Consider a formula of the form $\exists \bar{y}\, \chi \wedge \mathsf{dom}(u)$, where χ is an arbitrary formula over a schema $\mathbf{S}$. It is easy to verify that it is equivalent to

$$\bigvee_{R \in \mathbf{S}} \bigvee_{i \in \{1,\ldots,\mathrm{ar}(R)\}} \exists \bar{y}\, \exists z_1 \ldots \exists z_{\mathrm{ar}(R)-1}\, \chi \wedge R\big(z_1, \ldots, z_{i-1}, u, z_i, \ldots, z_{\mathrm{ar}(R)-1}\big)$$

where $z_1, \ldots, z_{i-1}, z_{i+1}, \ldots, z_{\mathrm{ar}(R)-1}$ are new variables not occurring in χ. Indeed, if a is a value that occurs in the input database, then a must occur in a tuple of some relation R at some position $i \in \{1, \ldots, \mathrm{ar}(R)\}$. Using this transformation, we can eventually eliminate all the relational atoms of the form $\mathsf{dom}(\cdot)$ in ψ', and obtain a formula $\psi'' = \psi''_1 \vee \cdots \vee \psi''_m$ such that $\psi'(\bar{x})$ and $\psi''(\bar{x})$ are equivalent queries, and $\psi''_i(\bar{x})$ is a syntactically valid CQ for each $i \in [m]$, which in turn implies that $\psi''(\bar{x})$ is a UCQ. □

Let $\mathrm{FO}[\wedge, \vee, \exists]$ be the fragment of FO that corresponds to the closure of relational atoms *and* equational atoms under $\wedge$, $\vee$ and $\exists$. This is known as the *existential positive* fragment of FO, and is typically denoted as $\exists\mathrm{FO}^+$; hence, from now on, by $\exists\mathrm{FO}^+$ we actually mean $\mathrm{FO}[\wedge, \vee, \exists]$. In other words, $\exists\mathrm{FO}^+$ is the fragment of FO obtained by explicitly adding equality to $\mathrm{FO}^{\mathrm{rel}}[\wedge, \vee, \exists]$. It is easy to show that the language of $\exists\mathrm{FO}^+$ queries is strictly more expressive than the language of UCQs, and thus, by Theorem 30.3, also the language of $\mathrm{FO}^{\mathrm{rel}}[\wedge, \vee, \exists]$ queries. Consider, for example, the $\exists\mathrm{FO}^+$ query $q = \varphi(x)$ with $\varphi = (x = a)$, where $a \in \mathsf{Const}$. Clearly, for every database D, $q(D) = \{(a)\}$ even if $a \notin \mathrm{Dom}(D)$. However, given a database D' such that $a \notin \mathrm{Dom}(D')$, for every 1-ary UCQ q', it holds that $q'(D') = \emptyset$ since the output of a CQ, and thus of a UCQ, on D' consists of tuples of constants from $\mathrm{Dom}(D')$.

Observe that the $\exists\text{FO}^+$ query q above uses equality among a variable and a constant. It turns out that this is crucial for showing that $\exists\text{FO}^+$ queries form a strictly more expressive language than UCQs. Interestingly, the language of queries based on $\text{FO}^{\text{rel,var}=}[\wedge, \vee, \exists]$, that is, the fragment of FO that corresponds to the closure of relational atoms and equational atoms of the form $x = y$, where both x and y are variables, under $\wedge$, $\vee$ and $\exists$, has the same expressive power as the language of UCQs.

Theorem 30.4

The language of UCQs and the language of $\text{FO}^{\text{rel,var}=}[\wedge, \vee, \exists]$ queries are equally expressive.

Proof. By definition, a UCQ is trivially an $\text{FO}^{\text{rel,var}=}[\wedge, \vee, \exists]$ query. It remains to show that an $\text{FO}^{\text{rel,var}=}[\wedge, \vee, \exists]$ query $\varphi(\bar{x})$ can be equivalently expressed as a UCQ. This is done by first observing that equational atoms that mention only variables can be eliminated by using atoms of the form $\mathsf{dom}(\cdot)$, where dom is a unary relation that stores all the values in the given database. Indeed, for each equational atom $x = y$ in φ, we replace y by x everywhere in φ and $\bar{x}$, and add the atom $\mathsf{dom}(x)$ to the conjunction. To see why the latter is needed, consider, for example, the query $\psi(x, y)$ with $\psi = (x = y)$. We cannot just throw away the equational atom; instead, this query is equivalent to $\psi'(x, x)$ with $\psi' = \mathsf{dom}(x)$. Hence, after the above transformation, we obtain an $\text{FO}^{\text{rel}}[\wedge, \vee, \exists]$ query $\varphi'(\bar{x}')$ that is equivalent to $\varphi(\bar{x})$ over databases equipped with the unary relation dom. Since, as discussed in the proof of Theorem 30.3, $\mathsf{dom}(\cdot)$ atoms can be eliminated with the help of disjunction, we can convert $\varphi'(\bar{x}')$ into an $\text{FO}^{\text{rel}}[\wedge, \vee, \exists]$ query $\varphi''(\bar{x}')$ that is equivalent to $\varphi(\bar{x})$ over *all* databases. Finally, by Theorem 30.3, we know that there exists a UCQ that is equivalent to $\varphi''(\bar{x}')$, and thus to $\varphi(\bar{x})$, and the claim follows. □

We have seen that UCQs are not powerful enough for expressing every $\exists\text{FO}^+$ query. We have also seen that the key reason for this is the fact that UCQs, although can express equality among variables, cannot express equality among variables and constants. The question that comes is whether the addition of equality among variables and constants to UCQs leads to a language that can express every $\exists\text{FO}^+$ query. A UCQ *with variable-constant equality* $\varphi(\bar{x})$ is defined as a UCQ with the only difference that a disjunct of φ can be a conjunction of relational atoms and equational atoms of the form $(x = a)$, where x is a variable and a is a constant. By using the same ideas as in the proofs of Theorems 30.3 and 30.4, it is easy to show the following:

Theorem 30.5

The language of UCQs with variable-constant equality and the language of $\exists\text{FO}^+$ queries are equally expressive.

It is important to stress that the transformations described in the proofs of the above expressive power results can be costly. Already, the transformation of an $\mathrm{FO}^{\mathrm{rel}}[\wedge,\vee,\exists]$ query into a UCQ may lead to an exponentially sized query. Consider, e.g., an $\mathrm{FO}^{\mathrm{rel}}[\wedge,\vee,\exists]$ query $\varphi(\bar{x})$, where φ is of the form

$$(\varphi_1 \vee \varphi_1') \wedge \cdots \wedge (\varphi_n \vee \varphi_n')$$

and $\varphi_i(\bar{x})$, $\varphi_i'(\bar{x})$ are CQs, for every $i \in [n]$. Representing $\varphi(\bar{x})$ as a UCQ requires transforming an FO formula in conjunctive normal form into an FO formula in disjunctive normal form, resulting in a UCQ consisting of 2^n CQs. Consequently, even though UCQs and $\mathrm{FO}^{\mathrm{rel}}[\wedge,\vee,\exists]$ (or $\mathrm{FO}^{\mathrm{rel,var}=}[\wedge,\vee,\exists]$) queries have the same expressive power, some problems related to them will have different complexity (whenever the size of the query matters). The same holds for UCQs with variable-constant equality and $\exists\mathrm{FO}^{+}$ queries.

Union of Conjunctive Queries as a Fragment of RA

We know, by Theorem 13.7, that the language of CQs has the same expressive power as the language of SPJ queries. Recall that SPJ is the fragment of RA that is built from base expressions $R \in \mathsf{Rel}$ (crucially, base expressions of the form $\{a\}$ with $a \in \mathsf{Const}$ are not included), and allows for selection, projection, and Cartesian product. Furthermore, conditions in selections are conjunctions of equalities. It should not come as a surprise the fact that by adding union to SPJ we get a fragment of RA, called *select-project-join-union* (SPJU), that has the same expressive power as UCQs.

Theorem 30.6

The language of UCQs and the language of SPJU queries are equally expressive.

Proof. The fact that every UCQ can be expressed as an SPJU query immediately follows from Theorem 13.7, which shows that every CQ can be expressed as an SPJ query. Indeed, a UCQ $q_1 \cup \cdots \cup q_n$ is equivalent to the SPJU query $e_1 \cup \cdots \cup e_n$, where e_i is an SPJ query that is equivalent to q_i, for each $i \in [n]$.

Consider now an SPJU k-ary query e. We proceed to show that e can be expressed as a UCQ. This is done in three main steps:

- First, we propagate union through other operations to become the outermost operation by applying the following simple rules:

$$\sigma_\theta(e_1 \cup e_2) \rightsquigarrow \sigma_\theta(e_1) \cup \sigma_\theta(e_2)$$
$$\pi_\alpha(e_1 \cup e_2) \rightsquigarrow \pi_\alpha(e_1) \cup \pi_\alpha(e_2)$$
$$e_1 \times (e_2 \cup e_3) \rightsquigarrow (e_1 \times e_2) \cup (e_1 \times e_3).$$

By applying the above rules, we get an SPJU query

$$e' \;=\; e_1 \cup \cdots \cup e_n$$

where e_i is an SPJ query, for each $i \in [n]$.

- Let $\varphi_i(x_i^1, \ldots, x_i^k)$ be the CQ that is equivalent to the SPJ query e_i, for each $i \in [n]$; such a CQ always exists due to Theorem 13.7. Let

$$\varphi \;=\; \varphi_1 \vee \cdots \vee \varphi_n.$$

- We finally convert φ into an FO formula ψ such that $\psi(z_1, \ldots, z_k)$, where $z_1, \ldots, z_k$ are distinct variables not occurring in φ, is an $\mathrm{FO}^{\mathrm{rel,var}=}[\wedge, \vee, \exists]$ query that is equivalent to the query e', and thus, to the query e. This suffices to show our claim since, by Theorem 30.4, we get that $\psi(z_1, \ldots, z_k)$ can be equivalently expressed as a UCQ.

  Consider an arbitrary disjunct φ_i of φ. Let $P_{\varphi_i} = \{P_1, \ldots, P_\ell\}$, where $\ell \leq k$ is the number of distinct variables occurring in $(x_i^1, \ldots, x_i^k)$, be the partition of the set of integers $[k]$ such that, for every $j, j' \in [k]$, j, j' belong to the same set of P_{φ_i} if and only if $x_i^j = x_i^{j'}$. For example, with $k = 5$ and $(x_i^1, \ldots, x_i^5) = (x, y, x, z, y)$, $P_{\varphi_i} = \{\{1, 3\}, \{2, 5\}, \{4\}\}$. Let ψ_i be the formula obtained from φ_i as follows: for every set $\{j_1, \ldots, j_m\} \in P_{\varphi_i}$, replace in φ_i the variable $x_i^{j_1}$ (note that $x_i^{j_1} = x_i^{j_2} = \cdots = x_i^{j_m}$) with the variable z_{j_1}, and add as a conjunct the conjunction of equational atoms

$$\bigwedge_{j \in \{j_2, \ldots, j_m\}} (z_{j_1} = z_j).$$

  It is easy to verify that $\psi(z_1, \ldots, z_k)$ with

$$\psi \;=\; \psi_1 \vee \cdots \vee \psi_n$$

  is an $\mathrm{FO}^{\mathrm{rel,var}=}[\wedge, \vee, \exists]$ query that is equivalent to e', as needed. □

The following is an immediate corollary of Theorems 30.3, 30.4 and 30.6.

Corollary 30.7

The language of $\mathrm{FO}^{\mathrm{rel}}[\wedge, \vee, \exists]$ (or even $\mathrm{FO}^{\mathrm{rel,var}=}[\wedge, \vee, \exists]$) queries and the language of SPJU queries are equally expressive.

It should be clear that the inability of SPJ queries to state base expressions of the form $\{a\}$ with $a \in \mathsf{Const}$ it is crucial for the validity of Theorem 30.6. Indeed, the addition of such base expressions to the SPJU fragment of RA leads to the strictly more expressive language of *positive relational algebra* (RA^+) queries. Interestingly, by providing a proof similar to that of Theorem 30.6, we can show that adding base expressions of the form $\{a\}$ with $a \in \mathsf{Const}$ to SPJU corresponds to the addition of variable-constant equality to UCQs.

Theorem 30.8

The language of UCQs with variable-constant equality and the language of RA^+ queries are equally expressive.

The following is an immediate corollary of Theorems 30.5 and 30.8 that relates the languages of $\exists\mathrm{FO}^+$ queries and RA^+ queries.

Corollary 30.9

The language of $\exists\mathrm{FO}^+$ queries and the language of RA^+ queries are equally expressive.

Note that the transformations described in the proofs of the above expressive power results (in particular, in the proof of Theorem 30.6) can be costly. For example, given an SPJU query of the form

$$(e_1 \cup e_1') \times \cdots \times (e_n \cup e_n')$$

where e_i, e_i' are SPJ queries, for each $i \in [n]$, after propagating the union during the first step of the transformation in the proof of Theorem 30.6, we get an SPJU query that is the union of 2^n SPJ queries. Therefore, even though UCQs and SPJU queries are equally expressive, some problems related to them will have different complexity (whenever the size of the query matters). The same holds for UCQs with variable-constant equality and RA^+ queries.

Preservation Under Homomorphisms

We have already seen that CQs are preserved under homomorphisms (Proposition 14.6). In other words, given a k-ary CQ $q = \varphi(\bar{x})$ over a schema $\mathbf{S}$, for every two databases D and D' of $\mathbf{S}$, and tuples $\bar{a} \in \mathrm{Dom}(D)^k$ and $\bar{b} \in \mathrm{Dom}(D')^k$,

$$(D, \bar{a}) \to_{\mathrm{Dom}(\varphi)} (D', \bar{b}) \text{ and } \bar{a} \in q(D) \text{ implies } \bar{b} \in q(D').$$

It is not difficult to show that preservation under homomorphisms extends to UCQs with variable-constant equality (and thus, to $\exists\mathrm{FO}^+$ queries).

Proposition 30.10

Every UCQ with variable-constant equality is preserved under homomorphisms.

Proof. Let $q = \varphi(\bar{x})$ be a k-ary UCQ with variable-constant equality over a schema $\mathbf{S}$. Assume that $\varphi = \varphi_1 \vee \cdots \vee \varphi_n$, and let q_i be the query $\varphi_i(\bar{x})$, for each $i \in [n]$; note that these queries are not CQs since they use equational atoms. Consider two databases D and D' of $\mathbf{S}$, and tuples $\bar{a} \in \mathrm{Dom}(D)^k$ and $\bar{b} \in \mathrm{Dom}(D')^k$ such that $(D, \bar{a}) \to_{\mathrm{Dom}(\varphi)} (D', \bar{b})$ and $\bar{a} \in q(D)$. Since $q(D) = \bigcup_{i=1}^{n} q_i(D)$, it is clear that $\bar{a} \in q_i(D)$ for some $i \in [n]$. By providing a proof similar to that of Proposition 14.6, which shows that CQs are preserved under homomorphisms, we can show that q_i is preserved under homomorphisms. Therefore, $\bar{b} \in q_i(D)$, which in turn implies that $\bar{b} \in q(D)$, as needed. □

It is far more remarkable, though, that the converse is true, that is, every FO query that is preserved under homomorphisms can be expressed as a UCQ with variable-constant equality.

Theorem 30.11

Consider an FO query q that is preserved under homomorphisms. There exists a UCQ with variable-constant equality that is equivalent to q.

This is a deep result whose proof is beyond the scope of this book, but it is very important and found many applications in the foundations of databases. An immediate corollary of Proposition 30.10 and Theorem 30.11 is that:

Corollary 30.12

The language of UCQs with variable-constant equality and the language of FO queries preserved under homomorphisms are equally expressive.

It is important to stress that many preservation results known in logic are true on arbitrary (finite or infinite) structures, but fail on finite structures. Preservation results of this kind can be transferred to the database setting only for possibly infinite databases, but not for (finite) databases, which is of course what is of interest to us. Remarkably, Theorem 30.11 is a rare exception that holds in the case of (finite) databases.

Query Evaluation

A general rule of thumb is that whatever is true about the evaluation of CQs, is true about the evaluation of their unions. We illustrate this by analyzing the combined complexity of evaluating UCQs and acyclic UCQs; concerning the data complexity, both problems are in DLogSpace due to Theorem 7.3.

Evaluation of UCQs

We first concentrate on UCQ-Evaluation. Recall that this is the problem of checking whether $\bar{a} \in q(D)$ for a UCQ q, a database D, and a tuple $\bar{a}$ over $\mathrm{Dom}(D)$. We show that indeed UCQ-Evaluation has the same (combined) complexity as CQ-Evaluation.

Theorem 30.13

UCQ-Evaluation is NP-complete.

Proof. It is clear that the NP-hardness is inherited from CQ-Evaluation, which we know is NP-hard (Theorem 15.1). We proceed to show the upper bound. Consider a UCQ $q(\bar{x})$ of the form $q_1 \cup \cdots \cup q_n$, a database D, and a tuple $\bar{a} \in \mathrm{Dom}(D)$. By Theorem 14.2, for $i \in [n]$, $\bar{a} \in q_i(D)$ if and only if $(q_i, \bar{x}) \to (D, \bar{a})$. Thus, we need to show that checking whether there exists an integer $i \in [n]$ and a homomorphism from $(q_i, \bar{x})$ to $(D, \bar{a})$ is in NP. This is done by guessing an integer $i \in [n]$ and a function $h : \mathrm{Dom}(A_{q_i}) \to \mathrm{Dom}(D)$, and then verifying that h is a homomorphism from $(A_{q_i}, \bar{x})$ to $(D, \bar{a})$, i.e., h is the identity on $\mathrm{Dom}(A_{q_i}) \cap \mathsf{Const}$, $R(\bar{u}) \in A_{q_i}$ implies $R(h(\bar{u})) \in D$, and $h(\bar{x}) = \bar{a}$. Since all the above steps are feasible in polynomial time, the claim follows. □

It is not difficult to show, by providing a proof similar to that of Theorem 30.13, that evaluating UCQs with variable-constant equality remains in NP. What is more interesting is the fact that evaluating $\exists\mathrm{FO}^+$ queries, as well as RA^+ queries, is also in NP. We have seen that every $\exists\mathrm{FO}^+$ can be converted into an equivalent UCQ with variable-constant equality (Theorem 30.5); the same holds for RA^+ queries (Theorem 30.8). However, the conversion can be very costly; it may take, in general, exponential time. Therefore, knowing that evaluating UCQs with variable-constant equality is in NP does not immediately imply that evaluating $\exists\mathrm{FO}^+$ and RA^+ queries is also in NP. Hence, one has to adopt a more refined procedure than simply converting the given $\exists\mathrm{FO}^+$ or RA^+ query into a UCQ with variable-constant equality (see Exercise 4.1).

Evaluation of Acyclic UCQs

We now consider the problem of evaluating *acyclic* UCQ*s*, that is, UCQs of the form $q_1 \cup \cdots \cup q_n$, where, for each $i \in [n]$, the CQ q_i is acyclic. Recall that a CQ is acyclic if its associated hypergraph is acyclic (Definition 20.4). We further know that acyclic CQs can be efficiently evaluated. More precisely, by Theorem 21.3, checking whether $\bar{a} \in q(D)$ for an acyclic CQ query q, a database D, and a tuple $\bar{a}$ over $\mathrm{Dom}(D)$ is feasible in time $O(\|D\| \cdot \log \|D\| \cdot \|q\|)$. It is easy to show that the same holds for acyclic UCQs.

Theorem 30.14

Consider an acyclic UCQ q, a database D, and a tuple $\bar{a}$ over $\mathrm{Dom}(D)$. Checking whether $\bar{a} \in q(D)$ is feasible in time $O(\|D\| \cdot \log \|D\| \cdot \|q\|)$.

Proof. We need to check whether $\bar{a} \in \bigcup_{i=1}^{n} q_i(D)$. By Theorem 21.3, for every $i \in [n]$, checking whether $\bar{a} \in q_i(D)$ is feasible in time $O(\|D\| \cdot \log \|D\| \cdot \|q_i\|)$. This implies that checking whether $\bar{a} \in \bigcup_{i=1}^{n} q_i(D)$ can be done in time

$$O\left(||D|| \cdot \log \|D\| \cdot \max_{i \in [n]}\{||q_i||\}\right).$$

Since $\max_{i \in [n]}\{||q_i||\} \leq ||q||$, the total time used is

$$O(\|D\| \cdot \log \|D\| \cdot \|q\|)$$

and the claim follows. □

31

Static Analysis of Unions of Conjunctive Queries

In this chapter, we study the containment and equivalence problems for unions of conjunctive queries, as well as the task of minimizing such queries.

Containment and Equivalence

We first focus on UCQ-Containment, the problem of deciding whether a UCQ q is contained in a UCQ q', that is, whether $q(D) \subseteq q'(D)$ for every database D. We show that it has the same complexity as CQ-Containment. But first we present a useful result that characterizes when q is contained in q' in terms of containment of the individual CQs occurring in q and q'.

Proposition 31.1

Consider two UCQs $q = q_1 \cup \cdots \cup q_n$ and $q' = q'_1 \cup \cdots \cup q'_m$. The following are equivalent:

1. $q \subseteq q'$.
2. For every $i \in [n]$, there exists $j \in [m]$ such that $q_i \subseteq q'_j$.

Proof. For showing that (1) implies (2), consider an arbitrary integer $i \in [n]$. We proceed to show that there exists $j \in [m]$ such that $q_i \subseteq q'_j$. Assume that $\bar{a} \in q_i(D)$ for some database D and tuple $\bar{a}$ over $\mathrm{Dom}(D)$. Thus, $\bar{a} \in q(D)$, and, by hypothesis, we get that $\bar{a} \in q'(D)$. The latter implies that there exists $j \in [m]$ such that $\bar{a} \in q'_j(D)$, which shows that $q_i \subseteq q'_j$, as needed.

For showing that (2) implies (1), assume that $\bar{a} \in q(D)$ for some database D and tuple $\bar{a}$ over $\mathrm{Dom}(D)$. Clearly, there exists $i \in [n]$ such that $\bar{a} \in q_i(D)$. By hypothesis, there exists $j \in [m]$ such that $q_i \subseteq q'_j$, and thus, $\bar{a} \in q'_j(D)$. The latter implies that $\bar{a} \in q'(D)$, which in turn shows that $q \subseteq q'$, as needed. □

We are now ready to pinpoint the complexity of UCQ-Containment.

Theorem 31.2

UCQ-Containment is NP-complete.

Proof. It is clear that the NP-hardness is inherited from CQ-Containment, which we know is NP-hard (Theorem 16.3). Consider now two UCQs $q = q_1 \cup \ldots \cup q_n$ and $q' = q'_1 \cup \cdots \cup q'_m$. We proceed to show that checking whether $q \subseteq q'$ is in NP. We assume that, for $i \in [n]$ and $j \in [m]$, the output tuple of q_i and q'_j is $\bar{x}$ and $\bar{x}'$, respectively. By the Homomorphism Theorem (Theorem 16.4) and Proposition 31.1, to check whether $q \subseteq q'$ it suffices to do the following:

- for each $i \in [n]$, guess an integer $j_i \in [m]$, and a function $h : \mathrm{Dom}(A_{q'_{j_i}}) \to A_{q_i}$, and
- for each $i \in [n]$, verify that h_i is a homomorphism from $(q'_{j_i}, \bar{x}')$ to $(q_i, \bar{x})$.

Since both steps are feasible in polynomial time, we conclude that deciding whether $q \subseteq q'$ is in NP, and the claim follows. □

An immediate corollary of Theorem 31.2 is that the equivalence problem for UCQs, that is, given two UCQs q, q', check whether $q \equiv q'$, is in NP since it boils down to two containment checks: $q \subseteq q'$ and $q' \subseteq q$. The NP-hardness is inherited from CQ-Equivalence (Theorem 16.8).

Corollary 31.3

UCQ-Equivalence is NP-complete.

Recall that query evaluation remains NP-complete even if we consider $\mathrm{FO}^{\mathrm{rel}}[\wedge, \vee, \exists]$ and SPJU queries (or even $\exists\mathrm{FO}^+$ and RA^+ queries), despite the fact that these languages allow us to express UCQs in a more succinct way. However, this is not true in the case of containment, where we can show that the complexity increases. We illustrate this for SPJU-Containment, that is, the problem of deciding whether an SPJU query is contained in another SPJU query. The treatment for $\mathrm{FO}^{\mathrm{rel}}[\wedge, \vee, \exists]$, and the more expressive languages $\exists\mathrm{FO}^+$ and RA^+, is similar and is left as an exercise (see Exercise 4.3).

We proceed to show that SPJU-Containment is Π_2^p-complete. This essentially tells us that, given two SPJU queries e and e', the problem of deciding whether $e \not\subseteq e'$ is in $\Sigma_2^p = \mathrm{NP}^{\mathrm{NP}}$, i.e., it can be solved via a nondeterminisitc algorithm that runs in polynomial time assuming that it has access to an oracle that can solve any problem in NP. In other words, the complement of SPJU-Containment is Σ_2^p-complete.[1] To show this we need some preparation.

We associate to an SPJU query e a set of SPJ queries, denoted $\mathrm{SPJ}(e)$. This is done by induction on the structure of e; essentially, for every union

[1] This is actually the first time in the book that we encounter the complexity classes Σ_2^p and Π_2^p, which contain NP. For further details see Appendix B.

$e_1 \cup e_2$ that occurs in e, we look at possible ways of resolving this union, i.e., choosing e_1 or e_2. The set $\mathrm{SPJ}(e)$ is formally defined as follows:

$$\mathrm{SPJ}(e) \;=\; \begin{cases} \{R\} & \text{if } e = R \\ \{\sigma_\theta(e'') \mid e'' \in \mathrm{SPJ}(e')\} & \text{if } e = \sigma_\theta(e') \\ \{\pi_\alpha(e'') \mid e'' \in \mathrm{SPJ}(e')\} & \text{if } e = \pi_\alpha(e') \\ \{e_1' \times e_2' \mid e_1' \in \mathrm{SPJ}(e_1) \text{ and } e_2' \in \mathrm{SPJ}(e_2)\} & \text{if } e = e_1 \times e_2 \\ \mathrm{SPJ}(e_1) \cup \mathrm{SPJ}(e_2) & \text{if } e = e_1 \cup e_2. \end{cases}$$

The following is easily shown by structural induction.

Proposition 31.4

Consider an SPJU query e, and assume that $\mathrm{SPJ}(e) = \{e_1, \ldots, e_n\}$. For every $i \in [n]$, e_i is an SPJ query, and $e \equiv e_1 \cup \cdots \cup e_n$.

It is clear that Proposition 31.4 provides an algorithm for solving the problem SPJU-Containment: given two SPJU queries e_1 and e_2, compute the sets $\mathrm{SPJ}(e_1)$ and $\mathrm{SPJ}(e_2)$, and then check whether, for every $e_1' \in \mathrm{SPJ}(e_1)$, there exists $e_2' \in \mathrm{SPJ}(e_2)$ such that $e_1' \subseteq e_2'$; the latter is essentially a containment check among two CQs, since an SPJ query can be easily converted into a CQ, which can be performed using the algorithm underlying Theorem 16.3. However, this is a very naive algorithm, which only shows that SPJU-Containment is in ExpTime. Indeed, it explicitly constructs the sets $\mathrm{SPJ}(e_1)$ and $\mathrm{SPJ}(e_2)$, which are in general of exponential size, and then performs exponentially many containment checks among SPJ queries. To establish the desired Π_2^p upper bound, we need to rely on a refined version of the above algorithm.

The key observation towards such a refined procedure is that finding a query $e' \in \mathrm{SPJ}(e)$, for an SPJU query e, amounts to "resolving" each union in e. In other words, having the parse tree T_e of the expression e, for every union node in T_e, with two subtrees under it, we keep only one of those subtrees, while the other one is replaced by $\bot$. The obtained tree T_e' is essentially the parse tree of an SPJ query from $\mathrm{SPJ}(e)$. Consider, for example, the query

$$e \;=\; (e_1 \cup e_2) \times (e_3 \cup (e_4 \cup e_5)),$$

where $e_1, \ldots, e_5$ are $\cup$-free. One way to resolve the union nodes in T_e is

$$(e_1 \cup \bot) \times (e_3 \cup \bot),$$

which leads to $e_1 \times e_3 \in \mathrm{SPJ}(e)$. Another way is

$$(\bot \cup e_2) \times (\bot \cup (\bot \cup e_5)),$$

which leads to $e_2 \times e_3 \in \text{SPJ}(e)$. If union occurs k times in e, this gives us 2^k expressions that can result from resolving those union nodes, as each one gives us two choices. From the above discussion, it is clear that we can guess a way to resolve the union nodes in T_e in polynomial time, or, in other words, we can guess an SPJ query from $\text{SPJ}(e)$ in polynomial time. This fact allows us to devise the refined procedure for SPJU-Containment.

Before doing this, we need to establish an intermediate complexity result, which will be crucial in the complexity analysis of this refined procedure. Furthermore, it illustrates the fact that restricting the language of the left-hand side query in the containment check has an impact on the complexity.

Proposition 31.5

Consider an SPJ query e, and an SPJU query e'. The problem of deciding whether $e \subseteq e'$ is in NP.

Proof. By Proposition 31.4, it suffices to show that the problem of checking whether there exists an SPJ query $e'' \in \text{SPJ}(e')$ such that $e \subseteq e''$ is in NP. This is done by guessing an SPJ query e'' from $\text{SPJ}(e')$, and a mapping $h : \text{Dom}(A_{q_{e''}}) \to \text{Dom}(A_{q_e})$, where $q_e(\bar{x})$ and $q_{e''}(\bar{y})$ are the CQs obtained after converting e and e'' into CQs, which is always possible due to Theorem 13.7, and then verifying that h is a homomorphism from $(q_{e''}, \bar{y})$ to $(q_e, \bar{x})$. The correctness of this procedure is guaranteed by the Homomorphism Theorem (Theorem 16.4). Since both steps are feasible in polynomial time (recall that an SPJ query from $\text{SPJ}(e)$ can be guessed in polynomial time), we conclude that checking whether $e \subseteq e'$ is in NP, and the claim follows. □

Proposition 31.5 essentially tells us that SPJU-Containment remains NP-complete whenever the left-hand side query in the containment check does not use union. However, as already mentioned, the complexity increases when considering the problem in its general form without any assumptions.

Theorem 31.6

SPJU-Containment is Π_2^p-complete.

Proof. Consider two SPJU queries e and e'. We proceed to show that checking whether $e \not\subseteq e'$ is in $\Sigma_2^p = \text{NP}^{\text{NP}}$, which in turn implies that SPJU-Containment is in Π_2^p. By Proposition 31.4, it suffices to show that the problem of checking whether there exists an SPJ query $\hat{e} \in \text{SPJ}(e)$ such that $\hat{e} \not\subseteq e'$ is in NP^{NP}. This is done by simply guessing an SPJ query $\hat{e}$ from $\text{SPJ}(e)$, and verifying that $\hat{e} \not\subseteq e'$. It is clear that the "guess" step can be performed in polynomial time. Concerning the "verify" step, by Proposition 31.5, it can be performed in constant time assuming that we have access to an oracle that can solve any problem in NP. In particular, the oracle takes as input the queries $\hat{e}$ and e',

and does the following: if $\hat{e} \subseteq e'$, then return `false`; otherwise, return `true`. Therefore, checking whether $e \not\subseteq e'$ is in $\mathrm{NP}^{\mathrm{NP}}$, as needed.

To prove the Π_2^p-hardness one can provide a polynomial-time reduction from the problem $\forall\exists$QSAT (Exercise 4.2). □

An immediate corollary of Theorem 31.6 is that the equivalence problem for SPJU queries, that is, given two SPJU queries e, e', check whether $e \equiv e'$, is in Π_2^p since it boils down to two containment checks: $e \subseteq e'$ and $e' \subseteq e$. The Π_2^p-hardness is shown via an easy reduction from SPJU-Containment. Given two SPJU queries e, e' of arity k, $e \subseteq q'$ iff $\pi_{(1,\ldots,k)}(e \bowtie_\theta e') \equiv e$, where

$$\theta \;=\; (1 \doteq k+1) \wedge (2 \doteq k+2) \wedge \cdots \wedge (k \doteq 2k).$$

We therefore conclude that:

Corollary 31.7

SPJU-Equivalence is Π_2^p-complete.

Minimization

In Chapter 17, we studied the notion of minimization of CQs, which aims to provide equivalent CQs that are also minimal. More precisely, given a CQ q over a schema $\mathbf{S}$, a CQ q' over $\mathbf{S}$ is a minimization of q if $q \equiv q'$, and for every CQ q'' over $\mathbf{S}$, $q' \equiv q''$ implies $|A_{q'}| \leq |A_{q''}|$. We have also seen how the minimization of a CQ, which is unique (up to variable renaming), can be computed by simply removing atoms from its body. We proceed to discuss how the ideas developed around CQ minimization can be extended to UCQs. We start by defining the notion of minimization for UCQs.

Definition 31.8: Minimization of UCQs

Consider a UCQ $q = q_1 \cup \cdots \cup q_n$ over a schema $\mathbf{S}$. A UCQ $q' = q'_1 \cup \cdots \cup q'_m$, for $m \leq n$, over $\mathbf{S}$ is a *minimization* of q if the following hold:

1. $q \equiv q'$,
2. for every $i \in [m]$, and CQ p over $\mathbf{S}$, $q'_i \equiv p$ implies $|A_{q'_i}| \leq |A_p|$, and
3. for every UCQ $q'' = q''_1 \cup \cdots \cup q''_\ell$, $q' \equiv q''$ implies $m \leq \ell$.

In simple words, q' is a minimization of q if it is equivalent to q, each CQ of q' has the smallest number of atoms among all the CQs that are equivalent to it, and q' has the smallest number of CQs among all the UCQs that are equivalent to it. We proceed to show that minimizations of a UCQ q can be found by simply removing atoms from the body of its CQs (i.e., by computing a core of its CQs), as well as removing CQs from it. Moreover, although q may have several minimizations, they are all the same (up to variable renaming).

Minimization via Atom and CQ Removals

We start be defining the notion of core for UCQs, which is a generalization of the notion of core for CQs given in Definition 17.2.

Definition 31.9: Core of a UCQ

Consider a UCQ $q(\bar{x}) = q_1 \cup \cdots \cup q_n$. A UCQ $q'(\bar{x}) = q'_1 \cup \cdots \cup q'_m$ is a *core* of q if the following hold:

1. $m \leq n$, and there is a set of integers $\{i_1, \ldots, i_m\} \subseteq [n]$ such that, for every $j \in [m]$, q'_j is a core of q_{i_j},
2. for every $i \in [n]$, there exists $j \in [m]$ such that $q_i \subseteq q'_j$, and
3. for every $i \in [m]$, there is no $j \in [m] - \{i\}$ such that $q'_i \subseteq q'_j$.

The first condition in Definition 31.9 states that q' can be obtained from q by eliminating some of its CQs, and then computing a core of the remaining CQs, the second condition ensures that $q \equiv q'$, and the third condition states that q' is minimal with respect to the number of CQs occurring in it. We show that the notion of core captures our intention of constructing a minimization.

Proposition 31.10

Every UCQ q has at least one core, and every core of q is a minimization of q.

Proof. We first show that we can always construct a core of q. Assuming that $q = q_1 \cup \cdots \cup q_n$, we first replace q_i with a core of it, for each $i \in [n]$, which we know always exists by Proposition 17.4, and get a UCQ $q' = q'_i \cup \cdots \cup q'_n$. If q' is a core of itself, then the claim follows. Assume now that this is not the case. This means that condition (3) in the definition of core (Definition 31.9) is violated, which in turn implies that there exists $i \in [n]$ such that $q'_i \subseteq q'_j$ for some $j \in [n] - \{i\}$. If q'' obtained from q' by removing the CQ q'_i is a core of itself, then it is clear that q'' is a core of q. Otherwise, we iteratively apply the above argument until we reach a core of q.

We now proceed to show that a core of $q(\bar{x}) = q_1 \cup \cdots \cup q_n$ is a minimization of it. Towards a contradiction, assume that $q'(\bar{x}) = q'_1 \cup \cdots \cup q'_m$ is a core of q but not a minimization of q. This implies that one of the following holds:

1. there exists $i \in [n]$ and a CQ p such that $q'_i \equiv p$ and $|A_p| < |A_{q'_i}|$, or
2. there exists a UCQ $q'' = q''_1 \cup \cdots \cup q''_\ell$ such that $q' \equiv q''$ and $\ell < m$.

Assuming (1) holds, by Proposition 31.1, we can conclude that there exists a CQ $p'(\bar{x})$ such that $(q'_i, \bar{x}) \to (p', \bar{x})$ and $A_{p'} \subseteq A_{q'_i}$. This contradicts our hypothesis that q' is a core of q.

Algorithm 12 ComputeCoreUCQ(q)

Input: A UCQ $q(\bar{x}) = q_1 \cup \ldots \cup q_n$
Output: A UCQ $q^*(\bar{x})$ that is a core of $q(\bar{x})$

1: $Q := \emptyset$
2: **for** $i = 1$ **to** n **do**
3: $\quad Q := Q \cup \{\text{ComputeCore}(q_i)\}$
4: **while** there are distinct CQs $q', q'' \in Q$ such that $q' \subseteq q''$ **do**
5: $\quad Q := Q - \{q'\}$
6: **return** $q^* = \bigcup_{q' \in Q} q'$

Assume now (2). Since $q' \subseteq q''$, by Proposition 31.1 we get that, for every $i \in [m]$, there exists $j \in [\ell]$ such that $q'_i \subseteq q''_j$. Since $\ell < m$, we conclude that there exists $i, i' \in [m]$, with $i \neq i'$, and $j^* \in [\ell]$ such that $q'_i \subseteq q''_{j^*}$ and $q'_{i'} \subseteq q''_{j^*}$. Now, since $q'' \subseteq q'$, by Proposition 31.1 we get that, for each $j \in [\ell]$, there exists $i \in [m]$ such that $q''_j \subseteq q'_i$. Therefore, there exists $i^* \in [m]$ such that $q''_{j^*} \subseteq q'_{i^*}$. This implies that $q'_i \subseteq q'_{i^*}$ and $q'_{i'} \subseteq q'_{i^*}$. Consequently, there exist $i \in [m]$ and $j \in [m] - \{i\}$ such that $q'_i \subseteq q'_j$, which again contradicts our hypothesis that q' is a core of q, and the claim follows. □

By Proposition 31.10, computing a minimization of a UCQ boils down to computing a core of it. This can be done by applying the iterative procedure ComputeCoreUCQ, given in Algorithm 12, which in turn relies on ComputeCore, given in Algorithm 4, that computes the core of a CQ. It is clear that, for a UCQ q, ComputeCoreUCQ(q) terminates after finitely many steps. It is also easy to verify that ComputeCoreUCQ is correct.

Lemma 31.11. *Given a* UCQ q, ComputeCoreUCQ(q) *is a core of* q.

Let is clarify that ComputeCoreUCQ is a nondeterministic algorithm since the procedure ComputeCore is nondeterministic. Moreover, there may be several CQs satisfying the condition of the while loop (in particular, there may be several CQs that must be removed from the set Q), but we do not specify how such a CQ is selected. Actually, the CQ q' of Q that is eventually removed from Q at step 5 is chosen nondeterministically. Therefore, the final result computed by the algorithm depends on the computation of ComputeCore at step 3, as well as how the CQs to be removed from Q are chosen at step 5. Consequently, different executions of ComputeCore(q) may compute cores of q that are syntactically different. However, as we discuss next, different minimizations of a UCQ q are isomorphic, which in turn implies, due to Proposition 31.10, that different cores of q are isomorphic.

Uniqueness of Minimization

We say that two UCQs $q(\bar{x}) = q_1 \cup \cdots \cup q_n$ and $q'(\bar{x}') = q'_1 \cup \cdots \cup q'_m$ are *isomorphic* if one can be turned into the other via renaming of variables, i.e.,

there is a bijection $\delta : \{q_1, \ldots, q_n\} \to \{q'_1, \ldots, q'_m\}$, which means that $n = m$, such that, for every $i \in [n]$, q_i and $\delta(q_i)$ are isomorphic.

Proposition 31.12

Consider a UCQ $q(\bar{x})$, and let $q'(\bar{x}')$ and $q''(\bar{x}'')$ be minimizations of q. Then q' and q'' are isomorphic.

Proof. Assume that $q' = q'_1 \cup \cdots \cup q'_n$ and $q'' = q''_1 \cup \cdots \cup q''_n$. We need to show that there is a bijection $\delta : \{q'_1, \ldots, q'_n\} \to \{q''_1, \ldots, q''_n\}$ such that, for every $i \in [n]$, q'_i and $\delta(q'_i)$ are isomorphic. We first show an auxiliary lemma:

Lemma 31.13. *There is a bijection $\tau : [n] \to [n]$ such that, for each $i \in [n]$, $q'_i \equiv q''_{\tau(i)}$, and there is no integer $j \in [n] - \{\tau(i)\}$ such that $q'_i \equiv q''_j$.*

Proof. To prove the claim, it suffices to show that, for each $i \in [n]$:

1. there exists $j \in [n]$ such that $q'_i \equiv q''_j$, and
2. for each $j, k \in [n]$, $q'_i \equiv q''_j$ and $q'_i \equiv q''_k$ implies $j = k$.

