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

Database Theory

Querying Data

(Preliminary Version)

August 19, 2022

Santiago Paris
Bayreuth Edinburgh

Contents

1 Introduction

This is a very preliminary partial release of the upcoming book "Database Theory", which will be about the foundational and mathematical principles of databases in their various forms. The early parts focus on an overview of the relational model, and on processing some of the most commonly occurring relational queries. Later parts 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 sometimes cut corners intentionally. 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 first-order logic on trees, XPath, and tree pattern queries, and their evaluation- and static analysis problems.

(IX) Expressive Languages for Tree-Structured Data

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

(X) 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.

Proofreaders

The present version has benefited from valuable comments of (in alphabetical order): Antoine Amarilli, Johannes Doleschal, Matthias Niewerth, Thomas Schwentick, Jef Wijsen

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 $[1, i]$. We write $\mathbb{Q}$ for the set of rational numbers and $\mathbb{Q}_{\geq 0}$ for the set of nonnegative rational numbers.

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

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

A *k-ary mathematical relation* is a set of k-ary tuples. Let T be a binary mathematical relation and let $S = \{a \mid \text{there exists } b \text{ such that } (a, b) \in T \text{ or } (b, a) \in T\}$. We say that

- T is *transitive* if, for all $a, b, c \in S$, we have that $(a, b) \in T$ and $(b, c) \in T$ implies that $(a, c) \in T$;
- T is *reflexive* if $(a, a) \in T$ for every $a \in S$;
- T is *antisymmetric* if, for all $a, b \in S$, we have that $(a, b) \in T$ and $(b, a) \in T$ implies that $a = b$;
- T is *total*, if, for all $a, b \in S$, we have that $(a, b) \in T$ or $(b, a) \in T$;

- T is a *partial order* if it is transitive, reflexive, and antisymmetric; and
- T is a *total order* if it is a partial order and total.

We also define the following standard operations on T.

- The *transitive closure* of T, denoted by T^+, is defined as $\{(a, b) \mid (a, b) \in T$ or there exist $a_1, \dots, a_n$ such that $n \geq 1$, $(a, a_1) \in T$, $(a_n, b) \in T$ and $(a_i, a_{i+1}) \in T$ for every $i \in [n-1]\}$.
- The *reflexive transitive closure* of T, denoted by T^*, is defined as $T^+ \cup \{(a, a) \mid a \in S\}$.

Finally, we say that T is a *successor relation* if

- for every $a \in S$, there exists at most one $b \in S$ such that $(a, b) \in T$, and
- T^* is a total order.

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 $\mathrm{Dom}(f)$ for the set of elements from S on which f is defined.

We say that a function $f : S \to T$ 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 $\bar{a} = (a_1, \dots, a_k) \in S^k$, then $f(\bar{a}) = (f(a_1), \dots, f(a_k))$. 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 pair (V, E), where V is a finite set of *nodes*, and E is a finite set of *edges*. We distinguish between *directed* and *undirected* graphs, which only differ in the way their set E of edges is defined:

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

In simple words, in directed graphs edges are pairs of nodes, and the order in which the nodes are given indicates the direction of the edge, that is, (u, v) is an edge from u to v, whereas (v, u) is an edge from v to u. In undirected graphs, edges are simply subsets of V of cardinality 2, and the order in which the nodes are given does not matter, i.e., $\{u, v\}$ and $\{v, u\}$ is the same (undirected) edge between the nodes u and v.

A graph $G_1 = (V_1, E_1)$ is a *subgraph* of a graph $G_2 = (V_2, E_2)$ if $V_1 \subseteq V_2$ and $E_1 \subseteq E_2$. Moreover, if both G_1 and G_2 are directed graphs, then G_1 is the *subgraph of* G_2 *induced by* V_1 if $E_1 = E_2 \cap (V_1 \times V_1)$, and if both G_1 and G_2 are undirected graphs, then G_1 is the *subgraph of* G_2 *induced by* V_1 if $E_1 = E_2 \cap \{S \subseteq V_1 \mid |S| = 1 \text{ or } |S| = 2\}$.

An *undirected path* in a (directed or undirected) graph $G = (V, E)$ is a non-empty sequence of nodes $\pi = u_0 u_1 \cdots u_k$ for $k \geq 0$, where

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

We say that π is *path from* u_0 *to* u_k and has *length* k. (The path of length zero is from u_0 to u_0.) A graph is *connected* if, for every pair of nodes $u, v \in V$, there is an undirected path from u to v.

A *path* in a directed graph $G = (V, E)$ is a non-empty sequence of nodes $\pi = u_0 u_1 \cdots u_k$ for $k \geq 0$, where $(u_{i-1}, u_i) \in E$ for every $i \in [k]$. Similar to undirected paths, we call π is a *path from* u_0 *to* u_k and its *length* is k.

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

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

The notions of subtree and induced subtree are naturally derived from the notions of subgraph and induced subgraph. Moreover, if v is the parent of u in a tree T, then u is called a child of v, and if v does not have any children, then v is called a leaf of T. Finally, if there exists a path from v to u in T, then v is said to be an *ancestor* of u, and u is said to be a *descendant* of v.

It is common in Computer Science (unlike other fields such as Biology) to graphically depict trees with their root on top and the edges directed downwards. Therefore, even if we deal with directed trees, we may omit the direction of the edges in figures, which is typically specified via an arrow.

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 $t = (A_1 : a_1, \dots, A_k : a_k)$, meaning that $t(A_i) = a_i$ for every $i \in [k]$. Notice that, according to this notation, $(A_1 : a_1, A_2 : a_2)$ and $(A_2 : a_2, A_1 : a_1)$ 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 $\mathbf{S}$* is the set $\mathbf{S}(R)$. The *arity of R under $\mathbf{S}$*, 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 $R[A_1, \dots, A_k]$ instead of $\mathbf{S}(R) = \{A_1, \dots, A_k\}$ 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 $(a_1, \dots, a_k)$, $(b_1, \dots, b_k)$, 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 $\mathrm{Dom}(D)$. Let us stress that this simplification leads to a clash of terminology. Indeed, given a function $f : S \to T$, the set S is typically called the domain of f. Therefore, since D is formally defined as a function, one may think that the domain of D is the domain of D seen as a function, which is a finite set of relation names. We will *never* use the term domain, and the notation $\mathrm{Dom}(D)$, to refer to the domain of the function D.

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*. By abuse of terminology, we will also refer to a mathematical relation as a *relation*, but it will always be clear from the context whether we mean a relation instance or a mathematical relation.

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 treat them as sets of relation names. Thus, we will usually treat a schema $\mathbf{S}$ as the finite set $\mathrm{Dom}(\mathbf{S})$. This means that whenever we write $\mathbf{S} = \{R_1, \dots, R_n\}$, we actually mean that $\mathrm{Dom}(\mathbf{S}) = \{R_1, \dots, R_n\}$. In the unnamed case, we may also write

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

for the fact that $\mathrm{Dom}(\mathbf{S}) = \{R_1, \dots, R_n\}$ 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 can be seen as sets, in particular, as sets of facts. In the unnamed perspective, 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 $R((a_1, \dots, a_k))$ to $R(a_1, \dots, a_k)$. We will usually treat 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 $\{(R,A) \mid R \in \mathrm{Dom}(\mathbf{S}) \text{ and } A \in \mathbf{S}(R)\}$. 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 $D' : \mathrm{Dom}(\mathbf{S}') \to \mathsf{uRI}$ of $\mathbf{S}'$ is defined as follows: for every $R \in \mathrm{Dom}(\mathbf{S}')$,

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

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 $\mathbf{S}' : \mathsf{Rel} \to \mathcal{P}_{\text{fin}}(\mathsf{Att})$ as follows: $\text{Dom}(\mathbf{S}') = \text{Dom}(\mathbf{S})$, and $\mathbf{S}'(R) = \{\#_1, \dots, \#_{\text{ar}_\mathbf{S}(R)}\}$ for every $R \in \text{Dom}(\mathbf{S})$. Moreover, for every database D of $\mathbf{S}$, a semantically equivalent database $D' : \text{Dom}(\mathbf{S}') \to \mathsf{nRI}$ of $\mathbf{S}'$ is defined as follows: for every $R \in \text{Dom}(\mathbf{S}')$,

$$D'(R) \;=\; \{(\#_1 \colon a_1, \dots, \#_k \colon a_k) \mid (a_1, \dots, a_k) \in D(R)\}.$$ [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 $R[A, B, \dots]$ 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

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

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 $\mathcal{L}_2 \subsetneq \mathcal{L}_1$.

[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.

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

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

disjoint from Const, Rel, and Att. If φ is a logical formula and $\bar{x} = (x_1, \dots, x_k) \in \mathsf{Var}^k$ 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 $\bar{a} = (a_1, \dots, a_k) \in \mathsf{Const}^k$ 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 $\texttt{true}$, and $\{\}$ as $\texttt{false}$. 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 $q = \varphi()$.

Next, we introduce relational atoms, which will be useful throughout the book and, in particular, for defining the syntax of query languages that are based on logic. When R is a k-ary relation name 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 $R(u_1, \dots, u_k)$ instead of $R((u_1, \dots, u_k))$. 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.

Computational Complexity Analysis

We will use two different cost models for analyzing the computational complexity of algorithmic problems such as the ones introduced above.

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
- say that the problem "is solvable in" time 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 define the size $\|o\|$ of syntactic objects o that we will consider for complexity analysis. In particular:

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

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

Hence, if D is a nonempty database instance of schema $\{R_1, \dots, R_n\}$, then

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

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

The size $\|q\|$ of a query q will depend on the formalisms that we will use throughout the book for representing q, such as first-order logic or Datalog. This means that the size of a query specified using first-order logic will be defined differently than the size of a query specified in Datalog, because the sizes of the associated first-order formula or Datalog program will be different. Throughout the book, we will therefore define the size of queries at the point where we define the class of queries in question.

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" [18], 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, [16, 24, 27]. 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 [26]. For additional information about computability theory, we provide a primer in Appendix B. Furthermore, we refer the reader to [19, 21, 28]; standard texts on complexity theory are [2, 23, 32]. For the foundations of finite model theory and descriptive complexity, the reader is referred to [17, 20, 22].

Table of Notation

Hard table:

Symbol(s)	Meaning
Rel	set of relation names
Att	set of attribute names
Const	set of values that can appear in a database
Var	set of variables

Soft table (we may violate this convention, but try to do it only rarely):

Symbol(s)	Usual meaning
D	database
$R, S, \ldots$	relation names
S	sometimes a set
$A, B, \ldots$	attribute names
$U, V, \ldots$	sets of attribute names
$\mathbf{S}$	database schema
$x, y, z, \ldots$	variables
$a, b, c, \ldots$	constants
$u, v, \ldots$	variables or constants
A_q	set of atoms of a CQ q

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 occurrences 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 $(\varphi \rightarrow \psi)$ for $((\neg\varphi) \vee \psi)$ and $(\varphi \leftrightarrow \psi)$ for $((\varphi \rightarrow \psi) \wedge (\psi \rightarrow \varphi))$. 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:

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

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

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

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

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

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 $\mathrm{FV}(\varphi)$ to $\mathrm{Dom}(D) \cup \mathrm{Dom}(\varphi) \subseteq \mathsf{Const}$, where $\mathrm{Dom}(\varphi)$ is the set of constants mentioned in φ. For example, for the formula $R(x,y,a)$, η is the function $\{x,y\} \to \mathrm{Dom}(D) \cup \{a\}$. 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 $R(u_1,\dots,u_k)$, then $(D,\eta) \models \varphi$ if $R(\eta(u_1),\dots,\eta(u_k)) \in D$.
- 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 $a \in \mathrm{Dom}(D) \cup \mathrm{Dom}(\varphi)$.
- If $\varphi = \forall x\, \psi$, then $(D,\eta) \models \varphi$ if $(D,\eta[x/a]) \models \psi$ for *each* constant $a \in \mathrm{Dom}(D) \cup \mathrm{Dom}(\varphi)$.

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.

For a first-order query $q = \varphi(\bar{x})$, we define its *size* $\|q\|$ 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

$$q_2(D) \;=\; \{(‘2’, ‘\text{Billie}’)\}.$$

- 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 $q(D) =$ `true` if $D \models \varphi$, and $q(D) =$ `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 conditions over sets of integers, which are 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 an 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 $\bar{a} \models \theta$:

$\bar{a} \models i \doteq j$	if $a_i = a_j$	$\bar{a} \models i \doteq a$	if $a_i = a$
$\bar{a} \models i \neq j$	if $a_i \neq a_j$	$\bar{a} \models i \neq a$	if $a_i \neq a$
$\bar{a} \models \theta \wedge \theta'$	if $\bar{a} \models \theta$ and $\bar{a} \models \theta'$	$\bar{a} \models \theta \vee \theta'$	if $\bar{a} \models \theta$ or $\bar{a} \models \theta'$
$\bar{a} \models \neg\theta$	if $\bar{a} \models \theta$ does not hold		

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 $A \doteq B$, $A \doteq a$, $A \neq B$, and $A \neq a$, 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.

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

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

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

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 $t = (A_1 \colon a_1, \dots, A_k \colon a_k)$ in $D(R)$, for a relation name R of sort $\{A_1, \dots, A_k\}$ (with $(R, A_1) \lessdot \cdots \lessdot (R, A_k)$), into a tuple $t' = (a_1, \dots, a_k)$ 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 $\{A_1, \dots, A_k\}$ 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 $\{A_1, \dots, A_k\}$, 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. $\square$

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. Of course SQL is a language with a multitude of features, but its very core captures relational algebra queries. For now, we concentrate on that core part of the language and demonstrate its correspondence with relational algebra. The set of queries we consider are of the form

```
Q1, Q2 :=   SELECT [DISTINCT] <select_list>
            FROM <from_list>
            WHERE <condition>
        |   Q1 UNION Q2
        |   Q1 EXCEPT Q2
```

The `from_list` provides the list of relation names and subqueries used in the query, and also their *aliases*. For example, we can put `R AS R1` on the list, 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 can also put Q `AS Name` on the list, in which case the subquery Q

is evaluated and the result of it, which is a relation, is given the name `Name`. We shall do both in the examples very soon.

The `select_list` containts constants or attributes of relation names mentioned in `from_list`. For example, if we had `R AS R1` in `from_list` and `R` has an attribute `A`, we can have a reference to `R1.A` in `select_list`. Likewise, if `from_list` contained Q `AS Name` and the output of Q contained attribute `B`, we can refer to `Name.B`. Constants can be output as well, e.g., for example, `5 AS C` will output the constant 5 as value of attribute C.

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 44. 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 = Pr2.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 $\neg(u_1 = u_2)$.

This query could alternatively be written as

```
SELECT T.id
FROM (SELECT P.pid AS id,
             Pr1.prname AS prof1,
             Pr2.prname AS prof2
      FROM Person AS P, Profession AS Pr1, Profession AS Pr2
      WHERE P.pid = Pr1.pid AND P.pid = Pr2.pid) AS T
WHERE NOT (T.prof1 = T.prof2)
```

that has a subquery in FROM.

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.

Core SQL to Relational Algebra

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  Q₁ AS S₁, . . . , Qₘ AS Sₘ
WHERE  condition
```

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

$$\boldsymbol{\rho}_{\text{out}}\left(\pi_{\{\alpha_1,\dots,\alpha_n\}}\left(\sigma_{\text{condition}}\left(\boldsymbol{\rho}_1(Q'_1) \bowtie \cdots \bowtie \boldsymbol{\rho}_m(Q'_m)\right)\right)\right) ,$$

where each Q'_i is the translation of Q_i into relational algebra. When Q_i is a relation R in the database, then $Q'_i = R$.

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 $-$.

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

$$\texttt{SELECT R.A}_1\ \texttt{AS A}_1,\ \ldots,\ \texttt{R.A}_n\ \texttt{AS A}_n\ \texttt{FROM R AS R}$$

Of course in real SQL one can omit `AS R` and also use shorthand `*` for listing all attributes, but here we keep the complete notation for the inductive construction.

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

$$\texttt{SELECT}\ a\ \texttt{AS A}$$

For the induction, we will assume that we can write all queries Q_e with a `SELECT` statement of the form

$$\texttt{SELECT}\ X_1\ \texttt{AS A}_1,\ \ldots,\ X_n\ \texttt{AS A}_n$$

where

- each X_i is either a constant a or of the form `Q.A` or `R.A` for a query or a relation in `FROM`, and
- all attribute names $\texttt{A}_i$ for $i \in [n]$ are different

and a `FROM` statement that, if present, is of the form

$$\texttt{FROM Q}_1\ \texttt{AS T}_1,\ \ldots,\ \texttt{Q}_m\ \texttt{AS T}_m$$

where each $\texttt{Q}_i$ is either a subquery or a relation name, and all the names $\texttt{T}_j$ for $j \in [m]$ are different, and also different from names of relations in the database.

Selection and Projection. Assume that e is translated into

SELECT X_1 AS $\mathtt{A}_1, \ldots, X_n$ AS $\mathtt{A}_n$
FROM $\mathtt{Q}_1$ AS $\mathtt{T}_1, \ldots, \mathtt{Q}_m$ AS $\mathtt{T}_m$
WHERE condition

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

  SELECT X_1 AS $\mathtt{A}_1, \ldots, X_n$ AS $\mathtt{A}_n$
  FROM $\mathtt{Q}_1$ AS $\mathtt{T}_1, \ldots, \mathtt{Q}_m$ AS $\mathtt{T}_m$
  WHERE condition AND C_θ

  where C_θ expresses the condition θ in SQL syntax, but uses the names X_i instead of $\mathtt{A}_i$ due to the ordering in which SQL applies aliases. For instance, if θ is $(A_1 \doteq A_2) \wedge \neg(A_3 \doteq 1)$ then C_θ is (X_1 = X_2) AND NOT (X_3 = 1).
  To illustrate this with an example, consider the relation $R[A, B]$ and the query Q_e given as

```
SELECT R.A AS A, R.B AS C, 5 AS D
FROM R AS R
WHERE R.A = R.B
```

  and assume that θ is $(A \doteq 3) \vee \neg(C \doteq D)$. Then $\sigma_\theta(e)$ is translated into

```
SELECT R.A AS A, R.B AS C, 5 AS D
FROM R AS R
WHERE R.A = R.B AND ((R.A = 3) OR NOT (R.B = 5))
```

- Furthermore, $\pi_\alpha(e)$ is translated into

  SELECT X_{i_1} AS $\mathtt{A}_{i_1}, \ldots, X_{i_k}$ AS $\mathtt{A}_{i_k}$
  FROM $\mathtt{Q}_1$ AS $\mathtt{T}_1, \ldots, \mathtt{Q}_m$ AS $\mathtt{T}_m$
  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 $\ldots, \mathtt{Q}_i.\mathtt{A}_j$ AS A, ...
FROM $\mathtt{Q}_1$ AS $\mathtt{T}_1, \ldots, \mathtt{Q}_m$ AS $\mathtt{T}_m$
WHERE condition

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

SELECT $\ldots, \mathtt{Q}_i.\mathtt{A}_j$ AS B, ...
FROM $\mathtt{Q}_1$ AS $\mathtt{T}_1, \ldots, \mathtt{Q}_m$ AS $\mathtt{T}_m$
WHERE condition

Join. Assume now that we have expressions e_1 and e_2 that are translated into queries Q_{e_1} and Q_{e_2}, respectively. Then their natural join $e_1 \bowtie e_2$ is translated into

SELECT $T_1.*$ AS $\bar{\texttt{A}}$, $\{T_2.C \mid C \in \text{sort}(Q_{e_2}) - \text{sort}(Q_{e_1})\}$ AS $\bar{\texttt{B}}$
FROM $\texttt{Q}_{e_1}$ AS $\texttt{T}_1$, $\texttt{Q}_{e_2}$ AS $\texttt{T}_2$
WHERE $T_1.X_1 = T_2.X_1$ AND $\cdots$ AND $T_1.X_k = T_2.X_k$

where $X_1, \ldots, X_k$ are the attributes in $\text{sort}(Q_{e_1}) \cap \text{sort}(Q_{e_2})$ and $T_1.*$ abbreviates the list of all attributes of Q_{e_1}, with each of them assigned a name from the tuple $\bar{\texttt{A}}$. That is, all attributes of Q_{e_1} are in the output, together with attributes of Q_{e_2} that do not occur in Q_{e_1}, appropriately renamed, but only if the common attributes of Q_{e_1} and Q_{e_2} are equal. Let us illustrate this case with an example. Consider the relation names $R[A, B, D]$, $S[B, C]$, $T[A, C, D]$, and the two queries

Q_{e_1} =
```
SELECT R.A AS A, S1.C AS C, R.D AS D
FROM R AS R, S AS S1
WHERE R.B = S1.B
```

and

Q_{e_2} =
```
SELECT T.A AS A, S2.B AS B, T.D AS D
FROM S AS S2, T AS T
WHERE S2.C = T.C
```

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

SELECT $\texttt{T}_1$.A AS A, $\texttt{T}_2$.B AS B, $\texttt{T}_1$.C AS C, $\texttt{T}_1$.D AS D,
FROM $\texttt{Q}_{e_1}$ AS $\texttt{T}_1$, $\texttt{Q}_{e_2}$ AS $\texttt{T}_2$
WHERE $\texttt{T}_1$.A $=$ $\texttt{T}_2$.A AND $\texttt{T}_1$.D $=$ $\texttt{T}_2$.D

Notice that we have ordered the attributes of the resulting query alphabetically, slightly deviating from the construction.

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

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

Union. 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.

Other SQL Features Captured by RA

A very important feature of SQL is using *subqueries*. We have seen them in FROM, but they can be used in conditions in WHERE as well. 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 IN and EXISTS, potentially with negation NOT, 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
- empty(e), checking if the result of the evaluation of e is empty.

Such an 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 44.

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 33.

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 39 and 40.

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 33), 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 $R \in \text{Dom}(\mathbf{S})$, then the FO query is $\varphi_e(x_1, \ldots, x_{\text{ar}(R)})$, 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 $i \doteq j$, $i \doteq a$, $i \not\doteq j$, or $i \not\doteq a$, 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: $\varphi(x_1, \dots, x_n)$ is an FO query even if the free variables of φ are a subset of $\{x_1, \dots, x_n\}$. The semantics of such a query $\varphi(x_1, \dots, x_n)$ is the usual semantics of the FO query $\varphi'(x_1, \dots, x_n)$, 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 $\varphi(x_1, \dots, x_n)$. 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 $y_1, \dots, y_m$,
- $\{x_1, \dots, x_n\} = \{y_{k+1}, \dots, y_m\}$, and
- φ_{qf} only uses the Boolean operators $\vee$ and $\neg$.

Let $\mathrm{Dom}(\varphi) = \{a_1, \dots, a_\ell\}$. 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 $\psi(y_1, \dots, y_m)$.
(2) For all the other subformulae ψ of φ, it holds that $\psi = Q_j \cdots Q_1 \, \varphi_{\mathsf{qf}}$, for $j \in [k]$, $\mathrm{FV}(\psi) = \{y_{j+1}, \dots, y_m\}$, and the expression e_ψ, which has arity $m - j$, is equivalent to the FO query $Q_j \cdots Q_1 \, \varphi_{\mathsf{qf}}(y_{j+1}, \dots, y_m)$.

The inductive construction defines the expression

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

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. For simplicity, we consider the case where the query does not use constant values and leave the extension to arbitrary FO queries as an exercise. We start with the upper bound. Consider an FO query $q = \varphi(\bar{x})$ without constants, a database D, and a tuple $\bar{a}$. First, we observe that, if $\bar{a}$ contains values $a'_1, \ldots, a'_\ell$ that are not in $\mathrm{Dom}(D)$, we can replace them with arbitrary elements $a''_1, \ldots, a''_\ell$ in $\mathsf{Const} - \mathrm{Dom}(D)$ that can be represented using $O(\|D\|)$ many bits. 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 that FO-Evaluation is in PSPACE, 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 $\textsc{Evaluation}(\varphi, D, \eta) =$ `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.

```
1: if $\varphi$ is of the form $R(\bar{x})$ then return $R(\eta(\bar{x})) \in D$
2: else if $\varphi$ is of the form $(x_i = x_j)$ then return $\eta(x_i) = \eta(x_j)$
3: else if $\varphi$ is of the form $(x_i = a)$ then return $\eta(x_i) = a$
4: else if $\varphi$ is of the form $\neg\varphi'$ then return $\neg\textsc{Evaluation}(\varphi', D, \eta)$
5: else if $\varphi$ is of the form $\varphi' \vee \varphi''$ then
6:     return $\textsc{Evaluation}(\varphi', D, \eta) \vee \textsc{Evaluation}(\varphi'', D, \eta)$
7: else if $\varphi$ is of the form $\exists x\, \varphi'$ then
8:     return $\bigvee_{a \in \mathrm{Dom}(D)} \textsc{Evaluation}(\varphi', D, \eta[x/a])$
9:                                   $\triangleright$ $\eta[x/a]$ extends $\eta$ by setting $\eta(x) = a$.
```

Lemma 7.2. $\textsc{Evaluation}$*(φ, D, η) 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 $\neg$Evaluation(φ', D, η), which, by induction hypothesis, can be done using $O(\|\varphi\| \cdot \log \|D\|)$ space.
- When $\varphi = \varphi' \vee \varphi''$, the algorithm computes Evaluation$(\varphi', D, \eta) \vee$ Evaluation(φ'', D, η), 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 $\bigvee_{a \in \mathrm{Dom}(D)}$ Evaluation$(\varphi', D, \eta[x/a])$. This is done by iterating over the constants of 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 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 One(x), and each occurrence of the literal $\neg x$ by $\neg$One(x). The only reason why we add Zero(0) to the database is to ensure that 0 is in the active domain. 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 Dom(D) that are compatible with $\bar{x}$, and output $\bar{a}$ if and only if Evaluation(φ, D, η) $=$ `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 q-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 it simply calls the procedure Evaluation given in Algorithm 1 with input φ, D and η. More precisely, $\text{Evaluation}_\varphi(D, \eta)$ does the following: if $\text{Evaluation}(\varphi, D, \eta) = \texttt{true}$, then return `true`; otherwise, return `false`. From the correctness of Evaluation, 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

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

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.
- $\mathrm{First}(\cdot)$ contains the first element from the linear order $\prec$.
- $\mathrm{Succ}(\cdot,\cdot)$ encodes the successor relation over the linear order $\prec$.
- $\mathrm{Symbol}_a(t, c)$: at time instant t, the tape cell c contains the symbol a.
- $\mathrm{Head}(t, c)$: at time instant t, the head points at cell c.
- $\mathrm{State}(t, p)$: 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

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

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:

$$\begin{aligned}\forall x \forall y \left(\neg(x = y) \rightarrow (x \prec y \vee y \prec x)\right) \wedge \\ \forall x\, \neg(x \prec x) \wedge \\ \forall x \forall y \forall z \left((x \prec y \wedge y \prec z) \rightarrow x \prec z\right).\end{aligned}$$

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 First$(\cdot)$ contains the smallest element over $\prec$:

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

The Sentence φ_{succ}

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

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

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 FV(ψ) consists of the variables in $\bar{x}$:

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

where

- φ_{start} defines the start configuration $sc(\varepsilon)$,
- $\varphi_{\text{consistent}}$ 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

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

and the machine M is in state s

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

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 $\delta(p_i, a) = (p_j, b, \text{dir})$,

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

where

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

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

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

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 the schema $\{R[2]\}$. In this way, we can view both S and S' as a directed 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 directed 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 directed 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 directed 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 13.

10

Functional Dependencies

In a relational database 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

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

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

$$R : U \rightarrow V$$

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 $R : U \rightarrow V$ 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 $R : U \rightarrow V$ over $\mathbf{S}$, denoted $D \models \sigma$, if for each pair of tuples $\bar{a}, \bar{b} \in R^D$,

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

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 $R : U \rightarrow V$. 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 $\sigma = R : U \rightarrow V$ 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 $h_{\bar{u},\bar{v}} : \mathrm{Dom}(S) \rightarrow \mathrm{Dom}(S)$ be a function such that

$$h_{\bar{u},\bar{v}}(w) \;=\; \begin{cases} u_i & \text{if } w = v_i \text{ and } u_i < v_i, \text{ for some } i \in [k], \\[2ex] v_i & \text{if } w = u_i \text{ and } v_i < u_i, \text{ for some } i \in [k], \\[2ex] w & \text{otherwise.} \end{cases}$$

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

$$S' \;=\; \begin{cases} \perp & \text{if there is an } i \in [k] \text{ with } u_i \neq v_i \text{ and } u_i, v_i \in \mathsf{Const}, \\[2ex] h_{\bar{u},\bar{v}}(S) & \text{otherwise.} \end{cases}$$

[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 $\sigma = R : U \rightarrow V$ 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 $\sigma = R : U \rightarrow V$ 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 $\sigma = R : U \rightarrow V$ 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 52 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 $S_0, \dots, S_n$ and $S'_0, \dots, S'_m$ 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 $\mathrm{Chase}(S, \Sigma)$, which is defined as S_n for some (any) finite chase sequence $S_0, \dots, S_n$ 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, $\mathrm{Chase}(S, \Sigma)$ is either the symbol $\perp$, or a finite set of relational atoms. It is not difficult to verify that in the latter case, $\mathrm{Chase}(S, \Sigma)$ satisfies Σ. Actually, this follows from the definition of successful chase sequences.

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

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 $s = S_0, S_1, \dots, S_n$ 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 $\mathrm{Chase}(S, \Sigma) \neq \perp$, we can refer to *the* chase homomorphism of S under Σ, denoted $h_{S,\Sigma}$. It should be clear that $\mathrm{Chase}(S, \Sigma) \neq \perp$ implies $h_{S,\Sigma}(S) = \mathrm{Chase}(S, \Sigma)$.

By Lemma 10.6, $\mathrm{Chase}(S, \Sigma)$ can be computed after finitely many steps. Furthermore, assuming that $\mathrm{Chase}(S, \Sigma) \neq \perp$, 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. $\mathrm{Chase}(S, \Sigma)$ *can be computed in polynomial time. Furthermore, if $\mathrm{Chase}(S, \Sigma) \neq \perp$, then $h_{S,\Sigma}$ can be computed in polynomial time.*

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

Lemma 10.11. *Let S' be a set of atoms over $\mathbf{S}$ 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 17 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 $i \neq j$, it holds that $x_i \neq x_j$ and $y_i \neq y_j$, and
- for each $i \in \{1, \ldots, \text{ar}(R)\}$, we have $x_i = y_i$ 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 $R : \{i_1, \ldots, i_k\} \rightarrow \{j_1, \ldots, j_\ell\}$, 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 $(S_\sigma, \bar{z}) \rightarrow (D, \bar{c})$. Since $D \models \Sigma$ and $\text{Chase}(S_\sigma, \Sigma) \neq \bot$, by Lemma 10.11

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

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:

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

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

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

Intuitively, an IND $R[i_1, \dots, i_k] \subseteq P[j_1, \dots, j_k]$ 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 $R[i_1, \dots, i_k] \subseteq P[j_1, \dots, j_k]$ 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

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

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 $R[i_1, \dots, i_k] \subseteq P[j_1, \dots, j_k]$. To check whether $D \models \sigma$ we need to check that, for every tuple $(a_1, \dots, a_{\mathrm{ar}(R)}) \in R^D$, there exists a tuple $(b_1, \dots, b_{\mathrm{ar}(P)}) \in P^D$ such that $(a_{i_1}, \dots, a_{i_k}) = (b_{j_1}, \dots, b_{j_k})$. 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 43 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 $\mathbf{S}$ 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 $\{R(x_1, \ldots, x_{\mathrm{ar}(R)})\}$, where $x_1, \ldots, x_{\mathrm{ar}(R)}$ 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* $\mathrm{Chase}(S_\sigma, \Sigma) \models \sigma$.

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, $\mathrm{Chase}(S_\sigma, \Sigma) \models \Sigma$, and therefore, $\mathrm{Chase}(S_\sigma, \Sigma) \models \sigma$.

($\Leftarrow$) 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 $(a_1, \ldots, a_{\mathrm{ar}(R)}) \in R^D$. Recall also that S_σ is of the form $\{R(x_1, \ldots, x_{\mathrm{ar}(R)})\}$. 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

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

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

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

which in turn implies that there exists $(c_1, \ldots, c_{\mathrm{ar}(P)}) \in P^D$ 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 $\sigma = R[i_1, \dots, i_m] \subseteq P[j_1, \dots, j_m]$ and a tuple of variables $\bar{x} = (x_1, \dots, x_{\text{ar}(R)})$, 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 $\star \notin \{x_1, \dots, x_{\text{ar}(R)}\}$, which should be understood as a placeholder for new variables. Formally, $\mathsf{new}^\star(\sigma, \bar{x}) = P(y_1, \dots, y_{\text{ar}(P)})$, 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 $R_1(\bar{x}_1), \dots, R_n(\bar{x}_n)$, 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
- $\pi_{(i_1,\dots,i_m)}(\bar{x}_1) = \pi_{(j_1,\dots,j_m)}(\bar{x}_n)$.

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:    $\mathit{Check} := b$, where $b \in \{0, 1\}$
13: until $\mathit{Check} = 1$
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.

Exercises

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 query does not use constants, i.e., elements from Const. How can the proof be extended to FO-Evaluation in general, i.e., allowing for general FO queries?

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 $q(D) = \mathtt{true}$.

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

(Very preliminary version)

The relational model was introduced by Codd [10]. The equivalence between first-order queries and relational algebra was shown in [11]. Furthermore, Codd invented the notion of functional dependency.

Data complexity was originated by Vardi [30].

Inclusion dependencies were first defined, named, and studied in [7]. This paper contains also, among other things, Theorem 11.6, that finite and infinite implication of inclusion dependencies are the same, and Theorem 11.9 that the implication problem for inclusion dependencies is PSPACE-complete.

Part II

Conjunctive Queries

12

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 12.1: 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 12.1 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 Answer$(\bar{x})$

that appears on the left of the :– symbol is called the *head* of the rule, while the expression $R_1(\bar{u}_1), \dots, R_n(\bar{u}_n)$ 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 12.2: 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:

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

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:

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

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 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 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

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

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

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

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 12.3: 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 12.3, 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 12.4: Evaluation of CQs

Let $\mathbf{S}$ be the schema from Example 3.2, which has been also used in Example 12.2. Let D be the database of $\mathbf{S}$ shown in Figure 3.1; we recall the relations Person and Profession in Figure 12.1. The following CQ q can be used to retrieve the ids and names of actors:

$$\text{Answer}(x, y) \ \ {:\!-} \ \ \text{Person}(x, y, z), \text{Profession}(x, \text{‘actor’}).$$

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. 12.1: The relations Person and Profession for Example 12.4.

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 12.4 we consider also the Boolean CQ q'

$$\text{Answer} \;\;{:}{-}\;\; \text{Person}(x, y, z), \text{Profession}(x, \text{‘actor’}),$$

which is the Boolean version of q in Example 12.4, then $q'(D) = \texttt{true}$ since the assignment η is consistent with D. On the other hand, given the CQ q''

$$\text{Answer} \;\;{:}{-}\;\; \text{Person}(x, y, z), \text{Profession}(x, \text{‘nurse’}),$$

$q''(D) = \texttt{false}$ 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 12.5: 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 12.6

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 12.7

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

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

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

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 12.8: From CQs to SPJ Queries

Consider the CQ q defined as

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

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 $\varphi(x_1, \dots, x_{i-1}, x_{i+1}, \dots, x_k)$ if x_i occurs among the x_j's with $j \neq i$, and $\exists x_i\, \varphi(x_1, \dots, x_k)$ otherwise.

- If e_1 is k-ary with $q_{e_1} = \varphi_1(x_1, \dots, x_k)$ and $\varphi_1 = \exists \bar{z}\, \psi_1$, and e_2 is m-ary with $q_{e_2} = \varphi_2(y_1, \dots, y_m)$ and $\varphi_2 = \exists \bar{w}\, \psi_2$, then $q_{(e_1 \times e_2)}$ is the CQ $\varphi(x_1, \dots, x_k, y_1, \dots, y_m)$ 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 12.6 and 12.7 that relates the languages of $\mathrm{FO}^{\mathrm{rel}}[\wedge, \exists]$ and SPJ queries.

Corollary 12.9

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

13

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

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

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 13.1: 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 12.3), 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.,

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

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 13.2

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 13.3: CQ Evaluation via Homomorphisms

Let D and q be the database and the CQ, respectively, that have been considered in Example 12.4. We know that $q(D) = \{(\text{‘1’}, \text{‘Aretha’})\}$. By the characterization given in Theorem 13.2, 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:

$$\begin{aligned} \mathrm{Person}(h(x), h(y), h(z)) &= \mathrm{Person}(\text{‘1’}, \text{‘Aretha’}, \text{‘MPH’}) \\ \mathrm{Profession}(h(x), h(\text{‘actor’})) &= \mathrm{Profession}(\text{‘1’}, \text{‘actor’}) \end{aligned}$$

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 $\mathsf{V}_C : \mathsf{Const} \to \mathsf{Const} \cup \mathsf{Var}$ such that

- V_C is the identity on C and
- $\mathsf{V}_C(a) \in \mathsf{Var}$ for every $a \notin C$.

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.

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

Example 13.4: 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 $(D_1,(a,b)) \rightarrow_{C_1} (D_2,(c,c))$ since

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

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

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

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

We can now define the notion of preservation under homomorphisms.

Definition 13.5: 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

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

We then show the following for CQs.

Proposition 13.6

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 $(D,\bar{a}) \rightarrow_C (D',\bar{b})$ 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 $(\mathsf{V}_C(D), \mathsf{V}_C(\bar{a})) \rightarrow (D',\bar{b})$. By Theorem 13.2, $(q,\bar{x}) \rightarrow (D,\bar{a})$ 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 13.2, $\bar{b} \in q(D')$, 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 13.7

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 13.6, we get that $\bar{a} \in q(D')$. □

Preservation under Direct Products

The second preservation result stated here concerns *direct products*. Given two directed 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 13.8: 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

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

We then show the following for CQs.

Proposition 13.9

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 13.2, 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 $R(u_1, \dots, u_n)$ is an atom in q. Then $R(h(u_1), \dots, h(u_n)) \in D$ and $R(g(u_1), \dots, g(u_n)) \in D'$. Hence,

$$R\big(f(u_1), \dots, f(u_n)\big) \quad = \quad R\big(\tau(h(u_1), g(u_1)), \dots, \tau(h(u_n), g(u_n))\big)$$

belongs to $D \otimes D'$, proving that f is a homomorphism from q to $D \otimes D'$. Thus, by Theorem 13.2, $D \otimes D' \models q$, as needed. □

Expressiveness of CQs

The above preservation results allow us to delineate the expressiveness boundaries of CQs. By Theorem 12.7, 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

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

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 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 $D' \not\models q$.

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.

[2] Conjunctive queries with inequality are studied in-depth in Chapter 30.

14

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 14.1

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 13.2, $\bar{a} \in q(D)$ 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

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

which essentially stores the undirected graph G, but without loops of the form $\{v\}$ that may occur in E (that is, $\{v\} \in E$). 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

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

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 $\mathrm{Dom}(D)$, and asks whether $\bar{a} \in q(D)$, is in $\mathcal{C}$.

Corollary 14.2

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 14.3: 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 14.4: 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 14.5: 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 14.5.

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 $|S| = k$ 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 $(L_2, \kappa_2) \in$ FPT, then $(L_1, \kappa_1) \in$ 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 $\mathrm{FV}(\psi) = \{x_1, \ldots, x_n\}$.

Definition 14.6: 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$_\varphi$.

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$_\varphi$. 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 $\mathbf{S} \cup \{\mathrm{Elem}\}$

$$\forall x \forall y \left((\mathrm{Elem}(x) \wedge \mathrm{Elem}(y) \wedge x \neq y) \to \mathrm{Edge}(x, y) \right).$$

We proceed to show that there is an FPT-reduction from p-Clique to p-WD$_\varphi$. Consider an input to p-Clique given by $G = (V, E)$ and $k \geq 1$. Let

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

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 $S \subseteq \mathrm{Dom}(D)$ 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$_\varphi$ 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$_\varphi$ 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, $\bigcup_{t \geq 1} \mathrm{W}[t]$.

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 14.7

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 14.1, 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 14.1, 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:

$$\begin{aligned}
&\forall x \forall y \forall z \left((\mathrm{Hom}(x,y) \wedge \mathrm{Hom}(x,z)) \rightarrow y = z \right) \wedge \\
&\forall x \forall y \left(\mathrm{Hom}(x,y) \rightarrow \left(\mathrm{Var}(x) \wedge \mathrm{Const}(y) \right) \right) \wedge \\
&\forall x_1 \forall y_1 \forall x_2 \forall y_2 \left((\mathrm{Edge}_1(x_1,y_1) \wedge \mathrm{Hom}(x_1,x_2) \wedge \mathrm{Hom}(y_1,y_2)) \rightarrow \mathrm{Edge}_2(x_2,y_2) \right).
\end{aligned}$$

We show that there is an FPT-reduction from p-CQ-Evaluation to p-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 $\{x_1, \dots, x_n\}$ are the variables occurring in q, we define the database D' as

$$\begin{aligned}
D \;\cup\; &\{\mathrm{Const}(a) \mid a \in \mathrm{Dom}(D)\} \;\cup\; \{\mathrm{Var}(a_{x_1}), \dots, \mathrm{Var}(a_{x_n})\} \\
&\cup\; \{\mathrm{Edge}_1(a_{x_i}, a_{x_j}) \mid \mathrm{Edge}(x_i, x_j) \text{ is an atom occuring in } q\} \\
&\qquad\qquad\qquad\qquad \cup\; \{\mathrm{Edge}_2(a,b) \mid \mathrm{Edge}(a,b) \in D\}.
\end{aligned}$$

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 $\forall x \forall y \forall z\, ((\mathrm{Hom}(x,y) \wedge \mathrm{Hom}(x,z)) \rightarrow y = z)$ 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 p-WD$_\varphi$, which shows that p-CQ-Evaluation $\in$ W[1] (for constant-free Boolean CQs over a single binary relation). □

15

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 15.1: 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:     $q := (q - R(\bar{u}))$
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 16.

Containment

We first concentrate on CQ-Containment. We start by illustrating the notion of containment for CQs via a simple example.

Example 15.2: CQ Containment

Consider the CQ

$$\begin{aligned} q_1 &= \text{Answer}(y_1) \text{ :- } \text{Person}(x_1, y_1, z_1), \text{Profession}(x_1, \text{`actor'}),\\ &\qquad\qquad\qquad\qquad \text{City}(z_1, \text{`Los Angeles'}, \text{`United States'}) \end{aligned}$$

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 15.3

CQ-Containment is NP-complete.

The proof of Theorem 15.3 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 15.4: 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 13.2 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 13.2 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 $(q', \bar{x}') \rightarrow (q, \bar{x})$, 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 13.2, 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 13.2. 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 15.5: Homomorphism Theorem

Consider again the CQs q_1 and q_2 from Example 15.2, and recall that $q_1 \subseteq q_2$. This is confirmed by the Homomorphism Theorem since

$$(q_2, y_2) \;\rightarrow\; (q_1, y_1).$$

This is the case since the function $h : \mathrm{Dom}(A_{q_2}) \rightarrow \mathrm{Dom}(A_{q_1})$ 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 15.6

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 15.4, we conclude that

$$q \subseteq q' \;\text{ if and only if }\; (A_{q'}, \bar{x}') \rightarrow (A_q, \bar{x}).$$

We can also show that

$$(A_{q'}, \bar{x}') \rightarrow (A_q, \bar{x}) \;\text{ if and only if }\; (A_{q'}, \bar{x}') \;\rightarrow\; (\mathsf{G}_{A_q}(A_q), \mathsf{G}_{A_q}(\bar{x})).$$

Indeed, if $(A_{q'}, \bar{x}') \rightarrow (A_q, \bar{x})$ 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 $(A_{q'}, \bar{x}') \rightarrow (\mathsf{G}_{A_q}(A_q), \mathsf{G}_{A_q}(\bar{x}))$ 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 13.2, we get that

$$(A_{q'}, \bar{x}') \rightarrow (\mathsf{G}_{A_q}(A_q), \mathsf{G}_{A_q}(\bar{x})) \;\text{ if and only if }\; \mathsf{G}_{A_q}(\bar{x}) \in q'(\mathsf{G}_{A_q}(A_q)).$$

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 15.6 shows. In the proof of Corollary 15.6, 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 13, we can convert a database D into a set of relational atoms via the injective function $\mathsf{V}_C : \mathsf{Const} \rightarrow \mathsf{Const} \cup \mathsf{Var}$, 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 15.7

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:

$$D \models q \quad \text{if and only if } q_D \subseteq q.$$

Proof. By Theorem 13.2, we conclude that

$$D \models q \quad \text{if and only if} \quad q \rightarrow D.$$

It is easy to show that

$$q \rightarrow D \quad \text{if and only if} \quad q \rightarrow q_D.$$

Indeed, if $q \rightarrow D$ is witnessed via h, then we get that $\mathsf{V}_C \circ h$ is a homomorphism from q to q_D. Conversely, assuming that $q \rightarrow q_D$ is witnessed via g, $\mathsf{V}_C^{-1} \circ g$ is a homomorphism from q to D. By Theorem 15.4, we conclude that

$$q \rightarrow q_D \quad \text{if and only if} \quad q_D \subseteq q.$$

From the above equivalences, we get that $D \models q$ if and only if $q_D \subseteq q$. □

By Theorem 14.1, CQ-Evaluation is in NP, and thus, Corollary 15.6 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 14.1 is Boolean), Corollary 15.7 implies that CQ-Containment is NP-hard. Therefore, CQ-Containment is NP-complete, which establishes Theorem 15.3.

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 15.8

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 15.3, 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 15.3). Consider two Boolean CQs

$$q = \text{Answer} \text{ :- } R_1(\bar{u}_1), \ldots, R_n(\bar{u}_n) \qquad q' = \text{Answer} \text{ :- } R'_1(\bar{u}'_1), \ldots, R'_m(\bar{u}'_m),$$

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

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

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. □

16
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 16.1: 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 $\text{Answer}(\bar{x}) \text{ :- } R_1(\bar{u}_1), \dots, R_n(\bar{u}_n)$. The CQ q' obtained from q by removing the atom $R_i(\bar{u}_i)$, for some $i \in [n]$, is

$$\text{Answer}(\bar{x}') \ \text{:-} \ R_1(\bar{u}_1), \dots, R_{i-1}(\bar{u}_{i-1}), R_{i+1}(\bar{u}_{i+1}), \dots, R_n(\bar{u}_n),$$

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 $\text{Answer}(x, y) \text{ :- } R(x), S(y)$, then we obtain the CQ $\text{Answer}(y) \text{ :- } S(y)$ as the variable x is only mentioned in $R(x)$. On the other hand, if we remove the atom $R(x)$ from the CQ $\text{Answer}(x, y) \text{ :- } R(x), T(x, y)$, then we obtain the CQ $\text{Answer}(x, y) \text{ :- } T(x, y)$ 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 15.4). 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 15.4). 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 16.2: 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 16.2 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 16.3: 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 16.4

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 16.2) is violated, which in turn implies that there is a CQ $q'(\bar{x})$ with $A_{q'} \subsetneq A_q$ such that $(q, \bar{x}) \rightarrow (q', \bar{x})$. 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 16.5. *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*

$$(q_1, \bar{y}_1) \rightarrow (q_3, \bar{y}_1) \quad \textit{and} \quad A_{q_3} \subsetneq A_{q_1}.$$

Proof. By Theorem 15.4, we conclude that

$$(q_1, \bar{y}_1) \rightarrow (q_2, \bar{y}_2) \quad \text{and} \quad (q_2, \bar{y}_2) \rightarrow (q_1, \bar{y}_1).$$

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 $(q_1, \bar{y}_1) \rightarrow (q_3, \bar{y}_1)$. 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 16.5, we conclude that there exists a CQ $q'''(\bar{x})$ such that $(q', \bar{x}) \rightarrow (q''', \bar{x})$ and $A_{q'''} \subsetneq A_{q'}$. This contradicts our hypothesis that q' is a core of q, and the claim follows. □

By Proposition 16.4, 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 16.6. *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 $q^*(\bar{x})$ :- $R_1(\bar{u}_1), \dots, R_n(\bar{u}_n)$,  where $S = \{R_1(\bar{u}_1), \dots, R_n(\bar{u}_n)\}$
```

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 16.2), 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 16.3 (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 16.7

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 15.4,

$$(q', \bar{x}') \to (q'', \bar{x}'') \qquad \text{and} \qquad (q'', \bar{x}'') \to (q', \bar{x}').$$

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 16.8. *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 15.4, $q' \equiv q'''$. 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 15.4, 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 16.8, 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. $\square$

From Proposition 16.4, which tells us that a core of a CQ q is a minimization of q, and Proposition 16.7, we immediately get the following corollary:

Corollary 16.9

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 16.9 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.

17

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 16, 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 17.1: 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 13.2, 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 17.2

CQ-Containment-FD is NP-complete.

It is clear that the NP-hardness is inherited from CQ containment without constraints (see Theorem 15.3). 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 17.1. 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 $\mathrm{Chase}(q, \Sigma)$ instead of $\mathrm{Chase}(A_q, \Sigma)$, 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 17.3

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. $\mathrm{Chase}(q, \Sigma) \neq \bot$ implies $(A_{q'}, \bar{x}') \to (\mathrm{Chase}(q, \Sigma), h_{q,\Sigma}(\bar{x}))$.

Proof. For brevity, let $S = \mathrm{Chase}(q, \Sigma)$ 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 13.2, 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 13.2, $(q, \bar{x}) \to (D, \bar{a})$. 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 13.2, $\bar{a} \in q'(D)$, which implies that $q \subseteq_\Sigma q'$.

Since in both cases we get that $q \subseteq_\Sigma q'$, the claim follows. $\square$

The following is an easy consequence of Theorem 17.3 and Theorem 13.2.