We first show claim (1). Since q and q' are minimizations of q, we conclude that $q \equiv q'$ and $q \equiv q''$, and thus, $q' \equiv q''$. By Proposition 31.1, we get that, for each $i \in [n]$, there exists $j \in [n]$ such that $q'_i \subseteq q''_j$. But, again by Proposition 31.1, there exists $k \in [n]$ such that $q''_j \subseteq q'_k$. Necessarily, $q'_i = q'_k$; otherwise, q' is equivalent to the UCQ obtained from q' after eliminating q'_i (since $q'_i \subseteq q'_k$), which contradicts the fact that q' is a minimization of q.

We now prove claim (2). Towards a contradiction, assume that there exists $i \in [n]$, and distinct integers $j, k \in [n]$ such that $q'_i \equiv q''_j$ and $q'_i \equiv q''_k$. This implies that the UCQ obtained from q'' after eliminating one of the CQs q''_j or q''_k (since $q''_j \equiv q''_k$) is equivalent to q'', which contradicts the fact that q'' is a minimization of q. This completes the proof of Lemma 31.13. □

Having the bijection $\tau : [n] \to [n]$ provided by Lemma 31.13, we define the bijection $\delta : \{q'_1, \ldots, q'_n\} \to \{q''_1, \ldots, q''_n\}$ as follows: for $i \in [n]$, $\delta(q'_i) = q''_{\tau(i)}$. It remains to argue that, for every $i \in [n]$, q'_i and $\delta(q'_i)$ are isomorphic. It is clear, by the definition of τ, that $q'_i \equiv \delta(q'_i)$. Moreover, since q', q'' are minimizations of q, we conclude that the CQs q'_i and $\delta(q'_i)$ are minimizations of some CQ (for example, of q'_i or $\delta(q'_i)$). Therefore, by Proposition 17.7, we get that q'_i and $\delta(q'_i)$ are isomorphic, and the claim follows. □

From Proposition 31.10, which tells us that a core of a UCQ q is a minimization of q, and Proposition 31.12, we get the following corollary:

Corollary 31.14

Consider a UCQ q, and let q' and q'' be cores of q. It holds that q' and q'' are isomorphic.

Recall that different executions of the nondeterministic procedure ComputeCoreUCQ on some UCQ q, may compute cores of q that are syntactically different. However, Corollary 31.14 tells us that those cores differ only on the names of their variables. In other words, cores of q computed by different executions of ComputeCoreUCQ(q) are the same up to variable renaming.

32

Unions of Conjunctive Queries with Inequalities

It is not difficult to show that UCQs, or even UCQs with variable constant-equality that are equally expressive to $\exists\text{FO}^+$ queries, are not powerful enough for expressing simple queries that involve negation such as the query

$$q \;=\; \exists x \exists y \, (\text{Edge}(x, y) \wedge \neg(x = y)),$$

which essentially asks whether a graph has an edge (v, u) that is not a loop, i.e., v and u are different nodes. Observe that q is not preserved under homomorphisms: for $D = \{R(a, b)\}$ and $D' = \{R(c, c)\}$, we have that $D \to_\emptyset D'$, but $D \models q$ while $D' \not\models q$. On the other hand, we know by Proposition 30.10 that UCQs (even with variable-constant equality) are preserved under homomorphisms, which immediately implies that q cannot be expressed as a UCQ (even with variable-constant equality).

This raises the question whether we can add negation to UCQs without increasing the complexity of query evaluation, and, more importantly, without losing the decidability of containment and equivalence. Of course, by adding arbitrary negation to UCQs, which essentially means that we add negation to $\text{FO}^{\text{rel}}[\wedge, \vee, \exists]$ queries, we obtain a language for which query evaluation is PSPACE-hard, while containment and equivalence are undecidable. Indeed, the PSPACE-hardness proof of Theorem 7.1, as well as the undecidability proof of Theorem 8.3, hold even for $\text{FO}^{\text{rel}}[\wedge, \vee, \exists]$ queries. On the other hand, there are examples of languages obtained by adding a tamed negation to UCQs, and still enjoy the good computational properties of UCQs. The best known such example is the language of UCQ*s with inequality*, that is, UCQs that can also use expressions of the form $\neg(v = u)$ as the query q given above. In this chapter, we introduce the language of UCQs with inequality, and study the main computational problems: evaluation and containment.

Conjunctive Queries with Inequality

Before we introduce and study UCQs with inequality, it is important to study and understand first CQs with inequalities. Note that in the rest of the chapter we write $v \neq u$ as an abbreviation for $\neg(v = u)$.

Syntax of Conjunctive Queries with Inequality

We start with the syntax of conjunctive queries with inequalities.

Definition 32.1: Syntax of CQs with Inequality

A *conjunctive query with inequality* ($\text{CQ}^{\neq}$) over a schema $\mathbf{S}$ is an FO query $\varphi(\bar{x})$ over $\mathbf{S}$ where φ is a formula of the form

$$\exists \bar{y} \left(R_1(\bar{u}_1) \wedge \cdots \wedge R_n(\bar{u}_n) \wedge v_1 \neq v'_1 \wedge \cdots \wedge v_m \neq v'_m \right)$$

for $n \geq 1$ and $m \geq 0$, where

- for each $i \in [n]$, $R_i(\bar{u}_i)$ is a relational atom, and $\bar{u}_i$ a tuple of constants and variables mentioned in $\bar{x}$ and $\bar{y}$, and
- for each $i \in [m]$, v_i is a variable mentioned in $\bar{u}_k$ for some $k \in [n]$, and v'_i is a variable mentioned in $\bar{u}_k$ for some $k \in [n]$ or a constant.

Note that the second item of the definition requires that each variable that participates in at least one inequality appears also in at least one relational atom. This is a common assumption in CQs with inequality that, as we discuss below, allows us to show that homomorphisms provide an alternative way to describe the evaluation of CQs with inequality in the same way as for CQs.

It is common to represent CQs with inequality via a rule-like syntax. In particular, the $\text{CQ}^{\neq}$ $\varphi(\bar{x})$ given in Definition 13.1 can be written as the *rule*

$$\text{Answer}(\bar{x}) \ \text{:-} \ R_1(\bar{u}_1), \ldots, R_n(\bar{u}_n), v_1 \neq v'_1, \ldots, v_m \neq v'_m \,,$$

where Answer is a relation name not in $\mathbf{S}$, and its arity (under the singleton schema $\{\text{Answer}\}$) is equal to the arity of q. The relational atom $\text{Answer}(\bar{x})$ that appears on the left of the :- symbol is the *head* of the rule, while the expression that appears on the right of the :- symbol is the *body* of the rule. In general, we use the rule-like syntax for CQs. Nevertheless, for convenience, we will freely interpret a CQ as an FO query or as a rule.

Semantics of Conjunctive Queries with Inequality

Since a $\text{CQ}^{\neq}$ is an FO query, the definition of its output on a database can be inherited from Definition 3.6. More precisely, given a database D of a schema $\mathbf{S}$, and a k-ary $\text{CQ}^{\neq}$ $q = \varphi(\bar{x})$ over $\mathbf{S}$, where $k \geq 0$, the *output* of q on D is

$$q(D) \;=\; \{\bar{a} \in \mathrm{Dom}(D)^k \mid D \models \varphi(\bar{a})\}\,.$$

Recall that every variable that occurs in an inequality of q it also occurs in a relational atom of q. For this reason, the output of q only consists of tuples of constants from $\mathrm{Dom}(D)$. It is easy to verify that if we drop this condition, then the output may mention constants that occur in the query but not in the database. Consider, for example, the FO query $q = \varphi(x)$ with

$$\varphi \;=\; \exists x\,(R(x) \wedge x \neq y \wedge a \neq b),$$

where x, y are variables and a, b are constants, which is not a CQ with inequality since y does not occur in a relational atom. Clearly, for $D = \{R(a)\}$, $q(D) = \{(b)\}$, while the constant b does not belong to $\mathrm{Dom}(D)$.

As for plain CQs, there is a more intuitive (and equivalent) way of defining the semantics of CQs with inequality when they are viewed as rules. The body of a $\mathrm{CQ}^{\neq}$ q of the form $\mathrm{Answer}(\bar{x})$:– body can be seen as a pattern that must be matched with the database D via an assignment η that maps the variables in q to $\mathrm{Dom}(D)$. For each such assignment η, if η applied to this pattern produces only facts of D, and at the same time respects all the inequalities, it means that the pattern matches with D via η, and the tuple $\eta(\bar{x})$ is an output of q on D. We proceed to formalize this informal description.

Consider a database D and a $\mathrm{CQ}^{\neq}$ q of the form

$$\mathrm{Answer}(\bar{x}) \;\text{:–}\; R_1(\bar{u}_1), \ldots, R_n(\bar{u}_n), v_1 \neq v_1', \ldots, v_m \neq v_m'\,.$$

An *assignment* for q over D is a function η from the set of variables in q to $\mathrm{Dom}(D)$. We say that η is *consistent* with D if

$$R_i(\eta(\bar{u}_i)) \in D \qquad \text{and} \qquad \eta(v_j) \neq \eta(v_j')$$

for each $i \in [n]$ and $j \in [m]$, where the fact $R_i(\eta(\bar{u}_i))$ is obtained by replacing each variable x in $\bar{u}_i$ with $\eta(x)$, and leaving the constants in $\bar{u}_i$ untouched. The consistency of η with D essentially means that the body of q matches with D via η. We can now define what is the output of a $\mathrm{CQ}^{\neq}$ on a database.

Definition 32.2: Evaluation of CQs with Inequality

Given a database D of a schema $\mathbf{S}$, and a $\mathrm{CQ}^{\neq}$ $q(\bar{x})$ over $\mathbf{S}$, the *output* of q on D is defined as the set of tuples

$$q(D) \;=\; \{\eta(\bar{x}) \mid \eta \text{ is an assignment for } q \text{ over } D \text{ consistent with } D\}\,.$$

It is an easy exercise to show that the semantics of $\mathrm{CQ}^{\neq}$ inherited from the semantics of FO queries in Definition 3.6, and the semantics of CQs given in Definition 32.2, are equivalent, i.e., for a CQ $q = \varphi(\bar{x})$ and a database D,

$$\{\bar{a} \in \mathrm{Dom}(D)^k \mid D \models \varphi(\bar{a})\} = \{\eta(\bar{x}) \mid \eta \text{ is an assignment for } q \text{ over } D \text{ consistent with } D\}\,.$$

Evaluation and Homomorphisms

We proceed to discuss how homomorphisms emerge in the context of CQs with inequality. In particular, we show that they provide an alternative way to describe the evaluation of CQs with inequality. Given a $\text{CQ}^{\neq}$ q of the form

$$\text{Answer}(\bar{x}) \ \coloneq R_1(\bar{u}_1), \ldots, R_n(\bar{u}_n), v_1 \neq v'_1, \ldots, v_m \neq v'_m \,,$$

we define the sets of relational atoms and inequalities

$$A_q^+ \ = \ \{R_1(\bar{u}_1), \ldots, R_n(\bar{u}_n)\} \qquad \text{and} \qquad A_q^- \ = \ \{v_1 \neq v'_1, \ldots, v_m \neq v'_m\},$$

respectively. As usual, $\text{Dom}(A_q^+)$ collects all the variables and constants occurring in the relational atoms of A_q^+. We further write $\text{Dom}(A_q^-)$ for the set of constants occurring in A_q^-. Notice that there may be constants in $\text{Dom}(A_q^-)$ that do not occur in $\text{Dom}(A_q^+)$ since, according to Definition 32.1, only the variables (not the constants) that occur in an inequality must also occur in a relational atom. Having the above sets in place, we can naturally talk about homomorphisms from CQs with inequality to databases.

Definition 32.3: Homomorphisms from $\text{CQ}^{\neq}$ to Databases

Consider a $\text{CQ}^{\neq}$ $q(\bar{x})$ over a schema $\mathbf{S}$, and a database D of $\mathbf{S}$. We say that there is a *homomorphism from q to D*, written as $q \to D$, if there exists a function $h : \text{Dom}(A_q^+) \cup \text{Dom}(A_q^-) \to \text{Dom}(D) \cup \text{Dom}(A_q^-)$ that is a homomorphism from A_q^+ to D, is the identity on $\text{Dom}(A_q^-)$, and, for every $v \neq u \in A_q^-$, $h(v) \neq h(u)$. We say that there is a *homomorphism from $(q, \bar{x})$ to $(D, \bar{a})$*, written as $(q, \bar{x}) \to (D, \bar{a})$, if $h(\bar{x}) = \bar{a}$.

To define the output of a $\text{CQ}^{\neq}$ $q(\bar{x})$ on a database D (see Definition 32.2), we used the notion of assignment for q over D, which is a function from the set of variables in q to $\text{Dom}(D)$. The output of q on D consists of all the tuples $\eta(\bar{x})$, where η is an assignment for q over D that is consistent with D, i.e.,

$$R_i(\eta(\bar{u}_i)) \in D \qquad \text{and} \qquad \eta(v_j) \neq \eta(v'_j)$$

for each $i \in [n]$ and $j \in [m]$. Since, for $i \in [n]$, $R_i(\eta(\bar{u}_i))$ is the fact obtained after replacing each variable x in $\bar{u}_i$ with $\eta(x)$, and leaving the constants in $\bar{u}_i$ untouched, such an assignment η corresponds to a function $h : \text{Dom}(A_q^+) \cup \text{Dom}(A_q^-) \to \text{Dom}(D) \cup \text{Dom}(A_q^-)$, which is the identity on the set of constants occurring in q, such that $R(h(\bar{u}_i)) = R(\eta(\bar{u}_i))$. But, of course, this is the same as saying that h is a homomorphism from q to D. Therefore, $q(D)$ is the set of all tuples $h(\bar{x})$, where h is a homomorphism from q to D, i.e., the set of all tuples $\bar{a}$ over $\text{Dom}(D)$ with $(q, \bar{x}) \to (D, \bar{a})$. This leads to an alternative characterization of $\text{CQ}^{\neq}$ evaluation in terms of homomorphisms.

Theorem 32.4

For a database D of a schema $\mathbf{S}$, and a $\text{CQ}^{\neq}$ $q(\bar{x})$ of arity $k \geq 0$ over $\mathbf{S}$,

$$q(D) \;=\; \{\bar{a} \in \text{Dom}(D)^k \mid (q, \bar{x}) \to (D, \bar{a})\}.$$

Unlike plain CQs, CQs with inequality are not preserved under homomorphisms. This has been already illustrated at the beginning of the chapter via the $\text{CQ}^{\neq}$ Answer :– Edge$(x, y), x \neq y$, which asks for the existence of an edge in a graph that is not a loop. On the other hand, we can easily show that CQs with inequality remain monotone, i.e., given a $\text{CQ}^{\neq}$ q over a schema $\mathbf{S}$, and two databases D and D' of $\mathbf{S}$, $D \subseteq D'$ implies $q(D) \subseteq q(D')$.

Proposition 32.5

Every $\text{CQ}^{\neq}$ is monotone.

Proof. Let $q(\bar{x})$ be a $\text{CQ}^{\neq}$ over a schema $\mathbf{S}$. Consider the databases D, D' of $\mathbf{S}$ such that $D \subseteq D'$, and assume that $\bar{a} \in q(D)$. By Theorem 32.4, we get that $(q, \bar{x}) \to (D, \bar{a})$. Since $D \subseteq D'$, we immediately get that $(q, \bar{x}) \to (D', \bar{a})$, and thus, again by Theorem 32.4, $\bar{a} \in q(D')$, as needed. □

The fact that CQs with inequality are monotone allows us to clarify the expressiveness boundaries of $\text{CQ}^{\neq}$. In particular, we can show that the negation allowed in $\text{CQ}^{\neq}$ is indeed quite restricted. Note that the following arguments have been already used in Chapter 14 to show that CQs cannot express negative relational atoms and difference.

$\text{CQ}^{\neq}$ cannot express negative relational atoms. The reason is because such queries are not monotone. Consider, for example, the FO query

$$q \;=\; \neg P(a),$$

where a is a constant. If we take $D = \emptyset$ and $D' = \{P(a)\}$, then $D \subseteq D'$ but $D \models q$ while $D' \not\models q$.

$\text{CQ}^{\neq}$ cannot express difference. This is because difference is not monotone. Consider, for example, the FO query

$$q \;=\; \exists x(P(x) \wedge \neg Q(x)).$$

For $D = \{P(a)\} \subseteq D' = \{P(a), Q(a)\}$, we have that $D \models q$ while $D' \not\models q$.

Query Evaluation

We now proceed to study the complexity of $\mathsf{CQ}^{\neq}$-Evaluation, that is, the problem of checking whether $\bar{a} \in q(D)$ for a $\mathrm{CQ}^{\neq}$ query q, a database D, and a tuple $\bar{a}$ over $\mathrm{Dom}(D)$. We actually show that it has the same (combined) complexity as CQ-Equivalence; concerning the data complexity, it is clear that $\mathsf{CQ}^{\neq}$-Evaluation is in DLogSpace due to Theorem 7.3.

Theorem 32.6

$\mathsf{CQ}^{\neq}$-Evaluation is NP-complete.

Proof. It is clear that the NP-hardness is inherited from CQ-Evaluation, which we know is NP-hard (Theorem 15.1). We proceed to show the upper bound. Consider a $\mathrm{CQ}^{\neq}$ $q(\bar{x})$, a database D, and a tuple $\bar{a} \in \mathrm{Dom}(D)$. By Theorem 32.4, $\bar{a} \in q(D)$ if and only if $(q, \bar{x}) \to (D, \bar{a})$. Therefore, we need to show that checking whether there exists a homomorphism from $(q, \bar{x})$ to $(D, \bar{a})$ is in NP. This is done by guessing a function $h : \mathrm{Dom}(A_q^+) \cup \mathrm{Dom}(A_q^-) \to \mathrm{Dom}(D) \cup \mathrm{Dom}(A_q^-)$ that is the identity on constants, and then verifying that (i) h is a homomorphism from $(A_q^+, \bar{x})$ to $(D, \bar{a})$, and (ii) for every $v \neq u \in A_q^-$, $h(v) \neq h(u)$. Since both steps are feasible in polynomial time, we conclude that checking whether $(q, \bar{x}) \to (D, \bar{a})$ is in NP, and the claim follows. □

Containment

We now focus on $\mathsf{CQ}^{\neq}$-Containment, the problem of deciding whether a $\mathrm{CQ}^{\neq}$ is contained in another $\mathrm{CQ}^{\neq}$. We show that this problem is decidable, but its complexity is higher than CQ-Containment, that is, Π_2^p-complete.

Recall that for CQs (without inequality) the building block underlying the procedure for checking containment is the Homomorphism Theorem (Theorem 16.4) that provides a useful characterization of containment in terms of homomorphisms: given two CQs $q(\bar{x})$ and $q'(\bar{x})$, $q \subseteq q'$ iff $(q', \bar{x}') \to (q, \bar{x})$. It is not difficult to see, however, that this is no longer true once inequalities are allowed. This can be easily shown via a simple example.

Example 32.7: The Homomorphism Theorem Fails for $\mathrm{CQ}^{\neq}$

Consider the (Boolean) queries

$$q = \text{Answer} \text{ :– } \mathrm{Edge}(x, y)$$
$$q' = \text{Answer} \text{ :– } \mathrm{Edge}(x', y'), x' \neq y'.$$

It is clear that $q' \to q$, that is, there is a homomorphism h from $A_{q'}^+$ to A_q with $h(x') \neq h(y')$, but $q \not\subseteq q'$: for $D = \{R(a, a)\}$, $D \models q$ but $D \not\models q'$.

This should not be surprising since the existence of an edge in a graph does immediately imply that there is a non-loop edge.

As the above example illustrates, the key reason why a result similar to the Homomorphism Theorem fails for CQs with inequality is because homomorphisms do not compose. In other words, $q' \to q$ and $q \to D$ does not allow us to conclude that $q' \to D$ by simply composing homomorphisms. Observe that after composing h', where $h'(x') = x$ and $h'(y') = y$, which witnesses the fact that $q' \to q$, with h, where $h(x) = h(y) = a$, which witnesses that $q \to D$, we get the function g with $g(x') = g(y') = a$ that is not a homomorphism from q' to D since the inequality is not preserved. The above discussion indicates that we need a version of the Homomorphism Theorem that somehow ensures the following: no matter how the homomorphism h, which witnesses the fact that $q \to D$, looks like, $h \circ h'$ is a homomorphism from q' to D.

For a CQ (without inequality) $q(\bar{x})$, and a CQ$^{\neq}$ $q'(\bar{x}')$, we write $(q', \bar{x}') \to (q, \bar{x})$ for the fact that there exists a homomorphism from $(A_{q'}, \bar{x}')$ to $(A_q, \bar{x})$, which in turn is defined in exactly the same way as the notion of homomorphism from a CQ$^{\neq}$ to a database (see Definition 32.3).

Given a CQ$^{\neq}$ $q(\bar{x})$ of the form

$$\text{Answer}(\bar{x}) \ \coloneq R_1(\bar{u}_1), \ldots, R_n(\bar{u}_n), v_1 \neq v_1', \ldots, v_m \neq v_m' \,,$$

let H_q be the set of all functions $h : \text{Dom}(A_q^+) \cup \text{Dom}(A_q^-) \to \text{Dom}(A_q^+) \cup \text{Dom}(A_q^-)$ such that (i) h is the identity on constants, and (ii) for every $v \neq u \in A_q^-$, $h(v) \neq h(u)$. Intuitively, H_q collects all the possible ways that q can be mapped to a database via a homomorphism. For a function $h \in H_q$, we write $h(q^+)$ for the CQ (without inequality)

$$\text{Answer}(h(\bar{x})) \ \coloneq R_1(h(\bar{u}_1)), \ldots, R_n(h(\bar{u}_n)) \,,$$

i.e., the CQ obtained from q by eliminating the inequalities and then replacing each term u with the term $h(u)$. We are now ready to establish the version of the Homomorphism Theorem for CQs with inequality.

Theorem 32.8

Let $q(\bar{x})$ and $q'(\bar{x}')$ be CQ$^{\neq}$. The following are equivalent:

1. $q \subseteq q'$.
2. For each $h \in H_q$, $(q', \bar{x}') \to (h(q^+), h(\bar{x}))$.

Proof. We first establish that (1) implies (2). Consider an arbitrary function $h \in H_q$; for brevity, let $p = h(q^+)$. Note that p is a CQ without inequalities. By definition of H_q, h is a homomorphism from $(q, \bar{x})$ to $(p, h(\bar{x}))$. Since G_{A_p} is a bijective homomorphism from $(p, h(\bar{x}))$ to $(\mathsf{G}_{A_p}(A_p), \mathsf{G}_{A_p}(h(\bar{x})))$, we conclude

that $(\mathsf{G}_{A_p} \cup \mu) \circ h$, where μ is the identity on the set of constants $\mathrm{Dom}(A^-_{q'}) - \mathrm{Dom}(A_p)$, is a homomorphism from $(q, \bar{x})$ to $(\mathsf{G}_{A_p}(A_p), \mathsf{G}_{A_p}(h(\bar{x})))$. By Theorem 32.4, $\mathsf{G}_{A_p}(h(\bar{x})) \in q(\mathsf{G}_{A_p}(A_p))$. Since, by hypothesis, $q \subseteq q'$, we conclude that $\mathsf{G}_{A_p}(h(\bar{x})) \in q'(\mathsf{G}_{A_p}(A_p))$. By applying again Theorem 32.4, we get that there exists a homomorphism g from $(q', \bar{x}')$ to $(\mathsf{G}_{A_p}(A_p), \mathsf{G}_{A_p}(h(\bar{x})))$. Since G_{A_p} is a bijection, $(\mathsf{G}^{-1}_{A_p} \cup \mu') \circ g$, where μ' is the identity on the set of constants $\mathrm{Dom}(A^-_q) - \mathrm{Dom}(A_p)$, is a homomorphism from $(q', \bar{x}')$ to $(p, h(\bar{x}))$.

We now proceed to show that (2) implies (1). Given a database D, assume that $\bar{a} \in q(D)$. By Theorem 32.4, there exists a homomorphism g from $(q, \bar{x})$ to $(D, \bar{a})$. Let h be a function from $\mathrm{Dom}(A^+_q) \cup \mathrm{Dom}(A^-_q)$ to itself such that

- h is the identity on constants,
- for every two variables $x, y \in \mathrm{Dom}(A^+_q)$, $h(x) = h(y)$ iff $g(x) = g(y)$, and
- for every variable $x \in \mathrm{Dom}(A^+_q)$, $h(x)$ is a variable.

In simple words, h unifies the variables in q^+ that are mapped by g to the same constant of $\mathrm{Dom}(D)$. It is easy to verify that that such a function always exists and belongs to H_q. It is also clear that $(h(q^+), h(\bar{x})) \to (D, \bar{a})$ witnessed via a bijective homomorphism g'. Since $h \in H_q$, by hypothesis, there exists a homomorphism h' from $(q', \bar{x}')$ to $(h(q^+), h(\bar{x}))$. Since g' is a bijection, we get that $(g' \cup \mu) \circ h'$, where μ is the identity on the set of constants $\mathrm{Dom}(A^-_{q'}) - \mathrm{Dom}(A_{h(q^+)})$, is a homomorphism from $(q', \bar{x}')$ to $(D, \bar{a})$. By Theorem 32.4, we get that $\bar{a} \in q'(D)$, and the claim follows. □

The next example illustrates how Theorem 32.8 is used in order to confirm what has been discussed in Example 32.7.

Example 32.9: Application of Theorem 32.8

Consider again the CQs q and q' given in Example 32.7, and recall that $q \not\subseteq q'$. This is confirmed by Theorem 32.8 since, for the function $h \in H_q$ with $h(x) = h(y) = x$, we can conclude that there is no homomorphism from q' to $h(q)$. Indeed, the only way to map q' to $h(q)$ is via the function g with $g(x') = g(y') = x$, which is not a homomorphism from q' to $h(q)$.

From Theorem 32.8, we immediately get a procedure for $\mathsf{CQ}^{\neq}$-Containment: given two CQs with inequality $q(\bar{x})$ and $q'(\bar{x}')$, return `true` if, for every function $h \in H_q$, $(q', \bar{x}) \to (h(q^+), h(\bar{x}))$; otherwise, return `false`. However, this naive approach only shows that $\mathsf{CQ}^{\neq}$-Containment is in ExpTime since H_q consists, in general, of exponentially many functions, i.e., we need to perform exponentially many homomorphism checks, while each one takes exponential time. By providing a more clever procedure, we can establish the following.

Theorem 32.10

$\mathsf{CQ}^{\neq}$-Containment is Π_2^p-complete.

Proof. Consider two $\mathrm{CQ}^{\neq}$ $q(\bar{x})$ and $q'(\bar{x}')$. We proceed to show that checking whether $q \not\subseteq q'$ is in $\Sigma_2^p = \mathrm{NP}^{\mathrm{NP}}$, which in turn implies that $\mathsf{CQ}^{\neq}$-Containment is in Π_2^p. By Theorem 32.8, it suffices to show that the problem of checking whether there exists a function $h \in H_q$ such that there is no homomorphism from $(q', \bar{x}')$ to $(h(q^+), h(\bar{x}))$ is in $\mathrm{NP}^{\mathrm{NP}}$. This can be done by simply guessing a function h from H_q, and the verifying that indeed there is no homomorphism from $(q', \bar{x}')$ to $(h(q^+), h(\bar{x}))$. It is clear that the "guess" step is feasible in polynomial time. Concerning the "verify" step, we first observe the following, which can be easily shown via a simple guess-and-check algorithm:

Lemma 32.11. *Given a $CQ^{\neq}$ $q_1(\bar{x}_1)$ and a CQ (without inequality) $q_2(\bar{x}_2)$, the problem of deciding whether $(q_1, \bar{x}_1) \to (q_2, \bar{x}_2)$ is in* NP.

By Lemma 32.11, we conclude that the "verify" step can be performed in constant time assuming we have access to an oracle that can solve any problem in NP. In particular, the oracle takes as input the queries q' and $h(q^+)$, and does the following: if $(q', \bar{x}') \to (h(q^+), h(\bar{x}))$, then return `false`; otherwise, return `true`. Therefore, checking whether $q \not\subseteq q'$ is in $\mathrm{NP}^{\mathrm{NP}}$, as needed.

For the Π_2^p-hardness of $\mathsf{CQ}^{\neq}$-Containment see Exercise 4.4. □

With Theorem 32.10 in place, we can easily pinpoint the complexity of the equivalence problem for $\mathrm{CQ}^{\neq}$: given two $\mathrm{CQ}^{\neq}$ q and q', check whether $q \equiv q'$, i.e., whether $q(D) = q'(D)$ for every database D. We show that:

Theorem 32.12

$\mathsf{CQ}^{\neq}$-Equivalence is Π_2^p-complete.

Proof. For the upper bound, it suffices to observe that $q \equiv q'$ iff $q \subseteq q'$ and $q' \subseteq q$. This implies that $\mathsf{CQ}^{\neq}$-Equivalence is in Π_2^p since, by Theorem 32.10, the problem of deciding whether $q \subseteq q'$ and $q' \subseteq q$ is in Π_2^p.

For the lower bound, we provide an easy reduction from $\mathsf{CQ}^{\neq}$-Containment that is Π_2^p-hard (Theorem 32.10); in fact, this holds even for Boolean queries. Given two Boolean $\mathrm{CQ}^{\neq}$ q, q', we can easily construct a $\mathrm{CQ}^{\neq}$ $q_\cap$ that computes the intersection of q and q', i.e., for every database D, $q(D) \cap q'(D) = q_\cap(D)$, by merging the bodies of q and q'; a similar construction has been already used in the proof of Theorem 16.8 that deals with CQ-Equivalence. Since $q \subseteq q'$ iff $q \equiv q_\cap$, we get that $\mathsf{CQ}^{\neq}$-Equivalence is Π_2^p-hard, and the claim follows. □

Adding Union to $\mathrm{CQ}^{\neq}$

We now proceed to add the union operator to CQs with inequality.

Definition 32.13: Union of Conjunctive Queries with Inequality

A *union of conjunctive queries with inequality* (UCQ$^{\neq}$) over a schema $\mathbf{S}$ is an FO query $\varphi(\bar{x})$ over $\mathbf{S}$ where φ is a formula of the form

$$\varphi_1 \vee \cdots \vee \varphi_n$$

for $n \geq 1$, where $\mathrm{FV}(\varphi) = \mathrm{FV}(\varphi_i)$ and $\varphi_i(\bar{x})$ is a CQ$^{\neq}$, for every $i \in [n]$.

As for UCQs without inequality, for notational convenience, we denote a UCQ$^{\neq}$ $q = \varphi(\bar{x})$ with $\varphi = \varphi_1 \vee \cdots \vee \varphi_n$ as $q_1 \cup \cdots \cup q_n$, where q_i is the CQ$^{\neq}$ $\varphi_i(\bar{x})$, for each $i \in [n]$. It is easy to verify that, given a database D of a schema $\mathbf{S}$, and a UCQ$^{\neq}$ $q = q_1 \cup \cdots \cup q_n$ over $\mathbf{S}$, $q(D) = q_1(D) \cup \cdots \cup q_n(D)$.

Example 32.14: Union of Conjunctive Queries with Inequality

Consider the relational schema from Example 3.2:

```
Person [ pid, pname, cid ]
Profession [ pid, prname ]
City [ cid, cname, country ]
```

The UCQ $\varphi(y)$, where $\varphi = \varphi_1 \vee \varphi_2$ with

$$\varphi_1 = \exists x \exists z \big(\mathrm{Person}(x, y, z) \wedge \mathrm{Profession}(x, \text{`computer scientist'}) \wedge \mathrm{City}(z, w, \text{`Greece'}) \wedge w \neq \text{`Athens'}\big).$$

and

$$\varphi_2 = \exists x \exists z \big(\mathrm{Person}(x, y, z) \wedge \mathrm{Profession}(x, \text{`computer scientist'}) \wedge \mathrm{City}(z, w, \text{`Chile'}) \wedge w \neq \text{`Santiago'}\big).$$

can be used to retrieve the list of names of computer scientists that were born in Greece or Chile, but not in the capital city of the country.

Query Evaluation

We proceed with UCQ$^{\neq}$-Evaluation, the problem of checking whether $\bar{a} \in q(D)$ for a UCQ$^{\neq}$ q, a database D, and a tuple $\bar{a}$ over $\mathrm{Dom}(D)$. We show that it has the same (combined) complexity as CQ$^{\neq}$-Evaluation; concerning the data complexity, both problems are in DLogSpace due to Theorem 7.3.

Theorem 32.15

UCQ$^{\neq}$-Evaluation is NP-complete.

Proof. It is clear that the NP-hardness is inherited from CQ-Evaluation, which we know is NP-hard (Theorem 15.1). We proceed to show the upper bound. Consider a UCQ$^{\neq}$ $q(\bar{x})$ of the form $q_1 \cup \cdots \cup q_n$, a database D, and a tuple $\bar{a} \in \mathrm{Dom}(D)$. By Theorem 32.4, for $i \in [n]$, $\bar{a} \in q_i(D)$ if and only if $(q_i, \bar{x}) \to (D, \bar{a})$. Thus, we need to show that checking whether there exists an integer $i \in [n]$ and a homomorphism from $(q_i, \bar{x})$ to $(D, \bar{a})$ is in NP. This is done by guessing an integer $i \in [n]$ and a function $h : \mathrm{Dom}(A^+_{q_i}) \cup \mathrm{Dom}(A^-_{q_i}) \to \mathrm{Dom}(D) \cup \mathrm{Dom}(A^-_{q_i})$, and then verifying that h is indeed a homomorphism from $(A_{q_i}, \bar{x})$ to $(D, \bar{a})$, i.e., is a homomorphism from $A^+_{q_i}$ to D, is the identity on $\mathrm{Dom}(A^-_{q_i})$, for every $v \neq u \in A^-_{q_i}$, $h(v) \neq h(u)$, and $h(\bar{x}) = \bar{a}$. Since all the above steps are feasible in polynomial time, the claim follows. □

Containment

We finally concentrate on UCQ$^{\neq}$-Containment, the problem of deciding whether a UCQ$^{\neq}$ q is contained in a UCQ$^{\neq}$ q', that is, whether $q(D) \subseteq q'(D)$ for every database D. We show that it has the same complexity as CQ$^{\neq}$-Containment. To this ends, we first observe that Proposition 31.1 for UCQs (without inequality) holds also for UCQs with inequality. In fact, the proof of Proposition 31.1 only exploits the fact that, for a database D, and a UCQ $q = q_1 \cup \cdots \cup q_n$, $q(D) = \bigcup_{i \in [n]} q_i(D)$. Since the latter property holds even for UCQs with inequality, the very same proof shows the following.

Proposition 32.16

Consider two UCQ$^{\neq}$ $q = q_1 \cup \cdots \cup q_n$ and $q' = q'_1 \cup \cdots \cup q'_m$. The following are equivalent:

1. $q \subseteq q'$.
2. For every $i \in [n]$, there exists $j \in [m]$ such that $q_i \subseteq q'_j$.

We are now ready to pinpoint the complexity of UCQ$^{\neq}$-Containment.

Theorem 32.17

UCQ$^{\neq}$-Containment is Π^p_2-complete.

Proof. It is clear that the Π^p_2-hardness is inherited from CQ$^{\neq}$-Containment, which we know is Π^p_2-hard (Theorem 32.10). Consider now two UCQ$^{\neq}$ $q = q_1 \cup \ldots \cup q_n$ and $q' = q'_1 \cup \cdots \cup q'_m$. We proceed to show that checking whether

$q \not\subseteq q'$ is in $\Sigma_2^p = \mathrm{NP}^{\mathrm{NP}}$, which in turn implies that $\mathsf{UCQ}^{\neq}$-$\mathsf{Containment}$ is in Π_2^p. We assume that, for $i \in [n]$ and $j \in [m]$, the output tuple of q_i and q'_j is $\bar{x}$ and $\bar{x}'$, respectively. By Theorem 32.8 and Proposition 32.16, to check whether $q \not\subseteq q'$ it suffices to do the following:

- guess an $i \in [n]$, and, for every $j \in [m]$, guess a function $h_j \in H_{q_i}$, and
- for each $j \in [m]$, verify that there is no homomorphism from $(q'_j, \bar{x}')$ to $(h_j(q_i^+), h_j(\bar{x}))$.

It is clear that the "guess" step is feasible in polynomial time. Moreover, by Lemma 32.11, the "verify" step can be performed in polynomial time assuming access to an oracle that can solve any problem in NP. In particular, for each $j \in [m]$, the oracle is called with input the queries q'_j and $h_j(q_i^+)$, and does the following: if $(q'_j, \bar{x}') \to (h_j(q_i^+), h_j(\bar{x}))$, then return `false`; otherwise, return `true`. Therefore, deciding whether $q \not\subseteq q'$ is in $\mathrm{NP}^{\mathrm{NP}}$, as needed. □

An immediate corollary of Theorem 32.17 is that the equivalence problem for UCQs with inequality, that is, given two $\mathrm{UCQ}^{\neq}$ q, q', check whether $q \equiv q'$, is in Π_2^p since it boils down to two containment checks: $q \subseteq q'$ and $q' \subseteq q$. The Π_2^p-hardness is inherited from $\mathsf{CQ}^{\neq}$-$\mathsf{Equivalence}$ (Theorem 32.12).

Corollary 32.18

$\mathsf{UCQ}^{\neq}$-$\mathsf{Equivalence}$ is Π_2^p-complete.

33

The Limits of First-Order Queries

We have seen that the language of UCQs has the same expressive power as the language of $\mathrm{FO}^{\mathrm{rel}}[\wedge, \vee, \exists]$ queries (Theorem 30.3), or even $\mathrm{FO}^{\mathrm{rel,var=}}[\wedge, \vee, \exists]$ queries (Theorem 30.4). Adding variable-constant equality, i.e., allowing equational atoms of the form $(x = a)$, where x is a variable and a a constant, to the above languages, leads to the strictly more expressive language of $\exists\mathrm{FO}^{+}$ queries (Theorem 30.5). If we further add negation to $\exists\mathrm{FO}^{+}$ queries, we then get the full power of FO queries, which is strictly more expressive than $\exists\mathrm{FO}^{+}$ queries. Indeed, universal quantification can be expressed by means of negation and existential quantification: $\forall x\, \varphi$ is equivalent to $\neg\exists x\, \neg\varphi$. In fact, practical languages, such as SQL, do not offer explicit universal quantification, but instead express universal statements via negated existential statements.

We already know some interesting facts about the language of FO queries. In terms of expressive power, it is equally expressive to the language of RA queries (Theorem 6.1). This tells us that adding negation to $\exists\mathrm{FO}^{+}$ queries is the same as extending RA^{+} queries with the difference operation, and allowing inequalities in conditions. The problem of evaluating FO queries is PSPACE-complete in combined complexity (Theorem 7.1), and in DLOGSPACE in data complexity (Theorem 7.3). On other hand, static analysis for FO queries, unlike $\exists\mathrm{FO}^{+}$ queries or $\mathrm{UCQ}^{\neq}$, is undecidable (Theorems 8.1 and 8.3).

We are now more interested in the limitations of FO queries, as they will justify what practical languages need to add on top of the language of FO queries. In this chapter, we present two fundamental inexpressibility results concerning constant-free first-order queries:

- They cannot express nontrivial statements about cardinalities of sets (for example, is the cardinality of a set even?).
- They cannot compare cardinalities of relations.

Let us stress that the results presented in this chapter do not hold for FO queries with constants. This is discussed further in the comments for Part IV.

An Easy Expressiveness Bound

We start by providing a rather preliminary result on the expressive power of constant-free FO queries (Theorem 33.3), which in turn allows us to conclude that such queries cannot express nontrivial statements about the cardinalities of sets. Note that an FO sentence φ is called *constant-free* if it does not mention any constants, that is, the set $\mathrm{Dom}(\varphi)$ is empty. We further call an FO query $\varphi(\bar{x})$ constant-free if φ is constant-free.

Let $\mathbf{S} = \{R[1]\}$, i.e., is a schema consisting of a single unary relation name R. Databases of $\mathbf{S}$ are essentially sets, i.e., they store the elements of a set R. It is easy to see that two databases D and D' of $\mathbf{S}$ with $|D| = |D'|$ satisfy exactly the same constant-free FO sentences over $\mathbf{S}$, i.e., for every such FO sentence φ over $\mathbf{S}$, $D \models \varphi$ iff $D' \models \varphi$. This is because D and D' are the same up to renaming of constants. We can thus define for a sentence φ over $\mathbf{S}$

$$\mu_n(\varphi) \;=\; \begin{cases} 1 & \text{if } D \models \varphi, \text{ for every } D \in \mathrm{Inst}(\mathbf{S}) \text{ with } |D| = n \\ \\ 0 & \text{if } D \not\models \varphi, \text{ for every } D \in \mathrm{Inst}(\mathbf{S}) \text{ with } |D| = n\,. \end{cases}$$

We can then show the following useful technical result:

Proposition 33.1

Consider a constant-free FO sentence φ over $\mathbf{S} = \{R[1]\}$. There is $k \in \mathbb{N}$ such that either $\mu_n(\varphi) = 1$ for all $n \geq k$, or $\mu_n(\varphi) = 0$ for all $n \geq k$.

To prove the above result we need a few basic notions and facts about first-order logic. A (possibly infinite) set T of first-order sentences over a schema $\mathbf{S}$ is called a *first-order theory* over $\mathbf{S}$, or simply a *theory* over $\mathbf{S}$. The notion of satisfaction of an FO sentence by a database (see Definition 3.3) can be naturally extended to possibly infinite databases. We call a possibly infinite database of a schema $\mathbf{S}$ a *model* of a theory T over $\mathbf{S}$ if, for every sentence $\varphi \in T$, $D \models \varphi$. We further say that T is *consistent* if it has at least one model. We know that a consistent theory T over $\mathbf{S}$ has always a countably infinite model since $\mathbf{S}$ is finite and Const is countably infinite; the latter is a consequence of a basic result in logic known as the Löwenheim-Skolem Theorem. A sentence φ is a consequence of a theory T, written $T \models \varphi$, if every model of T satisfies φ. We further know that if $T \models \varphi$, then there exists a finite subset T_0 of T such that $T_0 \models \varphi$; this is known as the Compactness Theorem of first-order logic. We are now ready to give the proof of Proposition 33.1.

Proof (of Proposition 33.1). Consider the theory $T = \{\psi_n \mid n \in \mathbb{N}\}$, where

$$\psi_n \;=\; \exists x_1 \cdots \exists x_n \bigwedge_{i,j \in [n] \,:\, i<j} \neg(x_i = x_j),$$

i.e., it states that there are n distinct elements. Clearly, T is consistent since any possibly infinite database of $\mathbf{S}$ is a model of T. It is easy to show that:

Lemma 33.2. *Either* $T \models \varphi$ *or* $T \models \neg\varphi$.

Proof. Notice that $T \models \varphi$ and $T \models \neg\varphi$ cannot be both true since in this case there exists a possibly infinite database D of $\mathbf{S}$ such that $D \models \varphi$ and $D \models \neg\varphi$, which cannot be the case. Assume now that $T \not\models \varphi$ and $T \not\models \neg\varphi$. This implies that both theories $T \cup \{\varphi\}$ and $T \cup \{\neg\varphi\}$ are consistent, and thus, they have countably infinite models. Since there is only one countably infinite model, up to isomorphism, we get a contradiction as it cannot satisfy both $\varphi, \neg\varphi$. □

We now proceed to show the claim by considering the two cases provided by Lemma 33.2: either $T \models \varphi$ or $T \models \neg\varphi$. Assume first that $T \models \varphi$. By the Compactness Theorem, there exists a finite subset T_0 of T such that $T_0 \models \varphi$. Let ψ_k be the sentence with the largest index k among the sentences of T_0. It is clear that $\psi_k \models \varphi$ since $\psi_k \models \psi_m$ whenever $m \leq k$. Therefore, for every database D of $\mathbf{S}$ with $|D| \geq k$, $D \models \varphi$. This implies that $\mu_n(\varphi) = 1$ for every $n \geq k$. Analogously, if $T \models \neg\varphi$, then we can show that there is $k \in \mathbb{N}$ such that $\mu_n(\varphi) = 0$ for every $n \geq k$, and the claim follows. □

We can use Proposition 33.1 to show that only simple properties of cardinalities of sets can be expressed using constant-free FO queries. We proceed to make this more precise. Given a set of integers $C \subseteq \mathbb{N}$, let q_C be a Boolean query over the schema $\mathbf{S} = \{R[1]\}$ that asks whether the cardinality of the input database is equal to an integer of C. In other words, for every database D of $\mathbf{S}$, $D \models q_C$ iff $|D| \in C$. Interestingly, we can precisely characterize when q_C is expressible as a constant-free FO query.

Theorem 33.3

Let $C \subseteq \mathbb{N}$, and $\mathbf{S}$ be the schema $\{R[1]\}$. The following are equivalent:

1. There is a constant-free FO query q over $\mathbf{S}$ such that $q_C \equiv q$.
2. Either C is a finite set, or $\mathbb{N} - C$ is a finite set.

Proof. We first prove that (1) implies (2). Since, by hypothesis, q_C can be expressed as a constant-free FO query, Proposition 33.1 implies that there is an integer $k \in \mathbb{N}$ such that one of the following statements hold:

(i) For every $D \in \text{Inst}(\mathbf{S})$ with $|D| \geq k$, $D \models q_C$.
(ii) For every $D \in \text{Inst}(\mathbf{S})$ with $|D| \geq k$, $D \not\models q_C$.

Assuming that (i) holds, there are finitely many integers, let say $i_1, \ldots, i_m$, for $m \geq 0$, such that, given a database D' of $\mathbf{S}$ with $|D'| \in \{i_1, \ldots, i_m\}$, $D' \not\models q_C$.

This in turn implies that $\mathbb{N} - C$ is finite. Analogously, when (ii) holds, we can show that C is finite, and statement (2) follows.

We now show that (2) implies (1). This is shown by constructing a Boolean constant-free FO query q over $\mathbf{S}$ such that $q_C \equiv q$. We first observe that, given an integer $k \in \mathbb{N}$, it is easy to construct an FO sentence φ_k over $\mathbf{S}$ that is satisfied only by databases D over $\mathbf{S}$ with $|D| = k$; in particular, φ_k is

$$\exists x_1 \cdots \exists x_k \left(\bigwedge_{i=1}^{k} R(x_i) \wedge \bigwedge_{i,j \in [k] \,:\, i<j} \neg(x_i = x_j) \right) \wedge \forall x_1 \cdots \forall x_{k+1} \left(\bigwedge_{i=1}^{k+1} R(x_i) \rightarrow \bigvee_{i,j \in [k+1] \,:\, i<j} x_i = x_j \right),$$

where the first conjunct states that $|D| \geq k$, while the second conjunct states that $|D| \leq k$. By exploiting the fact that either C is finite, or $\mathbb{N} - C$ is finite, the desired Boolean constant-free FO query q is defined as $\varphi()$, where

$$\varphi \;=\; \begin{cases} \bigvee_{i \in C} \varphi_i & \text{if } C \text{ is finite} \\ \neg \left(\bigvee_{i \in \mathbb{N} - C} \varphi_i \right) & \text{if } \mathbb{N} - C \text{ is finite}. \end{cases}$$

It is easy to verify that $q_C \equiv q$, and the claim follows. □

According to Theorem 33.3, it is impossible to check using a constant-free FO query whether the cardinality of a set is even, or, more generally, whether is divisible by some number $n \in \mathbb{N}$. Such a query is of the form q_C where C is an infinite set, and thus, not expressible as a constant-free FO query.

Zero-One Law

Although Theorem 33.3 allows us to conclude that constant-free FO queries cannot express nontrivial statements about the cardinalities of sets, it is not powerful enough to tell us something about comparisons of cardinalities of relations. We proceed to establish a stronger property of FO sentences than the one established by Proposition 33.1 above, known as *zero-one law*, which will allow us to derive inexpressibility results concerning cardinality comparisons.

We start by reformulating the definition of the function μ_n used above. We assume that the values occurring in a database are integers in order to be able to enumerate them. Then, for an FO sentence φ over a schema $\mathbf{S}$, let

$$\mu_n(\varphi) \;=\; \frac{|\{D \in \mathrm{Inst}(\mathbf{S}) \mid \mathrm{Dom}(D) = [n] \text{ and } D \models \varphi\}|}{|\{D \in \mathrm{Inst}(\mathbf{S}) \mid \mathrm{Dom}(D) = [n]\}|}.$$

In simple words, $\mu_n(\varphi)$ is the proportion of databases of $\mathbf{S}$ with $\mathrm{Dom}(D) = [n]$ that satisfy φ. Notice that this new definition of the function μ_n applies to

arbitrary schemas, not only to those with a single unary relation. The intuition behind the quantity $\mu_n(\varphi)$ can be described in purely probabilistic terms. Consider the finite set of databases D of $\mathbf{S}$ with $\mathrm{Dom}(D) = [n]$. Then $\mu_n(\varphi)$ is the probability that a database one picks uniformly at random from this set satisfies φ. Now, by taking the limit $\lim_{n\to\infty} \mu_n(\varphi)$, we essentially describe the asymptotic behavior of the sequence $(\mu_i(\varphi))_{i>0}$; intuitively, it defines the probability that a randomly picked database satisfies φ.

Definition 33.4: 0–1 Law

We say that an FO sentence φ over a schema $\mathbf{S}$ enjoys the 0–1 law if

$$\lim_{n\to\infty} \mu_n(\varphi) \ \in \ \{0, 1\}.$$

Intuitively, if an FO sentence φ over a schema $\mathbf{S}$ enjoys the 0–1 law, then it is either satisfied by almost all the databases of $\mathbf{S}$, or violated by almost all the databases of $\mathbf{S}$. This is the case for constant-free FO sentences over a schema with a single unary relation name. Observe that there exists only one database D of $\mathbf{S} = \{R[1]\}$ with $\mathrm{Dom}(D) = [n]$. Therefore, for a constant-free FO sentence φ over $\mathbf{S}$, Proposition 33.1 says that the sequence $(\mu_i(\varphi))_{i>0}$ eventually stabilizes, namely $\lim_{n\to\infty} \mu_n(\varphi) \in \{0, 1\}$. Interestingly, this can be shown for every constant-free FO sentence over an arbitrary schema.

Before showing this, let us stress that not all logical sentences enjoy the 0–1 law. There are, for example, FO sentences that mention constants that do not enjoy the 0–1 law; this is discussed further in the comments for Part IV. Another example is the sentence φ_{EVEN}, expressed in a logical formalism that goes beyond first-order logic,[1] that checks whether the cardinality of a database of the schema $\mathbf{S} = \{R[1]\}$ is even, i.e., for every database D of $\mathbf{S}$, $D \models \varphi_{\text{EVEN}}$ iff $|D|$ is even. Thus, $\mu_n(\varphi_{\text{EVEN}})$ is 1 when n is even, and 0 when n is odd, which means that the limit $\lim_{n\to\infty} \mu_n(\varphi)$ does not even exist.

Theorem 33.5: 0–1 Law

Every constant-free FO sentence enjoys the 0–1 law.

Proof. We shall not prove the result in its full generality, but instead consider the special case of constant-free FO sentences over a schema $\mathbf{S}$ with two unary relation names, i.e., $\mathbf{S} = \{R[1], S[1]\}$.[2] This special case builds on the proof of Proposition 33.1, illustrates key elements in the proof of the 0–1 law, and allows us to derive corollaries about cardinality comparisons.

[1] The sentence φ_{EVEN} can actually be expressed using second-order logic that extends first-order logic by allowing quantification over sets of domain elements.

[2] Ideas on how to extend the proof to graphs, i.e., to schemas with a single binary relation name, are explained in Exercise 4.12.

The key elements are the same as in the general proof of the 0–1 law, and are summarized in the next technical lemma:

Lemma 33.6. *There exists a first-order theory T over $\boldsymbol{S}$ such that:*

1. $\lim_{n\to\infty} \mu_n(\psi) = 1$ *for each* $\psi \in T$, *and*
2. T *has a unique, up to isomorphism, countably infinite model.*

Before we give the proof of the lemma, let us explain how it can be used to show that every constant-free FO sentence enjoys the 0–1 law. Let T be the theory provided by Lemma 33.6. Condition (2) of the lemma, and the same argument as in the proof of Lemma 33.2, show that for every constant-free FO sentence φ, either $T \models \varphi$ or $T \models \neg\varphi$. Consider now a constant-free FO sentence φ over $\mathbf{S}$. We proceed by case analysis:

- Assume that $T \models \varphi$. By the Compactness Theorem, φ is a consequence of a finite subset $\{\psi_1, \ldots, \psi_m\}$ of sentences of T. Since $\lim_{n\to\infty} \mu_n(\psi_i) = 1$ for each $i \in [m]$, we get that $\lim_{n\to\infty} \mu_n(\bigwedge_{i=1}^m \psi_i) = 1$. Therefore, we have that $\lim_{n\to\infty} \mu_n(\varphi) = 1$ since $\mu_n(\varphi) \geq \mu_n(\bigwedge_{i=1}^m \psi_i)$.
- Assume now that $T \models \neg\varphi$. We apply the same argument to $\neg\varphi$, and conclude that $\lim_{n\to\infty} \mu_n(\neg\varphi) = 1$, which implies that $\lim_{n\to\infty} \mu_n(\varphi) = 0$.

Hence, $\lim_{n\to\infty} \mu_n(\varphi) \in \{0, 1\}$. We proceed with the proof of Lemma 33.6.

Proof (of Lemma 33.6). We construct a theory T over $\mathbf{S}$, and then show that satisfies the conditions (1) and (2). For each $k \in \mathbb{N}$, T contains the sentences

$$\psi_k(RS) \;=\; \exists x_1 \cdots \exists x_n \left(\bigwedge_{i,j\in[n]\,:\,i<j} \neg(x_i = x_j) \wedge \bigwedge_{i\in[n]} (R(x_i) \wedge S(x_i)) \right),$$

which states that, for a database D, $R^D \cap S^D$ has at least k elements,

$$\psi_k(R\bar{S}) \;=\; \exists x_1 \cdots \exists x_n \left(\bigwedge_{i,j\in[n]\,:\,i<j} \neg(x_i = x_j) \wedge \bigwedge_{i\in[n]} (R(x_i) \wedge \neg S(x_i)) \right),$$

which states that $R^D - S^D$ has at least k elements, and

$$\psi_k(\bar{R}S) \;=\; \exists x_1 \cdots \exists x_n \left(\bigwedge_{i,j\in[n]\,:\,i<j} \neg(x_i = x_j) \wedge \bigwedge_{i\in[n]} (\neg R(x_i) \wedge S(x_i)) \right),$$

which states that $S^D - R^D$ has at least k elements.

We first observe that the theory T is consistent. Indeed, it has a countably infinite model D such that $R^D \cup S^D = \mathbb{N}$ (recall the assumption that the values

occurring in a database are integers), and $R^D \cap S^D$, $R^D - S^D$, and $S^D - R^D$ are countably infinite; for example, we can take $R^D = \{3n, 3n+1 \mid n \in \mathbb{N}\}$ and $S^D = \{3n, 3n+2 \mid n \in \mathbb{N}\}$. Notice that any two such infinite databases are isomorphic, and thus, up to isomorphism, T has only one countably infinite model, satisfying condition (2) of Lemma 33.6.

We now prove that T satisfies the first condition. We show that, for each integer $k \in \mathbb{N}$, the sentences $\psi_k(RS), \psi_k(R\bar{S})$, and $\psi_k(\bar{R}S)$ are true in almost all databases of $\mathbf{S}$. We do this as an illustration for the sentence $\psi_k(RS)$. To this end, we consider its negation stating that, for a database D, $|R^D \cap S^D| < k$. We first provide an upper bound for $\mu_n(\neg\psi_k(RS))$:

- The numerator of $\mu_n(\neg\psi_k(RS))$ coincides with the number of different ways that we can choose two sets R and S from $[n]$ such that $R \cup S = [n]$ and $|R \cap S| < k$. For each $j < k$, we have $\binom{n}{j}$ ways to choose an intersection $R \cap S$ of cardinality less than k. Then we need to choose the elements of $R - S$ from the remaining $n - j$ elements, and there are 2^{n-j} ways of doing so. The remaining elements must belong to $S - R$ since $R \cup S = [n]$. Thus, there are $\binom{n}{j} \cdot 2^{n-j}$ ways to choose two sets from $[n]$ whose intersection has exactly j elements, and whose union contains all the elements of $[n]$. This means that there are at most

$$\sum_{j \in [0,k-1]} \binom{n}{j} \cdot 2^{n-j} \;\leq\; n^k \cdot 2^n$$

  ways of choosing two sets whose intersection has cardinality less than k.
- The denominator of $\mu_n(\neg\psi_k(RS))$ coincides with the number of ways to choose two sets R and S from $[n]$ such that $R \cup S = [n]$. There are 3^n ways of doing so since, for each element of $[n]$, there are 3 possibilities: it can either belong to $R \cap S$, or $R - S$, or $S - R$.

From the above analysis, we conclude that

$$\mu_n(\neg\psi_k(RS)) \;\leq\; n^k \left(\frac{2}{3}\right)^n$$

which means $\lim_{n\to\infty} \mu_n(\neg\psi_k(RS)) = 0$, and therefore

$$\lim_{n\to\infty} \mu_n(\psi_k(RS)) \;=\; 1.$$

The proof for $\psi_k(R\bar{S})$ and $\psi_k(\bar{S}R)$ are very similar. This shows that T satisfies both conditions (1) and (2), and concludes the proof of Lemma 33.6. □

This completes the proof of Theorem 33.5. □

Theorem 33.5 allows us to establish inexpressibility results concerning cardinality comparisons. For $\diamond \in \{<, \leq, =\}$, let $q_\diamond$ be a Boolean query over the

schema $\mathbf{S} = \{R[1], S[1]\}$ that compares the cardinalities of the relations R^D and S^D for a database D according to the comparison $\diamond$. In other words, for every database D of $\mathbf{S}$, the query $q_\diamond$ is true in D iff the comparison $|R^D| \diamond |S^D|$ is true. We now show via a 0–1 law argument that none of the comparisons $|R| = |S|$, or $|R| < |S|$, or $|R| \leq |S|$, is expressible as a constant-free FO query.

Theorem 33.7

Let $\mathbf{S} = \{R[1], S[1]\}$. For every $\diamond \in \{<, \leq, =\}$, there is no constant-free FO query q over $\mathbf{S}$ such that $q_\diamond \equiv q$.

Proof. We prove the result for the case of $q_<$ via a 0–1 law argument; a very similar argument works for $q_\leq$. The case of $q_=$ can be also shown via a 0–1 law argument, and is left as an exercise (see Exercise 4.9).

Let $F_n^=$ be the number of all databases D of $\mathbf{S}$ such that $\mathrm{Dom}(D) = [n]$ and $|R^D| = |S^D|$; we define $F_n^<$ and $F_n^>$ likewise. From the proof of Theorem 33.5 we know that $F_n^= + F_n^< + F_n^> = 3^n$. Moreover, by symmetry, $F_n^< = F_n^>$, and thus, $F_n^= + 2F_n^< = 3^n$. We first estimate the value $F_n^=$. To have a database D in which $|R^D| = |S^D|$, for every $k \leq \lfloor n/2 \rfloor$, we can pick k elements to belong to $R^D - S^D$, from the remaining $n - k$ elements we can pick k elements to belong to $S^D - R^D$, and the remaining ones belong to $R^D \cap S^D$. Hence

$$F_n^= = \sum_{k \leq \lfloor n/2 \rfloor} \binom{n}{k} \binom{n-k}{k}$$

and one can show (Exercise 4.11) that

$$\lim_{n \to \infty} \frac{F_n^=}{3^n} = 0.$$

Assume now that there is a constant-free FO query $q = \varphi()$ such $q_< \equiv q$, i.e., q expresses the condition $|R^D| < |S^D|$. Then

$$\lim_{n \to \infty} \mu_n(\varphi) = \lim_{n \to \infty} \frac{F_n^<}{3^n} = \lim_{n \to \infty} \frac{3^n - F_n^=}{2 \cdot 3^n} = \frac{1}{2} - \lim_{n \to \infty} \frac{F_n^=}{2 \cdot 3^n} = \frac{1}{2}$$

which contradicts the 0–1 law, and the claim follows. □

34

Adding Aggregates and Grouping

When the core of SQL was presented in Chapter 5, a commonly used feature of it was omitted, namely *aggregation*. It is typically used together with *grouping* to apply numerical functions to entire columns of a relation. Recall that one of the relation names in the schema that we usually use in examples is

```
City [ cid, cname, country ]
```

If we want to know how many cities each country has, we can write in SQL

```
SELECT country, COUNT(cid) AS city_count
FROM City
GROUP BY country
```

For each value of the country attribute, it groups together all the tuples having this value, and then counts the number of occurrences of cid in such a group and outputs it as the value of the new attribute city_count. Here, `COUNT` is an *aggregate function*: it applies to a collection, and produces a single numerical value. The standard aggregates of SQL, in addition to `COUNT`, are `SUM` and `AVG` that compute the sum and the average of a collection of *numbers*, as well as `MIN` and `MAX` that compute the minimum and the maximum.

Queries with aggregates are extremely common and useful in practice; for example, "find the average grade for each class" or "find the total cost of products sold to each country". The addition of aggregate functions, however, takes us out of the realm of FO and RA queries. Indeed, by Theorem 33.7, we know that cardinality comparisons are not expressible via FO queries. However, they are easily expressible with the help of aggregate functions:

```
SELECT DISTINCT 1 FROM R, S
WHERE (SELECT COUNT(*) FROM R) > (SELECT COUNT(*) FROM S)
```

outputs 1 if $|R| > |S|$, and nothing otherwise.

In this chapter, we introduce a query language with aggregates and grouping based on RA. Of course one can also define a logical language with aggregates, but this is cumbersome for the reasons that are explained below. Note that in Chapter 35, we analyze queries that cannot be expressed even if we have the powerful features of aggregates and grouping. Let us now discuss the technical issues that arise due to aggregates:

Numerical Attributes. So far we assumed that database elements come from a countably infinite set Const of values. With the addition of aggregates, however, we can no longer make this assumption, as we need to distinguish attributes that are *numerical*. For example, we can only apply AVG over numbers. Thus, in the description of relational schemas, it is no longer sufficient to simply state what the arity of each relation name is. In addition, we need to provide information about attributes that are numerical. The standard approach for solving this technical issue is to consider *two-sorted schemas*: there will be columns populated by the usual values from Const, and columns populated by values from a numerical domain Num; e.g., the natural numbers $\mathbb{N}$, or the integers $\mathbb{Z}$, or the rationals $\mathbb{Q}$.

Infinite Numerical Domains. The second issue manifests itself when we deal with logical languages for aggregates. In Chapter 3, we defined the satisfaction of a logical formula over the active domain of the database and the formula (recall that Definition 3.3 defines the active domain semantics of first-order logic). However, aggregates can produce new numerical values that do not occur in the active domain. Therefore, the satisfaction of formulae must be defined with respect to the entire infinite numerical domain Num. This leads to the situation where a logical formula φ may be satisfied by an infinite number of assignments of values to its free variables, and thus, the expression $\varphi(\bar{x})$, for some tuple $\bar{x}$ over the free variables of φ, may not define a query since its output on a database may be infinite. Although we can define a logical language with aggregates that does not exhibit this problem, its syntax is cumbersome. Therefore, we present the language with aggregates and grouping at the level of relational algebra, where the above problem does not arise. We present an extension of first-order logic with aggregates in Chapter 35 that uses a relatively simple syntax, and show that RA with aggregates and grouping translates to it. However, this logical language will not serve as the basis for defining a query language with aggregates, but rather as a technical tool for analyzing the expressive power of aggregates.

Two-Sorted Schemas, Databases and Queries

We first revisit the notions of database schema, database instance, and query in order to take into account the fact that relations can now have both ordinary and numerical values. These are actually straightforward adaptations of

the definitions given in Chapter 2, but, for the sake of completeness and readability, we proceed to properly introduce those notions. As usual, for technical clarity, we adopt the unnamed perspective.

Each relation name R in a schema should come not simply with its arity k, but rather with a tuple $\boldsymbol{\tau}$ of arity k over $\{\mathsf{o}, \mathsf{n}\}$, where o indicates a column of ordinary type taking values from Const, and n indicates a column of numerical type taking values from a set of numerical values Num. The formal definition of two-sorted database schemas follows.

Definition 34.1: Two-Sorted Database Schema

An *two-sorted (database) schema* is a partial function

$$\mathbf{S} \ : \ \mathsf{Rel} \to \{\mathsf{o}, \mathsf{n}\}^k$$

for $k \in \mathbb{N}$ such that $\mathrm{Dom}(\mathbf{S})$ is finite. For a relation name $R \in \mathrm{Dom}(\mathbf{S})$, the *arity of R under $\boldsymbol{S}$*, denoted $\mathrm{ar}_{\mathbf{S}}(R)$, is defined as k.

In order to avoid heavy notation, we write $\mathrm{ar}(R)$ instead of $\mathrm{ar}_{\mathbf{S}}(R)$ for the arity of R under $\mathbf{S}$. We may even write $R : \boldsymbol{\tau}$ to indicate that $\mathbf{S}(R) = \boldsymbol{\tau}$. As for plain schemas, a two-sorted schema can be naturally seen as a finite set of relation names. We may also write $\mathbf{S} = \{R_1 : \boldsymbol{\tau}_1, \ldots, R_n : \boldsymbol{\tau}_n\}$ for the fact that $\mathrm{Dom}(\mathbf{S}) = \{R_1, \ldots, R_n\}$ and $\mathbf{S}(R_i) = \boldsymbol{\tau}_i$ for each $i \in [n]$.

The elements of database tuples are coming from the set of values Const, and the set of numerical values Num, namely a *two-sorted database tuple* is an element of $(\mathsf{Const} \cup \mathsf{Num})^k$ for some $k \in \mathbb{N}$. A *two-sorted relation instance* is a *finite* set S of two-sorted database tuples of the same arity k. We say that k is the *arity* of S, denoted by $\mathrm{ar}(S)$. By tsRI (for *two-sorted relation instances*) we denote the set of all such relation instances. The formal definition of a database instance of a two-sorted schema follows.

Definition 34.2: Two-Sorted Database Instance

A *database instance* D of a two-sorted schema $\mathbf{S}$ is a function

$$D \ : \ \mathrm{Dom}(\mathbf{S}) \to \mathsf{tsRI}$$

such that, for every $R \in \mathrm{Dom}(\mathbf{S})$, the following hold:

- $\mathrm{ar}(D(R)) = \mathrm{ar}_{\mathbf{S}}(R)$, and
- with $\mathbf{S}(R) = (\tau_1, \ldots, \tau_k)$ and $D(R) = (a_1, \ldots, a_k)$, $a_i \in \mathsf{Const}$ if $\tau_i = \mathsf{o}$, and $a_i \in \mathsf{Num}$ if $\tau_i = \mathsf{n}$, for every $i \in [k]$.

We will refer to a database instance of a two sorted schema as a two-sorted database, or simply database whenever is clear that the underlying schema is two-sorted. The active domain (or simply domain) of a two-sorted database

D is defined in the same way as the active domain of a plain database given in Chapter 2, and is denoted $\mathrm{Dom}(D)$; we will *never* use the term domain, and the notation $\mathrm{Dom}(D)$, to refer to the domain of the function D, i.e., $\mathrm{Dom}(\mathbf{S})$. A two-sorted database can be naturally seen as a finite set of facts.

We can now naturally define two-sorted queries as functions that map two-sorted databases to *finite* sets of tuples of the same type over $\mathsf{Const} \cup \mathsf{Num}$. For a two-sorted schema $\mathbf{S}$, we write $\mathrm{Inst}(\mathbf{S})$ for the set of all databases of $\mathbf{S}$.

Definition 34.3: Two-Sorted Queries

Consider a two-sorted database schema $\mathbf{S}$. A *query of type* $(\tau_1, \ldots, \tau_k) \in \{\mathsf{o}, \mathsf{n}\}^k$, for $k \geq 0$, over $\mathbf{S}$ is a function of the form

$$q \ : \ \mathrm{Inst}(\mathbf{S}) \to \mathcal{P}_{\mathrm{fin}}((\mathsf{Const} \cup \mathsf{Num})^k)$$

such that, for every $D \in \mathrm{Inst}(\mathbf{S})$ with $q(D) = (a_1, \ldots, a_k)$, $a_i \in \mathsf{Const}$ if $\tau_i = \mathsf{o}$, and $a_i \in \mathsf{Num}$ if $\tau_i = \mathsf{n}$, for every $i \in [k]$.

Regarding the numerical domain Num, we assume that it comes with:

- *(Numerical) predicates*: a predicate P or arity $k > 0$ over Num is a subset of Num^k. For brevity, we say that $P(a_1, \ldots, a_k)$ holds if $(a_1, \ldots, a_k) \in P$. Examples of binary predicates are $=$ and $<$.
- *(Numerical) functions*: a function of arity $k > 0$ over Num is a function of the form $f : \mathsf{Num}^k \to \mathsf{Num}$. Examples of binary functions are $+$ and $\times$.
- *Aggregate functions* or *aggregates*: an aggregate $\mathcal{F}$ over Num is a function that maps bags (or multisets) of elements of Num to Num. In a bag, unlike a set, an element can appear multiple times; e.g., $\{\!|1, 1, 2, 4|\!\}$ is a bag that has two occurrences of 1, and one occurrence of 2 and 4. We shall use the brackets $\{\!|\ |\!\}$ to distinguish bags from sets.

Let us stress that aggregates must be applied to bags rather than sets since values in databases can repeat. Here is a simple example that illustrates this.

Example 34.4: Aggregates over Bags

Consider the two-sorted database

$$D \ = \{R(a, 1), R(b, 1), R(c, 2), R(d, 4)\}.$$

It is clear that the second column of R^D is the bag

$$\{\!|1, 1, 2, 4|\!\}.$$

Assume now that we are interested in computing the average of the numerical values occurring in the second column of R^D. Applying the aver-

age aggregate to $\{\!|1,1,2,4|\!\}$ will correctly produce the value 2. However, if we apply it to the set $\{1,2,4\}$, we get an incorrect value $2.333\cdots$.

Syntax of RA with Aggregates and Grouping

Assuming a set Ω of predicates, functions, and aggregates over the numerical domain Num, we are going to define *relational algebra with aggregates and grouping*, denoted $\mathrm{RA}_{\mathsf{Aggr}}(\Omega)$. Its expressions, unlike expressions of RA, will be *typed*. The type of an expression is again a tuple over $\{\mathsf{o},\mathsf{n}\}$ indicating which attributes of the output are ordinary and which are numerical. In addition to the usual operations, we add the operations of selection based on numerical predicates, applying functions, and applying aggregates.

Before giving the formal definition of $\mathrm{RA}_{\mathsf{Aggr}}(\Omega)$, we first need to introduce some auxiliary notions. For a tuple $\boldsymbol{\tau} = (\tau_1,\ldots,\tau_k) \in \{\mathsf{o},\mathsf{n}\}^k$, where $k \in \mathbb{N}$, we inductively define $(\Omega,\boldsymbol{\tau})$-*terms*, and their associated *types*, as follows:

- Every $a \in \mathsf{Const}$ is an $(\Omega,\boldsymbol{\tau})$-term of type o.
- Every integer $i \in [k]$ is an $(\Omega,\boldsymbol{\tau})$-term of type τ_i.
- If f is an m-ary numerical function from Ω, and $t_1,\ldots,t_m$ are $(\Omega,\boldsymbol{\tau})$-terms of type n, then $f(t_1,\ldots,t_m)$ is an $(\Omega,\boldsymbol{\tau})$-term of type n.

We write $\tau(t)$ for the type of an $(\Omega,\boldsymbol{\tau})$-term t, and $\textsc{Var}(t)$ for the set of all integers that appear in t. An $(\Omega,\boldsymbol{\tau})$-*condition* θ is a Boolean combination of statements of the form $i \doteq j$ and $i \not\doteq j$, for $i,j \in [k]$, and $P(i_1,\ldots,i_m)$, where $i_1,\ldots,i_m \in [k]$ and P is an m-ary predicate from Ω.

Definition 34.5: Syntax of RA with Aggregates and Grouping

Consider a set Ω of predicates, functions, and aggregates over Num. We inductively define $RA_{\mathsf{Aggr}}(\Omega)$ *expressions* over a two-sorted schema $\mathbf{S}$, and their associated *types*, as follows:

Base Expression. If $R : \boldsymbol{\tau}$ belongs to $\mathbf{S}$, then R is an $\mathrm{RA}_{\mathsf{Aggr}}(\Omega)$ expression over $\mathbf{S}$ of type $\boldsymbol{\tau}$.

Selection. If e is an $\mathrm{RA}_{\mathsf{Aggr}}(\Omega)$ expression over $\mathbf{S}$ of type $\boldsymbol{\tau}$, and θ is an $(\Omega,\boldsymbol{\tau})$-condition, then $\sigma_\theta(e)$ is an $\mathrm{RA}_{\mathsf{Aggr}}(\Omega)$ expression over $\mathbf{S}$ of type $\boldsymbol{\tau}$.

Projection. If e is an $\mathrm{RA}_{\mathsf{Aggr}}(\Omega)$ expression of type $\boldsymbol{\tau} = (\tau_1,\ldots,\tau_k)$, for $k \geq 0$, and $\alpha = (t_1,\ldots,t_m)$, for $m \geq 0$, is a list of $(\Omega,\boldsymbol{\tau})$-terms, then $\pi_\alpha(e)$ is an $\mathrm{RA}_{\mathsf{Aggr}}(\Omega)$ expression over $\mathbf{S}$ of type $(\tau(t_1),\ldots,\tau(t_m))$.

Cartesian Product. If e_1, e_2 are $\mathrm{RA}_{\mathsf{Aggr}}(\Omega)$ expressions over $\mathbf{S}$ of type $(\tau_1,\ldots,\tau_k)$ and $(\tau'_1,\ldots,\tau'_m)$, for $k,m \geq 0$, respectively, then $(e_1 \times e_2)$ is an $\mathrm{RA}_{\mathsf{Aggr}}(\Omega)$ expression over $\mathbf{S}$ of type $(\tau_1,\ldots,\tau_k,\tau'_1,\ldots,\tau'_m)$.

Aggregation and Grouping. If e is an $\mathrm{RA}_{\mathsf{Aggr}}(\Omega)$ expression over $\mathbf{S}$ of type $\boldsymbol{\tau} = (\tau_1, \ldots, \tau_k)$, for $k \geq 0$, $\alpha = (i_1, \ldots, i_m)$, for $m \geq 0$, is a list of integers from $[k]$, and $t_1, \ldots, t_\ell$, for $\ell \geq 0$, are $(\Omega, \boldsymbol{\tau})$-terms of type n such that $\textsc{Var}(t_i) \cap \{i_1, \ldots, i_m\} = \emptyset$, for each $i \in [\ell]$, then

$$\mathsf{Aggr}_\alpha[\mathcal{F}_1(t_1), \ldots, \mathcal{F}_\ell(t_\ell)](e)$$

where $\mathcal{F}_1, \ldots, \mathcal{F}_\ell$ are aggregates from Ω, is an $\mathrm{RA}_{\mathsf{Aggr}}(\Omega)$ expression over $\mathbf{S}$ of type $(\tau_{i_1}, \ldots, \tau_{i_m}, \mathsf{n}, \ldots, \mathsf{n})$ with n repeated ℓ times.

Union. If e_1, e_2 are $\mathrm{RA}_{\mathsf{Aggr}}(\Omega)$ expressions over $\mathbf{S}$ of type $\boldsymbol{\tau}$, then $(e_1 \cup e_2)$ is an $\mathrm{RA}_{\mathsf{Aggr}}(\Omega)$ expression over $\mathbf{S}$ of type $\boldsymbol{\tau}$.

Difference. If e_1, e_2 are $\mathrm{RA}_{\mathsf{Aggr}}(\Omega)$ expressions over $\mathbf{S}$ of type $\boldsymbol{\tau}$, then $(e_1 - e_2)$ is an $\mathrm{RA}_{\mathsf{Aggr}}(\Omega)$ expression over $\mathbf{S}$ of type $\boldsymbol{\tau}$.

Concerning the base expression in Definition 34.5, observe the difference with the definition of RA (Definition 4.1). In particular, in ordinary RA we have base expressions $\{a\}$ of arity 1, where $a \in \mathsf{Const}$. Thus, one would expect in Definition 34.5 base expressions of the form $\{a\}$ of type o. However, such base expressions would be redundant since every element of Const is a term, and thus, are expressed via expressions of the form $\pi_{(a)}(R)$ of type o.

Semantics of RA with Aggregates and Grouping

The semantics of the standard relational algebra operations is the same as it was presented in Chapter 4. For numerical selection conditions, $P(i_1, \ldots, i_m)$ is true in a tuple $(a_1, \ldots, a_k)$ iff $i_1, \ldots, i_m$ correspond to columns of type n and $P(a_{i_1}, \ldots, a_{i_m})$ holds. For example, $<(1, 3)$ is true in a tuple (a_1, a_2, a_3) iff the first and the third components are numerical and $a_1 < a_3$.

It remains to explain the new generalized projection, and aggregation with grouping. We first provide informal explanations by means of SQL examples.

Example 34.6: Generalized Projection

Projection allows us compute functions on attributes and output them. For example, for $R[A, B, C]$ with $(R, A) \lessdot (R, B) \lessdot (R, C)$, the SQL query

```
SELECT R.A+R.C, R.B*R.C
FROM R
```

is translated into the $\mathrm{RA}_{\mathsf{Aggr}}(\Omega)$ expression of type (n, n)

$$\pi_{(\mathsf{add}(1,3),\mathsf{mult}(2,3))}(R)$$

assuming that Ω contains the binary functions add and mult such that

$$\mathsf{add}(x, y) = x + y \quad \text{and} \quad \mathsf{mult}(x, y) = x \cdot y.$$

Grouping and aggregation can also be explained intuitively by using SQL queries. In particular, $\mathsf{Aggr}_{(i_1,\ldots,i_m)}[\mathcal{F}_1(t_1), \ldots, \mathcal{F}_\ell(t_\ell)](R)$ corresponds to

```
SELECT   i_1, ..., i_m, F_1(t_1), ..., F_ℓ(t_ℓ)
FROM     R
GROUP BY i_1, ..., i_m
```

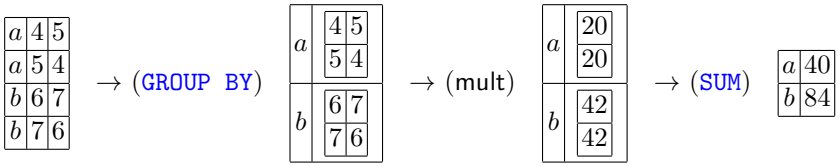
assuming that the attributes of R are $1, \ldots, k$.

Example 34.7: Aggregation and Grouping

For a ternary relation R, the evaluation of

$$\mathsf{Aggr}_{(1)}[\texttt{SUM}(\mathsf{mult}(2, 3))](R)$$

is illustrated below:



The GROUP BY clause does the grouping relative to the first attribute, keeping duplicates if necessary. Then, the term values are computed, again preserving duplicates; for example, for both tuples $(4, 5)$ and $(5, 4)$, the result of the multiplication is 20, and thus two copies are kept. Finally, the aggregate sums up the values of those terms.

We now formally define the semantics of generalized projection, and aggregation with grouping. We first need some auxiliary notions. Consider an $(\Omega, \boldsymbol{\tau})$-term t, where $\boldsymbol{\tau} = (\tau_1, \ldots, \tau_k) \in \{\mathsf{o}, \mathsf{n}\}^k$. A tuple $\bar{a} = (a_1, \ldots, a_n) \in (\mathsf{Const} \cup \mathsf{Num})^n$, for $n \geq k$, is *compatible* with t if, for every $i \in \textsc{Var}(t)$, $\tau_i = \mathsf{o}$ implies $a_i \in \mathsf{Const}$, and $\tau_i = \mathsf{n}$ implies $a_i \in \mathsf{Num}$. We define the *evaluation* of t over a tuple $\bar{a} = (a_1, \ldots, a_n)$ that is compatible with it, denoted $\mathrm{eval}(t|\bar{a})$:

- if $t = a$ with $a \in \mathsf{Const}$, then $\mathrm{eval}(t|\bar{a}) = a$,
- if $t = i$ with $i \in [k]$, then $\mathrm{eval}(t|\bar{a}) = a_i$, and
- if $t = f(t_1, \ldots, t_m)$, then $\mathrm{eval}(t|\bar{a}) = f\big(\mathrm{eval}(t_1|\bar{a}), \ldots, \mathrm{eval}(t_m|\bar{a})\big)$.