Corollary 17.4

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 $S = \mathrm{Chase}(q, \Sigma)$, 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, $\mathrm{Chase}(q, \Sigma)$ can be computed in polynomial time. Moreover, if $\mathrm{Chase}(q, \Sigma) \neq \bot$, then the chase homomorphism $h_{q,\Sigma}$ can be also computed in polynomial time. Since CQ-Evaluation is in NP (see Theorem 14.1), we conclude that CQ-Containment-FD is also in NP, and Theorem 17.2 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 17.5: Containment of CQs Under INDs

Consider the CQs q_1 and q_2 defined as

$$\begin{aligned} \mathrm{Answer}(x_1, y_1) &\ \text{:-}\ R(x_1, y_1), R(y_1, z_1), P(z_1, y_1) \\ \mathrm{Answer}(x_2, y_2) &\ \text{:-}\ R(x_2, y_2), R(y_2, z_2), S(x_2, y_2, z_2), \end{aligned}$$

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

$$\mathrm{Answer}(x_1, y_1) \ \text{:-}\ R(x_1, y_1), R(y_1, z_1), P(z_1, y_1), S(x_1, y_1, w_1), R(y_1, w_1),$$

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 17.2), this is not true for INDs:

Theorem 17.6

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 17.5. 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 17.3. 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 12.3), 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

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

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

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

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 17.7: 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 13.1 merely extends to possibly infinite databases, which allows us to state a result analogous to Theorem 13.2:

Theorem 17.8

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 17.3.

Theorem 17.9

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 17.10: 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 17.9 and 17.10, combined with Theorem 17.8, is the following:

Corollary 17.11

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:

$$q \subseteq_\Sigma q' \quad \text{if and only if} \quad \mathsf{G}_S(\bar{x}) \in q'(\mathsf{G}_S(S)).$$

Due to Corollary 17.11, 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 17.6, 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 $\sigma = R[i_1, \dots, i_m] \subseteq P[j_1, \dots, j_m]$, a tuple $\bar{u} = (u_1, \dots, u_{\text{ar}(R)})$, 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, $\mathsf{new}^V(\sigma, \bar{u}) = P(v_1, \dots, v_{\text{ar}(P)})$, 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 $i, j \in [\text{ar}(P)] - \{j_1, \dots, j_m\}$, $i \neq j$ 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 $V_1, \dots, V_n$, for $n \geq 0$, $\mathcal{S}$ is a sequence of disjoint sets of relational atoms $S_0, \dots, S_n$, 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 $\ell \in [\text{ar}(P)] - \{j_1, \dots, j_m\}$.

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 $(A_{q'}, \bar{x}') \rightarrow (A, \bar{x})$. 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 $\text{ContainmentWitness}(q, q', \Sigma)$

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.

```
 1: $A_\triangledown := A$, where $A \subseteq A_q$ with $|A| \leq |A_{q'}|$
 2: $A_\triangleright := \emptyset$
 3: $Q := A$, where $A \subseteq A_\triangledown$
 4: $V := \{y_1, \ldots, y_m\} \subset \mathsf{Var} - \text{Dom}(A_q)$ for some $m \in [3 \cdot |A_{q'}| \cdot \max_{R \in \mathbf{S}}\{\text{ar}(R)\}]$
 5: repeat
 6:     repeat
 7:         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
 8:             $N := \mathsf{new}^V(\sigma, \bar{u})$
 9:             $V := V - \text{Dom}(\{N\})$
10:             $A_\triangleright := A_\triangleright \cup \{N\}$
11:         if $|A_\triangleright| < |A_{q'}|$ then
12:             $Next := b$, where $b \in \{0, 1\}$
13:         else
14:             $Next := 1$
15:     until $Next = 1$
16:     if $A_\triangleright = \emptyset$ then
17:         return false
18:     $V := V \cup ((\text{Dom}(A_\triangledown) \cap \mathsf{Var}) - (\text{Dom}(A_\triangleright) \cup \text{Dom}(Q)))$
19:     $A_\triangledown := A_\triangleright$
20:     $A_\triangleright := \emptyset$
21:     $Q := Q \cup A$, where $A \subseteq A_\triangledown$
22:     if $|Q| < |A_{q'}|$ then
23:         $Evaluate := b$, where $b \in \{0, 1\}$
24:     else
25:         $Evaluate := 1$
26: until $Evaluate = 1$
27: return $\mathsf{G}_Q(\bar{x}) \in q'(\mathsf{G}_Q(Q))$
```

Lemma 17.12. *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 $S = \mathrm{Chase}(q, \Sigma)$, 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 17.11 and Lemma 17.12, 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 $3 \cdot |A_{q'}| \cdot \max_{R \in \mathbf{S}}\{\mathrm{ar}(R)\}$ 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 $\mathrm{Dom}(A_{\triangledown})$, 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 **S**, and an IND σ over **S**, and asks whether $\Sigma \models \sigma$, i.e., whether for every database over **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 $\sigma = R[i_1, \dots, i_k] \subseteq P[j_1, \dots, j_k]$. The CQ q is defined as

$$\mathrm{Answer}(x_{i_1}, \dots, x_{i_k}) \ \text{:-}\ R(x_1, \dots, x_{\mathrm{ar}(R)}),$$

while the CQ q' is defined as

$$\mathrm{Answer}(x_{i_1}, \dots, x_{i_k}) \ \text{:-}\ R(x_1, \dots, x_{\mathrm{ar}(R)}), P(x_{f(1)}, \dots, x_{f(\mathrm{ar}(R))}),$$

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 $\{j_1, \dots, j_k\}$ 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 $q \subseteq_\Sigma q'$.

Exercises

Exercise 2.1. Let q be the CQ given in Example 12.8. 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 12.7, 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 12.7. 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 12.7, 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 $\bar{a} = (a_1, \dots, a_n)$ and $\bar{b} = (b_1, \dots, b_n)$, define the n-ary tuple $\bar{a} \otimes \bar{b}$ as $\big(\tau_C(a_1, b_1), \dots, \tau_C(a_n, b_n)\big)$. 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 $(L_2, \kappa_2) \in \text{FPT}$, then $(L_1, \kappa_1) \in \text{FPT}$.

Exercise 2.11. Recall that in the proof of the fact that p-CQ-Evaluation is in W[1] (see Theorem 14.7), 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 15.7 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

$$\text{Answer} \ \text{:-} \ E(x_1, y_1), E(y_1, z_1), E(z_1, w_1), E(w_1, x_1), E(x_2, y_2), E(y_2, x_2).$$

Assume that E is used to represent the edge relation of a directed graph G. What does q_1 check for G? Compute the core of q_1.

(ii) Consider the Boolean CQ q_2 over the schema $\{R[1], S[1]\}$ defined as

$$\text{Answer} \ \text{:-} \ R(x), S(x), R(y), S(y).$$

Compute the core of q_2.

(iii) Consider the CQ q_3 over the schema $\{R[1], S[1]\}$ defined as

$$\text{Answer}(x, y) \ \text{:-}\ R(x), S(x), R(y), S(y).$$

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* (CQ$^=$) similarly to a CQ, but we additionally allow equational atoms. Such queries can therefore be written as rules

$$\text{Answer}(\bar{x}) \ \text{:-}\ R_1(\bar{u}_1), \ldots, R_n(\bar{u}_n), y_1 = v_1, \ldots, y_k = v_k,$$

where $\{v_1, \ldots, v_k\} \subseteq \mathsf{Var} \cup \mathsf{Const}$.

1. Why are the queries $\text{Answer}(x) \ \text{:-}\ x = y$ and $\text{Answer}(x) \ \text{:-}\ x = a$ not expressible as CQs?
2. Prove that (i) Theorem 14.1, (ii) Theorem 15.3, and (iii) Theorem 15.8 also hold for CQ$^=$.
3. Prove that queries in 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 17.12.

Exercise 2.23. Prove that the reduction at the end of Chapter 17 from IND-Implication to CQ-Containment-IND, which establishes that the latter is PSpace-hard, is correct.

Bibliographic Comments

(Very preliminary version)

Conjunctive queries were first studied in [8].

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 14.1, 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.

18

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.

Conjunctive Queries and Hypergraphs

We have seen in Theorem 14.1 that the evaluation problem for CQs is NP-complete. So, the reader may wonder why databases are so successful in practice, even though the most fundamental database problem on the most common class of queries is NP-complete. The crux is that the *syntactic shape* of a CQ plays a key role on how complex is its evaluation. For instance, assume that we have a database D of the schema $\{E[2]\}$, i.e., D can be understood as a directed graph with E being the edge relation. Evaluating the CQ

$$\text{Answer} \coloneq E(x_1, x_2), E(x_2, x_3), E(x_3, x_4), E(x_4, x_1), E(x_1, x_3), E(x_2, x_4)$$

can be seen as matching a variant of the 4-clique, namely a graph of the form

$$\begin{tikzcd}
x_4 \arrow[d] & x_3 \arrow[l] \\
x_1 \arrow[r] \arrow[ur] & x_2 \arrow[u] \arrow[ul]
\end{tikzcd}$$

in D. In fact, this correspondence between evaluation of CQs and graph matching is precisely what we used in Theorem 14.1 to reduce 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 18.1: 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 (recall the definition of tree from Chapter 2), while preserving the connectivity of elements that occur in different hyperedges.

Definition 18.2: Join Tree and Acyclic Hypergraph

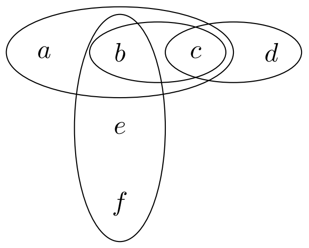
Given a hypergraph $H = (V, E)$, a tree T is a *join tree* of H if

- 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.

Example 18.3: 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:

$$\begin{tikzcd}
& \{a,b,c\} \arrow[dl, dash] \arrow[d, dash] \arrow[dr, dash] & \\
\{b,c\} & \{c,d\} & \{b,e,f\}
\end{tikzcd}$$

Recall that by convention, the root of T_1 is depicted on top and its edges are directed downwards. 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:

$$\begin{tikzcd}
& \{a,b,c\} \arrow[dl, dash] \arrow[d, dash] \\
\{b,c\} & \{c,d\}
\end{tikzcd}$$

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 $\{c, e\}$ as a children of $\{a, b, c\}$.

$$\begin{tikzcd}
& \{a,b,c\} \arrow[dl, dash] \arrow[d, dash] \arrow[dr, dash] \arrow[drr, dash] & & \\
\{b,c\} & \{c,d\} & \{b,e,f\} & \{c,e\}
\end{tikzcd}$$

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

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

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 18.4: Acyclicity of CQs

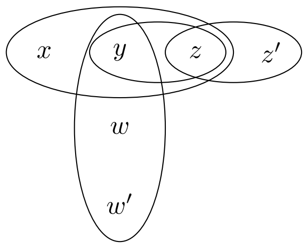
An *acyclic conjunctive query* (ACQ) is a conjunctive query q such that its associated hypergraph H_q is acyclic.

Example 18.5: Acyclic and Non-Acyclic CQs

Consider the following CQ q_1:

$$\text{Answer}(x, y) \ :\!-\ R(x, y, z), T(y, z), S(y, w, w'), T(z, z').$$

Then we have that H_{q_1} is the following hypergraph:



We know from Example 18.3 that H_{q_1} is an acyclic hypergraph, as H_{q_1} can be obtained from the hypergraph H_1 in Example 18.3 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:

$$\text{Answer}(x, y) \coloneq R'(x, y, z, a), T'(y, z, a), S(y, w, w'), T'(z, z', b)$$

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:

$$\text{Answer}(x, y) \coloneq R(x, y, z), T(y, z), S(y, w, w'), T(z, z'), T(z, w)$$

is not acyclic as the hypergraph H_{q_3} is not acyclic. Notice that this latter fact is also obtained from Example 18.3, as H_{q_3} can be obtained from the hypergraph H_2 in Example 18.3 by renaming the nodes of H_2. Finally, consider the following CQ q_4:

$$\text{Answer}(x, y) \coloneq R(x, y, z), R(y, y, z), T(y, z), S(y, w, w'), T(z, z').$$

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 18.6: 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):

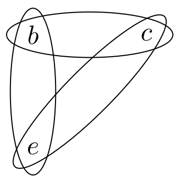
1. Delete a vertex that appears in at most one hyperedge.
2. Delete a hyperedge that is contained in another hyperedge.

The proof of Proposition 18.6 is left as an exercise for the reader (see Exercise 3.1). The characterization given in this proposition 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 18.6 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 in polynomial time a join tree of H_q when q is acyclic (see Exercise 3.2).

Example 18.7: Application of GYO Algorithm

Consider the hypergraph H_1 in Example 18.3. 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 18.3, 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 summarize this result in the following proposition.

Proposition 18.8: CQ Acyclicity Checking

There exists a linear-time algorithm that, given a CQ q, checks whether q is acyclic, and if this is the case constructs a join tree of H_q.

We will exploit this proposition later in the presentation of efficient evaluation algorithms for acyclic conjunctive queries.

19

Efficiently Evaluating Boolean ACQs

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 16).

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. In the named relational algebra, one would define the semijoin as follows:

Semijoin. If e_1, e_2 are named RA expressions of sort U_1 and U_2, respectively, then their *semijoin* $(e_1 \ltimes e_2)$ is a named RA expression of sort U_1. It is defined as $(e_1 \ltimes e_2) := \pi_{U_1}(e_1 \bowtie e_2)$.

In this chapter, we use the semijoin operator on outputs of conjunctive queries. 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 19.1: 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:

$$\begin{aligned} q &= \text{Answer}(x,y,z) \ \text{:-}\ R(x,y,z) \\ q' &= \text{Answer}(z,y,w) \ \text{:-}\ S(z,y,w). \end{aligned}$$

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 commutative.

We now explain the relationship between the CQs in ACQ and the semijoin operator. Let $q = \text{Answer} \ \text{:-}\ R_1(\bar{u}_1), \ldots, R_n(\bar{u}_n)$ 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 $\bar{u}_i$. 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 $q_s = \text{Answer}(\bar{y}_i) \ \text{:-}\ R_{j_1}(\bar{u}_{j_1}), \ldots, R_{j_p}(\bar{u}_{j_p})$, 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 $q = \text{Answer} \ \text{:-}\ R_1(\bar{u}_1), \ldots, R_n(\bar{u}_n)$ is a Boolean ACQ, and suppose that T is a 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 $Q_r = \text{Answer}(\bar{y}_\ell) \ \text{:-}\ R_1(\bar{u}_1), \ldots, R_n(\bar{u}_n)$, 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 following 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 19.2

Let q be a Boolean ACQ, T a 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 $s_1, \dots, s_p$, then

$$Q_s(D) \;=\; \bigcap_{i=1}^{p} \big(q_s(D) \ltimes Q_{s_i}(D)\big)\,.$$

Proof. Assume that $q = \text{Answer} :\!\!-\, R_1(\bar{u}_1), \dots, R_n(\bar{u}_n)$ 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 $s_1, \dots, s_p$. Moreover, assume $s = X_\ell$, $s_1 = X_{k_1}$, …, $s_p = X_{k_p}$, where $\ell, k_1, \dots, k_p$ 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 19.2 to evaluate a Boolean ACQ, as shown in Algorithm 6. The correctness of the algorithm follows from Proposition 19.2—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 join tree of $H_q$
 2: $N :=$ the set of nodes of $T$
 3: $r :=$ the root of $T$
 4: while $N \neq \emptyset$ do
 5:     Choose $s \in N$ 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 $s_1, \ldots, s_p$ be the children of $s$ in $T$
11:         $Q_s(D) := \bigcap_{i=1}^{p} \big(q_s(D) \ltimes Q_{s_i}(D)\big)$
12:     $N := N - \{s\}$
13: if $Q_r(D) \neq \emptyset$ then
14:     return true
15: else
16:     return false
```

We now analyze the complexity of the algorithm. We first notice that Proposition 18.8 tell us that a join tree T of H_q can be computed 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 time needed to compute $q(D) \ltimes q'(D)$, given $q(D)$ and $q'(D)$, is $O(N \log N)$ with $N = \|q(D)\| + \|q'(D)\|$. 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 19.3

ACQ-Evaluation can be solved in time $O(\|D\| \cdot \log \|D\| \cdot \|q\|)$.

Proof. We already proved in the preceding analysis that the theorem holds for Boolean ACQs. Assume now that we are given a non-Boolean ACQ $q = \text{Answer}(\bar{x}) \text{ :- } R_1(\bar{u}_1), \ldots, R_n(\bar{u}_n)$, 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 19.4: Consistency Property

Let $q = \text{Answer} \text{ :- } R_1(\bar{u}_1), \ldots, R_n(\bar{u}_n)$ be a Boolean ACQ and D a database, and $q_i = \text{Answer}(\bar{x}_i) \text{ :- } R_i(\bar{u}_i)$ 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 $i \in [n]$.

For the other direction, assume that nonempty sets $S_1, \ldots, S_n$ as described in Item 2 exist. Let T be an arbitrary join tree of H_q. One can then prove by induction the following for each node s of T (recall the notation in Algorithm 6):

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, ..., S_n as described in Item 2 of Proposition 19.4.

Algorithm 7 CONSISTENCY(q, D)

```
Input: A Boolean ACQ $q = $ Answer :- $R_1(\bar{u}_1), \ldots, R_n(\bar{u}_n)$ 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:     $S_i := S_i \ltimes S_j$
4: if $S_i \neq \emptyset$ for every $i \in [n]$ then
5:     return true
6: else
7:     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 empty 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 correct.

Proposition 19.5

Given a Boolean ACQ q and a database D, we have that $q(D) = \texttt{true}$ if and only if $\textsc{Consistency}(q, D) = \texttt{true}$.

We leave the proof of Proposition 19.5 as an exercise for the reader (see Exercise 3.6).

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 3.7). Interestingly, the consistency algorithm continues being correct for the evaluation problem of such a class of CQs.

Proposition 19.6

Let q be a Boolean CQ whose core is acyclic and D a database. Then we have that $q(D) = \texttt{true}$ if and only if $\textsc{Consistency}(q, D) = \texttt{true}$.

Proof. Let q be a CQ whose core q' is acyclic. It is then the case that:

$$\begin{array}{lll} q(D) = \texttt{true} & \text{if and only if} & q'(D) = \texttt{true} \\ & \text{if and only if} & \textsc{Consistency}(q', D) = \texttt{true} \\ & \text{if and only if} & \textsc{Consistency}(q, D) = \texttt{true}. \end{array}$$

The first equivalence holds since $q \equiv q'$, the second one since q' is acyclic (based on Proposition 19.5), and the last one given the fact that if there exists a homomorphism from q to q' and $\textsc{Consistency}(q', D) = \texttt{true}$, then $\textsc{Consistency}(q, D) = \texttt{true}$ (see Exercise 3.9). □

As a corollary, we obtain the following, for the class $\mathsf{ACoreCQ}$ of CQs that have an acyclic core.

Theorem 19.7

$\mathsf{ACoreCQ}$-$\mathsf{Evaluation}$ is in PTIME.

20

Efficiently Evaluating General ACQs

In this chapter we show how the ideas from Chapter 19 can be extended to obtain an efficient algorithm for computing the output of acyclic conjunctive queries. More precisely, we study the following problem.

Problem: ACQ-Answering

Input: A query q from ACQ and a database D
Output: $q(D)$

So, in contrast to ACQ-Evaluation, where the task is to test if a given tuple $\bar{a}$ is an element of $q(D)$, we now need to compute the entire set $q(D)$.

Before we dive into the details and present an algorithm that we claim to be efficient, we need to clarify what "efficient algorithm" in this context actually means. Until now, we have always studied *decision problems* in the book, which are problems that answered with `true` or `false`. Algorithms for such problems are typically considered to be efficient if *their runtime is always polynomial in the size of the input.*[1] For answering ACQs, however, this definition arguably does not make much sense, because if $q(\bar{x})$ is an ACQ and D is a database, then $q(D)$ can contain exponentially many tuples in $\|D\| + \|q\|$. We illustrate this in an example.

Example 20.1: ACQs with Exponentially Large Output

For $n \in \mathbb{N}$, consider the database D_n containing the facts $R(1,i)$ for every $i \in [n]$ and the CQ q_n defined as

$$\text{Answer}(x, y_1 \ldots, y_n) \text{ :- } R(x, y_1), R(x, y_2), \ldots, R(x, y_n) \; .$$

[1] We realize that this definition of "efficiency" is painting with a very broad brush, even when considering decision problems. Depending on the concrete research field, this notion may need to be significantly refined.

Then $q_n(D_2)$ has 2^n many output tuples, which means that the output of an ACQ can be exponentially large in the size of the query and arbitrarily much larger than the size of the database. The number of tuples in $q_n(D_n)$ is n^n, which is exponential in both the size of the data and the size of the query.

Similarly, it also does not make sense to ask for an algorithm whose runtime is polynomial in the size of $q(D)$. If q is a Boolean query, whose output always has constant size, this requirement would mean that the algorithm would need to run in constant time. The notion that we adopt here is *total polynomial time*.

Definition 20.2: Total Polynomial Time

Let f be a computable function. An algorithm A is said to compute f in *total polynomial time* if there exists a polynomial function $p : \mathbb{N} \to \mathbb{R}_0^+$ such that, for every input x, algorithm A computes $f(x)$ within time $p(\|x\| + \|f(x)\|)$.

The notion of total polynomial time indeed seems to be a well-suited first step for measuring the efficiency of query evaluation algorithms. It gives algorithms the time to read the entire input and allows it to use polynomial time for every produced output tuple. We emphasise that, in practice, where faster is better, usually stronger guarantees are needed. Typically, one would desire less than linear time between subsequent output tuples, for example.

Yannakakis's Algorithm for Answering ACQs

We now present the full-fledged version of Yannakakis's algorithm that solves the ACQ-Answering problem. Let $q(\bar{x}) \coloneq R_1(\bar{u}_1), \dots, R_n(\bar{u}_n)$ be an 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 $\bar{u}_i$. In addition to Q_s and q_s, which we already defined in Chapter 19, 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 conjunctive query $A_s(\bar{y}_i) \coloneq R_1(\bar{u}_1), \dots, R_n(\bar{u}_n)$.

Proposition 20.3

Algorithm 8 correctly computes the sets $A_s(D)$.

Proof. TODO.

Algorithm 8 RealYannakakis(q, D)

```
Input: An ACQ $q(\bar{x})$ and a database $D$
Output: $q(D)$
 1: $T$ := a join tree of $H_q$
 2: $N_1, N_2, N_3$ := the set of nodes of $T$
 3: $r$ := the root of $T$
 4: while $N_1 \neq \emptyset$ do
 5:     Choose $s \in N_1$ such that no child of $s$ is in $N_1$
 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 $s_1, \ldots, s_p$ be the children of $s$ in $T$
11:         $Q_s(D) := \bigcap_{i=1}^{p} \big(q_s(D) \ltimes Q_{s_i}(D)\big)$
12:     $N_1 := N_1 - \{s\}$
13: $A_r := Q_r(D)$
14: while $N_2 \neq \emptyset$ do
15:     Choose $s \in N_2$ such that its parent is not in $N_2$
16:     for each child $s'$ of $s$ in $T$ do
17:         $A_{s'}(D) := Q_{s'}(D) \ltimes A_s(D)$
18:     $N_2 := N_2 - \{s\}$
19: Rename all $A_s(D)$ to $O_s(D)$        ▷ We will compute the output in the $O_s(D)$
20: while $N_3 \neq \emptyset$ do
21:     Choose $s \in N_3$ such that no child of $s$ is in $N_3$
22:     if $s$ is not a leaf in $T$ then
23:         Let $s_1, \ldots, s_p$ be the children of $s$ in $T$
24:         for $j = 1, \ldots, p$ do
25:             $O_s(D) := \pi_{s \cup \bar{x}}\big(O_s(D) \bowtie O_{s_j}(D)\big)$
26:     $N_3 := N_3 - \{s\}$
27: return $O_r(D)$
```

Theorem 20.4

Yannakakis's Algorithm solves ACQ-Answering in total polynomial time.

21

Treewidth

Acyclic conjunctive queries were defined in Chapter 18 by representing a CQ as a hypergraph, and then verifying whether this hypergraph can be encoded as a tree. A natural way to extend this idea is by measuring the similarity of a hypergraph with a tree, and then defining a hypergraph as "nearly-acyclic" if this similarity is bounded. In this chapter, we take a first step in this line of work by defining a graph representation for conjunctive queries, and then introducing the well-studied graph-theoretic notion of treewidth that measures the similarity of a graph with a tree (and, in particular, generalizes the notion of acyclicity for graphs). In this way, we obtain a first formalization of the concept of near-acyclicity of a conjunctive query, based on its underlying graph representation, which retains several of the good properties of acyclicity shown in Chapters 18 and 19. In particular, CQs of bounded treewidth can be evaluated in polynomial time. In Chapter 22, we will continue with this line of work by introducing a notion of similarity with trees for hypergraphs, which generalizes both the notion presented in this chapter and the notion of acyclicity given in Chapter 18.

The Treewidth of a Graph

To define the notion of treewidth for conjunctive queries, first we need to introduce this notion for undirected graphs, which in turn requires defining the concept of tree decomposition of an undirected graph.

Definition 21.1: Tree Decomposition of an Undirected Graph

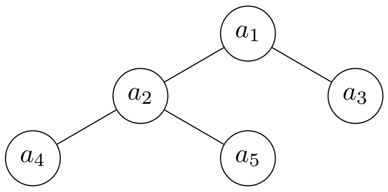
Let $G = (V, E)$ be an undirected graph. A *tree decomposition* of G is a pair (T, χ) such that $T = (V_T, E_T)$ is a non-empty tree, χ is a function that assigns a subset of V to each $s \in V_T$, and:

1. For each edge $\{a, b\} \in E$, there exists $s \in V_T$ such that $\{a, b\} \subseteq \chi(s)$.

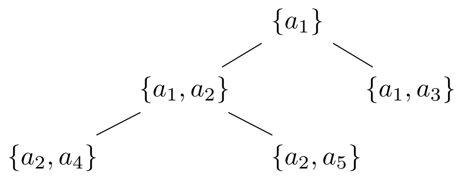
2. For each node $v \in V$, the set of nodes $\chi^{-1}(v) = \{s \in V_T \mid v \in \chi(s)\}$ is non-empty and induces a subtree (connected subgraph) of T.

Example 21.2: Tree Decomposition of an Undirected Graph

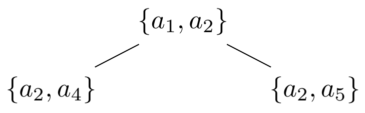
Let G_1 be the following undirected graph:



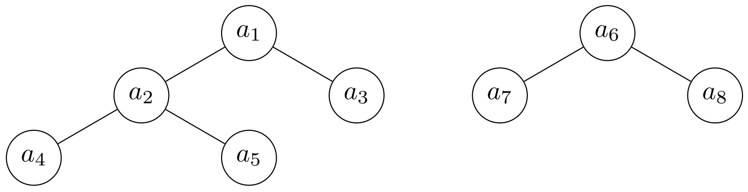
Then a tree decomposition (T_1, χ_1) of G_1 is depicted in the following figure, where the label of a node s of T_1 corresponds to $\chi_1(s)$:



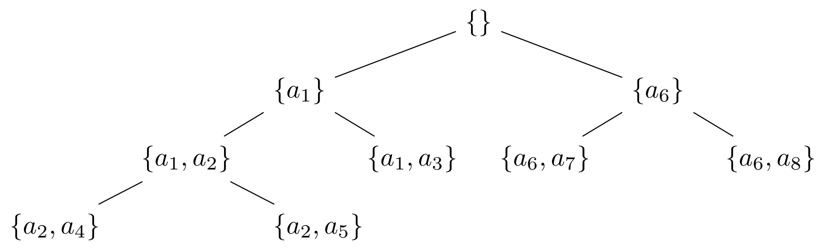
Notice that the set $\chi_1^{-1}(a_2)$ induces the following subtree of T_1:



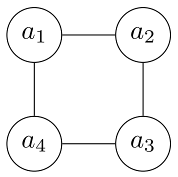
Now let G_2 be the following undirected graph:



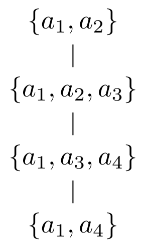
Then a tree decomposition of G_2 is depicted in the following figure:



Finally, let G_3 be the following undirected graph:



Then a tree decomposition of G_3 is depicted in the following figure:



The notion of treewidth of a graph is defined by considering all its possible tree decompositions.

Definition 21.3: Treewidth of an Undirected Graph

Let $G = (V, E)$ be an undirected graph. The width of a tree decomposition (T, χ) of G, where $T = (V_T, E_T)$, is the number $\max\{|\chi(s)| \mid s \in V_T\} - 1$. Moreover, the *treewidth* of G, denoted by $\text{tw}(G)$, is defined is the minimum width over all tree decompositions of G.

Let G_1, G_2 and G_3 be the undirected graphs in Example 21.2. From the tree decomposition of G_1 given in this example, we know that $\text{tw}(G_1) \leq 1$. Besides, every such a decomposition (T, χ) has to include a node s such that $\{a_1, a_2\} \subseteq \chi(s)$, so that the width of (T, χ) is at least 1. Hence, we conclude that $\text{tw}(G_1) = 1$. In fact, the term "-1" is included in Definition 21.9 to let trees have treewidth 1.

In the same way, it is possible to conclude that $\text{tw}(G_2) = 1$. Moreover, from the tree decomposition of G_3 given in Example 21.2, we know that $\text{tw}(G_3) \leq 2$. In what follows, we introduce some tools that allow us to conclude that $\text{tw}(G_3) = 2$. In fact, as a more general result, we obtain that the treewidth of a cycle is 2, which can be interpreted as an indication of how close a cycle is to a tree.

Lemma 21.4. *Given two undirected graphs G_1 and G_2, if G_1 is a subgraph of G_2, then $tw(G_1) \leq tw(G_2)$.*

Proof. Let (T, χ) be a tree decomposition of G_2, and assume that $G_1 = (V_1, E_1)$. Then for every node s of T, define $\chi'(s) = \chi(s) \cap V_1$. It is straightforward to prove that (T, χ') is a tree decomposition of G_1. Moreover, the width of (T, χ') is at most the width of (T, χ). Hence, given that (T, χ) is an arbitrary tree decomposition of G_2, we conclude that $\text{tw}(G_1) \leq \text{tw}(G_2)$. □

In the following lemma, we need a notion of separation for graphs. More precisely, assume that $G = (V, E)$ is an undirected graph, and $V_1, V_2, S \subseteq V$. Then S separates V_1 from V_2 in G if for every $v_1 \in V_1$, $v_2 \in V_2$ and undirected path π from v_1 to v_2 in G, a node in S occurs in π.

Lemma 21.5. *Let (T, χ) be a tree decomposition of an undirected graph G, where $T = (V_T, E_T)$. Moreover, let $(t, u) \in E_T$ and $V_u = \{v \in V_T \mid v$ is a descendant of u in $T\}$. Then $\chi(t) \cap \chi(u)$ separates $\chi(V_u)$ from $\chi(V_T - V_u)$ in G.*

The proof Lemma 21.5 is left as an exercise for the reader. By using the previous two lemmas, we can establish the following values for the treewidth of a graph.

Proposition 21.6

Assuming that G is an undirected graph, all of the following statements are true.

1. If G consists of n nodes, where $n \geq 0$, then $\text{tw}(G) \leq n - 1$.
2. If G is a cycle with at least three nodes, then $\text{tw}(G) = 2$.
3. G is acyclic if and only if $\text{tw}(G) \leq 1$.
4. If G is a clique with n nodes, where $n \geq 0$, then $\text{tw}(G) = n - 1$.

Proof.

1. Assume that $G = (V, E)$, where $|V| = n$. By consider a tree decomposition (T, χ) of G such that T consists of a single node s with $\chi(s) = V$, we conclude that $\text{tw}(G) \leq |V| - 1 = n - 1$.

2. Assume that $G = (V, E)$, where $V = \{1, 2, 3, \dots, n\}$, $n \geq 3$ and $E = \{\, \{i, i+1\} \mid i \in [1, n-1] \,\} \cup \{\, \{1, n\} \,\}$. It is straightforward to generalize the construction for graph G_3 in Example 21.2 to show that $\text{tw}(G) \leq 2$. For the sake of contradiction, assume that $\text{tw}(G) \leq 1$. Then there exists a tree decomposition (T, χ) of G such that $T = (V_T, E_T)$ and $|\chi(s)| \leq 2$ for every $s \in V_T$. Moreover, assume that for every $(t, u) \in E_T$, it holds that $\chi(t) \neq \chi(u)$ (if this is not the case, then u can be removed, and the children of u can become children of t).
   Given that $\{1, 2\}, \{1, n\} \in E$, there exist nodes $s_1, s_2 \in V_T$ such that $\chi(s_1) = \{1, 2\}$ and $\chi(s_2) = \{1, n\}$. First, assume that s_2 is a descendant of s_1. Then given that $s_1 \neq s_2$, there exists an edge $(s_1, u) \in E_T$ such that $s_2 \in V_u$, where $V_u = \{v \in V_T \mid v \text{ is a descendant of } u \text{ in } T\}$. By Lemma 21.5, we know that $\chi(s_1) \cap \chi(u)$ separates $\chi(V_u)$ from $\chi(V_T - V_u)$ in G. Hence, given that $2 \in \chi(V_T - V_u)$, $n \in \chi(V_u)$, $2, 3, \dots, n$ is an undirected path from 2 to n in G and $\chi(s_1) = \{1, 2\}$, we have that $2 \in \chi(s_1) \cap \chi(u)$. Moreover, given that $1 \in \chi(s_1)$, $1 \in \chi(s_2)$ and T is a tree decomposition of G, we have that $1 \in \chi(u)$. Therefore, given that $|\chi(u)| \leq 2$, we conclude that $\chi(u) = \chi(s_1) = \{1, 2\}$, which contradicts one of our initial assumptions. Second, assume that s_2 is not a descendant of s_1. Then given that $s_1 \neq s_2$, there exists an edge $(t, s_1) \in E_T$ such that $s_2 \in V_T - V_{s_1}$, where $V_{s_1} = \{v \in V_T \mid v \text{ is a descendant of } s_1 \text{ in } T\}$. By Lemma 21.5, we know that $\chi(t) \cap \chi(s_1)$ separates $\chi(V_{s_1})$ from $\chi(V_T - V_{s_1})$ in G. Hence, given that $2 \in \chi(V_{s_1})$, $n \in \chi(V_T - V_{s_1})$, $2, 3, \dots, n$ is an undirected path from 2 to n in G and $\chi(s_1) = \{1, 2\}$, we have that $2 \in \chi(t) \cap \chi(s_1)$. Moreover, given that $1 \in \chi(s_1)$, $1 \in \chi(s_2)$ and T is a tree decomposition of G, we have that $1 \in \chi(t)$. Therefore, given that $|\chi(t)| \leq 2$, we conclude that $\chi(t) = \chi(s_1) = \{1, 2\}$, which again contradicts one of our initial assumptions.

3. First, assume that G is acyclic. Notice that if G is the empty graph, then $\text{tw}(G) = -1$, and if G consists only of isolated nodes, then $\text{tw}(G) = 0$. Hence, assume that G is the disjoint union of some trees containing at least one edge. Then it is straightforward to generalize the construction for graph G_2 in Example 21.2 to show that $\text{tw}(G) = 1$.
   Second, assume that G is not acyclic. Then G has as a subgraph a cycle G' with at least three nodes. By Part 2 of this proposition, we have that $\text{tw}(G') = 2$. Hence, we conclude that $\text{tw}(G) \geq 2$, given that $\text{tw}(G) \geq \text{tw}(G')$ by Lemma 21.4.

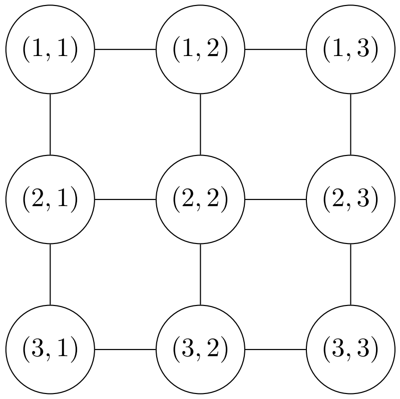
4. If G is a clique with n nodes and $n \in [0, 3]$, then it has already been shown that $\text{tw}(G) = n-1$ in the previous parts of this proposition. Hence, assume that $G = (V, E)$, where $V = \{1, \dots, n\}$, $n \geq 4$ and $E = \{\, \{i, j\} \mid i, j \in [1, n] \text{ and } i \neq j \,\}$. For the sake of contradiction, assume that $\text{tw}(G) < n-1$. Then there exists a tree decomposition (T, χ) of G such that $T = (V_T, E_T)$ and $|\chi(s)| \leq n - 1$ for every $s \in V_T$. Moreover, assume that for every

$(t, u) \in E_T$, it holds that $\chi(t) \neq \chi(u)$ (if this is not the case, then u can be removed, and the children of u can become children of t).
Let $s_1 \in V_T$ such that $\chi(s_1)$ is maximal in the sense that there is no $s \in V_T$ such that $\chi(s_1) \subsetneq \chi(s)$. Given that $|\chi(s_1)| \leq n - 1$, there exists $i \in [1, n]$ such that $i \notin \chi(s_1)$. Moreover, given that (T, χ) is a tree decomposition of G, there exists $s_2 \in V_T$ such that $i \in \chi(s_2)$. First, assume that s_2 is a descendant of s_1. Then given that $s_1 \neq s_2$, there exists an edge $(s_1, u) \in E_T$ such that $s_2 \in V_u$, where $V_u = \{v \in V_T \mid v \text{ is a descendant of } u \text{ in } T\}$. By Lemma 21.5, we know that $\chi(s_1) \cap \chi(u)$ separates $\chi(V_u)$ from $\chi(V_T - V_u)$ in G. Let $j \in \chi(s_1)$. Then given that $j \in \chi(V_T - V_u)$, $i \in \chi(V_u)$, j, i is an undirected path from j to i in G and $i \notin \chi(s_1)$, we have that $j \in \chi(s_1) \cap \chi(u)$. Hence, $\chi(s_1) \subseteq \chi(u)$, from which we conclude that $\chi(s_1) = \chi(u)$ by maximality of $\chi(s_1)$. But this contradicts one of our initial assumptions. The second case of this proof, where s_2 is not a descendant of s_1, is left as an exercise for the reader. □

As a final example, we consider the case of grids. Given $k \geq 1$, the $(k \times k)$-grid is defined as the undirected graph:

$$G_{k \times k} = ([k] \times [k], \{\{(i, j), (i', j')\} \mid |i - i'| + |j - j'| = 1\}.$$

For example, the (3×3)-grid is depicted in the following figure:



In the following proposition, we show that grids have unbounded treewidth.

Proposition 21.7

$\text{tw}(G_{k \times k}) = k$ for every $k \geq 1$.

Proof. Notice that we have already proved in Proposition 21.6 that $\text{tw}(G_{1 \times 1}) = 1$ and $\text{tw}(G_{2 \times 2}) = 2$, so consider $k \geq 3$.

We start by showing that $\text{tw}(G_{k\times k}) \leq k$. Let $T = (V_T, E_T)$ be a tree defined as:

$$\begin{aligned}
&V_T = \{v_{i,j} \mid i \in [k-1] \text{ and } j \in [k]\},\\
&E_T = \{(v_{i,j}, v_{i,j+1}) \mid i \in [k-1] \text{ and } j \in [k-1]\} \cup \{(v_{i,k}, v_{i+1,1}) \mid i \in [k-2]\}.
\end{aligned}$$

Moreover, let χ be the following function that assigns a subset of $[k] \times [k]$ to each node $v_{i,j}$ of T:

$$\chi(v_{i,j}) = \{(i,j), \ldots, (i,k), (i+1,1), \ldots, (i+1,j)\}$$

In what follows, we prove that (T, χ) is a tree decomposition of $G_{k\times k}$, from which we conclude that $\text{tw}(G_{k\times k}) \leq k$ since the width of (T, χ) is k (in fact, $|\chi(v_{i,j})| = k+1$ for every node $v_{i,j} \in V_T$). But before doing this proof, we provide the tree decomposition obtained for the grid $G_{3\times 3}$:

$$\begin{array}{c}
\{(1,1),(1,2),(1,3),(2,1)\}\\
|\\
\{(1,2),(1,3),(2,1),(2,2)\}\\
|\\
\{(1,3),(2,1),(2,2),(2,3)\}\\
|\\
\{(2,1),(2,2),(2,3),(3,1)\}\\
|\\
\{(2,2),(2,3),(3,1),(3,2)\}\\
|\\
\{(2,3),(3,1),(3,2),(3,3)\}
\end{array}$$

In particular, this is a tree decomposition of $G_{3\times 3}$ as $\chi^{-1}((i,j))$ induces a subtree of T for each node (i,j) of $G_{3\times 3}$. For example, the following is the subtree of T induced by $\chi^{-1}((2,2))$:

$$\begin{array}{c}
\{(1,2),(1,3),(2,1),(2,2)\}\\
|\\
\{(1,3),(2,1),(2,2),(2,3)\}\\
|\\
\{(2,1),(2,2),(2,3),(3,1)\}\\
|\\
\{(2,2),(2,3),(3,1),(3,2)\}
\end{array}$$

Let us consider now the general definition of T. For every edge $\{(i,j),(i,j+1)\}$ of $G_{k\times k}$ such that $i \in [k-1]$, it holds that $\{(i,j),(i,j+1)\} \subseteq \chi((i,j))$, and for every edge $\{(k,j),(k,j+1)\}$ of $G_{k\times k}$, it holds that $\{(k,j),(k,j+1)\} \subseteq$

$\chi((k-1,k))$. Moreover, for every edge $\{(i,j),(i+1,j)\}$ of $G_{k\times k}$, it holds that $\{(i,j),(i+1,j)\} \subseteq \chi((i,j))$. From this reasoning, it is also possible to conclude that $\chi^{-1}((i,j))$ is not empty for every node (i,j) of $G_{k\times k}$, so it only remains to prove that $\chi^{-1}((i,j))$ induces a (connected) subtree of T to show that (T,χ) is a tree decomposition of $G_{k\times k}$. This latter condition can be easily proved by noticing that for every $i \in [2,k-1]$ and $j \in [k]$:

$$\begin{aligned}\chi^{-1}((1,j)) &= \{v_{1,1},\ldots,v_{1,j}\},\\ \chi^{-1}((i,j)) &= \{v_{i-1,j},\ldots,v_{i,j}\},\\ \chi^{-1}((k,j)) &= \{v_{k-1,j},\ldots,v_{k-1,k}\},\end{aligned}$$

and by considering the definition of the edge relation E_T of T. This concludes the proof that $\mathrm{tw}(G_{k\times k}) \leq k$.

To complete the proof of the proposition, we need to show that $\mathrm{tw}(G_{k\times k}) \geq k$. In what follows, we show that $\mathrm{tw}(G_{k\times k}) \geq k-1$, and we leave as an exercise for the reader to prove the tight upper bound $\mathrm{tw}(G_{k\times k}) \geq k$.

To prove that $\mathrm{tw}(G_{k\times k}) \geq k-1$, we need to introduce some terminology and prove a technical lemma. Let $G=(V,E)$ be a graph and $W \subseteq V$. Given $S \subseteq V$, define $(G-S)$ as the subgraph of G induced by the set of nodes $V-S$. Moreover, S is said to be a balanced W-separator if every connected component of $(G-S)$ contains at most $|W|/2$ elements of W.

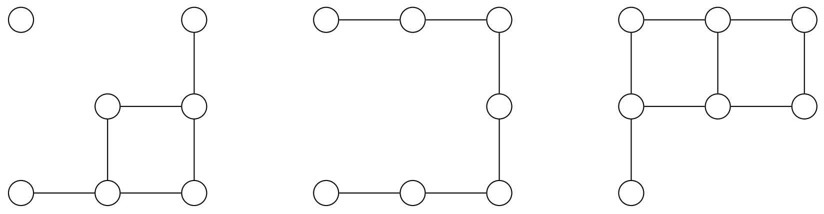
Lemma 21.8. *Let $G=(V,E)$ be a graph such that $tw(G) \leq k$ and $W \subseteq V$. Then there exists $S \subseteq V$ such that S is a balanced W-separator and $|S| \leq k+1$.*

Proof. Given that $\mathrm{tw}(G) \leq k$, there exists a tree decomposition (T,χ) of G such that $T=(V_T,E_T)$ and the width of (T,χ) is at most k. Given a node u of T, recall from Lemma 21.5 the notation $V_u = \{v \in V_T \mid v \text{ is a descendant of } u \text{ in } T\}$. Then let u_0 be a node of T such that: $\chi(V_{u_0})$ contains more than $|W|/2$ elements of W, and $\chi(V_u)$ contains at most $|W|/2$ elements of W for every child u of u_0. Notice that such a node u_0 exists since $\chi(V_{root})$ contains more than $|W|/2$ elements of W, where *root* is the root of T, given that $W \subseteq V$ and $V = \chi(V_{root})$.

Next we prove that $\chi(u_0)$ is a balanced W-separator, from which we conclude the lemma holds since $|\chi(u_0)| \leq k+1$ (given that the width of (T,χ) is at most k). Assume that $u_1, \ldots, u_\ell$ are the children of u_0 in T, and consider a connected component C of $(G-\chi(u_0))$. First, assume that $C \cap \chi(V_{u_i}) \neq \emptyset$ for some $i \in [\ell]$. For the sake of contradiction, assume that $C \not\subseteq \chi(V_{u_i})$, and let v_1, v_2 be nodes of C such that $v_1 \notin \chi(V_{u_i})$ and $v_2 \in \chi(V_{u_i})$. Given that C is a connected component of $(G-\chi(u_0))$, there exists an undirected path π from v_1 to v_2 in $(G-\chi(u_0))$. By Lemma 21.5, we know that $\chi(u_0) \cap \chi(u_i)$ separates $\chi(V_T - V_{u_i})$ from $\chi(V_{u_i})$ in G. Hence, given that $v_1 \in \chi(V_T - V_{u_i})$ and $v_2 \in \chi(V_{u_i})$, a node of π occurs in $\chi(u_0) \cap \chi(u_i)$. But this implies that π is not a path in $(G-\chi(u_0))$, which contradicts our initial assumption. We conclude that $C \subseteq \chi(V_{u_i})$ and, thus, C contains at most $|W|/2$ elements of W as $\chi(V_{u_i})$ contains at most $|W|/2$ elements of W by definition of u_0. Second, assume that

$C \cap \chi(V_{u_i}) = \emptyset$ for every $i \in [\ell]$. Then we have that $C \cap [\bigcup_{i\in[\ell]} \chi(V_{u_i})] = \emptyset$. Moreover, we know that $C \cap \chi(u_0) = \emptyset$ since C is a connected component of $(G - \chi(u_0))$. Hence, given that $\chi(V_{u_0}) = \chi(u_0) \cup [\bigcup_{i\in[\ell]} \chi(V_{u_i})]$, we have that $C \cap \chi(V_{u_0}) = \emptyset$ and, thus, C contains at most $|W|/2$ elements of W as $\chi(V_{u_0})$ contains more than $|W|/2$ elements of W. This concludes the proof of the lemma.

To see how Lemma 21.8 is used to prove that $\mathrm{tw}(G_{k\times k}) \geq k-1$, first consider $k = 3$. For the sake of contradiction, suppose it is not the case that $\mathrm{tw}(G_{3\times 3}) \geq 2$, so that $\mathrm{tw}(G_{3\times 3}) \leq 1$. Then by considering $W = [3] \times [3]$ in Lemma 21.8, we know that there exists $S \subseteq [3] \times [3]$ such that $|S| \leq 2$ and S is a balanced $([3] \times [3])$-separator, so that every connected component of $(G_{3\times 3} - S)$ has at most 4 elements. If $|S| = 1$, then $(G_{3\times 3} - S)$ has one connected component with 8 elements, so the previous condition does not hold. Hence, we are left with the possibility that $|S| = 2$. We depict in the following figure the result of removing some of such sets S from $G_{3\times 3}$:



In all the above cases, there are connected components with more than 4 elements, so the sets S in these examples are not balanced $([3]\times[3])$-separators. By simple inspecting the remaining alternatives for S, it is easy to conclude that each graph resulting by removing two nodes from $G_{3\times 3}$ has a connected component with more than 4 elements. This leads to a contradiction to the fact that S is a balanced $([3] \times [3])$-separator, and to a contradiction to our initial assumption that $\mathrm{tw}(G_{3\times 3}) \leq 1$.

In general, if we assume that $\mathrm{tw}(G_{k\times k}) \leq k-2$, then by considering $W = [k] \times [k]$ in Lemma 21.8, we know that there exists $S \subseteq [k] \times [k]$ such that $|S| \leq k-1$ and S is a balanced $([k]\times[k])$-separator. However, this leads to a contradiction, as it is possible to prove that for every $S \subseteq [k] \times [k]$ such that $|S| \leq k-1$, there exists a connected component C of $(G_{k\times k} - S)$ with more than $k^2/2$ elements. This last property is left as an exercise for the reader.

The Treewidth of a Conjunctive Query

To define the treewidth of a conjunctive query q, we need to consider a graph representation of the structure of the variables occurring in q. Formally, assume that q is the following CQ:

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

Then the *Gaifman graph* of q, denoted by $G_q = (V, E)$, is defined as the following undirected graph. The set of nodes V is the set of variables y such that y occurs in $\bar{u}_i$ for some $i \in [n]$, and y does not occur in $\bar{x}$. Moreover, for every pair of distinct variables y and z in V, it holds that $\{y, z\} \in E$ if and only if y, z occur in $\bar{u}_i$ for some $i \in [n]$. For example, if q_1 is the CQ

$$\text{Answer}(x, y) \ :\!-\ R(x, y, z), R(y, y, z), S(y, w, w'), T(z, z', w),$$

then $G_{q_1} = (V_1, E_1)$, where $V_1 = \{z, w, w, z'\}$ and $E_1 = \{\{w, w'\}, \{z, z'\}, \{z, w\}, \{z', w\}\}$.