Furthermore, given a two-sorted relation R of arity $n \in \mathbb{N}$ consisting of tuples that are compatible with t, a list of integers $\alpha = (i_1, \ldots, i_m)$, for $m \geq 0$, from $[n]$, and a tuple $\bar{a} \in \pi_\alpha(R)$, we define the bag

$$B(\bar{a}, R, t) \;=\; \{\!|\; \mathrm{eval}(t|\bar{c}) \mid \bar{c} \in R \text{ and } \bar{a} = \pi_{\alpha}(\bar{c}) \;|\!\}\,.$$

For instance, going back to Example 34.7, for $t = \mathsf{mult}(2,3)$,

$$B((a), R, t) \;=\; \{\!|\; \mathrm{eval}(t|(a,4,5)), \mathrm{eval}(t|(a,5,4)) \;|\!\} \;=\; \{\!|\; 20, 20 \;|\!\}\,.$$

Definition 34.8: Semantics of Projection and Aggregation

Consider a set Ω of predicates, functions, and aggregates over Num. Let D be a database of a two-sorted schema $\mathbf{S}$, and e an $\mathrm{RA}_{\mathsf{Aggr}}(\Omega)$ over $\mathbf{S}$. We define the *output* $e(D)$ of e on D as follows:

- If $e = \pi_{(t_1,\ldots,t_m)}(e_1)$, where e_1 is an $\mathrm{RA}_{\mathsf{Aggr}}(\Omega)$ expression, then

$$e(D) \;=\; \{(\mathrm{eval}(t_1|\bar{a}), \ldots, \mathrm{eval}(t_m|\bar{a})) \;\mid\; \bar{a} \in e_1(D)\}\,.$$

- If $e = \mathsf{Aggr}_{(i_1,\ldots,i_m)}[\mathcal{F}_1(t_1), \ldots, \mathcal{F}_\ell(t_\ell)](e_1)$, where e_1 is an $\mathrm{RA}_{\mathsf{Aggr}}(\Omega)$ expression, then

$$\big\{\big(\bar{a}, \mathcal{F}_1\big(B(\bar{a}, R, t_{j_1})\big), \ldots, \mathcal{F}_\ell\big(B(\bar{a}, R, t_{j_\ell})\big)\big) \;\big|\; \bar{a} \in \pi_{(i_1,\ldots,i_m)}(e_1(D))\big\}\,.$$

It is clear that $\mathrm{RA}_{\mathsf{Aggr}}(\Omega)$ expressions readily define queries over two-sorted schemas. Indeed, if e is an $\mathrm{RA}_{\mathsf{Aggr}}(\Omega)$ expression, then the *output* of e on a two-sorted database D is $e(D)$. We thus may refer to e as a *query*.

Here is another example, slightly more involved than the ones given above, of expressing an SQL query as an RA query with aggregates and grouping.

Example 34.9: RA with Aggregates and Grouping

For $R[A, B, C]$ with $(R, A) \lessdot (R, B) \lessdot (R, C)$, the SQL query

```
SELECT R.A, SUM(R.B), AVG(R.B*R.C)
FROM R
GROUP BY R.A
HAVING SUM(R.B) > AVG(R.C)
```

is translated into the $\mathrm{RA}_{\mathsf{Aggr}}(\Omega)$ query

$$\pi_{(1,2,4)}\left(\sigma_{2>3}\left(\mathsf{Aggr}_{(1)}\left[\mathtt{SUM}(2), \mathtt{AVG}(3), \mathtt{AVG}(\mathsf{mult}(2,3))\right](R)\right)\right)$$

assuming Ω contains the predicate $<$, the numerical function mult, and the aggregate functions $\mathtt{SUM}$ and $\mathtt{AVG}$ with the obvious meaning. The aggregate expression groups by the first attribute, and computes the sum of the second, and the averages of the third and the product of the second and the third attributes, which then become the second, third,

and fourth attributes. The selection $\sigma_{2>3}$ enforces the condition in the `HAVING` clause, and the projection outputs the attributes listed in `SELECT`.

Complexity of RA with Aggregates and Grouping

We proceed to study the complexity of evaluating $\mathrm{RA}_{\mathsf{Aggr}}(\Omega)$ queries for some set Ω of predicates, functions, and aggregates over Num. Note that the query evaluation problem for $\mathrm{RA}_{\mathsf{Aggr}}(\Omega)$ is defined in a slightly different way than the query evaluation problem for the query languages that we have seen so far. In particular, the input database is two-sorted, while the candidate tuple mentions both constants from Const and values from Num.

Problem: $\mathrm{RA}_{\mathsf{Aggr}}(\Omega)$-Evaluation

Input: A query e from $\mathrm{RA}_{\mathsf{Aggr}}(\Omega)$, a two-sorted database D, a tuple $\bar{a}$ over $\mathsf{Const} \cup \mathsf{Num}$

Output: `true` if $\bar{a} \in e(D)$, and `false` otherwise

We can also talk about the data complexity of $\mathrm{RA}_{\mathsf{Aggr}}(\Omega)$-Evaluation. As discussed in Chapter 2, when we study the data complexity of query evaluation, we essentially consider the query to be fixed, and only the database and the candidate output are part of the input. Formally, we are interested in the complexity of e-Evaluation for an $\mathrm{RA}_{\mathsf{Aggr}}(\Omega)$ query e, which takes as input a two-sorted database D and a tuple $\bar{a}$ over $\mathsf{Const} \cup \mathsf{Num}$, and asks whether $\bar{a} \in e(D)$. As usual, $\mathrm{RA}_{\mathsf{Aggr}}(\Omega)$-Evaluation is in $\mathcal{C}$ in data complexity, for some complexity class $\mathcal{C}$, if e-Evaluation is in $\mathcal{C}$ for every $\mathrm{RA}_{\mathsf{Aggr}}(\Omega)$ query e.

Given an $\mathrm{RA}_{\mathsf{Aggr}}(\Omega)$ query e, to check whether $\bar{a} \in e(D)$, we actually need to compute $e(D)$. Indeed, the only way to check if a numerical value equals the output of an aggregate function is to compute the entire bag of values to which the aggregate is applied. It is clear that the complexity of computing $e(D)$, and therefore, the complexity of $\mathrm{RA}_{\mathsf{Aggr}}(\Omega)$-Evaluation, heavily relies on how complex is to compute predicates, functions, and aggregates from Ω. We concentrate on predicates, functions, and aggregates that are easily computable, and show that, although evaluating $\mathrm{RA}_{\mathsf{Aggr}}(\Omega)$ queries is in general intractable, it becomes tractable when we focus on data complexity.

We say that a k-ary predicate P is computable in polynomial time if, for a tuple $\bar{a} \in \mathsf{Num}^k$, we can check in polynomial time whether $P(\bar{a})$ holds. We also say that that a k-ary function f is computable in polynomial time if $f(\bar{a})$ can be computed in polynomial time. Analogously, we can talk about aggregates that are computable in polynomial time. By a simple inspection of each operation of $\mathrm{RA}_{\mathsf{Aggr}}(\Omega)$, it is not difficult to show the following result:

Theorem 34.10

Consider a set Ω of predicates, functions, and aggregates over Num that are computable in polynomial time. Then $\mathsf{RA}_{\mathsf{Aggr}}(\Omega)$-Evaluation is in ExpTime, and in PTime in data complexity.

35

Inexpressibility of Recursive Queries

Aggregation and grouping are powerful features that allow us to express interesting counting properties. There are some common queries nonetheless that cannot be expressed even if aggregation and grouping are available. These are queries requiring recursive computation, the subject of Chapters 36 – 39. A canonical query of this type is reachability in directed graphs, i.e., given a directed graph G, compute all the pairs of nodes (u, v) in G such that v is reachable from u. This query can be computed by the following simple algorithm: a node v is reachable from a node u if

1. there is an edge from u to v, or
2. there is an edge from u to some node w such that v is reachable from w.

This algorithm is indeed recursive since the second item defines reachability in terms of itself. Such a description allows us to extract arbitrarily long paths from the input graph. For example, in the graph with nodes $\{0, \ldots, n\}$, and edges $(i, i+1)$ for all $i \in [n-1]$, a node j is reachable from i iff $i < j$. Thus, the reachability query will extract paths of any length from 1 to n. On the other hand, we can show that no matter how we express the reachability query as an RA query e with aggregates and grouping, e will be able to extract paths up to a certain length that solely depends on e itself. This essentially tells us that RA with aggregates and grouping cannot express the reachability query. Our goal is to show this fact. We start by defining the reachability query.

Definition 35.1: The Reachability Query

The *reachability query*, denoted q_{reach}, over $\mathbf{S} = \{E : (\mathsf{o}, \mathsf{o})\}$ is defined as the query of type (o, o) such that, for every database D of $\mathbf{S}$, $(a, b) \in q_{\mathsf{reach}}(D)$ iff $E(a, b) \in D$, or there are constants $c_1, \ldots, c_n \in \mathrm{Dom}(D)$, for $n > 0$, such that $\{E(a, c_1), E(c_1, c_2), \ldots, E(c_{n-1}, c_n), E(c_n, b)\} \subseteq D$.

We can then show the following inexpressibility result:

Theorem 35.2

Consider a numerical domain Num, and a set Ω of predicates, functions, and aggregates over Num. There is no $\mathrm{RA}_{\mathsf{Aggr}}(\Omega)$ query e over the schema $\mathbf{S} = \{E : (\mathsf{o}, \mathsf{o})\}$ such that $q_{\mathsf{reach}} \equiv e$.

The rest of the chapter is devoted to showing the above theorem. Actually, our main task is to prove a more powerful result, that is, we are going to establish a locality property stating that an $\mathrm{RA}_{\mathsf{Aggr}}(\Omega)$ query can only "see" up to a certain distance in the input database, determined by the query itself, and thus is not be able to extract paths of arbitrarily many different lengths as the reachability query does. Let us make this idea more precise.

Locality of Queries

Given a two-sorted database D, its *Gaifman graph*, denoted G_D, is an undirected graph whose nodes are the elements of $\mathrm{Dom}(D)$, and whose edges are $\{a, b\}$ such that a and b appear together in some fact of D, i.e., there is a fact of the form $R(\ldots, a, \ldots, b, \ldots)$ in D. Given $a, b \in \mathrm{Dom}(D)$, the *distance* of a and b in D, denoted $d_D(a, b)$, is the length of the shortest path in G_D from a to b; by convention, $d_D(a, a) = 0$, and $d_D(a, b) = \infty$ if there is no path in G_D from a to b. For a tuple $\bar{a} = (a_1, \ldots, a_n)$ over $\mathrm{Dom}(D)$, the distance of $\bar{a}$ and b in D, denoted $d_D(\bar{a}, b)$, is $\min_{i \in [n]}\{d_D(a_i, b)\}$. The *radius-$r$ ball* of $\bar{a}$ in D, for $r \geq 0$, denoted $B_r^D(\bar{a})$, is the set $\{b \in \mathrm{Dom}(D) \mid d_D(a_i, b) \leq r \text{ for } i \in [n]\}$. The *radius-$r$ neighborhood* (or simply *r-neighborhood*) of $\bar{a}$ in D, denoted $N_r^D(\bar{a})$, is the set of facts $\{R(\bar{b}) \in D \mid \bar{b} \text{ is over } B_r^D(\bar{a})\}$, i.e., the set of all facts of D that contain only elements of $B_r^D(\bar{a})$. The tuple $\bar{a}$ should be understood as a tuple of distinguished elements in $N_r^D(\bar{a})$. We say that two r-neighborhoods $N_r^D(\bar{a})$ and $N_r^D(\bar{b})$ are *isomorphic* if there exists a bijection $h : \mathrm{Dom}(N_r^D(\bar{a})) \to \mathrm{Dom}(N_r^D(\bar{b}))$ such that $h(\bar{a}) = \bar{b}$, and $R(\bar{c}) \in N_r^D(\bar{a})$ implies $R(h(\bar{c})) \in N_r^D(\bar{b})$. We can now define the notion of locality for queries.

Definition 35.3: Locality of Queries

A query q of type $\boldsymbol{\tau} \in \{\mathsf{o}, \mathsf{n}\}^k$, for $k \geq 0$, over a two-sorted schema $\mathbf{S}$ is *r-local*, for $r \geq 0$, if, for every database D of $\mathbf{S}$, and every two tuples $\bar{a}, \bar{b} \in (\mathsf{Const} \cup \mathsf{Num})^k$ such that $N_r^D(\bar{a})$ and $N_r^D(\bar{b})$ are isomorphic, $\bar{a} \in q(D)$ iff $\bar{b} \in q(D)$. A query is called *local* if it is r-local for some $r \geq 0$.

We proceed to show that, as long as we concentrate on relations of ordinary type, RA queries with aggregates and grouping of ordinary type, that is, queries that output only constants of Const, are local.

Theorem 35.4

Consider a numerical domain Num, and a set Ω of predicates, functions, and aggregates over Num. Every $\mathrm{RA}_{\mathsf{Aggr}}(\Omega)$ query of type $\{\mathsf{o}\}^k$, for $k \geq 0$, over a schema $\mathbf{S} = \{R_1 : \boldsymbol{\tau}_1, \ldots, R_n : \boldsymbol{\tau}_n\}$ such that, for each $i \in [n]$, $\boldsymbol{\tau}_i \in \{\mathsf{o}\}^{k_i}$ for $k_i \geq 0$, is local.

Note that Theorem 35.4 only talks about relations of ordinary type. What if instead we consider arbitrary two-sorted relations that can have values from Num? In this case there is no known result analogous to Theorem35.4. In fact, proving such a result would resolve deep open problems in complexity theory (see Exercise 4.18 for further explanations).

Before we give the proof of Theorem 35.4, let us explain how it is used in order to obtain Theorem 35.2. It actually tells us that to show Theorem 35.2 it suffices to show that the reachability query q_{reach} is not local. By contradiction, assume that q_{reach} is r-local, for some $r \geq 0$. Consider then the database D of $\mathbf{S} = \{E : (\mathsf{o}, \mathsf{o})\}$ consisting of the facts $E(i, i+1)$ for each $i \in [0, 5r]$. Then, the r-neighborhoods $N_r^D(r, 4r)$ and $N_r^D(4r, r)$ are isomorphic. Indeed, each of these r-neighborhoods is a disjoint union of two chains of length $2r$, with the distinguished elements in the middle of those chains. Therefore, by locality, $(r, 4r) \in q_{\mathsf{reach}}(D)$ iff $(4r, r) \in q_{\mathsf{reach}}(D)$. This contradicts the fact that $4r$ is reachable from r, but not vice versa; in other words, $(r, 4r) \in q_{\mathsf{reach}}(D)$ and $(4r, r) \notin q_{\mathsf{reach}}(D)$. Therefore, q_{reach} is not local, as needed.

The rather long proof of Theorem 35.4 consists of three main steps:

FO with Aggregates. We first present an extension of FO with aggregates, and show that RA with aggregates and grouping translates to it. Note that this logical language is a useful technical tool for showing Theorem 35.4, but it cannot serve as the basis for defining a query language with aggregates; further details are given below.

Counting Logic. We then express an RA query with aggregates and grouping, via its translation into the extension of FO with aggregates, in an infinitary counting logic that is easier to analyze mathematically.

Locality of Counting Logic. We finally prove that the counting logic enjoys a locality property analogous to that for $\mathrm{RA}_{\mathsf{Aggr}}(\Omega)$ queries stated in Theorem 35.4, which in turn implies Theorem 35.4 itself.

For the sake of clarity, the second and third steps are focussing on queries and logical formulae that do not mention ordinary constants from Const. Extending the proof to handle constants is the subject of Exercise 4.14.

First-Order Logic with Aggregates

Considering the set Ω of predicates, functions, and aggregates over the numerical domain Num, we are going to define *first-order logic with aggregates*,

denoted $\mathrm{FO}_{\mathsf{Aggr}}(\Omega)$. Similarly to $\mathrm{RA}_{\mathsf{Aggr}}(\Omega)$, the logical language $\mathrm{FO}_{\mathsf{Aggr}}(\Omega)$ must be *typed*, in particular, each variable should come with its type. We therefore assume that the set Var is partitioned into two infinite sets $\mathsf{Var}_{\mathsf{o}}$ and $\mathsf{Var}_{\mathsf{n}}$ of variables of type o and n, respectively. We use $x, y, z, \ldots$ for variables from $\mathsf{Var}_{\mathsf{o}}$, which will be ranging over the set of constants Const, and $\imath, \jmath, \ldots$ for variables from $\mathsf{Var}_{\mathsf{n}}$, which will be ranging over the numerical domain Num.

Definition 35.5: Syntax of FO with Aggregates

Consider a two-sorted schema $\mathbf{S}$. We define *Ω-terms* (relative to $\mathbf{S}$) with their associated *types*, and $\mathrm{FO}_{\mathsf{Aggr}}(\Omega)$ *formulae* over $\mathbf{S}$, by mutual induction as follows:

Ω-terms

- Each constant of Const and variable of $\mathsf{Var}_{\mathsf{o}}$ is an Ω-term of type o.
- Each value $a \in \mathsf{Num}$ is an Ω-term of type n, whose set of free variables $\mathrm{FV}(a)$ is empty.
- Each variable $\imath \in \mathsf{Var}_{\mathsf{n}}$ is an Ω-term of type n, whose set of free variables $\mathrm{FV}(\imath)$ is $\{\imath\}$.
- If f is an m-ary numerical function from Ω, and $t_1, \ldots, t_m$ are Ω-terms of type n, then $f(t_1, \ldots, t_m)$ is an Ω-term of type n, whose set of free variables $\mathrm{FV}(f(t_1, \ldots, t_m))$ is $\mathrm{FV}(t_1) \cup \cdots \cup \mathrm{FV}(t_m)$.
- If φ is an $\mathrm{FO}_{\mathsf{Aggr}}(\Omega)$ formula over $\mathbf{S}$, t is an Ω-term of type n, and $\mathcal{F}$ is an aggregate of Ω, then

$$\mathsf{Aggr}_{\mathcal{F}}(\bar{u})\,(\varphi, t)$$

  where $\bar{u} = (u_1, \ldots, u_k)$ is a tuple of variables over Var such that $\{u_1, \ldots, u_k\} \subseteq \mathrm{FV}(t)$, and $\mathrm{FV}(t) \subseteq \mathrm{FV}(\varphi)$, is an Ω-term of type n, whose set of free variables $\mathrm{FV}(\mathsf{Aggr}_{\mathcal{F}}(\bar{u})\,(\varphi, t))$ is $\mathrm{FV}(\varphi) - \{u_1, \ldots, u_k\}$.

$\mathrm{FO}_{\mathsf{Aggr}}(\Omega)$ Formulae

- If $a \in \mathsf{Const}$, and $x \in \mathsf{Var}_{\mathsf{o}}$, then $x = a$ is an atomic formula, whose set of variables $\mathrm{FV}(x = a)$ is $\{x\}$.
- If $x, y \in \mathsf{Var}_{\mathsf{o}}$, then $x = y$ is an atomic formula, whose set of free variables $\mathrm{FV}(x = y)$ is $\{x, y\}$.
- If t is an Ω-term of type n, and $\imath \in \mathsf{Var}_{\mathsf{n}}$, then $\imath = t$ is an atomic formula, whose set of free variables $\mathrm{FV}(\imath = t)$ is $\{\imath\} \cup \mathrm{FV}(t)$.
- If $u_1, \ldots, u_k$ are Ω-terms (not necessarily distinct) from $\mathsf{Const} \cup \mathsf{Num} \cup \mathsf{Var}$, where u_i is of type τ_i for each $i \in [k]$, and $R : (\tau_1, \ldots, \tau_k)$

belongs to $\mathbf{S}$, then $R(u_1, \ldots, u_k)$ is an atomic formula, whose set of free variables $\mathrm{FV}(R(u_1, \ldots, u_k))$ is $\{u_1, \ldots, u_k\} \cap \mathsf{Var}$.

- If $u_1, \ldots, u_k$ are Ω-terms (not necessarily distinct) from $\mathsf{Num} \cup \mathsf{Var}_\mathsf{n}$, and P is a k-ary numerical predicate from Ω, then $P(u_1, \ldots, u_k)$ is an atomic formula, whose set of free variables $\mathrm{FV}(P(u_1, \ldots, u_k))$ is $\{u_1, \ldots, u_k\} \cap \mathsf{Var}_\mathsf{n}$.
- If φ_1 and φ_2 are formulae, then $(\varphi_1 \wedge \varphi_2)$ and $(\varphi_1 \vee \varphi_2)$ are formulae, whose set of free variables $\mathrm{FV}(\varphi_1 \wedge \varphi_2) = \mathrm{FV}(\varphi_1 \vee \varphi_2)$ is $\mathrm{FV}(\varphi_1) \cup \mathrm{FV}(\varphi_2)$.
- If φ is a formula, then $(\neg\varphi)$ is a formula, whose set of free variables $\mathrm{FV}(\neg\varphi)$ is $\mathrm{FV}(\varphi)$.
- If φ is a formula and $u \in \mathsf{Var}$, then $(\exists u\, \varphi)$ and $(\forall u\, \varphi)$ are formulae, whose set of free variables $\mathrm{FV}(\exists u\, \varphi) = \mathrm{FV}(\forall u\, \varphi)$ is $\mathrm{FV}(\varphi) - \{u\}$.

We will omit the outermost brackets of $\mathrm{FO}_{\mathsf{Aggr}}(\Omega)$ formulae. To define the semantics of $\mathrm{FO}_{\mathsf{Aggr}}(\Omega)$, we need the notion of assignment for Ω-terms and formulae. Given an Ω-term t of type n, an *assignment η for t* is a function from $\mathrm{FV}(t)$ to $\mathsf{Const} \cup \mathsf{Num}$ such that $\eta(u) \in \mathsf{Const}$ if $u \in \mathsf{Var}_\mathsf{o}$, and $\eta(u) \in \mathsf{Num}$ if $u \in \mathsf{Var}_\mathsf{n}$. Similarly, given an $\mathrm{FO}_{\mathsf{Aggr}}(\Omega)$ formula φ, and a database D, an *assignment η for φ over D* is a function from $\mathrm{FV}(\varphi)$ to $\mathrm{Dom}(D) \cup \mathrm{Dom}(\varphi) \cup \mathsf{Num}$, where $\mathrm{Dom}(\varphi)$ is the set of constants and numerical values occurring in φ, such that $\eta(u) \in \mathsf{Const}$ if $u \in \mathsf{Var}_\mathsf{o}$, and $\eta(u) \in \mathsf{Num}$ if $u \in \mathsf{Var}_\mathsf{n}$. We write $\eta[u/a]$, for $u \in \mathsf{Var}$ and $a \in \mathsf{Const} \cup \mathsf{Num}$, for the assignment that modifies η by setting $\eta(u) = a$. To avoid heavy notation, we extend an assignment to be the identity on $\mathsf{Const} \cup \mathsf{Num}$. The semantics of $\mathrm{FO}_{\mathsf{Aggr}}(\Omega)$ follows.

Definition 35.6: Semantics of FO with Aggregates

Consider a two-sorted schema $\mathbf{S}$. We define the value of Ω-terms (relative to $\mathbf{S}$) of type n, and the satisfaction of $\mathrm{FO}_{\mathsf{Aggr}}(\Omega)$ formulae over $\mathbf{S}$, by mutual induction as follows.

Value of Ω-terms

Let t be an Ω-term of type n, and η an assignment for t. The *value* of t in a database D of $\mathbf{S}$ under η, denoted $t^{D,\eta}$, is defined as follows:

- If $t = a$ with $a \in \mathsf{Num}$, then $t^{D,\eta} = a$.
- If $t = \imath$ with $\imath \in \mathsf{Var}_\mathsf{n}$, then $t^{D,\eta} = \eta(\imath)$.
- If $t = f(t_1, \ldots, t_m)$, then $t^{D,\eta} = f(t_1^{D,\eta}, \ldots, t_m^{D,\eta})$.
- If $t = \mathsf{Aggr}_{\mathcal{F}}(\bar{u})\,(\varphi, t_0)$, with H being the set of all assignments η' for φ over D that agree with η on $\mathrm{FV}(t_0)$ such that $(D, \eta') \models \varphi$, then $t^{D,\eta} = \mathcal{F}(\{\!|\}\!|)$ if H is infinite, otherwise $t^{D,\eta} = \mathcal{F}\left(\{\!|t_0^{D,\eta'} \mid \eta' \in H|\}\!|\right)$.

Satisfaction of $\mathrm{FO}_{\mathsf{Aggr}}(\Omega)$ Formulae

Let φ be a formula over $\mathbf{S}$, and η an assignment for φ. We define when φ is *satisfied* in a database D of $\mathbf{S}$ under η, written $(D, \eta) \models \varphi$, as follows:

- If φ is $x = a$, then $(D, \eta) \models \varphi$ if $\eta(x) = a$.
- If φ is $x = y$, then $(D, \eta) \models \varphi$ if $\eta(x) = \eta(y)$.
- If φ is $\imath = t$, then $(D, \eta) \models \varphi$ if $\eta(x) = t^{D,\eta}$.
- If φ is $R(u_1, \ldots, u_k)$, then $(D, \eta) \models \varphi$ if $R(\eta(u_1), \ldots, \eta(u_k)) \in D$.
- If φ is $P(u_1, \ldots, u_k)$, then $(D, \eta) \models \varphi$ if $(\eta(u_1), \ldots, \eta(u_k)) \in P$.
- If $\varphi = \varphi_1 \wedge \varphi_2$, then $(D, \eta) \models \varphi$ if $(D, \eta) \models \varphi_1$ and $(D, \eta) \models \varphi_2$.
- If $\varphi = \varphi_1 \vee \varphi_2$, then $(D, \eta) \models \varphi$ if $(D, \eta) \models \varphi_1$ or $(D, \eta) \models \varphi_2$.
- If $\varphi = \neg\psi$, then $(D, \eta) \models \varphi$ if $(D, \eta) \models \psi$ does not hold.
- If $\varphi = \exists u\, \psi$, then $(D, \eta) \models \varphi$ if $(D, \eta[u/a]) \models \psi$ for *some* $a \in (\mathrm{Dom}(D) \cup \mathrm{Dom}(\varphi)) \cap \mathsf{Const}$ if $u \in \mathsf{Var_o}$, and $a \in \mathsf{Num}$ if $u \in \mathsf{Var_n}$.
- If $\varphi = \forall x\, \psi$, then $(D, \eta) \models \varphi$ if $(D, \eta[x/a]) \models \psi$ for *each* $a \in \mathrm{Dom}(D) \cup (\mathrm{Dom}(\varphi) \cap \mathsf{Const})$ if $u \in \mathsf{Var_o}$, and $a \in \mathsf{Num}$ if $u \in \mathsf{Var_n}$.

There is a crucial difference between first-order logic with aggregates, and first-order logic as defined in Chapter 3. Given an $\mathrm{FO}_{\mathsf{Aggr}}(\Omega)$ formula φ, we may have infinitely many assignments η for φ over a database D such that $(D, \eta) \models \varphi$. Consider, for example, the formula $\varphi = R(x) \wedge (\jmath = \imath + 1)$, and the database $D = \{R(a)\}$ with $a \in \mathsf{Const}$. Assuming that Num is the set of integers, for every $\eta : \{x, \imath, \jmath\} \to \{a\} \cup \mathsf{Num}$ with $\eta(x) = a$ and $\eta(\jmath) = \eta(\imath) + 1$, $(D, \eta) \models \varphi$, and it is clear that there are infinitely many such assignments since the numerical variables range over the infinite domain Num, and there are infinitely many pairs of integers (i, j) such that $j = i + 1$. This implies that first-order logic with aggregates cannot be used to define a query language as we did with ordinary FO in Chapter 3. Nevertheless, given an expression $\varphi(\bar{u})$, where φ is an $\mathrm{FO}_{\mathsf{Aggr}}(\Omega)$ formula, and $\bar{u}$ is a tuple over $\mathrm{FV}(\varphi)$ such that each free variable of φ occurs in $\bar{u}$ at least once, we can define the *output* of $\varphi(\bar{u})$ on a database D, denoted $\varphi(\bar{u})(D)$, in the obvious way, but we cannot call $\varphi(\bar{u})$ a query since $\varphi(\bar{u})(D)$ may be infinite. Indeed, the output of $\varphi(x, \imath, \jmath)$, where $\varphi = R(x) \wedge (\jmath = \imath + 1)$, on $D = \{R(a)\}$ is the infinite set of triples (a, i, j) with $j = i + 1$. One can define a logic with aggregates that can in turn be used to define a query language, but the syntax is much more cumbersome.

We now show that that every RA query with aggregates and grouping can be expressed via FO with aggregates.

Proposition 35.7

Consider an $\mathrm{RA}_{\mathsf{Aggr}}(\Omega)$ query e over a two-sorted schema $\mathbf{S}$. There exists an $\mathrm{FO}_{\mathsf{Aggr}}(\Omega)$ formula φ_e over $\mathbf{S}$, and a tuple $\bar{u}_e$ over $\mathrm{FV}(\varphi_e)$ that

mentions all the variables of $\mathrm{FV}(\varphi_e)$, such that $e(D) = \varphi_e(\bar{u}_e)(D)$, for every database D of $\mathbf{S}$.

Proof. The proof is by induction on the structure of e. Most of the cases are treated in the same way as in the proof of Theorem 6.1, in particular, the proof that every RA query can be equivalently written as an FO query. We proceed to discuss the two new case, namely generalized projection, and grouping with aggregation. For an $(\Omega, \boldsymbol{\tau})$-term t, where $\boldsymbol{\tau} = (\tau_1, \ldots, \tau_k) \in \{\mathsf{o}, \mathsf{n}\}^k$, and a tuple of variables $\bar{u} = (u_1, \ldots, u_k)$ with u_i being of type τ_i, for each $i \in [k]$, we write $t(\bar{u})$ for the Ω-term obtained from t by replacing each $i \in \textsc{Var}(t)$ with u_i. We are now ready to proceed with the translations:

- Assume first that e is $\pi_{(t_1,\ldots,t_m)}(e')$. By induction hypothesis, there exists an $\mathrm{FO}_{\mathsf{Aggr}}(\Omega)$ formula $\varphi_{e'}$, and a tuple $\bar{u}_{e'} = (u_1, \ldots, u_k)$ over $\mathrm{FV}(\varphi_{e'})$, such that $\varphi_{e'}(\bar{u}_{e'})$ expresses e'. We define the $\mathrm{FO}_{\mathsf{Aggr}}(\Omega)$ formula
$$\varphi_e \;=\; \exists u_1 \cdots \exists u_k \left(\varphi_{e'} \;\wedge\; \bigwedge_{i=1}^{m} v_i = t_i(\bar{u}_{e'}) \right),$$
and the tuple $\bar{u}_e = (v_1, \ldots, v_m)$ over $\mathrm{FV}(\varphi_e)$.
- Assume now that e is $\mathsf{Aggr}_{(i_1,\ldots,i_m)}[\mathcal{F}_1(t_1), \ldots, \mathcal{F}_\ell(t_\ell)](e')$; for the sake of clarity, we assume that $(i_1, \ldots, i_m) = (1, \ldots, m)$, but the same construction can be applied to any list of integers. By induction hypothesis, there exists an $\mathrm{FO}_{\mathsf{Aggr}}(\Omega)$ formula $\varphi_{e'}$, and a tuple $\bar{u}_{e'} = (u_1, \ldots, u_k)$ over $\mathrm{FV}(\varphi_{e'})$, such that $\varphi_{e'}(\bar{u}_{e'})$ expresses e'. We define the $\mathrm{FO}_{\mathsf{Aggr}}(\Omega)$ formula
$$\varphi_e \;=\; \exists w_{m+1} \cdots \exists w_k \left(\psi_{e'} \;\wedge \right.$$
$$\left. \bigwedge_{i=1}^{\ell} v_i = \mathsf{Aggr}_{\mathcal{F}_i}(w_{m+1}, \ldots, w_k)\,(\psi_{e'}, t_i(u_1, \ldots, u_m, w_{m+1}, \ldots, w_k)) \right),$$
where $\psi_{e'}$ is obtained from $\varphi_{e'}$ by replacing u_i with w_i for each $i \in [m+1, k]$, and the tuple $\bar{u}_e = (u_1, \ldots, u_m, v_1, \ldots, v_\ell)$ over $\mathrm{FV}(\varphi_e)$.

It is easy to verify the correctness of the above translations. □

An Infinitary Counting Logic

We now proceed with the second step of the proof of Theorem 35.2, where the goal is to express an $\mathrm{RA}_{\mathsf{Aggr}}(\Omega)$ query, by exploiting Proposition 35.7, into a convenient infinitary counting logic. We proceed to introduce the counting logic $\mathbf{L}_{\mathbf{C}}$, and its sublogic $\mathbb{L}_{\mathbf{C}}$, which we prove to be equivalent to $\mathbf{L}_{\mathbf{C}}$. Recall that, for the sake of clarity, in this step we focus on queries and formulae that do not mention constants from Const. In what follows, every value of Num, and every variable of Var, is a term of the respective type, and of rank 0.

Definition 35.8: An Infinitary Counting Logic

Consider a two-sorted schema $\mathbf{S}$. We define *formulae of* $\mathbf{L_C}$ over $\mathbf{S}$, and their associated *rank*, by induction as follows:

- If $x, y \in \mathsf{Var_o}$, then $x = y$ is an atomic formula with $\text{rank}(x = y) = 0$.
- If $a \in \mathsf{Num}$, and $\imath \in \mathsf{Var_n}$, then $\imath = a$ is an atomic formula with $\text{rank}(\imath = a) = 0$.
- If $\imath, \jmath \in \mathsf{Var_n}$, then $\imath = \jmath$ is an atomic formula with $\text{rank}(\imath = \jmath) = 0$.
- If $u_1, \ldots, u_k$ are terms (not necessarily distinct), where u_i is of type τ_i for each $i \in [k]$, and $R : (\tau_1, \ldots, \tau_k)$ belongs to $\mathbf{S}$, then $R(u_1, \ldots, u_k)$ is an atomic formula with $\text{rank}(R(u_1, \ldots, u_k)) = 0$.
- If Φ is a (possibly infinite) set of formulae, and $k = \sup_{\varphi \in \Phi} \text{rank}(\varphi)$ is finite, then $\psi = \left(\bigwedge_{\varphi \in \Phi} \varphi\right)$ and $\psi' = \left(\bigvee_{\varphi \in \Phi} \varphi\right)$ are formulae with $\text{rank}(\psi) = \text{rank}(\psi') = k$.
- If φ is a formula, then $(\neg\varphi)$ is a formula with $\text{rank}(\neg\varphi) = \text{rank}(\varphi)$.
- If φ is a formula, $\bar{u} = (u_1, \ldots, u_k)$ is a tuple over Var, and $n \in \mathbb{N}$, then $\psi = \left(\exists^{\geq n} \bar{u}\, \varphi\right)$ is a formula with $\text{rank}(\psi) = \text{rank}(\varphi) + k$.

We further define *formulae of* $\mathbb{L}_\mathbf{C}$ over $\mathbf{S}$, and their associated *rank*, in the same way as $\mathbf{L_C}$ formulae, with the difference that only quantification of the form $\exists^{\geq n} x$, where x is a single variable type o, is allowed:

- If φ is a formula of $\mathbb{L}_\mathbf{C}$, $x \in \mathsf{Var_o}$, and $n \in \mathbb{N}$, then $\psi = \left(\exists^{\geq n} x\, \varphi\right)$ is a formula of $\mathbb{L}_\mathbf{C}$ with $\text{rank}(\psi) = \text{rank}(\varphi) + 1$.

Let us stress that $\mathbf{L_C}$, and thus $\mathbb{L}_\mathbf{C}$, consists of formulae of finite rank since in the definition of the infinitary conjunctions and disjunctions we explicitly restrict the rank to be finite; otherwise, we may get formulae of infinite rank, e.g., if $\Phi = \{\varphi_1, \varphi_2, \ldots\}$ with $\text{rank}(\varphi_i) = i$ for each $i > 0$, then rank $\left(\bigvee_{\varphi \in \Phi} \varphi\right)$ is infinite. The set of free variables of an $\mathbf{L_C}$ formula φ, denoted $\text{FV}(\varphi)$, as well as the semantics of $\mathbf{L_C}$, are defined in the expected way. Let us only discuss the details in the case of a formula of the form $\psi = \exists^{\geq n} \bar{u}\, \varphi$, which essentially states that there exist at least n witnesses for $\bar{u}$. For an assignment η for ψ over a database D, i.e., a function that maps $\text{FV}(\psi)$ to $\text{Dom}(D) \cup \mathsf{Num}$, ψ is satisfied in D under η, written $(D, \eta) \models \psi$, if there are at least n assignments η' for φ over D that agree with η on $\text{FV}(\psi)$ such that $(D, \eta') \models \varphi$. We can use the shorthand $\exists^{=n} \bar{u}\, \varphi$ to say that there are *exactly* n such assignments for φ over D, that is, $\exists^{\geq n} \bar{u}\, \varphi \wedge \neg \exists^{\geq n+1} \bar{u}\, \varphi$, which does not alter the rank. Given an expression $\varphi(\bar{u})$, where φ is an $\mathbf{L_C}$ formula, and $\bar{u}$ is a tuple over $\text{FV}(\varphi)$ such that each free variable of φ occurs in $\bar{u}$ at least once, we can define the *output* of $\varphi(\bar{u})$ on a database D, denoted $\varphi(\bar{u})(D)$, in the obvious way.

Proposition 35.9

Consider an $\mathrm{FO}_{\mathsf{Aggr}}(\Omega)$ formula φ over a two-sorted schema $\mathbf{S}$, and a tuple $\bar{u}$ over $\mathrm{FV}(\varphi)$ that mentions all the variables of $\mathrm{FV}(\varphi)$. There exists an $\mathbb{L}_{\mathbf{C}}$ formula ψ over $\mathbf{S}$, with $\mathrm{FV}(\varphi) = \mathrm{FV}(\psi)$, such that $\varphi(\bar{u})(D) = \psi(\bar{u})(D)$, for every database D of $\mathbf{S}$.

Proof. We first show the statement for the counting logic $\mathbf{L}_{\mathbf{C}}$.

Lemma 35.10. *There is an $\mathbf{L}_{\mathbf{C}}$ formula $\varphi^{\diamond}$ over $\boldsymbol{S}$, with $FV(\varphi) = FV(\varphi^{\diamond})$, such that $\varphi(\bar{u})(D) = \varphi^{\diamond}(\bar{u})(D)$, for every database D of $\boldsymbol{S}$.*

Proof. We translate every Ω-term t of type n occurring in φ into an $\mathbf{L}_{\mathbf{C}}$ formula $\alpha_t^{\imath}$, where $\imath$ is a distinguished free variable of $\alpha_t^{\imath}$. Intuitively, $\alpha_t^{\imath}$ states that $\imath$ is the value of t, that is, $(D, \eta) \models \alpha_t^{\imath}$ iff $t^{D,\eta} = \eta(\imath)$. We further translate the formula φ into an $\mathbf{L}_{\mathbf{C}}$ formula $\varphi^{\diamond}$ with $\mathrm{FV}(\varphi) = \mathrm{FV}(\varphi^{\diamond})$.

To ensure that $\varphi^{\diamond}$ is indeed an $\mathbf{L}_{\mathbf{C}}$ formula, we need to show that it has finite rank. To this end, we first need to transfer the notion of rank to Ω-terms and $\mathrm{FO}_{\mathsf{Aggr}}(\Omega)$ formulae by mutual induction. Let t be an Ω-term:

- If $t \in \mathsf{Num} \cup \mathsf{Var}$, then $\mathrm{rank}(t) = 0$.
- If $t = f(t_1, \ldots, t_m)$, then $\mathrm{rank}(t) = \max_{i \in [m]}\{\mathrm{rank}(t_i)\}$.
- If $t = \mathsf{Aggr}_{\mathcal{F}}(\bar{v})\,(\varphi', t')$, then $\mathrm{rank}(t) = \max\{\mathrm{rank}(\varphi'), \mathrm{rank}(t')\} + k$, where k is the arity of the tuple $\bar{v}$.

Consider now an $\mathrm{FO}_{\mathsf{Aggr}}(\Omega)$ formula φ':

- If φ' is $x = y$, then $\mathrm{rank}(\varphi') = 0$.
- If φ' is $\imath = t$, then $\mathrm{rank}(\varphi') = \mathrm{rank}(t)$.
- If φ' is $R(v_1, \ldots, v_k)$, then $\mathrm{rank}(\varphi') = \max_{i \in [k]}\{\mathrm{rank}(v_i)\}$.
- If φ' is $\varphi_1 \wedge \varphi_2$ or $\varphi_1 \vee \varphi_2$, then $\mathrm{rank}(\varphi') = \max\{\mathrm{rank}(\varphi_1), \mathrm{rank}(\varphi_2)\}$.
- If φ' is $\neg\varphi_1$, then $\mathrm{rank}(\varphi') = \mathrm{rank}(\varphi_1)$.
- If φ' is $\exists v\, \varphi_1$ or $\forall v\, \varphi_1$, then $\mathrm{rank}(\varphi') = \mathrm{rank}(\varphi_1) + 1$.

We are now ready to provide the translation of an Ω-term t of type n occurring in φ, and of φ itself, into $\mathbf{L}_{\mathbf{C}}$ by mutual induction. In what follows, given an $\mathbf{L}_{\mathbf{C}}$ formula χ, a variable $\imath \in \mathrm{FV}(\chi)$, and a value $a \in \mathsf{Num}$, we write $\chi[\imath/a]$ for the formula obtained from χ after replacing $\imath$ with a.

Translation of an Ω-term t occurring in φ into an $\mathbf{L}_{\mathbf{C}}$ formula $\alpha_t^{\imath}$

- If $t = a$ with $a \in \mathsf{Num}$, then $\alpha_t^{\imath} = (\imath = a)$ of rank 0.
- If $t = \imath$ with $\imath \in \mathsf{Var}_{\mathsf{n}}$, then $\alpha_t^{\imath} = (\imath = \imath)$ of rank 0.

- If $t = f(t_1, \ldots, t_m)$, then

$$\alpha_t^\imath \;=\; \bigvee_{\substack{(a_1,\ldots,a_m,a_{m+1})\in \mathsf{Num}^{m+1}\,:\\ f(a_1,\ldots,a_m)=a_{m+1}}} \left(\bigwedge_{i\in[m]} \alpha_{t_i}^{\jmath_i}[\jmath_i/a_i] \;\rightarrow\; \imath = a_{m+1} \right)$$

  of rank $\max_{i\in[m]}\{\mathrm{rank}(\alpha_{t_i}^{\jmath_i})\} \leq \mathrm{rank}(t)$, where $\imath$ is a new numerical variable not occurring in $\alpha_{t_i}^{\jmath_i}$, for each $i \in [m]$.

- If $t = \mathsf{Aggr}_{\mathcal{F}}(\bar{v})\,(\varphi', t')$, with $\mathcal{B}$ being the set of all bags (finite or infinite) over Num, then

$$\alpha_t^\imath \;=\; \bigvee_{B\in\mathcal{B}} \big(\chi_B \wedge \zeta_B \wedge \imath = \mathcal{F}(B)\big),$$

  where $\imath$ is a new numerical variable not occurring in χ_B and ζ_B, and χ_B and ζ_B are defined as follows. For $B \in \mathcal{B}$, we write $\mathrm{supp}(B)$ for its *support*, i.e., the set of elements that appear in it, and $\sharp(a, B)$ for the number of occurrences of a in B. We then define

$$\chi_B \;=\; \bigwedge_{a\in\mathrm{supp}(B)} \exists^{=\sharp(a,B)}\bar{v} \left((\varphi')^\diamond \;\wedge\; \alpha_{t'}^{\jmath}[\jmath/a] \right)$$

  stating that the values of t', as $\bar{v}$ ranges over tuples satisfying φ', have exactly the same multiplicities as in B, and

$$\zeta_B \;=\; \forall\bar{v} \left((\varphi')^\diamond \;\rightarrow\; \bigvee_{a\in\mathrm{supp}(B)} \alpha_{t'}^{\jmath}[\jmath/a] \right)$$

  stating that only elements of B are values of t' as $\bar{v}$ ranges over tuples satisfying φ'. It is easy to verify that $\alpha_t^\imath$ is of rank $\max\{\mathrm{rank}((\varphi')^\diamond), \mathrm{rank}(\alpha_{t'}^{\jmath})\}+k \leq \max\{\mathrm{rank}(\varphi'), \mathrm{rank}(t')\}+k = \mathrm{rank}(t)$, where k is the arity of the tuple $\bar{v}$. Moreover, it should be clear that $\alpha_t^\imath$ essentially states that, for some bag $B \in \mathcal{B}$, the values of t' form exactly B, and the value of t is the value of the aggregate $\mathcal{F}$ on B.

Translation of the formula φ into an $\mathbf{L_C}$ formula $\varphi^\diamond$

- If φ is the atomic formula $x = y$ or $R(\bar{v})$, then $\varphi^\diamond$ is precisely φ, and thus, $\mathrm{rank}(\varphi^\diamond) = \mathrm{rank}(\varphi)$.
- If φ is the atomic formula $\imath = t$, then $\varphi^\diamond$ is $\alpha_t^\imath$ with $\mathrm{rank}(\varphi^\diamond) \leq \mathrm{rank}(t)$.
- If φ is $P(v_1, \ldots, v_k)$, then $\varphi^\diamond$ is

$$\bigvee_{(a_1,\ldots,a_k)\in P} \left(\bigwedge_{i\in[k]} \alpha_{v_i}^{\jmath_i}[\jmath_i/a_i] \right)$$

  with $\mathrm{rank}(\varphi^\diamond) \leq \max_{i\in[m]}\{\mathrm{rank}(\alpha_{v_i}^{\jmath_i})\} \leq \mathrm{rank}(\varphi)$.

- If φ is $\varphi_1 \wedge \varphi_2$, $\varphi_1 \vee \varphi_2$, $\neg\varphi_1$, then $\varphi^\diamond$ is $\varphi_1^\diamond \wedge \varphi_2^\diamond$, $\varphi_1^\diamond \vee \varphi_2^\diamond$, $\neg\varphi_1^\diamond$, respectively, of with $\mathrm{rank}(\varphi^\diamond) = \mathrm{rank}(\varphi)$.
- If φ is $\exists v\, \varphi_1$, $\forall v\, \varphi_1$, then $\varphi^\diamond$ is $\exists v\, \varphi_1^\diamond$, $\neg\exists v\, \neg\varphi_1^\diamond$, respectively, with $\mathrm{rank}(\varphi^\diamond) = \mathrm{rank}(\varphi)$.

This completes the translation of Ω-terms of type n occurring in φ, and of φ itself. It can be verified that $\varphi^\diamond$ is indeed an $\mathbf{L_C}$ formula, with $\mathrm{FV}(\varphi) = \mathrm{FV}(\varphi^\diamond)$, such that $\varphi(\bar{u})(D) = \varphi^\diamond(\bar{u})(D)$, for every database D of $\mathbf{S}$. □

We now proceed to show that the $\mathbf{L_C}$ formula $\varphi^\diamond$ provided by Lemma 35.10 can be converted into an $\mathbb{L}_\mathbf{C}$ formula ψ such that $\varphi^\diamond(\bar{u})$ and $\psi(\bar{u})$ have the same output on every database of $\mathbf{S}$, which will prove Proposition 35.9.

Lemma 35.11. *There exists an $\mathbb{L}_\mathbf{C}$ formula ψ over $\boldsymbol{S}$, with $FV(\varphi^\diamond) = FV(\psi)$, such that $\varphi^\diamond(\bar{u})(D) = \psi(\bar{u})(D)$, for every database D of $\boldsymbol{S}$.*

Proof. To prove the claim, we need to replace quantifiers of the form $\exists^{\geq n}\bar{v}\,\psi'$ in $\varphi^\diamond$ with $\exists^{\geq n}x\,\psi'$, where $x \in \mathsf{Var_o}$, without increasing the rank. We explain how this is done when $\bar{v}$ is binary; the general proof is then by induction on the arity of $\bar{v}$, using the case when $\bar{v}$ is binary as the base step.

Consider a subformula $\exists^{\geq n}(v,w)\,\psi'$ of $\varphi^\diamond$, where $v, w \in \mathrm{FV}(\psi')$. The idea of replacing this by simpler quantifiers is to say that there are at least k_1 v's for which there exist exactly ℓ_1 w's satisfying ψ', and there are exactly k_2 v's for which there exist exactly ℓ_2 w's satisfying ψ', and so on, with all the ℓ_i's being distinct in order to ensure that the same pair of values is never counted twice. Formally, a finite set of pairs of integers $\{(k_1,\ell_1),\ldots,(k_s,\ell_s)\}$ is an *n-witness* if $\sum_{i=1}^{s} k_i \cdot \ell_i \geq n$, and $\ell_i \neq \ell_j$ for each $i,j \in [s]$ with $i \neq j$. Let $\mathcal{W}_n$ be the set of all n-witnesses, which is clearly infinite. Then, $\exists^{\geq n}(v,w)\,\psi'$ is replaced by the infinitary disjunction

$$\bigvee_{\{(k_1,\ell_1),\ldots,(k_s,\ell_s)\}\in\mathcal{W}_n} \left(\bigwedge_{i=1}^{s} \exists^{\geq k_i} v\, \exists^{=\ell_i} w\, \psi' \right)$$

whose rank is $\mathrm{rank}(\exists^{\geq n}(v,w)\,\psi') = \mathrm{rank}(\psi') + 2$.

According to the definition of $\mathbb{L}_\mathbf{C}$, we can only quantify variables of type o. Thus, we need to eliminate from the above infinitary disjunction the quantifiers over numerical variables. This is done by using infinitary disjunctions as follows: a formula of the form $\exists^{\geq n}\imath\,\psi''$, where $\imath \in \mathrm{FV}(\psi'')$, is written as

$$\bigvee_{A\subseteq\mathsf{Num}\,:\,|A|\geq n} \left(\bigwedge_{a\in A} \psi''[\imath/a] \right)$$

whose rank is $\mathrm{rank}(\exists^{\geq n}\imath\,\psi'')-1$. This completes the proof of Lemma 35.11. □

By Lemma 35.10 and 35.11, we get an $\mathbb{L}_\mathbf{C}$ formula ψ, with $\mathrm{FV}(\varphi) = \mathrm{FV}(\psi)$, such that $\varphi(\bar{u})(D) = \psi(\bar{u})(D)$, for every database D of $\mathbf{S}$, as needed. □

Locality of Counting Logic

We proceed with the last step of the proof of Theorem 35.4. The goal is to establish a locality property for the counting logic $\mathbb{L}_{\mathbf{C}}$ analogous to that for $\mathrm{RA}_{\mathsf{Aggr}}(\Omega)$ queries stated in Theorem 35.4. Given an expression $\varphi(\bar{u})$, where φ is an $\mathbb{L}_{\mathbf{C}}$ formula, and $\bar{u}$ is a tuple over $\mathrm{FV}(\varphi)$ that mentions all the variables of $\mathrm{FV}(\varphi)$, although is not a query, we can naturally apply the definition of locality for queries (see Definition 35.3) to it. We proceed to show that every such expression, as long as it is over a schema with relation names of ordinary type, and its free variables are of ordinary type, is local.

Proposition 35.12

Consider an $\mathbb{L}_{\mathbf{C}}$ formula φ over a schema $\mathbf{S} = \{R_1 : \boldsymbol{\tau}_1, \ldots, R_n : \boldsymbol{\tau}_n\}$ with $\boldsymbol{\tau}_i \in \{\mathsf{o}\}^{k_i}$ and $k_i \geq 0$, for each $i \in [n]$, of rank k such that $\mathrm{FV}(\varphi) \subseteq \mathsf{Var}_{\mathsf{o}}$, and a tuple $\bar{x}$ over $\mathrm{FV}(\varphi)$ that mentions all the variables of $\mathrm{FV}(\varphi)$. It holds that $\varphi(\bar{x})$ is $(3^k - 1)/2$-local.

Proof. For clarity, we assume that $\mathbf{S} = \{R : (\mathsf{o}, \mathsf{o})\}$, but of course the proof generalizes to arbitrary schemas $\{R_1 : \boldsymbol{\tau}_1, \ldots, R_n : \boldsymbol{\tau}_n\}$ with $\boldsymbol{\tau}_i \in \{\mathsf{o}\}^{k_i}$ and $k_i \geq 0$, for each $i \in [n]$. We proceed by induction on the rank of φ.

For the base step, assume that $\mathrm{rank}(\varphi) = 0$. This implies that φ is an atomic formula $x = y$ or $R(x, y)$. It is clear that, for a database D of $\mathbf{S}$, and tuples $\bar{a}, \bar{b}$ over Const, the fact that $N_0^D(\bar{a})$ and $N_0^D(\bar{b})$ are isomorphic implies that $\bar{a} \in \varphi(\bar{x})(D)$ iff $\bar{b} \in \varphi(\bar{x})(D)$. Therefore, φ is 0-local, as needed.

Assume now that $\mathrm{rank}(\varphi) = k$ for $k > 0$. We first observe that infinitary conjunction and infinitary disjunction, as well as negation, do not alter locality. More precisely, assuming that Φ is a (possibly infinite) set of formulae such that, for each $\psi \in \Psi$, $\mathrm{FV}(\varphi) = \mathrm{FV}(\psi)$ and $\psi(\bar{x})$ is r-local, and $\psi' = \bigwedge_{\varphi \in \Psi} \psi$ or $\psi' = \bigvee_{\psi \in \Psi} \psi$, then $\psi'(\bar{x})$ is r-local. Analogously, if ψ is a formula such that $\mathrm{FV}(\varphi) = \mathrm{FV}(\psi)$ and $\psi(\bar{x})$ is r-local, and $\psi' = \neg\psi$, then $\psi'(\bar{x})$ is r-local. Consequently, it suffices to show that, for an arbitrary subformula $\psi = \exists^{\geq n} y\, \psi'$ of φ with $\mathrm{rank}(\psi) = k$, $\psi(\bar{x})$ is $(3^k - 1)/2$-local. Clearly, $\mathrm{rank}(\psi') = k - 1$, and thus, by induction hypothesis, $\psi'(\bar{x}, y)$ is $(3^{k-1} - 1)/2$-local. We proceed to show the following key lemma; for brevity, let $r = (3^{k-1} - 1)/2$-local.

Lemma 35.13. *It holds that $\psi(\bar{x})$ is $(3r + 1)$-local.*

Proof. Consider a database D of $\mathbf{S}$, and two tuples $\bar{a}, \bar{b}$ over Const such that $N_{3r+1}^D(\bar{a})$ and $N_{3r+1}^D(\bar{b})$ are isomorphic. We need to establish that $\bar{a} \in \psi(\bar{x})(D)$ iff $\bar{b} \in \psi(\bar{x})(D)$. To this end, it suffices to show that there exists a bijection f from $\mathrm{Dom}(D)$ to $\mathrm{Dom}(D)$ such that $N_r^D(\bar{a}, c)$ and $N_r^D(\bar{b}, f(c))$ are isomorphic, for every $c \in \mathrm{Dom}(D)$. Indeed, if such a bijection exists, the claim follows. Assuming that $\bar{a} \in \psi(\bar{x})(D)$, we can find n elements of $\mathrm{Dom}(D)$ such that $(\bar{a}, c_i) \in \psi'(\bar{x}, y)(D)$, for each $i \in [n]$. But since $N_r^D(\bar{a}, c)$ and $N_r^D(\bar{b}, f(c))$

are isomorphic, and f is a bijection, the distinct elements $f(c_1), \ldots, f(c_n)$ of $\mathrm{Dom}(D)$ witness that $\bar{b} \in \psi(\bar{x})(D)$, as needed. Likewise, we can show that $\bar{b} \in \psi(\bar{x})(D)$ implies $\bar{a} \in \psi(\bar{x})(D)$ by using the bijection f^{-1} instead of f. The rest of the proof is devoted to showing the existence of the bijection f.

Consider a database D' of $\mathbf{S}'$, and a constant $d \in \mathrm{Dom}(D')$ such that, for every $e \in \mathrm{Dom}(D')$, $d_D(d, e) \leq r$. Let $c \in \mathrm{Dom}(N^D_{2r+1}(\bar{a}))$, and assume that there exists a bijection $\mu : \mathrm{Dom}(N^D_r(c)) \to \mathrm{Dom}(D')$ such that $\mu(c) = d$, and $R(\bar{e}) \in N^D_r(c)$ implies $R(\mu(\bar{e})) \in D'$, in which case we say that $N^D_r(c)$ and (D', d) are isomorphic. Since $N^D_{3r+1}(\bar{a})$ and $N^D_{3r+1}(\bar{b})$ are isomorphic, let say witnessed by the bijection h, we get that $N^D_r(h(c))$ and (D', d) are isomorphic. Therefore, the number of constants $c \in \mathrm{Dom}(N^D_{2r+1}(\bar{a}))$ such that $N^D_r(c)$ is isomorphic to (D', d), and the number of constants $c \in \mathrm{Dom}(N^D_{2r+1}(\bar{b}))$ such that $N^D_r(c)$ is isomorphic to (D', d), are the same. Hence, the same holds for $\mathrm{Dom}(D) - \mathrm{Dom}(N^D_{2r+1}(\bar{a}))$ and $\mathrm{Dom}(D) - \mathrm{Dom}(N^D_{2r+1}(\bar{b}))$.

Since the database D' of $\mathbf{S}$ was chosen arbitrarily, we get that there is a bijection $f : \mathrm{Dom}(D) - \mathrm{Dom}(N^D_{2r+1}(\bar{a})) \to \mathrm{Dom}(D) - \mathrm{Dom}(N^D_{2r+1}(\bar{b}))$ such that $N^D_r(c)$ and $N^D_r(f(c))$ are isomorphic, for every $c \in \mathrm{Dom}(D) - \mathrm{Dom}(N^D_{2r+1}(\bar{a}))$. We extend f to all the elements of $\mathrm{Dom}(D)$ by letting $f(c) = h(c)$, for every $c \in \mathrm{Dom}(N^D_{2r+1}(\bar{a}))$. To conclude the proof, it remains to show that $N^D_r(\bar{a}, c)$ and $N^D_r(\bar{b}, f(c))$ are isomorphic, for every constant $c \in \mathrm{Dom}(D)$. Assume first that $c \in \mathrm{Dom}(N^D_{2r+1}(\bar{a}))$. Then, for every constant $e \in \mathrm{Dom}(N^D_r(c))$, $d_D(\bar{a}, e) \leq 3r + 1$, and thus, $N^D_r(\bar{a}, c)$ and $N^D_r(\bar{b}, f(c))$ are isomorphic since f is the isomorphism h. Assume now that $c \notin \mathrm{Dom}(N^D_{2r+1}(\bar{a}))$. Then, $N^D_r(\bar{a}, c)$ is the disjoint union of $N^D_r(\bar{a})$ and $N^D_r(c)$. Hence, $N^D_r(\bar{b}, f(c))$ is the disjoint union of $N^D_r(\bar{b})$ and $N^D_r(f(c))$. Therefore, $N^D_r(\bar{a}, c)$ and $N^D_r(\bar{b}, f(c))$ are isomorphic since they are the disjoint union of isomorphic sets of facts. □

It is straightforward to see that Proposition 35.12 follows by Lemma 35.13 since $3(3^{k-1} - 1)/2 + 1 = (3^k - 1)/2$. □

We can now finalize the proof of Theorem 35.4. Consider an $\mathrm{RA}_{\mathsf{Aggr}}(\Omega)$ query e of type $\{\mathsf{o}\}^k$, for $k \geq 0$, over a schema $\mathbf{S}$ as in the statement of Theorem 35.4. By Proposition 35.7 and Proposition 35.9, we get that there exists an $\mathbb{L}_{\mathbf{C}}$ formula φ_e over $\mathbf{S}$ with $\mathrm{FV}(\varphi_e) \subseteq \mathsf{Var}_{\mathsf{o}}$, and a tuple $\bar{x}_e$ over $\mathrm{FV}(\varphi_e)$ that mentions all the variables of $\mathrm{FV}(\varphi_e)$, such that $e(D) = \varphi_e(\bar{x}_e)(D)$, for every database D of $\mathbf{S}$. Therefore, by Proposition 35.12, we conclude that e is $(3^k - 1)/2$-local, where k is the rank of the formula φ_e, as needed.

36

Adding Recursion: Datalog

As discussed in Chapter 35, a serious limitation of relational algebra with aggregates, and in fact all of the query languages encountered so far in the previous chapters, is their inability to express recursive queries such as the reachability query. In this chapter, we introduce a rule-based language, called Datalog, that is powerful enough to express such queries. It can be seen as an extension of UCQs with the key feature of recursion.

Syntax of Datalog

We start by defining the syntax of Datalog rules by using a rule-based syntax similar to that of CQs when seen as rules.

Definition 36.1: Syntax of Datalog

A *Datalog rule* over a schema $\mathbf{S}$ is an expression of the form

$$R_0(\bar{x}) \text{ :- } R_1(\bar{u}_1), \ldots, R_n(\bar{u}_n)$$

for $n \geq 1$, where

- $R_i \in \mathbf{S}$, for each $i \in [0, n]$,
- $R_i(\bar{u}_i)$ is a relational atom, and $\bar{u}_i$ is a tuple of constants and variables, for each $i \in [n]$,
- $R_0(\bar{x})$ is a relational atom, and $\bar{x}$ is a tuple of variables, and
- each variable mentioned in $\bar{x}$ is also mentioned in $\bar{u}_k$ for some $k \in [n]$; this is known as the *safety condition*.

A *Datalog program* over $\mathbf{S}$ is a finite set of Datalog rules over $\mathbf{S}$.

As we shall see, the key idea underlying Datalog queries is to declaratively specify what the query output should be by means of a Datalog program. The Datalog program that provides the specification of the reachability query over directed graphs follows.

Example 36.2: Graph Reachability

Consider the following (named) database schema:

```
Edge [ node1, node2 ]
Reachable [ node1, node2 ]
```

The Edge relation stores the edges of the input directed graph G, and the Reachable relation stores the pairs of nodes (v, u) of G such that u is reachable from v. We can now inductively compute the Reachable relation via the following Datalog program over the above schema:

$$\begin{aligned} \text{Reachable}(x, y) &\text{ :- } \text{Edge}(x, y) \\ \text{Reachable}(x, y) &\text{ :- } \text{Reachable}(x, z), \text{Edge}(z, y)\,. \end{aligned}$$

The first rule, which is the base step of the inductive definition, simply states that if there is an edge from x to y, then y is reachable from x. The second rule, which corresponds to the inductive step, states that if z is reachable from x and there is an edge from z to y, then y is reachable from x. Notice that the second rule is recursive in the sense that the definition of the relation Reachable depends on itself.

The relational atom that appears on the left of the :- symbol in a Datalog rule is called the *head* of the rule, while the expression that appears on the right of the :- symbol is called the *body* of the rule. Given a Datalog program Π over a schema $\mathbf{S}$, it is crucial to have a way to distinguish between the relation names of $\mathbf{S}$ that appear only in the bodies of the rules of Π, and those that appear in the head of at least one rule of Π. In particular, a relation name $R \in \mathbf{S}$ occurring in the Datalog program Π is called:

- *extensional* if there is no rule of the form $R(\bar{x})$:- body in Π, that is, R occurs only in rule-bodies, and
- *intensional* if there exists at least one rule of the form $R(\bar{x})$:- body in Π, that is, R appears in the head of at least one rule of Π.

Intuitively, extensional relation names correspond to the input relations, while intensional relation names correspond to the relations that are computed by the Datalog program. The *extensional (database) schema* of Π, denoted $\mathsf{edb}(\Pi)$, consists of the extensional relation names in Π, while the *intentional schema* of Π, denoted $\mathsf{idb}(\Pi)$, is the set of all intentional relation names in Π.

The *schema* of Π, denoted $\mathsf{sch}(\Pi)$, is the set of relation names $\mathsf{edb}(\Pi) \cup \mathsf{idb}(\Pi)$. Note that $\mathsf{sch}(\Pi)$ is, in general, a subset of $\mathbf{S}$ since it might be the case that some relation names of $\mathbf{S}$ do not appear in Π.

Semantics of Datalog

An interesting property of Datalog programs is the fact that their semantics can be defined either declaratively by adopting a *model-theoretic* approach, or operationally by following a *fixpoint* approach. In the model-theoretic approach, the Datalog rules are considered as logical sentences asserting a property of the desired result, while in the fixpoint approach the semantics is defined as a particular solution of a fixpoint procedure.

Model-Theoretic Semantics

Recall that a set Φ of first-order sentences over a schema $\mathbf{S}$ is called a *first-order theory* over $\mathbf{S}$, or simply a *theory* over $\mathbf{S}$. A database of $\mathbf{S}$ is a *model* of the theory Φ if, for every sentence $\varphi \in \Phi$, $D \models \varphi$.[1] The idea underlying the model-theoretic approach is to consider a Datalog program Π as a first-order theory Φ_Π over $\mathsf{sch}(\Pi)$ that describes the desired outcome of the program. In other words, the desired outcome is a particular model of Φ_Π; hence the name model-theoretic semantics. However, there might be infinitely many models of the theory Φ_Π, which means that the theory alone does not uniquely determine the desired outcome of the program. It is therefore crucial to specify which model is the intended outcome. We proceed to formalize the above discussion. In particular, we are going to explain how a Datalog program is converted into a first-order theory, and which model of this theory is the intended one.

Definition 36.3: From a Program to a Theory

Given a Datalog rule ρ of the form $R_0(\bar{x}) \text{ :- } R_1(\bar{u}_1), \ldots, R_n(\bar{u}_n)$, we write φ_ρ for the first-order sentence

$$\forall x_1 \cdots \forall x_m \, (R_1(\bar{u}_1) \wedge \cdots \wedge R_n(\bar{u}_n) \;\rightarrow\; R_0(\bar{x})),$$

where $x_1, \ldots, x_m$ are the variables in ρ. Given a Datalog program Π, we define the first-order theory Φ_Π over $\mathsf{sch}(\Pi)$ as $\{\varphi_\rho \mid \rho \in \Pi\}$.

For brevity, we refer to the models of a Datalog program Π meaning the models of the theory Φ_Π. Interestingly, the notion of satisfaction of a sentence φ_ρ, where $\rho \in \Pi$, by a database D of $\mathsf{sch}(\Pi)$, can be characterized by means of

[1] The notion of first-order theory, together with the notion of model of such a theory, have been already used in Chapter 33.

homomorphisms. Similarly to CQs, the body of a Datalog rule can be viewed as a set of atoms. More precisely, given a Datalog rule ρ of the form

$$R_0(\bar{x}) \text{ :- } R_1(\bar{u}_1), \ldots, R_n(\bar{u}_n)$$

we define the set of relational atoms

$$A_\rho \;=\; \{R_1(\bar{u}_1), \ldots, R_n(\bar{u}_n)\}.$$

We can thus talk about homomorphisms from rule-bodies to databases. It is then easy to verify the following proposition that provides a useful characterization of rule satisfaction that will be used in our later proofs:

Proposition 36.4

Consider a Datalog rule ρ of the form $R_0(\bar{x}) \text{ :- } R_1(\bar{u}_1), \ldots, R_n(\bar{u}_n)$, and a database D. The following are equivalent:

1. $D \models \varphi_\rho$.
2. For every homomorphism h from A_ρ to D, $R_0(h(\bar{x})) \in D$.

It should be clear that a Datalog program Π admits infinitely many models. We proceed to explain how we choose the intended one. The idea is that the intended model should not contain more atoms than needed for satisfying Φ_Π. In other words, from all the models of Φ_Π, we choose the $\subseteq$-*minimal* ones, i.e., those models D such that, for every atom $R(\bar{a}) \in D$, it is the case that $D - \{R(\bar{a})\}$ is not a model of Φ_Π. Based on this simple idea, we proceed to define the semantics of a Datalog program on an input database.

Given a Datalog program Π and a database D of $\mathsf{edb}(\Pi)$, we define

$$\mathsf{MM}(\Pi, D) \;=\; \{D' \mid D' \text{ is a } \subseteq\text{-minimal model of } \Pi \text{ and } D \subseteq D'\}.$$

We can show that $\mathsf{MM}(\Pi, D)$ contains *exactly one* database, which will give rise to the semantics of Π on D. But first we need to establish an auxiliary result. Let $\mathsf{B}(\Pi, D)$ be the union of D with the set of all relational atoms that can be formed using relation names from $\mathsf{idb}(\Pi)$ and constants from $\mathrm{Dom}(D)$:

$$\mathsf{B}(\Pi, D) \;=\; D \;\cup\; \left\{R(\bar{a}) \mid R \in \mathsf{idb}(\Pi) \text{ and } \bar{a} \in \mathrm{Dom}(D)^{\mathrm{ar}(R)}\right\}.$$

We can show the following:

Lemma 36.5. *Consider a Datalog program Π, and a database D of $\mathsf{edb}(\Pi)$. It holds that $\mathsf{B}(\Pi, D)$ is a model of Π that contains D.*

Proof. The fact that $\mathsf{B}(\Pi, D)$ contains D follows by definition. It remains to show that $\mathsf{B}(\Pi, D)$ is a model of Π. Consider an arbitrary rule $\rho \in \Pi$ of the

form $R_0(\bar{x}) \text{ :- } R_1(\bar{u}_1), \ldots, R_n(\bar{u}_n)$, and assume that there is a homomorphism h from A_ρ to D. Due to the safety condition, every variable in $\bar{x}$ occurs in A_ρ. This implies that $h(\bar{x})$ is a tuple over $\mathrm{Dom}(D)$. Since $R \in \mathsf{idb}(\Pi)$ we conclude that $R(h(\bar{x})) \in \mathsf{B}(\Pi, D)$, and thus $\mathsf{B}(\Pi, D) \models \varphi_\rho$. Consequently, $\mathsf{B}(\Pi, D)$ is a model of Φ_Π, and thus, a model of Π, as needed. □

We are now ready to show the claimed statement concerning $\mathsf{MM}(\Pi, D)$:

Proposition 36.6

Consider a Datalog program Π, and a database D of $\mathsf{edb}(\Pi)$. Then

$$|\mathsf{MM}(\Pi, D)| \;=\; 1.$$

Proof. By Lemma 36.5, $\mathsf{B}(\Pi, D)$ is a model of Π that contains D. Therefore, there exists a subset of $\mathsf{B}(\Pi, D)$ that belongs to $\mathsf{MM}(\Pi, D)$, which implies that $|\mathsf{MM}(\Pi, D)| \geq 1$. Assume now that $|\mathsf{MM}(\Pi, D)| \geq 2$, and let $D_1, \ldots, D_\ell$, for $\ell \geq 2$, be its members. We proceed to show that the database

$$D_\cap \;=\; D_1 \cap \cdots \cap D_\ell$$

is a model of Π that contains D, which contradicts the fact that $D_1, \ldots, D_\ell$ are $\subseteq$-minimal. Since $D \subseteq D_i$, for each $i \in [\ell]$, we get that $D \subseteq D_\cap$. Consider now a Datalog rule $\rho \in \Pi$ of the form $R_0(\bar{x}) \text{ :- } R_1(\bar{u}_1), \ldots, R_n(\bar{u}_n)$, and assume there exists a homomorphism h from A_ρ to $D_\cap$. For each $i \in [\ell]$, D_i is a model of Π, and thus, $R_0(h(\bar{x})) \in D_i$. Therefore, $R_0(h(\bar{x})) \in D_\cap$, which means, due to Proposition 36.4, that $D_\cap \models \varphi_\rho$. Hence, $D_\cap$ is a model of Π, as needed. □

Having the above result in place, we are now ready to define the semantics of a Datalog program on an input database.

Definition 36.7: Semantics of Datalog

Given a Datalog program Π, and a database D of $\mathsf{edb}(\Pi)$, the *output of* Π *on* D, denoted $\Pi(D)$, is the $\subseteq$-minimal model of Π that contains D.

By Proposition 36.6, we get that $\Pi(D)$ is uniquely determined by the program and the database, and thus, Definition 36.7 provides a well-defined semantics for Datalog programs. The crucial question that comes up is whether we can devise an algorithm that computes the semantics of a Datalog program Π on a database D. The next result provides such an algorithm. Let $\mathsf{M}(\Pi, D)$ be all the subsets of $\mathsf{B}(\Pi, D)$ that are models of Π and contain D. Formally,

$$\mathsf{M}(\Pi, D) \;=\; \{D' \mid D' \text{ is a model of } \Pi \text{ and } D \subseteq D' \subseteq \mathsf{B}(\Pi, D)\}.$$

Interestingly, the intersection of the databases occurring in $\mathsf{M}(\Pi, D)$ coincides with the $\subseteq$-minimal model of Π that contains D. By giving a proof similar to that of Proposition 36.6, we can show that $\bigcap \mathsf{M}(\Pi, D)$ is a model of Π that contains D, while the fact that is a $\subseteq$-minimal model follows by construction.

Theorem 36.8

Consider a Datalog program Π, and a database D of $\mathsf{edb}(\Pi)$. Then

$$\Pi(D) \;=\; \bigcap_{D' \in \mathsf{M}(\Pi, D)} D'.$$

It is clear that Theorem 36.8 suggests the following procedure for computing the semantics of a Datalog program Π on a database D: construct all the possible subsets of $\mathsf{B}(\Pi, D)$ that are models of Π and contain D, and then compute their intersection. However, this is computationally a very expensive procedure. As we shall see in the next section, the fixpoint approach provides a more efficient algorithm for computing the database $\Pi(D)$.

Fixpoint Semantics

We present an alternative way to define the semantics of Datalog that relies on an operator called the *immediate consequence operator*. This operator is applied on a database in order to produce new relational atoms. The model-theoretic semantics presented above coincides with the smallest solution of a fixpoint equation that involves the immediate consequence operator.

Definition 36.9: Immediate Consequence Operator

Consider a Datalog program Π, and a database D of $\mathsf{sch}(\Pi)$. A relational atom $R(\bar{a})$ is an *immediate consequence* for Π and D if:

1. $R(\bar{a}) \in D$, or
2. There exists a rule $\rho \in \Pi$ of the form $R(\bar{x}) \text{ :- } R_1(\bar{u}_1), \ldots, R_n(\bar{u}_n)$ such that $(A_\rho, \bar{x}) \to (D, \bar{a})$.

The *immediate consequence operator* of Π is defined as the function

$$T_\Pi \;:\; \mathrm{Inst}(\mathsf{sch}(\Pi)) \;\to\; \mathrm{Inst}(\mathsf{sch}(\Pi))$$

such that

$$T_\Pi(D) \;=\; \{R(\bar{a}) \mid R(\bar{a}) \text{ is an immediate consequence for } \Pi \text{ and } D\}.$$

A database D of $\mathsf{sch}(\Pi)$ is called a *fixpoint* of T_Π if $T_\Pi(D) = D$.

The next lemma, which is easy to prove, collects some useful properties of the T_Π operator that we are going to use below.

Lemma 36.10. *Consider a Datalog program Π. The following hold:*

1. *T_Π is monotone, i.e., for every two databases D and D' of $\mathsf{sch}(\Pi)$, $D \subseteq D'$ implies $T_\Pi(D) \subseteq T_\Pi(D')$.*
2. *A database D of $\mathsf{sch}(\Pi)$ is a model of Π if and only if $T_\Pi(D) \subseteq D$.*
3. *Every fixpoint of T_Π is a model of Π.*

We are now ready to establish the following crucial result, which states that the model-theoretic semantics of a Datalog program on a database D coincides with the $\subseteq$-minimal fixpoint of T_Π that contains D.

Theorem 36.11

Consider a Datalog program Π, and a database D of $\mathsf{edb}(\Pi)$. It holds that $\Pi(D)$ is the $\subseteq$-minimal fixpoint of T_Π that contains D.

Proof. We first show that $\Pi(D)$ is a fixpoint of T_Π, i.e., $T_\Pi(\Pi(D)) = \Pi(D)$. Since $\Pi(D)$ is a model of Π, Lemma 36.10 implies that $T_\Pi(\Pi(D)) \subseteq \Pi(D)$. By Lemma 36.10, T_Π is monotone, and thus, $T_\Pi(T_\Pi(\Pi(D))) \subseteq T_\Pi(\Pi(D))$. Therefore, by Lemma 36.10, $T_\Pi(\Pi(D))$ is a model of Π that contains D. But since $\Pi(D)$ is the $\subseteq$-minimal mode of Π that contains D, we immediately get that $\Pi(D) \subseteq T_\Pi(\Pi(D))$. Consequently, $T_\Pi(\Pi(D)) = \Pi(D)$. By Lemma 36.10, each fixpoint of T_Π that contains D is a model of Π that contains D. Hence, $\Pi(D)$ is the $\subseteq$-minimal fixpoint of T_Π that contains D. □

It remains to explain how the $\subseteq$-minimal fixpoint of the T_Π operator that contains the database D is constructed. This is essentially done by iteratively applying the T_Π operator starting from the database D.

Definition 36.12: Application of the T_Π Operator

Consider a Datalog program Π, and a database D of $\mathsf{edb}(\Pi)$. We define

$$T_\Pi^0(D) \;=\; D \qquad \text{and} \qquad T_\Pi^{i+1}(D) \;=\; T_\Pi(T_\Pi^i(D)), \text{ for } i \in \mathbb{N},$$

and we let

$$T_\Pi^\infty(D) \;=\; \bigcup_{i \geq 0} T_\Pi^i(D).$$

At first glance, the construction of $T_\Pi^\infty(D)$ requires infinitely many iterations. However, since $T_\Pi^\infty(D) \subseteq \mathsf{B}(\Pi, D)$, it is the case that $T_\Pi^\infty(D)$ is obtained in at most $|\mathsf{B}(\Pi, D)|$ iterations. It is easy to verify that

$$T_\Pi^\infty(D) \;=\; T_\Pi^{|\mathsf{B}(\Pi,D)|}(D).$$

We now show the following result, which essentially states that the semantics of a Datalog program Π on a database D can be computed by iteratively applying the operator T_Π starting from D until a fixpoint is reached.

Theorem 36.13

Consider a Datalog program Π, and a database D of $\mathsf{edb}(\Pi)$. It holds that $T_\Pi^\infty(D)$ is the $\subseteq$-minimal fixpoint of T_Π that contains D.

Proof. Recall first that the following hold:

$$T_\Pi^\infty(D) \;=\; T_\Pi^{|\mathsf{B}(\Pi,D)|}(D) \qquad \text{and} \qquad T_\Pi(T_\Pi^{|\mathsf{B}(\Pi,D)|}(D)) \;=\; T_\Pi^{|\mathsf{B}(\Pi,D)|}(D).$$

Therefore, $T_\Pi^\infty(D)$ is a fixpoint of T_Π that contains D. It remains to show that $T_\Pi^\infty(D)$ is $\subseteq$-minimal, or, equivalently, for every fixpoint D' of T_Π that contains D, $T_\Pi^\infty(D) \subseteq D'$. Fix such a fixpoint D'. We can show via an easy inductive argument that $T_\Pi^i(D) \subseteq D'$, for every $i \in \mathbb{N}$, which implies that $T_\Pi^\infty(D) \subseteq D'$. In fact, $T_\Pi^0(D) \subseteq D'$ since $T_\Pi^0(D) = D$. Moreover, $T_\Pi^i(D) \subseteq D'$ implies $T_\Pi(T_\Pi^i(D)) = T_\Pi^{i+1}(D) \subseteq T_\Pi(D') = D'$ by monotonicity of T_Π. □

The next result is an immediate corollary of Theorems 36.11 and 36.13:

Corollary 36.14

Consider a Datalog program Π, and a database D of $\mathsf{edb}(\Pi)$. Then

$$\Pi(D) \;=\; T_\Pi^\infty(D).$$

Datalog Queries

Recall that a k-ary query q produces a finite set of k-ary tuples $q(D) \subseteq \mathsf{Const}^k$, for every database D. Datalog programs can be used to define queries. In order to do this, we simply specify together with a Datalog program Π a relation name R from $\mathsf{idb}(\Pi)$ that indicates the relation that collects the output of the query. In other words, given a database D of $\mathsf{edb}(\Pi)$, after computing the database $\Pi(D)$, the output of the query is the set of tuples $\bar{a}$ over $\mathrm{Dom}(D)$ such that $R(\bar{a}) \in \Pi(D)$. For example, the Datalog query over $\{\mathrm{Edge}[2]\}$ that computes the pairs (v, u) such that u is reachable from v is $(\Pi, \mathrm{Reachable})$, where Π is the Datalog program over $\{\mathrm{Edge}[2], \mathrm{Reachability}[2]\}$ given in Example 36.2. The formal definition of Datalog queries follows.

Definition 36.15: Datalog Queries

A *Datalog query* over a schema $\mathbf{S}$ is a pair (Π, R), where Π is a Datalog program over a schema $\mathbf{S} \cup \mathbf{S}'$, with $\mathbf{S}'$ being a schema disjoint from $\mathbf{S}$, such that $\mathsf{edb}(\Pi) \subseteq \mathbf{S}$, $\mathsf{idb}(\Pi) \subseteq \mathbf{S}'$, and $R \in \mathsf{idb}(\Pi)$.

Having the semantics of a Datalog program Π on a database D (see Definition 36.7), we can naturally define what is the output of a Datalog query (Π, R) on D; simply collect the tuples in the relation R after computing $\Pi(D)$.

Definition 36.16: Evaluation of Datalog Queries

Given a database D of a schema $\mathbf{S}$, and a Datalog query $q = (\Pi, R)$ over $\mathbf{S}$, the *output* of q on D is defined as the set of tuples

$$q(D) \;=\; \left\{ \bar{a} \in \mathsf{Const}^{\mathrm{ar}(R)} \mid R(\bar{a}) \in \Pi(D) \right\}.$$

It is clear that the set $q(D)$ belongs to $\mathcal{P}(\mathsf{Const}^{\mathrm{ar}(R)})$. However, to be able to say that q defines a query over $\mathbf{S}$ as in Definition 2.5, we need to ensure that $q(D) \in \mathcal{P}_{\mathrm{fin}}(\mathsf{Const}^{\mathrm{ar}(R)})$, i.e., the output of q on D is finite. This is guaranteed by the following result, which is an immediate consequence of Theorem 36.8, and the fact that, for every database $D' \in \mathsf{M}(\Pi, D)$, $\mathrm{Dom}(D') = \mathrm{Dom}(D)$.

Proposition 36.17

For a database D of schema $\mathbf{S}$, and a Datalog query $q = (\Pi, R)$ over $\mathbf{S}$,

$$q(D) \;=\; \left\{ \bar{a} \in \mathrm{Dom}(D)^{\mathrm{ar}(R)} \mid R(\bar{a}) \in \Pi(D) \right\}.$$

Since $\mathrm{Dom}(D)$ is finite, Proposition 36.17 implies that $q(D) \in \mathcal{P}_{\mathrm{fin}}(\mathsf{Const}^{k})$, and thus, q defines a query over $\mathbf{S}$ in the sense of Definition 2.5.