Definition 21.9: Treewidth of a Conjunctive Query

The treewidth of a conjunctive query q, denoted by $\text{tw}(q)$, is defined as $\text{tw}(G_q)$.

For each fixed $k \geq 1$, we write $\mathsf{TW}(k)$ for the set of CQs whose treewidth is at most k.

Efficient Evaluation of Conjunctive Queries with Bounded Treewidth

To provide an efficient algorithm to evaluate a CQ in $\mathsf{TW}(k)$, for a fixed value $k \geq 1$, we first need an efficient procedure to construct a tree decomposition for such a query. To do this, we notice that there exists a linear-time algorithm that, given an undirected graph G with $\text{tw}(G) \leq k$, construct a tree decomposition of G of width at most k. Hence, given that G_q can be constructed from q in time $O(\|q\|^2)$, we obtain the following result.

Proposition 21.10: Construction of a tree decomposition

Fix $k \geq 1$. Then there exists an algorithm that, given a CQ q in $\mathsf{TW}(k)$, constructs a tree decomposition of G_q of width at most k in time $O(\|q\|^2)$.

By using this proposition, it is possible to prove the following.

Theorem 21.11

Fix $k \geq 1$. Then $\mathsf{TW}(k)$-$\mathsf{Evaluation}$ can be solved in time $O(\|D\|^{k+1} \cdot (\|q\| + \|\bar{a}\|)^4 \cdot (\log \|D\| + \log \|q\| + \log \|\bar{a}\|))$.

Proof. We start by considering a database D and a Boolean CQ q

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

such that q is in $\mathsf{TW}(k)$. Our goal is to provide a polynomial-time algorithm for computing $q(D)$. Let (T, χ) be a tree decomposition of G_q obtained by using the algorithm in Proposition 21.10. We know that the width of (T, χ) is at most k, and we assume that $T = (V_T, E_T)$.

Let $\widehat{q}$ be a Boolean CQ defined as follows from q and T. For every node $s \in V_T$, let R_s be a new relation name of arity $|\chi(s)|$, and $\bar{x}_s$ be a tuple of variables containing exactly the variables in $\chi(s)$ (in an arbitrary order). Then $\widehat{q}$ is the following Boolean CQ:

$$\text{Answer} \ \coloneq R_1(\bar{u}_1), \ldots, R_n(\bar{u}_n), \bigwedge_{s \in V_T} R_s(\bar{x}_s).$$

Moreover, let $\widehat{D}$ be a database defined as follows from D. For every $i \in [n]$, we have that $R_i^{\widehat{D}} = R_i^D$, and for every $s \in V_T$, we have that $R_s^{\widehat{D}} = \text{Dom}(D)^{|\chi(s)|}$. From the definitions of $\widehat{q}$ and $\widehat{D}$, it is clear that $q(D) = \widehat{q}(\widehat{D})$.

Next we show that $\widehat{q}$ is an acyclic conjunctive query (as defined in Chapter 18). Let $i \in [n]$ and X_i be the set of variables occurring in $\bar{u}_i$. Then there exists $s \in V_T$ such that $X_i \subseteq \chi(s)$. For the sake of contradiction, assume that this property does not hold, and let G_{X_i} be the subgraph of G_q induced by X_i. We note that G_{X_i} is a clique with $\ell = |X_i|$ nodes by definition of G_q. Let χ' be the restriction of function χ to X_i: $\chi'(s) = \chi(s) \cap X_i$ for every $s \in V_T$. It is straightforward to prove that (T, χ') is a tree decomposition of G_{X_i}. Moreover, given that $X_i \not\subseteq \chi(s)$ for every $s \in V_T$, we have that $|\chi'(s)| < \ell$ for every $s \in V_T$, and, thus, the width of (T, χ') is at most $\ell - 2$. But then we conclude that $\text{tw}(G_{X_i}) \leq \ell - 2$, which leads to a contradiction to Part 4 of Proposition 21.6 (given that G_{X_i} is a clique with ℓ nodes).

Let T' be a tree constructed from (T, χ) as follows. For every $i \in [n]$, let s_i be a node in T such that $X_i \subseteq \chi(s_i)$. Then for every $s \in V_T$, we have that $\chi(s)$ is a node of T', and if t is a child of s in T, then $\chi(t)$ is a child of $\chi(s)$ in T'. Moreover, for every $i \in [n]$, we have that X_i is a child of $\chi(s_i)$. By definition of T' and given that (T, χ) is a tree decomposition of G_q, we have that T' is a join tree of $\widehat{q}$ (see Chapter 18 for a definition of join tree), from which we conclude that $\widehat{q}$ is an acyclic CQ.

Given that $\widehat{q}$ is an acyclic CQ, we have by Theorem 19.3 that $\widehat{q}(\widehat{D})$ can be computed in time $O(\|\widehat{D}\| \cdot \log \|\widehat{D}\| \cdot \|\widehat{q}\|)$. Given that $\text{tw}(G_q) \leq k$, we have that $|\chi(s)| \leq k+1$ for every $s \in V_T$. Hence, we have that $\|\widehat{D}\|$ is $O(\|D\| + \|D\|^{k+1} \cdot \|T\|)$, and we conclude that $\|\widehat{D}\|$ is $O(\|D\|^{k+1} \cdot \|q\|^2)$. Moreover, we have that $\|\widehat{q}\|$ is $O(\|q\| + \|T\|)$ and, thus, $\|\widehat{q}\|$ is $O(\|q\|^2)$. Therefore, given that $q(D) = \widehat{q}(\widehat{D})$ and $\widehat{q}$ can be constructed in time $O(\|q\|^2)$, we conclude that $q(D)$ can be computed in time $O(\|q\|^2 + (\|D\|^{k+1} \cdot \|q\|^2) \cdot \|q\|^2 \cdot \log(\|D\|^{k+1} \cdot \|q\|^2))$ and, thus, in time $O(\|D\|^{k+1} \cdot \|q\|^4 \cdot (\log \|D\| + \log \|q\|))$. This concludes the proof for the case of Boolean CQs in $\mathsf{TW}(k)$.

Assume now that we are given a database D, a non-Boolean CQ q

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

such that q is in $\mathsf{TW}(k)$, 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 CQ by simultaneously replacing in $q(\bar{x})$ each free variable x_i in $\bar{x} = (x_1, \dots, x_k)$ by its corresponding value a_i in $\bar{a} = (a_1, \dots, a_k)$. We denote this Boolean CQ as $q_{\bar{a}}$. By definition of the notion of Gaifman graph, we have that $G_{q_{\bar{a}}} = G_q$. Hence, we have that $q_{\bar{a}}$ is in $\mathsf{TW}(k)$ and, therefore, we can compute $q_{\bar{a}}(D)$ in time $O(\|D\|^{k+1} \cdot \|q_{\bar{a}}\|^4 \cdot (\log \|D\| + \log \|q_{\bar{a}}\|))$ by the previous discussion. Given that $\|q_{\bar{a}}\|$ is $O(\|q\| + \|\bar{a}\|)$ and $\bar{a} \in q(D)$ if and only if $q_{\bar{a}}(D) = \texttt{true}$, we conclude that it can be verified whether $\bar{a} \in q(D)$ in time $O(\|D\|^{k+1} \cdot (\|q\| + \|\bar{a}\|)^4 \cdot (\log \|D\| + \log(\|q\| + \|\bar{a}\|)))$ and, thus, in time $O(\|D\|^{k+1} \cdot (\|q\| + \|\bar{a}\|)^4 \cdot (\log \|D\| + \log \|q\| + \log \|\bar{a}\|))$. This concludes the proof of the theorem.

22

Generalized Hypertreewidth

A significant number of real-world CQs are not acyclic, but are in some sense "nearly-acyclic". Bounded generalized hypertreewidth provides a natural formalization of the notion of near-acyclicity. Unlike treewidth, this notion extends acyclicity. It also retains several of the good properties of the latter. In particular, CQs of bounded generalized hypertreewidth can be evaluated in polynomial time. Importantly, most of the CQs found in real-world situations are of small generalized hypertreewidth, thus establishing the practical relevance of the concept.

The notion of generalized hypertreewidth

The definition of generalized hypertreewidth is based on the important notions of *tree decompositions* and *generalized hypertree decompositions*. We have already defined the notion of tree decomposition of a graph, but here we extend it in the expected way to hypergraphs. Let $H = (V, E)$ be a hypergraph. A *tree decomposition* of $H = (V, E)$ is a pair (T, χ), formed by a tree T and a mapping χ that assigns a subset of the nodes in V to each node $s \in T$, for which the following statements hold:

1. For each edge $e \in E$, there is a node $s \in T$ such that $e \subseteq \chi(s)$.
2. For each node $v \in V$, the set of nodes $s \in T$ for which v occurs in $\chi(s)$ is connected.

Definition 22.1: Generalized Hypertree Decomposition

A *generalized hypertree decomposition* of $H = (V, E)$ is a triple (T, χ, λ) such that:

1. (T, χ) is a tree decomposition of H.

2. λ is a mapping that assigns a subset of the hyperedges in E to each node $s \in T$.
3. For each node $s \in T$, it is the case that $\chi(s) \subseteq \bigcup_{e \in \lambda(s)} e$.

In other words, a generalized hypertree decomposition (T, χ, λ) of H extends the tree decomposition (T, χ) by *covering* each set $\chi(s)$ of nodes, for $s \in T$, with a set $\lambda(s)$ of hyperedges from H.

The *width* of a node s in the generalized hypertree decomposition (T, χ, λ) is the number of atoms in $\lambda(s)$. The width of (T, χ, λ) is the maximal width of the nodes of T. The *generalized hypertreewidth* of a hypergraph is the minimum width of its generalized hypertree decompositions.

The notion of generalized hypertreewidth of a CQ is defined as follows:

Definition 22.2: Generalized hypertreewidth

The generalized hypertreewidth of a CQ q corresponds to the generalized hypertreewidth of its associated hypergraph H_q.

For each fixed $k \geq 1$, we write $\mathsf{GHW}(k)$ for the set of CQs whose generalized hypertreewidth is at most k. That is, CQs in $\mathsf{GHW}(k)$ are those which admit generalized hypertree decompositions of the form (T, χ, λ) in which each set of variables $\chi(s)$, for $s \in T$, is covered by a set $\lambda(s)$ of at most k edges in H_q.

It is easy to prove that $\mathsf{GHW}(k) \subsetneq \mathsf{GHW}(k+1)$, for each $k \geq 1$. As an example, let us recall the CQ

$$q \;=\; \text{Answer}(x, y) \;\text{:-}\; R(x, y, z), S(y, z), S(y, w, w'), T(z, z'), T(z, w),$$

which was introduced in Chapter 18. The CQ q is in $\mathsf{GHW}(2)$. This is witnessed by the generalized hypertree decomposition (T, χ, λ) such that T consists of two nodes r and t for which $\chi(r) = \{x, y, z, w, w'\}$, $\lambda(r) = \{\{x, y, z\}, \{y, w, w'\}\}$, $\chi(t) = \{z, z'\}$, and $\lambda(t) = \{\{z, z'\}\}$. On the other hand, q is not in $\mathsf{GHW}(1)$. This is because CQs in $\mathsf{GHW}(1)$ are acyclic (see Proposition 22.3 below) and we know from Example 18.5 that q is not acyclic.

By slightly abusing notation, in the rest of the section we assume for simplicity that if q is a CQ and (T, λ, χ) is a generalized hypertree decomposition of H_q, then $\chi(s)$ corresponds to a set of atoms from q, for each node $s \in T$.

Bounded generalized hypertreewidth and acyclicity

The notion of bounded generalized hypetreewidth properly subsumes acyclicity. More in particular, acyclic CQs coincide with the CQs of generalized hypertreewidth one.

Proposition 22.3

GHW(1) is the class of acyclic CQs.

Proof. It is easy to see that each acyclic CQ is in GHW(1). In fact, let q be an acyclic CQ whose set of atoms is $A_q = \{R_1(\bar{u}_1), \dots, R_m(\bar{u}_m)\}$ and let T be an arbitrary join tree of q. Then T can be turned into a generalized hypertree decomposition of q of width one as follows. For each node $s \in T$ which is associated with the set X_i of variables that are mentioned in $\bar{u}_i$, for $1 \leq i \leq m$, we define $\chi(s) = X_i$ and $\lambda(s) = \{R_i(\bar{u}_i)\}$. On the other hand, assume that q is in GHW(1) and (T, χ, λ) is a generalized hypertree decomposition of q of width one. From (T, χ, λ) one can construct a join tree of q as follows. Each node $s \in T$ is associated with the set of variables mentioned in the single atom in $\lambda(s)$. If two such nodes end up being associated with the same set of variables, we simply delete one of them (provided that it is not the root). Clearly, then, each node $s \in T$ is associated with the set X_i of variables mentioned in some tuple $\bar{u}_i$, for $1 \leq i \leq m$, and no two distinct nodes are associated with the same X_i. Still, there might be several $1 \leq i \leq m$ such that X_i is associated with no node in T. However, by definition of generalized hypertree decomposition, for each such an $1 \leq i \leq m$ there must be a node $s \in T$ such that $X_i \subseteq \chi(s) \subseteq X_j$, assuming that s is associated with the set X_j of variables mentioned in $\bar{u}_j$, for $1 \leq j \leq m$. One can then create a new children s' of s which is associated with X_i. This construction yields a join tree of q. □

Tractable evaluation based on consistency

It is shown next that CQs of bounded generalized hypertreewidth can be evaluated in polynomial time by extending the consistency criterion for acyclic CQs developed in Proposition 19.4. For reasons explained in Chapter 18, we concentrate on Boolean CQs.

Let $q = \text{Answer}() \text{ :- } R_1(\bar{u}_1), \dots, R_m(\bar{u}_m)$ be a Boolean CQ. For each $S \subseteq \{1, \dots, m\}$, let us define $q_S(\bar{x}_S)$ to be the CQ whose set of atoms is $\{R_i(\bar{u}_i) \mid i \in S\}$ and $\bar{x}_S$ is a tuple that consists precisely of the variables mentioned in such atoms. Our consistency criterion establishes the following:

Proposition 22.4

Fix $k \geq 1$. Let $q = \text{Answer}() \text{ :- } R_1(\bar{u}_1), \dots, R_m(\bar{u}_m)$ be a Boolean CQ in GHW(k) and D a database. Then the following are equivalent:

1. $q(D) = \texttt{true}$.
2. For each $S \subseteq \{1, \dots, m\}$ with at most k elements, there is a nonempty set $\text{Cons}(q_S(D)) \subseteq q_S(D)$ such that, for each $S' \subseteq$

$\{1, \dots, m\}$ with at most k atoms it is the case that

$$\text{Cons}(q_S(D)) \;=\; \text{Cons}(q_S(D)) \ltimes \text{Cons}(q_{S'}(D)).$$

That is, each tuple in $\text{Cons}(q_S(D))$ is consistent with some tuple in $\text{Cons}(q_{S'}(D))$, for every $S, S' \subseteq \{1, \dots, m\}$ with at most k atoms.

Proof. Suppose first that $q(D) = \texttt{true}$; i.e., there is a homomorphism h from q to D. It is not hard to see then that one can choose $\text{Cons}(q_S(D))$ to be $\{h(\bar{x}_S)\}$, for each $S \subseteq \{1, \dots, m\}$ with at most k elements. Suppose, on the other hand, that for each $S \subseteq \{1, \dots, m\}$ with at most k elements a nonempty set $\text{Cons}(q_S(D))$ as described in (2) exists. Let (T, χ, λ) be an arbitrary rooted and directed generalized hypertree decomposition of q of width k. This implies that $\text{Cons}(q_{\lambda(t)}(D))$ is well-defined for each node $s \in T$, as $|\lambda(s)| \leq k$ by definition. One can then prove by induction the following for each node $s \in T$ (see Exercise 3.14): Let X_s be the set of variables mentioned in sets of the form $\chi(s')$, for s' a descendant of s in T (including s itself) and $S(X_s)$ the set of all atoms in q that only mention variables in X_s. Then $\text{Cons}(q_{\lambda(s)}(D)) \subseteq q'_{S(X_s)}(D)$, where $q'_{S(X_s)}$ is the CQ that has the same set of atoms as $q_{S(X_s)}$ but its tuple of free variables is $\bar{x}_{\lambda(s)}$. In particular, for the root r of T it is the case that $\text{Cons}(q_{\lambda(r)}(D)) \subseteq q'_{S(X_r)}(D)$. Therefore, since $\text{Cons}(q_{\lambda(r)}(D))$ is nonempty one can conclude that $q'_{S(X_r)}(D)$ is also nonempty. This implies that there is at least one homomorphism from $q'_{S(X_r)}$ to D. But the atoms of $q'_{S(X_r)}$ and q are the same by definition, and thus $q(D) = \texttt{true}$. □

As for acyclic CQs, this result allows us to construct a simple greatest-fixed point procedure that checks for the existence of sets $\text{Cons}(q_S(D))$ as described in item (2) of Proposition 22.4. The algorithm, called k-Consistency, is presented next:

Algorithm 9 k-Consistency(q, D)

Input: A Boolean CQ q :- $R_1(\bar{u}_1), \dots, R_m(\bar{u}_m)$ in $\mathsf{GHW}(k)$ and a database D.
Output: If $q(D) = \texttt{true}$, then $\text{Cons}(q_S(D)) \neq \emptyset$ for each $S \subseteq \{1, \dots, m\}$ with at most k elements. Otherwise, `fail`.

```
1: $\text{Cons}(q_S(D)) := q_S(D)$, for each $S \subseteq \{1, \dots, m\}$ with at most $k$ elements
2: while $\text{Cons}(q_S(D)) \neq \text{Cons}(q_S(D)) \ltimes \text{Cons}(q_{S'}(D))$ for $S, S' \subseteq \{1, \dots, m\}$ with
   at most $k$ elements do
3:     $\text{Cons}(q_S(D)) := \text{Cons}(q_S(D)) \ltimes \text{Cons}(q_{S'}(D))$
4: if $\text{Cons}(q_S(D)) = \emptyset$ for some $S \subseteq \{1, \dots, m\}$ with at most $k$ elements then
5:     fail
```

It is not hard to see that k-Consistency runs in polynomial time, for each fixed $k \geq 1$. In fact, a naïve analysis shows that k-Consistency(q, D) can be implemented in time $O(||D||^{2k} \cdot ||q||^{2k})$ based on the following observations:

1. For each $S \subseteq \{1, \ldots, m\}$ with at most k elements, the value of $q_S(D)$ can be computed in time $O(||D||^k)$. Therefore, the initialization step of the algorithm in which $\text{Cons}(q_S(D))$ is set to be $q_S(D)$, for each $S \subseteq \{1, \ldots, m\}$ with at most k elements, runs in time $O(||D||^k \cdot ||q||^k)$.
2. Each further step of the algorithm in which $\text{Cons}(q_S(D))$ is set to be $\text{Cons}(q_S(D)) \ltimes \text{Cons}(q_{S'}(D))$, for each $S, S' \subseteq \{1, \ldots, m\}$ with at most k elements, takes time $O(||D||^k \cdot ||q||^k)$. Since each such a step deletes at least one tuple from some $\text{Cons}(q_S(D))$, for an $S \subseteq \{1, \ldots, m\}$ with at most k elements, the maximum number of steps performed by the algorithm is bounded by $O(||D||^k \cdot ||q||^k)$.

The soundness and completeness of k-Consistency is established next:

Proposition 22.5

Let q be a Boolean CQ in $\mathsf{GHW}(k)$ and D a database. Then:

$$q(D) = \texttt{true} \quad \Longleftrightarrow \quad k\text{-Consistency}(q, D) \neq \texttt{fail}.$$

The proof of Proposition 22.5 is left as an exercise for the reader (see Exercise 3.15). As a corollary, one obtains the following:

Theorem 22.6

Fix $k \geq 1$. $\mathsf{GHW}(k)$-Evaluation can be solved in polynomial time $O(||D||^{2k} \cdot ||q||^{2k})$.

Bounded generalized hypertreewidth of the core

Recall that the Consistency procedure for evaluating acyclic CQs, presented in Chapter 18, continues being sound for the class of CQs whose core is acyclic. The k-Consistency algorithm presented above preserves this good behavior, this time with respect to the class of CQs whose core is in $\mathsf{GHW}(k)$:

Proposition 22.7

Fix $k \geq 1$. Let q be a Boolean CQ whose core is in $\mathsf{GHW}(k)$ and D a database. Then:

$$q(D) = \texttt{true} \quad \iff \quad k\text{-}\textsc{Consistency}(q, D) \neq \texttt{fail}.$$

The proof of this result is similar to the proof of Proposition 19.6. As a corollary, one then obtains the following:

Theorem 22.8

Fix $k \geq 1$. The evaluation problem for the class of CQs whose core is in $\mathsf{GHW}(k)$ can be solved in polynomial time.

Computing generalized hypertree decompositions for faster evaluation

CQs in $\mathsf{GHW}(k)$ can be evaluated in polynomial time, for each fixed $k \geq 1$, via the k-Consistency procedure. Such a procedure assumes that the input CQ q is in $\mathsf{GHW}(k)$, i.e., that a generalized hypertree decomposition of q of width k exists, but no such a decomposition is required to be computed. On the other hand, as we will see later, having access to a generalized hypertree decomposition of width k with good properties helps improving the cost of evaluation for CQs in $\mathsf{GHW}(k)$. This is in line with the case of acyclic CQs, for which we know that computing a join tree allows us to perform evaluation in linear time using Yannakakis's algorithm.

Having access to a generalized hypertree decomposition is not a problem for the case $k = 1$; in fact, $\mathsf{GHW}(1)$ corresponds to the class of acyclic CQs, and from an acyclic CQ we can always compute a join tree (or, equivalently, a generalized hypertree decomposition), in linear time. Unfortunately, for $k > 1$ this good property no longer holds as the following result shows:

Theorem 22.9

Fix $k > 1$. Assuming PTime $\neq$ NP, there is no polynomial time algorithm that given an input CQ q computes a generalized hypertree decomposition of width k whenever $q \in \mathsf{GHW}(k)$.

Proof. The proof relies on the following difficult result:

Proposition 22.10

Fix $k > 1$. The problem of checking if a given CQ is in $\mathsf{GHW}(k)$ is NP-complete.

In fact, assume for the sake of contradiction that for some $k > 1$ there is a polynomial time algorithm $\mathcal{A}$ that given an input CQ q computes a generalized hypertree decomposition of width k whenever $q \in \mathsf{GHW}(k)$. We show then that there is a polynomial time algorithm $\mathcal{A}'$ that checks whether a given CQ is in $\mathsf{GHW}(k)$, thus contradicting Proposition 22.10. Take an arbitrary CQ q and run $\mathcal{A}$ on q. The algorithm $\mathcal{A}'$ accepts iff $\mathcal{A}$ outputs a generalized hypertree decomposition of q of width k (the latter can be checked in polynomial time). It is easy to see, then, that $\mathcal{A}'$ accepts iff q is in $\mathsf{GHW}(k)$. □

Does this result completely rule out the possibility of using generalized hypertree decompositions for query evaluation? Not necessarily, for the following reasons. It can be proved that if q is in $\mathsf{GHW}(k)$, then there is a generalized hypertree decomposition of q of width k with at most n nodes, where n is the number of variables in q. Therefore, in order to check if $q \in \mathsf{GHW}(k)$, and, if so, compute a generalized hypertree decomposition of q of width k with at most n nodes, we can do the following: Iterate over all tuples of the form (T, χ, λ), where T is a tree with at most n nodes, χ is a mapping that assigns a subset of the variables in q to each node $s \in T$, and λ is a mapping that assigns a subset of at most k atoms from q to each node $s \in T$. Then check if any of them is a generalized hypertree decomposition of q. This takes time $2^{||q||^c}$, for some integer $c \geq 1$. While this algorithm exhibits an exponential behavior, it is not completely impractical as the problem corresponds to a static analysis task for which the input, the CQ q, is often small.

We can then establish the following:

Theorem 22.11

Fix $k \geq 1$. Then $\mathsf{GHW}(k)$-Evaluation can be solved in time $2^{||q||^c} + O(||D||^k \cdot ||q||)$, for some integer $c \geq 1$.

Proof. First compute in time $2^{||q||^c}$, for some integer $c \geq 1$, a generalized hypertree decomposition of q of width k with at most n nodes, where n is the number of variables in q. Recall that for each node $s \in T$ the CQ $q_{\lambda(s)}$ is defined as follows: the atoms of $q_{\lambda(s)}$ are precisely those in $\lambda(s)$ and the free variables of $q_{\lambda(s)}$ are all the variables that are mentioned in such atoms. Compute then the value of $q_{\lambda(s)}(D)$, for each $t \in T$. Each $q_{\lambda(s)}(D)$ can be computed in time $O(||D||^k)$, and thus computing them all takes time $O(||D||^k \cdot ||T||)$, which is $O(||D||^k \cdot ||q||)$. By mimicking Yanankakis's algorithm, we inductively compute the values $Q_{\lambda(s)}(D)$'s, for s a node in T, which are defined as follows:

- If s is a leaf of T, then $Q_{\lambda(s)}(D) = q_{\lambda(s)}(D)$.
- If s has children $s_1, \ldots, s_p$, then $Q_{\lambda(s)}(D) = \bigcap_{1 \leq i \leq p} q_{\lambda(s)}(D) \ltimes Q_{\lambda(s_i)}(D)$.

This takes time $O(||D||^k \cdot ||q||)$ since $||q_{\lambda(s)}||$ is $O(||D||^k)$, for each node $s \in T$, and T has at most n nodes. It can be proved then (see Exercise **??**) that for each node $s \in T$ we have that:

$$Q_{\lambda(s)}(D) \;=\; q'_{S(X_s)}(D),$$

where $q'_{S(X_s)}$ is as defined in the proof of Proposition 22.4. In particular, the atoms of $q'_{S(X_s)}$ are precisely the atoms of q that only mention variables that appear in the subtree of T rooted in s, and the free variables of $q'_{S(X_s)}$ are those that are mentioned in $\lambda(s)$. The algorithm then accepts if $Q_{\lambda(r)}(D) \neq \emptyset$, where r is the root of T. In fact, from the previous observation it is the case that $Q_{\lambda(r)}(D) = q'_{S(X_r)}(D)$. Moreover, clearly the atoms of $q'_{S(X_r)}$ and q are the same. It follows then that $Q_{\lambda(r)}(D) \neq \emptyset$ if and only if there is a homomorphism from q to D, i.e., $q(D) = \texttt{true}$. □

The bound for evaluation of CQs in $\mathsf{GHW}(k)$ obtained in Theorem 22.11 is better, in many practical situations, than the one offered by the consistency algorithm. In fact, from Theorem 22.11 we obtain that the problem can be solved in time $2^{||q||^c} + O(||D||^k \cdot ||q||)$. This is better than the $O(||D||^{2k} \cdot ||q||^{2k})$ obtained by applying the k-Consistency algorithm whenever $2^{||q||^c}$ is $O(||D||^{2k})$. But this is not uncommon, as often the database D is very large and the CQ q is orders of magnitude smaller.

23

The Necessity of Bounded Treewidth

If a class $\mathcal{C}$ of CQs has bounded generalized hypertreewidth, then the evaluation problem for $\mathcal{C}$ can be solved in polynomial time. Moreover, this positive behavior continues to hold even if $\mathcal{C}$ itself does not have bounded generalized hypertreewidth, but the class $\mathcal{C}_{\text{core}}$ of all cores of CQs in $\mathcal{C}$ does. More formally, it follows from Proposition 22.7 that if $\mathcal{C}_{\text{core}}$ satisfies that there is a $k \geq 1$ such that every CQ $q \in \mathcal{C}_{\text{core}}$ is in $\mathsf{GHW}(k)$, then the evaluation problem for $\mathcal{C}$ can be solved in polynomial time.

A crucial question at this stage is whether this notion exhausts the space of tractability for CQ evaluation. That is, whether for every class $\mathcal{C}$ of CQs the following are equivalent:

1. Evaluation for $\mathcal{C}$ can be solved in polynomial time.
2. $\mathcal{C}_{\text{core}}$ has bounded generalized hypertreewidth.

Perhaps not surprisingly, it can be shown that this is not the case in general; in fact, there are more general notions of bounded CQ-width, e.g., bounded *fractional hypertreewidth*, that lead to tractability of CQ evaluation and properly extend the notion of bounded generalized hypertreewidth.

On the other hand, it is shown in this section that there is one important scenario in which conditions 1 and 2 expressed above are equivalent (at least under widely-held complexity theoretical assumptions); namely, when the arity of the underlying schemas of the CQs in $\mathcal{C}$ is fixed in advance. This includes the important case in which all CQs in $\mathcal{C}$ come from the same schema. In other words, notions such as bounded fractional hypertreewidth, that ensure tractability of CQ evaluation, properly extend bounded generalized hypertreewidth only when schemas of unbounded arity are allowed.

The equivalence of conditions 1 and 2 over schemas of fixed arity is obtained by proving that if $\mathcal{C}$ is a class of CQs over bounded arity schemas such that $\mathcal{C}_{\text{core}}$ is not of bounded generalized hypertreewidth, then the evaluation problem for $\mathcal{C}$ is W[1]-complete. Thus, under the standard assumption that W[1]-complete problems are not tractable, one can conclude the evaluation problem for $\mathcal{C}$ cannot be solved in polynomial time. But not only that, under the assumption that FPT $\neq$ W[1] one also obtains a stronger result: if $\mathcal{C}$ is a class of CQs over fixed arity schemas, then the evaluation problem for $\mathcal{C}$ is tractable if and only if the evaluation problem for $\mathcal{C}$ is fixed-parameter tractable if and only if $\mathcal{C}_{\text{core}}$ has bounded generalized hypertreewidth. In other words, at least in this restricted scenario the notion of fixed parameter tractability does not add to the usual notion of tractability.

Fixed Arity Schemas and CQs of Bounded Treewidth

It is easy to see that the notion of bounded generalized hypertreewidth properly extends the notion of bounded treewidth. As an example, consider the class formed by all boolean CQs q_n, for $n \geq 3$, defined as follows:

$$q_n \;=\; \text{Answer}() \leftarrow \bigwedge_{1 \leq i,j \leq n} E(x_i, x_j), T_n(x_1, \ldots, x_n),$$

where $\bigwedge_{1 \leq i \leq j \leq n} E(x_i, x_j)$ is a shortening for the fact that all atoms of the form $E(x_i, x_j)$, for $1 \leq i, j \leq n$, are in q_n. Notice that $\{q_n \mid n \geq 3\} \subseteq \mathsf{GHW}(1)$ as every CQ q_n in $\mathcal{C}$ is acyclic: this is witnessed by the generalized hypertree decomposition that contains a single node labeled with all variables in $\{x_1, \ldots, x_n\}$ that is covered by the single atom $T_n(x_1, \ldots, x_n)$. On the other hand, it is the case that $q_{n+1} \notin \mathsf{TW}(n)$, for each $n > 1$. This follows directly from the fact that the treewidth of the $(n+1)$-clique is exactly n.

Notice, on the other hand, that there is no bound on the arity of the schemas over which the CQs in $\{q_n \mid n \geq 3\}$ are defined (as q_n contains the n-ary atom $T_n(x_1, \ldots, x_n)$). This is in fact necessary, since over fixed arity schemas the notions of bounded treewidth and bounded generalized hypertreewidth coincide. This is formally stated below:

Lemma 23.1. *Let q be a CQ defined over a schema all of whose relation symbols have arity at most c, for $c \geq 1$. Then for every $k \geq 1$:*

- $q \in \mathsf{GHW}(k)$ *implies* $q \in \mathsf{TW}(ck - 1)$.
- $q \in \mathsf{TW}(k)$ *implies* $q \in \mathsf{GHW}(k + 1)$.

In particular, if $\mathcal{C}$ is a class of CQs defined over fixed arity schemas, then $\mathcal{C}$ is of bounded treewidth iff $\mathcal{C}$ is of bounded generalized hypertreewidth.

The proof of this result is left as an easy exercise for the reader.

The Main Result

The main result of this section, which is the characterization of the tractable classes of CQs over fixed arity schemas in terms of the notion of bounded treewidth, is stated next. Only the version for boolean CQs is presented, but there is a simple extension of the result that characterizes tractability for arbitrary CQs (see Exercise **??**):

Theorem 23.2

Assume that FPT $\neq$ W[1]. Let $\mathcal{C}$ be a recursively enumerable class of boolean CQs such that there is a bound on the arity of the relation symbols that are mentioned by CQs in $\mathcal{C}$. Then the following are equivalent:

1. $\mathcal{C}$-Evaluation can be solved in polynomial time.
2. $\mathcal{C}$-Evaluation is FPT.
3. $\mathcal{C}_{\text{core}}$ has bounded treewidth.

The restriction on $\mathcal{C}$ to be recursively enumerable can be removed if one assumes a stronger complexity theoretical assumption; namely, that the *non-uniform* versions of FPT and W[1] are also different.

Overall Idea Behind the Proof of Theorem 23.2

The implication from (1) to (2) is straightforward. The implication from (3) to (1) follows from Proposition 22.7 and Lemma 23.1. The implication from (2) to (3) is proved next via the contrapositive. To do this, it is shown that if $\mathcal{C}_{\text{core}}$ does not have bounded treewidth, then the evaluation problem for $\mathcal{C}_{\text{core}}$ is W[1]-hard under fpt-reductions and, therefore, not in FPT based on the assumption that FPT $\neq$ W[1].

Let $\mathcal{C}$ be a recursively enumerable class of CQs over fixed arity schemas such that $\mathcal{C}_{\text{core}}$ is not of bounded treewidth. The proof constructs an ftp-reduction from the parameterized problem p-CLIQUE to p-$\mathcal{C}$-Evaluation. Recall that p-CLIQUE is the problem of given a simple graph $G = (V, E)$ and an integer $k \geq 1$, decide if G has a k-clique using k as a parameter. As mentioned in Chapter 14, the problem p-CLIQUE is W[1]-complete under fpt-reductions.

The construction makes use of a deep result in graph theory presented below. The $(n \times m)$-*grid* is the undirected graph whose vertices are the pairs (i, j), for $1 \leq i \leq n$ and $1 \leq j \leq m$, and whose set of edges is:

$$\big\{\{(i,j),(i+1,j)\} \mid 1 \leq i < n, 1 \leq j \leq m\big\} \quad \cup \\ \big\{\{(i,j),(i,j+1)\} \mid 1 \leq i \leq n, 1 \leq j < m\big\}.$$

It can be proved that the $(n \times n)$-grid has treewidth n, for each $n > 1$. A *minor* of a simple graph G is a graph H that can be obtained from a subgraph G' of G by contracting edges. Then:

Theorem 23.3: Excluded Grid Theorem

There is a function $t : \mathbb{N} \to \mathbb{N}$, such that for every $k \geq 1$ and simple graph G of treewidth at least $t(k)$ it is the case that G contains a $(k \times K)$-grid as a minor, for $K = \binom{k}{2}$.

Let $G = (V, E)$ and $H = (V', E')$ be simple graphs. A *minor map* from H to G is a mapping $\mu : V' \to 2^V$ that satisfies the following:

(M1) If n is a node of H, then $\mu(n)$ is a connected and nonempty subset of the nodes of G.
(M2) The sets of the form $\mu(n)$, for n a node of H, are pairwise disjoint.
(M3) For every edge $(n_1, n_2) \in E'$, there exist nodes $n_1' \in \mu(n_1)$ and $n_2' \in \mu(n_2)$ such that $(n_1', n_2') \in E$.

A minor map is said to be onto if, in addition, the sets of the form $\mu(n)$, for n a node of H, define a partition of V.

The following is left as an easy exercise for the reader (see Exercise **??**):

Lemma 23.4. *H is a minor of G if and only if there is a minor map μ from H to G. If, in addition, G is connected, then there exists an onto minor map μ from H to G.*

The reduction from p-CLIQUE to p-$\mathcal{C}$-Evaluation is explained next. Consider an input for p-CLIQUE given by the pair (G, k), where G is a simple graph and $k \geq 1$ is an integer. Since $\mathcal{C}_{\text{core}}$ is not of bounded treewidth, there is some CQ $q \in \mathcal{C}$ whose core q' has treewidth at least $t(k)$. This means that there is at least one connected component q^* of q' whose treewidth is at least $t(k)$. By Lemma 23.1 then, the Gaifman graph G_{q^*} of q^* has treewidth at least $t(k)$, and from the Excluded Grid Theorem it is the case that the $(k \times K)$-grid is a minor of G_{q^*}, where $K = \binom{k}{2}$. Notice that q^* is also a core. Moreover, since the $(k \times K)$-grid is a minor of G_{q^*} and G_{q^*} is connected, Lemma 23.4 implies that there exists an onto minor map μ from the $(k \times K)$-grid to G_{q^*}.

Based on G, q^*, and μ, the proof constructs a database $D = D(G, q^*, \mu)$ over the schema of q^* such that:

$$q^*(D) = \texttt{true} \iff G \text{ contains a } k\text{-clique}.$$

Let us assume that $q' \setminus q^*$ is the CQ that is obtained from q' by removing all atoms in the connected component q^*. Let us denote by D' the disjoint union of D and (the canonical database of) $q' \setminus q^*$. Notice that because q' is a core and q^* is connected, it must be the case that if h is a homomorphism from q' to D' then h maps q^* to D. Therefore:

$$q'(D') = \texttt{true} \iff q^*(D) = \texttt{true} \iff G \text{ contains a } k\text{-clique}.$$

Since q' is the core of q, one concludes that:

$$q(D') = \texttt{true} \iff G \text{ contains a } k\text{-clique.}$$

The construction of $D = D(G, q^*, \mu)$ and the proof that $q^*(D) = \texttt{true}$ if and only if G contains a k-clique are presented later. To conclude that the reduction is indeed an fpt-reduction, it is necessary to further establish the following facts:

1. There is a computable function $g : \mathbb{N} \to \mathbb{N}$ such that $\|q\| \leq g(k)$.
2. There is a computable function $f : \mathbb{N} \to \mathbb{N}$ such that the pair (D', q) can be constructed in time $f(k) \cdot \|G\|^{O(1)}$ from (G, k).

Condition 1 holds since q depends exclusively on k and $\mathcal{C}$ is recursively enumerable. To show that condition 2 also holds, it is sufficient to show the following:

3. There is a computable function $f : \mathbb{N} \to \mathbb{N}$ such that $D = D(G, q^*, \mu)$ can be constructed in time $f(k) \cdot \|G\|^{O(1)}$ from (G, k).

This is because D' is the disjoint union of D and $q' \setminus q^*$ and the latter is computable from q. Condition 3 is established during the construction of D.

The Construction of $D(G, q^*, \mu)$

Recall that the set $\{1, \ldots, n\}$ is denoted $[n]$, for each $n \geq 1$. Let (G, k) be an input to p-CLIQUE, and assume that $G = (V, E)$ and $K = \binom{k}{2}$. It is convenient to interchangeably interpret the columns of the $(k \times K)$-grid as elements of $[K]$ and unordered pairs of elements over $[k]$. For that, let us define an arbitrary bijection φ from the set $[K]$ to the set of all unordered pairs of elements over $[k]$. The notation $i \in \varphi(p)$, for $i \in [k]$ and $p \in [K]$, is just a shortening for the fact that the integer i is contained in the pair $\varphi(p)$.

As explained before, q^* is a CQ in $\mathcal{C}$ that satisfies the following: It is a core, it is connected, and the $(k \times K)$-grid is a minor of G_{q^*}. In addition, $\mu : [k] \times [K] \to 2^V$ is a minor map satisfying the aforementioned conditions M1, M2, and M3. The database $D = D(G, q', \mu)$ is then defined over the alphabet of q^* as follows:

The domain The domain of D consists of all tuples:

$$(v, e, i, p, x) \in \big(V \times E \times [k] \times [K] \times \mathrm{Dom}(q^*)\big),$$

such that the following holds: $v \in e \iff i \in \varphi(p)$ and, in addition, $x \in \mu(i, p)$.

The facts Let us define the projection $\Pi : D \rightarrow \mathrm{Dom}(q^*)$ such that $\Pi(v, e, i, p, x) = x$. One can assume that Π extends to tuples by defining it component-wise. Then for every fact of the form $R(\bar{x}) \in D_{q^*}$, it is the case that D contains all facts of the form $R(\bar{b})$ such that $\Pi(\bar{b}) = \bar{x}$ and the following conditions hold for any two elements $b = (v, e, i, p, x)$ and $b' = (v', e', i', p', x')$ in the domain of D that are mentioned in $\bar{b}$:

(C1) If $i = i'$ then $v = v'$.
(C2) If $p = p'$ then $e = e'$.

Before establishing the correctness of the construction, let us analyze the cost of computing $D = D(G, q^*, \mu)$. First of all, q^* and μ can be computed from q, which in turn depends only on k. Once they are computed, it is possible to construct D in time:

$$O(|V| \cdot |E| \cdot k \cdot K \cdot \|q^*\|)^r,$$

where r is the maximum arity of a relation symbol mentioned in q^*. But such a maximum arity is fixed by assumption, implying that there is a computable function $f : \mathbb{N} \rightarrow \mathbb{N}$ such that $D = D(G, q^*, \mu)$ can be constructed in time $f(k) \cdot \|G\|^{O(1)}$ from (G, k).

Correctness of the Construction

It is finally shown that $q^*(D) = \texttt{true}$ if and only if G contains a k-clique. This is proved in the two lemmas that follow:

Lemma 23.5. *If G contains a k-clique, then $q^*(D) = \texttt{true}$.*

Proof. Let $\{v_1, \dots, v_k\}$ be a set of vertices that defines a k-clique in G. For $p \in [K]$ with $\varphi(p) = \{i, j\}$, let us define e_p to be the edge $\{v_i, v_j\}$. Therefore, it is possible to define a mapping $h : q^* \rightarrow \mathrm{Dom}(D)$ in such a way that:

$$h(x) \;=\; (v_i, e_p, i, p, x),$$

where $i \in [k]$ and $p \in [K]$ are the elements that satisfy that $x \in \mu(i, p)$. In fact, $h(x)$ belongs to D as by definition it is the case that $v_i \in e_p \iff i \in \varphi(p)$.

Next it is shown that $h : q^* \rightarrow \mathrm{Dom}(D)$ is a homomorphism, thus implying $q^*(D) = \texttt{true}$. Consider a fact of the form $R(\bar{x})$ in q^*, where $\bar{x} = (x_1, \dots, x_r)$. Let $i_1, \dots, i_r$ and $p_1, \dots, p_r$ be such that $x_j \in \mu(i_j, p_j)$, for each $1 \leq j \leq r$. Then:

$$h(\bar{x}) \;=\; \big((v_{i_1}, e_{p_1}, i_1, p_1, x_1), \dots, (v_{i_r}, e_{p_r}, i_r, p_r, x_r)\big).$$

Conditions (C1) and (C2) described above are trivially satisfied, and thus $R(h(\bar{x})) \in D$. □

Lemma 23.6. *If $q^*(D) = \texttt{true}$, then G contains a k-clique.*

Proof. Since $q^*(D) = \texttt{true}$, there is a homomorphism $h : q^* \to D$. Notice that, by definition, Π is a homomorphism from D to q^*. Then $f = \Pi \circ h$ is a homomorphism from q^* to q^*, and thus it is also an isomorphism since q^* is a core. One can assume, without loss of generality, that f is the identity. If not, simply consider $h \circ f^{-1}$ as the homomorphism instead of h.

As $f = \Pi \circ h$ is the identity, for each element $x \in \mathrm{Dom}(q^*)$ such that $x \in \mu(i,p)$, for $i \in [k]$ and $p \in [K]$, it must be the case that

$$h(x) \;=\; (v_x, e_x, i, p, x),$$

for some $v_x \in V$ and $e_x \in E$ such that $v_x \in e_x \iff i \in \varphi(p)$. To prove the existence of a k-clique in G it is necessary to establish several properties of the $h(x)$'s, for $x \in \mathrm{Dom}(q^*)$, as stated in the following claims:

Claim 23.7. *For each $i \in [k]$, $p \in [K]$, and $x, x' \in \mu(i,p)$, it is the case that $v_x = v_{x'}$ and $e_x = e_{x'}$.*

Proof. Since $\mu(i,p)$ is connected in q^*, it suffices to prove the claim for x, x' such that there is an edge between x and x' in G_{q^*} Let $R(\bar{x})$ be a fact in q^* such that both x and x' are mentioned in $\bar{x}$. Then $R(h(\bar{x})) \in D$, and thus $v_x = v_{x'}$ and $e_x = e_{x'}$ since conditions (C1) and (C2) hold. □

Claim 23.8. *For each $i, i' \in [k]$, $p \in [K]$, $x \in \mu(i,p)$, and $x' \in \mu(i',p)$, it is the case that $e_x = e_{x'}$.*

Proof. Let us assume without loss of generality that $i \leq i'$. If $i = i'$ then the result holds from Claim 23.7. Let us suppose then that $i < i'$. It is sufficient to establish the result for the case $i' = i + 1$, as all other cases follow easily by induction.

Since there is an edge between $\{i,p\}$ and $\{i+1,p\} = \{i',p\}$ in the $(k \times K)$-grid and μ is a minor map, there is an edge between an element $y \in \mu(i,p)$ and an element $y' \in \mu(i',p)$ in G_{q^*}. Thus, there is a fact $R(\bar{y})$ in q^* such that both y and y' are mentioned in $\bar{y}$. Let us assume without loss of generality that $\bar{y} = (y, y', \ldots)$. Since $R(h(\bar{y})) \in D$, it must be the case that $e_y = e_{y'}$ by condition (C2). But by Claim 23.7, it is the case that $e_y = e_x$ and $e_{y'} = e_{x'}$. This finishes the proof of the claim. □

Claim 23.9. *For each $i \in [k]$, $p, p' \in [K]$, $x \in \mu(i,p)$, and $x' \in \mu(i,p')$, it is the case that $v_x = v_{x'}$.*

Proof. Analogous to the proof of Claim 23.8 and left as an exercise for the reader. □

Summing up, the previous claims imply that there are vertices $v_1, \ldots, v_k$ and edges $e_1, \ldots, e_K$ in G satisfying the following:

- For each $i \in [k]$ and element $x \in \mathrm{Dom}(q^*)$ such that $x \in \mu(i,p)$, for some $p \in [K]$, it is the case that $h(x)$ is of the form (v_i, e, i, p, x).

- For each $p \in [K]$ and element $x \in \mathrm{Dom}(q^*)$ such that $x \in \mu(i,p)$, for some $i \in [k]$, it is the case that $h(x)$ is of the form (v, e_p, i, p, x).

Fix an arbitrary pair $\{i,j\}$ with $1 \leq i < j \leq k$ and consider the pair $p \in [K]$ such that $\varphi(p) = \{i,j\}$. It is possible to prove that $e_p = \{v_i, v_j\}$, and, therefore, that $\{v_1, \dots, v_k\}$ defines a k-clique in G. In fact, since μ is a minor map, there are elements $x \in \mu(i,p)$ and $x' \in \mu(j,p)$. From the previous remarks, $h(x) = (v_i, e_p, i, p, x)$ and $h(x') = (v_j, e_p, j, p, x)$. Now, since $v_i \in e_p \iff i \in \varphi(p)$ and $v_j \in e_p \iff j \in \varphi(p)$ by definition, we conclude that $e_p = \{v_i, v_j\}$. This finishes the proof of Lemma 23.6.

25

Bounding the Join Size

So far, the search for efficient CQ evaluation methods has focused on the so-called *structural approach*. The underlying idea is to exploit structural properties of the input CQs, such as bounded generalized hypertreewidth, to develop tractable evaluation algorithms. This approach, however, disregards quantitative aspects such as the cardinalities of different relations in the database, which play a fundamental role in query evaluation.

The goal of this chapter is to present a result that brings together both structural and quantitative aspects. It does so by providing a tight bound on the size of the answer $q(D)$ of a CQ q over a database D in terms of the cardinalities of the relations in D and a sophisticated notion of width for q based on *fractional covers*. As we will see in the next chapter, this result also provides the tools for developing worst-case optimal evaluation algorithms for CQs.

Join Queries

We consider in this chapter the named perspective for relational algebra, as defined in Chapter 4. Recall that, under this perspective, the join $R_1 \bowtie R_2$ of relations $R_1[U_1]$ and $R_2[U_2]$ is the set of tuples t of sort $U_1 \cup U_2$ such that there exist $t_1 \in R_1$ and $t_2 \in R_2$ with $t(A) = t_1(A)$ for every $A \in U_1$ and $t(A) = t_2(A)$ for every $A \in U_2$. That is, if A is a common attribute name in R_1 and R_2, then t_1 and t_2 have the same value for their A-attribute.

For the sake of presentation, this chapter only deals with CQs that represent *join queries*, which we define next.

Definition 25.1: Join Query

A *join query* over a named schema $\mathbf{S}$ is a named RA query over $\mathbf{S}$ of the form

$$R_1 \bowtie \cdots \bowtie R_n \,.$$

We will assume throughout the chapter that $R_1, \dots, R_n$ are pairwise different. Notice that this is possible because the join operator $\bowtie$ is commutative, associative, and idempotent (that is, $(R \bowtie R)(D) = R(D)$ for every database D). Join queries therefore correspond to CQs of the form $\text{Answer}(\bar{x}) \coloneq R_1(\bar{y}_1), \dots, R_n(\bar{y}_n)$ where

(1) each relation name R_i with $i \in [n]$ occurs exactly once;

(2) for every $i \in [n]$, no variables are repeated in the tuple $\bar{y}_i$; and

(3) every variable that appears in some tuple $\bar{y}_i$ for $i \in [n]$ also appears in $\bar{x}$.