At the beginning of the chapter, we claimed that Datalog extends UCQs with the feature of recursion. We can easily show that indeed Datalog leads to a strictly more expressive language that is able to express recursive queries:

Theorem 36.18

The language of Datalog queries is strictly more expressive than the language of UCQs.

Proof. From Theorem 35.2, we conclude that the reachability query on directed graphs cannot by expressed as a UCQ, but we have already seen that it can be easily expressed as a Datalog query. On the other hand, it is straightforward to see that every UCQ $q(\bar{x}) = q_1 \cup \cdots \cup q_n$ can be equivalently written as the Datalog query $(\Pi_q, \mathrm{Answer}(\bar{x}))$, where Π_q consists of the CQs $q_1, \ldots, q_n$ seen as rules of the form $\mathrm{Answer}(\bar{x}) :- \mathrm{body}$. □

Recall from Chapter 30 that UCQs with variable-constant equality form a strictly more expressive language than UCQs. It turns out that there exists a UCQ with variable-constant equality that cannot be expressed as a Datalog query, which means that Datalog and UCQs with variable-constant equality form incomparable languages in terms of expressive power. This is because UCQs with variable-constant equality may have in their output constants not

from the domain of the database, which is not possible for Datalog queries (Proposition 36.17). Consider, for example, the simple query $q = \varphi(x)$ with

$$\varphi \;=\; (x = a),$$

where a is a constant. For $D = \{R(b)\}$, we get that $q(D) = \{(a)\}$.

37

Expressiveness of Datalog Queries

We have already seen in the previous chapter that Datalog queries are strictly more expressive than UCQs. We have also seen an easy inexpressibility result, i.e., there are UCQs with variable-constant equality (in fact, the query $\varphi(x)$ with $\varphi = (x = a)$) that cannot be expressed as a Datalog query. The question that comes up is how Datalog queries compare in terms of expressive power with UCQs with inequality, and more generally, whether Datalog queries can express negation, or at least a restricted form of negation.

Our goal in this chapter is to show that Datalog queries are inherently positive. Note that the easy inexpressibility result that Datalog queries cannot express $\varphi(x)$ with $\varphi = (x = a)$ relies on the property of Datalog queries provided by Proposition 36.17, that is, the output of a Datalog query mentions only constants from the domain of the database. However, this property is not powerful enough to show that Datalog queries are inherently positive. We proceed to establish that Datalog queries are preserved under homomorphisms and monotone, and then use those properties to show that indeed Datalog queries cannot express inequality, negative relational atoms, and difference.

Preservation Under Homomorphisms

The notion of preservation under homomorphisms for Datalog queries is defined in the same way as for FO queries. For a Datalog program Π, we write $\mathrm{Dom}(\Pi)$ for the set of constants occurring in the rules of Π.

Definition 37.1: Preservation Under Homomorphisms

Consider a Datalog query $q = (\Pi, R)$ over a schema $\mathbf{S}$. We say that q is *preserved under homomorphisms* if, for every two databases D and D' of $\mathbf{S}$, and tuples $\bar{a} \in \mathrm{Dom}(D)^{\mathrm{ar}(R)}$ and $\bar{b} \in \mathrm{Dom}(D')^{\mathrm{ar}(R)}$, it holds that

$$(D, \bar{a}) \to_{\mathrm{Dom}(\Pi)} (D', \bar{b}) \text{ and } \bar{a} \in q(D) \text{ implies } \bar{b} \in q(D').$$

We proceed to show that Datalog queries are preserved under homomorphisms. The key idea underlying this result is that a Datalog query q over a schema $\mathbf{S}$ can be converted into an equivalent UCQ q' over $\mathbf{S}$ providing that q' can have infinitely many disjuncts, and each disjunct can be a CQ with infinitely many existentially quantified variables and conjuncts. Such CQs and UCQs are called *infinitary*. The evaluation of infinitary CQs and UCQs is defined in the same way as for CQs and UCQs, respectively. It is also easy to show that every infinitary UCQ is preserved under homomorphisms; the proof is essentially the same as the one of Proposition 30.10, which establishes that UCQs are preserved under homomorphisms. Therefore, to show that Datalog queries are preserved under homomorphisms it suffices to show that a Datalog query over $\mathbf{S}$ can be converted into an equivalent infinitary UCQ over $\mathbf{S}$.

Proposition 37.2

Consider a Datalog query $q = (\Pi, R)$ over a schema $\mathbf{S}$. There exists an infinitary UCQ $q' = \varphi(\bar{x})$ over $\mathbf{S}$ with $\mathrm{Dom}(\Pi) = \mathrm{Dom}(\varphi)$ and $q \equiv q'$.

For technical clarity, we show the above result only for Boolean queries. Nevertheless, the given proof illustrates the key elements that are used in the proof for non-Boolean queries, which we leave as an exercise. We first need to introduce the basic notions of *unification* and *unfolding*.

We say that two atoms $R(\bar{u})$ and $P(\bar{v})$ *unify* if the there exists a function $\gamma : \mathsf{Const} \cup \mathsf{Var} \to \mathsf{Const} \cup \mathsf{Var}$, which is the identity on Const and the set of variables not mentioned in $\bar{u}$ and $\bar{v}$, such that $R(\gamma(\bar{u})) = P(\gamma(\bar{v}))$; γ is called a *unifier for* $R(\bar{u})$ *and* $P(\bar{v})$. Observe that for $R(\bar{u})$ and $P(\bar{v})$ to unify it is a necessary condition that R and P are the same relation names, and $\bar{u}, \bar{v}$ have the same arity. A *most general unifier* for $R(\bar{u})$ and $P(\bar{v})$ is a unifier γ for them such that, for every other unifier γ' for $R(\bar{u})$ and $P(\bar{v})$, there exists a function $\theta : \mathsf{Const} \cup \mathsf{Var} \to \mathsf{Const} \cup \mathsf{Var}$ such that $\gamma' = \theta \circ \gamma$. It is easy to show that, for any two atoms $R(\bar{u})$ and $P(\bar{v})$,

- if $R(\bar{u})$ and $P(\bar{v})$ unify, then there is a most general unifier for them, and
- if γ_1 and γ_2 are most general unifiers for $R(\bar{u})$ and $P(\bar{v})$, then, for every relational atom $S(\bar{w})$, it holds that $S(\gamma_1(\bar{w}))$ and $S(\gamma_2(\bar{w}))$ are the same up to variable renaming.

These facts allow us to refer to *the* most general unifier for $R(\bar{u})$ and $P(\bar{v})$.

We now proceed with the second basic notion, that is, the unfolding of a CQ with a Datalog program, which relies on unification. Let q be the CQ

$$\text{Answer} \coloneq R_1(\bar{u}_1), \ldots, R_n(\bar{u}_n)$$

and ρ be the Datalog rule

$$P_0(\bar{y}) \coloneq P_1(\bar{v}_1), \ldots, P_m(\bar{v}_m).$$

We can always assume that q and ρ do not share variables since we can simply rename the variables occurring in ρ without changing its semantic meaning. Assume now that, for $i \in [n]$, $R_i(\bar{u}_i)$ and $P_0(\bar{y})$ unify, and let γ be their most general unifier. The *unfolding of q with ρ using γ*, denoted $q_{\rho,\gamma}$, is the CQ

$$\begin{aligned}\text{Answer} \coloneq\ & R_1(\gamma(\bar{u}_1)), \ldots, R_{i-1}(\gamma(\bar{u}_{i-1})), \\ & R_{i+1}(\gamma(\bar{u}_{i+1})), \ldots, R_n(\gamma(\bar{u}_n)), P_1(\gamma(\bar{v}_1)), \ldots, P_m(\gamma(\bar{v}_m)).\end{aligned}$$

Let $\{R_{i_1}(\bar{u}_{i_1}), \ldots, R_{i_\ell}(\bar{u}_{i_\ell})\}$ be the set that collects all the relational atoms in the body of q that unify with $P_0(\bar{y})$, and let γ_{i_j} be the most general unifier for $R_{i_j}(\bar{u}_{i_j})$ and $P_0(\bar{y})$, for each $j \in [\ell]$. The *unfolding of q with ρ*, denoted $\text{Unfold}_\rho(q)$, is defined the set of CQs $\{q_{\rho,\gamma_{i_1}}, \ldots, q_{\rho,\gamma_{i_\ell}}\}$. Now, for a Datalog program Π, the *unfolding of q with Π*, denoted $\text{Unfold}_\Pi(q)$, is

$$\bigcup_{\rho \in \Pi} \text{Unfold}_\rho(q).$$

Therefore, $\text{Unfold}_\Pi(\cdot)$ can be seen as an operator that takes as input a CQ and computes all the CQs that can be obtained by unfolding q with the Datalog rules of Π. We can then define the set of all CQs that can be obtained starting from q and exhaustively applying the $\text{Unfold}_\Pi(\cdot)$ operator.

Definition 37.3: Application of the $\text{Unfold}_\Pi(\cdot)$ Operator

Consider a Datalog program Π over $\mathbf{S}$, and a CQ q over $\mathbf{S}$. We define

$$\text{Unfold}^0_\Pi(q) = \{q\} \qquad \text{and} \qquad \text{Unfold}^{i+1}_\Pi(q) = \bigcup_{q' \in \text{Unfold}^i_\Pi(q)} \text{Unfold}_\Pi(q')$$

for $i \geq 0$, and let

$$\text{Unfold}^\infty_\Pi(q) = \bigcup_{i \geq 0} \text{Unfold}^i_\Pi(q).$$

We are now ready to show Proposition 37.2.

Proof (of Proposition 37.2). We prove this for Boolean queries, i.e., we assume that the Datalog query $q = (\Pi, R)$ is Boolean, that is, $\text{ar}(R) = 0$. We further assume, without affecting the generality of the proof, that there is exactly one rule $\rho_R \in \Pi$ of the form $R() \coloneq$ body, and there is no rule in Π that mentions R in its body, i.e., we assume that the intensional relation name R occurs only in the head of ρ_R. We can indeed make this assumption since we can always

rewrite q into an equivalent Datalog query with the above property: construct Π^* by replacing every occurrence of the relation name R in Π with a new relation name R^* not occurring in $\mathsf{sch}(\Pi)$, and then consider the query

$$q^* \;=\; (\Pi^* \cup \{R() \coloneq R^*()\}, R)$$

It is clear that $\mathsf{edb}(\Pi) = \mathsf{edb}(\Pi^*)$ and $q(D) = q^*(D)$ for every D of $\mathsf{edb}(\Pi)$. In what follows, let q_R for the Boolean CQ such that $A_{q_R} = A_{\rho_R}$, i.e., q_R is the Boolean CQ that has as its body the body of the rule ρ_R.

We are now ready to define the desired infinitary UCQ over $\mathbf{S}$ by using the $\mathrm{Unfold}_\Pi(\cdot)$ operator. We write $\mathrm{Unfold}^\infty_\Pi(q_R)_{|\mathbf{S}}$ for the subset of $\mathrm{Unfold}^\infty_\Pi(q_R)$ that keeps only the CQs over $\mathbf{S}$, that is, the CQs that use only relation names from $\mathbf{S}$. We then define the infinitary UCQ

$$q^\Pi_R \;=\; \bigcup_{q' \in \mathrm{Unfold}^\infty_\Pi(q_R)_{|\mathbf{S}}} q'.$$

By construction, q^Π_R is an infinitary UCQ over $\mathbf{S}$, and Π and q^Π_R mention exactly the same constants. It remains to show that q and q^Π_R are equivalent, i.e, for every database D of $\mathbf{S}$, $q(D) = q^\Pi_R(D)$. To this end, it suffices to show that, for a database D of $\mathbf{S}$, the following are equivalent:

1. $A_{\rho_R} \to \Pi(D)$.
2. There exists a sequence of CQs $(q_i)_{i\in[0,n]}$, for some $n \geq 0$, such that:
   - $q_0 = q_R$,
   - $q_i \in \mathrm{Unfold}_\Pi(q_{i-1})$, for each $i \in [n]$, and
   - $A_{q_n} \to D$.

The Direction (1) $\Rightarrow$ (2)

By hypothesis, there exists a sequence of databases $(D_i)_{i\in[0,n]}$, for some $n \geq 0$, such that: (i) $D_0 = D$, (ii) for each $i \in [n]$, $D_i = D_{i-1} \cup \{P(\bar{a})\}$, where $P(\bar{a}) \in T_\Pi(D_{i-1})$, i.e., there is $\rho_{i-1} \in \Pi$ of the form $P(\bar{x}) \coloneq$ body such that $(A_{\rho_{i-1}}, \bar{x}) \to (D_{i-1}, \bar{a})$ via a homomorphism h_{i-1}, and (iii) $A_{\rho_R} \to D_n$. We proceed to show the following auxiliary lemma:

Lemma 37.4. *There exists a sequence of CQs $(q_i)_{i\in[0,n]}$ such that:*

- $q_0 = q_R$,
- $q_i = q_{i-1}$ *or* $q_i \in \mathrm{Unfold}_{\rho_{n-i}}(q_{i-1})$, *for each* $i \in [n]$, *and*
- $A_{q_i} \to D_{n-i}$, *for each* $i \in [n]$.

Proof. We proceed by induction on the length of $(D_i)_{i\in[0,n]}$. For the base case the statement holds trivially since $A_{\rho_R} \to D_0$, which in turn implies that $A_{q_R} \to D_0$. In other words, the desired sequence of CQs consists only of q_0.

We proceed with the inductive step. By hypothesis, $A_{\rho_R} \to D_n$ via a homomorphism μ, and thus, $A_{q_R} \to D_n$ via μ. We consider two cases:

- Assume first that $P(h_{i-1}(\bar{x})) \notin \mu(A_{A_{q_R}})$, or $P(h_{i-1}(\bar{x})) \in \mu(A_{A_{q_R}})$ and $P(h_{i-1}(\bar{x})) \in D_{n-1}$. This implies that $A_{q_R} \to D_{n-1}$. By induction hypothesis, there exists a sequence of CQs $(q'_i)_{i\in[0,n-1]}$, where $q'_0 = q_R$, and, for each $i \in [n-1]$, $q'_i = q'_{i-1}$ or $q'_i \in \text{Unfold}_{\rho_{n-i}}(q'_{i-1})$, and $A_{q_i} \to D_{n-i}$. Therefore, the claim follows due to the sequence of CQs $q'_0, q'_0, q'_1, \ldots, q'_{n-1}$.
- The interesting case is when $P(h_{n-1}(\bar{x})) \in D_n - D_{n-1}$. It is clear that there exists $P(\bar{u}) \in A_{q_R}$ such that $P(\mu(\bar{u})) = P(h_{n-1}(\bar{x}))$, which means that $\gamma = \mu \cup h_{n-1}$ is a unifier for $P(\bar{u})$ and $P(\bar{x})$. This implies that there exists a most general unifier $\hat{\gamma}$ for $P(\bar{u})$ and $P(\bar{x})$. Let $\hat{q}$ be the unfolding of q_R with ρ_{n-1} using $\hat{\gamma}$. We can show that $A_{\hat{q}} \to D_{n-1}$. By definition of most general unifiers, $\gamma = \theta \circ \hat{\gamma}$ for some function $\theta : \mathsf{Const} \cup \mathsf{Var} \to \mathsf{Const} \cup \mathsf{Var}$. It is clear that θ is a homomorphism from $A_{\hat{q}}$ to D_{n-1} since γ maps $A_{\rho_{n-1}}$ to D_{n-1}. Therefore, $A_{\hat{q}} \to D_{n-1}$ as claimed above.

  By induction hypothesis, there is a sequence of CQs $(q'_i)_{i\in[0,n-1]}$, where $q'_0 = \hat{q}$, and, for each $i \in [n-1]$, $q'_i = q'_{i-1}$ or $q'_i \in \text{Unfold}_{\rho_{n-1}}(q'_{i-1})$, and $A_{q_i} \to D_{n-1}$. The claim follows due to the sequence of CQs $q_R, q'_0, \ldots, q'_{n-1}$.

This completes the proof of Lemma 37.4. □

We can now complete the proof of the direction (1) ⇒ (2). Let $(q_i)_{i\in[0,n]}$ be the sequence of CQs provided by Lemma 37.4. Clearly, $q_0 = q_R$ and $A_{q_n} \to D_n$. However, it is not the case that $q_i \in \text{Unfold}_{\Pi}(q_{i-1})$, for each $i \in [n]$, due to the fact that some CQs in $(q_i)_{i\in[0,n]}$ are simply repeated. This can be easily fixed by removing the redundant CQs. Let Ind be the set of indices

$$\left\{ i_j \mid j \in [n] \text{ and } q_{i_j} \notin \text{Unfold}_{\Pi}(q_{i_j-1}) \right\}.$$

Observe that $0 \notin Ind$, and that, for each $k \in Ind$, $q_k = q_{k-1}$. Therefore, the sequence of CQs $(q'_i)_{i\in[0,m]}$, where $m \leq n$, obtained from $(q_i)_{i\in[0,n]}$ by simply removing the CQs $\{q_k \mid k \in Ind\}$ is such that $q'_0 = q_R$, $q'_i \in \text{Unfold}_{\Pi}(q'_{i-1})$, for each $i \in [m]$, and $A_{q'_m} \to D$. This implies that (2) holds, as needed.

The Direction (2) ⇒ (1)

We first establish the following auxiliary lemma:

Lemma 37.5. *For every $i \in [n]$, $A_{q_i} \to \Pi(D)$ implies $A_{q_{i-1}} \to \Pi(D)$.*

Proof. By hypothesis, there exists a homomorphism h that maps A_{q_i} to $\Pi(D)$. Since $q_i \in \text{Unfold}_{\Pi}(q_{i-1})$, we conclude that $q_i \in \text{Unfold}_{\rho}(q_{i-1})$ for some $\rho \in \Pi$ of the form $P(\bar{x})$:– body. This means that there exists an atom $P(\bar{u}) \in A_{q_{i-1}}$ that unifies with $P(\bar{x})$, and q_i is the unfolding of q_{i-1} with ρ using the most general unifier γ for $P(\bar{x})$ and $P(\bar{u})$. We show that the function $\mu = h \circ \gamma$ is a homomorphism from $A_{q_{i-1}}$ to $\Pi(D)$, which witnesses that $A_{q_{i-1}} \to \Pi(D)$.

Since h maps A_{q_i} to $\Pi(D)$, we get that h maps $\gamma(A_{q_{i-1}} - \{P(\bar{u})\})$ to $\Pi(D)$, i.e., μ maps $A_{q_{i-1}} - \{P(\bar{u})\}$ to $\Pi(D)$. It remains to show that $P(\mu(\bar{u})) \in \Pi(D)$.

Since $\gamma(A_\rho) \subseteq \gamma(A_{q_i})$, we conclude that h maps $\gamma(A_\rho)$ to $\Pi(D)$, i.e., μ is a homomorphism from A_ρ to $\Pi(D)$. This implies that $P(\mu(\bar{x})) \in \Pi(D)$. Observe that $P(\mu(\bar{x})) = P(\mu(\bar{u}))$. Indeed, since $\gamma(\bar{x}) = \gamma(\bar{u})$, we get that

$$P(\mu(\bar{x})) \;=\; P(h(\gamma(\bar{x}))) \;=\; P(h(\gamma(\bar{u}))) \;=\; P(\mu(\bar{u})),$$

which in turn implies that $P(\mu(\bar{u})) \in \Pi(D)$, and the claim follows. □

We can now complete the proof of the direction (2) ⇒ (1). By hypothesis, $A_{q_n} \to D$, and thus, $A_{q_n} \to \Pi(D)$; the latter holds due to the monotonicity of CQs (Corollary 14.7). By repeatedly applying Lemma 37.5, we get that $A_{q_0} \to \Pi(D)$. Since $q_0 = q_R$ and $A_{q_R} = A_{\rho_R}$, we conclude that $A_{\rho_R} \to \Pi(D)$. □

By Proposition 37.2, and the fact that infinitary UCQs are preserved under homomorphisms, we immediately get the following result:

Corollary 37.6

Every Datalog query is preserved under homomorphisms.

Another key property is that of monotonicity. Recall that a query q over a schema $\mathbf{S}$ is *monotone* if, for every two databases D, D' of $\mathbf{S}$, we have that

$$D \subseteq D' \;\text{ implies }\; q(D) \subseteq q(D').$$

We can show that homomorphism preservation implies monotonicity of Datalog queries. In fact, the proof is exactly the same as the one of Corollary 14.7, which establishes that every CQ is monotone.

Corollary 37.7

Every Datalog query is monotone.

Datalog Queries and Negation

We now delineate the expressiveness boundaries of Datalog queries. We show that they cannot express inequality, negative relational atoms, and difference.

Datalog queries cannot express inequality. This is because already CQs with inequality are not preserved under homomorphisms. Consider

$$q_1 \;=\; \exists x \exists y \left(R(x,y) \wedge x \neq y\right).$$

For $D = \{R(a,b)\}$ and $D' = \{R(c,c)\}$, we have that $D \to_\emptyset D'$. However, $D \models q_1$ while $D' \not\models q_1$. As a second example, consider the CQ$^{\neq}$

$$q_2 \;=\; \exists x\,(S(x) \wedge x \neq a),$$

where a is a constant. Given $D = \{S(b)\}$ and $D' = \{S(a)\}$, we have that $D \rightarrow_{\{a\}} D'$. However, $D \models q_2$ while $D' \not\models q_2$.

Datalog queries cannot express negative relational atoms. The reason is because such queries are not monotone. Consider the query

$$q \;=\; \neg P(a),$$

where a is a constant. If we take $D = \emptyset$ and $D' = \{P(a)\}$, then $D \subseteq D'$ but $D \models q$ while $D' \not\models q$.

Datalog queries cannot express difference. This is because difference is not monotone. Consider, for example, the FO query

$$q \;=\; \exists x(P(x) \wedge \neg Q(x)).$$

For $D = \{P(a)\} \subseteq D' = \{P(a), Q(a)\}$, we have that $D \models q$ while $D' \not\models q$.

38

Datalog Query Evaluation

In this chapter, we study the complexity of evaluating Datalog queries, that is, Datalog-Evaluation. This is the problem of checking whether $\bar{a} \in q(D)$ for a Datalog query q, a database D, and a tuple $\bar{a}$ over $\mathrm{Dom}(D)$.

Combined Complexity

We first look at the combined complexity of the problem, i.e., when the input consists of a Datalog query $q = (\Pi, R)$, a database D of $\mathsf{edb}(\Pi)$, and a tuple $\bar{a} \in \mathrm{Dom}(D)^{\mathrm{ar}(R)}$. Recall that the fixpoint approach for defining the semantics of Datalog programs provides an algorithm for computing the database $\Pi(D)$. In particular, by Corollary 36.14, $\Pi(D) = T_\Pi^\infty(D)$, which in turn implies that

$$\bar{a} \in q(D) \text{ if and only if } R(\bar{a}) \in T_\Pi^\infty(D).$$

We proceed to analyze the time complexity of checking whether the fact $R(\bar{a})$ belongs to $T_\Pi^\infty(D)$. Recall that for computing $T_\Pi^\infty(D)$ we need to apply the T_Π operator at most $|\mathsf{B}(\Pi, D)|$ times. It is easy to verify that

$$|\mathsf{B}(\Pi, D)| \leq |\mathsf{sch}(\Pi)| \cdot |\mathrm{Dom}(D)|^{\mathrm{ar}(\Pi)},$$

where $\mathrm{ar}(\Pi)$ is the maximum arity over all relation names of $\mathsf{sch}(\Pi)$. We now analyze the time complexity of the i-th application of the T_Π operator. Let *maxvar* and *maxbody* be the maximum number of variables and body atoms, respectively, in a rule of Π. The i-th application of T_Π takes time

$$O(|\Pi| \cdot |\mathrm{Dom}(D)|^{maxvar} \cdot maxbody \cdot |T_\Pi^{i-1}(D)|)$$

since, for each $\rho \in \Pi$, we need to consider all the possible functions h, which are the identity on Const, from the variables and constants in ρ to $\mathrm{Dom}(D)$, and then check whether h is a homomorphism from the set of atoms in the body of ρ to $T_\Pi^{i-1}(D)$. Recall that $|T_\Pi^{i-1}(D)| \leq |\mathsf{B}(\Pi, D)|$. Hence, each application of T_Π takes exponential time. Summing up, we need to apply the

T_Π operator exponentially many times, and each application takes exponential time. Consequently, $T_\Pi^\infty(D)$ can be computed in exponential time, which implies that checking if $R(\bar{a}) \in T_\Pi^\infty(D)$ is feasible in exponential time.

One may think that there is a more clever procedure than naively computing $T_\Pi^\infty(D)$ that allows us to show that the complexity of Datalog-Evaluation matches the complexity of UCQ-Evaluation, that is, NP-complete. However, we can show that exponential time is the best that we can achieve.

Theorem 38.1

Datalog-Evaluation is ExpTime-complete.

Proof. We have already seen that Datalog-Evaluation is in ExpTime. We proceed to show that Datalog-Evaluation is ExpTime-hard. This is done by showing that an arbitrary language L in ExpTime is polynomial time reducible to Datalog-Evaluation. Let $M = (Q, \Sigma, \delta, s)$ be a (deterministic) Turing Machine that decides L in exponential time; details on Turing Machines can be found in Appendix B. The goal is, on input w, to construct in polynomial time in $|w|$ a database D, and a Boolean Datalog query $q = (\Pi, \text{Yes})$, i.e., Yes is a 0-ary relation name, such that

$$M \text{ accepts } w \quad \text{ if and only if } \quad q(D) = \texttt{true}.$$

We first describe the high level idea of the reduction.

Consider a pair $(p, a) \in (Q - \{\text{“yes”}, \text{“no”}\}) \times \Sigma$. The transition rule $\delta(p, a) = (p', b, \text{dir})$ expresses the following if-then statement:

if at some time instant t of the computation of M on w, we have that M is in state p, the head points to the tape cell c, and c contains the symbol a

then at time instant $t+1$, we have that M is in state p', the cell c contains b, and the head points to the cell c', where c' is the cell right to c (respectively, the cell left to c, c itself) if dir $=\rightarrow$ (respectively, dir $=\leftarrow$, dir $= -$).

We can naturally encode such an if-then statement via Datalog rules since a Datalog rule is essentially an if-then statement. This in turn allows us to describe the complete evolution of M on input w from its start configuration $sc(w)$ to configuration c that can be reached in 2^m steps, where $m = |w|^k$ for some $k \in \mathbb{N}$. To achieve this, we need a way to refer to the i-th time instant of the computation of M on w, and the i-th tape cell of M, where $0 \leq i \leq 2^m - 1$. This can be done by representing the time instances and the tape cells from 0 to $2^m - 1$ by tuples of size m over $\{0, 1\}$, on which the functions “next time instant” and “next tape cell” are realized by means of a successor relation Succ^m from a linear order $\preceq^m$ on $\{0, 1\}^m$. We now formalize this description.

The Extensional and Intensional Schema

We begin by describing the extensional and intensional schema of Π. As we shall see, there will be relations Succ^i, First^i and Last^i, for each $i \in [m]$, which tell the successor, the first, and the last element from a linear order $\preceq^i$ on $\{0,1\}^i$, respectively, that will be inductively constructed by Π starting from Succ^1, First^1 and Last^1. The extensional schema $\mathsf{edb}(\Pi)$ is

$$\{\text{Succ}^1, \text{First}^1, \text{Last}^1\},$$

where Succ^1 is a binary relation name, and First^1, Last^1 are unary relation names. The intensional schema $\mathsf{idb}(\Pi)$ is defined as

$$\{\text{Symbol}_a \mid a \in \Sigma\} \cup \{\text{Head}\} \cup \{\text{State}_p \mid p \in Q\} \cup \{\text{Yes}\} \cup \bigcup_{i \in [2,m]} \{\text{Succ}^i, \text{First}^i, \text{Last}^i\} \cup \{\preceq^m\},$$

where the arity of the relations names Symbol_a, Head, and $\preceq^m$ is $2m$, of State_p is m, of Succ^i is $2i$, of First^i and Last^i is i, and of Yes is 0.

The intuitive meaning of the relation names of $\mathsf{idb}(\Pi)$, apart from Succ^i, First^i, Last^i, and $\preceq^m$ that have been discussed above, is as follows:

- $\text{Symbol}_a(t, c)$: at time instant t, the tape cell c contains the symbol a.
- $\text{Head}(t, c)$: at time instant t, the head points at cell c.
- $\text{State}_p(t)$: at time instant t, M is in state p.
- Yes(): M has reached an accepting configuration.

Having $\mathsf{edb}(\Pi)$ and $\mathsf{idb}(\Pi)$ in place, we can now proceed with the definition of the database D and the Datalog program Π.

The Database D

We only need to store the relations Succ^1, First^1, and Last^1, which form the base case of the inductive definition of Succ^i, First^i, and Last^i. In particular,

$$D \;=\; \{\text{Succ}^1(0,1), \text{First}^1(0), \text{Last}^1(1)\}.$$

The Program Π

The program Π, which is responsible for faithfully describing the evolution of M on w starting from $sc(w)$, is the union of the following five programs:

1. $\Pi_{\preceq}$ that inductively constructs Succ^i, $\preceq^i$, and First^i, for each $i \in [m]$.
2. Π_{start} that constructs the start configuration $sc(w) = (s, \triangleright, w, \sqcup, \ldots, \sqcup)$.
3. Π_{δ} that simulates the transition function of M.
4. Π_{inertia} that ensures that the tape cells that have not been changed at time instant t keep their values at time instant $t+1$.

5. Π_{accept} that checks whether M has reached an accepting configuration.

The definitions of the above Datalog program follow. For notational convenience, we write $\bar{x}$ for $x_1, \ldots, x_m$, and $\bar{x}_i$ for $x_{i,1}, \ldots, x_{i,m}$.

The Program $\Pi_{\preceq}$. For each $i \in [m-1]$, we add the Datalog rules:

$$\begin{aligned}
\text{Succ}^{i+1}(z, \bar{x}, z, \bar{y}) &:- \text{Succ}^{i}(\bar{x}, \bar{y}), \text{First}^{1}(z) \\
\text{Succ}^{i+1}(z, \bar{x}, z, \bar{y}) &:- \text{Succ}^{i}(\bar{x}, \bar{y}), \text{Last}^{1}(z) \\
\text{Succ}^{i+1}(z, \bar{x}, v, \bar{y}) &:- \text{Succ}^{1}(z, v), \text{Last}^{i}(\bar{x}), \text{First}^{i}(\bar{y}) \\
\text{First}^{i+1}(x, \bar{y}) &:- \text{First}^{1}(x), \text{First}^{i}(\bar{y}) \\
\text{Last}^{i+1}(x, \bar{y}) &:- \text{Last}^{1}(x), \text{Last}^{i}(\bar{y}) \\
\preceq^{m}(\bar{x}, \bar{y}) &:- \text{Succ}^{m}(\bar{x}, \bar{y}) \\
\preceq^{m}(\bar{x}, \bar{z}) &:- \preceq^{m}(\bar{x}, \bar{y}), \text{Succ}^{m}(\bar{y}, \bar{z}).
\end{aligned}$$

The Program $\Pi_{\textbf{start}}$. Assuming that $w = a_0, \ldots, a_{|w|-1}$, we add the rules:

$$\begin{aligned}
\text{State}_{s}(\bar{x}) &:- \text{First}^{m}(\bar{x}) \\
\text{Symbol}_{a_0}(\bar{x}, \bar{x}) &:- \text{First}^{m}(\bar{x}) \\
\text{Symbol}_{a_1}(\bar{x}_0, \bar{x}_1) &:- \text{First}^{m}(\bar{x}_0), \text{Succ}^{m}(\bar{x}_0, \bar{x}_1) \\
&\vdots \\
\text{Symbol}_{a_{|w|-1}}(\bar{x}_0, \bar{x}_i) &:- \text{First}^{m}(\bar{x}_0), \text{Succ}^{m}(\bar{x}_0, \bar{x}_1), \ldots, \text{Succ}^{m}(\bar{x}_{|w|-2}, \bar{x}_{|w|-1}) \\
\text{Symbol}_{\sqcup}(\bar{x}_0, \bar{y}) &:- \text{First}^{m}(\bar{x}_0), \text{Succ}^{m}(\bar{x}_0, \bar{x}_1), \ldots, \\
&\qquad \text{Succ}^{m}(\bar{x}_{|w|-2}, \bar{x}_{|w|-1}), \preceq^{m}(\bar{x}_{|w|-1}, \bar{y}) \\
\text{Head}(\bar{x}, \bar{x}) &:- \text{First}^{m}(\bar{x})
\end{aligned}$$

The Program Π_{δ}. For each pair $(p, a) \in (Q - \{\text{“yes”}, \text{“no”}\}) \times \Sigma$, with $\delta(p, a) = (p', b, \text{dir})$, we add the following Datalog rules. For brevity, let

$$\Phi_{(p,a)}(\bar{x}, \bar{y}, \bar{z}) = \text{State}_{p}(\bar{x}), \text{Head}(\bar{x}, \bar{y}), \text{Symbol}_{a}(\bar{x}, \bar{y}), \text{Succ}^{m}(\bar{x}, \bar{z}).$$

The following rules change the state from p to p', and the symbol from a to b at the next time instant of the computation:

$$\begin{aligned}
\text{State}_{p'}(\bar{z}) &:- \Phi_{(p,a)}(\bar{x}, \bar{y}, \bar{z}) \\
\text{Symbol}_{b}(\bar{z}, \bar{y}) &:- \Phi_{(p,a)}(\bar{x}, \bar{y}, \bar{z}).
\end{aligned}$$

The next rule, which is responsible for moving the head, depends on the direction $\text{dir} \in \{\rightarrow, \leftarrow, -\}$. In particular, if $\text{dir} = \rightarrow$, then we add the rule

$$\text{Head}(\bar{z}, \bar{v}) :- \Phi_{(p,a)}(\bar{x}, \bar{y}, \bar{z}), \text{Succ}^{m}(\bar{y}, \bar{v}).$$

If $\text{dir} = \leftarrow$, then we add the rule

$$\text{Head}(\bar{z}, \bar{v}) \ :- \ \Phi_{(p,a)}(\bar{x}, \bar{y}, \bar{z}), \text{Succ}^m(\bar{v}, \bar{y}).$$

Finally, if dir $= -$, then we add the rule

$$\text{Head}(\bar{z}, \bar{y}) \ :- \ \Phi_{(p,a)}(\bar{x}, \bar{y}, \bar{z}).$$

The Program $\Pi_{\textbf{inertia}}$. Recall that this program is responsible for, essentially, copying the content of the tape cells that have not been affected during the transition from time instant t to time instant $t + 1$. The following rule achieves this for the tape cells coming before the current cell

$$\text{Symbol}_a(\bar{v}, \bar{y}) \ :- \ \text{Symbol}_a(\bar{x}, \bar{y}), \text{Head}(\bar{x}, \bar{z}), \preceq^m (\bar{y}, \bar{z}), \text{Succ}^m(\bar{x}, \bar{v}).$$

The next rule does the same for the tape cells coming after the current cell

$$\text{Symbol}_a(\bar{x}, \bar{y}) \ :- \ \text{Symbol}_a(\bar{x}, \bar{y}), \text{Head}(\bar{x}, \bar{z}), \preceq^m (\bar{z}, \bar{y}), \text{Succ}^m(\bar{x}, \bar{v}).$$

The Program $\Pi_{\textbf{accept}}$. Finally, we check whether M has reached an accepting configuration via the Datalog rule

$$\text{Yes} \ :- \ \text{State}_{\text{“yes”}}(\bar{x}).$$

It is not difficult to verify that D and Π can be constructed from M and w in polynomial time. It is also not hard to see that Π faithfully describes the computation of M on input w. This means that, with $q = (\Pi, \text{Yes})$, M accepts w if and only if $q(D) = \texttt{true}$ (we leave the proof as an exercise). □

Data Complexity

We now concentrate on the data complexity of Datalog-Evaluation. As discussed in Chapter 2, when we study the data complexity of query evaluation, we essentially consider the query to be fixed, and only the database and the candidate output are considered as input. Formally, we are interested in the complexity of the problem q-Evaluation for a Datalog query q, which takes as input a database D and a tuple $\bar{a}$ over $\text{Dom}(D)$, and asks whether $\bar{a} \in q(D)$. As usual, by convention, we say that Datalog-Evaluation is $\mathcal{C}$-complete in data complexity for a complexity class $\mathcal{C}$ if q-Evaluation is in $\mathcal{C}$ for every Datalog query q, and there exists a Datalog query q such that q-Evaluation is $\mathcal{C}$-hard. We show that fixing the query has an impact on the complexity of the problem, that is, Datalog-Evaluation, from provably intractable, becomes tractable.

Theorem 38.2

Datalog-Evaluation is PTIME-complete in data complexity.

Proof. The upper bound follows from the analysis performed at the beginning of the chapter. Fix a Datalog query $q = (\Pi, R)$. Given a database D of $\mathsf{edb}(\Pi)$, and a tuple $\bar{a}$ over $\mathrm{Dom}(D)$, the analysis performed above shows that $T_\Pi^\infty(D)$ can be computed in time $O(|\mathrm{Dom}(D)|^k)$ for some $k \in \mathbb{N}$ that solely depends on q, which implies that checking whether $R(\bar{a}) \in T_\Pi^\infty(D)$ is feasible in time $O(|\mathrm{Dom}(D)|^k)$. Therefore, q-Evaluation is in PTime, as needed.

For the lower bound we provide a reduction from a standard PTime-hard problem known as *monotone circuit value*. For $n \in \mathbb{N}$, an *n-input, single-output monotone Boolean circuit* is a directed acyclic graph C with exactly n nodes without incoming edges, the *sources*, and exactly one node without outgoing edges, the *sink*. All the nodes that are not sources are labeled with either $\wedge$ or $\vee$ ($\neg$ is not allowed, hence the term monotone). We write $C(v_1, \ldots, v_n)$ to indicate that the i-th source of C is the node v_i, for $i \in [n]$. An *input* to such a Boolean circuit $C(v_1, \ldots, v_n)$ is a tuple $(w_1, \ldots, w_n) \in \{0,1\}^n$. The *output of* $C(v_1, \ldots, v_n)$ *on* $(w_1, \ldots, w_n)$, denoted $C(w_1, \ldots, w_n)$, is defined as expected. Formally, we recursively assign to every node v a value b_v as follows:

- $b_{v_i} = w_i$, for each $i \in [n]$, and
- for every node $u \notin \{v_1, \ldots, v_n\}$, assuming that the two incoming edges of u are coming from u_1 and u_2, $b_u = b_{u_1} \diamond b_{u_2}$, where $\diamond$ is the label of u.

The output $C(w_1, \ldots, w_n)$ is defined as b_{v_s}, where v_s is the sink of C. We are now ready to introduce the monotone circuit value problem:

Problem: MCVP

Input: An n-input, single-output monotone Boolean circuit $C(\bar{v})$, and a tuple $\bar{w} \in \{0,1\}^n$, where $n \in \mathbb{N}$

Output: `true` if $C(\bar{w}) = 1$, and `false` otherwise

Our goal is to show that there exists a Datalog query q such that MCVP can be reduced in q-Evaluation via a reduction that is computable in deterministic logarithmic space. Intuitively, the query q should specify a generic procedure for evaluating monotone Boolean circuits. This can be straightforwardly done via the query $q = (\Pi, \mathrm{Yes})$, where Π is the Datalog program

$$\begin{aligned}
\mathrm{True}(x) &\text{ :- } \mathrm{Or}(x,y,z), \mathrm{True}(y)\\
\mathrm{True}(x) &\text{ :- } \mathrm{Or}(x,y,z), \mathrm{True}(z)\\
\mathrm{True}(x) &\text{ :- } \mathrm{And}(x,y,z), \mathrm{True}(y), \mathrm{True}(z)\\
\mathrm{Yes} &\text{ :- } \mathrm{Sink}(x), \mathrm{True}(x).
\end{aligned}$$

We now proceed to show that MCVP can be reduced in q-Evaluation via a reduction that is computable in deterministic logarithmic space. Consider an instance of MCVP, i.e., an n-input, single-output monotone Boolean circuit

$C(v_1, \ldots, v_n)$, and a tuple $\bar{w} = (w_1, \ldots, w_n) \in \{0,1\}^n$, for an integer $n \in \mathbb{N}$. For brevity, we write $u_i = u_j \wedge u_k$ for the fact that the node u_i is labeled by $\wedge$, and its incoming edges are coming from the nodes u_j and u_k; analogously, we write $u_i = u_j \vee u_k$. We define the database $D_{C,\bar{w}}$ of $\mathsf{edb}(\Pi)$ as follows:

$$
\begin{aligned}
&\{\mathrm{True}(v_i) \mid i \in [n] \text{ and } w_i = 1\} \\
\cup\ &\{\mathrm{And}(u_i, u_j, u_k) \mid u_i, u_j, u_k \text{ are nonsource nodes of } C, \text{ and } u_i = u_j \wedge u_k\} \\
\cup\ &\{\mathrm{Or}(u_i, u_j, u_k) \mid u_i, u_j, u_k \text{ are nonsource nodes of } C, \text{ and } u_i = u_j \vee u_k\} \\
\cup\ &\{\mathrm{Sink}(v_s) \mid v_s \text{ is the sink of } C\}.
\end{aligned}
$$

It is clear that the database $D_{C,\bar{w}}$ can be computed in deterministic logarithmic space in the size of C and $\bar{w}$. It is also easy to verify that

$$C(\bar{w}) = 1 \quad \text{if and only if} \quad q(D_{C,\bar{w}}) = \mathtt{true}.$$

Therefore, q-Evaluation is PTIME-hard, and the claim follows. □

39

Static Analysis of Datalog Queries

In this chapter, we discuss central static analysis tasks for Datalog queries that are important for query optimization purposes. In fact, we consider the three fundamental tasks that we have also studied for first-order and conjunctive queries, namely satisfiability, containment, and equivalence. We also discuss a new static analysis task, known as boundedness, that is relevant for recursive query languages such as Datalog. In simple words, a Datalog query is bounded if it can be equivalently rewritten as a Datalog query without recursion.