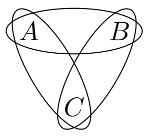
Condition (3) states that join queries do not have bound variables, i.e., they are projection free. For several of the results presented next this restriction is not essential, and in fact such results continue to hold for arbitrary CQs at the cost of more complicated proofs.

A Gentle Introduction

As a gentle introduction, consider the join query

$$q_\triangle = R[A, B] \bowtie S[B, C] \bowtie T[C, A] ,$$

and the hypergraph



that visualizes the structure of $q_\triangle$. For simplicity, let us assume that we evaluate $q_\triangle$ on a database D where R, S, and T have the same number n of tuples. How many tuples can there be in $q_\triangle(D)$?

Trivially, $|q_\triangle(D)|$ is at most n^3, because $|q_\triangle(D)|$ is at most as large as the size $|R \times S \times T|$ of the Cartesian product of R, S, and T. But we can also see that $|q_\triangle(D)|$ is at most $n^2 = |R \times S|$. This is because we can obtain $q_\triangle(D)$ as follows. We first compute $R \times S$ and select those tuples in $R \times S$ that agree on the B-attribute. Finally, we observe that joining the result of the previous step with T can only remove further tuples: it selects those tuples that agree with some tuple in T on their A- and C-attribute.

This idea can be generalized when we consider *edge covers* of the hypergraph of a join query q of the form $R_1 \bowtie \cdots \bowtie R_n$. Let $\{A_1, \dots, A_m\}$ be the set of attributes used by q, i.e., $\cup_{i \in [n]} \mathbf{S}(R_i)$. We define the hypergraph $H_q = (V_q, E_q)$ of q to consist of the set of nodes $V_q = \{A_1, \dots, A_m\}$ and hyperedges $E_q = \{\mathbf{S}(R_i) \mid i \in [n]\}$.

Definition 25.2: Edge Cover

An *edge cover* of a hypergraph $H = (V, E)$ is a subset $E' \subseteq E$ of its edges such that, for each node $v \in V$, there is an edge $e \in E'$ with $v \in e$.

The intuition of edge dovers is that the set of selected edges in E' "cover" each node of H. For example, the hyperedges $\{A, B\}$ and $\{B, C\}$ (corresponding to $R[A, B]$ iand $S[B, C]$ in $q_\triangle$) are an edge cover of $H_{q_\triangle}$. The principle that we used to give the n^2 upper bound for $|q_\triangle(D)|$ can be generalized as follows.

Theorem 25.3

Let $q = R_1 \bowtie \cdots \bowtie R_n$ be a join query and let E be an edge cover of its hypergraph H_q. Then, for every database D, we have that

$$|q(D)| \leq \prod_{\mathbf{S}(R_i) \in E} |R_i^D| \qquad \in O(|D|^{|E|}) \; .$$

We will prove a generalization of this result as Theorem 25.8.

Whereas Theorem 25.3 gives an upper bound for $|q(D)|$, what can we say about lower bounds? Consider again the query $q_\triangle$. We can see that, for every $k \in \mathbb{N}$, the database D_k, in which $R[A, B] = \{(A : i, B : 1) \mid i \in [k]\}$, $S[B, C] = \{(B : 1, C : 1)\}$, and $T[C, A] = \{(C : 1, A : i) \mid i \in [k]\}$ returns k answers to q. Indeed, we have that $q(D_k) = \{(A : i, B : 1, C : 1) \mid i \in [k]\}$.

Interestingly, also this idea can be generalized using a standard graph-theoretical notion, namely *independent sets*.

Definition 25.4: Independent Set

An *independent set* of a hypergraph $H = (V, E)$ is a subset $V' \subseteq V$ of its nodes such that, for each edge $e \in E$, it holds that $|e \cap V'| \leq 1$.

The intuition behind independent sets is that all distinct node pairs in V' are "independent", i.e., not connected by an edge. For example, the node A is an independent set of $H_{q_\triangle}$ which, incidentally, we used to construct the databases D_k. Similar to before, we can also generalize this idea.

Proposition 25.5

Let $q = R_1 \bowtie \cdots \bowtie R_n$ be a join query and let V be an independent set of its hypergraph H_q. Then, for every $k \in \mathbb{N}$, there is a database D with

$$k^{|V|} \leq |q(D)|$$

and every relation in D has at most k tuples.

Again, we will prove a generalization of this result as Proposition 25.20.

The AGM Bound

We now embark on generalizing the results from the gentle introduction and start with focusing on upper bounds. We consider the following generalization of the edge covers from Definition 25.2.

Definition 25.6: Fractional Edge Cover

A *fractional edge cover* of a hypergraph $H = (V, E)$ is a function

$$f : E \to \mathbb{Q}_{\geq 0}$$

such that, for each node $v \in V$, it holds that $\sum_{v \in e} f(e) \geq 1$. The *weight* of f is the value $\sum_{e \in E} f(e)$.

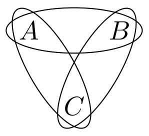
It is easy to see that each edge cover is indeed also a fractional edge cover. Let us illustrate the difference between edge covers and their fractional variant on an example.

Example 25.7

Consider again the join query

$$q_{\triangle} \ = \ R[A, B] \bowtie S[B, C] \bowtie T[C, A] \ ,$$

and its hypergraph $H_{q_{\triangle}}$



By slight abuse of notation, let us denote the hyperege $\{A, B\}$ as R, the hyperedge $\{B, C\}$ as S, and $\{C, A\}$ as T. We can then write the conditions for a fractional edge cover f of $H_{q_{\triangle}}$ as the following system of equations.

$$\begin{aligned} &f(R), f(S), f(T) \ \geq \ 0, \\ &f(R) + f(T) \ \geq \ 1, \quad f(R) + f(S) \ \geq \ 1, \quad f(S) + f(T) \ \geq \ 1 \ . \end{aligned}$$

The equation $f(R) + f(T) \geq 1$, for example, is obtained by considering attribute A and the fact that R and T are the edges in $H_{q_\triangle}$ that contain A. One solution to this system of equations, i.e., a fractional edge cover, is $f(R) = f(S) = f(T) = 1/2$. The weight of this fractional edge cover is $3/2$.

Recall that in a database D, we denote by R^D the relation instance associated to relation name R, that is, R^D is the set of tuples $\bar{a}$ such that $R(\bar{a}) \in D$. With this notation, we can state the AGM bound, which received its name from the last names of Albert Atserias, Martin Grohe, and Dániel Marx.

Theorem 25.8: AGM Bound

Consider a join query $q = R_1 \bowtie \cdots \bowtie R_n$ over schema $\mathbf{S}$ and a fractional edge cover f of q. Then, for every database D, we have that

$$|q(D)| \;\leq\; \prod_{i=1}^{n} |R_i^D|^{f(\mathbf{S}(R_i))} .$$

Before proving the theorem, it is worth illustrating the application of the AGM bound on a specific query.

Example 25.9

Applying the AGM bound to the fractional cover of Example 25.7, we obtain that, over every database D,

$$|q(D)| \;\leq\; \sqrt{|R^D| \cdot |S^D| \cdot |T^D|} .$$

We could also consider another fractional cover, namely $f(R) = f(S) = 1$ and $f(T) = 0$. Applying the AGM bound to this solution implies that

$$|q(D)| \;\leq\; |R^D| \cdot |S^D| .$$

Which bound is better depends on the underlying database D.

- Consider first the case when $R^D = S^D = T^D$, i.e., when the interpretations of all three relations over D coincide. Moreover, assume that R^D does not contain any tuple of the form (a, a). Then D can be naturally seen as a graph without loops and with $M = |R^D|$ edges, and, thus, $|q(D)|$ is equal to three times the number of directed triangles in such a graph. (Notice that each directed triangle is counted three times as a tuple $(A\colon a, B\colon b, C\colon c)$ is considered to be different from a tuple $(A\colon b, B\colon c, C\colon a)$). In this case, the first bound is bet-

ter, as it implies that $|q(D)| \leq M^{3/2}$, while the second one implies that $|q(D)| \leq M^2$.
The fact that the maximum number of directed triangles in a graph with M edges is bounded by

$$\frac{M^{3/2}}{3}$$

is non-trivial, which illustrates the power of Theorem 25.8 as a tool for obtaining meaningful bounds on the size of the evaluation of a join query.

- Consider now the case when $|R^D| = |S^D| = 1$ and $|T^D| = M$. Then the second bound is tighter, as it implies that $|q(D)| = 1$ while the first one implies that $|q(D)| \leq \sqrt{M}$.

Proof of the AGM Bound

We will now prove the AGM bound. The proof makes use of a key *query decomposition lemma*, which is in turn proved by applying the following version of Hölder's inequality, which we state without proof.

Theorem 25.10: Hölder's Inequality

Let p, r be positive integers, $y_1, \ldots, y_r$ be non-negative real numbers such that $y_1 + \cdots + y_r \geq 1$, and $a_{i,j}$ be a non-negative real number for every $i \in [p]$ and $j \in [r]$. Then it holds that

$$\sum_{i=1}^{p} \prod_{j=1}^{r} a_{i,j}^{y_j} \leq \prod_{j=1}^{r} \left(\sum_{i=1}^{p} a_{i,j} \right)^{y_j}.$$

From now on, fix a join query $q = R_1 \bowtie \cdots \bowtie R_n$ over schema $\mathbf{S}$, and assume that $X = \{A_1, \ldots, A_m\}$ is the set of attributes occurring in q. Given $Y \subseteq X$, define $\mathcal{R}(Y)$ as $\bigcup_{A_j \in Y} \mathcal{R}(A_j)$. That is, $\mathcal{R}(Y)$ is the set of those R_i's that mention some attribute in Y, for $i \in [n]$. Moreover, assuming that $\mathcal{R}(Y) = \{R_{i_1}, \ldots, R_{i_\ell}\}$, where $1 \leq i_1 < \cdots < i_\ell \leq n$, define query

$$q_Y = \pi_{Y \cap \mathbf{S}(R_{i_1})}(R_{i_1}) \bowtie \cdots \bowtie \pi_{Y \cap \mathbf{S}(R_{i_\ell})}(R_{i_\ell}). \tag{25.1}$$

Recall that $\mathbf{S}(R)$ is the set of attributes associated to relation name R under schema $\mathbf{S}$. Thus, query q_Y defines the join of the projection over Y of those R_i's that mention at least some attribute in Y.[1]

[1] Recall that, in order for a projection $\pi_\alpha(e)$ to be well defined, it is required that α is a subset of the sort U of e. This is why we intersect with $\mathbf{S}(R_{i_j})$ in every projection.

Finally, to state the query decomposition lemma, recall the definition of the semijoin operator $\ltimes$ from Chapter 18. In particular, recall that two tuples $\bar{a}$ and $\bar{b}$ are said to be consistent if they have the same value in each shared attribute. Thus, the term $R \ltimes \{\bar{a}\}$ is used in the lemma to denote the set of tuples in R that are consistent with $\bar{a}$.

Lemma 25.11 (Query Decomposition Lemma). *Assume that f is a fractional edge cover for the hypergraph H_q of q and consider an arbitrary partition $\{Y, Z\}$ of X. Then for every database D it holds that*

$$\sum_{\bar{a} \in q_Y(D)} \prod_{R_i \in \mathcal{R}(Z)} |R_i^D \ltimes \{\bar{a}\}|^{f(\boldsymbol{S}(R_i))} \leq \prod_{i=1}^{n} |R_i^D|^{f(\boldsymbol{S}(R_i))}.$$

Here, we assume that all relations R_i^D are non-empty.

Proof. Let A be an arbitrary attribute in Y. Define $Y' = Y - \{A\}$ and $Z' = Z \cup \{A\}$. Next we show that

$$\sum_{\bar{a} \in q_Y(D)} \prod_{R_i \in \mathcal{R}(Z)} |R_i^D \ltimes \{\bar{a}\}|^{f(\mathbf{S}(R_i))} \leq \sum_{\bar{a}' \in q_{Y'}(D)} \prod_{R_i \in \mathcal{R}(Z')} |R_i^D \ltimes \{\bar{a}'\}|^{f(\mathbf{S}(R_i))}. \tag{25.2}$$

The lemma is then obtained by repeatedly applying (25.2) until Y' is empty, in which case the right-hand side of the inequality is precisely $\prod_{i=1}^{n} |R_i^D|^{f(\mathbf{S}(R_i))}$.

Each tuple $\bar{a} \in q_Y(D)$ can be decomposed as a pair $\bar{a}', v$ such that $\bar{a}' \in q_{Y'}(D)$ and v is the value of $\bar{a}$ for attribute A. In what follows, we use $(\bar{a}', v)$ as an alternative notation for tuple $\bar{a}$. Then the left-hand side of equation (25.2) can be expressed as

$$\sum_{\bar{a}' \in q_{Y'}(D)} \sum_{v \,:\, (\bar{a}', v) \in q_Y(D)} \prod_{R_i \in \mathcal{R}(Z)} |R_i^D \ltimes \{(\bar{a}', v)\}|^{f(\mathbf{S}(R_i))},$$

which in turn can be rewritten as

$$\sum_{\bar{a}' \in q_{Y'}(D)} \sum_{v \,:\, (\bar{a}', v) \in q_Y(D)} \Bigg(\Bigg(\prod_{R_i \in \mathcal{R}(Z)} |R_i^D \ltimes \{(\bar{a}', v)\}|^{f(\mathbf{S}(R_i))} \Bigg) \cdot \Bigg(\prod_{R_i \in \mathcal{R}(Z') - \mathcal{R}(Z)} 1^{f(\mathbf{S}(R_i))} \Bigg) \Bigg). \tag{25.3}$$

Notice that equation (25.3) is equivalent to

$$\sum_{\bar{a}' \in q_{Y'}(D)} \sum_{v \,:\, (\bar{a}', v) \in q_Y(D)} \prod_{R_i \in \mathcal{R}(Z')} |R_i^D \ltimes \{(\bar{a}', v)\}|^{f(\mathbf{S}(R_i))}, \tag{25.4}$$

since for those R_i in $\mathcal{R}(Z') - \mathcal{R}(Z)$, it holds that the set of attributes of R_i is contained in Y and, thus, $|R_i^D \ltimes \{(\bar{a}', v)\}| = 1$. Moreover, given that $R_i^D \ltimes \{(\bar{a}', v)\} = R_i^D \ltimes \{\bar{a}'\}$ for each $R_i \in \mathcal{R}(Z') - \mathcal{R}(A)$, equation (25.4) can be expressed as

$$\sum_{\bar{a}' \in q_{Y'}(D)} \prod_{R_i \in \mathcal{R}(Z') - \mathcal{R}(A)} |R_i^D \ltimes \{\bar{a}'\}|^{f(\mathbf{S}(R_i))} \sum_{v \,:\, (\bar{a}', v) \in q_Y(D)} \prod_{R_i \in \mathcal{R}(A)} |R_i^D \ltimes \{(\bar{a}', v)\}|^{f(\mathbf{S}(R_i))}. \quad (25.5)$$

This is the moment in which Hölder's inequality stated in Claim 25.10 is applied to obtain that the value expressed in equation (25.5) is bounded by

$$\sum_{\bar{a}' \in q_{Y'}(D)} \prod_{R_i \in \mathcal{R}(Z') - \mathcal{R}(A)} |R_i^D \ltimes \{\bar{a}'\}|^{f(\mathbf{S}(R_i))} \prod_{R_i \in \mathcal{R}(A)} \left(\sum_{v \,:\, (\bar{a}', v) \in q_Y(D)} |R_i^D \ltimes \{(\bar{a}', v)\}| \right)^{f(\mathbf{S}(R_i))}. \quad (25.6)$$

Observe that it is possible to apply Claim 25.10 because: (i) f is a fractional cover of q and, thus, $\sum_{R_i \in \mathcal{R}(A)} f(\mathbf{S}(R_i)) \geq 1$; and (ii) $|R_i^D \ltimes \{(\bar{a}', v)\}| \geq 0$ for every $R_i \in \mathcal{R}(A)$ and v such that $(\bar{a}', v) \in q_Y(D)$.

To conclude, the value expressed in Equation (25.6) is bounded by

$$\sum_{\bar{a}' \in q_{Y'}(D)} \prod_{R_i \in \mathcal{R}(Z') - \mathcal{R}(A)} |R_i^D \ltimes \{\bar{a}'\}|^{f(\mathbf{S}(R_i))} \prod_{R_i \in \mathcal{R}(A)} |R_i^D \ltimes \{\bar{a}'\}|^{f(\mathbf{S}(R_i))}. \quad (25.7)$$

This is because for every $R_i \in \mathcal{R}(A)$: (i) $R_i^D \ltimes \{(\bar{a}', v)\} \subseteq R_i^D \ltimes \{\bar{a}'\}$; and (ii) if $(\bar{a}', v_1)$ and $(\bar{a}', v_2)$ are in $q_Y(D)$, with $v_1 \neq v_2$, then $R_i^D \ltimes \{(\bar{a}', v_1)\}$ and $R_i^D \ltimes \{(\bar{a}', v_2)\}$ are disjoint given that $A \in \mathbf{S}(R_i)$. Finally, equation (25.7) can be simplified as

$$\sum_{\bar{a}' \in q_{Y'}(D)} \prod_{R_i \in \mathcal{R}(Z')} |R_i^D \ltimes \{\bar{a}'\}|^{f(\mathbf{S}(R_i))},$$

which finishes the proof of the lemma. □

We now move to the proof of Theorem 25.8, which is done by induction on the size of the set $X = \{A_1, \ldots, A_m\}$ of attributes occurring in q. Notice that this theorem trivially holds if $|R_i^D| = 0$ for some $i \in [n]$. Thus, we assume in the following proof that $|R_i^D| \geq 1$ for every $i \in [n]$.

- **Base case.** We have that $|X| = 1$, and hence each relation R_i is unary for each $\ell \in [n]$. Then we have that

$$\begin{aligned}|q(D)| &\leq \min_{\ell \in [n]} |R_\ell^D| \\ &\leq \big(\min_{\ell \in [n]} |R_\ell^D|\big)^{\sum_{i=1}^n f(\mathbf{S}(R_i))} \\ &= \prod_{i=1}^n \big(\min_{\ell \in [n]} |R_\ell^D|\big)^{f(\mathbf{S}(R_i))} \\ &\leq \prod_{i=1}^n |R_i^D|^{f(\mathbf{S}(R_i))} ,\end{aligned}$$

where the second inequality holds given that $\min_{\ell \in [n]} |R_\ell^D| \geq 1$ and that $\sum_{i=1}^n f(\mathbf{S}(R_i)) \geq 1$.

- **Inductive case.** We have that $|X| = m$ with $m > 1$. Let $Y = \{A_1, \ldots, A_{m-1}\}$. Let $\mathbf{S}'$ be the schema such that $\mathbf{S}'(R) = \{A_m\}$ for every $R \in \mathcal{R}(A_m)$. Let q' be the natural join query over $\mathbf{S}'$, that is,

$$q' = \Join_{R \in \mathbf{S}'} R \, .$$

Notice that q' is in fact the query $q_{\{A_m\}} = q_{X-Y}$ as in (25.1), which computes the intersection over all the unary relations R over schema $\mathbf{S}'$. For every tuple $\bar{a} \in q_Y(D)$, let $D_{\bar{a}}$ be the database over $\mathbf{S}'$ such that $R^{D_{\bar{a}}} = \pi_{\{A_m\}}(R^D \ltimes \{\bar{a}\})$ for each $R \in \mathcal{R}(A_m)$. Then we have that

$$q(D) = \bigcup_{\bar{a} \in q_Y(D)} \big(q'(D_{\bar{a}}) \times \{\bar{a}\}\big) \tag{25.8}$$

and, therefore,

$$|q(D)| \leq \sum_{\bar{a} \in q_Y(D)} |q'(D_{\bar{a}})| \, .$$

Since f satisfies $\sum_{A_m \in \mathbf{S}(R_i)} f(\mathbf{S}(R_i)) \geq 1$, we obtain by induction hypothesis for every $\bar{a} \in q_Y(D)$ that

$$|q'(D_{\bar{a}})| \leq \prod_{R_i \in \mathcal{R}(A_m)} |R_i^{D_{\bar{a}}}|^{f(\mathbf{S}(R_i))} = \prod_{R_i \in \mathcal{R}(A_m)} |R_i^D \ltimes \{\bar{a}\}|^{f(\mathbf{S}(R_i))} \, .$$

It is possible then to conclude that

$$|q(D)| \leq \sum_{\bar{a} \in q_Y(D)} |q'(D_{\bar{a}})| \leq$$
$$\sum_{\bar{a} \in q_Y(D)} \prod_{R_i \in \mathcal{R}(A_m)} |R_i^D \ltimes \{\bar{a}\}|^{f(\mathbf{S}(R_i))} \leq \prod_{i=1}^n |R_i^D|^{f(\mathbf{S}(R_i))} ,$$

where the last inequality holds by Lemma 25.11.

This finishes the proof of Theorem 25.8.

Fractional Independent Sets

In the remainder of this chapter, we will prove that the AGM bound is tight. To this end, we will consider a fractional version of *independent sets*.

Definition 25.12: Fractional Independent Set

A *fractional independent set* of a hypergraph $H = (V, E)$ is a function

$$g : V \to \mathbb{Q}_{\geq 0}$$

such that, for each edge $e \in E$, it holds that $\sum_{v \in e} g(v) \leq 1$. The *weight* of g is the sum $\sum_{v \in V} g(v)$.

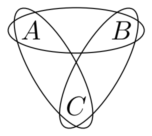
Fractional independent sets assign a nonnegative weight to each node in the hypergraph, such that the sum of the weights of the nodes in a single hyperedge should not exceed 1. This generalizes the condition of the "classical" independent set problem that states that at most one node per edge is allowed in the independent set. We illustrate the notion on an example.

Example 25.13

Consider again the join query

$$q_\triangle = R[A, B] \bowtie S[B, C] \bowtie T[C, A] ,$$

and its hypergraph $H_{q_\triangle}$



We can write the conditions for a fractional independent set g of $H_{q_\triangle}$ as the following system of equations.

$$\begin{aligned} &g(A), g(B), g(C) \geq 0, \\ &g(A) + g(B) \leq 1, \quad g(B) + g(C) \leq 1, \quad g(C) + g(A) \leq 1 . \end{aligned}$$

The equation $g(A) + g(B) \leq 1$ is obtained by considering the hyperedge $\{A, B\}$. One solution to this system of equations, i.e., a fractional independent set, is $g(A) = g(B) = g(C) = 1/2$. The weight of this fractional independent set is $3/2$.

Connection to Linear Programming

Fractional edge covers and fractional independent sets are closely related to each other, which will become clear when we explore their connection to linear programming. To this end, we first need to introduce some background on linear programs.

Definition 25.14: Linear Program

Let $a_1, \dots, a_n$ be real numbers and $x_1, \dots, x_n$ be variables. A *linear function (over $x_1, \dots, x_n$)* is a function f of the form

$$f(x_1, \dots, x_n) = a_1 x_1 + a_2 x_2 + \cdots + a_n x_n \, .$$

If f is a linear function and b a real number, then the equations

$$f(x_1, \dots, x_n) \leq b \quad \text{and} \quad f(x_1, \dots, x_n) \geq b$$

are *linear inequalities*. A *linear program (over $x_1, \dots, x_n$)* consists of a linear function f, a set of linear inequalities, and a *task*, which is to *minimize* or to *maximize*.

Let us illustrate how we will denote linear programs by means of a few examples.

Example 25.15

When we denote $f(R)$, $f(S)$, and $f(T)$ as variables x_R, x_S, and x_T, respectively, then we can associate a linear program to the system of equations in Example 25.7, namely

$$\begin{array}{rl}
\text{minimize} & x_R + x_S + x_T \\
\text{subject to} & x_R + x_T \geq 1 \\
& x_R + x_S \geq 1 \\
& x_S + x_T \geq 1 \\
\text{and} & x_R \geq 0 \qquad x_S \geq 0 \qquad x_T \geq 0 \, .
\end{array}$$

Here, the linear function is $x_R + x_S + x_T$, the task is to minimize, and we have a set of six linear equations.

Likewise, when we denote $g(A)$, $g(B)$, and $g(C)$ as variables y_A, y_B, and y_C, respectively, then we can associate a linear program to the system of equations in Example 25.13, namely

$$\begin{array}{rl} \text{maximize} & y_A + y_B + y_C \\ \text{subject to} & y_A + y_B \leq 1 \\ & y_B + y_C \leq 1 \\ & y_C + y_A \leq 1 \\ \text{and} & y_A \geq 0 \qquad y_B \geq 0 \qquad y_C \geq 0\,. \end{array}$$

Here, the task is to maximize.

Definition 25.16: Feasible Solutions and Optimal Values

Let L be a linear program with linear inequalities

$$f_1(x_1, \dots, x_n)\ \theta_1\ b_1$$
$$\vdots$$
$$f_k(x_1, \dots, x_n)\ \theta_k\ b_k\ ,$$

where $\theta_i \in \{\leq, \geq\}$ for every $i \in [k]$. A *feasible solution* to L is an assignment $\rho : \{x_1, \dots, x_n\} \to \mathbb{R}$ such that $f_i(\rho(x_1), \dots, \rho(x_n))\ \theta_i\ b_i$ holds for every $i \in [k]$.

Let L additionally have the linear function f. If the task of L is to minimize (resp., to maximize), then its *optimal value*, if it exists, is the smallest (resp., largest) value of $f(\rho(x_1), \dots, \rho(x_n))$ such that ρ is a feasible solution to L. In this case, we call ρ an *optimal solution* to L.

Example 25.17

Both ρ_1 with $\rho_1(x_R) = \rho_1(x_S) = \rho_1(x_T) = 1/2$ and ρ_2 with $\rho_2(x_R) = \rho_2(x_S) = 1$ and $\rho_2(x_T) = 0$ are feasible solutions to the first linear program in Example 25.15. One can show that the optimal value is $3/2$ and therefore ρ_1 is an optimal solution.

Linear Programming Duality

We will use the Strong Duality Theorem of linear programming, which we state next. To this end, the *dual* of a linear program of the form

$$\begin{array}{lll} \text{minimize} & \displaystyle\sum_{i=1}^{n} c_i x_i & \\ \text{subject to} & \displaystyle\sum_{i=1}^{n} a_{i,j} x_i \geq b_j & \text{for each } j \in [m] \ , \\ \text{and} & x_i \geq 0 & \text{for each } i \in [n] \end{array}$$

over the variables $x_1, \dots, x_n$ is the linear program

$$\begin{array}{lll} \text{maximize} & \displaystyle\sum_{j=1}^{m} b_j y_j & \\ \text{subject to} & \displaystyle\sum_{j=1}^{m} a_{i,j} y_j \leq c_i & \text{for each } i \in [n] \ , \\ \text{and} & y_j \geq 0 & \text{for each } j \in [m] \end{array}$$

over the variables $y_1, \dots, y_m$.

Theorem 25.18: Strong Duality Theorem

Let ρ^* be the optimal value of a linear program P and let τ^* be the optimal value of its dual program P_D. Then $\rho^* = \tau^*$.

We are now ready to show the key observation that ties the previous notions in this chapter together with linear programs. The observation is that, if we consider a join query q, then the linear program that maximizes the weight of the fractional independent set of H_q is the dual of the linear program that minimizes the weight of the fractional edge cover of H_q. For instance, in Example 25.15, the second linear program is the dual of the first one. We make this more precise next.

Let $q = R_1 \bowtie \cdots \bowtie R_n$ be a join query over some schema $\mathbf{S}$ and assume that $\{A_1, \dots, A_m\}$ is the set of all attributes occurring in q. Recall that we use $\mathcal{R}(A_j)$ to denote the set of relation names R_i in $\{R_1, \dots, R_n\}$ that use attribute A_j under schema $\mathbf{S}$, i.e., such that $A_j \in \mathbf{S}(R_i)$.

Let us denote the linear program

$$\begin{array}{lrcll} \text{minimize} & \displaystyle\sum_{i=1}^{n} x_i & & & \\ \text{subject to} & \displaystyle\sum_{R_i \in \mathcal{R}(A_j)} x_i & \geq & 1 & \text{for each } j \in [m] \ , \\ \text{and} & x_i & \geq & 0 & \text{for each } i \in [n] \ , \end{array}$$

that minimizes the weight of the fractional edge cover of H_q, as $\mathrm{EC}(H_q)$. Likewise, let us denote the linear program

$$\begin{array}{lll} \text{maximize} & \displaystyle\sum_{j=1}^{m} y_j & \\ \text{subject to} & \displaystyle\sum_{A_j \in \mathbf{S}(R_i)} y_j \leq 1 & \text{for each } i \in [n] \\ \text{and} & y_j \geq 0 & \text{for each } j \in [m] \,. \end{array}$$

that maximizes the weight of the fractional independent set of H_q, as $\mathrm{IS}(H_q)$. Notice that $\mathrm{IS}(H_q)$ is indeed the dual program of $\mathrm{EC}(H_q)$.

Let us call a fractional edge cover f of H_q *minimal* if its weight equals the optimal value of $\mathrm{EC}(H_q)$; and a fractional independent set g *maximal* if its weight equals the optimal value of $\mathrm{IS}(H_q)$. Then Theorem 25.18 implies the following Corollary.

Corollary 25.19

Let q be a join query and H_q be its hypergraph. Let f be a minimal fractional edge cover and g be a maximal fractional independent set of H_q. Then the weight of f equals the weight of g.

Optimality of the AGM Bound

We show next that the bound stated in Theorem 25.8 is strict.

Proposition 25.20

Let $q = R_1 \bowtie \cdots \bowtie R_n$ be a join query over schema $\mathbf{S}$ and let f be a minimal edge cover of the hypergraph H_q of q. Then there exist arbitrarily large databases D such that $|R_i^D| \geq 1$ for each $i \in [n]$ and

$$|q(D)| \;=\; \prod_{i=1}^{n} |R_i^D|^{f(\mathbf{S}(R_i))} \,.$$

Proof. Assume that $\{A_1, \ldots, A_m\}$ is the set of attributes occurring in q. Let f be a minimal fractional edge cover and g be a maximal fractional independent set of H_q. Let f^* be the weight of f and g^* be the weight of g.

We now define a database D_k for each $k \in \mathbb{N}$. For each $i \in [n]$, let σ_i be a function such that $\mathbf{S}(R_i) = \{A_{\sigma_i(1)}, \ldots, A_{\sigma_i(r)}\}$, where $r = \mathrm{ar}_{\mathbf{S}}(R_i)$. We define the database D_k as

$$R_i^{D_k} = \big\{(A_{\sigma_i(1)} : j_1, \ldots, A_{\sigma_i(r)} : j_r) \mid \\ j_\ell \in \{1, \ldots, \lfloor k^{g(A_{\sigma_i(\ell)})} \rfloor\} \text{ for every } \ell \in [r]\big\} \,.$$

It is straightforward to show that

$$q(D_k) = \{1, \dots, \lfloor k^{g(A_1)} \rfloor\} \times \cdots \times \{1, \dots, \lfloor k^{g(A_m)} \rfloor\} \,.$$

Since $g(A_j) \in \mathbb{Q}$ for every $j \in [m]$, there are arbitrarily large k such that every value $k^{g(A_j)}$ is an integer. In this case, all relations $R_i^{D_k}$ have size k and we have that $|q(D_k)| = k^{\sum_{j=1}^m g(A_j)}$. We then obtain that

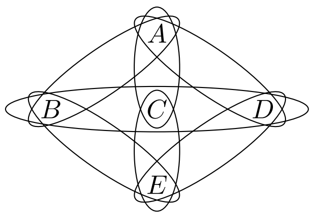
$$\prod_{i=1}^n |R_i^{D_k}|^{f(\mathbf{S}(R_i))} = k^{\sum_{i=1}^n f(\mathbf{S}(R_i))} = k^{f^*} = k^{g^*} \,.$$

Here, the first equality holds because $|R_i^{D_k}| = k$. The second equality is by definition of f^*, and the third is because $f^* = g^*$ (Corollary 25.19). This concludes the proof. □

We illustrate the construction of Proposition 25.20 with an example.

Example 25.21

Consider a join query with the following hypergraph H:



This hypergraph has a fractional edge cover f that assigns weight $\frac{1}{3}$ to every edge, amounting to a total weight of $\frac{7}{3}$. It has a fractional independent set g that assigns

- weight $\frac{1}{3}$ to B, C, and D; and
- weight $\frac{2}{3}$ to A and E,

also amounting to a total weight of $\frac{7}{3}$. Therefore, f is minimal and g is maximal. We now use the construction in the proof of Proposition 25.20 to construct a database from g. Let us choose a value of k such that $k^{g(v)}$ is an integer for every node v of H. This is the case for $k = 8$, since $\sqrt[3]{8} = 2$ and $8^{2/3} = 4$. The construction in Proposition 25.20 now defines the database D_k as

A / E	B / C / D
1	1
1	2
2	1
2	2
3	1
3	2
4	1
4	2

B	C	D
1	1	1
1	1	2
1	2	1
1	2	2
2	1	1
2	1	2
2	2	1
2	2	2

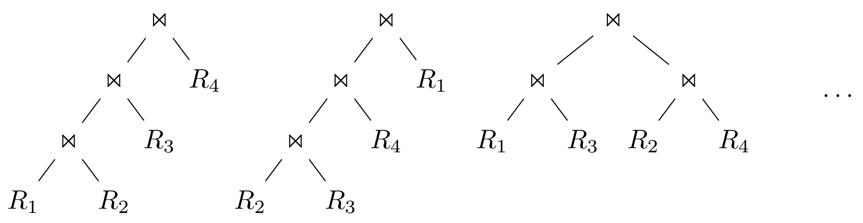
where every binary relation, i.e., over attributes $\{A, B\}$, $\{A, C\}$, $\{A, D\}$, $\{E, B\}$, $\{E, C\}$, and $\{E, D\}$ is like the relation on the left and the ternary relation like the relation on the right.

26

Worst-Case Optimal Join Algorithms

Join Plans

In a database system, join queries are typically evaluated using *join plans*, which aim at finding a clever ordering in which to perform the joins. Assume, for example, that we have a join query $q = R_1 \bowtie R_2 \bowtie R_3 \bowtie R_4$. Due to the commutativity and associativity of the join operator, we can for example consider the trees



Each of these trees represent a join plan: if we have a database D with relations $R_1, \ldots, R_4$ and evaluate the joins of q in a bottom-up fashion over these trees, we end up with $q(D)$ at the root of each tree.

Since joins are very expensive operators in database systems, it makes sense to think about which join plan leads to the fastest computation of the result. Here, it is important to know that, in general, joining big relations typically takes more time than joining small relations. Query optimizers therefore try to perform joins that produce small results early, since it is easier and faster to continue the computation with a small intermediate result than with a large one. For instance, if $|R_1 \bowtie R_2(D)|$ is much smaller than $|R_2 \bowtie R_3(D)|$, we may prefer the leftmost join plan to the middle one. Since the join plans we considered here always take the join of two relations, they are called *pairwise join plans*.

Are Pairwise Join Plans Sub-Optimal?

Perhaps surprisingly, there exist cases in which every possible pairwise join plan produces after the first step an intermediate result that contains more tuples than the entire output. Consider the join query

$$q(A, B, C) \coloneq R(A, B), S(B, C), T(C, A)$$

and, for $n \in \mathbb{N}$, the database D_n in which $R[A, B] = S[B, C] = T[C, A]$ is the relation

1	1
1	2
⋮	⋮
1	$n/2$
2	1
⋮	⋮
$n/2$	1

which consists of $n - 1$ tuples.

Any join plan that starts with joining two relations will produce an intermediate result that is quadratic in n, because, if we consider $q_{RS} = R \bowtie S$, $q_{TR} = T \bowtie R$, and $q_{ST} = S \bowtie T$, then we have

$$|q_{RS}(D_n)| = |q_{TR}(D_n)| = |q_{ST}(D_n)| = \left(\frac{n}{2}\right)^2 = \frac{n^2}{4} \, .$$

But, according to the AGM bound, we have that $|q(D)| \leq n^{3/2}$.

This example shows that no query evaluation algorithm that uses join plans in the way that we explained in the beginning of this chapter has a runtime proportional to the number of tuples in the output. Furthermore, it raises the quesion whether such an algorithm even exists. We will show in this chapter that there exists an algorithm that can evaluate join queries q on databases D in time $\tilde{O}(nm|q(D)|)$, where n and m are the number of atoms and attributes in q, respectively. Here, the $\tilde{O}$-notation hides logarithmic factors, see Appendix A.

Worst-Case Optimal Join By Example

Consider again the join query $q(A, B, C) \coloneq R(A, B), S(B, C), T(C, A)$ and the database D in Figure 26.1. Here, we represent atoms $R(a, b)$, $S(a, b)$, and $T(a, b)$ with green, blue, and orange arrows from a to b, respectively.

The algorithm that we present in this section initializes three sets L_1, L_2, and L_3 as empty and computes $q(D)$ in the following steps, each of which focuses on an attribute A, B, or C:

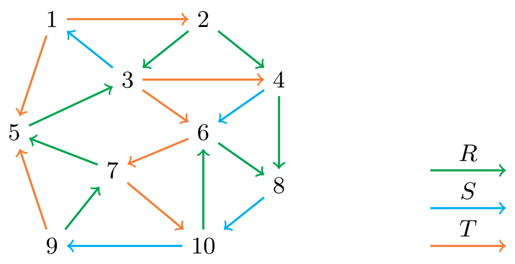


Fig. 26.1: Visualization of a database for illustrating worst-case optimal join

(1) Compute $L_1 := \pi_A(R \bowtie T)$.

(2) For each $a \in L_1$,

- compute the values b in $\pi_B(R \bowtie S)$ such that $(a, b) \in R$ and
- add the pairs (a, b) to L_2.

(3) For each $(a, b) \in L_2$,

- compute the values c in $\pi_C(S \bowtie T)$ such that $(b, c) \in S$ and $(c, a) \in T$
- add the triples (a, b, c) to L_3.

(4) Return L_3.

On the database in Figure 26.1, the algorithm therefore computes

(1) $L_1 = \{5, 2, 6, 7, 4, 10\}$;

(2) $L_2 = \{(5, 3), (2, 3), (2, 4), (6, 8), (4, 8)\}$;

(3) $L_3 = \{(5, 3, 1), (2, 3, 1)\}$.

A Worst-Case Optimal Join Algorithm

We now describe a join algorithm that is worst-case optimal. As in Chapter 25, fix a join query $q = R_1 \bowtie \cdots \bowtie R_n$ over schema $\mathbf{S}$, and assume that $X = \{A_1, \ldots, A_m\}$ is the set of attributes occurring in q. Given $Y \subseteq X$, recall that $\mathcal{R}(Y) = \bigcup_{A_j \in Y} \mathcal{R}(A_j)$ is the set of those R_i's that mention some attribute in Y, for $i \in [n]$. Moreover, assuming that $\mathcal{R}(Y) = \{R_{i_1}, \ldots, R_{i_\ell}\}$, where $1 \leq i_1 < \cdots < i_\ell \leq n$, recall that query q_Y is

$$\pi_{Y \cap \mathbf{S}(R_{i_1})}(R_{i_1}) \bowtie \cdots \bowtie \pi_{Y \cap \mathbf{S}(R_{i_\ell})}(R_{i_\ell}) .$$

Algorithm 10, called AE-Join (short for "Attribute Elimination Join") describes a worst-case optimal join algorithm by repeated elimination of attributes. The algorithm considers the attributes in $\{A_1, \ldots, A_m\}$ one by one

Algorithm 10 AEJOIN(q, D)

Input: Join query q using attributes $A_1, \ldots, A_m$ and database D
Output: $q(D)$

```
1: $L_0 := \{()\}$                                  ▷ $L_0$ is the set with the empty tuple
2: for $i = 1, \ldots, m$ do
3:     $L_i := \emptyset$
4:     for each tuple $\bar{a} \in L_{i-1}$ do
5:         $V := \bigcap_{R \in \mathcal{R}(A_i)} \pi_{\{A_i\}}(R^D \ltimes \{\bar{a}\})$
6:         $L_i := L_i \cup (\{\bar{a}\} \times V)$
7: return $L_m$
```

and completely deals with it before going to the next. As such, it iteratively computes sets $L_0, \ldots, L_m$, where each L_i can be seen as the result of the query, *considering only the attributes* $A_1, \ldots, A_i$. As we show next, L_m is the result of q on D.

Proposition 26.1

Given a join query q and a database D, we have that AEJOIN(q, D) returns $q(D)$.

Proof. We show by induction on $i \in [m]$ that $L_i = q_Z(D)$, with $Z = \{A_1, \ldots, A_i\}$. The result then follows since $q_Z = q$ when $Z = \{A_1, \ldots, A_m\}$. For the base case $i = 1$, this holds trivially, as we have that

$$L_1 \;=\; \bigcap_{R \in \mathcal{R}(A_1)} \pi_{\{A_1\}} R^D \;=\; q_{\{A_1\}}(D)$$

by definition. Consider now the inductive case i, for $i > 1$, and define $Y = \{A_1, \ldots, A_{i-1}\}$. Notice that, by definition of $q_{\{A_i\}}$, it holds that $V = q_{\{A_i\}}(R^{D_{\bar{a}}})$, where $D_{\bar{a}}$ is the database over $\mathbf{S}'$ such that $R^{D_{\bar{a}}} = \pi_{\{A_i\}}(R^D \ltimes \{\bar{a}\})$ for each $R \in \mathcal{R}(A_i)$. Hence,

$$L_i \;=\; \bigcup_{\bar{a} \in q_Y(D)} \left(q_{\{A_i\}}(R^{D_{\bar{a}}}) \times \{\bar{a}\}\right).$$

But, as explained in Equation (25.8) and the analysis that precedes it, the right-hand side of this expression coincides with $q_Z(D)$. □

We invite the reader to compare how Algorithm 10 implements the procedure on the triangle query that we described before. Furthermore, notice that the exact execution of algorithm is highly dependent on the ordering $A_1, \ldots, A_m$ of the attributes in the input. Indeed, the for-loop considers increasing values of i from 1 to m, which corresponds to first considering A_1, then A_2 etc. However, we will show that the algorithm is worst-case optimal independent of the ordering $A_1, \ldots, A_m$ of the attributes.

Complexity Analysis

We will prove that Algorithm 10 runs in time

$$\tilde{O}\left(n \cdot m \cdot \prod_{j=1}^{n} |R_j^D|^{x_j}\right),$$

where $(x_1, \ldots, x_n)$ is a fractional edge cover of q.

In order to reach this bound, we will rely on a couple of assumptions. To facilitate the analysis, we assume that all relations R^D mentioned in q are non-empty. Notice that this assumption is without loss of generality, since it can be checked in constant time before evaluating the query. Second, we assume that there exists an ordering on the values in the database, since we will work with sorted lists.

Furthermore, we rely on the following complexity assumption on how the relations in D can be accessed:

Claim 26.2. *Let R be a relation containing n tuples. We can construct a data structure such that, given an attribute name A_i of R, and a tuple $\bar{a} = (A_1 : a_1, \ldots, A_{i-1} : a_{i-1})$, we can get access to an increasingly sorted list λ of values b such that $(\bar{a}, b) \in \pi_{\{A_1,\ldots,A_i\}} R$ in time $O(i \log n)$.*

Notice that not all attributes $A_1, \ldots, A_i$ have to be attributes of R. Finally, we will rely on the following claim about sorted lists:

Claim 26.3. *Given n lists $\lambda_1, \ldots, \lambda_n$ of increasingly sorted values, we can compute the set $V = \{j \mid j$ occurs in each list $\lambda_1, \ldots, \lambda_n\}$ in time $\tilde{O}(n \cdot \min_{i \in [n]} |\lambda_i|)$.*

We will prove Claims 26.2 and 26.3 in Chapter 27.

Using these claims, we show that the algorithm can be implemented such that for every $i \in [m]$,

(1) at the end of the ith for-loop in line 2, we have computed L_i in time

$$\tilde{O}\left(n \cdot i \cdot \prod_{j=1}^{n} |R_j^D|^{x_j}\right)$$

since the start of the algorithm and

(2) given a tuple $\bar{a}$, the set V can be computed in time

$$\tilde{O}\left(n \cdot \prod_{R_j \in \mathcal{R}(A_i)} |R_j^D \ltimes \bar{a}|^{x_j}\right).$$

Notice that we measure the time in (1) and (2) differently. In (1) we count the number of steps since the start of the algorithm and in (2) we consider the tuple $\bar{a}$ as input, i.e., we only measure from the beginning of an iteration of the inner for-loop. Our desired bound then follows from (1), taking $i = m$.

We proceed by induction on i. To this end, assume that $i = 1$. In this case, notice that $L_1 = V = \cap_{R \in \mathcal{R}(A_1)} \pi_{\{A_1\}} R^D$. According to Claim 26.2, for each $R \in \mathcal{R}(A_1)$, we can get access to a sorted list λ_R containing the values $\pi_{A_1} R$. Using Claim 26.3, we can compute the set V, which are the values that occur in each of these lists, in time

$$\tilde{O}\left(n \cdot \min_{R \in \mathcal{R}(A_1)} |R^D|\right) .$$

Furthermore, we have that

$$\begin{aligned} n \cdot \min_{R \in \mathcal{R}(A_1)} |R^D| &\leq n \cdot \prod_{R_j \in \mathcal{R}(A_1)} \big(\min_{R \in \mathcal{R}(A_1)} |R^D|\big)^{x_j} \\ &\leq n \cdot \prod_{R_j \in \mathcal{R}(A_1)} |R_j^D|^{x_j} \\ &\leq n \cdot \prod_{j=1}^{n} |R_j^D|^{x_j} , \end{aligned}$$

where the first inequality holds because $(x_1, \ldots, x_n)$ is a fractional edge cover and the last line holds because we assumed (without loss of generality) that all relations R_j^D are non-empty.

For general i, we first note that, given a tuple $\bar{a}$, we can compute the set V, analogously to the $i = 1$ case, in time

$$\tilde{O}\left(n \cdot \min_{R \in \mathcal{R}(A_i)} |R^D \ltimes \bar{a}|\right)$$

using Claims 26.2 and 26.3. Furthermore,

$$n \cdot \min_{R \in \mathcal{R}(A_i)} |R^D \ltimes \bar{a}| \;\leq\; n \cdot \prod_{R_j \in \mathcal{R}(A_i)}^{n} |R_j^D \ltimes \bar{a}|^{x_j} ,$$

again analogously to the $i = 1$ case.

We will now prove that, at the end of an iteration of the for-loop in line 2, we can compute L_i in

$$\tilde{O}\left(n \cdot i \cdot \prod_{j=1}^{n} |R_j^D|^{x_j}\right)$$

steps since the beginning of the entire algorithm. By induction, we can finish computing L_{i-1} after

$$\tilde{O}\left(n \cdot (i-1) \cdot \prod_{j=1}^{n} |R_j^D|^{x_j}\right)$$

steps, which we then use to compute L_i. So we need to show that we can compute L_i using an additional

$$\tilde{O}\Big(n \cdot \prod_{j=1}^{n} |R_j^D|^{x_j}\Big)$$

steps, making use of L_{i-1}. Indeed, given L_{i-1}, we can compute L_i in

$$\tilde{O}\Big(\sum_{\bar{a} \in L_{i-1}} \Big(n \cdot \prod_{R_j \in \mathcal{R}(A_i)} |R_j^D \ltimes \bar{a}|^{x_j}\Big)\Big)$$

steps and, taking $Y = \{A_1, \ldots, A_{i-1}\}$ we have that

$$\begin{aligned}
\sum_{\bar{a} \in L_{i-1}} \Big(n \cdot \prod_{R_j \in \mathcal{R}(A_i)} |R_j^D \ltimes \bar{a}|^{x_j}\Big) &= n \cdot \sum_{\bar{a} \in L_{i-1}} \Big(\prod_{R_j \in \mathcal{R}(A_i)} |R_j^D \ltimes \bar{a}|^{x_j}\Big) \\
&= n \cdot \sum_{\bar{a} \in q_Y(D)} \Big(\prod_{R_j \in \mathcal{R}(A_i)} |R_j^D \ltimes \bar{a}|^{x_j}\Big) \\
&\leq n \cdot \sum_{\bar{a} \in q_Y(D)} \Big(\prod_{R_j \in \mathcal{R}(\{A_i, \ldots, A_m\})} |R_j^D \ltimes \bar{a}|^{x_j}\Big) \\
&\leq n \cdot \prod_{j=1}^{n} |R_j^D|^{x_j}
\end{aligned}$$

where the last inequality is by Lemma 25.11, the Query Decomposition Lemma.

27

Leapfrog Triejoin

In Chapter 26, we outlined and analyzed a worst-case optimal join algorithm. The algorithm itself, however, is still described in a relatively high-level fashion and its complexity analysis relies on Claim 26.2 and 26.3, which we have not proved yet. In this chapter, we describe a more detailed approach and prove Claims 26.2 and 26.3.

For readability, our presentation in this chapter is sometimes going to be somewhat informal. The reason is that this allows us to use standard concepts from object-oriented programming.

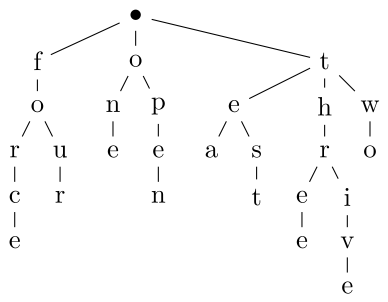
Tries

A *trie*, also known as a *prefix tree*, is a data structure that is commonly used to store a set of words in a tree. We informally introduce tries here before discussing in the next section how we will use them to represent relations in more detail.

Assume that we have a set of words S over the Latin alphabet, that is, the set $\mathbb{A} = \{\text{a, b, c, } \ldots, \text{ z}\}$ of lowercase symbols. A trie that represents the words in S will be a tree in which every node is labeled with a symbol from $\mathbb{A}$, except for the root, which carries a special label "$\bullet$". Furthermore, every root-to-leaf path is labeled with a word from S (ignoring the special label of the root), the symbols of siblings are alphabetically ordered from left to right, and siblings always carry different labels.

Example 27.1: A Trie for a Set of Words

The following is a trie for the words {force, four, one, open, tea, test, three, thrive, two}.



Representing Relations using Tries

In this chapter, we use tries to represent relations, as illustrated in the next example. In order to do this, assume that we have a database D over some schema $\mathbf{S}$. Furthermore, let $\{A_1, \dots, A_m\}$ be all the attribute names in D or, more formally, $\{A_1, \dots, A_m\} = \cup_{R \in \mathrm{Dom}(\mathbf{S})} \mathbf{S}(R)$.

In this chapter we will fix an ordering of the attribute names. The ordering is arbitrary and the presented results hold independently of the ordering we choose, but it is important that the ordering is consistent throughout this chapter. For simplicity, let us simply order the attribute names by their index, that is, $A_1, \dots, A_m$. Furthermore, again for simplicity, we will assume that the domain of D is the set of natural numbers $\mathbb{N}$, but the results hold for any ordered domain.

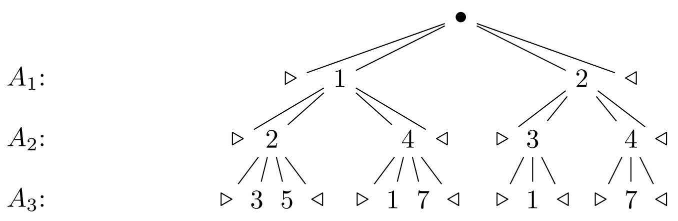
We now explain how we represent a relation R^D as a trie. It will be similar to the trie for words that we presented before, but some details will differ. Intuitively, when we represent a relation R^D with attributes $\mathbf{S}(R) = \{A_{i_1}, \dots, A_{i_k}\}$ with $i_1 < \cdots < i_k$, we first build the trie for the set of words $\{a_{i_1} \cdots a_{i_k} \mid (A_{i_1} : a_{i_1}, \dots, A_{i_k} : a_{i_k}) \in R^D\}$, where we use the ordering in $\mathbb{N}$ to order siblings left to right instead of the alphabetical ordering. Then, to every sequence of siblings, we add a node labeled $\triangleright$ as a new leftmost node and a node labeled $\triangleleft$ as a new rightmost node. These new nodes are added for technical reasons, as they will simplify some of the algorithms later in the chapter. Since siblings in tries are ordered increasingly and since we use data values in $\mathbb{N}$ in this chapter, we will assume that $\triangleright < n$ and $\triangleleft > n$ for every $n \in \mathbb{N}$.

Example 27.2: A Trie Representing a Relation

Consider the following relation R:

A_1	A_2	A_3
1	2	3
1	2	5
1	4	1
1	4	7
2	3	1
2	4	7
4	3	7

We will use the following trie-like representation of R:



The trie is obtained from R by considering the attribute ordering A_1, A_2, A_3 from the root to the leafs, and by adding the end markers $\triangleright$ and $\triangleleft$ to each sequence of siblings.

We now define the trie representation of a relation more formally.

Definition 27.3: Trie of a Relation

Let D be a database over named schema $\mathbf{S}$ and let R^D be a relation of D. Let $\prec$ be an ordering on attribute names and let $\mathbf{S}(R) = \{A_1, \dots, A_k\}$ with $A_1 \prec \cdots \prec A_k$. The trie $T_R^{\prec}$ is the trie obtained from R^D by

- first constructing the trie for the set of words $\{a_1 \cdots a_k \mid (A_1 : a_1, \dots, A_k : a_k) \in R_D\}$ and
- adding new nodes labeled $\triangleright$ and $\triangleleft$ to the left and right of each non-empty sequence of siblings, respectively.

Notice that $T_R^{\prec}$ is indeed a trie, i.e., for the set of words $W \cup W^{\triangleright} \cup W^{\triangleleft}$, where

- $W = \{a_1 \cdots a_k \mid (A_1 : a_1, \ldots, A_k : a_k) \in R_D\}$,
- $W^{\triangleright} = \{a_1 \cdots a_i \triangleright \mid 0 \leq i < k \text{ and } a_1 \cdots a_k \in W\}$,
- $W^{\triangleleft} = \{a_1 \cdots a_i \triangleleft \mid 0 \leq i < k \text{ and } a_1 \cdots a_k \in W, i < k\}$.

(When $i = 0$, then $a_1 \cdots a_i \triangleright$ is simply the word $\triangleright$; similar for $a_1 \cdots a_i \triangleleft$.)

In the remainder of the chapter, if we write T_R, i.e., we don't explicitly mention the ordering $\prec$, it means that we are assuming that R has attribute names A_i, with i ranging over a finite subset of $\mathbb{N}$, and that the attribute ordering is in the order of the increasing index numbers.

Trie Iterators

In order to interact with tries, we will use so-called *trie iterators*. Iterators are a common concept in object-oriented programming, which means that readers who have some experience with programming may already be familiar with the concept. Intuitively, iterators are used to traverse a collection of objects, and we will use them to traverse tries. Conceptually, for the purpose of this book, they can be understood as follows.

Definition 27.4: Trie Iterator

Let T be a trie. A *trie iterator* for T is a variable I that can be bound to a node in T on which the following operations can be performed.

$I.\mathrm{init}(\mathrm{trie}\ T)$	binds I to the root of T
$I.\mathrm{value}()$	returns the value of the node that I is bound to
$I.\mathrm{next}()$	binds I to the right sibling of the node it is bound to
$I.\mathrm{open}()$	binds I to the leftmost child of the node it is bound to
$I.\mathrm{up}()$	binds I to the parent of the node it is bound to
$I.\mathrm{seek}(\mathrm{value}\ v)$	binds I to its first sibling with value at least v

We call I *freshly initialized on* T, if $I.\mathrm{init}(T)$ has been executed and no other operations have been performed on I afterwards.

We will assume in this chapter that we already have an implementation trie iterators available. Notice that such a trie iterator is indeed easy to implement (we leave this as an exercise). Furthermore, we assume that this implementation can perform the operations $I.\mathrm{init}(T)$, $I.\mathrm{value}()$, $I.\mathrm{next}()$, $I.\mathrm{open}()$, and $I.\mathrm{up}()$ in constant time and $I.\mathrm{seek}(v)$ in time $O(\log n)$, where n is the number of nodes in the trie. Notice that $I.\mathrm{seek}(v)$ can be easily implemented to run in this time using binary search trees. (In real database systems, one would of course use a more advanced data structure that can elegantly deal with updates to the data, such as B-trees.)

Let us illustrate our use of trie iterators with an example.

Algorithm 11 Trie-Enumerate($I, \bar{a}$)

```
Input: A trie iterator $I$ on trie $T_R$ and a tuple $\bar{a} = (A_1 : a_1, \ldots, A_n : a_n)$
Output: $\sigma_{A_1 \doteq a_1 \wedge \cdots \wedge A_n \doteq a_n} R$
1: if arity($\bar{a}$) = depth($T_R$) then return $\bar{a}$
2: else
3:     $I$.open()
4:     $I$.next()
5:     while $I$.value() $\neq \triangleleft$ do
6:         Trie-Enumerate($I, (\bar{a}, I$.value()))
7:         $I$.next()
8:     $I$.up()
```

Example 27.5: Using a Trie Iterator

Consider the trie T_R for the relation R in Example 27.2. Let I be a trie iterator. If we perform $I.\,\mathrm{init}(T_R)$, then I is initialized and bound to the root of T_R. At this point, $I.\,\mathrm{value}()$ would return $\bullet$. If we then perform the sequence of operations

$$I.\,\mathrm{open}(),\ I.\,\mathrm{next}(),\ I.\,\mathrm{open}(),\ I.\,\mathrm{next}(),\ I.\,\mathrm{open}()\ ,$$

then I is bound to the leftmost node at depth three (corresponding to the values of attribute A_3), labeled $\triangleright$. At this point, the sequence of operations $I.\,\mathrm{seek}(4)$, $I.\,\mathrm{value}()$ would return 5. Indeed, the seek operation would search the leftmost sibling with value at least 3. Since the siblings have values 3, 5, and $\triangleleft$, the first one with value at least 4 has value 5.

At this point, we have everything in place to prove Claim 26.2, which we repeat here for convenience:

Claim 26.2. *Let R be a relation containing n tuples. We can construct a data structure such that, given an attribute name A_i of R, and a tuple $\bar{a} = (A_1 : a_1, \ldots, A_{i-1} : a_{i-1})$, we can get access to an increasingly sorted list λ of values b such that $(\bar{a}, b) \in \pi_{\{A_1,\ldots,A_i\}} R$ in time $O(i \log n)$.*

Proof. Given a tuple $\bar{a} = (A_1 : a_1, \ldots, A_{i-1} : a_{i-1})$, we can navigate to the starting node of λ by initializing a trie iterator I with T_R and then calling $I.\,\mathrm{seek}(a_1)$, $\ldots$, $I.\,\mathrm{seek}(a_{i-1})$, $I.\,\mathrm{seek}(b)$. At that point, we can navigate through λ by calling $I.\,\mathrm{next}()$. □

We now show that, using a trie iterator on T_R, it is easy to output the relation R^D. We summarize the algorithm in Algorithm 11. It should be called with a freshly initialized iterator on T_R and the empty tuple.

Proposition 27.6

Consider a relation R and its trie T_R. Let I be a freshly initialized iterator on T_R. Then

$$\text{Enumerate}(I, ()) \text{ returns the set of tuples in } R^D \ .$$

The Leapfrog Algorithm

In this section we present the *Leapfrog* algorithm, which works on unary relations. Given unary relations $R_1, \dots, R_n$ over the same attribute, Leapfrog computes the output

$$R_1 \bowtie \cdots \bowtie R_n \ .$$

Notice that, in this case, $R_1 \bowtie \cdots \bowtie R_n$ is the same as $R_1 \cap \cdots \cap R_n$, so the algorithm can also be understood as an algorithm that computes the intersection of n sets. Furthermore, it will prove Claim 26.3.

Example 27.7: List Representation of Unary Relations

Consider the unary relation $R = \{1, 5, 7, 9\}$. We will represent this unary relation as the sorted list

$$\triangleright \quad 1 \quad 5 \quad 7 \quad 9 \quad \triangleleft \ .$$

Observe that the list representation of a unary relation is almost the same as its trie representation. (We only omit the root of the trie.) For navigating through such sorted lists, we will use so-called *list iterators*, which are similar to trie iterators but only provide a subset of the operations.

Definition 27.8: List Iterator

Let L be a sorted list. A *list iterator* for L is a variable I that can be bound to a node in L on which the following operations can be performed.

$I.\text{init}(\text{list } L)$	binds I to the leftmost node in L
$I.\text{value}()$	returns the value of the node that I is bound to
$I.\text{next}()$	binds I to the node to the immediate right of the node it is bound to
$I.\text{seek}(\text{value } v)$	binds I to the leftmost node to the right of the current node, with value at least v

Similar to trie iterators, we will assume that we have an implementation of list iterators available. Furthermore, we assume that this implementation can

Algorithm 12 Leapfrog class

```
Input: List iterators $J_0, \ldots, J_{n-1}$ for unary relations $R_0, \ldots, R_{n-1}$
Provides: List iterator for $R_0 \bowtie \cdots \bowtie R_{n-1}$

 1: internal records: List iterators $I_0, \ldots, I_{n-1}$

 2: constructor Leapfrog($J_0, \ldots, J_{n-1}$)         ▷ Constructor for leapfrog iterator
 3:     $(I_0, \ldots, I_{n-1}) := (J_0, \ldots, J_{n-1})$

 4: function value()
 5:     return $I_p$. value()

 6: function sync()
 7:     max := $I_0$. value()
 8:     min := $I_{n-1}$. value()
 9:     $p := 1$
10:     while min $\neq$ max do
11:         $I_p$. seek(max)
12:         max := $I_p$. value()
13:         $p := p + 1 \mod n$
14:         min := $I_p$. value()

15: function next()
16:     $I_0$. next()
17:     sync()

18: function seek(value $v$)
19:     $I_0$. seek($v$)
20:     sync()
```

Algorithm 13 Leapfrog-Join

```
Input: Initialized list iterators $I_1, \ldots, I_n$ for the unary relations $R_1, \ldots, R_n$
Output: $R_1 \bowtie \cdots \bowtie R_n$
21: $I$ := Leapfrog($I_1, \ldots, I_n$)
22: while $I$. value() $\neq \triangleleft$ do
23:     $I$. next()
24:     return $I$. value()
```

do $I.\,\mathrm{init}(L)$, $I.\,\mathrm{value}()$, and $I.\,\mathrm{next}()$ in constant time and $I.\,\mathrm{seek}(v)$ in time $O(\log n)$, where n is the number of elements in the list. These are the same complexities that we assumed for trie iterators.

Let $R_1, \ldots, R_n$ be a set of unary relations over the same attribute and let $I_1, \ldots, I_n$ be initialized iterators for their respective list representations. Algorithms 12 and 13 describe the code on how to compute $R_1 \bowtie \cdots \bowtie R_n$. Algorithm 12 provides the *Leapfrog class*, for which an instance can be constructed using a set of list operators $I_0, \ldots, I_{n-1}$ (lines 2–3). (We start numbering from zero in this class because it simplifies the code due to the

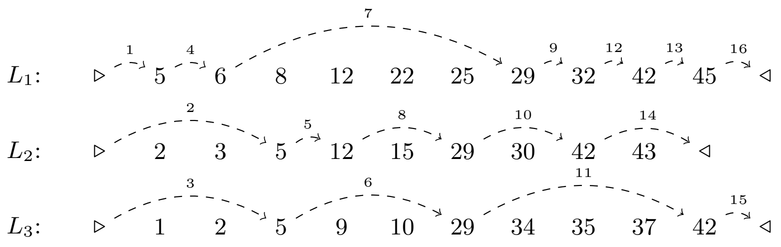


Fig. 27.1: Illustration of the Leapfrog algorithm

modulo operator in line 13.) This class provides the functionality of a list iterator over the *list of common values* seen by $I_0, \ldots, I_{n-1}$.

As such, the list iterator provided by Algorithm 12 can be used by Algorithm 13 to enumerate all values in $R_1 \bowtie \cdots \bowtie R_n$. Algorithm 13 should be started with calling Leapfrog-Join$(I_1, \ldots, I_n)$, where all iterators $I_1, \ldots, I_n$ are freshly initialized, that is, they are all bound to the leftmost nodes in the sorted lists, all carrying the value $\triangleright$.

Example 27.9: Leapfrog Algorithm

Consider the following lists:

L_1:	$\triangleright$	5	6	8	12	22	25	29	32	42	45	$\triangleleft$
L_2:	$\triangleright$	2	3	5	12	15	29	30	42	43	$\triangleleft$	
L_3:	$\triangleright$	1	2	5	9	10	29	34	35	37	42	$\triangleleft$

Algorithm 13 initializes all iterators on the leftmost element of each list, labeled $\triangleright$. The constructor of Algorithm 12 then provides it with a list iterator for the relation $R_1 \bowtie \cdots \bowtie R_n$. The while loop in Algorithm 13 uses this list operator to output the values in $R_1 \bowtie \cdots \bowtie R_n$: It skips over $\triangleright$ and continues calling next() on this iterator until we reach $\triangleleft$.

Now let us look at how next() is implemented. The function advances the first iterator to the next element on line 16 and then calls the sync() function, which is the main internal function of the list iterator that searches the next common value in $I_0, \ldots, I_{n-1}$.

In order to do this, the sync() function keeps track of the smallest and largest value among all the current values of the lists. In our example, we will have max $= 5$ (since I_0 already advanced one step) and min $= \triangleright$. The sync() function then uses the value p to iterate through $I_0, \ldots, I_{n-1}$ in a round robin fashion. As it does so, it always seeks the smallest value that is at least max in iterator I_p, and updates the values min and max. The function stops when all iterators point to the same value, that is, when min $=$ max. In our example, the sync() function performs the jumps

2 and 3 (using seek(5)) in the following image in Figure 27.1. At this point, all iterators are synchronized on the value 5, which is returned on line 24.

In the next iteration of the while-loop on line 22, we call $I.\text{next}()$ again, which advances the first iterator again (jump 4 in the image above), after which sync() performs jumps 5–8. At this point, we output the value 29. It can be checked that the next iteration of the while-loop on line 22 performs jumps 9–12, after which we output 42, and the last iteration performs jumps 13–16.

We can now prove:

Proposition 27.10

Let $R_1, \ldots, R_n$ be a set of unary relations over the same attribute and let $L_1, \ldots, L_n$ be their respective list representations. Then Leapfrog-Join$(L_1, \ldots, L_n)$ outputs $R_1 \bowtie \cdots \bowtie R_n$. Furthermore, the algorithm runs in time $\tilde{O}(n \min_{i \in [n]} |R_i|)$.

Proof (Sketch). We only provide the main argument concerning the run time. The crux here is that Leapfrog-Join cycles through the lists $L_1, \ldots, L_n$ a number of times (in the seek() function). Each such cycle moves the iterator for each list at least one step to the right. Therefore, the number of such cycles is in $O(\min_{i \in [n]} |R_i|)$.

Furthermore, each such cycle consists of $O(n)$ seek operations, each of which costs $O(\log(|R_1|) + \cdots + \log(|R_n|))$ time. Altogether, these are $\tilde{O}(n \min_{i \in [n]} |R_i|)$ many steps. □

Notice that Proposition 27.10 immediately implies Claim 26.3:

Claim 26.3. *Given n lists $\lambda_1, \ldots, \lambda_n$ of increasingly sorted values, we can compute the set $V = \{j \mid j$ occurs in each list $\lambda_1, \ldots, \lambda_n\}$ in time $\tilde{O}(n \cdot \min_{i \in [n]} |\lambda_i|)$.*

The Triejoin Algorithm

We now want to use the Leapfrog class to build an algorithm that can take initialized trie iterators $I_1, \ldots, I_n$ for arbitrary relations $R_1, \ldots, R_n$ as input and compute $R_1 \bowtie \cdots \bowtie R_n$. It is important for this algorithm, however, that the tries for which we have the iterators are consistent with the same ordering $\prec$ of attributes. The ordering itself can be arbitrary,[1] but it needs to be the same ordering for all the iterators $I_1, \ldots, I_n$.

[1] This is convenient in practice, because indexes may not be available on every attribute of a relation.

The principal idea of the algorithm is similar as before. The algorithm is divided into a *Triejoin class* (Algorithm 14) and *Triejoin* (Algorithm 15), which the triejoin class to produce the result of the join. Notice that we use *Trie-Enumerate* (Algorithm 11) to produce the output.

The Triejoin class maintains a counter $\ell \in \mathbb{N}$ to remember the A central insight for the algorithm is that, after open() has been called on a node u, then the methods value(), next(), and seek(v) work exactly like a list iterator on siblings of the different tries. More precisely, if we are processing attribute A_ℓ, the method works exactly like Leapfrog on the unary relations

$$\pi_{A_\ell}\left(\sigma_{A_1 \doteq a_1, \ldots, A_{\ell-1} \doteq a_{\ell-1}}(R)\right) .$$

The Triejoin class maintains as internal records a number of trie iterators $I_1, \ldots, I_n$, which are the iterators over the tries that represent $R_1, \ldots, R_n$ respectively. A set of list iterators $H_1, \ldots, H_m$ is used to perform Leapfrog on lists of siblings. Then, it uses a number $\ell \in \mathbb{N}$ to remember its current depth or, equivalently, the attribute A_ℓ it is currently processing. Finally, the class use sets of integers $S_1, \ldots, S_m$ for maintaining, for each $i \in [m]$, the set of relations R_k that use the attribute A_i.

The main work in Algorithm 14 lies in the function open() on line 5. This function first moves one level deeper by increasing ℓ, then computes the set S_ℓ representing the relations that use attribute A_ℓ, calls open() on all iterators that have the attribute A_ℓ, and then initializes the list iterator H_ℓ using the Leapfrog constructor on the iterators that have the attribute A_ℓ. (Here, we use the notation $(I_k)_{k \in S_\ell}$ to denote the tuple $(I_{j_1}, \ldots, I_{j_p})$ where $S_\ell = \{j_1, \ldots, j_p\}$ and $j_1 < \cdots < j_p$.)

Example 27.11: Triejoin Algorithm

Consider the tries from Figure 27.2a representing relations R_1, R_2, and R_3 (from left to right), respectively.

The Triejoin algorithm (Algorithm 15) first builds a trie iterator I with the constructor of the Triejoin class (Algorithm 14) with the initialized trie iterators for T_1, T_2, and T_3. As such, these initialized iterators are stored in the internal fields I_1, I_2, and I_3 of I.

The first operation that is performed on I is I.open(), which calls I_1.open() and I_3.open(), because T_1 and T_3 are the only tries that have values for attribute A_1. Then, I.next() is called, which performs the leapfrog algorithm to find the next common value at depth 1 of T_1 and T_3, which is 6.

Subsequently, I.open() is called again, which calls I_2.open() and I_3.open(); the iterators for tries with values for A_2. As such, the internal list iterator H_2 will iterate through the lists $\triangleright$ 5 7 $\triangleleft$ and $\triangleright$ 2 5 7 $\triangleleft$ using the Leapfrog algorithm. As soon as the first common value (i.e., 5) is found, the iterator H_2 is paused and we move to depth 3 for the first time.

Algorithm 14 Triejoin class

Input: Trie iterators $J_1, \ldots, J_n$ for relations $R_1, \ldots, R_n$ over attributes $A_1, \ldots, A_m$. The trie iterators need to have the attributes ordered according to the same total order $A_1 \prec A_2 \prec \cdots \prec A_m$

Provides: Trie iterator for $R_1 \bowtie \cdots \bowtie R_n$

```
 1: internal records:
      • Trie iterators $I_1, \ldots, I_n$
      • List iterators $H_1, \ldots, H_m$
      • Integer $\ell$
      • Sets of integers $S_1, \ldots, S_m$
 2: constructor Triejoin($J_1, \ldots, J_n$)
 3:     $(I_1, \ldots, I_n) := (J_1, \ldots, J_n)$
 4:     $\ell := 0$
 5: function open()
 6:     $\ell := \ell + 1$
 7:     $S_\ell := \{k \mid A_\ell \in \mathbf{S}(R_k)\}$                    ▷ S is the schema
 8:     for every $k \in S_\ell$ do
 9:         $I_k$.open()
10:     $H_\ell :=$ Leapfrog($(I_k)_{k \in S_\ell}$)    ▷ $(I_k)_{k \in S_\ell}$: tuple containing the $I_k$ with $k \in S_\ell$
11: function value()
12:     return $J_\ell$.value()
13: function next()
14:     $H_\ell$.next()
15: function up()
16:     for every $k \in S_\ell$ do
17:         $I_k$.up()
18:     $\ell := \ell - 1$
19: function seek(value $v$)
20:     $H_\ell$.seek($v$)
```

Algorithm 15 Triejoin

Input: Initialized trie iterators $I_1, \ldots, I_n$ for the tries representing relations $R_1, \ldots, R_n$. The tries need to have their attributes ordered according to the same total order $A_1 \prec A_2 \prec \cdots \prec A_m$

Output: $R_1 \bowtie \cdots \bowtie R_n$

```
21: $I :=$ Triejoin($I_1, \ldots, I_n$)
22: Trie-Enumerate($I$, ())
```

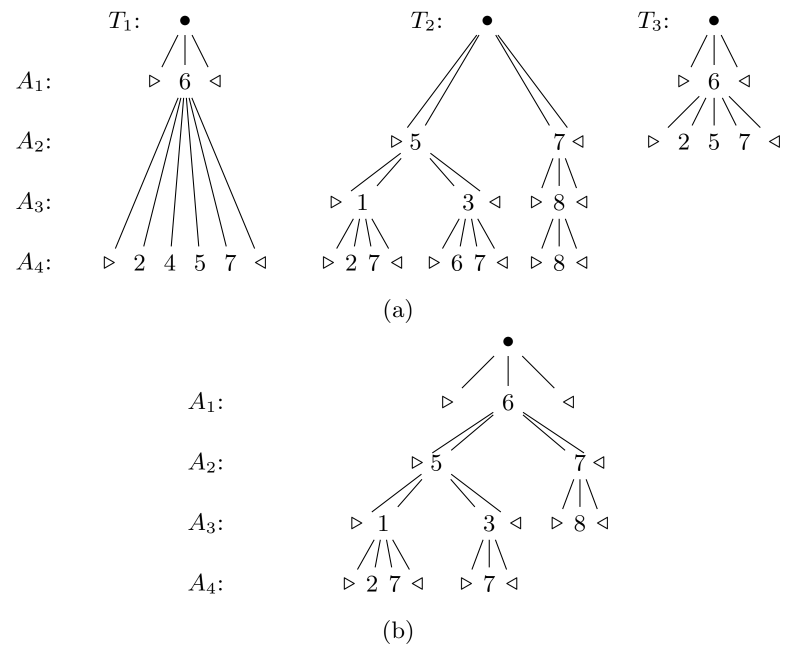


Fig. 27.2: Input tries (27.2a) and output trie (27.2b) for Example 27.11

For moving to depth 3, we only call $I_2.\mathrm{open}()$. The Leapfrog algorithm on depth 3 at this point is very easy, since there is only one list, namely $\triangleright$ 1 3 $\triangleleft$. After the first value (i.e., 1) is found, the iterator H_3 is paused and we move to depth 4 for the first time.

To this end, we call $I_1.\mathrm{open}()$ and $I_2.\mathrm{open}()$ and our task is to do the Leapfrog algorithm on the lists $\triangleright$ 2 4 5 7 $\triangleleft$ and $\triangleright$ 2 7 $\triangleleft$ using list iterator H_4. Here, Leapfrog proceeds all the way to the end of the lists, since we arrived at the leafs. After finding the common values 2 and 7, we call $I_1.\mathrm{up}()$ and $I_2.\mathrm{up}()$ and resume the work of iterator H_3.

When iterator H_3 finds the next value (i.e., 3), we again move to depth 4, but this time on the lists $\triangleright$ 2 4 5 7 $\triangleleft$ and $\triangleright$ 6 7 $\triangleleft$.

Essentially, the operation of the Triejoin algorithm, does a depth-first left-to-right pass over the "result trie" of the join, which is the trie in Figure 27.2b. In fact, if we initialize the Triejoin class as we did in this example, it produces a trie iterator for the trie in Figure 27.2b.

Notice that this trie can contain only partially complete tuples, like $(A_1 : 1, A_2 : 7, A_3 : 8)$. These are tuples that we were able to join on the first three attributes, but not on the fourth one.

We state the following result without proof.

Theorem 27.12

Let D be a database, $q = R_1 \bowtie \cdots \bowtie R_n$ be a join query using m attributes, and $(x_1, \ldots, x_m)$ be a fractional edge cover of q. Then Leapfrog Triejoin computes $q(D)$ in time

$$\tilde{O}\left(n \cdot m \cdot \prod_{i=1}^{n} |R_j^D|^{x_i}\right) .$$

Exercises

Exercise 3.1. The goal of this exercise is to prove Proposition 18.6 in several steps.

(a) Define an *ear* of a hypergraph $H = (V, E)$ to be an edge $e \in E$ for which there exists a distinct edge e_w such that the nodes of e can be partitioned in two sets:

- nodes that only appear in e and
- nodes that appear in e_w.

We call e_w a *witness* for e. Notice that each edge that is contained in another edge is an ear.
Assume that $|E| = n$. An *ear decomposition* of H is an ordered sequence

$$e_1, \ldots, e_n$$

of the edges in E such that, for each $i \in [1, n-1]$, edge e_i is an ear in the hypergraph $(V, E - \{e_1, \ldots, e_{i-1}\})$.
Prove that a hypergraph has an ear decomposition if and only if the procedure in Proposition 18.6 can delete all the vertices of H.

(b) Prove that a hypergraph H acyclic if and only if it has an ear decomposition. To this end, show that the edges in a join tree of H correspond to the ear/witness relationship in an ear decomposition.

Exercise 3.2. Based on Proposition 18.6, design a polynomial-time algorithm that computes a join tree of an acyclic CQ q.

Exercise 3.3. Prove that $q(D) \ltimes q'(D)$, given $q(D)$ and $q'(D)$, can be computed in time $O((\|q(D)\| + \|q'(D)\|) \cdot \log(\|q(D)\| + \|q'(D)\|))$.

Exercise 3.4. Extend Yannakakis's algorithm to show that the set $q(D)$, for $q(\bar{x})$ an acyclic CQ and D a database, can be computed in time $O(\|D\| \cdot$

$\log \|D\| \cdot \|q\| \cdot \|q(D)\|)$. In addition, if the set of free variables of q is contained in at least one node of the join tree of q, then the latter can be improved to $O(\|D\| \cdot \log \|D\| \cdot \|q\|)$.

Exercise 3.5. Complete the proof of Proposition 19.4.

Exercise 3.6. Complete the proof of Proposition 19.5.

Exercise 3.7. Prove that there are CQs with arbitrarily many atoms that are not acyclic, yet its core is acyclic.

Exercise 3.8. While the class of CQs whose core is acyclic defines an "island of tractability" for CQ evaluation, checking membership into such an island is an intractable problem. In particular, checking whether a CQ has an acyclic core is NP-complete (this is in stark contrast with acyclicity recognition, which can be solved in linear time). You are asked to prove this fact.

Exercise 3.9. Let q, q' be Boolean CQs and D a database. Prove that if there exists a homomorphism from q to q' and $\textsc{Consistency}(q', D) = \texttt{true}$, then $\textsc{Consistency}(q, D) = \texttt{true}$.

Exercise 3.10. Prove Lemma 21.5.

Exercise 3.11. Complete the proof of Part 4 of Proposition 21.6.

Exercise 3.12. Let $G_{k \times k}$ be the $(k \times k)$-grid. Prove by induction on $k \geq 3$ that for every $S \subseteq [k] \times [k]$ with $|S| \leq k-1$, there exists a connected component C of $(G_{k \times k} - S)$ with more than $k^2/2$ elements.

Exercise 3.13. Prove that $\text{tw}(G_{k \times k}) \geq k$.

Exercise 3.14. Complete the proof of Proposition 22.4.

Exercise 3.15. Prove Proposition 22.5.

Bibliographic Comments

(Very preliminary version)

Acyclic database schemas were first defined, named, and studied, in the two papers [14] and [3]. In particular, several desirable properties of acyclic database schemes were identified and studied in [3]. The notion of α-acyclicity of hypergraphs used in Chapter 18 was introduced in [13].

In [29], it is given a linear-time algorithm for checking whether a CQ q is acyclic, and for constructing a join tree of H_q if the latter is the case.

Yannakakis's algorithm was proposed in [33]. The correctness of the Consistency Algorithm was proved in [3], and the fact that this algorithm continues being correct for the class of CQs whose cores are acyclic was shown in [9].

The notion of treewidth was introduced in [25]. Lemmas 21.5 and 21.8, as well as the proof that $\text{tw}(G_{k \times k}) \geq k-1$, were taken from [15]. For a fixed $k \geq 1$, it was shown in [4] the fact that there exists a linear-time algorithm that, given an undirected graph G with $\text{tw}(G) \leq k$, constructs a tree decomposition of G of width at most k. Proposition 21.10 is a corollary of this result.

Part IV

Expressive Languages

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 28.1: 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 $n \geq 1$, with $\mathrm{FV}(\varphi) = \mathrm{FV}(\varphi_i)$ and $\varphi_i(\bar{x})$ being 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 easy 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 28.2: 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

$$\varphi_1 \;=\; \exists x \exists z \big(\text{Person}(x, y, z) \;\wedge\; \text{Profession}(x, \text{`computer scientist'}) \;\wedge\; \text{City}(z, \text{`Athens'}, \text{`Greece'})\big).$$

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, UCQ s 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 28.3

The language of UCQ s 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 12.5): 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 variables 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, $\psi(\bar{x})(D) = \psi'(\bar{x})(D)$ for every database D equipped with the unary relation dom. 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. $\square$

Let $\mathrm{FO}[\wedge, \vee, \exists]$ be the fragment of FO that corresponds to the closure of relational *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 UCQ s, and thus, by Theorem 28.3, 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, $q(D) = \{(a)\}$ for every database D, even if $a \notin \mathrm{Dom}(D)$. However, given a database D' such that $a \notin \mathrm{Dom}(D')$, for every unary UCQ q', it holds that $\{(a)\} \notin q'(D')$ 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 $\exists\mathrm{FO}^+$ query q above uses equality among a variable and a constant. It turns out that this is crucial for showing that $\exists\mathrm{FO}^+$ queries form a strictly more expressive language than UCQ s. Interestingly, the language of queries based on $\mathrm{FO}^{\mathrm{rel,var=}}[\wedge,\vee,\exists]$, 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 UCQ s.

Theorem 28.4

The language of UCQ s and the language of $\mathrm{FO}^{\mathrm{rel,var=}}[\wedge,\vee,\exists]$ queries are equally expressive.

Proof. By definition, a UCQ is trivially an $\mathrm{FO}^{\mathrm{rel,var=}}[\wedge,\vee,\exists]$ query. It remains to show that an $\mathrm{FO}^{\mathrm{rel,var=}}[\wedge,\vee,\exists]$ 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 as before 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 $\mathrm{FO}^{\mathrm{rel}}[\wedge,\vee,\exists]$ 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 28.3, $\mathsf{dom}(\cdot)$ atoms can be eliminated with the help of disjunction, we can convert $\varphi'(\bar{x}')$ into an $\mathrm{FO}^{\mathrm{rel}}[\wedge,\vee,\exists]$ query $\varphi''(\bar{x}')$ that is equivalent to $\varphi(\bar{x})$ over *all* databases. Finally, by Theorem 28.3, we know that there exists a UCQ that is equivalent to $\varphi''(\bar{x}')$, and thus to $\varphi(\bar{x})$, and the claim follows. $\square$

We have seen that UCQ s are not powerful enough for expressing every $\exists\mathrm{FO}^+$ query. We have also seen that the key reason for this is the fact that UCQ s, although can express equality among variables, cannot express equality among variables and constants. The question that comes up is whether the addition of equality among variables and constants to UCQ s leads to a language that can express every $\exists\mathrm{FO}^+$ 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 28.3 and 28.4, it is easy to show the following:

Theorem 28.5

The language of UCQ s with variable-constant equality and the language of $\exists\mathrm{FO}^+$ 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 UCQ s and $\mathrm{FO}^{\mathrm{rel}}[\wedge, \vee, \exists]$ (or $\mathrm{FO}^{\mathrm{rel},\mathrm{var}=}[\wedge, \vee, \exists]$) 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 UCQ s with variable-constant equality and $\exists\mathrm{FO}^+$ queries.

Union of Conjunctive Queries as a Fragment of RA

We know, by Theorem 12.7, 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 UCQ s.

Theorem 28.6

The language of UCQ s 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 12.7, 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:

$$\begin{aligned}
\sigma_\theta(e_1 \cup e_2) &\rightsquigarrow \sigma_\theta(e_1) \cup \sigma_\theta(e_2)\\
\pi_\alpha(e_1 \cup e_2) &\rightsquigarrow \pi_\alpha(e_1) \cup \pi_\alpha(e_2)\\
e_1 \times (e_2 \cup e_3) &\rightsquigarrow (e_1 \times e_2) \cup (e_1 \times e_3).
\end{aligned}$$

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 12.7. Let

$$\varphi \;=\; \varphi_1 \vee \cdots \vee \varphi_n.$$

- At this point, one may think that the above step leads to the desired UCQ that is equivalent to the SPJU query e. However, there is no guarantee that, for each $i, j \in [n]$ with $i \neq j$, it holds that $\text{FV}(\varphi_i) = \text{FV}(\varphi_j)$. We proceed to 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 $\text{FO}^{\text{rel,var=}}[\wedge, \vee, \exists]$ query that is equivalent to the query e', and thus, to the query e. This suffices to show our claim since, by Theorem 28.4, 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 $\text{FO}^{\text{rel,var=}}[\wedge, \vee, \exists]$ query that is equivalent to e', as needed. □

The following is an immediate corollary of Theorems 28.3, 28.4 and 28.6.

Corollary 28.7

The language of $\text{FO}^{\text{rel}}[\wedge, \vee, \exists]$ (or even $\text{FO}^{\text{rel,var=}}[\wedge, \vee, \exists]$) 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 28.6. 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 28.6, 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 UCQ s.

Theorem 28.8

The language of UCQ s with variable-constant equality and the language of RA^+ queries are equally expressive.

The following is an immediate corollary of Theorems 28.5 and 28.8 that relates the languages of $\exists\mathrm{FO}^+$ queries and RA^+ queries.

Corollary 28.9

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 28.6) 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 28.6, we get an SPJU query that is the union of 2^n SPJ queries.

Preservation Under Homomorphisms

We have already seen that CQs are preserved under homomorphisms (Proposition 13.6). 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 $\bar{b} \in \mathrm{Dom}(D')^k$,

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

It is not difficult to show that preservation under homomorphisms extends to UCQ s with variable-constant equality (and thus, to $\exists\mathrm{FO}^+$ queries).

Proposition 28.10

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 $(D, \bar{a}) \rightarrow_{\mathrm{Dom}(\varphi)} (D', \bar{b})$ 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 13.6, 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 28.11

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 28.10 and Theorem 28.11 is that:

Corollary 28.12

The language of UCQ s 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 28.11 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 UCQ s and acyclic UCQ s; 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 28.13

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 14.1). 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 13.2, 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 28.13, that evaluating UCQ s 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 28.5); the same holds for RA^+ queries (Theorem 28.8). However, the conversion can be very costly; it may take, in general, exponential time. Therefore, knowing that evaluating UCQ s 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, UCQ s 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 18.4). We further know that acyclic CQs can be efficiently evaluated. More precisely, by Theorem 19.3, 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 UCQ s.

Theorem 28.14

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 19.3, 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

$$O\left(||D|| \cdot \log \|D\| \cdot \sum_{i \in [n]} ||q_i||\right) \;=\; O(\|D\| \cdot \log \|D\| \cdot \|q\|)$$

and the claim follows. □

29

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 29.1

Consider two UCQ s $q = q_1 \cup \cdots \cup q_n$ 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. We first show that (1) implies (2). We assume that the output tuple of q and q' is $\bar{x}$ and $\bar{x}'$, respectively. This means that, for each $i \in [n]$ and $j \in [m]$, the output tuple of q_i and q'_j is $\bar{x}$ and $\bar{x}'$, respectively. Consider an arbitrary integer $i \in [n]$. Recall that $\mathsf{G}_{A_{q_i}}$ is a homomorphism from A_{q_i} to the grounding of A_{q_i}, Thus, by Theorem 13.2, we get that $\mathsf{G}_{A_{q_i}}(\bar{x}) \in q_i(\mathsf{G}_{A_{q_i}}(A_{q_i}))$. Since $q \subseteq q'$, we have that $\mathsf{G}_{A_{q_i}}(\bar{x}) \in q'(\mathsf{G}_{A_{q_i}}(A_{q_i}))$, and hence, $\mathsf{G}_{A_{q_i}}(\bar{x}) \in q'_j(\mathsf{G}_{A_{q_i}}(A_{q_i}))$ for some $j \in [m]$. As in the proof of the Homomorphism Theorem (Theorem 15.4), we can show that there exists a homomorphism h from $(q'_j, \bar{x}')$ to $(q_i, \bar{x})$. From the Homomorphism Theorem itself, we conclude then that $q_i \subseteq q'_j$.

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 29.2

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 15.3). Consider now two UCQ s $q = q_1 \cup \ldots \cup q_n$ 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 15.4) and Proposition 29.1, 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 29.2 is that the equivalence problem for UCQ s, that is, given two UCQ s 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 15.8).

Corollary 29.3

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 UCQ s 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 $e_1 \cup e_2$ 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 29.4

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 29.4 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 15.3. 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 $\perp$. The obtained tree T'_e is essentially the parse tree of an SPJ query from $\mathrm{SPJ}(e)$. Consider, for example, the query

$$e \;=\; (e_1 \cup e_2) \times (e_3 \cup (e_4 \cup e_5)),$$

[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.

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 $e_2 \times e_5 \in \mathrm{SPJ}(e)$. 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. However, each way to resolve the union nodes in T_e can be carried out in polynomial time, or, in other words, we can guess any SPJ query from $\mathrm{SPJ}(e)$ 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 29.5

Consider an SPJ query e, and an SPJU query e'. The problem of deciding whether $e \subseteq e'$ is in NP.

Proof. By Proposition 29.4, it suffices to show that checking whether there exists an SPJ query $e'' \in \mathrm{SPJ}(e')$ such that $e \subseteq e''$ is in NP. This is done by guessing an SPJ query e'' from $\mathrm{SPJ}(e')$, and a mapping $h : \mathrm{Dom}(A_{q_{e''}}) \to \mathrm{Dom}(A_{q_e})$, where $q_e(\bar{x})$ and $q_{e''}(\bar{y})$ are the CQs obtained after converting e and e'' into CQs by applying the translation given in Theorem 12.7. It is easy to see that such translation can be carried out in polynomial time. We finally verify 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 15.4). Since both steps are feasible in polynomial time (recall that an SPJ query from $\mathrm{SPJ}(e)$ can be guessed in polynomial time), we conclude that checking whether $e \subseteq e'$ is in NP, and the claim follows. □

Proposition 29.5 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 29.6

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 $\Sigma_2^p = \mathrm{NP}^{\mathrm{NP}}$, which in turn implies that SPJU-Containment is in Π_2^p. By Proposition 29.4, it suffices to show that the problem of checking whether there exists an SPJ query $\hat{e} \in \mathrm{SPJ}(e)$ such that $\hat{e} \not\subseteq e'$ is in $\mathrm{NP}^{\mathrm{NP}}$. This is done by simply guessing an SPJ query $\hat{e}$ from $\mathrm{SPJ}(e)$, 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 29.5, 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 ∀∃QSAT (Exercise 4.2). □

An immediate corollary of Theorem 29.6 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, it holds that $e \subseteq e'$ iff $\pi_{(1,\ldots,k)}(e \bowtie_\theta e') \equiv e$, where $\theta = (1 \doteq k+1) \wedge (2 \doteq k+2) \wedge \cdots \wedge (k \doteq 2k)$. Therefore:

Corollary 29.7

SPJU-Equivalence is Π_2^p-complete.

Minimization

In Chapter 16, 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 $\mathbf{S}$ with $q' \equiv q''$ it holds that $|A_{q'}| \leq |A_{q''}|$. 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 UCQ s. We start by defining the notion of minimization for UCQ s.

Definition 29.8: 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 $\mathbf{S}$, if $q'_i \equiv p$ then $|A_{q'_i}| \leq |A_p|$, and

3. for every UCQ $q'' = q''_1 \cup \cdots \cup q''_\ell$, if $q' \equiv q''$ then $m \leq \ell$.

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 UCQ s 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 UCQ s, which is a generalization of the notion of core for CQs given in Definition 16.2.