As we shall see, we can effectively check whether a Datalog query is satisfiable. On the other hand, the problem of checking whether a Datalog query is contained into (or is equivalent to) another Datalog query, as well as the problem of checking whether a Datalog query is bounded, are undecidable.

Satisfiability

We start by considering the satisfiability problem: given a Datalog query $q = (\Pi, R)$, is there a database D of $\mathsf{edb}(\Pi)$ such that $q(D) \neq \emptyset$? We proceed to show that this problem is decidable:

Theorem 39.1

Datalog-Satisfiability is decidable.

Proof. We prove the result for constant-free Datalog queries, that is, Datalog queries such that the rules occurring in the program do not mention constants, and leave the general case as an exercise. Consider such a query $q = (\Pi, R)$. We first characterize the satisfiability of q via a very simple database. Let

$$D_\Pi = \{P(\star, \ldots, \star) \mid P \in \mathsf{edb}(\Pi)\},$$

where $\star$ is a value from Const. We can show the following lemma:

Lemma 39.2. *It holds that q is satisfiable if and only if $q(D_\Pi) \neq \emptyset$.*

Proof. It is clear that if $q(D_\Pi) \neq \emptyset$, then q is satisfiable witnessed by the simple database D_Π. Assume now that q is satisfiable. This implies that there exists a database D over $\mathsf{edb}(\Pi)$ such that $q(D) \neq \emptyset$. Let $h : \mathrm{Dom}(D) \to \{\star\}$ be the function that maps each constant occurring in D to $\star$. Clearly, $h(D) \subseteq D_\Pi$, which in turn allows us to show that $h(\Pi(D)) \subseteq \Pi(D_\Pi)$; the latter can be shown via an easy inductive argument. Therefore, $R(\star, \ldots, \star) \in \Pi(D_\Pi)$, which in turn implies that $\{\star\}^{\mathrm{ar}(R)} \in q(D_\Pi)$, and the claim follows. □

Lemma 39.2 leads to the following simple procedure for checking whether the Datalog query $q = (\Pi, R)$ is satisfiable:

$$\text{if } \{\star\}^{\mathrm{ar}(R)} \in q(D_\Pi), \text{then yes; otherwise, no.}$$

By Theorem 38.1, checking whether $\{\star\}^{\mathrm{ar}(R)} \in q(D_\Pi)$ is decidable, which in turn implies that Datalog-Satisfiability is decidable, and the claim follows. □

Containment and Equivalence

We now concentrate on the containment problem for Datalog: given two Datalog queries $q_1 = (\Pi_1, R_1)$ and $q_2 = (\Pi_2, R_2)$ over a schema $\mathbf{S}$ (in particular, with $\mathsf{edb}(\Pi_1) = \mathsf{edb}(\Pi_2)$), is it the case that $q_1 \subseteq q_2$. We can show that:

Theorem 39.3

Datalog-Containment is undecidable.

The above result is shown via a reduction from a known undecidable problem, namely containment for context-free grammars. A context-free grammar is a set of production rules that describe how to produce words over a certain alphabet. Consider, for example, the grammar G consisting of the rules

$$S \to AA \qquad A \to a \qquad A \to b.$$

The first rule states that we can replace S with AA, while the other two rules state that A can be replaced with a or b. Assuming that S is the starting point of the production, and $\{a, b\}$ is the underlying alphabet, the above grammar produces the words aa, ab, ba, bb. Let us formalize the above discussion.

A *context-free grammar* (CFG) is a tuple (N, T, P, S), where

- N is a finite set, the *non-terminal symbols*,
- T is a finite set disjoint from N, the *terminal symbols*,
- P is a finite subset of $N \times (N \cup T)^*$, the *production rules*, and
- $S \in N$, the *start symbol*.

For any two words $v, w \in (N \cup T)^*$, we say that v *directly yields* w, written $v \Rightarrow w$, if there exists $(x, y) \in P$ and $z_1, z_2 \in (N \cup T)^*$ such that $v = z_1 x z_2$ and $w = z_1 y z_2$. Now, for any two words $v, w \in (N \cup T)^*$, we say that v *yields* w, written as $v \Rightarrow^* w$, if there exists $k \geq 1$, and words $z_1, \ldots, z_k \in (N \cup T)^*$ such that $v = z_1 \Rightarrow z_2 \cdots \Rightarrow z_k = w$. The *language* of G, denoted $L(G)$, is the set of words $\{w \in T^* \mid S \Rightarrow^* w\}$, that is, all the words w over T that can be obtained starting from S and applying production rules of P.

Given two context-free grammars G_1 and G_2, we say that G_1 is *contained* in G_2, denoted $G_1 \subseteq G_2$, if $L(G_1) \subseteq L(G_2)$. The containment problem for context-free grammars is defined as expected:

Problem: CFG-Containment

Input: Two context-free grammars G_1 and G_2
Output: `true` if $G_1 \subseteq G_2$, and `false` otherwise

Proof (of Theorem 39.3). We provide a reduction from CFG-Containment to Datalog-Containment. In other words, given two context-free grammars G_1 and G_2, the goal is to construct two Datalog queries q_1 and q_2 such that $G_1 \subseteq G_2$ if and only if $q_1 \subseteq q_2$. We first explain how to transform a CFG into a Datalog query. Let us clarify that, in what follows, given a CFG $G = (N, T, P, S)$, we assume that P is a finite subset of $N \times ((N - \{S\}) \cup T)^* - \{\epsilon\}$, where ϵ denotes the empty string. In other words, there is no rule in P that produces the empty string, and the start symbol does not occur in the right-hand side of a rule. This does not affect the generality of our proof since CFG-Containment remains undecidable even with the above simplifying assumptions.

We proceed to define the Datalog program Π_G, where $G = (N, T, P, S)$ is a CFG. The extensional schema $\mathsf{edb}(\Pi_G)$ and intensional schema $\mathsf{idb}(\Pi_G)$ are

$$\{\text{Symbol}_A \mid A \in T\} \qquad \text{and} \qquad \{\text{Symbol}_A \mid A \in N\},$$

respectively, where all the relations are binary. For each production rule in P

$$(A, A_1 \cdots A_n),$$

for $n \geq 1$, we add to Π_G the Datalog rule

$$\text{Symbol}_A(x_1, x_{n+1}) \text{ :- } \text{Symbol}_{A_1}(x_1, x_2), \text{Symbol}_{A_2}(x_2, x_3), \ldots, \text{Symbol}_{A_n}(x_n, x_{n+1}).$$

We finally define the Datalog query $q_G = (\Pi_G, \text{Symbol}_S)$.

Example 39.4: From CFG to Datalog

Consider the CFG $G = (N, T, P, S)$, where $N = \{A, S\}$, $T = \{a, b\}$, and $P = \{(S, Aa), (A, abA), (A, aa)\}$. The Datalog program Π_G is

$$\begin{aligned}
\text{Symbol}_S(x_1, x_3) &\coloneq \text{Symbol}_A(x_1, x_2), \text{Symbol}_a(x_2, x_3) \\
\text{Symbol}_A(x_1, x_4) &\coloneq \text{Symbol}_a(x_1, x_2), \text{Symbol}_b(x_2, x_3), \text{Symbol}_A(x_3, x_4) \\
\text{Symbol}_A(x_1, x_3) &\coloneq \text{Symbol}_a(x_1, x_2), \text{Symbol}_a(x_2, x_3),
\end{aligned}$$

while the query $q_G = (\Pi_G, \text{Symbol}_S)$.

To show the correctness of the above construction, we need to introduce the notion of proof tree of an atom from a Datalog program. Roughly speaking, such a proof tree explains how an atom can be derived from a Datalog program, i.e., it provides a proof for that atom. As we shall see below, this notion is closely related to the notion of derivation tree in context-free languages that essentially explains how a word can be derived from a CFG.

Consider an atom $R(\bar{a})$, with $\bar{a} \in \mathsf{Const}^{\text{ar}(R)}$, and a Datalog program Π. A *proof tree of* $R(\bar{a})$ *from* Π is a labeled rooted tree $T = (V, E, \lambda)$, where λ is a function from V to the set of atoms that can be formed using relations from $\mathsf{sch}(\Pi)$ and constants from Const, such that

1. assuming that $v \in V$ is the root node of T, $\lambda(v) = R(\bar{a})$, and
2. for each internal node $v \in V$ with children $u_1, \ldots, u_n$ for $n \geq 1$, there exists a rule $\rho \in \Pi$ of the form $R_0(\bar{x}_0) \coloneq R_1(\bar{x}_1), \ldots, R_n(\bar{x}_n)$, and a function h from the constants and variables in ρ to Const, which is the identity on Const, such that $\lambda(v) = R_0(h(\bar{x}_0))$ and $\lambda(u_i) = R_i(h(\bar{x}_i))$, for each $i \in [n]$.

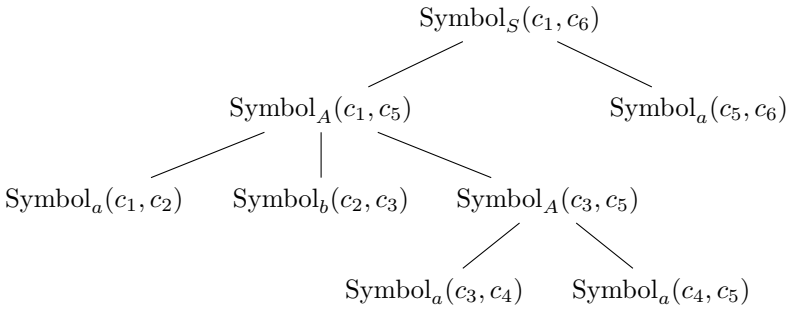
We say that the sequence of atoms $R_1(\bar{a}_1), \ldots, R_n(\bar{a}_n)$ is *induced* by the proof tree T if, assuming that the leaf nodes of T are $v_1, \ldots, v_n$ (in this order), then $\lambda(v_i) = R_i(\bar{a}_i)$, for each $i \in [n]$. Given a database D of $\mathsf{edb}(\Pi)$, *a proof tree of* $R(\bar{a})$ *from* Π *and* D is a proof tree $T = (V, E, \lambda)$ of $R(\bar{a})$ from Π such that, for each $v \in V$, $\lambda(v) \in \mathsf{B}(\Pi, D)$, i.e., $\lambda(v)$ is an atom with a relation from $\mathsf{sch}(\Pi)$ and constants from $\text{Dom}(D)$, and for each leaf node v of T, $\lambda(v) \in D$.

Example 39.5: Proof Tree

Consider the Datalog program Π_G obtained from the CFG G as in Example 39.4. A proof tree of the atom $\text{Symbol}_S(c_1, c_6)$ from Π_G and

$$\begin{aligned}
D \supseteq \{&\text{Symbol}_a(c_1, c_2), \text{Symbol}_b(c_2, c_3), \text{Symbol}_a(c_3, c_4), \\
&\text{Symbol}_a(c_4, c_5), \text{Symbol}_a(c_5, c_6)\}
\end{aligned}$$

is the following one



Clearly, the above proof tree induces the sequence of atoms

$$\text{Symbol}_a(c_1, c_2), \text{Symbol}_b(c_2, c_3), \text{Symbol}_a(c_3, c_4), \\ \text{Symbol}_a(c_4, c_5), \text{Symbol}_a(c_5, c_6).$$

It is an easy exercise to show the following lemma:

Lemma 39.6. *Consider a Datalog query $q = (\Pi, R)$, a database D of $\mathsf{edb}(\Pi)$, and a tuple $\bar{a} \in Dom(D)^{ar(R)}$. The following are equivalent:*

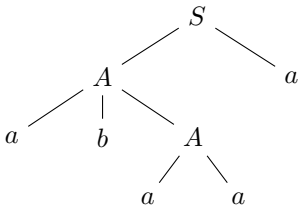
1. $\bar{a} \in q(D)$.
2. *There exists a proof tree of $R(\bar{a})$ from Π and D.*

The next technical lemma makes apparent the intention underlying the transformation of a CFG G to a Datalog program Π_G.

Lemma 39.7. *Consider a CFG $G = (N, T, P, S)$, and two words $a_1 \cdots a_n \in T^*$ and $c_1 \cdots c_{n+1} \in \mathsf{Const}^*$, for $n \geq 1$. The following are equivalent:*

1. $a_1 \cdots a_n \in L(G)$.
2. *There exists a proof tree of $\text{Symbol}_S(c_1, c_{n+1})$ from Π_G that induces the sequence of atoms $\text{Symbol}_{a_1}(c_1, c_2), \ldots, \text{Symbol}_{a_n}(c_n, c_{n+1})$.*

The proof of the above lemma, which is left as an exercise, relies on the correspondence between proof trees and derivation trees in context-free languages. For example, the following tree is a derivation tree of the word *abaaa* from the CFG G given in Example 39.4



The correspondence between the proof tree of the atom $\text{Symbol}_S(c_1, c_6)$ given above, and the derivation tree of the word *abaaa* should be apparent. By exploiting the above technical lemmas, we can now show the following result:

Proposition 39.8

Consider the CFGs $G_i = (N_i, T_i, P_i, S_i)$ for $i \in \{1, 2\}$. Then

$$G_1 \subseteq G_2 \quad \text{if and only if} \quad q_{G_1} \subseteq q_{G_2}.$$

Proof. We only show the 'if' direction; the 'only if' direction is shown analogously. Consider a database D of $\mathsf{edb}(\Pi_{G_1})$, and assume that $(c, d) \in q_{G_1}(D)$. By Lemma 39.6, there exists a proof tree of $\text{Symbol}_{S_1}(c, d)$ from Π_{G_1} and D. Assume that this proof tree induces the sequence of atoms

$$s \;=\; \text{Symbol}_{a_1}(c_1, c_2), \text{Symbol}_{a_2}(c_2, c_3), \ldots, \text{Symbol}_{a_n}(c_n, c_{n+1}),$$

where $c = c_1$, $d = c_{n+1}$, and $a_1 \cdots a_n \in T_1^*$. By Lemma 39.7, we conclude that $a_1 \cdots a_n \in L(G_1)$. Since, by hypothesis, $G_1 \subseteq G_2$, we get that $a_1 \cdots a_n \in L(G_2)$. By Lemma 39.7, there exists a proof tree of $\text{Symbol}_{S_2}(c_1, c_{n+1})$ from Π_{G_2} that induces the sequence of atoms s. Since the atoms in s are atoms of D, Lemma 39.6 implies that $(c, d) \in q_{G_2}(D)$, and the claim follows. □

Since CFG-Containment is undecidable, Proposition 39.8 implies the same for Datalog-Containment. This completes the proof of Theorem 39.3. □

Let us now turn our attention on the equivalence problem: given two Datalog queries q and q', is it the case that $q \equiv q'$. By exploiting the fact that the containment problem is undecidable, we can easily show the following:

Theorem 39.9

Datalog-Equivalence is undecidable.

Proof. It suffices to reduce Datalog-Containment to Datalog-Equivalence. Consider two Datalog queries $q_1 = (\Pi_1, R_1)$ and $q_2 = (\Pi_2, R_2)$, where $\mathsf{edb}(\Pi_1) = \mathsf{edb}(\Pi_2)$. We assume, without loss of generality, that $\mathsf{idb}(\Pi_1) \cap \mathsf{idb}(\Pi_2) = \emptyset$. We define the Datalog query

$$q_{12} \;=\; (\Pi_1 \cup \Pi_2 \cup \{R_{12}(\bar{x}) \text{ :- } R_1(\bar{x}),\ R_{12}(\bar{x}) \text{ :- } R_2(\bar{x})\}, R_{12}),$$

where R_{12} is a new relation not occurring in $\mathsf{sch}(\Pi_1) \cup \mathsf{sch}(\Pi_2)$. It is easy to verify that $q_1 \subseteq q_2$ if and only if $q_{12} \equiv q_2$, and the claim follows. □

Boundedness

As discussed in Chapter 36, given a Datalog program Π, and a database D of $\mathsf{edb}(\Pi)$, the semantics of Π on D, i.e., the database $\Pi(D)$, can be computed by repeatedly applying the immediate consequence operator T_Π of Π starting from D until a fixpoint is reached; in fact, by Corollary 36.14, $\Pi(D) = T_\Pi^\infty(D)$. We have also seen that the construction of $T_\Pi^\infty(D)$ does not require infinitely many iterations. Actually, there exists an integer $k \leq |\mathsf{B}(\Pi, D)|$ such that $T_\Pi^\infty(D) = T_\Pi^k(D)$. The smallest integer k such that $T_\Pi^\infty(D) = T_\Pi^k(D)$ is called the *stage of* Π *and* D, denoted $\mathsf{stage}(\Pi, D)$.

Given a Datalog program Π, it is generally the case that, for some arbitrary database D over $\mathsf{edb}(\Pi)$, the integer $\mathsf{stage}(\Pi, D)$ depends on both Π and D. This essentially means that the Datalog program Π is inherently recursive, or, in other words, the depth of recursion of Π is unbounded. On the other hand, if there is a uniform upper bound (i.e., a bound that depends only on Π) for $\mathsf{stage}(\Pi, D)$, then the recursion of Π is bounded, which actually means that Π is non-recursive despite the fact that syntactically may look recursive. The following example illustrates that a bounded (seemingly recursive) Datalog program can be replaced by an equivalent non-recursive Datalog program.

Example 39.10: Program Boundedness

Consider the Datalog program Π consisting of the rules

$$P(x, y) \text{ :- } R(x), P(z, y) \qquad P(x, y) \text{ :- } S(x, y).$$

Notice that Π is syntactically recursive due to the first rule (P depends on itself). However, Π is bounded, and equivalent to the program

$$P(x, y) \text{ :- } R(x), S(z, y) \qquad P(x, y) \text{ :- } S(x, y),$$

that is non-recursive. We can safely replace the atom $P(z, y)$ in the first rule with the atom $S(z, y)$, which leads to a non-recursive program, since it does not share any variable with the atom $R(x)$.

Boundedness for Datalog programs is defined as expected:

Definition 39.11: Program Boundedness

A Datalog program Π is *bounded* if there exists $k \in \mathbb{N}$ such that, for every database D of $\mathsf{edb}(\Pi)$, $\mathsf{stage}(\Pi, D) \leq k$.

As explained above, boundedness essentially removes from Datalog programs the feature of recursion. Therefore, it should not come as a surprise the fact that a Datalog query (Π, R), where Π is bounded, can always be written

as a UCQ. An interesting question at this point is whether the opposite holds, namely whenever a Datalog query (Π, R) is equivalent to a UCQ, then Π is bounded. It is easy to see that, in general, this is not the case.

Example 39.12: Program Boundedness and UCQs

Consider the Datalog query $q = (\Pi, R)$, where Π consists of the rules

$$\begin{aligned} P(x,y) &\text{ :- } S(x,y) \\ P(x,y) &\text{ :- } P(x,z), S(z,y) \\ R(x,y) &\text{ :- } S(x,y) \\ R(x,y) &\text{ :- } T(x,y). \end{aligned}$$

It is clear that Π is not bounded due to the first two rules that compute the transitive closure of the binary relation P. On the other hand, q is equivalent to the UCQ $q' = \varphi(x,y)$ with

$$\varphi \;=\; S(x,y) \;\vee\; T(x,y).$$

Indeed, for every database D of $\mathsf{edb}(\Pi) = \{S, T\}$, $q(D) = q'(D)$ since the relation name R depends only on S and T.

Observe that the key reason why the query $q = (\Pi, R)$ from Example 39.12 can be written as a UCQ, despite the fact that Π is not bounded, is because the relation name R does not depend on a recursive relation name, but only on non-recursive ones (in this case, on the extensional relation names S and T). This leads to the notion of boundedness of relation names.

Given a Datalog program Π, and a database D of $\mathsf{edb}(\Pi)$, analogously to the stage of Π and D, for a relation name $R \in \mathsf{idb}(\Pi)$ we define the *stage of R with respect to Π and D* as the smallest integer k with $R^{T_\Pi^\infty(D)} = R^{T_\Pi^k(D)}$, denoted $\mathsf{stage}_{\Pi,D}(R)$. Considering again the Example 39.12, although the stage of Π and D is not bounded, $\mathsf{stage}_{\Pi,D}(R) = 1$. This essentially tells that, even though the program Π may be inherently recursive, the part of it that is responsible for computing the relation R is actually non-recursive. We can now define when a Datalog query (instead of a program) is bounded.

Definition 39.13: Query Boundedness

A Datalog query $q = (\Pi, R)$ is *bounded* if there exists a $k \in \mathbb{N}$ such that, for every database D of $\mathsf{edb}(\Pi)$, $\mathsf{stage}_{\Pi,D}(R) \leq k$.

The notion of query boundedness essentially removes from Datalog queries the feature of recursion. Therefore, it should be expected that a bounded Datalog query (Π, R) can always be written as a UCQ, even if Π is not bounded.

What is more interesting, though, is the fact that query boundedness, unlike program boundedness, characterises the fragment of Datalog queries that can be written, not only as UCQs, but actually as FO queries.

Theorem 39.14

Let $q = (\Pi, R)$ be a Datalog query over $\mathbf{S}$. The following are equivalent:

1. q is bounded.
2. There exists an FO query q' over $\mathbf{S}$ such that $q \equiv q'$.
3. There exists a UCQ q'' over $\mathbf{S}$ such that $q \equiv q''$.

Proof. We discuss how (1) implies (2) can be shown, and leave the formal proof as an exercise. We know from Proposition 37.2 that there is an infinitary UCQ q' over $\mathbf{S}$ such that $q \equiv q'$. Recall that q' is defined by exhaustively applying the $\mathrm{Unfold}_\Pi(\cdot)$ operator, introduced in Chapter 37, starting from a certain CQ q_R obtained from q, and then keeping only the infinitary CQs over $\mathbf{S}$.[1] Now, in the case of bounded Datalog queries, it can be shown that q' is an ordinary UCQ over $\mathbf{S}$, and thus, an FO query over $\mathbf{S}$. This is due to the fact that the $\mathrm{Unfold}_\Pi(\cdot)$ operator, starting from q_R, it constructs a CQ over $\mathbf{S}$ after finitely many steps, in fact, in at most k steps, where $k \geq 0$ is the integer that bounds $\mathsf{stage}_{\Pi,D}(R)$, for every database D of $\mathsf{edb}(\Pi)$.

We now proceed to show the direction (2) implies (3). By hypothesis, there is an FO query q' over $\mathbf{S}$ such that $q \equiv q'$. By Corollary 37.6, we get that q' is preserved under homomorphisms. This in turn implies, due to Theorem 30.11, that there is a UCQ with variable-constant equality $\hat{q}$ over $\mathbf{S}$ such that $q' \equiv \hat{q}$. It remains to explain how the equational atoms in $\hat{q}$ can be eliminated. Since $q \equiv \hat{q}$, by Proposition 36.17, we get that, for every database D of $\mathbf{S}$, $\hat{q}(D)$ consists of tuples over $\mathrm{Dom}(D)$, i.e., it is not possible to have a value in the output of $\hat{q}$ on D that occurs in $\hat{q}$ but not in D. Assuming that $\hat{q} = \varphi(\bar{x})$, we then conclude the following: if a variable y in φ occurs in an equational atom, but not in a relational atom, then y is not among the free variables of φ, i.e., y is not mentioned in $\bar{x}$. This observation allows us to eliminate an equational atom $(y = a)$ from φ by simply replacing each occurrence of y with a, and then removing $(y = a)$ from φ. This eventually leads to a UCQ q'' over $\mathbf{S}$ such that $\hat{q} \equiv q''$, and thus, $q \equiv q''$, as needed.

We finally show that (3) implies (1). Consider an arbitrary database D of $\mathsf{edb}(\Pi)$, and a tuple $\bar{a}$ over $\mathrm{Dom}(D)$. It suffices to show that $R(\bar{a}) \in \Pi(D)$ implies $R(\bar{a}) \in T_\Pi^k(D)$ for some $k \in \mathbb{N}$ that does not depend on D and $\bar{a}$. Indeed, if this is the case, then, for every database D of $\mathsf{edb}(\Pi)$, $\mathsf{stage}_{\Pi,D}(R) \leq k$, which means that q is bounded. By hypothesis, there exists a UCQ q'', let say of the form $q_1 \cup \cdots \cup q_n$, such that $q \equiv q''$. Therefore, if $R(\bar{a}) \in \Pi(D)$, which

[1] Note that the proof of Proposition 37.2 was given for Boolean queries, but it can be extended to non-Boolean queries; this extension was left as an exercise.

means that $\bar{a} \in q(D)$, then $\bar{a} \in q''(D)$. Let m be the maximum number of atoms occurring in the CQs $q_1, \ldots, q_n$, that is, $m = \max_{i \in [n]}\{|A_{q_i}|\}$. It is clear that there is a database $D' \subseteq D$ with $|D'| \leq m$ such that $\bar{a} \in q''(D')$. Since $q \equiv q''$, $\bar{a} \in q(D')$, which implies that $R(\bar{a}) \in \Pi(D')$ or $R(\bar{a}) \in T_\Pi^\infty(D')$. Recall that $T_\Pi^\infty(D') = T_\Pi^{|\mathsf{B}(\Pi,D')|}(D')$ with $|\mathsf{B}(\Pi, D')| \leq |\mathsf{sch}(\Pi)| \cdot |\mathrm{Dom}(D')|^{\mathrm{ar}(\Pi)}$, where $\mathrm{ar}(\Pi)$ is the maximum arity over all relation names of $\mathsf{sch}(\Pi)$. Since $|D'| \leq m$, we get that $|\mathrm{Dom}(D')| \leq m \cdot \mathrm{ar}(\Pi)$. Observe that $T_\Pi^{|\mathsf{B}(\Pi,D')|}(D') \subseteq T_\Pi^{|\mathsf{B}(\Pi,D')|}(D)$. Therefore, $R(\bar{a}) \in T_\Pi^k(D)$ for some $k \in \mathbb{N}$ that does not depend on D and $\bar{a}$ (it only depends on q'' and Π), and the claim follows. □

It is clear that checking whether a Datalog program or query is bounded are important static analysis tasks that are relevant for optimization purposes.

Problem: Datalog-PBoundedness

Input: A Datalog program Π
Output: `true` if Π is bounded, and `false` otherwise

Problem: Datalog-QBoundedness

Input: A Datalog query q
Output: `true` if q is bounded, and `false` otherwise

It turns out that both problems are undecidable. It can be shown via a reduction from the *Post Correspondence Problem*, a classical undecidable problem, that checking whether a Datalog program is bounded is undecidable. This can be then easily transferred to query boundedness via an easy reduction. Consider a Datalog program Π. We define the Datalog query $q = (\Pi \cup \{\rho\}, R)$, where, assuming that $\mathsf{idb}(\Pi) = \{P_1, \ldots, P_n\}$, ρ is the Datalog rule

$$R(x_1^1, \ldots, x_{\mathrm{ar}(P_1)}^1, \ldots, x_1^n, \ldots, x_{\mathrm{ar}(P_n)}^n) \coloneq P_1(x_1^1, \ldots, x_{\mathrm{ar}(P_1)}^1), \ldots, P_n(x_1^n, \ldots, x_{\mathrm{ar}(P_n)}^n)$$

and R is a $(\mathrm{ar}(P_1) + \cdots + \mathrm{ar}(P_n))$-ary relation name not occurring in $\mathsf{idb}(\Pi)$. It is easy to see that Π is bounded iff q is bounded. We then have that:

Theorem 39.15

Datalog-PBoundedness and Datalog-QBoundedness are undecidable.

40

Exercises for Part IV

Exercise 4.1. Prove that query evaluation for $\exists\text{FO}^+$ and RA^+ is in NP.

Exercise 4.2. Prove that SPJU-Containment containment is Π_2^p-hard.

Exercise 4.3. Prove that containment for $\exists\text{FO}^+$ and RA^+ is Π_2^p-complete.

Exercise 4.4. Prove that $\text{CQ}^{\neq}$-Containment is Π_2^p-hard.

Exercise 4.5. The language $\text{CQ}^<$ is defined in the same way as $\text{CQ}^{\neq}$ (see Definition 32.1), but instead of $\neq$ we use $<$, assuming that there is an order on the set of constant Const from which database entries are drawn. Analogously, we can define the language $\text{UCQ}^<$. Prove that the problem of containment remains decidable for $\text{CQ}^<$ and $\text{UCQ}^<$.

Exercise 4.6. The class BCCQ consists of Boolean combinations of CQs, i.e., queries obtained by repeatedly applying the operations of union ($q_1 \cup q_2$), intersection ($q_1 \cap q_2$), and difference ($q_1 - q_2$) to CQs of the same arity with the obvious semantics. Prove that containment for BCCQs is decidable.

Exercise 4.7. Prove that the sentences $\psi_k(R\bar{S})$ and $\psi_k(\bar{R}S)$, used in the proof of Theorem 33.5, are true in almost all databases of $\mathbf{S} = \{R[1], S[1]\}$.

Exercise 4.8. In general, Theorem 33.5 (0–1 law) does not hold if we focus on a restricted class C of databases, i.e., for an FO sentence φ, $\mu_n(\varphi)$ is defined by considering only databases from the class C. Let $C_\subseteq$ be the class of databases D of the schema $\mathbf{S} = \{R[1], S[1]\}$ such that $S^D \subseteq R^D$. Adapt the proof of Theorem 33.5, given in Chapter 32, to show that the 0–1 law holds even if we focus on the class of databases $C_\subseteq$. Use this result to show that the *parity* query q over $\mathbf{S}$, which checks whether the cardinality of the relation B is even, is not expressible as an FO query if we focus on the class of databases $C_\subseteq$.

Exercise 4.9. Use Exercise 4.8 (not just the result but the proof that you produced) to infer the following: for every Boolean FO query q over the schema

$\mathbf{S} = \{R[1], S[1]\}$, there exist numbers $k \geq 0$ and $m \geq 0$ such that, for every database D of $\mathbf{S}$, $D \models q$ iff $|S^D| \geq k$ and $|R^D - S^D| \geq m$.

Use this fact to derive Theorem 33.7 when $\diamond$ is $=$, i.e., to show that the query $q_=$ over $\mathbf{S}$, which checks whether, for a database D of $\mathbf{S}$, $|R^D| = |S^D|$, cannot be expressed as a constant-free FO query over $\mathbf{S}$.

Exercise 4.10. Let $\varphi_\geq()$ be the constant-free FO query over $\mathbf{S} = \{R[1], S[1]\}$ such that, for every database D of $\mathbf{S}$, $D \models \varphi_\geq()$ iff $|R^D| \geq |S^D|$. Analogously, we define the constant-free FO queries $\varphi_>()$ and $\varphi_=()$ over $\mathbf{S}$. Show that

$$\lim_{n\to\infty} \mu_n(\varphi_\geq) = \frac{1}{2} \quad \text{and} \quad \lim_{n\to\infty} \mu_n(\varphi_>) = \frac{1}{2}.$$

Also show that

$$\lim_{n\to\infty} \mu_n(\varphi_=) = 0.$$

Exercise 4.11. Recall the estimate used in the proof of Theorem 33.7

$$F_n^= = \sum_{k\leq\lfloor n/2\rfloor} \binom{n}{k}\binom{n-k}{k}.$$

Show that

$$\lim_{n\to\infty} \frac{F_n^=}{3^n} \;=\; 0.$$

Exercise 4.12. Recall that Theorem 33.5 (0–1 law) was shown for the speacial case of constant-free FO sentences over a schema with two unary relation names. Prove the result for the schema $\mathbf{S} = \{E[2]\}$ with a single binary relation name (i.e., for undirected graphs). Recall that you need to construct a theory T which has a unique, up to isomorphism, countable model, and whose sentences are true in almost all databases of $\mathbf{S}$. Such a theory T has sentences $\psi_{k,m}$ that express the following: for every two disjoint sets X and Y of nodes of cardinalities k and m, respectively, there exists a node z such that there are edges (z, x) for each $x \in X$, and there is no edge (z, y) for $y \in Y$. While the proof of condition 2 of Lemma 33.6 follows the same ideas as those we saw, the proof of condition 1 is more elaborate and requires ideas not seen in this book; the interested reader is advised to consult [11].

Exercise 4.13. Give an example of an FO sentence *with constants* for which Theorem 33.5 (0–1 law) does not hold. A much more difficult task is to show that, for every FO sentence φ with constants, $\lim_{n\to\infty} \mu_n(\varphi)$ exists, and is a rational number with the denominator being of the form 2^k for some $k \geq 0$.

Exercise 4.14. Recall that Theorem 35.2 was shown for $\mathrm{RA}_{\mathsf{Aggr}}(\Omega)$ without constants. Extend the proof to the version of $\mathrm{RA}_{\mathsf{Aggr}}(\Omega)$ that allows the use of constants from Const. This is done in the following four steps:

1. First extend $\mathbf{L_C}$ and $\mathbb{L}_\mathbf{C}$ with constants from Const, and show that a translation of an $\mathrm{RA}_{\mathsf{Aggr}}(\Omega)$ expression that uses constants from $\{c_1, \ldots, c_n\} \subsetneq \mathsf{Const}$ into $\mathbf{L_C}$, and then into $\mathbb{L}_\mathbf{C}$, can be carried out in such a way that only constants from $\{c_1, \ldots, c_n\}$ are present in formulae.
2. Next, prove a locality result for the extension of $\mathbb{L}_\mathbf{C}$ with constants. Consider an expression $\varphi(\bar{x})$, where φ is an $\mathbb{L}_\mathbf{C}$ formula using constants from $\{c_1, \ldots, c_n\}$ over a schema $\mathbf{S} = \{R_1 : \boldsymbol{\tau}_1, \ldots, R_n : \boldsymbol{\tau}_n\}$ with $\boldsymbol{\tau}_i \in \{\mathsf{o}\}^{k_i}$ and $k_i \geq 0$, for each $i \in [n]$, such that $\mathrm{FV}(\varphi) \subseteq \mathsf{Var}_\mathsf{o}$, and $\bar{x}$ is a tuple over $\mathrm{FV}(\varphi)$ that mentions all the variables of $\mathrm{FV}(\varphi)$. There exists $r \geq 0$ such that, for a database D of $\mathbf{S}$, and tuples $\bar{a}, \bar{b}$ over $\mathrm{Dom}(D)$, if $N_r^D(\bar{a}, \bar{c})$ is isomorphic to $N_r^D(\bar{b}, \bar{c})$, for $\bar{c} = (c_1, \ldots, c_n)$, then either both $\bar{a}, \bar{b}$ belong to $\varphi(\bar{x})(D)$, or none of them belongs to $\varphi(\bar{x})(D)$.
3. Finally, using the above locality result for the extension of $\mathbb{L}_\mathbf{C}$ with constants, show that a modified version of the reachability query (that incorporates constants from $\{c_1, \ldots, c_n\}$) still violates locality.

Exercise 4.15. Use the translation of Theorem 35.7 to find syntactic restrictions on $\mathrm{FO}_{\mathsf{Aggr}}(\Omega)$ that lead to a logical formalism that can be used to define a query language, i.e., to ensure that the output of an expression $\varphi(\bar{u})$, where φ is a formula from the restricted formalism, and $\bar{u}$ a tuple of variables over $\mathrm{FV}(\varphi)$ that mentions all the variables of $\mathrm{FV}(\varphi)$, on a database is always finite.

Exercise 4.16. Recall that Proposition 35.12 was shown for the schema $\mathbf{S} = \{R : (\mathsf{o}, \mathsf{o})\}$. Generalize the proof to arbitrary schemas $\{R_1 : \boldsymbol{\tau}_1, \ldots, R_n : \boldsymbol{\tau}_n\}$ with $\boldsymbol{\tau}_i \in \{\mathsf{o}\}^{k_i}$ and $k_i \geq 0$, for each $i \in [n]$.

Exercise 4.17. There is a different notion of locality, known in the literature as *Hanf-locality* (as opposed to *Gaifman-locality* used in Chapter 35). Given two databases D and D' of a two-sorted schema $\mathbf{S}$, and tuples $\bar{a}$ and $\bar{a}'$ of the same arity over $\mathrm{Dom}(D)$ and $\mathrm{Dom}(D')$, respectively, we write $(D, \bar{a}) \sim_r (D', \bar{a}')$ if there is a bijection $f : \mathrm{Dom}(D) \to \mathrm{Dom}(D')$ such that, for every $b \in \mathrm{Dom}(D)$, $N_r^D(\bar{a}, b)$ is isomorphic to $N_r^{D'}(\bar{a}', f(b))$. Note that, in particular, $(D, \bar{a}) \sim_r (D', \bar{a}')$ implies $|\mathrm{Dom}(D)| = |\mathrm{Dom}(D')|$. A query q of type $\boldsymbol{\tau} \in \{\mathsf{o}, \mathsf{n}\}^k$, for $k \geq 0$, over $\mathbf{S}$ is *Hanf-local* if there exists $r \geq 0$ such that, for every $\bar{a}, \bar{a}' \in (\mathsf{Const} \cup \mathsf{Num})^k$, $(D, \bar{a}) \sim_r (D', \bar{a}')$ implies $\bar{a} \in q(D)$ iff $\bar{a}' \in q(D')$.

Prove Theorem 35.4 for Hanf-locality instead of Gaifman locality. To do so, show, by extending the argument in the proof of Proposition 35.12, the following: for an $\mathbb{L}_\mathbf{C}$ formula φ over a schema $\mathbf{S} = \{R_1 : \boldsymbol{\tau}_1, \ldots, R_n : \boldsymbol{\tau}_n\}$ with $\boldsymbol{\tau}_i \in \{\mathsf{o}\}^{k_i}$ and $k_i \geq 0$, for each $i \in [n]$, such that $\mathrm{FV}(\varphi) \subseteq \mathsf{Var}_\mathsf{o}$, and a tuple $\bar{x}$ over $\mathrm{FV}(\varphi)$ that mentions all the variables of $\mathrm{FV}(\varphi)$, $\varphi(\bar{x})$ is Hanf-local.

Exercise 4.18. Consider a restriction of $\mathbf{L_C}$ where infinitary connectives are not allowed, we have quantification over variables of both types, and we have counting terms $\sharp x\, \varphi(x, \bar{y})$ counting the number of elements of the input database satisfying φ. Concerning the semantics, is defined in the expected

way with the crucial restriction that numerical variables can only range over the values $[0, n-1]$, where n is the number of elements in the input database.

Prove that, for every expression $\varphi()$, where φ is a sentence from this restricted logic, there exists a Boolean $\mathrm{RA}_{\mathsf{Aggr}}(\Omega)$ query e, where Ω contains $<$, $+$, and $\cdot$, and the summation aggregate $\sum$, such that, for every database D of a schema $\mathbf{S}$, $\varphi()(D) = e(D)$ for the following two cases:

1. $\mathbf{S} = \{R_1 : \boldsymbol{\tau}_1, \ldots, R_n : \boldsymbol{\tau}_n\}$ such that, for each $i \in [n]$, $\boldsymbol{\tau}_i \in \{\mathsf{n}\}^{k_i}$ for $k_i \geq 0$, i.e., we focus on databases where all elements are of type n.
2. $\mathbf{S} = \{R_1 : \boldsymbol{\tau}_1, \ldots, R_n : \boldsymbol{\tau}_n\}$ such that, for each $i \in [n]$, $\boldsymbol{\tau}_i \in \{\mathsf{o}\}^{k_i}$ for $k_i \geq 0$, i.e., we focus on databases where all elements are of type o, and we have access to an order relation $<$ over the constants of Const.

The importance of these results stems from the fact the restricted logic we defined captures a uniform version of a complexity class called TC^0 (this stands for threshold circuits of constant depth). TC^0 has not been separated from others above it such as PTime or NLogSpace. In particular, this means that bounds on the expressivity of $\mathrm{RA}_{\mathsf{Aggr}}(\Omega)$ cannot be proved either over ordered non-numerical domains, or numerical domains, without resolving deep problems in complexity theory.