Definition 29.9: 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 29.9 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 29.10

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 16.4, 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 29.9) 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

Algorithm 16 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:     $Q := Q \cup \{\text{ComputeCore}(q_i)\}$
4: while there are distinct CQs $q', q'' \in Q$ such that $q' \subseteq q''$ do
5:     $Q := Q - \{q'\}$
6: return $q^* = \bigcup_{q' \in Q} q'$
```

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, we have that q'_i is not a core of itself, and hence q'_i cannot be a core of a CQ in q. This contradicts the fact that q' is a core of q. Assume now (2). Since $q' \subseteq q''$, by Proposition 29.1 we get that, for every $i \in [m]$, there exists $j \in [\ell]$ such that $q'_i \subseteq q''_j$. Then, from the fact that $\ell < m$, we obtain that there exist $i, i' \in [m]$, 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 29.1 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 29.10, 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 16, 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 29.11. *Given a* UCQ q*,* ComputeCoreUCQ(q) *is a core of* q*.*

Let us 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 29.10, that different cores of q are isomorphic.

Uniqueness of Minimization

We say that two UCQ s $q(\bar{x}) = q_1 \cup \cdots \cup q_n$ 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 $i \in [n]$ the CQs q_i and $\delta(q_i)$ are isomorphic.

Proposition 29.12

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 $i \in [n]$ the CQs q'_i and $\delta(q'_i)$ are isomorphic. We first show an auxiliary lemma:

Lemma 29.13. *There is a bijection $\tau : [n] \to [n]$ such that $q'_i \equiv q''_{\tau(i)}$, for each $i \in [n]$.*

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 $j, k \in [n]$, if $q'_i \equiv q''_j$ and $q'_i \equiv q''_k$ then $j = k$.

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 29.1, we get that, for each $i \in [n]$, there exists $j \in [n]$ such that $q'_i \subseteq q''_j$. But, again by Proposition 29.1, 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 conclude then that $q'_i \equiv q''_j$.

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 29.13. □

Having the bijection $\tau : [n] \to [n]$ provided by Lemma 29.13, we define the bijection $\delta : \{q'_1, \dots, q'_n\} \to \{q''_1, \dots, q''_n\}$ as $\delta(q'_i) = q''_{\tau(i)}$, for each $i \in [n]$. It remains to argue that, for every $i \in [n]$, the CQs q'_i and $\delta(q'_i)$ are isomorphic. Since q' and q'' are minimizations of q, both q'_i and $\delta(q'_i)$ must be cores of themselves. Hence, $\delta(q'_i)$ is a minimization of q'_i, and thus, by Proposition 16.7, we get that q'_i and $\delta(q'_i)$ are isomorphic. □

From Proposition 29.10, which tells us that a core of a UCQ q is a minimization of q, and Proposition 29.12, we get the following corollary:

Corollary 29.14

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 29.14 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.

30

Unions of Conjunctive Queries with Inequalities

It is not difficult to show that UCQ s, or even UCQ s 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 $D \rightarrow_{\emptyset} D'$, but $D \models q$ while $D' \not\models q$. On the other hand, we know by Proposition 28.10 that UCQ s (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 there are languages obtained by adding a tamed negation to UCQ s without increasing the complexity of query evaluation, and, more importantly, without losing the decidability of containment and equivalence. The best known such language is that of UCQ *s with inequality*, that is, UCQ s 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 UCQ s with inequality, and study the main computational problems: evaluation and containment.

Conjunctive Queries with Inequality

Before we introduce and study UCQ s 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 30.1: Syntax of CQs with Inequality

A *conjunctive query with inequality* (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 CQ$^{\neq}$ $\varphi(\bar{x})$ given in Definition 12.1 can be written as the *rule*

$$\text{Answer}(\bar{x}) \ \ {:}{-}\ R_1(\bar{u}_1), \ldots, R_n(\bar{u}_n), v_1 \neq v_1', \ldots, v_m \neq v_m' \,,$$

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 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. Nevertheless, for convenience, we will freely interpret a CQ with inequality as an FO query or as a rule.

Semantics of Conjunctive Queries with Inequality

Since a 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 CQ$^{\neq}$ $q = \varphi(\bar{x})$ over $\mathbf{S}$, where $k \geq 0$, the *output* of q on D is

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

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 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 $q = \varphi(y)$ 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 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 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 CQ$^{\neq}$ on a database.

Definition 30.2: Evaluation of CQs with Inequality

Given a database D of a schema $\mathbf{S}$, and a 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 CQ$^{\neq}$ inherited from the semantics of FO queries in Definition 3.6, and the semantics of CQs given in Definition 30.2, 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 CQ$^{\neq}$ q of the form

$$\mathrm{Answer}(\bar{x}) \ \text{:–}\ R_1(\bar{u}_1), \ldots, R_n(\bar{u}_n), v_1 \neq v_1', \ldots, v_m \neq v_m' ,$$

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, $\mathrm{Dom}(A_q^+)$ collects all the variables and constants occurring in the relational atoms of A_q^+. We further write $\mathrm{Dom}(A_q^-)$ for the set of constants occurring in A_q^-. Notice that there may be constants in $\mathrm{Dom}(A_q^-)$ that do not occur in $\mathrm{Dom}(A_q^+)$ since, according to Definition 30.1, 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 30.3: Homomorphisms from CQ$^{\neq}$ to Databases

Consider a CQ$^{\neq}$ $q(\bar{x})$ over a schema $\mathbf{S}$, and a database D of $\mathbf{S}$.

- A *homomorphism from q to D* is a function $h : \mathrm{Dom}(A_q^+) \cup \mathrm{Dom}(A_q^-) \to \mathrm{Dom}(D) \cup \mathrm{Dom}(A_q^-)$ that is a homomorphism from A_q^+ to D, is the identity on $\mathrm{Dom}(A_q^-)$, and $h(v) \neq h(u)$ for every $v \neq u \in A_q^-$. We write $q \to D$ if such a homomorphism exists.
- A *homomorphism from $(q, \bar{x})$ to $(D, \bar{a})$* is a homomorphism h from q to D such that $h(\bar{x}) = \bar{a}$. We write $(q, \bar{x}) \to (D, \bar{a})$ if such a homomorphism exists.

To define the output of a CQ$^{\neq}$ $q(\bar{x})$ on a database D (see Definition 30.2), we used the notion of assignment for q over D, which is a function from the set of variables in q to $\mathrm{Dom}(D)$. 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., $R_i(\eta(\bar{u}_i)) \in D$ and $\eta(v_j) \neq \eta(v_j')$ for each $i \in [n]$ and $j \in [m]$. Since $R_i(\eta(\bar{u}_i))$ 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, for $i \in [n]$, such an assignment η corresponds to a function $h : \mathrm{Dom}(A_q^+) \cup \mathrm{Dom}(A_q^-) \to \mathrm{Dom}(D) \cup \mathrm{Dom}(A_q^-)$, 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 $\mathrm{Dom}(D)$ with $(q, \bar{x}) \to (D, \bar{a})$. This leads to an alternative characterization of CQ$^{\neq}$ evaluation.

Theorem 30.4

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

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

Unlike plain CQs, CQs with inequality are not preserved under homomorphisms. This has been already illustrated at the beginning of the chapter via the 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 CQ$^{\neq}$ q over a schema $\mathbf{S}$, and two databases D and D' of $\mathbf{S}$, if $D \subseteq D'$ then $q(D) \subseteq q(D')$.

Proposition 30.5

Every CQ$^{\neq}$ is monotone.

Proof. Let $q(\bar{x})$ be a 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 30.4, 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 30.4, $\bar{a} \in q(D')$, as needed. □

The fact that CQs with inequality are monotone allows us to clarify the expressiveness boundaries of CQ$^{\neq}$. In particular, we can show that the negation allowed in CQ$^{\neq}$ is indeed quite restricted. Note that the following arguments have been already used in Chapter 13 to show that CQs cannot express negative relational atoms and difference.

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 $D' \not\models q$.

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 CQ$^{\neq}$-Evaluation, that is, the problem of checking whether $\bar{a} \in q(D)$ for a CQ$^{\neq}$ query q, a database D, and a tuple $\bar{a}$ over Dom(D). We actually show that it has the same (combined) complexity as CQ-Equivalence; concerning the data complexity, it is clear that CQ$^{\neq}$-Evaluation is in DLogSpace due to Theorem 7.3.

Theorem 30.6

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 14.1). 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 30.4, $\bar{a} \in q(D)$ 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 15.4) 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 30.7: The Homomorphism Theorem Fails for $\mathrm{CQ}^{\neq}$

Consider the (Boolean) queries

$$\begin{aligned} q &= \text{Answer} \; \text{:--}\; \text{Edge}(x, y) \\ q' &= \text{Answer} \; \text{:--}\; \text{Edge}(x', y'), x' \neq y'. \end{aligned}$$

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 $D = \{R(a, a)\}$ we have that $D \models q$ 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, as defined for these queries, do not compose. In other words, the fact that $q' \to q$ and $q \to D$, for q, q' CQs with inequalities, 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 30.3).

Given a CQ$^{\neq}$ $q(\bar{x})$ of the form

$$\text{Answer}(\bar{x}) \ \text{:-}\ R_1(\bar{u}_1), \ldots, R_n(\bar{u}_n), v_1 \neq v_1', \ldots, v_m \neq v_m' \,,$$

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 into a homomorphic image of itself. For a function $h \in H_q$, we write $h(q^+)$ for the CQ (without inequality)

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

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 30.8

Let $q(\bar{x})$ and $q'(\bar{x}')$ be CQ$^{\neq}$s. The following are equivalent:

1. $q \subseteq q'$.
2. $(q', \bar{x}') \to (h(q^+), h(\bar{x}))$, for each $h \in H_q$.

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, we have that 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 $\text{Dom}(A_{q'}^-) - \text{Dom}(A_p)$, is a homomorphism from $(q, \bar{x})$ to $(\mathsf{G}_{A_p}(A_p), \mathsf{G}_{A_p}(h(\bar{x})))$. By Theorem 30.4, $\mathsf{G}_{A_p}(h(\bar{x})) \in q(\mathsf{G}_{A_p}(A_p))$. 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 30.4, 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, $(\mathsf{G}_{A_p}^{-1} \cup \mu') \circ g$, where μ' is the identity on the set of constants $\text{Dom}(A_q^-) - \text{Dom}(A_p)$, 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 30.4, there exists a homomorphism g from $(q, \bar{x})$ to $(D, \bar{a})$. Let h be a function from $\text{Dom}(A_q^+) \cup \text{Dom}(A_q^-)$ 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
- $h(x)$ is a variable, for every variable $x \in \mathrm{Dom}(A_q^+)$.

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 $(h(q^+), h(\bar{x})) \to (D, \bar{a})$ is 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 30.4, we get that $\bar{a} \in q'(D)$, and the claim follows. □

The next example illustrates how Theorem 30.8 is used in order to confirm what has been discussed in Example 30.7.

Example 30.9: Application of Theorem 30.8

Consider again the CQs q and q' given in Example 30.7, and recall that $q \not\subseteq q'$. This is confirmed by Theorem 30.8 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 30.8, we immediately get a procedure for CQ$^{\neq}$-Containment: given two CQs with inequality $q(\bar{x})$ and $q'(\bar{x}')$, return `true` if $(q', \bar{x}) \to (h(q^+), h(\bar{x}))$, for every function $h \in H_q$; otherwise, return `false`. However, this naive approach only shows that 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 30.10

CQ$^{\neq}$-Containment is Π_2^p-complete.

Proof. Consider two 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 CQ$^{\neq}$-Containment is in Π_2^p. By Theorem 30.8, 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 30.11. *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 30.11, 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 NP^{NP}, as needed.

For the Π_2^p-hardness of CQ$^{\neq}$-Containment see Exercise 4.4. □

With Theorem 30.10 in place, we can easily pinpoint the complexity of the equivalence problem for CQ$^{\neq}$: given two 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 30.12

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 CQ$^{\neq}$-Equivalence is in Π_2^p since, by Theorem 30.10, 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 CQ$^{\neq}$-Containment that is Π_2^p-hard (Theorem 30.10); in fact, this holds even for Boolean queries. Given two Boolean CQ$^{\neq}$ q, q', we can easily construct a 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 15.8 that deals with CQ-Equivalence. Since $q \subseteq q'$ iff $q \equiv q_\cap$, we get that CQ$^{\neq}$-Equivalence is Π_2^p-hard, and the claim follows. □

Adding Union to CQ$^{\neq}$

We now proceed to add the union operator to CQs with inequality.

Definition 30.13: 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 of the form

$$\varphi_1 \vee \cdots \vee \varphi_n$$

for $n \geq 1$, $\text{FV}(\varphi) = \text{FV}(\varphi_i)$, and $\varphi_i(\bar{x})$ is a CQ$^{\neq}$ for every $i \in [n]$.

As for UCQ s 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 $\mathbf{S}$, we have that $q(D) = q_1(D) \cup \cdots \cup q_n(D)$.

Example 30.14: 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

$$\varphi_1 \;=\; \exists x \exists z \, \big(\text{Person}(x, y, z) \;\wedge\; \text{Profession}(x, \text{`computer scientist'}) \;\wedge\; \text{City}(z, w, \text{`Greece'}) \;\wedge\; w \neq \text{`Athens'}\big).$$

and

$$\varphi_2 \;=\; \exists x \exists z \, \big(\text{Person}(x, y, z) \;\wedge\; \text{Profession}(x, \text{`computer scientist'}) \;\wedge\; \text{City}(z, w, \text{`Chile'}) \;\wedge\; w \neq \text{`Santiago'}\big).$$

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 $\mathsf{UCQ}^{\neq}$-$\mathsf{Evaluation}$, the problem of checking whether $\bar{a} \in q(D)$ for a UCQ$^{\neq}$ q, a database D, and a tuple $\bar{a}$ over Dom(D). We show that it has the same (combined) complexity as $\mathsf{CQ}^{\neq}$-$\mathsf{Evaluation}$; concerning the data complexity, both problems are in DLogSpace due to Theorem 7.3.

Theorem 30.15

$\mathsf{UCQ}^{\neq}$-$\mathsf{Evaluation}$ is NP-complete.

Proof. It is clear that the NP-hardness is inherited from CQ-Evaluation, which we know is NP-hard (Theorem 14.1). 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 $\bar{a} \in \text{Dom}(D)$. By Theorem 30.4, 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 $h : \text{Dom}(A^+_{q_i}) \cup \text{Dom}(A^-_{q_i}) \to \text{Dom}(D) \cup \text{Dom}(A^-_{q_i})$, 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 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 end, we first observe that Proposition 29.1 for UCQ s (without inequality) holds also for UCQ s with inequality. More precisely, the proof of Proposition 29.1 relies on 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 UCQ s with inequality, the same proof shows the following:

Proposition 30.16

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 30.17

UCQ$^{\neq}$-Containment is Π_2^p-complete.

Proof. It is clear that the Π_2^p-hardness is inherited from CQ$^{\neq}$-Containment, which we know is Π_2^p-hard (Theorem 30.10). 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 $\Sigma_2^p = \text{NP}^{\text{NP}}$, which in turn implies that UCQ$^{\neq}$-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 30.8 and Proposition 30.16, 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 30.11, 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 NP^{NP}, as needed. □

An immediate corollary of Theorem 30.17 is that the equivalence problem for UCQ s with inequality, that is, given two 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 CQ$^{\neq}$-Equivalence (Theorem 30.12).

Corollary 30.18

UCQ$^{\neq}$-Equivalence is Π_2^p-complete.

31

The Limits of First-Order Queries: Recursion

We have seen that the language of UCQ s has the same expressive power as the language of $\mathrm{FO}^{\mathrm{rel}}[\wedge, \vee, \exists]$ queries (Theorem 28.3), and that adding equational atoms $x = a$ between variables and constants leads to the strictly more expressive language of $\exists\mathrm{FO}^+$ queries (Theorem 28.5). If we further add negation to $\exists\mathrm{FO}^+$ queries, we then get the full power of FO queries. 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, this is how SQL typically expresses universal quantification, as it lacks explicit statements for it.

We have already learned some interesting facts about the language of FO queries. Theorem 6.1 states that it has the same expressive power as the language of RA queries. 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).

In the next two chapters, we address a different type of questions about the language of FO queries: what are the limitations of its expressive power? In other words, what kinds of queries FO cannot express. Knowing this justifies what practical languages need to add on top of the language of FO queries.

In this chapter, we study the inexpressibility in FO, and thus in RA, of queries requiring recursive computation. Note that recursion is the subject of Chapters 35 – 38. A canonical query of this type is reachability in directed graphs. Given a directed graph G, this query computes all the pairs of nodes (u, v) in G such that v is reachable from u. It 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 $\{0, \dots, n\}$ 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.

To show that such queries cannot be defined in FO, we present a fundamental property of FO queries, that is, *locality*. Intuitively, locality means that FO queries can only talk about objects that they see in a fixed neighbourhood of their free variables (this will be made precise below). This will let us prove that we cannot express queries such as reachability in directed graphs using FO queries. In particular, since seeing if there exists a path from one node to another may involve paths of arbitrary length, such a query is not local.

Notion of Locality and Its Use

To build towards the notion of locality, we need to define some auxiliary terminology. Given a 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 the sets $\{a, b\}$ such that a and b appear together in some fact of D, i.e., there is a fact of the form $R(\dots, a, \dots, b, \dots)$ in D. Given $a, b \in \mathrm{Dom}(D)$, the *distance* of a and b in D, denoted by $d_D(a, b)$, 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 $\bar{a} = (a_1, \dots, a_n)$ 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 $N_r^D(\bar{a})$. Finally, 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 $R(\bar{c}) \in N_r^D(\bar{a})$ if and only if $R(h(\bar{c})) \in N_r^D(\bar{b})$. We can now define locality of queries.

Definition 31.1: Locality of Queries

A k-ary query q over schema $\mathbf{S}$ is *r-local*, for $r \geq 0$, if, for every database D of $\mathbf{S}$, and every two tuples $\bar{a}, \bar{b} \in \mathrm{Dom}(D)^k$ 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 $r \geq 0$.

Interestingly, we can show that FO queries are local. In fact, we can show more than that; we can further connect the number r witnessing the locality of an FO query to the depth of quantifier nesting in formulae, defined below.

Definition 31.2: Quantifier Rank of FO Formulae

The *quantifier rank* of an FO formula is inductively defined as follows:

- $\mathsf{rank}(\varphi) = 0$ if φ is atomic.
- $\mathsf{rank}(\varphi \vee \psi) = \mathsf{rank}(\varphi \wedge \psi) = \max\{\mathsf{rank}(\varphi), \mathsf{rank}(\psi)\}$.
- $\mathsf{rank}(\neg\varphi) = \mathsf{rank}(\varphi)$.
- $\mathsf{rank}(\exists x\, \varphi) = \mathsf{rank}(\forall x\, \varphi) = \mathsf{rank}(\varphi) + 1$.

Note that $\mathsf{rank}(\varphi)$ is not the total number of quantifiers, but rather the depth of their nesting, that is, the largest number of quantifiers one encounters in a branch of a parse tree of φ. For example, $\mathsf{rank}(\exists x\, S(x) \vee \forall y\, \neg R(y, y))$ is 1 and not 2. The formal statement about the locality of FO queries follows.

Theorem 31.3

Every FO query $q = \varphi(\bar{x})$ is r-local for $r = (3^{\mathsf{rank}(\varphi)} - 1)/2$.

Before we give the proof of Theorem 31.3, let us explain how it is used in order to obtain inexpressibility results for FO queries. Specifically, we look at the already seen *reachability* query, defined formally below.

Definition 31.4: The Reachability Query

The *reachability query*, denoted q_{reach}, over a schema $\mathbf{S} = \{E[2]\}$ is such that, for every database D of $\mathbf{S}$ and $a, b \in \mathrm{Dom}(D)$, $(a, b) \in q_{\mathsf{reach}}(D)$ if

- $E(a, b) \in D$, or
- there are constants $c_1, \ldots, c_n \in \mathrm{Dom}(D)$, for $n > 0$, such that $\{E(a, c_1), E(c_1, c_2), \ldots, E(c_{n-1}, c_n), E(c_n, b)\} \subseteq D$.

With Theorem 31.3 in place, to show that the reachability query cannot be expressed as an FO query, that is, there is no FO query q over $\{E[2]\}$ such that $q_{\mathsf{reach}} \equiv q$, it suffices to show that q_{reach} is not local.

Proposition 31.5

The reachability query q_{reach} over $\mathbf{S} = \{E[2]\}$ is not local.

Proof. By contradiction, we assume that q_{reach} is r-local, for some $r > 0$. Consider a database D of $\mathbf{S}$ consisting of the facts $E(i, i+1)$ for each $i \in [0, 5r - 1]$. In the output of q_{reach} on D we have all the pairs (i, j) with $i < j$. Note that the r-neighborhoods $N_r^D(r, 4r)$ and $N_r^D(4r, r)$ are isomorphic.

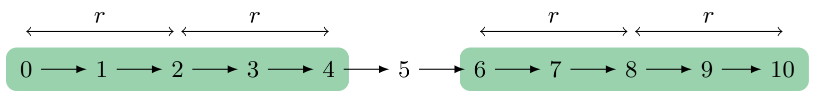


Fig. 31.1: Illustration of locality (with $r = 2$)

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. This is illustrated in Figure 31.1, where $E(i, j)$ is visualized using an arrow from i to j. Therefore, by locality,

$$(r, 4r) \in q_{\mathsf{reach}}(D) \quad \Longleftrightarrow \quad (4r, r) \in q_{\mathsf{reach}}(D)\,.$$

However, $(r, 4r) \in q_{\mathsf{reach}}(D)$ but $(4r, r) \notin q_{\mathsf{reach}}(D)$, since $r < 4r$, which leads to a contradiction. This implies that q_{reach} is not local, as needed. □

Proving Locality of FO

We proceed with the proof of Theorem 31.3. For clarity of the presentation, we assume that the FO query $q = \varphi(\bar{x})$ is over a schema $\mathbf{S}$ that consists of a binary relation R, but of course the proof generalizes to arbitrary schemas. We proceed by induction on the quantifier rank of φ.

Assume first that $\mathsf{rank}(\varphi) = 0$. This implies that φ is an atomic formula $x = y$ or $R(x, y)$. 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 means that the same atomic formulae are true about $\bar{a}$ and $\bar{b}$ since the neighborhood does not contain any other elements, and thus, $\bar{a} \in q(D)$ iff $\bar{b} \in q(D)$. Therefore, q is 0-local, as needed.

Assume now that $\mathsf{rank}(\varphi) = k$ for $k > 0$. We first observe that conjunction and disjunction, as well as negation, do not alter locality. More precisely, if both φ and ψ are r-local, then so are $\varphi \vee \psi$ and $\varphi \wedge \psi$, as well as $\neg\varphi$. Notice also that conjunction, disjunction and negation do not alter the quantifier rank. Thus, the induction step should treat the case where $\varphi(\bar{x}) = \exists y\, \psi(\bar{x}, y)$; we do not need to explicitly treat the case $\varphi(\bar{x}) = \forall y\, \psi(\bar{x}, y)$ since $\forall y\, \psi(\bar{x}, y)$ is equivalent to $\neg\exists y\, \neg\psi(\bar{x}, y)$, which has the same quantifier rank. Since $\mathsf{rank}(\varphi) = k$, we get that $\mathsf{rank}(\psi) = k - 1$. By induction hypothesis, the query $\psi(\bar{x}, y)$ is r-local, where $r = (3^{k-1} - 1)/2$. We proceed to show that $\varphi(\bar{x})$ is $(3^k - 1)/2$-local, or, equivalently, $3r + 1$-local.

Lemma 31.6. *It holds that $\varphi(\bar{x})$ 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 $\bar{a} \in \varphi(\bar{x})(D)$ iff $\bar{b} \in \varphi(\bar{x})(D)$. 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 $c \in \text{Dom}(D)$. Indeed, if such a bijection exists, the claim follows. Assuming that $\bar{a} \in \varphi(\bar{x})(D)$, we can find a witness $c \in \text{Dom}(D)$ of the existential quantifier $\exists y$ such that $(\bar{a}, c) \in \psi(\bar{x}, y)(D)$. But since $N_r^D(\bar{a}, c)$ and $N_r^D(\bar{b}, f(c))$ are isomorphic, $f(c)$ is a witness that $\bar{b} \in \varphi(\bar{x})(D)$, since $(\bar{b}, f(x))$ is in $\psi(\bar{x}, y)(D)$. For the converse, we use the bijection f^{-1} instead of f. The rest of the proof is devoted to showing the existence of the bijection f.

Assume that h is an isomorphism between $N_{3r+1}^D(\bar{a})$ and $N_{3r+1}^D(\bar{b})$. Consider an arbitrary $c \in B_{2r+1}^D(\bar{a})$. Since $B_r^D(c) \subseteq B_{3r+1}^D(\bar{a})$ and $B_D^r(h(c)) \subseteq B_{3r+1}^D(\bar{b})$, we thus obtain, from the fact that h is an isomorphism, that $N_r^D(\bar{a}, c)$ and $N_r^D(\bar{b}, h(c))$ are isomorphic; in fact the same isomorphism h witnesses this. Towards proving the existence of f, we define a *pointed* database as a pair (D', d) for $d \in \text{Dom}(D')$. An *isomorphism type* of pointed databases is the set of pointed databases that are all isomorphic to each other, and no pointed database from outside the set is isomorphic to them. In other words, it is an equivalence class with respect to the equivalence relation of being isomorphic. Note that each neighborhood $N_r^D(c)$ is a pointed database. Let $T_1, \ldots, T_k$ be all the isomorphism types of r-neighborhoods present in D, and let n_i be the number of elements of $c \in \text{Dom}(D)$ such that $N_r^D(c)$ belongs to T_i, for $i \in [k]$. Let $m_i(\bar{a})$ be the number of elements $c \in B_{2r+1}^D(\bar{a})$ such that $N_r^D(c) \in T_i$, and let $m_i(\bar{b})$ be similarly defined for $B_{2r+1}(\bar{b})$. Since $N_r^D(c)$ and $N_r^D(h(c))$ are isomorphic for $c \in B_{2r+1}^D(\bar{a})$, we obtain that $m_i(\bar{a}) = m_i(\bar{b})$, for $i \in [k]$. Now, consider $n_i - m_i(\bar{a})$; it is the number of elements $c \in \text{Dom}(D) - B_{2r+1}(\bar{a})$ such that $N_r^D(c) \in T_i$, and let $n_i - m_i(\bar{b})$ similarly count the number of elements $c \in \text{Dom}(D) - B_{2r+1}(\bar{b})$ such that $N_r^D(c) \in T_i$. Since $m_i(\bar{a}) = m_i(\bar{b})$, we see that $n_i - m_i(\bar{a}) = n_i - m_i(\bar{b})$, for $i \in [k]$. Hence,

$$|\{c \notin B_{2r+1}^D(\bar{a}) \mid N_r^D(c) \in T_i\}| \;=\; |\{c \notin B_{2r+1}^D(\bar{b}) \mid N_r^D(c) \in T_i\}|$$

and thus, there is a bijection $g : \text{Dom}(D) - B_{2r+1}^D(\bar{a}) \to \text{Dom}(D) - B_{2r+1}^D(\bar{b})$ so that $N_r(c)$ and $N_r(g(c))$ belong to the same set T_i, i.e., they are isomorphic.

We now define $f : \text{Dom}(D) \to \text{Dom}(D)$: on $B_{2r+1}^D(\bar{a})$ we let $f(c) = h(c)$, and on $\text{Dom}(D) - B_{2r+1}^D(\bar{a})$ we let $f(c) = g(c)$. For $c \in B_{2r+1}^D(\bar{a})$, we have already seen that $N_r^D(\bar{a}, c)$ and $N_r^D(\bar{b}, f(c))$ are isomorphic. If $c \notin B_{2r+1}^D(\bar{a})$, then $N_r^D(\bar{a}, c)$ is the disjoint union of $N_r^D(\bar{a})$ and $N_r^D(c)$, and $N_r^D(\bar{b}, f(c))$ is the disjoint union of $N_r^D(\bar{b})$ and $N_r^D(f(c))$. Thus, $N_r^D(\bar{a}, c)$ and $N_r^D(\bar{b}, f(c))$ are isomorphic since they are the disjoint union of isomorphic sets. □

This completes the proof of Theorem 31.3.

Other Forms of Locality

Locality as seen in Definition 31.1 is not the only manifestation of this property for FO. A different one, called *Hanf-locality*, is defined in Exercise 4.18. Essentially, it says that if we have two databases, D and D', and and a bijection f between their domains such that $N_r^D(a)$ and $N_r^{D'}(f(a))$ are isomorphic, for an appropriately chosen r (say, 3^k), then no Boolean FO query given by a formula φ of quantifier rank k distinguishes D form D': it is either true in both, or false in both.

A different type of property makes it especially easy to show that some queries are inexpressible in FO. We present it again for graphs, that is, for schemas consisting of a single binary relation name, but it can be stated for arbitrary schemas too. Given a graph, i.e., a database D of the schema $\{E[2]\}$, *degrees* of D are numbers of the form $|\{b \mid E(a,b) \in D\}|$ and $|\{b \mid E(b,a) \in D\}|$. These are essentially out-degrees and in-degrees of graph nodes. We write $\deg(D)$ for the set of all degrees that occur in D. We can then show the following property for binary FO queries over $\{E[2]\}$, i.e., FO queries that take a graph as their input and return a graph as well.

Proposition 31.7

Let q be a binary FO query over $\{E[2]\}$. There is a function $f : \mathbb{N} \to \mathbb{N}$ such that, for every database D of $\{E[2]\}$ and $k > 0$, if all elements of $\deg(D)$ are bounded by k, then $|\deg(q(D))| \leq f(k)$.

To use this to show that q_{reach} cannot be defined as an FO query, we refer again to the database D in the proof of Proposition 31.5 consisting of the facts $E(i, i+1)$ for $i \in [0, n-1]$. In this case, $\deg(D) = \{0, 1\}$. Assume q_{reach} can be expressed as an FO query. By Proposition 31.7, there exists a function $f : \mathbb{N} \to \mathbb{N}$ such that $|\deg(q_{\mathsf{reach}}(D))| \leq f(1)$. However, if $n > f(1)$, this does not hold: the output of $q_{\mathsf{reach}}(D)$ has all edges (i,j) for $0 \leq i < j \leq n$, and thus, $n+1$ different degrees, from 0 to n, are present in it.

32

The Limits of First-Order Queries: Counting

In this chapter, we continue the study of the limitations of FO queries and what these limitations tell us in terms of extending the capabilities of practical query languages, and present two fundamental inexpressibility results concerning constant-free FO 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 32.3), 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 $\mathbf{S} = \{R[1]\}$, that is, $\mathbf{S}$ 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

$$\mu_n(\varphi) \;=\; \begin{cases} 1 & \text{if } D \models \varphi, \text{ for every } D \in \mathrm{Inst}(\mathbf{S}) \text{ with } |D| = n \\ 0 & \text{if } D \not\models \varphi, \text{ for every } D \in \mathrm{Inst}(\mathbf{S}) \text{ with } |D| = n\,. \end{cases}$$

We can then show the following useful technical result:

Proposition 32.1

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 $D \models \varphi$ for every $\varphi \in T$. 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 32.1.

Proof (of Proposition 32.1). 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 32.2. *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 $\varphi, \neg\varphi$. $\square$

We now proceed to show the claim by considering the two cases provided by Lemma 32.2: 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 $|D| \geq k$ it is the case that $D \models \varphi$. 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. $\square$

We can use Proposition 32.1 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 32.3

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 32.1 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 $|D| \geq k$ it is the case that $D \models q_C$.
(ii) For every $D \in \text{Inst}(\mathbf{S})$ with $|D| \geq k$ it is the case that $D \not\models q_C$.

Assuming that (i) holds, there are finitely many integers, let us 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 32.3, 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 32.3 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 32.1 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

$$\mu_n(\varphi) \;=\; \frac{|\{D \in \text{Inst}(\mathbf{S}) \mid \text{Dom}(D) = [n] \text{ and } D \models \varphi\}|}{|\{D \in \text{Inst}(\mathbf{S}) \mid \text{Dom}(D) = [n]\}|}\,.$$

In simple words, $\mu_n(\varphi)$ is the proportion of databases of $\mathbf{S}$ with $\text{Dom}(D) = [n]$ 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 name. 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 $\text{Dom}(D) = [n]$. 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 32.4: 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 $\text{Dom}(D) = [n]$. Therefore, for a constant-free FO sentence φ over $\mathbf{S}$, Proposition 32.1 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 query φ_{EVEN}, expressed in a logical formalism that goes beyond first-order logic, 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 32.5: 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., $\mathbf{S} = \{R[1], S[1]\}$. Some ideas on how to extend the proof to graphs, i.e., to schemas with a single binary relation name, are explained in Exercise 4.13. This special case we study here builds on the proof of Proposition 32.1, illustrates key elements in the proof of the 0–1 law, and allows us to derive corollaries about cardinality comparisons.

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 32.6. *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 32.6. Condition (2) of the lemma, and the same argument as in the proof of Lemma 32.2, 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 $\{\psi_1,\dots,\psi_m\}$ 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 32.6.

Proof (of Lemma 32.6). We construct a theory T over $\mathbf{S}$, and then show that it satisfies conditions (1) and (2). For each $k \in \mathbb{N}$, the theory T contains

$$\psi_k(RS) \;=\; \exists x_1 \cdots \exists x_k \left(\bigwedge_{i,j \in [k] \,:\, i<j} \neg(x_i = x_j) \wedge \bigwedge_{i \in [k]} (R(x_i) \wedge S(x_i)) \right),$$

which states that, for a database D, the set $R^D \cap S^D$ has at least k elements,

$$\psi_k(R\bar{S}) \;=\; \exists x_1 \cdots \exists x_k \left(\bigwedge_{i,j \in [k] \,:\, i<j} \neg(x_i = x_j) \wedge \bigwedge_{i \in [k]} (R(x_i) \wedge \neg S(x_i)) \right),$$

which states that $R^D - S^D$ has at least k elements, and

$$\psi_k(\bar{R}S) \;=\; \exists x_1 \cdots \exists x_k \left(\bigwedge_{i,j \in [k] \,:\, i<j} \neg(x_i = x_j) \wedge \bigwedge_{i \in [k]} (\neg R(x_i) \wedge S(x_i)) \right),$$

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 $S^D = \{3n, 3n{+}2 \mid n \in \mathbb{N}\}$. It is easy to see 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 32.6.

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 $|R^D \cap S^D| < k$, for a database D. 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 32.6. □

This completes the proof of Theorem 32.5. □

Theorem 32.5 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 32.7

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 4.10).

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 32.5 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 4.12) 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. □

33
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 32.7, 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 34, 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 34 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 33.1: Two-Sorted Database Schema

A *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 33.2: 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 $D(R) = (a_1, \ldots, a_k)$, we have that $a_i \in \mathsf{Const}$ 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 as a 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 33.3: Two-Sorted Queries

Consider a two-sorted database schema $\mathbf{S}$. A *query of type* $(\tau_1, \dots, \tau_k) \in \{\mathsf{o}, \mathsf{n}\}^k$, 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 $q(D) = (a_1, \dots, a_k)$, it is the case that $a_i \in \mathsf{Const}$ if $\tau_i = \mathsf{o}$, and $a_i \in \mathsf{Num}$ if $\tau_i = \mathsf{n}$, for every $i \in [k]$.

Regarding the domain Num, we assume that it comes equipped with:

- *(Numerical) predicates*: a predicate P or arity $k > 0$ over Num is a subset of Num^k. For brevity, we say that $P(a_1, \dots, a_k)$ holds if $(a_1, \dots, a_k) \in P$. 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 33.4: 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 average 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 $\boldsymbol{\tau} = (\tau_1,\dots,\tau_k) \in \{\mathsf{o},\mathsf{n}\}^k$, 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 $t_1,\dots,t_m$ are $(\Omega,\boldsymbol{\tau})$-terms of type n, then $f(t_1,\dots,t_m)$ 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 $i \neq j$, for $i,j \in [k]$, and $P(i_1,\dots,i_m)$, where $i_1,\dots,i_m \in [k]$ and P is an m-ary predicate from Ω.

Definition 33.5: Syntax of RA with Aggregates and Grouping

Consider a set Ω of predicates, functions, and aggregates over Num. We inductively define $\mathit{RA}_{\mathsf{Aggr}}(\Omega)$ *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 $\boldsymbol{\tau} = (\tau_1,\dots,\tau_k)$, for $k \geq 0$, and $\alpha = (t_1,\dots,t_m)$, 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 $(\tau(t_1),\dots,\tau(t_m))$.

Cartesian Product. If e_1, e_2 are $\mathrm{RA}_{\mathsf{Aggr}}(\Omega)$ expressions over $\mathbf{S}$ of type $(\tau_1,\dots,\tau_k)$ and $(\tau'_1,\dots,\tau'_m)$, 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 $(\tau_1,\dots,\tau_k,\tau'_1,\dots,\tau'_m)$.

Aggregation and Grouping. If e is an $\mathrm{RA}_{\mathsf{Aggr}}(\Omega)$ expression over $\mathbf{S}$ of type $\boldsymbol{\tau} = (\tau_1, \dots, \tau_k)$, for $k \geq 0$, $\alpha = (i_1, \dots, i_m)$, for $m \geq 0$, is a list of integers from $[k]$, and $t_1, \dots, t_\ell$, for $\ell \geq 0$, are $(\Omega, \boldsymbol{\tau})$-terms of type n such that $\mathrm{VAR}(t_i) \cap \{i_1, \dots, i_m\} = \emptyset$, for each $i \in [\ell]$, then

$$\mathsf{Aggr}_\alpha[\mathcal{F}_1(t_1), \dots, \mathcal{F}_\ell(t_\ell)](e),$$

where $\mathcal{F}_1, \dots, \mathcal{F}_\ell$ are aggregates from Ω, is an $\mathrm{RA}_{\mathsf{Aggr}}(\Omega)$ expression over $\mathbf{S}$ of type $(\tau_{i_1}, \dots, \tau_{i_m}, \mathsf{n}, \dots, \mathsf{n})$ 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 33.5, 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 33.5 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, can be expressed via expressions of the form $\pi_{(a)}(R)$ of type o (based on the semantics of $\mathrm{RA}_{\mathsf{Aggr}}(\Omega)$ expressions as defined next).

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, $P(i_1, \dots, i_m)$ is true in a tuple $(a_1, \dots, a_k)$ iff $i_1, \dots, i_m$ correspond to columns of type n and $P(a_{i_1}, \dots, a_{i_m})$ 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 33.6: 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 following $\mathrm{RA}_{\mathsf{Aggr}}(\Omega)$ 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, $\mathsf{Aggr}_{(i_1,\dots,i_m)}[\mathcal{F}_1(t_1),\dots,\mathcal{F}_\ell(t_\ell)](R)$ corresponds to