Exercise 4.19. A *path system* is a tuple $P = (V, R, S, T)$, where V is a finite set of nodes, $R \subseteq V \times V \times V$, $S \subseteq V$ and $T \subseteq V$. A node $v \in V$ is *admissible* if $v \in T$, or there are admissible nodes $u, w \in V$ such that $(v, u, w) \in R$.

Show that the set of admissible nodes for P can be computed via a Datalog query. In other words, there exists a Datalog query $q = (\Pi, A)$, where A is unary relation, such that, for every path system P, $q(D_P)$ is the set of admissible nodes for P, where D_P stores P in the obvious way, i.e., V, S and T via unary relations, and R via a ternary relation.

Exercise 4.20. An undirected graph $G = (V, E)$ is *2-colorable* if there exists a function $f : V \to \{0, 1\}$ such that $(v, u) \in E$ implies $f(v) \neq f(u)$.

Show that *non-2-colorability* is expressible via a Datalog query, i.e., there exists a Datalog query $q = (\Pi, \mathrm{Yes})$, where Yes is a 0-ary relation, such that, for every undirected graph G, $q(D_G) = \mathtt{true}$ if and only if G is *not* 2-colorable, where D_G stores G via the binary relation $\mathrm{Edge}(\cdot, \cdot)$.

Exercise 4.21. Prove Theorem 36.8.

Exercise 4.22. Prove Lemma 36.10.

Exercise 4.23. Show that the query that asks whether an undirected graph is 2-colorable is not expressible via a Datalog query. To do so, exploit the fact that Datalog queries are monotone.

Exercise 4.24. Prove that there exists a monotone query that is not expressible via a Datalog query. The proof should not rely on any complexity-theoretic assumption. (It is very easy to show that this holds under the assumption that PTime $\neq$ NP.)

Exercise 4.25. A Datalog program Π is called *linear* if, for every rule $\rho \in \Pi$, the body of ρ mentions at most one relation from $\mathsf{idb}(\Pi)$. A Datalog query (Π, R) is linear if Π is linear.

Show that there is no linear Datalog query that computes the set of admissible nodes for a path system P. The proof should not rely on any complexity-theoretic assumption. (It is easier to show this statement if we assume that NLogSpace $\neq$ PTime.

Exercise 4.26. The *predicate graph* of a program Π is the directed graph $G_\Pi = (V, E)$, where V consists of the relations of $\mathsf{sch}(\Pi)$, and $(P, R) \in E$ if and only if there exists a rule $\rho \in \Pi$ of the form

$$R(\bar{x}) \text{ :- } \ldots, P(\bar{y}), \ldots$$

We call Π *non-recursive* if G_Π is acyclic. A Datalog query (Π, R) is non-recursive if Π is non-recursive.

Show that, for every non-recursive Datalog query $q = (\Pi, R)$, there exists a finite UCQ q' such that, for every database D of $\mathsf{edb}(\Pi)$, $q(D) = q'(D)$.

Exercise 4.27. Let D and q be the database and the Datalog query, respectively, constructed from the Turing machine M and the input word w in the proof of Theorem 38.1. Show that M accepts w if and only if $q(D) = \texttt{true}$.

Exercise 4.28. A Datalog program Π is called *guarded* if, for every rule $\rho \in \Pi$, the body of ρ has an atom that contains (or "guards") all the variables occurring in ρ. A Datalog query (Π, R) is guarded if Π is guarded.

Prove that the ExpTime-hardness shown in Theorem 38.1 holds even if we focus on guarded Datalog queries.

Exercise 4.29. Consider a Datalog query $q = (\Pi, R)$, where the arity of the relations of $\mathsf{sch}(\Pi)$ is bounded by some integer constant, a database D of $\mathsf{edb}(\Pi)$, and a tuple $\bar{a}$ of arity $\mathrm{ar}(R)$ over $\mathrm{Dom}(D)$. Show that the problem of deciding whether $\bar{a} \in q(D)$ is NP-complete.

Exercise 4.30. Consider a Datalog query $q = (\Pi, R)$, where $\mathsf{sch}(\Pi)$ consists only of 0-ary relations, and a database D of $\mathsf{edb}(\Pi)$. Show that the problem of deciding whether $q(D) = \texttt{true}$ is PTime-complete.

Exercise 4.31. Show that the evaluation problem for linear Datalog queries is PSpace-complete in combined complexity, and NLogSpace-complete in data complexity.

Exercise 4.32. Show that the evaluation problem for non-recursive Datalog queries is PSpace-complete in combined complexity, and in DLogSpace in data complexity.

References

1. S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.
2. S. Arora and B. Barak. *Computational Complexity - A Modern Approach*. Cambridge University Press, 2009.
3. D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Y. Vardi. Containment of conjunctive regular path queries with inverse. In *KR 2000, Principles of Knowledge Representation and Reasoning Proceedings of the Seventh International Conference, Breckenridge, Colorado, USA, April 11-15, 2000.*, pages 176–185, 2000.
4. K. Etessami, M. Y. Vardi, and T. Wilke. First-order logic with two variables and unary temporal logic. *Inf. Comput.*, 179(2):279–295, 2002.
5. H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database Systems - the Complete Book*. Pearson Education, 2009.
6. E. Grädel, P. Kolaitis, L. Libkin, M. Marx, J. Spencer, M. Vardi, and S. Weinstein. *Finite Model Theory and its Applications*. Springer, 2008.
7. P. R. Halmos. *Measure Theory*. Springer, 1974.
8. J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 2003.
9. N. Immerman. *Descriptive Complexity*. Springer, 1999.
10. D. Kozen. *Automata and Computability*. Springer, 1997.
11. L. Libkin. *Elements of Finite Model Theory*. Springer, 2004.
12. C. H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.
13. R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw-Hill, 2003.
14. K. H. Rosen. *Discrete Mathematics and its Applications*. McGraw-Hill, 2006.
15. A. Silberschatz, H. F. Korth, and S. Sudarshan. *Database System Concepts*. McGraw-Hill Book Company, 2005.
16. M. Sipser. *Introduction to the Theory of Computation*. PWS Publishing Company, 1997.

17. R. E. Tarjan and M. Yannakakis. Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. *SIAM J. Comput.*, 13(3):566–579, 1984.
18. I. Wegener. *Complexity Theory.* Springer, 2005.

Part X

Appendix: Theory of Computation

A

Big-O Notation

We write $\mathbb{R}_0^+$ for the set of non-negative real numbers, and $\mathbb{R}^+$ for the set of positive real numbers. We typically measure the performance of an algorithm, that is, the number of basic operations it performs, as a function of its input length. In other words, the performance of an algorithm can be captured by a function $f : \mathbb{N} \to \mathbb{R}_0^+$ such that $f(n)$ is the maximum number of basic operations that the algorithm performs on inputs of length n. However, since f may heavily depend on the details of the definition of basic operations, we usually concentrate on the overall and asymptotic behaviour of the algorithm. This is achieved via the well-known notion of *big-O notation*.

The big-O notation is typically defined for single variable functions such as f above. However, in the database setting, where the input to key problems usually consists of several different components, we generally have to deal with multiple variable functions. For example, the performance of a query evaluation algorithm, where the input consists of two distinct components, the *database* and the *query*, can be captured by a function $f : \mathbb{N}^2 \to \mathbb{R}_0^+$ such that $f(n, m)$ is the maximum number of basic operations that the algorithm performs on databases of size n and queries of size m. The notion of big-O notation for multiple variable functions follows:

Definition 1.1: Big-O Notation

Let $f, g : \mathbb{N}^\ell \to \mathbb{R}_0^+$, where $\ell \geq 1$. We say that

$$f(x_1, \ldots, x_\ell) \text{ is in } O(g(x_1, \ldots, x_\ell))$$

if there exist $k \in \mathbb{R}^+$ and $n_0 \in \mathbb{N}$ such that, for every $(x_1, \ldots, x_\ell)$ with $x_i \geq n_0$ for some $i \in [\ell]$, $f(x_1, \ldots, x_\ell) \leq k \cdot g(x_1, \ldots, x_\ell)$.

Notice that when $\ell = 1$, i.e., f, g are single variables function, Definition 1.1 coincides with the standard big-O notation for single variable functions.

B

Turing Machines and Complexity Classes

Many results in this book will provide bounds on computational resources (time and space), or key database problems such as query evaluation. These are often formulated in terms of membership in, or completeness for, complexity classes. Those, in turn, are defined using the basic model of computation, that is, Turing Machines. We now briefly recall basic concepts related to Turing Machines and complexity classes. For more details, the reader can consult standard textbooks on computability theory and computational complexity.

Turing Machines

Turing Machines can work in two modes: either as *acceptors*, for deciding whether an input string belongs to a given language (in which case we speak of *decision problems*), or as computational devices that compute the value of a function applied to its input. When a Turing Machine works as an acceptor, it typically contains a read-write tape, a model of computation that is convenient for defining time complexity classes, or a read-only input tape and a read-write working tape, a model that is convenient for defining space complexity classes. When a Turing Machine works as a computational device, it typically contains a read-only tape where the input is placed, a read-write working tape, and a write-only tape where the output computed by the Turing Machine is placed.

Turing Machines as Acceptors

We start with the definition of deterministic Turing Machines.

Definition 2.1: Deterministic Turing Machine

A *(deterministic) Turing Machine* (TM) is defined as a tuple

$$M \;=\; (Q, \Sigma, \delta, s)\,,$$

where

- Q is a finite set of states, including the *accepting state* "yes", and the *rejecting state* "no",
- Σ is a finite set of input symbols, called the *alphabet* of M, including the symbols $\sqcup$ (the *blank symbol*) and $\triangleright$ (the *left marker*),
- $\delta : (Q - \{\text{"yes"}, \text{"no"}\}) \times \Sigma \to Q \times \Sigma \times \{\rightarrow, \leftarrow, -\}$ is the *transition function* of M, and
- $s \in Q$ is the *start state* of M.

Accepting and rejecting states are needed for decision problems: they determine whether the input belongs to the language or not. Notice that, according to δ, the accepting and rejecting states do not have outgoing transitions.

A *configuration* of a TM $M = (Q, \Sigma, \delta, s)$ is a tuple

$$c \;=\; (q, u, v)\,,$$

where $q \in Q$, and u, v are words in Σ^* with u being always non-empty. If M is in configuration c, then the tape has content uv and the head is reading the last symbol of u. We use left markers, which means that u always starts with $\triangleright$. Moreover, the transition function δ is restricted in such a way that $\triangleright$ occurs exactly once in uv, and always as the first symbol of u.

Assume now that M is in a configuration $c = (q, ua, v)$, where $q \in Q - \{\text{"yes"}, \text{"no"}\}$, $a \in \Sigma$ and $u, v \in \Sigma^*$, and assume that $\delta(q, a) = (q', b, \text{dir})$, where $\text{dir} \in \{\rightarrow, \leftarrow, -\}$. Then, in one step, M enters the configuration $c' = (q', u', v')$, where u', v' is obtained from ua, v by replacing a with b and moving the head one step in the direction dir. By moving the head in the direction "$-$" we mean that the head stays in its place. Furthermore, the head cannot move left of the $\triangleright$ symbol (the transition function δ is restricted in such a way that this cannot happen: if $\delta(q, \triangleright) = (q', a, \text{dir})$, then $a = \triangleright$ and $\text{dir} \neq \leftarrow$). For example, if $c = (q, \triangleright 01, 100)$ and $\delta(q, 1) = (q', 0, \leftarrow)$, then $c' = (q', \triangleright 0, 0100)$. In this case, we write $c \to_M c'$, and we also write $c \to_M^m c'$ if c' can be reached from c in m steps, and $c \to_M^* c'$ if $c \to_M^m c'$ for some $m \geq 0$ (we assume that $c \to_M^0 c$). Finally, if $v = \varepsilon$ and $\text{dir} = \rightarrow$, then we insert an additional $\sqcup$-symbol in our configuration, that is $u' = ub\sqcup$ and $v' = \varepsilon$.

A TM M receives an input word $w = a_1 \cdots a_n$, where $n \geq 0$ and $a_i \in \Sigma - \{\sqcup, \triangleright\}$ for each $i \in [n]$. The *start configuration of M on input w* is $sc(w) = (s, \triangleright, w)$. We call a configuration c *accepting* if its state is "yes", and *rejecting* if its state is "no". The TM M *accepts* (respectively, *rejects*) input w if $sc(w) \to_M^* c$ for some accepting (respectively, rejecting) configuration c.

Nondeterministic Turing Machines as Acceptors

We also use nondeterministic Turing Machines as acceptors, which are defined similarly to deterministic ones, but with the key difference that the a state-symbol pair has more than one outgoing transitions.

Definition 2.2: Nondeterministic Turing Machine

A *nondeterministic Turing Machine* (NTM) is defined as a tuple

$$M = (Q, \Sigma, \delta, s),$$

where

- Q is a finite set of states, including the *accepting state* "yes", and the *rejecting state* "no",
- Σ is a finite set of input symbols, called the *alphabet* of M, including the symbols $\sqcup$ (the *blank symbol*) and $\triangleright$ (the *left marker*),
- $\delta : (Q - \{\text{"yes"}, \text{"no"}\}) \times \Sigma \to \mathcal{P}(Q \times \Sigma \times \{\rightarrow, \leftarrow, -\})$ is the *transition function* of M, and
- $s \in Q$ is the *start state* of M.

Observe that for a given configuration $c = (q, ua, v)$, where $q \in Q - \{\text{"yes"}, \text{"no"}\}$, $a \in \Sigma$ and $u \in \Sigma^*$, several alternatives (q', b, dir) can belong to $\delta(q, a)$, each one of which generates a successor configuration c' as in the case of (deterministic) TMs. If c' is a possible successor configuration of c, then we write $c \to_M c'$. Moreover, we write $c \to_M^m c'$ if there exists a sequence of configurations $c_1, \ldots, c_{m-1}$ such that $c \to_M c_1$, $c_1 \to_M c_2$, $\ldots$, $c_{m-1} \to_M c'$. In this case, notice that it is possible that $c \to_M^m c'$ and $c \to_M^n c'$ with $m \neq n$. Moreover, we write $c \to_M^* c'$ if there exists $m \geq 0$ such that $c \to_M^m c'$ (again, we assume that $c \to_M^0 c$).

Given an input word w for a NTM M, the start configuration $sc(w)$ of M, and accepting and rejecting configurations of M, are defined as in the deterministic case. Moreover, M accepts input w if there exists an accepting configuration c such that $sc(w) \to_M^* c$, and M rejects w otherwise (i.e., M rejects w if there is no accepting configuration c such that $sc(w) \to_M^* c$).

2-*Tape Turing Machines as Acceptors*

We now define Turing Machines that, apart from a read-write working tape, they also have a read-only input tape.

Definition 2.3: 2-Tape Deterministic Turing Machine

A 2-*tape (deterministic) Turing Machine* (2-TM) is defined as a tuple

$$M = (Q, \Sigma, \delta, s),$$

where

- Q is a finite set of states, including the *accepting state* "yes", and the *rejecting state* "no",
- Σ is a finite set of input symbols, called the *alphabet* of M, including the symbols $\sqcup$ (the *blank symbol*) and $\triangleright$ (the *left marker*),
- $\delta : (Q - \{\text{"yes"}, \text{"no"}\}) \times \Sigma \times \Sigma \to Q \times \{\rightarrow, \leftarrow, -\} \times \Sigma \times \{\rightarrow, \leftarrow, -\}$ is the *transition function* of M, and
- $s \in Q$ is the *start state* of M.

A *configuration* of a 2-TM is a tuple

$$c = (q, u_1, v_1, u_2, v_2),$$

where $q \in Q$ and, for every $i \in \{1, 2\}$, we have that $u_i, v_i \in \Sigma^*$ and u_i is not empty. If M is in configuration c, then the input tape has content $u_1 v_1$ and the head of this tape is reading the last symbol of u_1, while the working tape has content $u_2 v_2$ and the head of this tape is reading the last symbol of u_2. We use left markers, which means that u_i always starts with $\triangleright$. Besides, the transition function δ is restricted in such a way that $\triangleright$ occurs exactly once in $u_i v_i$, and always as the first symbol of u_i.

Assume that M is in a configuration $c = (q, u_1 a_1, v_1, u_2 a_2, v_2)$, where $q \in Q - \{\text{"yes"}, \text{"no"}\}$, $a_1, a_2 \in \Sigma$ and $u_1, v_1, u_2, v_2 \in \Sigma^*$, and assume that $\delta(q, a_1, a_2) = (q', \text{dir}_1, b, \text{dir}_2)$, where dir_i is a direction, i.e., one of $\{\rightarrow, \leftarrow, -\}$. Then in one step M enters configuration $c' = (q', u_1', v_1', u_2', v_2')$, where u_1', v_1' is obtained from $u_1 a_1, v_1$ by moving the head one step in the direction dir_1, and u_2', v_2' is obtained from $u_2 a_2, v_2$ by replacing a_2 with b and moving the head one step in the direction dir_2. Recall that by moving the head in the direction "$-$" we mean that the head stays in its place. Furthermore, the head cannot move left of the $\triangleright$ symbol (again, the transition function δ is restricted in such a way that this cannot happen). For example, if $c = (q, \triangleright 01, 100, \triangleright, \varepsilon)$ and $\delta(q, 1, \triangleright) = (q', \leftarrow, \triangleright, \rightarrow)$, then $c' = (q', \triangleright 0, 1100, \triangleright, \varepsilon)$. In this case, we write $c \to_M c'$. We also write $c \to_M^m c'$ if c' can be reached from c in m steps, and $c \to_M^* c'$ if $c \to_M^m c'$ for some $m \geq 0$ (we assume that $c \to_M^0 c$).

A 2-TM M receives an input word $w = a_1 \cdots a_n$, where $n \geq 0$ and $a_i \in \Sigma - \{\sqcup, \triangleright\}$ for each $i \in [n]$. The *start configuration of M on input w* is $sc(w) = (s, \triangleright, w, \triangleright, \sqcup)$. We call a configuration c *accepting* if its state is "yes", and

rejecting if its state is "no". The TM M *accepts* (respectively, *rejects*) input w if $sc(w) \rightarrow_M^* c$ for some accepting (respectively, rejecting) configuration c.

2-*Tape Nondeterministic Turing Machines as Acceptors*

As for TMs, we also have the nondeterministic version of 2-TMs.

Definition 2.4: 2-Tape Nondeterministic Turing Machine

A 2-*Tape Nondeterministic Turing Machine* (2-NTM) is a tuple

$$M \;=\; (Q, \Sigma, \delta, s)\,,$$

where

- Q is a finite set of states, including the *accepting state* "yes", and the *rejecting state* "no",
- Σ is a finite set of input symbols, called the *alphabet* of M, including the symbols $\sqcup$ (the *blank symbol*) and $\triangleright$ (the *left marker*),
- $\delta:\ (Q - \{\text{"yes"}, \text{"no"}\}) \times \Sigma \times \Sigma \rightarrow \mathcal{P}(Q \times \{\rightarrow, \leftarrow, -\} \times \Sigma \times \{\rightarrow, \leftarrow, -\})$ is the *transition function* of M, and
- $s \in Q$ is the *start state* of M.

It is clear that, for a configuration $c = (q, u_1, a_1v_1, u_2, a_2v_2)$, where $q \in Q - \{\text{"yes"}, \text{"no"}\}$, $a_1, a_2 \in \Sigma$ and $v_1, v_2 \in \Sigma^*$, several alternatives $(q', \mathrm{dir}_1, b, \mathrm{dir}_2)$ can belong to $\delta(q, a_1, a_2)$, each one of which generates a successor configuration c' as in the case of 2-TMs. If c' is a possible successor configuration of c, then we write $c \rightarrow_M c'$. Moreover, we write $c \rightarrow_M^m c'$ if there exists a sequence of configurations $c_1, \ldots, c_{m-1}$ such that $c \rightarrow_M c_1$, $c_1 \rightarrow_M c_2$, $\ldots$, $c_{m-1} \rightarrow_M c'$. In this case, notice that it is possible that $c \rightarrow_M^m c'$ and $c \rightarrow_M^n c'$ with $m \neq n$. Moreover, we write $c \rightarrow_M^* c'$ if there exists $m \geq 0$ such that $c \rightarrow_M^m c'$ (again, we assume that $c \rightarrow_M^0 c$).

Given an input word w for a 2-NTM M, the start configuration $sc(w)$ of M, and accepting and rejecting configurations of M, are defined as in the deterministic case. Moreover, M accepts input w if there exists an accepting configuration c such that $sc(w) \rightarrow_M^* c$, and M rejects w otherwise (i.e., M rejects w if there is no accepting configuration c such that $sc(w) \rightarrow_M^* c$).

Turing Machines as Computational Devices

If a 2-TM acts not as a language acceptor but rather as a device for computing a function f, then a write-only output tape is added and the states "yes" and "no" are replaced with a *halting state* "halt"; once the computation enters the halting state, the output tape contains the value $f(w)$ for the input w.

Definition 2.5: Turing Machine with Output

A *Turing Machine with output* (TMO) is a tuple

$$M \;=\; (Q, \Sigma, \delta, s)\,,$$

where

- Q is a finite set of states, including the *halting state* "halt",
- Σ is a finite set of input symbols, called the *alphabet* of M, including the symbols $\sqcup$ (the *blank symbol*) and $\triangleright$ (the *left marker*),
- $\delta:\ (Q - \{\text{"halt"}\}) \times \Sigma \times \Sigma \to Q \times \{\rightarrow, \leftarrow, -\} \times \Sigma \times \{\rightarrow, \leftarrow, -\} \times \Sigma$ is the *transition function* of M, and
- $s \in Q$ is the *start state* of M.

If $\delta(q, a_1, a_2) = (q', \mathrm{dir}_1, b, \mathrm{dir}_2, c)$, then $q', \mathrm{dir}_1, b, \mathrm{dir}_2$ are used exactly as in the case of a 2-TM accepting a language. Moreover, if $c \neq \sqcup$, then c is written on the output tape and the head of this tape is moved one position to the right; otherwise, no changes are made on this tape. The start configuration of a TMO M on input w is $sc(w) = (s, \triangleright, w, \triangleright, \varepsilon, \triangleright, \varepsilon)$. The output of M on input w is the word u such that $sc(w) \to_M^* (\text{"halt"}, u_1, v_1, u_2, v_2, \triangleright u, \varepsilon)$.

Complexity Classes

We proceed to introduce some central complexity classes that are used in this book. Recall that $\mathbb{R}_0^+$ is the set of non-negative real numbers. Given a function $f : \mathbb{N} \to \mathbb{R}_0^+$, a TM (respectively, NTM) M is said to run in *time* $f(n)$ if, for every input w and configuration c, $sc(w) \to_M^m c$ implies $m \leq f(|w|)$.[1] We further say that M *decides* a language L if M accepts every word in L and rejects every word not in L. Notice that this implies that M's computation is finite on every input. We define the classes of decision problems

$$\textsc{Time}(f(n)) \;=\; \{L \mid \text{there exists a TM that decides } L \text{ and runs in time } f(n)\}$$

and

$$\textsc{NTime}(f(n)) \;=\; \{L \mid \text{there exists an NTM that decides } L \text{ and runs in time } f(n)\}.$$

We use $\textsc{Time}(O(f(n)))$ for the union of all $\textsc{Time}(g(n))$ where $g(n) \in O(f(n))$. Furthermore, we use the following time complexity classes in this book:

[1] The running time of a TMO is defined in the same way.

Definition 2.6: Time Complexity Classes

$\textsc{PTime} = \bigcup_{k\in\mathbb{N}} \textsc{Time}(n^k)$ $\quad$ $\textsc{NP} = \bigcup_{k\in\mathbb{N}} \textsc{NTime}(n^k)$
$\textsc{ExpTime} = \bigcup_{k\in\mathbb{N}} \textsc{Time}(2^{n^k})$ $\quad$ $\textsc{NExpTime} = \bigcup_{k\in\mathbb{N}} \textsc{NTime}(2^{n^k})$
$2\textsc{ExpTime} = \bigcup_{k\in\mathbb{N}} \textsc{Time}(2^{2^{n^k}})$

Given a function $f : \mathbb{N} \to \mathbb{R}_0^+$, a 2-TM (respectively, 2-NTM) M is said to run in *space* $f(n)$ if, for every input w and configuration $c = (q, u_1, v_1, u_2, v_2)$, $sc(w) \to_M^* c$ implies $|u_2v_2| \leq f(|w|)$.[2] We say that M *decides* a language L if M accepts every word in L and rejects every word not in L. We define the classes of decision problems

$$\textsc{Space}(f(n)) = \{L \mid \text{there exists a 2-TM that decides } L \text{ and runs in space } f(n)\}$$

and

$$\textsc{NSpace}(f(n)) = \{L \mid \text{there exists a 2-NTM that decides } L \text{ and runs in space } f(n)\}.$$

We write $\textsc{Space}(O(f(n)))$ for the union of all $\textsc{Space}(g(n))$, where $g(n) \in O(f(n))$. We further use the following space complexity classes in this book:

Definition 2.7: Space Complexity Classes

$\textsc{DLogSpace} = \textsc{Space}(\log n)$ $\quad$ $\textsc{NLogSpace} = \textsc{NSpace}(\log n)$
$\textsc{PSpace} = \bigcup_{k\in\mathbb{N}} \textsc{Space}(n^k)$ $\quad$ $\textsc{NPSpace} = \bigcup_{k\in\mathbb{N}} \textsc{NSpace}(n^k)$
$\textsc{ExpSpace} = \bigcup_{k\in\mathbb{N}} \textsc{Space}(2^{n^k})$ $\quad$ $\textsc{NExpSpace} = \bigcup_{k\in\mathbb{N}} \textsc{NSpace}(2^{n^k})$

At this point, let us stress that we can always assume that the computation of a space-bounded 2-TM M is finite on every input word. Intuitively, since the space that M uses is bounded, the number of different configurations in which M can be is also bounded. Therefore, by maintaining a counter that "counts" the steps of M, we can guarantee that M will never fall in an unnecessarily long computation, which in turn allows us to assume that the computation of M is finite. Further details on this assumption can be found in any standard textbook on computational complexity.

For a complexity class $\mathcal{C}$, the class $\textsc{co}\mathcal{C}$ is defined as the set of complements of the problems in $\mathcal{C}$, that is, $\textsc{co}\mathcal{C} = \{\Sigma^* - L \mid L \in \mathcal{C}\}$. It is known that

[2] The running space of a TMO is defined without considering the output tape. More precisely, for every input w and configuration $c = (q, u_1, v_1, u_2, v_2, u_3, v_3)$, $sc(w) \to_M^* c$ implies $|u_2v_2| \leq f(|w|)$.

$$\begin{aligned}\textsc{DLogSpace} \subseteq \textsc{NLogSpace} \subseteq \textsc{PTime} \subseteq \textsc{NP} \subseteq \textsc{PSpace} = \textsc{NPSpace} \\ \subseteq \textsc{ExpTime} \subseteq \textsc{NExpTime} \subseteq \textsc{ExpSpace} = \textsc{NExpSpace} \subseteq \textsc{2ExpTime}\end{aligned}$$

$$\textsc{PTime} \subsetneq \textsc{ExpTime} \subsetneq \textsc{2ExpTime}$$

$$\textsc{NP} \subsetneq \textsc{NExpTime}$$

and that

$$\textsc{NLogSpace} = \textsc{coNLogSpace} \subsetneq \textsc{PSpace} \subsetneq \textsc{ExpSpace}$$

However, it is still not known whether PTime (and in fact DLogSpace) is properly contained in NP, whether PTime is properly contained in PSpace, and whether NP equals coNP.

Key concepts related to complexity classes are *reductions* between problems, and *hardness* and *completeness* of problems. For precise definitions the reader can consult any complexity theory textbook. A reduction between languages L and L' over an alphabet Σ is a function $f : \Sigma^* \to \Sigma^*$ such that $w \in L$ if and only if $f(w) \in L'$, for every $w \in \Sigma^*$. Let $\mathcal{C}$ be one of the complexity classes introduced above such that $\textsc{NP} \subseteq \mathcal{C}$ or $\textsc{coNP} \subseteq \mathcal{C}$. A *problem*, i.e., a language L, is *hard* for $\mathcal{C}$, or $\mathcal{C}$-hard, if every problem $L' \in \mathcal{C}$ is reducible to L via a reduction that is computable in polynomial time. If L is also in $\mathcal{C}$, then it is *complete* for $\mathcal{C}$, or $\mathcal{C}$-complete. For the complexity classes NLogSpace and PTime, the notions of hardness and completeness are defined in the same way, but with the crucial difference that we rely on reductions that are computable in deterministic logarithmic space. This is because a reduction is meaningful only within a class that is computationally stronger than the reduction.[3]

We say that a decision problem is *tractable* if it is in PTime. As such, problems that are hard for ExpTime are *provably intractable*. We call problems that are hard for NP or coNP *presumably intractable* (if we cannot make a stronger case and prove that they are not in PTime).

The most fundamental problem that is presumably intractable is the satisfiability problem of Boolean formulae. A *Boolean formula* is defined as follows:

- a variable $x \in \mathsf{Var}$ is a Boolean formula, and
- if φ_1 and φ_1 are Boolean formulae, then $(\varphi_1 \wedge \varphi_2)$, $(\varphi_2 \vee \varphi_2)$, and $(\neg\varphi_1)$ are Boolean formulae.

To define the semantics of such Boolean formulae, we need the notion of truth assignment. A *truth assignment* for a set of variables V is a function $f : V \to \{\mathtt{true}, \mathtt{false}\}$. Consider a Boolean formula φ, and a truth assignment f for the set of variables in φ. We define when f *satisfies* φ, written $f \models \varphi$:

[3] We could also define hardness for DLogSpace by using reductions that can be computed via a computation even more restrictive than deterministic logarithmic space, but this is not needed for the purposes of this book.

- If φ is a variable x, then $f \models \varphi$ if $f(x) = \texttt{true}$.
- If $\varphi = (\varphi_1 \wedge \varphi_2)$, then $f \models \varphi$ if $f \models \varphi_1$ and $f \models \varphi_2$.
- If $\varphi = (\varphi_1 \vee \varphi_2)$, then $f \models \varphi$ if $f \models \varphi_1$ or $f \models \varphi_2$.
- If $\varphi = (\neg\psi)$, then $f \models \varphi$ if $f \models \psi$ does not hold.

We say that φ is *satisfiable* if there exists a truth assignment f for the set of variables in φ such that $f \models \varphi$. The *Boolean satisfiability* problem or SAT, which is known to be an NP-complete problem, is defined as follows.

Problem: SAT

Input: A Boolean formula φ
Output: `true` if φ is satisfiable, and `false` otherwise

It is actually the first problem that was proven to be NP-complete, a result known as Cook-Levin Theorem that goes back to the 1970s.

A generalization of SAT is the satisfiability problem of quantified Boolean formulae. For a Boolean formula φ and a tuple of variables $\bar{x}$, we denote by $\varphi\langle\bar{x}\rangle$ the fact that φ uses precisely the variables in $\bar{x}$. A *quantified Boolean formula* ψ is an expression of the form

$$Q_1\bar{x}_1 Q_2\bar{x}_2 \cdots Q_n\bar{x}_n \, \varphi\langle\bar{x}_1, \ldots, \bar{x}_n\rangle \,,$$

where, for each $i \in [n]$, Q_i is either $\exists$ or $\forall$, and, for each $i \in [n-1]$, $Q_i = \exists$ implies $Q_{i+1} = \forall$ and $Q_i = \forall$ implies $Q_{i+1} = \exists$. Assuming that $Q_1 = \exists$, we say that ψ is *satisfiable* if there exists a truth assignment for $\bar{x}_1$ such that for every truth assignment for $\bar{x}_2$ there exists a truth assignment for $\bar{x}_3$, and so on up to $\bar{x}_n$, such that the overall truth assignment satisfies ψ. Analogously, we can define when ψ is satisfiable in the case $Q_1 = \forall$. The *quantified satisfiability* problem or QSAT, also known under the name *quantified Boolean formula* or QBF, which is the canonical PSPACE-complete problem, is defined as follows:

Problem: QSAT

Input: A quantified Boolean formula ψ
Output: `true` if ψ is satisfiable, and `false` otherwise

Notice that SAT is the special case of QSAT where ψ is of the form $\exists\bar{x}\,\varphi\langle\bar{x}\rangle$. Two special cases of QSAT will be particularly important for this book, namely the ones with exactly one quantifier alternation:

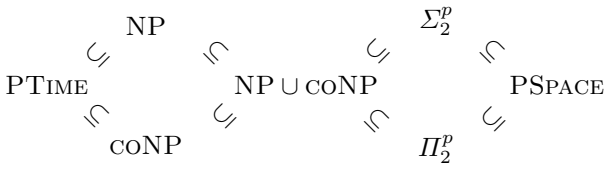
Problem: $\exists\forall$QSAT

Input: A quantified Boolean formula $\psi = \exists \bar{x}_1 \forall \bar{x}_2 \, \varphi\langle \bar{x}_1, \bar{x}_2 \rangle$
Output: `true` if ψ is satisfiable, and `false` otherwise

Problem: $\forall\exists$QSAT

Input: A quantified Boolean formula $\psi = \forall \bar{x}_1 \exists \bar{x}_2 \, \varphi\langle \bar{x}_1, \bar{x}_2 \rangle$
Output: `true` if ψ is satisfiable, and `false` otherwise

We define Σ_2^p as the class of decision problems reducible to $\exists\forall$QSAT in polynomial time. Similarly, Π_2^p is the class of decision problems reducible to $\forall\exists$QSAT in polynomial time. Recall that QSAT is PSPACE-complete. We know that



We finally remark that the smallest complexity class we consider here is DLOGSPACE. In database theory, and especially in its logical counterpart, that is, finite model theory, it is very common to consider parallel complexity classes, of which the smallest one is AC^0. These are circuit complexity classes, and the machinery needed to define them is not TMs but rather circuits, parameterized by their fan-in (the number of inputs to their gates), their size, and their depth. Due to the notational overhead this incurs, we shall not be using circuit-based classes in this book. The interested reader can consult books on finite model theory and descriptive complexity to understand the differences between DLOGSPACE and classes such as AC^0.

C

Input Encodings

To reason about the computational complexity of problems, we need to represent their inputs (such as databases, queries, and constraints) as inputs to Turing Machines, that is, as words over some finite alphabet.

Encoding of Queries and Constraints

Queries and constraints will most commonly be coming from a query language and a class of constraints, respectively, defined by a formal syntax. We thus associate a query and a set of constraints with its parse tree, which, of course, can be easily encoded as a word over a finite alphabet.

Encoding of Databases

For databases, the idea is that each value in the active domain can be encoded as a number in binary, and then use further separator symbols that allows us to faithfully encode the facts occurring in the database.

We assume a strict total order $<_{\mathsf{Rel}}$ on the elements of Rel, and a strict total order $<_{\mathsf{Const}}$ on the elements of Const. Consider a schema $\mathbf{S} = \{R_1, \ldots, R_n\}$ with $R_i <_{\mathsf{Rel}} R_{i+1}$ for each $i \in [n-1]$, and a database D of $\mathbf{S}$ with $\mathrm{Dom}(D) = \{a_1, \ldots, a_k\}$ and $a_i <_{\mathsf{Const}} a_{i+1}$ for each $i \in [k-1]$. We proceed to explain how D is encoded as a word over the alphabet

$$\Sigma \;=\; \{0, 1, \triangle, \#, \$, \square\}.$$

We first explain how constants, tuples, and relations are encoded:

- The constant $a_i \in \mathrm{Dom}(D)$, for $i \in [k]$, is encoded as the number i in binary, and we write $enc(a_i)$ for the obtained word over $\{0, 1\}$.
- A tuple $\bar{t} = (a_1, \ldots, a_\ell)$ over $\mathrm{Dom}(D)$, for $\ell \geq 0$, is encoded as the word

$$enc(\bar{t}) \;=\; \begin{cases} \square enc(a_1)\square \cdots \square enc(a_\ell)\square & \text{if } \ell > 0, \\ \square\square & \text{if } \ell = 0. \end{cases}$$

- A relation $R_i^D = \{\bar{t}_1, \ldots, \bar{t}_m\}$, for $i \in [n]$ and $m \geq 0$, is encoded as

$$enc(R_i^D) \;=\; \begin{cases} \$enc(\bar{t}_1)\$ \cdots \$enc(\bar{t}_m)\$ & \text{if } m > 0, \\ \$\$ & \text{if } m = 0. \end{cases}$$

We can now encode the database D as a word over Σ as follows:

$$enc(D) \;=\; \triangle enc(a_1) \triangle \cdots \triangle enc(a_k) \triangle \# enc(R_1^D) \# \cdots \# enc(R_n^D) \# \,.$$

The key property of the above encoding is that, for a database D of a schema $\mathbf{S}$, and a tuple $\bar{t}$, given as their encodings $enc(D)$ and $enc(\bar{t})$, respectively, we can check via a deterministic computation, which uses logarithmic space in the size fo $enc(D)$, whether $\bar{t} \in R^D$ for some $R \in \mathbf{S}$. In what follows, we write $enc(i)$ for the binary representation of an integer $i > 0$.

Lemma C.1. *Let $\boldsymbol{S}$ be the schema $\{R_1, \ldots, R_n\}$ with $R_1 <_{\mathsf{Rel}} \cdots <_{\mathsf{Rel}} R_n$. Consider a database D of $\boldsymbol{S}$, a tuple $\bar{t}$, and an integer $i \in [n]$, and let w be the word $\triangleright enc(D) \flat enc(\bar{t}) \flat enc(i)$ over $\Sigma \cup \{\triangleright, \sqcup, \flat\}$. There exists a 2-TM M with alphabet Σ such that the following hold:*

1. *M accepts w if and only if $\bar{t} \in R_i^D$, and*
2. *M runs in space $O(ar(R_i) \cdot \log |enc(D)|)$ if $ar(R_i) > 0$, and $O(\log |enc(D)|)$ if $ar(R_i) = 0$.*

Proof. We first give a high-level description of the 2-TM M; for brevity, we write *it* for the symbol read by the head of the input tape:

1. Let $ctr = 0$ – this is a counter maintained on the work tape in binary.
2. While $ctr \neq i$ do the following:
   a) If $it = \#$, then $ctr := ctr + 1$.
   b) Move the head of the input tape to the right so that it reads the first \$ symbol of $enc(R_i^D)$.
3. Move the head of the input tape to the right so that it reads the first $\square$ symbol of $enc(\bar{u})$, where $\bar{u}$ is the first tuple of R_i^D (i.e., $enc(R_i^D) = \$enc(\bar{u})\$ \cdots \$$), or the second \$ symbol of $enc(R_i^D)$ in case R_i^D is empty (which means that $enc(R_i^D) = \$\$$).
4. Erase the content of the work tape by replacing every symbol different than $\sqcup$ with $\sqcup$ (since ctr is not needed further), and move its head after the left marker $\triangleright$.
5. Repeat the following steps until $it = \#$ (which means that the relation R_i^D has been fully explored):
   a) While $it \neq \$$ do the following:

(i) Copy *it* to the work tape.

(ii) Move the head of both tapes to the right.

b) Assuming that $\triangleright u \sqcup$ is the content of the work tape, if $u = enc(\bar{t})$, then halt and accept; otherwise:

(i) Move the head of the input tape to the right so that it reads the first $\square$ symbol of the encoding of the next tuple of R_i^D, or the symbol # if the last tuple of R_i^D has just been explored. In other words, the head of the input tape reads the symbol to the right of the last \$ symbol read during the while loop of step (a).

(ii) Erase the content of the work tape by replacing every symbol different than $\sqcup$ with $\sqcup$ (since the copied tuple is not needed further), and move its head after the left marker $\triangleright$.

6. Halt and reject.

It is easy to verify that M accepts w if and only if $enc(R_i^D)$ is of the form $\$ \cdots \$ enc(\bar{t}) \$ \cdots \$$, or, equivalently, $\bar{t} \in R_i^D$. It remains to argue that M runs in the claimed space. At each step of the computation of M, the work tape holds either *ctr*, or the word $enc(\bar{t})$ for some $\bar{t} \in R_i^D$. The value of *ctr* (represented in binary) can be maintained using $O(|enc(i)|)$ bits. The encoding of a tuple of R_i^D takes space $O(\mathrm{ar}(R_i) \cdot \log |\mathrm{Dom}(D)|)$. Therefore, the space used is

$$O\left(\log |enc(i)| \ + \ \mathrm{ar}(R_i) \cdot \log |\mathrm{Dom}(D)|\right).$$

Since $|enc(i)| \leq |enc(D)|$ and $|\mathrm{Dom}(D)| \leq |enc(D)|$, we can conclude that the above 2-TM on input w runs in space $O(\mathrm{ar}(R_i) \cdot \log |enc(D)|)$ if $\mathrm{ar}(R_i) > 0$, and $O(\log |enc(D)|)$ if $\mathrm{ar}(R_i) = 0$, and the claim follows. $\square$

Note that the encoding described above is not the only way of encoding a database as a word over a finite alphabet. We could employ any other encoding as long as it enjoys the property established in Lemma C.1, without affecting the complexity results presented in this book.