```
SELECT   $i_1,\dots,i_m,\mathcal{F}_1(t_1),\dots,\mathcal{F}_\ell(t_\ell)$
FROM     R
GROUP BY $i_1,\dots,i_m$
```

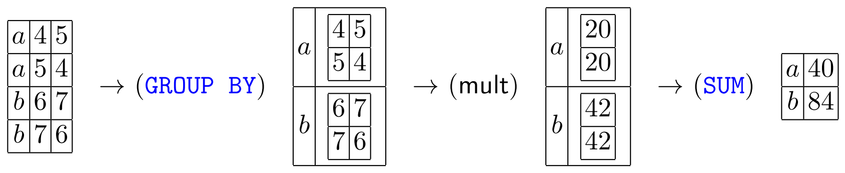
assuming that the attributes of R are $1,\dots,k$.

Example 33.7: Aggregation and Grouping

For a ternary relation R, the evaluation of

$$\mathsf{Aggr}_{(1)}[\mathtt{SUM}(\mathsf{mult}(2,3))](R)$$

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 define the semantics of generalized projection, and aggregation with grouping. We first need some auxiliary notions. Consider an $(\Omega,\boldsymbol{\tau})$-term t, where $\boldsymbol{\tau} = (\tau_1,\dots,\tau_k) \in \{\mathsf{o},\mathsf{n}\}^k$. A tuple $\bar{a} = (a_1,\dots,a_n) \in (\mathsf{Const} \cup \mathsf{Num})^n$, for $n \geq k$, is *compatible* with t if $\tau_i = \mathsf{o}$ implies $a_i \in \mathsf{Const}$, and $\tau_i = \mathsf{n}$ implies $a_i \in \mathsf{Num}$, for every $i \in \mathrm{VAR}(t)$. We define the *evaluation* of t over a tuple $\bar{a} = (a_1,\dots,a_n)$ that is compatible with it, denoted $\mathrm{eval}(t|\bar{a})$, as follows:

- 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 $t = f(t_1,\dots,t_m)$, then $\mathrm{eval}(t|\bar{a}) = f\big(\mathrm{eval}(t_1|\bar{a}),\dots,\mathrm{eval}(t_m|\bar{a})\big)$.

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 $\alpha = (i_1,\dots,i_m)$, for $m \geq 0$, from $[n]$, and a tuple $\bar{a} \in \pi_\alpha(R)$, we define the bag

$$B(\bar{a}, R, t) \;=\; \{\!\mid \mathrm{eval}(t|\bar{c}) \mid \bar{c} \in R \text{ and } \bar{a} = \pi_\alpha(\bar{c}) \mid\!\}\,.$$

For instance, going back to Example 33.7, for $t = \mathsf{mult}(2,3)$,

$$B((a), R, t) \;=\; \{\!\mid \mathrm{eval}(t|(a,4,5)), \mathrm{eval}(t|(a,5,4)) \mid\!\} = \{\!\mid 20, 20 \mid\!\}\,.$$

Definition 33.8: 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 $e = \pi_{(t_1,\dots,t_m)}(e_1)$, where e_1 is an $\mathrm{RA}_{\mathsf{Aggr}}(\Omega)$ expression, then
$$e(D) \;=\; \{(\mathrm{eval}(t_1|\bar{a}), \dots, \mathrm{eval}(t_m|\bar{a})) \;\mid\; \bar{a} \in e_1(D)\}\;.$$
- If $e = \mathsf{Aggr}_{(i_1,\dots,i_m)}[\mathcal{F}_1(t_1), \dots, \mathcal{F}_\ell(t_\ell)](e_1)$, where e_1 is an $\mathrm{RA}_{\mathsf{Aggr}}(\Omega)$ expression, then
$$\Big\{\big(\bar{a}, \mathcal{F}_1\big(B(\bar{a}, R, t_1)\big), \dots, \mathcal{F}_\ell\big(B(\bar{a}, R, t_\ell)\big)\big) \;\Big|\; \bar{a} \in \pi_{(i_1,\dots,i_m)}(e_1(D))\Big\}\;.$$

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 33.9: 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 `SUM` and `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: $\mathsf{RA}_{\mathsf{Aggr}}(\Omega)$-$\mathsf{Evaluation}$

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 $\mathsf{RA}_{\mathsf{Aggr}}(\Omega)$-$\mathsf{Evaluation}$. 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-$\mathsf{Evaluation}$, 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, $\mathsf{RA}_{\mathsf{Aggr}}(\Omega)$-$\mathsf{Evaluation}$ is in $\mathcal{C}$ in data complexity, for some complexity class $\mathcal{C}$, if e-$\mathsf{Evaluation}$ 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 $\mathsf{RA}_{\mathsf{Aggr}}(\Omega)$-$\mathsf{Evaluation}$, 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 33.10

Consider a set Ω of predicates, functions, and aggregates over Num that are computable in polynomial time. Then $\mathsf{RA}_{\mathsf{Aggr}}(\Omega)$-$\mathsf{Evaluation}$ is in ExpTime, and in PTime in data complexity.

Aggregates, Grouping, and Locality

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 the same recursive queries, such as reachability, that were already shown to be inexpressible in FO in Chapter 31. In fact, the tool we shall use to show this is also the same, namely locality. It turns out that adding any amount of aggregate and counting power does not break the locality of FO, i.e., queries can still only see within a fixed distance of their free variables. In particular, in this chapter we are going to show the following inexpressibility result:

Theorem 34.1

Consider a numerical domain Num, and an arbitrary set Ω of predicates, functions, and aggregates over Num. There is no $\mathrm{RA}_{\mathsf{Aggr}}(\Omega)$ query e over the schema $\mathbf{S} = \{E : (\mathsf{o}, \mathsf{o})\}$ that expresses the reachability query q_{reach}.

This will motivate extending query languages with explicit recursive computation, as will be done in Chapters 35 – 38.

Locality of Two-Sorted Queries

We need to lift the main definitions related to locality, seen in Chapter 31, to queries over databases that include values of the numerical type. This is very routine: the definitions of the Gaifman graph, radius-r balls and neighborhoods, and their isomorphism, do not change at all.

Definition 34.2: Locality of Queries

A query q of type $\boldsymbol{\tau} \in \{\mathsf{o}, \mathsf{n}\}^k$ 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 a locality result focussing on schemas that consist of relation names of ordinary type. For brevity, we say that as schema $\mathbf{S}$ is of ordinary type if, for each $R : \boldsymbol{\tau} \in \mathbf{S}$, $\boldsymbol{\tau} \in \{\mathsf{o}\}^k$ for some $k \geq 0$.

Theorem 34.3

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 of ordinary type is local.

What if instead we consider arbitrary two-sorted schemas? In this case there is no known result analogous to Theorem 34.3. In fact, proving an analog of Theorem 34.1 for relations of type (n, n) would resolve deep open problems in complexity theory (see Exercise 4.19 for further explanations).

The rather long proof of Theorem 34.3 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 34.3, 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 the locality of this counting logic. In fact, we shall see that the proof given in Chapter 31 applies with a few minor modifications.

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 4.15.

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 $\mathsf{Var_o}$, which will be ranging over the set of constants Const, and $\imath, \jmath, \ldots$ for variables from $\mathsf{Var_n}$, which will be ranging over the numerical domain Num.

Definition 34.4: 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 $\mathsf{Var_o}$ 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 $\imath \in \mathsf{Var_n}$ 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 $x \in \mathsf{Var_o}$, then $x = a$ is an atomic formula, whose set of variables $\mathrm{FV}(x = a)$ is $\{x\}$.
- If $x, y \in \mathsf{Var_o}$, 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 $\imath \in \mathsf{Var_n}$, 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 $\mathsf{Num} \cup \mathsf{Var_n}$, 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 34.5: 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 $t = f(t_1, \dots, t_m)$, then $t^{D,\eta} = f(t_1^{D,\eta}, \dots, t_m^{D,\eta})$.
- 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 $\mathrm{FV}(\varphi)$ and satisfy 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 (omitting Boolean connectives, which are defined in the standard way):

- 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 $\eta(\imath) = t^{D,\eta}$.
- If φ is $R(u_1, \dots, u_k)$, for R a k-ary relation symbol from $\mathbf{S}$, then $(D,\eta) \models \varphi$ if $R(\eta(u_1), \dots, \eta(u_k)) \in D$.
- If φ is $P(u_1, \dots, u_k)$, for P a k-ary numerical predicate from Ω, then $(D,\eta) \models \varphi$ if $(\eta(u_1), \dots, \eta(u_k)) \in P$.
- 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 $u \in \mathsf{Var}_{\mathsf{o}}$, and $a \in \mathsf{Num}$ if $u \in \mathsf{Var}_{\mathsf{n}}$.
- 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 $u \in \mathsf{Var}_{\mathsf{o}}$, and $a \in \mathsf{Num}$ if $u \in \mathsf{Var}_{\mathsf{n}}$.

We can now observe 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, any of the infinitely many assignments $\eta : \{x, \imath, \jmath\} \rightarrow \{a\} \cup \mathsf{Num}$ for which $\eta(x) = a$ and $\eta(\jmath) = \eta(\imath) + 1$ satisfies $(D, \eta) \models \varphi$. 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. One can still 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 34.6

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 cases, 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

$$\varphi_e = \exists w_{m+1} \cdots \exists w_k \left(\psi_{e'} \wedge \right.$$
$$\left. \bigwedge_{i=1}^{\ell} v_i = \mathsf{Aggr}_{\mathcal{F}_i}(w_{m+1}, \dots, w_k)\,(\psi_{e'}, t_i(u_1, \dots, u_m, w_{m+1}, \dots, w_k)) \right),$$

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 $\bar{u}_e = (u_1, \dots, u_m, v_1, \dots, v_\ell)$ 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 34.1, where the goal is to express an $\mathrm{RA}_{\mathsf{Aggr}}(\Omega)$ query, by exploiting Proposition 34.6, into a convenient infinitary counting logic. We proceed to introduce the counting logic $\mathbf{L_C}$, and its sublogic $\mathbb{L}_\mathbf{C}$, which we prove to be equivalent to $\mathbf{L_C}$. 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 34.7: 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 $x, y \in \mathsf{Var}_\mathsf{o}$, then $x = y$ is an atomic formula with $\mathsf{rank}(x = y) = 0$.
- If $a \in \mathsf{Num}$, and $\imath \in \mathsf{Var}_\mathsf{n}$, then $\imath = a$ is an atomic formula with $\mathsf{rank}(\imath = a) = 0$.
- If $\imath, \jmath \in \mathsf{Var}_\mathsf{n}$, then $\imath = \jmath$ is an atomic formula with $\mathsf{rank}(\imath = \jmath) = 0$.
- If $u_1, \dots, u_k$ are terms (not necessarily distinct), where u_i is of type τ_i for each $i \in [k]$, and $R : (\tau_1, \dots, \tau_k)$ belongs to $\mathbf{S}$, then $R(u_1, \dots, u_k)$ is an atomic formula with $\mathsf{rank}(R(u_1, \dots, u_k)) = 0$.
- If Φ is a (possibly infinite) set of formulae, and $k = \sup_{\varphi \in \Phi} \mathsf{rank}(\varphi)$ is finite, then $\psi = \left(\bigwedge_{\varphi \in \Phi} \varphi\right)$ and $\psi' = \left(\bigvee_{\varphi \in \Phi} \varphi\right)$ are formulae with $\mathsf{rank}(\psi) = \mathsf{rank}(\psi') = k$.
- If φ is a formula, then $(\neg\varphi)$ is a formula with $\mathsf{rank}(\neg\varphi) = \mathsf{rank}(\varphi)$.
- If φ is a formula, $\bar{u} = (u_1, \dots, u_k)$ is a tuple over Var, and $n \in \mathbb{N}$, then $\psi = \left(\exists^{\geq n} \bar{u}\, \varphi\right)$ is a formula with $\mathsf{rank}(\psi) = \mathsf{rank}(\varphi) + k$.

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 $\mathbb{L}_{\mathbf{C}}$, $x \in \mathsf{Var}_{\mathsf{o}}$, and $n \in \mathbb{N}$, then $\psi = \left(\exists^{\geq n} x\, \varphi\right)$ is a formula of $\mathbb{L}_{\mathbf{C}}$ with $\mathsf{rank}(\psi) = \mathsf{rank}(\varphi) + 1$.

Let us stress that $\mathbf{L}_{\mathbf{C}}$, 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 $\mathsf{rank}(\varphi_i) = i$ for each $i > 0$, then $\mathsf{rank}\left(\bigvee_{\varphi \in \Phi} \varphi\right)$ is infinite. The set of free variables of an $\mathbf{L}_{\mathbf{C}}$ formula φ, denoted $\mathrm{FV}(\varphi)$, as well as the semantics of $\mathbf{L}_{\mathbf{C}}$, 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 $\mathrm{FV}(\psi)$ to $\mathrm{Dom}(D) \cup \mathsf{Num}$, 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 $\mathrm{FV}(\psi)$ 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 $\mathbf{L}_{\mathbf{C}}$ 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.

Proposition 34.8

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 34.9. *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 $\mathsf{rank}(t) = 0$.
- If $t = f(t_1, \ldots, t_m)$, then $\mathsf{rank}(t) = \max_{i \in [m]}\{\mathsf{rank}(t_i)\}$.
- If $t = \mathsf{Aggr}_{\mathcal{F}}(\bar{v})\, (\varphi', t')$, then $\mathsf{rank}(t) = \max\{\mathsf{rank}(\varphi'), \mathsf{rank}(t')\} + k$, 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 $\mathsf{rank}(\varphi') = 0$.
- If φ' is $\imath = t$, then $\mathsf{rank}(\varphi') = \mathsf{rank}(t)$.
- If φ' is $R(v_1, \ldots, v_k)$, then $\mathsf{rank}(\varphi') = \max_{i\in[k]}\{\mathsf{rank}(v_i)\}$.
- If φ' is $\varphi_1 \wedge \varphi_2$ or $\varphi_1 \vee \varphi_2$, then $\mathsf{rank}(\varphi') = \max\{\mathsf{rank}(\varphi_1), \mathsf{rank}(\varphi_2)\}$.
- If φ' is $\neg\varphi_1$, then $\mathsf{rank}(\varphi') = \mathsf{rank}(\varphi_1)$.
- If φ' is $\exists v\, \varphi_1$ or $\forall v\, \varphi_1$, then $\mathsf{rank}(\varphi') = \mathsf{rank}(\varphi_1) + 1$.

We are now ready to provide the translation of an Ω-term t of type n occurring in φ, and of φ itself, into $\mathbf{L_C}$ by mutual induction. In what follows, given an $\mathbf{L_C}$ 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 $\mathbf{L_C}$ 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

$$\alpha_t^\imath \;=\; \bigvee_{\substack{(a_1,\ldots,a_m,a_{m+1})\in\mathsf{Num}^{m+1}:\\ f(a_1,\ldots,a_m)=a_{m+1}}} \left(\bigwedge_{i\in[m]} \alpha_{t_i}^{\jmath_i}[\jmath_i/a_i] \;\to\; \imath = a_{m+1} \right)$$

  of rank $\max_{i\in[m]}\{\mathsf{rank}(\alpha_{t_i}^{\jmath_i})\} \leq \mathsf{rank}(t)$, 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 $\max\{\mathsf{rank}((\varphi')^{\diamond}), \mathsf{rank}(\alpha_{t'}^{\jmath})\} + k \leq \max\{\mathsf{rank}(\varphi'), \mathsf{rank}(t')\} + k = \mathsf{rank}(t)$, 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, $\mathsf{rank}(\varphi^{\diamond}) = \mathsf{rank}(\varphi)$.
- If φ is the atomic formula $\imath = t$, then $\varphi^{\diamond}$ is $\alpha_t^{\imath}$ with $\mathsf{rank}(\varphi^{\diamond}) \leq \mathsf{rank}(t)$.
- 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 $\mathsf{rank}(\varphi^{\diamond}) \leq \max_{i \in [m]}\{\mathsf{rank}(\alpha_{v_i}^{\jmath_i})\} \leq \mathsf{rank}(\varphi)$.
- 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 $\mathsf{rank}(\varphi^{\diamond}) = \mathsf{rank}(\varphi)$.
- 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 $\mathsf{rank}(\varphi^{\diamond}) = \mathsf{rank}(\varphi)$.

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 34.9 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 34.8.

Lemma 34.10. *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 $\mathsf{rank}(\exists^{\geq n}(v, w)\, \psi') = \mathsf{rank}(\psi') + 2$.

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 $\mathsf{rank}(\exists^{\geq n}\imath\, \psi'') - 1$. This completes the proof of Lemma 34.10. □

By Lemma 34.9 and 34.10, 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

The last step of the proof of Theorem 34.3 is proving the locality of $\mathbb{L}_{\mathbf{C}}$.

Proposition 34.11

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 $\mathrm{FV}(\varphi)$. Then $\varphi(\bar{x})$ is $(3^k - 1)/2$-local.

Proof. We proceed exactly as in the proof of Theorem 31.3, using rank in place of quantifier rank. The cases of atomic formulae are identical. For infinitary connectives $\bigvee_{\varphi \in \Phi} \varphi$ and $\bigwedge_{\varphi \in \Phi} \varphi$, we note that k is the bound on the rank of all formulae $\varphi \in \Phi$, and thus, the same proof as in Theorem 31.3 applies. The case of negation does not change either.

The last step is to consider formulae of the form $\varphi(\bar{x}) = \exists^{\geq n} y\, \psi$, with $\mathsf{rank}(\varphi) = k$. Thus, $\mathsf{rank}(\psi) = k - 1$, and by the induction hypothesis, $\psi(\bar{x}, y)$ is r-local, where $r = (3^{k-1} - 1)/2$. As in Theorem 31.3, we must show that

φ is $(3r + 1)$-local. The proof is the same as the one given before. Indeed, recall that we established that, given a database D of $\mathbf{S}$, with tuples $\bar{a}, \bar{b}$ over Const such that $N^D_{3r+1}(\bar{a})$ and $N^D_{3r+1}(\bar{b})$ are isomorphic, we have a bijection $f : \mathrm{Dom}(D) \to \mathrm{Dom}(D)$ such that $N^D_r(\bar{a}, c)$ and $N^D_r(\bar{b}, f(c))$ are isomorphic, for each $c \in \mathrm{Dom}(D)$. By r-locality of ψ, we get that the number of witnesses to $\psi(\bar{a}, y)$ and $\psi(\bar{b}, y)$ are the same, and thus $\varphi(\bar{a})$ and $\varphi(\bar{b})$ are either simultaneously true, or simultaneously false. $\square$

We can now finalize the proof of Theorem 34.3. Given an $\mathrm{RA}_{\mathsf{Aggr}}(\Omega)$ query e of type $\{\mathsf{o}\}^k$, for $k \geq 0$, over a schema $\mathbf{S}$ without numerical types, by Propositions 34.6 and 34.8, 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 34.11, e is $(3^k - 1)/2$-local, where k is the rank of the formula φ_e.

35

Adding Recursion: Datalog

As discussed in Chapter 34, 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 35.1: Syntax of Datalog

A *Datalog rule* over a schema $\mathbf{S}$ is an expression of the form

$$R_0(\bar{x}) \coloneq R_1(\bar{u}_1), \ldots, R_n(\bar{u}_n)$$

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 35.2: 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:

$$\begin{aligned} \text{Reachable}(x, y) \ &:\!-\ \text{Edge}(x, y) \\ \text{Reachable}(x, y) \ &:\!-\ \text{Reachable}(x, z), \text{Edge}(z, y)\,. \end{aligned}$$

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 $D \models \varphi$, for every sentence $\varphi \in \Phi$.[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 35.3: From a Program to a Theory

Given a Datalog rule ρ of the form $R_0(\bar{x}) \coloneq R_1(\bar{u}_1), \ldots, R_n(\bar{u}_n)$, 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 32.

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

$$R_0(\bar{x}) \coloneq R_1(\bar{u}_1), \ldots, R_n(\bar{u}_n)$$

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 35.4

Consider a Datalog rule ρ of the form $R_0(\bar{x}) \coloneq R_1(\bar{u}_1), \ldots, R_n(\bar{u}_n)$, 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 35.5. *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 $R_0(\bar{x}) \coloneq R_1(\bar{u}_1), \ldots, R_n(\bar{u}_n)$, 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 35.6

Consider a Datalog program Π, and a database D of $\mathsf{edb}(\Pi)$. Then

$$|\mathsf{MM}(\Pi, D)| \;=\; 1.$$

Proof. By Lemma 35.5, $\mathsf{B}(\Pi, D)$ 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 $R_0(\bar{x}) \coloneq R_1(\bar{u}_1), \ldots, R_n(\bar{u}_n)$, 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 35.4, 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 35.7: 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 35.6, we get that $\Pi(D)$ is uniquely determined by the program and the database, and thus, Definition 35.7 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 35.6, 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 35.8

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 35.8 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 35.9: 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 $R(\bar{x}) \coloneq R_1(\bar{u}_1), \ldots, R_n(\bar{u}_n)$ such that $(A_\rho, \bar{x}) \rightarrow (D, \bar{a})$.

The *immediate consequence operator* of Π is defined as the function

$$T_\Pi \;:\; \mathrm{Inst}(\mathsf{sch}(\Pi)) \;\rightarrow\; \mathrm{Inst}(\mathsf{sch}(\Pi))$$

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 35.10. *Consider a Datalog program Π. The following hold:*

1. *T_Π is monotone, i.e., for every two databases D and D' of $\mathsf{sch}(\Pi)$, if $D \subseteq D'$ then $T_\Pi(D) \subseteq T_\Pi(D')$.*
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 35.11

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 35.10 implies that $T_\Pi(\Pi(D)) \subseteq \Pi(D)$. By Lemma 35.10, T_Π is monotone, and thus, $T_\Pi(T_\Pi(\Pi(D))) \subseteq T_\Pi(\Pi(D))$. Therefore, by Lemma 35.10, $T_\Pi(\Pi(D))$ 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 35.10, 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 35.12: 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 35.13

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, $T_\Pi^\infty(D) \subseteq D'$, for every fixpoint D' of T_Π that contains 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 35.11 and 35.13:

Corollary 35.14

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 35.2. The formal definition of Datalog queries follows.

Definition 35.15: 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 35.7), 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 35.16: 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 35.8, and the fact that $\mathrm{Dom}(D') = \mathrm{Dom}(D)$, for every database $D' \in \mathsf{M}(\Pi, D)$.

Proposition 35.17

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 35.17 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 UCQ s 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 35.18

The language of Datalog queries is strictly more expressive than the language of UCQs.

Proof. From Theorem 34.1, 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 (Π_q, Answer), where Π_q consists of the CQs $q_1, \ldots, q_n$ seen as rules of the form $\mathrm{Answer}(\bar{x})$:– body. □

Recall from Chapter 28 that UCQ s with variable-constant equality form a strictly more expressive language than UCQ s. 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 UCQ s with variable-constant equality form incomparable languages in terms of expressive power. This is because UCQ s 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 35.17). 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 $q(D) = \{(a)\}$.

36

Expressiveness of Datalog Queries

We have already seen in the previous chapter that Datalog queries are strictly more expressive than UCQ s. We have also seen an easy inexpressibility result, i.e., there are UCQ s 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 UCQ s 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 35.17, 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 36.1: 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

$$\text{if } (D,\bar{a}) \rightarrow_{\mathrm{Dom}(\varPi)} (D',\bar{b}) \text{ and } \bar{a} \in q(D) \;\; \text{then} \;\; \bar{b} \in q(D').$$

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; such UCQ s are called *infinitary*. The evaluation of infinitary UCQ s is defined in the same way as for UCQ s, that is, given a database D of a schema $\mathbf{S}$, and an infinitary UCQ $q = q_1 \cup q_2 \cup \cdots$ over $\mathbf{S}$, where each q_i for $i \geq 1$ is a CQ over $\mathbf{S}$, $q(D) = \bigcup_{i=1}^{\infty} q_i(D)$. It is easy to show that every infinitary UCQ is preserved under homomorphisms; the proof is essentially the same as the one of Proposition 28.10, which establishes that UCQ s 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 36.2

Consider a Datalog query $q = (\varPi, R)$ over a schema $\mathbf{S}$. There exists an infinitary UCQ $q' = \varphi(\bar{x})$ over $\mathbf{S}$ with $\mathrm{Dom}(\varPi) = \mathrm{Dom}(\varphi)$ 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 $\gamma : \mathsf{Const} \cup \mathsf{Var} \rightarrow \mathsf{Const} \cup \mathsf{Var}$, which is the identity on Const and the set of variables not mentioned in $\bar{u}$ and $\bar{v}$, such that $R(\gamma(\bar{u})) = P(\gamma(\bar{v}))$; such a function 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 $\theta : \mathsf{Const} \cup \mathsf{Var} \rightarrow \mathsf{Const} \cup \mathsf{Var}$ such that $\gamma' = \theta \circ \gamma$. For example, the atoms $R(x, y)$ and $R(z, a)$ unify due to the function $\gamma : \mathsf{Const} \cup \mathsf{Var} \rightarrow \mathsf{Const} \cup \mathsf{Var}$ that is the identity on $\mathsf{Const} \cup (\mathsf{Var} - \{x, y, z\})$, and assigns the constant a to the variables x, y, z, i.e., $\gamma(x) = \gamma(y) = \gamma(z) = a$. However, it is easy to verify that γ is not a most general unifier for $R(x, y)$ and $R(z, a)$. In particular, for the unifier γ' for $R(x, y)$ and $R(z, a)$ with $\gamma'(x) = \gamma'(z) = z$ and $\gamma'(y) = a$, we can show that there is no $\theta : \mathsf{Const} \cup \mathsf{Var} \rightarrow \mathsf{Const} \cup \mathsf{Var}$ such that $\gamma' = \theta \circ \gamma$. On the other hand, we can show that γ' is a most general unifier for $R(x, y)$ and $R(z, a)$. For instance, $\gamma = \theta \circ \gamma'$ with $\theta : \mathsf{Const} \cup \mathsf{Var} \rightarrow \mathsf{Const} \cup \mathsf{Var}$ being such that $\theta(u) = a$.

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 to introduce the notion of unfolding of a Boolean CQ with a Datalog program, which relies on unification. Let q be the Boolean CQ

$$\text{Answer} \coloneq R_1(\bar{u}_1), \dots, R_n(\bar{u}_n),$$

and ρ be the Datalog rule

$$P_0(\bar{y}) \coloneq P_1(\bar{v}_1), \dots, P_m(\bar{v}_m).$$

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. We denote by $q_{\rho,i,\gamma}$ the CQ

$$\begin{aligned}\text{Answer} \coloneq{} & R_1(\gamma(\bar{u}_1)), \dots, R_{i-1}(\gamma(\bar{u}_{i-1})),\\ & R_{i+1}(\gamma(\bar{u}_{i+1})), \dots, R_n(\gamma(\bar{u}_n)), P_1(\gamma(\bar{v}_1)), \dots, P_m(\gamma(\bar{v}_m)).\end{aligned}$$

Let $\{R_{i_1}(\bar{u}_{i_1}), \dots, R_{i_\ell}(\bar{u}_{i_\ell})\}$ 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 as the set of CQs $\{q_{\rho,i_1,\gamma_{i_1}}, \dots, q_{\rho,i_\ell,\gamma_{i_\ell}}\}$. 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 a Datalog rule from 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 36.3: 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 36.2.

Proof (of Proposition 36.2). We prove this for Boolean queries, i.e., we assume that the Datalog query $q = (\Pi, R)$ is Boolean, that is, $\mathrm{ar}(R) = 0$. We further assume, without affecting the generality of the proof, that there is exactly one rule $\rho_R \in \Pi$ of the form $R() \text{ :- body}$, 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

$$q^* \ = \ (\Pi^* \cup \{R() \text{ :- } R^*()\}, R).$$

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 $\mathrm{Unfold}_\Pi^\infty(q_R)_{|\mathbf{S}}$ for the subset of $\mathrm{Unfold}_\Pi^\infty(q_R)$ 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

$$q_R^\Pi \ = \ \bigcup_{q' \in \mathrm{Unfold}_\Pi^\infty(q_R)_{|\mathbf{S}}} q'.$$

By construction, q_R^Π is an infinitary UCQ over $\mathbf{S}$, and Π and q_R^Π mention exactly the same constants. It remains to show that q and q_R^Π are equivalent, i.e, $q(D) = q_R^\Pi(D)$, for every database D of $\mathbf{S}$. 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) ⇒ (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 $P(\bar{x}) \text{ :- body}$ 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 36.4. *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, $A_{q_R} \to D_n$ via a homomorphism μ. We consider two cases:

- Assume first that $P(h_{n-1}(\bar{x})) \not\in \mu(A_{q_R})$, or $P(h_{n-1}(\bar{x})) \in \mu(A_{q_R})$ and $P(h_{n-1}(\bar{x})) \in D_{n-1}$. 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 $q'_i \in \mathrm{Unfold}_{\rho_{n-i}}(q'_{i-1})$, and $A_{q_i} \to D_{n-i}$. Therefore, the claim follows due to the sequence of CQs $q'_0, q'_0, q'_1, \dots, q'_{n-1}$.
- The interesting case is when $P(h_{n-1}(\bar{x})) \not\in \mu(A_{q_R})$ and $P(h_{n-1}(\bar{x})) \in D_n - D_{n-1}$. 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 $\hat{\gamma}$. In fact, 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 $q'_i \in \mathrm{Unfold}_{\rho_{n-1}}(q'_{i-1})$, and $A_{q_i} \to D_{n-1}$. The claim follows due to the sequence of CQs $q_R, q'_0, \dots, q'_{n-1}$.

This completes the proof of Lemma 36.4. □

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 36.4. Clearly, $q_0 = q_R$ and $A_{q_n} \to D_n$. However, it is not the case that $q_i \in \mathrm{Unfold}_{\Pi}(q_{i-1})$, 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

$$\left\{ i_j \mid j \in [n] \text{ and } q_{i_j} \not\in \mathrm{Unfold}_{\Pi}(q_{i_j - 1}) \right\}.$$

Observe that $0 \not\in Ind$, 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 $q'_0 = q_R$, $q'_i \in \mathrm{Unfold}_{\Pi}(q'_{i-1})$, for each $i \in [m]$, and $A_{q'_m} \to D$. This implies that (2) holds, as needed.

The Direction (2) $\Rightarrow$ (1)

We first establish the following auxiliary lemma:

Lemma 36.5. *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. $\square$

We can now complete the proof of the direction (2) $\Rightarrow$ (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 13.7). By repeatedly applying Lemma 36.5, we get that $A_{q_0} \to \Pi(D)$. Since $q_0 = q_R$ and $A_{q_R} = A_{\rho_R}$, we conclude that $A_{\rho_R} \to \Pi(D)$. $\square$

By Proposition 36.2, and the fact that infinitary UCQ s are preserved under homomorphisms, we immediately get the following result:

Corollary 36.6

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 13.7, which establishes that every CQ is monotone.

Corollary 36.7

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 $D \to_{\{a\}} D'$. 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 $D' \not\models q$.

37

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 35.14, $\Pi(D) = T_{\Pi}^{\infty}(D)$, which in turn implies that

$$\bar{a} \in q(D) \quad \text{if and only if} \quad R(\bar{a}) \in T_{\Pi}^{\infty}(D).$$

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. We first 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}, for $i \in \{1, \ldots, |\mathsf{B}(\Pi, D)|\}$, takes time

$$O(|\Pi| \cdot |\mathrm{Dom}(D)|^{\mathit{maxvar}} \cdot \mathit{maxbody} \cdot |T_{\Pi}^{i-1}(D)|)$$

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, for each $i \in \{1, \ldots, |\mathsf{B}(\Pi, D)|\}$, $|T_{\Pi}^{i-1}(D)| \leq |\mathsf{B}(\Pi, D)|$. Hence, each application of the T_{Π} operator takes time

$$O(|\Pi| \cdot |\mathrm{Dom}(D)|^{\mathit{maxvar}} \cdot \mathit{maxbody} \cdot |\mathsf{B}(\Pi, D)|).$$

As said above, for computing $T_{\Pi}^{\infty}(D)$ we need to apply the T_{Π} operator at most $|\mathsf{B}(\Pi, D)|$ times, which implies that $T_{\Pi}^{\infty}(D)$ can be computed in time

$$O(|\Pi| \cdot |\mathrm{Dom}(D)|^{maxvar} \cdot maxbody \cdot |\mathsf{B}(\Pi, D)|^2).$$

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 $\mathsf{sch}(\Pi)$. Consequently, $T_{\Pi}^{\infty}(D)$ can be computed in exponential time, which implies that checking whether $R(\bar{a}) \in T_{\Pi}^{\infty}(D)$ 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 37.1

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 $q = (\Pi, \mathrm{Yes})$, 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 $\delta(p, a) = (p', b, \mathrm{dir})$ 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 $\mathrm{dir} = \rightarrow$ (respectively, $\mathrm{dir} = \leftarrow$, $\mathrm{dir} = -$).

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

$$\{\mathrm{Succ}^1, \mathrm{First}^1, \mathrm{Last}^1\},$$

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

$$\{\mathrm{Symbol}_a \mid a \in \Sigma\} \ \cup \ \{\mathrm{Head}\} \ \cup \ \{\mathrm{State}_p \mid p \in Q\} \ \cup \\ \{\mathrm{Yes}\} \ \cup \ \bigcup_{i \in [2,m]} \{\mathrm{Succ}^i, \mathrm{First}^i, \mathrm{Last}^i\} \ \cup \ \{\preceq^m\},$$

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:

- $\mathrm{Symbol}_a(t, c)$: at time instant t, the tape cell c contains the symbol a.
- $\mathrm{Head}(t, c)$: at time instant t, the head points at cell c.
- $\mathrm{State}_p(t)$: at time instant t, M is in state p.
- $\mathrm{Yes}()$: 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,

$$D \ = \ \{\mathrm{Succ}^1(0, 1), \mathrm{First}^1(0), \mathrm{Last}^1(1)\}.$$

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 $x_1, \ldots, x_k$, where $k \geq 0$ will be clear from the context, 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:

$$\begin{array}{rcl}
\mathrm{Succ}^{i+1}(z, \bar{x}, z, \bar{y}) & \text{:-} & \mathrm{Succ}^i(\bar{x}, \bar{y}), \mathrm{First}^1(z) \\
\mathrm{Succ}^{i+1}(z, \bar{x}, z, \bar{y}) & \text{:-} & \mathrm{Succ}^i(\bar{x}, \bar{y}), \mathrm{Last}^1(z) \\
\mathrm{Succ}^{i+1}(z, \bar{x}, v, \bar{y}) & \text{:-} & \mathrm{Succ}^1(z, v), \mathrm{Last}^i(\bar{x}), \mathrm{First}^i(\bar{y}) \\
\mathrm{First}^{i+1}(x, \bar{y}) & \text{:-} & \mathrm{First}^1(x), \mathrm{First}^i(\bar{y}) \\
\mathrm{Last}^{i+1}(x, \bar{y}) & \text{:-} & \mathrm{Last}^1(x), \mathrm{Last}^i(\bar{y}) \\
\preceq^m (\bar{x}, \bar{y}) & \text{:-} & \mathrm{Succ}^m(\bar{x}, \bar{y}) \\
\preceq^m (\bar{x}, \bar{z}) & \text{:-} & \preceq^m (\bar{x}, \bar{y}), \mathrm{Succ}^m(\bar{y}, \bar{z}).
\end{array}$$

The Program $\Pi_{\mathbf{start}}$. Assuming that $w = a_0, \ldots, a_{|w|-1}$, we add the rules:

$$\begin{array}{rcl}
\mathrm{State}_s(\bar{x}) & \text{:-} & \mathrm{First}^m(\bar{x}) \\
\mathrm{Symbol}_{a_0}(\bar{x}, \bar{x}) & \text{:-} & \mathrm{First}^m(\bar{x}) \\
\mathrm{Symbol}_{a_1}(\bar{x}_0, \bar{x}_1) & \text{:-} & \mathrm{First}^m(\bar{x}_0), \mathrm{Succ}^m(\bar{x}_0, \bar{x}_1) \\
 & \vdots & \\
\mathrm{Symbol}_{a_{|w|-1}}(\bar{x}_0, \bar{x}_i) & \text{:-} & \mathrm{First}^m(\bar{x}_0), \mathrm{Succ}^m(\bar{x}_0, \bar{x}_1), \ldots, \mathrm{Succ}^m(\bar{x}_{|w|-2}, \bar{x}_{|w|-1}) \\
\mathrm{Symbol}_{\sqcup}(\bar{x}_0, \bar{y}) & \text{:-} & \mathrm{First}^m(\bar{x}_0), \mathrm{Succ}^m(\bar{x}_0, \bar{x}_1), \ldots, \\
 & & \qquad \mathrm{Succ}^m(\bar{x}_{|w|-2}, \bar{x}_{|w|-1}), \preceq^m (\bar{x}_{|w|-1}, \bar{y}) \\
\mathrm{Head}(\bar{x}, \bar{x}) & \text{:-} & \mathrm{First}^m(\bar{x})
\end{array}$$

The Program Π_δ. For each pair $(p, a) \in (Q - \{\text{“yes”}, \text{“no”}\}) \times \Sigma$, with $\delta(p, a) = (p', b, \mathrm{dir})$, we add the following Datalog rules. For brevity, let

$$\Phi_{(p,a)}(\bar{x}, \bar{y}, \bar{z}) \;=\; \mathrm{State}_p(\bar{x}), \mathrm{Head}(\bar{x}, \bar{y}), \mathrm{Symbol}_a(\bar{x}, \bar{y}), \mathrm{Succ}^m(\bar{x}, \bar{z}).$$

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:

$$\begin{aligned}\mathrm{State}_{p'}(\bar{z}) &\;:\!-\; \Phi_{(p,a)}(\bar{x},\bar{y},\bar{z})\\ \mathrm{Symbol}_b(\bar{z},\bar{y}) &\;:\!-\; \Phi_{(p,a)}(\bar{x},\bar{y},\bar{z}).\end{aligned}$$

The next rule, which is responsible for moving the head, depends on the direction dir $\in \{\rightarrow,\leftarrow,-\}$. In particular, if dir $=\rightarrow$, then we add the rule

$$\mathrm{Head}(\bar{z},\bar{v}) \;:\!-\; \Phi_{(p,a)}(\bar{x},\bar{y},\bar{z}), \mathrm{Succ}^m(\bar{y},\bar{v}).$$

If dir $=\leftarrow$, then we add the rule

$$\mathrm{Head}(\bar{z},\bar{v}) \;:\!-\; \Phi_{(p,a)}(\bar{x},\bar{y},\bar{z}), \mathrm{Succ}^m(\bar{v},\bar{y}).$$

Finally, if dir $= -$, then we add the rule

$$\mathrm{Head}(\bar{z},\bar{y}) \;:\!-\; \Phi_{(p,a)}(\bar{x},\bar{y},\bar{z}).$$

The Program $\Pi_{\mathbf{inertia}}$. 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

$$\mathrm{Symbol}_a(\bar{v},\bar{y}) \;:\!-\; \mathrm{Symbol}_a(\bar{x},\bar{y}), \mathrm{Head}(\bar{x},\bar{z}), \preceq^m(\bar{y},\bar{z}), \mathrm{Succ}^m(\bar{x},\bar{v}).$$

The next rule does the same for the tape cells coming after the current cell

$$\mathrm{Symbol}_a(\bar{x},\bar{y}) \;:\!-\; \mathrm{Symbol}_a(\bar{x},\bar{y}), \mathrm{Head}(\bar{x},\bar{z}), \preceq^m(\bar{z},\bar{y}), \mathrm{Succ}^m(\bar{x},\bar{v}).$$

The Program $\Pi_{\mathbf{accept}}$. Finally, we check whether M has reached an accepting configuration via the Datalog rule

$$\mathrm{Yes} \;:\!-\; \mathrm{State}_{\text{“yes”}}(\bar{x}).$$

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 $q = (\Pi, \mathrm{Yes})$, 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 $\mathrm{Dom}(D)$, 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 37.2

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, called the *sources*, and exactly one node without outgoing edges, called 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 by recursively assigning 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 to 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 to 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

$$C(\bar{w}) = 1 \quad \text{if and only if} \quad q(D_{C,\bar{w}}) = \texttt{true}.$$

Therefore, q-Evaluation is PTIME-hard, and the claim follows. □

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. We call a Datalog query (Π, R) *constant-free* if the rules of the Datalog program Π do not mention constants.

Theorem 38.1

Datalog-Satisfiability is in ExpTime, and in PTime for constant-free Datalog queries.

Proof. Consider a Datalog query $q = (\Pi, R)$, and assume that $a_1, \dots, a_n \in \mathsf{Const}$ are the constants mentioned by Π. We first characterize the satisfiability of q via a very simple database. Let

$$D_\Pi \;= \left\{ P(\bar{c}) \mid P \in \mathsf{edb}(\Pi) \text{ and } \bar{c} \in \{\star, a_1, \dots, a_n\}^{\mathrm{ar}(P)} \right\},$$

where $\star$ is a value from $\mathsf{Const} - \{a_1, \dots, a_n\}$. We can show the following lemma:

Lemma 38.2. *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. Assume now that q is satisfiable. This implies that there exists a database D of $\mathsf{edb}(\Pi)$ such that $q(D) \neq \emptyset$, i.e., there exists a tuple $\bar{c}$ over $\mathrm{Dom}(D)$ with $R(\bar{c}) \in \Pi(D)$. Let $h : \mathrm{Dom}(D) \to \{\star, a_1, \ldots, a_n\}$ be the function that maps each element in $\mathrm{Dom}(D) - \{a_1, \ldots, a_n\}$ to $\star$ and each element in $\mathrm{Dom}(D) \cap \{a_1, \ldots, a_n\}$ to itself. 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, $R(h(\bar{c})) \in \Pi(D_{\Pi})$, which implies that $h(\bar{c}) \in q(D_{\Pi})$. Thus, $q(D_{\Pi}) \neq \emptyset$. □

Lemma 38.2 leads to the following simple procedure for checking whether the Datalog query $q = (\Pi, R)$ is satisfiable:

$$\text{if } q(D_{\Pi}) \neq \emptyset, \text{then yes; otherwise, no.}$$

From the analysis performed in Chapter 37 (see the discussion before Theorem 37.1), we conclude that checking whether $q(D_{\Pi}) \neq \emptyset$ is feasible in time

$$O\left(|\Pi| \cdot |\mathrm{Dom}(D_{\Pi})|^{maxvar} \cdot maxbody \cdot |\mathsf{sch}(\Pi)|^2 \cdot |\mathrm{Dom}(D_{\Pi})|^{2 \cdot \mathrm{ar}(\Pi)}\right),$$

where *maxvar* and *maxbody* are the maximum number of variables and body atoms, respectively, in a rule of Π. This implies that Datalog-Satisfiability is in ExpTime, as needed.

Now, observe that in the case where Π is constant-free, we have that

$$D_{\Pi} = \{P(\star, \ldots, \star) \mid P \in \mathsf{edb}(\Pi)\}.$$

Therefore, checking whether $q(D_{\Pi}) \neq \emptyset$ is feasible in time

$$O\left(|\Pi| \cdot maxbody \cdot |\mathsf{sch}(\Pi)|^2\right)$$

since $|\mathrm{Dom}(D_{\Pi})| = 1$. This implies that, for constant-free Datalog queries, Datalog-Satisfiability is in PTime, 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 38.3

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

$$S \rightarrow AA \qquad A \rightarrow a \qquad A \rightarrow b.$$

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 the symbol 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 38.3). 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

$$\{\mathrm{Symbol}_A \mid A \in T\} \qquad \text{and} \qquad \{\mathrm{Symbol}_A \mid A \in N\},$$

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

$$\mathrm{Symbol}_A(x_1, x_{n+1}) \ \text{:-}\ \mathrm{Symbol}_{A_1}(x_1, x_2), \mathrm{Symbol}_{A_2}(x_2, x_3), \ldots, \mathrm{Symbol}_{A_n}(x_n, x_{n+1}).$$

We finally define the Datalog query $q_G = (\Pi_G, \mathrm{Symbol}_S)$.

Example 38.4: 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

$$\begin{aligned}
\mathrm{Symbol}_S(x_1, x_3) &\ \text{:-}\ \mathrm{Symbol}_A(x_1, x_2), \mathrm{Symbol}_a(x_2, x_3)\\
\mathrm{Symbol}_A(x_1, x_4) &\ \text{:-}\ \mathrm{Symbol}_a(x_1, x_2), \mathrm{Symbol}_b(x_2, x_3), \mathrm{Symbol}_A(x_3, x_4)\\
\mathrm{Symbol}_A(x_1, x_3) &\ \text{:-}\ \mathrm{Symbol}_a(x_1, x_2), \mathrm{Symbol}_a(x_2, x_3),
\end{aligned}$$

while the query $q_G = (\Pi_G, \mathrm{Symbol}_S)$.

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 $\bar{a} \in \mathsf{Const}^{\mathrm{ar}(R)}$, 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 $R_0(\bar{x}_0) \text{ :- } R_1(\bar{x}_1), \ldots, R_n(\bar{x}_n)$, 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 $i \in [n]$.

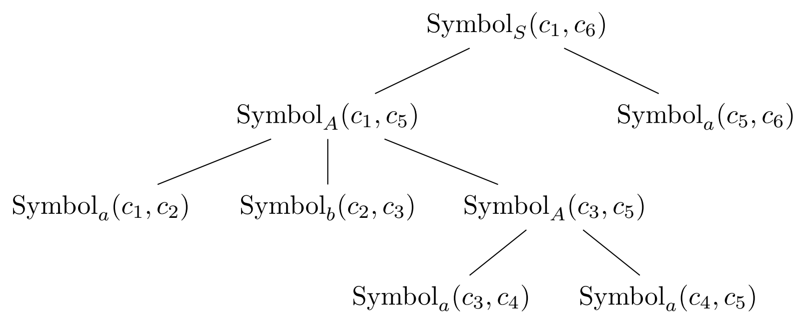
We say that the sequence of atoms $R_1(\bar{a}_1), \ldots, R_n(\bar{a}_n)$ is *induced* by the proof tree T if, assuming that the leaf nodes of T are $v_1, \ldots, v_n$ (in this order), then $\lambda(v_i) = R_i(\bar{a}_i)$, for each $i \in [n]$. Given a database D of $\mathsf{edb}(\Pi)$, *a proof tree 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 $\mathrm{Dom}(D)$, and for each leaf node v of T, $\lambda(v) \in D$.

Example 38.5: Proof Tree

Consider the Datalog program Π_G obtained from the CFG G as in Example 38.4. A proof tree of the atom $\mathrm{Symbol}_S(c_1, c_6)$ from Π_G and

$$D \supseteq \{\mathrm{Symbol}_a(c_1, c_2), \mathrm{Symbol}_b(c_2, c_3), \mathrm{Symbol}_a(c_3, c_4), \mathrm{Symbol}_a(c_4, c_5), \mathrm{Symbol}_a(c_5, c_6)\}$$

is the following one


$\mathrm{Symbol}_S(c_1, c_6)$
$\mathrm{Symbol}_A(c_1, c_5)$
$\mathrm{Symbol}_a(c_5, c_6)$
$\mathrm{Symbol}_a(c_1, c_2)$
$\mathrm{Symbol}_b(c_2, c_3)$
$\mathrm{Symbol}_A(c_3, c_5)$
$\mathrm{Symbol}_a(c_3, c_4)$
$\mathrm{Symbol}_a(c_4, c_5)$

Clearly, the above proof tree induces the sequence of atoms

$$\mathrm{Symbol}_a(c_1, c_2), \mathrm{Symbol}_b(c_2, c_3), \mathrm{Symbol}_a(c_3, c_4), \mathrm{Symbol}_a(c_4, c_5), \mathrm{Symbol}_a(c_5, c_6).$$

It is an easy exercise to show the following lemma:

Lemma 38.6. *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:*

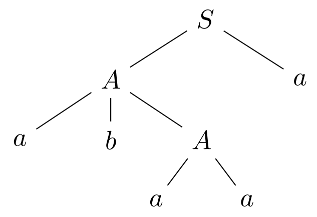
1. $\bar{a} \in q(D)$.
2. *There exists a proof tree of $R(\bar{a})$ from Π and D.*

The next technical lemma makes apparent the intention underlying the transformation of a CFG G to a Datalog program Π_G.

Lemma 38.7. *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 38.4.



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 38.8

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 38.6, 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 38.7, 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 38.7, 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 38.6 implies that $(c, d) \in q_{G_2}(D)$, and the claim follows. □

Since $\mathsf{CFG\text{-}Containment}$ is undecidable, Proposition 38.8 implies the same for $\mathsf{Datalog\text{-}Containment}$. This completes the proof of Theorem 38.3. □

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 38.9

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 name 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 35, 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 35.14, $\Pi(D) = T_\Pi^\infty(D)$. 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 of $\mathsf{edb}(\Pi)$, 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 it 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 38.10: 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 38.11: Program Boundedness

A Datalog program Π is *bounded* if there exists $k \in \mathbb{N}$ such that $\mathsf{stage}(\Pi, D) \leq k$, for every database D of $\mathsf{edb}(\Pi)$.

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 38.12: Program Boundedness and UCQ s

Consider the Datalog query $q = (\Pi, R)$, where Π consists of the rules

$$\begin{aligned} P(x,y) &\coloneq S(x,y) \\ P(x,y) &\coloneq P(x,z), S(z,y) \\ R(x,y) &\coloneq S(x,y) \\ R(x,y) &\coloneq T(x,y). \end{aligned}$$

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 38.12 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 $R^{T_\Pi^\infty(D)} =$

$R^{T^k_\Pi(D)}$, denoted $\mathsf{stage}_{\Pi,D}(R)$. Considering again the Example 38.12, 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 38.13: Query Boundedness

A Datalog query $q = (\Pi, R)$ is *bounded* if there exists a $k \in \mathbb{N}$ such that $\mathsf{stage}_{\Pi,D}(R) \leq k$, for every database D of $\mathsf{edb}(\Pi)$.

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 UCQ s, but actually as FO queries.

Theorem 38.14

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 36.2 that there is an infinitary UCQ q' over $\mathbf{S}$ such that $q \equiv q'$. Recall that q' is defined by exhaustively applying the $\text{Unfold}_\Pi(\cdot)$ operator, introduced in Chapter 36, 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 $\text{Unfold}_\Pi(\cdot)$ 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 36.6, we get that q' is preserved under homomorphisms. This in turn implies, due to Theorem 28.11, 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

[1] Note that the proof of Proposition 36.2 was given for Boolean queries, but it can be extended to non-Boolean queries; this extension was left as an exercise.

$q \equiv \hat{q}$, by Proposition 35.17, 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 $\mathsf{stage}_{\Pi,D}(R) \leq k$, for every database D of $\mathsf{edb}(\Pi)$, 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 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 $q \equiv q''$, we have 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. $\square$

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 $\mathsf{idb}(\Pi) = \{P_1, \ldots, P_n\}$, the Datalog rule ρ is

$$R(x_1^1, \ldots, x_{\mathrm{ar}(P_1)}^1, \ldots, x_1^n, \ldots, x_{\mathrm{ar}(P_n)}^n) \coloneq P_1(x_1^1, \ldots, x_{\mathrm{ar}(P_1)}^1), \ldots, \\ P_n(x_1^n, \ldots, x_{\mathrm{ar}(P_n)}^n)$$

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 38.15

Datalog-PBoundedness and Datalog-QBoundedness are undecidable.

Exercises

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. Let q, q' be $\text{CQ}^{\neq}$s. Prove that $q \subseteq q'$ if, and only if, for every $i \in [n]$, there exists $j \in [m]$ such that $q_i \subseteq q'_j$.

Exercise 4.6. The language $\text{CQ}^{<}$ is defined in the same way as $\text{CQ}^{\neq}$ (see Definition 30.1), 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 4.7. 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 4.8. Prove that the sentences $\psi_k(R\bar{S})$ and $\psi_k(\bar{R}S)$, used in the proof of Theorem 32.5, are true in almost all databases of $\mathbf{S} = \{R[1], S[1]\}$.

Exercise 4.9. In general, Theorem 32.5 (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 32.5, given in Chapter 30, 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 4.10. Use Exercise 4.9 (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 32.7 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 4.11. 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 4.12. Recall the estimate used in the proof of Theorem 32.7

$$F_n^{=} = \sum_{k \leq \lfloor n/2 \rfloor} \binom{n}{k}\binom{n-k}{k}.$$

Show that

$$\lim_{n\to\infty} \frac{F_n^{=}}{3^n} = 0.$$

Exercise 4.13. Recall that Theorem 32.5 (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 32.6 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 [22].

Exercise 4.14. Give an example of an FO sentence *with constants* for which Theorem 32.5 (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 4.15. Recall that Theorem 34.1 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 $\mathbf{L}_{\mathbf{C}}$ 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 $\mathbf{L}_{\mathbf{C}}$, 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 4.16. Use the translation of Theorem 34.6 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 4.17. Recall that Proposition 34.11 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 4.18. There is a different notion of locality, known in the literature as *Hanf-locality* (as opposed to *Gaifman-locality* used in Chapter 34). 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 34.3 for Hanf-locality instead of Gaifman locality. To do so, show, by extending the argument in the proof of Proposition 34.11, 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 4.19. Consider a restriction of $\mathbf{L}_{\mathbf{C}}$ 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 4.20. 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 4.21. 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, $q(D_G) = \texttt{true}$ if and only if G is *not* 2-colorable, where D_G stores G via the binary relation $\mathrm{Edge}(\cdot, \cdot)$.

Exercise 4.22. Prove Theorem 35.8.

Exercise 4.23. Prove Lemma 35.10.

Exercise 4.24. 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 4.25. 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 4.26. 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 4.27. 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

$$R(\bar{x}) \coloneq \ldots, P(\bar{y}), \ldots$$

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 4.28. 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 37.1. Show that M accepts w if and only if $q(D) = \mathtt{true}$.

Exercise 4.29. 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 37.1 holds even if we focus on guarded Datalog queries.

Exercise 4.30. 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 4.31. 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 $q(D) = \mathtt{true}$ is PTime-complete.

Exercise 4.32. Show that the evaluation problem for linear Datalog queries is PSpace-complete in combined complexity, and NLogSpace-complete in data complexity.

Exercise 4.33. Show that the evaluation problem for non-recursive Datalog queries is PSpace-complete in combined complexity, and in DLogSpace in data complexity.

Bibliographic Comments

To be done.

Part V

Uncertainty

39

Incomplete Databases

So far we assumed that all information in databases was complete: we have a set Const from which all entries in databases come, and for every fact $R(\bar{a})$ in a database, every element of the tuple $\bar{a}$ comes from Const. The reality of course is different: information is often missing, and databases are full of entries shown as null. These null elements can have different semantics: sometimes they mean that an attribute is not known (e.g., date of birth), sometimes they mean that an attribute is not applicable (e.g., date of death for a living person), and sometimes we simply do not have any information why no data value was provided and a null replaced it.

In the next two chapters we study the most commonly considered model of nulls representing missing, and currently unknown data. We define this model using the framework of sets of atoms, and explain what such incomplete instances mean. In fact, they can represent multiple complete databases, and thus query answering must be consistent with answers obtained in all of them.

In this chapter, we present the model of incomplete data and the notion of *certain* query answers. We then analyze its complexity (which is very high), look at what real-world databases do (obviously they keep the complexity low), thus arriving at the inevitable mismatch between the correct way of answering queries and what happens in reality. In the following chapter, we look at the ways of bridging this gap.

A Formal Model of Incomplete Data and Nulls

The idea of the model is simple: some entries in databases are replaced by variables, which are interpreted as values that exist but are currently unknown. To separate them from variables used in queries, we assume that the set Var of variables has a countably infinite subset Null of nulls whose elements will be denoted by $\perp_1, \perp_2, \perp_3, \ldots$. Unlike other elements of Var, however, they will not be explicitly referenced in queries. An example of such a database is shown in Figure 39.1. Some entries in the `Profession` table contain nulls,

Person

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

Profession

pid	prname
1	singer
1	$\bot_1$
1	$\bot_2$
2	singer
3	$\bot_1$
3	author

Fig. 39.1: A incomplete database of the schema in Example 3.2.

denoted by $\bot_1$ and $\bot_2$. They stand for two currently unknown professions of the person with id `1`, namely **`Aretha`**. Moreover, one of the nulls, $\bot_1$, is also a profession of the person with id `3`, i.e., `Bob`. Thus, while we do not know what these professions are, we do know that Aretha and Bob share a profession.

Definition 39.1: Incomplete Database Instances

An *incomplete database instance* $\mathcal{I}$ of a schema $\mathbf{S}$ is a finite set of atoms over $\mathbf{S}$, where all variables come from the set Null.

Since an incomplete instance $\mathcal{I}$ may contain both constants and nulls, so can its domain $\mathrm{Dom}(\mathcal{I})$. We thus use the notation $\mathrm{Dom}_{\mathsf{Const}}(\mathcal{I})$ for $\mathrm{Dom}(\mathcal{I}) \cap \mathsf{Const}$ and $\mathrm{Dom}_{\mathsf{Null}}(\mathcal{I})$ for $\mathrm{Dom}(\mathcal{I}) \cap \mathsf{Null}$.

An incomplete instance $\mathcal{I}$ may represent multiple complete instances. To start with, nulls in it may be replaces by elements of Const, i.e., by their real, but currently unknown values. If this is the only way to obtain further information about $\mathcal{I}$, one refers to the semantics under the *Closed World Assumption*, or CWA-semantics. Another, more permissive interpretation, permits not only replacing nulls with constant values but also adding tuples that may have been missing in $\mathcal{I}$. This is known as the semantics under the *Open World Assumption*, or OWA.

To define them formally, we use the machinery of homomorphisms. Recall that a homomorphism h on a set of atoms (i.e., an incomplete instance) $\mathcal{I}$ is a mapping h defined on $\mathrm{Dom}(\mathcal{I})$ such that $h(a) = a$ whenever $a \in \mathrm{Dom}_{\mathsf{Const}}(\mathcal{I})$, and otherwise $h(a)$ is in $\mathsf{Const} \cup \mathsf{Var}$. In providing semantics of incompleteness, we want to replace nulls with constant values rather than other nulls. Thus, we look at homomorphisms whose image is a subset of Const. These are called *valuations*.

Definition 39.2: Closed and Open-World Semantics

Given an incomplete instance $\mathcal{I}$ of schema $\mathbf{S}$, its CWA-semantics is defined as

$$[\![\mathcal{I}]\!]_{\text{CWA}} \;=\; \{D \in \text{Inst}(\mathbf{S}) \mid D = h(\mathcal{I}) \text{ and } h \text{ is a valuation}\}\,.$$

Its OWA-semantics is defined as

$$[\![D]\!]_{\text{OWA}} \;=\; \{D \in \text{Inst}(\mathbf{S}) \mid \text{ there is a valuation } h : \mathcal{I} \to D\}\,.$$

It is easy to see that $D \in [\![\mathcal{I}]\!]_{\text{OWA}}$ iff $D \supseteq D'$ for some $D' \in [\![\mathcal{I}]\!]_{\text{CWA}}$.

The database in Figure 3.1 of Example 3.2 belongs to $[\![\mathcal{I}]\!]_{\text{OWA}}$ for the instance $\mathcal{I}$ in Figure 39.1. It does not belong to $[\![\mathcal{I}]\!]_{\text{CWA}}$ due to the presence in it of relation `City`, and information about the person with `pid` 4, namely Freddie. However, if all the tuples about that person are eliminated from the instance in Figure 3.1 together with the relation `City`, then the resulting database will be in $[\![\mathcal{I}]\!]_{\text{CWA}}$.

Certain Answers

Given an incomplete instance $\mathcal{I}$ and a query q written in some query language, what is $q(\mathcal{I})$? Since the instance $\mathcal{I}$ represents multiple complete instances (those in $[\![\mathcal{I}]\!]_{\text{CWA}}$ or $[\![\mathcal{I}]\!]_{\text{OWA}}$, depending on the semantics), $q(\mathcal{I})$ must take into account query answers $q(D)$ for all D that $\mathcal{I}$ can represent.

To arrive at the definition of answers to q, consider initially Boolean queries. That is, $q(D)$ is either `true` or `false`. When can we say that the answer to q in $\mathcal{I}$ is `true`? This can be asserted with certainty only if $q(D) =$ `true` for all $D \in [\![\mathcal{I}]\!]$ (where $[\![\;]\!]$ is one of the semantics we use). In that case, we say that the *certain answer* to q over $\mathcal{I}$ is `true`; otherwise it is `false`. From now on, we shall use the notation $\mathsf{cert}_*(q, \mathcal{I})$ for certain answers, where $*$ is OWA or CWA. Recall that `true` is viewed as the set $\{()\}$ containing the empty tuple while `false` is viewed as the empty set. Thus, $\mathsf{cert}_*(q, \mathcal{I}) = \bigcap\{q(D) \mid D \in [\![\mathcal{I}]\!]_*\}$.

Next, we move to queries of arbitrary arity. Suppose we have an incomplete instance $\mathcal{I}$ and a tuple $\bar{a}$ that consists of elements of $\text{Dom}_{\mathsf{Const}}(\mathcal{I})$. When can we state with certainty that this tuple is in the answer to q on $\mathcal{I}$? Again, this happens if it is in all the answers $q(D)$, for $D \in [\![\mathcal{I}]\!]_*$. So again we can define $\mathsf{cert}_*(q, \mathcal{I})$ as $\bigcap\{q(D) \mid D \in [\![\mathcal{I}]\!]_*\}$.

The definition gets more complicated if a tuple $\bar{a}$ contains both nulls and constants. Then the previous definition does not work: if $\bar{a}$ contains nulls, it cannot belong to $q(D)$ for any complete D, as D itself contains no nulls. But how can a query return, with certainty, a tuple with nulls? To see this, consider a simple query that returns the relation `Profession` in Figure 39.1. We know with certainty that the tuple $\bar{a} = (1, \bot_1)$ is in it, and thus we expect it to be in the answer. The reason we know this with certainty is that for every valuation h, we have $h(\bar{a}) \in h($`Profession`$)$. We now extend this to arbitrary queries.

Definition 39.3: Certain Answers

Given a query q, and an incomplete instance $\mathcal{I}$, its certain answers under CWA and under OWA are defined by:

$$\bar{a} \in \mathsf{cert}_{\text{CWA}}(q, \mathcal{I}) \;\Leftrightarrow\; h(\bar{a}) \in q(h(\mathcal{I})) \text{ for every valuation } h$$

$$\bar{a} \in \mathsf{cert}_{\text{OWA}}(q, \mathcal{I}) \;\Leftrightarrow\; h(\bar{a}) \in q(D) \text{ for every valuation } h : \mathcal{I} \to D$$

If $\mathcal{I}$ has no nulls, then $[\![\mathcal{I}]\!]_{\text{CWA}} = \{\mathcal{I}\}$ and thus in this case $\mathsf{cert}_{\text{CWA}}(q, \mathcal{I}) = q(\mathcal{I})$. Hence, certain query answering under CWA properly generalizes query answering in complete databases.

For tuples with no nulls in them, this definition of certain answers coincides with the intersection of query answers in all databases represented by $\mathcal{I}$.

Proposition 39.4

Given a query q, an incomplete instance $\mathcal{I}$, and a tuple $\bar{a}$ without nulls, $\bar{a} \in \mathsf{cert}_*(q, \mathcal{I})$ iff $\bar{a} \in \bigcap\{q(D) \mid D \in [\![\mathcal{I}]\!]_*\}$, where $*$ is OWA or CWA.

Proof. We show this for CWA; for OWA the proof is very similar. Assume $\bar{a} \in \mathsf{cert}_{\text{CWA}}(q, \mathcal{I})$ and take an arbitrary valuation h. Then $h(\bar{a}) = \bar{a}$ since $\bar{a}$ has no nulls and thus $\bar{a} \in q(h(\mathcal{I}))$, proving $\bar{a} \in \bigcap\{q(D) \mid D \in [\![\mathcal{I}]\!]_{\text{CWA}}\}$. Conversely, assume $\bar{a} \in \bigcap\{q(D) \mid D \in [\![\mathcal{I}]\!]_{\text{CWA}}\}$. Take any valuation h; then $h(\mathcal{I}) \in [\![\mathcal{I}]\!]_{\text{CWA}}$ and hence $\bar{a} \in q(h(\mathcal{I}))$. Since $\bar{a}$ has no nulls, we have $h(\bar{a}) = \bar{a} \in q(h(\mathcal{I}))$, thus proving $\bar{a} \in \mathsf{cert}_{\text{CWA}}(q, \mathcal{I})$.

Example 39.5: Certain Answers to Queries

We consider two queries similar to those in Example 3.2, now asked on the incomplete instance $\mathcal{I}$ shown in Figure 39.1. As the language we use FO, with one clarification: as mentioned earlier, variables in its formulae use come from $\mathsf{Var} - \mathsf{Null}$. That is, elements of Null cannot be used as explicit constants in formulae.

The first query asks for names of persons with two different professions:

$$\exists x \exists z \exists u_1 \exists u_2 \big(\text{Person}(x, y, z) \wedge \\ \text{Profession}(x, u_1) \wedge \text{Profession}(x, u_2) \wedge \neg(u_1 = u_2)\big). \tag{39.1}$$

The certain answer to this query is empty, under both OWA and CWA interpretation. Indeed, Billie has only one profession, and even though it may seem like Aretha and Bob have multiple professions, there are possible completions of $\mathcal{I}$ where this is not the case. For example, a valuation where $h(\perp_1) = h(\perp_2) =$ 'singer' results in a complete database where Aretha has only one profession, while a valuation with $h(\bot_1) =$ author results in a complete database where the same is true of Bob.

The second asks for pairs of persons who share a profession:

$$\exists x \exists x' \exists z \exists z' \exists u \big(\text{Person}(x, y, z) \land \text{Person}(x', y', z') \land \\ (\text{Profession}(x, u) \land \text{Profession}(x', u))\big).$$

This query will return pairs of names, including all pairs of the form (n, n), but also (Aretha,Billie) and (Billie,Aretha) who share the profession 'singer', as well as (Aretha,Bob) and (Bob,Aretha), who too share a profession, $\bot_1$, though we do not know what it is. Still, in every database D that is equal or contains $h(\mathcal{I})$, this common profession will be represented by $h(\bot_1)$, and thus these pairs will be output by the query, making them certain answers.

It might seem from these examples that certain answers are the same under OWA and CWA. This is not the case however, and we shall see soon that they differ dramatically in terms of their complexity. To give a simple example where certain answers are different, consider a Boolean query q given by

$$\forall x \forall y \forall z \, \big(\text{Person}(x, y, z) \to \exists u \, \text{Profession}(x, u)\big)$$

asking whether every person has a recorded profession. Then $\mathsf{cert}_{\text{CWA}}(q, \mathcal{I}) =$ `true` while $\mathsf{cert}_{\text{OWA}}(q, \mathcal{I}) =$ `false`. For example, a database $D \in [\![\mathcal{I}]\!]_{\text{OWA}}$ obtained by giving values to nulls and by adding a tuple to table `Person` with an id that is not present in table `Profession` violates the condition expressed by q.

Complexity of Certain Answers

Next, we analyze the complexity of certain answers, and see that it is significantly higher than the complexity of query evaluation.

Theorem 39.6

- Under the CWA semantics, data complexity of certain answers to FO queries is coNP-complete and their combined complexity is PSpace-complete.
- Under the OWA semantics, combined complexity of certain answers to FO queries is undecidable.

Proof. Upper bounds. We look at the complement problem, namely whether $\bar{a} \notin \mathsf{cert}_{\text{CWA}}(q, \mathcal{I})$ for an incomplete instance $\mathcal{I}$, a query q, and a tuple $\bar{a}$. We will show that this problem is in PSpace if all of $\bar{a}, \mathcal{I}$, and q are inputs, and in NP, if q is fixed. This will suffice, since PSPACE is closed under complementation. Note that $\bar{a} \notin \mathsf{cert}_{\text{CWA}}(q, \mathcal{I})$ iff $h(\bar{a}) \notin q(h(\mathcal{I}))$ for some valuation. This looks like the proof: just guess that valuation h. However, to prove membership in NP, a guess must be of polynomial size, but the above h could be an arbitrary valuation, i.e., its range could be an arbitrary subset of the infinite set Const. Instead, we show that if $\bar{a} \notin \mathsf{cert}_{\text{CWA}}(q, \mathcal{I})$, then such h can be chosen among a finite set of valuations with a fixed range that is linear in the size of $\text{Dom}(\mathcal{I})$. This will prove membership in NP for a fixed q, as well as membership in PSPACE for combined complexity. Indeed, PSPACE is closed under nondeterminism (hence one can guess h) and checking $h(\bar{a}) \notin q(h(\mathcal{I}))$ is in PSPACE, since q is an FO query.

Assume that $\text{Dom}_{\mathsf{Null}}(\mathcal{I}) = \{\bot_1, \ldots, \bot_m\}$. Let $C = \{c_1, \ldots, c_m\}$ be a set of m elements of Const disjoint from $\text{Dom}(\mathcal{I})$ and $\text{Dom}(q)$, the set of constants mentioned in q. We claim that if $h(\bar{a}) \notin q(h(\mathcal{I}))$ for some h, then this also happens for some h whose range is contained in $\text{Dom}_{\mathsf{Const}}(\mathcal{I}) \cup \text{Dom}(q) \cup C$; clearly this will suffice. Consider h such that $h(\bar{a}) \notin q(h(\mathcal{I}))$ and let $i_1, \ldots, i_k$ be such that $h(\bot_{i_j}) \notin \text{Dom}_{\mathsf{Const}}(\mathcal{I}) \cup \text{Dom}(q)$ for $j \in [k]$. We then define h' that coincides with h everywhere except that it sets $h'(\bot_{i_j}) = c_{i_j}$ for $j \in [k]$. Let $\pi : \mathsf{Const} \to U$ be a bijection that is the identity on $\text{Dom}_{\mathsf{Const}}(\mathcal{I}) \cup \text{Dom}(q)$, sends each c_{i_j} to $h(\bot_{i_j})$, and is arbitrarily defined elsewhere. Note then that $\pi(h'(a)) = h(a)$ for every $a \in \text{Dom}(\mathcal{I}) \cup \text{Dom}(q)$. If we now assume that $h'(\bar{a}) \in q(h'(\mathcal{I}))$, then by genericity of FO queries (see Exercise 1.2) that would mean that $\pi(h'(\bar{a})) \in q\big(\pi(h'(\mathcal{I}))\big)$, i.e., $h(\bar{a}) \in q(h(\mathcal{I}))$, which is a contradiction. Hence, we can find a witness valuation with the range contained in $\text{Dom}_{\mathsf{Const}}(\mathcal{I}) \cup \text{Dom}(q) \cup C$. This finishes the proof of upper bounds.

Lower bounds. The PSPACE lower bound for combined complexity is already known for queries on databases with no nulls. Indeed, for such databases $\mathsf{cert}_{\text{CWA}}(q, \mathcal{I}) = q(\mathcal{I})$, and the latter is known to be PSPACE-hard.

We next show coNP-hardness of data complexity. Actually, we will look at the complement problem, which for Boolean queries is whether $\mathsf{cert}_{\text{CWA}}(q, \mathcal{I}) =$ `false` and prove it to be NP-hard, for a fixed query q. For this we reduce from graph 3-colorability, a well-known NP-complete problem. Given a graph $G = \langle V, E \rangle$, it is 3-colorable if there a coloring map $c : V \to \{r, g, b\}$ such that for every edge $(v, v') \in E$, the colors $c(v)$ and $c(v')$ are different. Now given such a graph with $V = \{v_1, \ldots, v_n\}$, we create an incomplete instance $\mathcal{I}_G$ of the schema containing relations $E[2]$ and $C[2]$ such that E is the edge relation of the graph, and C contains tuples $(v_1, \bot_1), \ldots, (v_n, \bot_n)$, where $\bot_1, \ldots, \bot_n$ are distinct nulls. The intuition is that these are yet unknown colors assigned to vertices; a database $D \in [\![\mathcal{I}_G]\!]_{\text{CWA}}$ then assigns colors to vertices. We next define a Boolean FO query q stating that such an assignment is not a 3-coloring. Specifically, this happens if

1. there are 4 or more different colors (values of the second column of C); or
2. there is an edge whose endpoints are given the same color.

The first condition is expressed by an FO query

$$\exists x_1 \exists x_2 \exists x_3 \exists x_4 \exists y_1 \exists y_2 \exists y_3 \exists y_4 \ \big(\bigwedge_{i=1}^{4} C(x_i, y_i) \wedge \text{diff}(y_1, y_2, y_3, y_4) \big)$$

where the formula diff(y_1, y_2, y_3, y_4) says that all of y_1, y_2, y_3, y_4 are pairwise distinct; the second condition is expressed by

$$\exists x \exists y \exists u \ \big(C(x, u) \wedge C(y, u) \wedge \neg(x = y) \big) .$$

Thus, their disjunction is an FO query q such that q is true in $h(\mathcal{I}_G)$ iff h, that assigns values to nulls, is not a 3-coloring of G. Hence $\mathsf{cert}_{\text{CWA}}(q, \mathcal{I}_G) = \texttt{false}$ iff there is at least one h that is a 3-coloring, i.e., iff the graph is 3-colorable. Hence, checking $\mathsf{cert}_{\text{CWA}}(q, \mathcal{I}_G) = \texttt{false}$ is NP-hard in data complexity, and thus checking $\mathsf{cert}_{\text{CWA}}(q, \mathcal{I}_G) = \texttt{true}$ is coNP-hard.

We finally show undecidability of the combined complexity of certain answers under OWA. Consider a Boolean FO query q given by a sentence φ, and let $\mathcal{I}_\emptyset$ be the empty instance, where every relation has no tuples. Then an arbitrary database D is in $[\![\mathcal{I}_\emptyset]\!]_{\text{OWA}}$. Hence $\mathsf{cert}_{\text{OWA}}(q, \mathcal{I}_\emptyset) = \texttt{true}$ iff φ is a valid sentence, i.e., true in every database. But we know from Chapter 8 that this is an undecidable condition. □

Note that Theorem 39.6 says nothing about data complexity of finding certain answers under OWA. Indeed, to prove undecidability, we use the fixed empty database and it was the query that varied. This alone rules out certain answers under OWA as a viable query answering strategy, but we can show an even stronger undecidability of data complexity. That is, there is a fixed FO query q such that $\mathsf{cert}_{\text{OWA}}(q, \mathcal{I})$ is undecidable when $\mathcal{I}$ is the only input; see Exercise 5.1.

Note also that in the coNP-hardness reduction in the proof, we used an example of an incomplete instance in which every nulls occurs exactly once. Furthermore, the fixed query used in the reduction is a union of CQs with inequalities, meaning that already for them the problem of finding certain answers is intractable.

SQL Nulls and Three-Valued Logic

If finding certain answers is intractable even under the CWA, what do real-life databases do? Clearly they cannot implement intractable exponential-time algorithms. They do not indeed, and instead they use a very different solution: SQL resorts to a *3-valued logic* with an extra truth value `unknown`, which is reserved for comparisons involving nulls. In fact in relational DBMSs there is just one single `null` value, i.e., we cannot indicate that two values currently unknown are equal. But this restriction, as we just saw, does not lower the complexity.

The 3-valued logic, used by SQL, and also known as Kleene's logic, is as follows:

- On `true` and `false`, operations $\wedge, \vee$, and $\neg$ have their standard Boolean interpretation;
- $\neg$`unknown` $=$ `unknown` (if we do not known a truth value, we do not know its negation);
- `true` $\vee$ `unknown` $=$ `true` (disjunction of `true` with anything is `true`) while `false` $\vee$ `unknown` $=$ `unknown` $\vee$ `unknown` $=$ `unknown` (if one truth value is unknown its disjunction with anything other than `true` is unknown);
- `false` $\wedge$ `unknown` $=$ `false` (conjunction of `false` with anything is `false`) while `true` $\wedge$ `unknown` $=$ `unknown` $\wedge$ `unknown` $=$ `unknown` (if one truth value is unknown its conjunction with anything other than `false` is unknown).

For the definition of satisfaction, it is no longer sufficient to use only the notion of $(\mathcal{I}, \eta) \models \varphi$, which says that φ evaluates to `true` under the assignment η in $\mathcal{I}$. In Boolean logic, the alternative to it is that φ evaluates to `false`, but now we need to account for the possibility of φ evaluating to `unknown`. The key rule that SQL uses is the following: *If at least one argument of a comparison is null, then the result of the comparison is* `unknown`.

Definition 39.7: $\mathsf{Eval}(\varphi, \mathcal{I}, \eta)$

$\mathsf{Eval}(\varphi, \mathcal{I}, \eta)$, the truth value to which φ evaluates in an incomplete instance $\mathcal{I}$ under assignment η, is defined as follows.

- $\mathsf{Eval}(x = y, \mathcal{I}, \eta)$ is

$$\begin{cases} \texttt{true} & \text{if } \eta(x), \eta(y) \in \mathrm{Dom}_{\mathsf{Const}}(\mathcal{I}) \text{ and } \eta(x) = \eta(y) \\ \texttt{false} & \text{if } \eta(x), \eta(y) \in \mathrm{Dom}_{\mathsf{Const}}(\mathcal{I}) \text{ and } \eta(x) \neq \eta(y) \\ \texttt{unknown} & \text{if } \eta(x) \in \mathrm{Dom}_{\mathsf{Null}}(\mathcal{I}) \text{ or } \eta(y) \in \mathrm{Dom}_{\mathsf{Null}}(\mathcal{I}) \end{cases}$$

- If a is a constant, then $\mathsf{Eval}(x = a, \mathcal{I}, \eta)$ is

$$\begin{cases} \texttt{true} & \text{if } \eta(x) = a \\ \texttt{false} & \text{if } \eta(x) \in \mathrm{Dom}_{\mathsf{Const}}(\mathcal{I}) \text{ and } \eta(x) \neq a \\ \texttt{unknown} & \text{if } \eta(x) \in \mathrm{Dom}_{\mathsf{Null}}(\mathcal{I}) \end{cases}$$

- $\mathsf{Eval}(R(u_1, \ldots, u_k), \mathcal{I}, \eta)$ is

$$\begin{cases} \texttt{true} & \text{if } R(\eta(u_1), \ldots, \eta(u_k)) \in \mathcal{I} \\ \texttt{false} & \text{otherwise} \end{cases}$$

- $\mathsf{Eval}(\varphi_1 \wedge \varphi_2, \mathcal{I}, \eta) = \mathsf{Eval}(\varphi_1, \mathcal{I}, \eta) \wedge \mathsf{Eval}(\varphi_2, \mathcal{I}, \eta)$
- $\mathsf{Eval}(\varphi_1 \vee \varphi_2, \mathcal{I}, \eta) = \mathsf{Eval}(\varphi_1, \mathcal{I}, \eta) \vee \mathsf{Eval}(\varphi_2, \mathcal{I}, \eta)$
- $\mathsf{Eval}(\neg\varphi_1, \mathcal{I}, \eta) = \neg\mathsf{Eval}(\varphi_1, \mathcal{I}, \eta)$
- $\mathsf{Eval}(\exists x\, \psi, \mathcal{I}, \eta) = \bigvee_{a \in \mathrm{Dom}(\mathcal{I})} \mathsf{Eval}(\psi, \mathcal{I}, \eta[x/a])$.
- $\mathsf{Eval}(\exists x\, \psi, \mathcal{I}, \eta) = \bigvee_{a \in \mathrm{Dom}(\mathcal{I})} \mathsf{Eval}(\psi, \mathcal{I}, \eta[x/a])$.

A formula then defines a query $q = (\varphi, \bar{x})$ whose result on an incomplete instance $\mathcal{I}$ under the 3-valued logic interpretation is defined as

$$q_{\mathsf{3vl}}(\mathcal{I}) \;=\; \{\eta(\bar{x}) \mid \mathsf{Eval}(\varphi, \mathcal{I}, \eta) = \mathtt{true}\}\,.$$

Thus, in computing query answers, we are only interested in tuples that are `true`, dismissing not only `false` but also `unknown`; in other words, `unknown` is a truth value used for evaluating conditions but not for producing query output. Notice that a query evaluated on an incomplete instance may return tuples with nulls.

To give an example, consider query (39.1) from Example 39.5, evaluated on the incomplete instance $\mathcal{I}$ in Figure 39.1. It asks for names of people having two different professions. This query will return nothing under the 3-valued interpretation as all comparisons $\neg(u_1 = u_2)$ in it will evaluate to either `false`, when u_1 and u_2 are the same, or `unknown`, when one of them is a null. Thus, when the free variable y is instantiated as Aretha or Bob (`pid` is 1 or 3), the formula evaluates to `unknown`, and when it is instantiated as Billie, with `pid` 2 and a single profession recorded, the formula is `false`.

By a simple extension of the query evaluation algorithm for FO queries, we can show

Proposition 39.8

Query evaluation of FO queries under the 3-valued semantics is in DLogSpace.

Thus, the 3-valued semantics cannot compute certain answers. The question is then how close its answers are to certain answers. In general, they can deviate from each other in two possible ways:

- SQL evaluation can produce false negatives, i.e., miss some certain answers: that is, $q_{\mathsf{3vl}}(\mathcal{I}) \subsetneq \mathsf{cert}_{\mathrm{CWA}}(q, \mathcal{I})$;
- SQL evaluation can produce false positives, i.e., give answers which are now certain: $\mathsf{cert}_{\mathrm{CWA}}(q, \mathcal{I}) \subsetneq q_{\mathsf{3vl}}(\mathcal{I})$.

We now show that the evaluation strategy based on 3-valued logic that is chosen by SQL can produce *both*. In the next chapter we study alternative query evaluation strategies that provide some guarantees on the quality of the answers.

Proposition 39.9

Query evaluation of FO queries under the 3-valued semantics can produce both false positives and false negatives.

Proof. Let $\mathcal{I}$ contain a binary relation R with tuples $(1, \bot)$ and $(2, \bot)$. Consider a Boolean query q given by $\exists x \exists y \ \big(R(x, y) \wedge \neg(x = y)\big\}$. Then $\mathsf{cert}_{\text{CWA}}(q, \mathcal{I}) = \mathtt{true}$. Indeed, $h(\bot)$ cannot be both 1 and 2 and thus in $h(\mathcal{I})$ at least one tuple will have different values. On the other hand, $q_{\mathsf{3vl}}(\mathcal{I}) = \emptyset$, since $\mathsf{Eval}(q, \mathcal{I}) = \mathtt{unknown}$ due to all the conditions $x = y$ evaluating to $\mathtt{unknown}$. Since $\mathtt{true} = \{()\}$, we have an example of a false negative.

To give an example of a false positive, consider an incomplete instance $\mathcal{I}$ with two unary relations R and S containing (1) and $(\bot)$ respectively. Let q be given by $\varphi(x) = R(x) \wedge \neg\big(R(x) \wedge \neg S(y)\big)$. Then its certain answer is empty, as witnessed by $h(\mathcal{I})$ with $h(\bot) = 2$. However, a routine inspection shows $q_{\mathsf{3vl}}(\mathcal{I}) = \{(1)\}$, thus producing a false positive. $\square$

Computing Certain Answers

To summarize the previous chapter, we have seen that:

- finding certain answers for FO queries is computationally intractable;
- SQL resorts to a 3-valued logic to achieve low complexity of query evaluation; however, this leads to the presence of both false positives and false negatives in query answers.

In this chapter we look at three different query evaluation strategies over incomplete instances that combine efficiency and correctness.

Naïve evaluation. One can try to evaluate a query naïve ly, as if nulls were actual values. This cannot always give us certain answers – otherwise they would have had tractable data complexity – but naïve evaluation avoids false negatives and for some queries actually computes certain answers.

Approximation: absolute guarantees. We show to evaluate all FO queries tractably, over incomplete instances, without producing false positives. This is achieved by means of query translation: a query q is translated into a query $q^{\mathbf{t}}$ whose evaluation is guaranteed to produce a subset of certain answers.

Approximation: probabilistic guarantees. A different approach is to compute the probability that a given tuple is in the answer for a randomly picked assignment of values to nulls. It turns out that for all FO queries, their naïve evaluation gives us correct answers with probability 1.

Naïve Evaluation and Correctness of Answers

We first need to formalize the idea of answering a query on an incomplete instance treating nulls as new constants. For this, we use the standard notion of satisfaction of FO formulae, but extended to incomplete instances. More precisely, to define $(\mathcal{I}, \eta) \models \varphi$, we follow, word by word, Definition 3.3 of satisfaction of FO formulae. Since Const and Null are disjoint sets, it means that equalities $\bot = a$ for a constant a and $\bot = \bot'$ for two distinct nulls evaluate to `false`, while $\bot = \bot$ evaluates to `true`.

Definition 40.1: Naïve Evaluation

The naïve evaluation of an FO query $q = (\varphi, \bar{x})$ on an incomplete instance $\mathcal{I}$ is defined as

$$\mathsf{naïve}(q, \mathcal{I}) \;=\; \{\eta(\bar{x}) \mid (\mathcal{I}, \eta) \models \varphi\}$$

where η ranges over all assignments $\bar{x} \mapsto \mathrm{Dom}(\mathcal{I}) \cup \mathrm{Dom}(q)$.

In other words, the definition is really unchanged compared to the usual definition of FO query evaluation except that free variables can now be assigned nulls, and not only constants. For example, given an incomplete instance with atoms $R(1, \bot), R(\bot, \bot), R(\bot, \bot')$, the naïve evaluation of the query $R(x, y) \wedge x = y$ produces $(\bot, \bot)$, while the naïve evaluation of $R(x, y) \wedge \neg(x = y)$ produces $(1, \bot)$ and $(\bot, \bot')$.

What is the relationship between naïve evaluation and certain answers? We saw that SQL's 3-valued logic-based evaluation can produce both false positives and false negatives. Naïve evaluation, on the other hand, avoids false negatives.

Proposition 40.2

For every FO query q, we have $\mathsf{cert}_{\text{OWA}}(q, \mathcal{I}) \subseteq \mathsf{cert}_{\text{CWA}}(q, \mathcal{I}) \subseteq \mathsf{naïve}(q, \mathcal{I})$.

Proof. The first inclusion is an immediate consequence of $[\![\mathcal{I}]\!]_{\text{CWA}} \subseteq [\![\mathcal{I}]\!]_{\text{OWA}}$. Assume next that q is given as $(\varphi, \bar{x})$ and $\bar{a} \in \mathsf{cert}_{\text{CWA}}(q, \mathcal{I})$. We say that a valuation h on $\mathrm{Dom}_{\mathsf{Null}}(\mathcal{I})$ is *simple* if it is a bijection and furthermore $h(\bot) \notin \mathrm{Dom}_{\mathsf{Const}}(\mathcal{I}) \cup \mathrm{Dom}(\varphi)$ for every $\bot \in \mathrm{Dom}_{\mathsf{Null}}(\mathcal{I})$. If $\bar{a} \in \mathsf{cert}_{\text{OWA}}(q, \mathcal{I})$ then $h(\bar{a}) \in q(h(\mathcal{I}))$ for every valuation h, and in particular for every simple valuation h. The latter means $(h(\mathcal{I}), h \circ \eta) \models \varphi$, where η assigns $\bar{a}$ to $\bar{x}$. Using $\mathrm{Dom}(\varphi)$-genericity of φ (see Exercise 1.2) and applying it to the bijection h^{-1}, we conclude that $(h^{-1}(h(\mathcal{I})), h^{-1} \circ h \circ \eta) \models \varphi$, i.e., $(\mathcal{I}, \eta) \models \varphi$, and thus $\bar{a} \in \mathsf{naïve}(q, \mathcal{I})$. □

At the same time, naïve evaluation can produce false positives. Consider an FO query $\varphi(x) = R(x) \wedge \neg S(x)$ evaluated on a database with facts $R(1), S(\bot)$. Its naive evaluation outputs 1, while certain answers are empty, as witnessed by the valuation $h(\bot) = 1$.

Nonetheless, for a large class of queries, naïve evaluation produces exactly the certain answers. The class depends on the semantics of incompleteness, and is larger for CWA. Indeed, under CWA, there are fewer instances in $[\![\mathcal{I}]\!]_{\text{CWA}}$ than in $[\![\mathcal{I}]\!]_{\text{OWA}}$ and thus it is "easier" for a tuple to be a certain answer.

Definition 40.3: Positive Formulae with Atomic Guards (PAG)

The class PAG of *positive atomic formulae with guards* is given by the following rules:

- every atomic formula is in PAG;
- if φ and ψ are in PAG, then so are $\varphi \wedge \psi$ and $\varphi \vee \psi$;
- if φ is in PAG, then so are $\exists x\, \varphi$ and $\forall x\, \varphi$;
- if φ is in PAG, and $\alpha(\bar{x})$ is an atomic formula in which no variable repeats, then $\forall \bar{x}\, \big(\alpha(\bar{x}) \rightarrow \varphi\big)$ is in PAG.

A PAG query is an FO query given by a PAG formula.

This class of formulae excludes the negation from FO, while keeping everything else, and replaces negation by a weak form that has a guarded implication $\alpha(\bar{x}) \rightarrow \varphi$ under the scope of a universal quantifier. These are rather common queries of the shape "objects that participate in every relationship of a certain kind", e.g., customers who buy every product. Using the schema of our example, such a query would be

$$\forall x_1 \forall x_2 \left(\texttt{Profession}(x_1, x_2) \rightarrow \exists w\, \texttt{Person}(x, y, w) \wedge \texttt{Profession}(x, x_2)\right)$$

asking asking for ids (x) and names (y) of people who have every profession listed in the database. These queries correspond to the *division* operation of relational algebra; see Exercise 5.4 for the explicit connection. Notice that the guard α is not allowed to repeat variables: for example, a guard $R(x, x)$ is not allowed, while $R(x, y)$ is.

Theorem 40.4

1. If q is a UCQ, then $\mathsf{naïve}(q, \mathcal{I}) = \mathsf{cert}_{\text{OWA}}(q, \mathcal{I}) = \mathsf{cert}_{\text{CWA}}(q, \mathcal{I})$.
2. If q is a PAG query, then $\mathsf{naïve}(q, \mathcal{I}) = \mathsf{cert}_{\text{CWA}}(q, \mathcal{I})$.

The key idea behind the proof is that the capturing of certain answers by naïve evaluation is very closely connected with preservation under homomorphisms, seen in Chapter 13. The intuition behind this connection is as follows: $[\![\mathcal{I}]\!]_{\text{OWA}}$ consists of all database instances D such that we have a homomorphism $\mathcal{I} \rightarrow D$. Thus, if we have a Boolean query q and it is naïve ly true in $\mathcal{I}$, and furthermore preserved under all homomorphisms, then q is true in all $D \in [\![\mathcal{I}]\!]_{\text{OWA}}$. This reasoning extends to queries with free variables and to the CWA semantics. For the latter, we need the notion of *strong onto homomorphisms*. These are homomorphisms $h : S \rightarrow S'$ between sets of atoms such that $h(S) = S'$.

We now recall the definition of preservation under homomorphisms. A query q is preserved under homomorphisms if $(D, \bar{a}) \rightarrow_{\text{Dom}(q)} (D', \bar{b})$ and

$\bar{a} \in q(D)$ imply $\bar{b} \in q(D')$, where $(D, \bar{a}) \to_C (D', \bar{b})$ means that there exists a homomorphism from $(\mathsf{V}_C(D), \mathsf{V}_C(\bar{a}))$ to $(D', \bar{b})$. Here V_C turns every constant in D other than those in C into a new null, see Chapter 13. If in the above definition we replace homomorphisms by strong onto homomorphisms, then we have the definition of preservation under strong onto homomorphisms.

Proposition 40.5

Let q be an FO query.

1. If q is preserved under homomorphisms, then $\mathsf{naïve}(q, \mathcal{I}) = \mathsf{cert}_{\text{OWA}}(q, \mathcal{I})$.
2. If q is preserved under strong onto homomorphisms, then $\mathsf{naïve}(q, \mathcal{I}) = \mathsf{cert}_{\text{CWA}}(q, \mathcal{I})$.

Proof. We prove the first item for the OWA semantics; the proof for CWA is very similar. Assume $q = (\varphi, \bar{x})$ is preserved under homomorphisms. We already know that $\mathsf{cert}_{\text{OWA}}(q, \mathcal{I}) \subseteq \mathsf{naïve}(q, \mathcal{I})$, see Proposition 40.2. For the converse, assume $\bar{a} \in \mathsf{naïve}(q, \mathcal{I})$. Then $(\mathcal{I}, \eta) \models \varphi$, where η maps $\bar{x}$ to $\bar{a}$. Suppose $h : \mathcal{I} \to D$ is an arbitrary valuation. Consider now $\mathsf{V}_{\text{Dom}(\varphi)}$; since it is a bijection, by Dom(φ)-genericity of φ we have $(\mathsf{V}_{\text{Dom}(\varphi)}(\mathcal{I}), \mathsf{V}_{\text{Dom}(\varphi)} \circ \eta) \models \varphi$. Next notice that $h \circ \mathsf{V}^{-1}_{\text{Dom}(\varphi)}$ is a homomorphism, as a composition of two homomorphisms. Thus, by homomorphism preservation, we obtain

$$\left(h \circ \mathsf{V}^{-1}_{\text{Dom}(\varphi)} \circ \mathsf{V}_{\text{Dom}(\varphi)}(\mathcal{I}),\ h \circ \mathsf{V}^{-1}_{\text{Dom}(\varphi)} \circ \mathsf{V}_{\text{Dom}(\varphi)} \circ \eta\right) \models \varphi\,,$$

or in other words $(D, h \circ \eta) \models \varphi$, and thus $h(\bar{a}) \in q(D)$, proving that $\bar{a} \in \mathsf{cert}_{\text{OWA}}(q, \mathcal{I})$. This proves Proposition 40.5. □

To complete the proof of Theorem 40.4, we simply use the fact that every UCQ is preserved under homomorphisms, see Proposition 28.10. The last remaining bit is to show that formulae in PAG are preserved under strong onto homomorphism. This is done by a routine induction on the structure of PAG formulae, see Exercise 5.5.

The connection between homomorphism preservation and coincidence of naïve evaluation and certain answers extends beyond FO queries (Exercise 5.9), but for FO queries the classes of formulae in Theorem 40.4 are optimal (Exercise 5.10).

Absolute Approximation of Certain Answers

Let us now summarize what we know about tractably evaluating queries on incomplete instances:

- we could use 3-valued evaluation, q_{3vl}, as SQL does (see Chapter 39) but then $q_{\mathsf{3vl}}(\mathcal{I})$ can produce both false positives and false negatives (see Proposition 39.9); or
- we could use naïve evaluation which always produces a superset of certain answers (Proposition 40.2) and thus no false negatives, but it can produce false positives.

As false positives may be viewed as the worse of the two, since they give completely false results, as opposed to omitting some true results, we now turn to *approximations* of FO queries and demonstrate that FO queries could be easily modified so that false positives could be avoided.

To achieve this, we transform an FO query $q = (\varphi, \bar{x})$ into another FO query $q^{\mathbf{t}} = (\varphi^{\mathbf{t}}, \bar{x})$ such that $\mathsf{naïve}(q^{\mathbf{t}}, \mathcal{I}) \subseteq \mathsf{cert}_{\text{CWA}}(q, \mathcal{I})$. Thus, evaluating $q^{\mathbf{t}}$ with its usual DLogSpace complexity is guaranteed to avoid false positives. Of course there is a simple and totally uninteresting way to define such $q^{\mathbf{t}}$ simply by setting $\varphi^{\mathbf{t}} = \mathtt{false}$. We obviously want a better approximation; at the very least we need to ensure that $q^{\mathbf{t}}(\mathcal{I}) = q(\mathcal{I})$ if $\mathcal{I}$ has no nulls, i.e., is a database rather than an incomplete instance.

To produce such a transformation we need two additional ingredients. One is an extra atomic formula $\mathsf{null}(x)$ testing if x is a null. Its semantics is

- $(\mathcal{I}, \eta) \models \mathsf{null}(x)$ iff $\eta(x) \in \mathrm{Dom}_{\mathsf{Null}}(\mathcal{I})$.

The second ingredient we need is a notion of *unification* of tuples. Given two tuples $\bar{u}$ and $\bar{w}$ of the same length over $\mathsf{Const} \cup \mathsf{Null}$, we say they are *unifiable*, and write $\bar{u} \Uparrow \bar{w}$, if there is a valuation h so that $h(\bar{u}) = h(\bar{w})$. For example, tuples $(1, \bot_1, 2)$ and $(\bot_1, 1, \bot_2)$ are unifiable via $h(\bot_1) = 1$ and $h(\bot_2) = 2$, which maps both into the same tuple $(1, 1, 2)$. On the other hand, $(1, \bot_1, 2)$ and $(\bot_1, 2, \bot_2)$ are not unifiable.

We make two observations about unifiability. First, the condition $\bar{x} \Uparrow \bar{y}$ can be expressed in FO with $\mathsf{null}(\cdot)$. For example, if $\bar{x} = (x)$ and $\bar{y} = (y)$, it is expressed by $\mathsf{null}(x) \vee \mathsf{null}(y) \vee (x = y)$. For tuples of higher arity, it is a more complicated case analysis, checking which elements among $(x_1, \ldots, x_n)$ and $(y_1, \ldots, y_n)$ are the same, which are different, and which are nulls. In other words, the explicit listing of all equalities and inequalities between x_is and y_js, as well as explicit listing of all statements $\mathsf{null}(x_i)$ and $\mathsf{null}(y_j)$ or their negations, which determine whether $\bar{x}$ and $\bar{y}$ unify (see Exercise 5.11 for more details).

An FO formula for checking whether $\bar{x} \Uparrow \bar{y}$ is of exponential size in $|\bar{x}|$. At the same time, checking whether $\bar{u} \Uparrow \bar{w}$ can be done in linear time (see Exercise 5.12). Thus we can also assume that $\bar{x} \Uparrow \bar{y}$ are new atomic formulae added to FO, and they have linear time evaluation complexity.

With these formulae in place, we can now produce a translation that eliminates false positives.

Definition 40.6: Translation $\varphi \mapsto (\varphi^{\mathbf{t}}, \varphi^{\mathbf{f}})$ of FO queries

- $R(\bar{x})^{\mathbf{t}} = R(\bar{x})$
- $(x = y)^{\mathbf{t}} = (x = y)$
- $(\varphi \wedge \psi)^{\mathbf{t}} = \varphi^{\mathbf{t}} \wedge \psi^{\mathbf{t}}$
- $(\neg\varphi)^{\mathbf{t}} = \varphi^{\mathbf{f}}$
- $(\exists x\, \varphi)^{\mathbf{t}} = \exists x\, \varphi^{\mathbf{t}}$

- $(R(\bar{x}))^{\mathbf{f}} = \neg\exists\bar{y}\, \big(R(\bar{y}) \wedge \bar{x} \Uparrow \bar{y}\big)$
- $(x = y)^{\mathbf{f}} = \neg(x = y) \wedge \neg\mathsf{null}(x) \wedge \neg\mathsf{null}(y)$
- $(\varphi \wedge \psi)^{\mathbf{f}} = \varphi^{\mathbf{f}} \vee \psi^{\mathbf{f}}$
- $(\neg\varphi)^{\mathbf{f}} = \varphi^{\mathbf{t}}$
- $(\exists x\, \varphi)^{\mathbf{f}} = \forall x\, \varphi^{\mathbf{f}}$

We now show that these translations give us a way to avoid one of false positives or false negatives. Specifically, $\varphi^{\mathbf{t}}$ avoids false positive answers while $\neg\varphi^{\mathbf{f}}$ avoids false negatives.

Theorem 40.7

For every incomplete instance $\mathcal{I}$, a FO query $q = (\varphi, \bar{x})$, its negation $\neg q = (\neg\varphi, \bar{x})$, and every valuation h on $\mathcal{I}$,

$$h(\mathsf{naïve}(q^{\mathbf{t}}, \mathcal{I})) \subseteq q(h(\mathcal{I})) \text{ and } h(\mathsf{naïve}(q^{\mathbf{f}}, \mathcal{I})) \subseteq \neg q(h(\mathcal{I}))\,.$$

Proof. Given q and $\mathcal{I}$ as in the statement of the theorem, we will show the following two implications, for every valuation h:

1. $(\mathcal{I}, \eta) \models \varphi^{\mathbf{t}} \Rightarrow (h(\mathcal{I}), h \circ \eta) \models \varphi$
2. $(\mathcal{I}, \eta) \models \varphi^{\mathbf{f}} \Rightarrow (h(\mathcal{I}), h \circ \eta) \models \neg\varphi$

These clearly imply the statement: if $\bar{a} \in \mathsf{naïve}(q^{\mathbf{t}}, \mathcal{I})$, then $(\mathcal{I}, \eta) \models \varphi^{\mathbf{t}}$ for η such that $\eta(\bar{x}) = \bar{a}$, and thus $(h(\mathcal{I}), h \circ \eta) \models \varphi$, implying that $h(\bar{a}) \in q(h(\mathcal{I}))$, and likewise for $q^{\mathbf{f}}$. We now prove these two statements by induction on the structure of φ.

Proof for $\varphi^{\mathbf{t}}$.

- Assume φ is $R(\bar{x})$; then $\varphi^{\mathbf{t}} = R(\bar{x})$. If $(\mathcal{I}, \eta) \models \varphi^{\mathbf{t}}$ then $R(\eta(\bar{x}))$ is an atom in $\mathcal{I}$ and hence $R(h(\eta(\bar{x})))$ is a fact in $h(\mathcal{I})$ and hence $(h(\mathcal{I}), h \circ \eta) \models \varphi$.
- Assume φ is $x = y$, and $\varphi^{\mathbf{t}} = \varphi$. If $(\mathcal{I}, \eta) \models \varphi^{\mathbf{t}}$ then $\eta(x) = \eta(y)$ and hence $h(\eta(x)) = h(\eta(y))$ implying $(h(\mathcal{I}), h \circ \eta) \models \varphi$.

- Assume φ is $\psi \wedge \chi$, and $\varphi^{\mathbf{t}} = \psi^{\mathbf{t}} \wedge \chi^{\mathbf{t}}$. If $(\mathcal{I}, \eta) \models \varphi^{\mathbf{t}}$ then $(\mathcal{I}, \eta) \models \psi^{\mathbf{t}}$ and $(\mathcal{I}, \eta) \models \chi^{\mathbf{t}}$ and by the induction hypothesis $(h(\mathcal{I}), h \circ \eta) \models \psi$ and $(h(\mathcal{I}), h \circ \eta) \models \chi$ and therefore $(h(\mathcal{I}), h \circ \eta) \models \varphi$.
- Assume φ is $\neg\psi$; then $\varphi^{\mathbf{t}} = \psi^{\mathbf{f}}$. If $(\mathcal{I}, \eta) \models \varphi^{\mathbf{t}}$ then $(\mathcal{I}, \eta) \models \psi^{\mathbf{f}}$ and by the induction hypothesis $(h(\mathcal{I}), h \circ \eta) \models \neg\psi$ and therefore $(h(\mathcal{I}), h \circ \eta) \models \varphi$.
- Assume φ is $\exists x \psi$; then $\varphi^{\mathbf{t}} = \exists x\, \psi^{\mathbf{t}}$. If $(\mathcal{I}, \eta) \models \varphi^{\mathbf{t}}$ then $(\mathcal{I}, \eta[a/x]) \models \psi^{\mathbf{t}}$ for some $a \in \mathrm{Dom}(\mathcal{I})$ and by the induction hypothesis $(h(\mathcal{I}), h \circ \eta[a/x]) \models \psi$ and therefore $(h(\mathcal{I}), h \circ \eta) \models \varphi$ as $h(a)$ is a witness for x.

Proof for $\varphi^{\mathbf{f}}$.

- Assume φ is $R(\bar{x})$. Towards contradiction, suppose that $(\mathcal{I}, \eta) \models \varphi^{\mathbf{f}}$ but $(h(\mathcal{I}), h \circ \eta) \models \varphi$. The latter means that $R\big(h(\eta(\bar{x}))\big)$ is a fact of $h(\mathcal{I})$. This happens if there an atom $R(\bar{u})$ in $\mathcal{I}$ such that $h(\bar{u}) = h(\eta(\bar{x}))$, i.e., $\bar{u} \Uparrow \eta(\bar{x})$. Thus, $(\mathcal{I}, \eta) \models \exists \bar{y}\, (R(\bar{y}) \wedge \bar{y} \Uparrow \bar{x})$, i.e., $(\mathcal{I}, \eta) \models \varphi^{\mathbf{f}}$, which is a contradiction.
- Assume φ is $x = y$, and $\varphi^{\mathbf{f}} = (x \neq y) \wedge \neg\mathsf{null}(x) \wedge \neg\mathsf{null}(y)$. If $(\mathcal{I}, \eta) \models \varphi^{\mathbf{f}}$ then $\eta(x)$ and $\eta(y)$ are different constants, and thus so are $h(\eta(x))$ and $h(\eta(y))$. Hence $(h(\mathcal{I}), h \circ \eta) \models \neg(x = y)$, i.e., $(h(\mathcal{I}), h \circ \eta) \models \neg\varphi$.
- Assume φ is $\psi \wedge \chi$, and $\varphi^{\mathbf{f}} = \psi^{\mathbf{f}} \vee \chi^{\mathbf{f}}$. If $(\mathcal{I}, \eta) \models \varphi^{\mathbf{f}}$ then by the induction hypothesis $(h(\mathcal{I}), h \circ \eta) \models \neg\psi \vee \neg\chi$ which is of course $\neg(\psi \wedge \chi)$.
- Assume φ is $\neg\psi$; then $\varphi^{\mathbf{f}} = \psi^{\mathbf{t}}$. If $(\mathcal{I}, \eta) \models \varphi^{\mathbf{f}}$ then by the induction hypothesis $(h(\mathcal{I}), h \circ \eta) \models \psi$ and therefore $(h(\mathcal{I}), h \circ \eta) \models \neg\varphi$.
- Assume φ is $\exists x\, \psi$; then $\varphi^{\mathbf{f}} = \forall x\, \psi^{\mathbf{f}}$. If $(\mathcal{I}, \eta) \models \varphi^{\mathbf{f}}$, then for every $a \in \mathrm{Dom}(\mathcal{I})$ we have $(\mathcal{I}, \eta[a/x]) \models \psi^{\mathbf{f}}$ and thus $(h(\mathcal{I}), h \circ \eta[a/x]) \models \neg\psi$ and hence $(h(\mathcal{I}), h \circ \eta) \models \neg\exists x\, \psi$ which is $\neg\varphi$. This concludes the proof. □

Corollary 40.8: Avoiding False Positives and Negatives

For every FO query q, the query $q^{\mathbf{t}}$ returns no false positives, and the query $\neg q^{\mathbf{f}}$ returns no false negatives. Moreover, on databases without nulls, $q(D) = q^{\mathbf{t}}(D) = \neg q^{\mathbf{f}}(D)$.

Proof. The first statement follows immediately from the theorem and the definition of certain answers. The second is by inspection of the translation, taking into account that without nulls, $\bar{x} \Uparrow \bar{y}$ can be replaced by $\bar{x} = \bar{y}$ and $\neg\mathsf{null}(x)$ by `true`. □

As an illustration, we return to the query $\varphi(x) = R(x) \wedge \neg S(x)$ seen earlier, for which naïve evaluation produces a false positive answer. Applying the transformation, we have

$$\begin{aligned}\varphi^{\mathbf{t}}(x) &= (R(x) \wedge \neg S(x))^{\mathbf{t}} = (R(x))^{\mathbf{t}} \wedge (\neg S(x))^{\mathbf{t}} \\ &= R(x) \wedge (S(x))^{\mathbf{f}} = R(x) \wedge \neg\exists y \left(S(y) \wedge x \Uparrow y\right) \\ &= R(x) \wedge \neg\exists y \left(S(y) \wedge (\mathsf{null}(x) \vee \mathsf{null}(y) \vee (x = y))\right)\end{aligned}$$

It is then easy to check that $\varphi^{\mathbf{t}}$ returns the empty set when evaluated naïvely on an incomplete instance with atoms $R(1)$ and $S(\bot)$.

We remark that while simple to present, the translation of Definition 40.6 is not going to be very efficient when implemented in relational algebra for queries involving negation. A couple of examples illustrating this are presented in Exercise 5.13, and an alternative more efficient translation is given in Exercise 5.14.

We conclude by showing that the above translation also helps us eliminate false positives or false negatives under the 3-valued evaluation used in SQL. Recall that at the end of Chapter 39, we defined $q_{\mathsf{3vl}}(\mathcal{I})$ as $\{\eta(\bar{x}) \mid \mathsf{Eval}(\varphi, \mathcal{I}, \eta) = \texttt{true}\}$, for a query $q = (\varphi, \bar{x})$. We now define $q^{\mathbf{t}}_{\mathsf{3vl}}$ and $q^{\mathbf{f}}_{\mathsf{3vl}}$ in the same way, but with φ replaced by $\varphi^{\mathbf{t}}$ and $\varphi^{\mathbf{f}}$. Of course we also need to extend Eval to new formulae $\mathsf{null}(x)$ and $\bar{x} \Uparrow \bar{y}$. These formulae can only take values $\texttt{true}$ and $\texttt{false}$. We have $\mathsf{Eval}(\mathsf{null}(x), \mathcal{I}, \eta) = \texttt{true}$ iff $\eta(x)$ is a null, and $\mathsf{Eval}((\bar{x} \Uparrow \bar{y}), \mathcal{I}, \eta) = \texttt{true}$ iff $\eta(\bar{x})$ and $\eta(\bar{y})$ unify.

Then we have an analog of Corollary 40.8.

Corollary 40.9: Avoiding False Positives and Negatives

For every FO query q, the query $q^{\mathbf{t}}_{\mathsf{3vl}}$ returns no false positives, and the complement of the query $q^{\mathbf{f}}_{\mathsf{3vl}}$ returns no false negatives. Moreover, on databases without nulls, both of these queries coincide with q.

The proof is by simple adaptation of the inductive proof of Theorem 40.7.

Probabilistic Approximation of Certain Answers

We now address the following question: how close to certainty is the query answer produced by the naïve evaluation? It turns out that the answer is "very close". Recall that $\bar{a} \in \mathsf{cert}_{\text{CWA}}(q, \mathcal{I})$ if $h(\bar{a}) \in q(h(\mathcal{I}))$ for every valuation h. The framework in which we pose our question is the following:

- Pick a valuation h uniformly at random. What is the probability that $h(\bar{a}) \in q(h(\mathcal{I}))$ holds?

We then show – again, describing the intuition behind the result – that for *all* FO queries q,

- this probability is either 0 or 1;

- it is 1 if and only if $\bar{a} \in \mathsf{naïve}(q, \mathcal{I})$.

Thus, naïve evaluation, while only providing absolute guarantees of certainty of query answers for a restricted class of queries, gives high probability guarantees for many more queries.

To prove such a result, there is still a technical problem: how do we choose a valuation $h : \mathrm{Dom}_{\mathsf{Null}}(\mathcal{I}) \to \mathsf{Const}$ uniformly at random? Since the set Const is infinite, there are infinitely many such valuations, and thus no uniform distribution on them. Instead, we can use a trick inspired by the 0–1 law seen in Chapter 32. Namely, we consider finite subsets of Const, restricted to which there are finitely many valuations. We then look at the asymptotic behavior of the probability that a tuple is in the answer.

To do this, fix some enumeration of elements of Const as $\{c_1, c_2, c_3, \dots\}$. The exact enumeration is not important, as we will see shortly. Let Const_n be the set $\{c_1, \dots, c_n\}$. Let $\mathcal{H}_n(\mathcal{I})$ be the finite set of all valuations $h : \mathrm{Dom}_{\mathsf{Null}}(\mathcal{I}) \to \mathsf{Const}_n$. For a k-ary query q, an incomplete instance $\mathcal{I}$, and a k-ary tuple over $\mathrm{Dom}(\mathcal{I})$, we now define

$$\mu_n(q, \mathcal{I}, \bar{a}) \;=\; \frac{|\{h \in \mathcal{H}_n \mid h(\bar{a}) \in q(h(\mathcal{I}))\}|}{|\mathcal{H}_n(\mathcal{I})|}$$

as the probability that a valuation h, picked uniformly at random from the finite set $\mathcal{H}_n(\mathcal{I})$, witnesses that $\bar{a}$ is in the answer to q on $\mathcal{I}$, i.e., the condition $h(\bar{a}) \in q(h(\mathcal{I}))$. Then we define its asymptotic behavior:

$$\mu(q, \mathcal{I}, \bar{a}) \;=\; \lim_{n \to \infty} \mu_n(q, \mathcal{I}, \bar{a})\,.$$

Notice that this value does not depend on the exact enumeration of Const: as long as n is big enough so that it contains $\mathrm{Dom}_{\mathsf{Const}}(\mathcal{I})$ and $\mathrm{Dom}(q)$, the value of $\mu_n(q, \mathcal{I}, \bar{a})$ does not depend on how the rest of Const is enumerated.

Intuitively, $\mu(q, \mathcal{I}, \bar{a})$ gives us the probability that a valuation – now without a restriction on its range - picked randomly witnesses that $\bar{a}$ is an answer. The closer this is to 1, the better $\bar{a}$ is as the answer to q. We now show that naïve evaluation gives us precisely the tuples that are good (although not absolutely certain) answers.

Theorem 40.10

For every FO k-ary query q, every incomplete instance $\mathcal{I}$, and every k-tuple $\bar{a}$,

$$\mu(q, \mathcal{I}, \bar{a}) \;=\; \begin{cases} 1 & \text{if } \bar{a} \in \mathsf{naïve}(q, \mathcal{I}) \\ 0 & \text{if } \bar{a} \notin \mathsf{naïve}(q, \mathcal{I})\,. \end{cases}$$

Proof. Consider again simple valuations $h : \mathrm{Dom}_{\mathsf{Null}}(\mathcal{I}) \to \mathsf{Const}_n$ that are bijections whose range contains no elements of $\mathrm{Dom}_{\mathsf{Const}}(\mathcal{I})$ or $\mathrm{Dom}(q)$, and let $\mathcal{H}_n^s(\mathcal{I})$ stand for the set of such valuations. We next show that $\lim_{n\to\infty} |\mathcal{H}_n^s(\mathcal{I})|/|\mathcal{H}_n(\mathcal{I})| = 1$.

Assume $|\mathrm{Dom}_{\mathsf{Null}}(\mathcal{I})| = m$ and $|\mathrm{Dom}_{\mathsf{Const}}(\mathcal{I}) \cup \mathrm{Dom}(q)| = k$. A valuation h fails to be simple if one of the following is true.

- The range of h contains an element of $\mathrm{Dom}_{\mathsf{Const}}(\mathcal{I}) \cup \mathrm{Dom}(q)$, i.e., $h(\perp)$ is in that set for some $\perp \in \mathrm{Dom}_{\mathsf{Null}}(\mathcal{I})$. There are at most $mk \cdot n^{m-1}$ such valuations. Indeed, we pick one of m nulls $\perp$ and one of k elements of $\mathrm{Dom}_{\mathsf{Const}}(\mathcal{I}) \cup \mathrm{Dom}(q)$ it is mapped into, and on the remaining $m-1$ nulls this valuation can be arbitrary.
- The valuation h is not a bijection: there are two nulls $\perp$ and $\perp'$ such that $h(\perp) = h(\perp')$. The number of such valuations is at most $\binom{m}{2} \cdot n \cdot n^{m-2}$. Indeed, there are $\binom{m}{2}$ ways to pick $\perp$ and $\perp'$, then n possible assignments to an element of Const_n, and on the remaining $m-2$ elements of $\mathrm{Dom}_{\mathsf{Null}}(\mathcal{I})$, the valuation can be chosen arbitrarily.

Summing up, the number of valuations not in $\mathcal{H}_n^s(\mathcal{I})$ is at most $(mk + \binom{m}{2}) \cdot n^{m-1}$. Since $|\mathcal{H}_n(\mathcal{I})| = n^m$, we have

$$\lim_{n\to\infty} \frac{|\mathcal{H}_n^s(\mathcal{I})|}{|\mathcal{H}_n(\mathcal{I})|} \geq \lim_{n\to\infty} \frac{n^m - (mk + \binom{m}{2}) \cdot n^{m-1}}{n^m} = 1 - \lim_{n\to\infty} \frac{mk + \binom{m}{2}}{n} = 1\,.$$

Next assume $\bar{a} \in \mathsf{naïve}(q, \mathcal{I})$. By $\mathrm{Dom}(q)$-genericity of FO queries (Exercise 1.2), we have $h(\bar{a}) \in q(h(\mathcal{I}))$ for each $h \in \mathcal{H}_n^s(\mathcal{I})$. Hence, $\mu(q, \mathcal{I}, \bar{a}) \geq \lim_{n\to\infty} |\mathcal{H}_n^s(\mathcal{I})|/|\mathcal{H}_n(\mathcal{I})| = 1$ which implies $\mu(q, \mathcal{I}, \bar{a}) = 1$. If $\bar{a} \notin \mathsf{naïve}(q, \mathcal{I})$, we apply the previous argument to the complement of q, obtaining $\mu(\neg q, \mathcal{I}, \bar{a}) = 1$ from which $\mu(q, \mathcal{I}, \bar{a}) = 0$ follows. $\square$

42
Inconsistent Databases

Let D be a database and Σ a set of constraints over the same schema $\mathbf{S}$. We call D *inconsistent* with respect to Σ if $D \not\models \Sigma$, that is, there is at least one constraint in Σ such that D does not satisfy. Here we focus on the case when Σ is a set of primary keys. Recall that this means that each $R \in \mathbf{S}$ comes equipped with its own key, i.e., $\mathit{key}(R) = A$, where $A = \emptyset$ or $A = [1, \ldots, p]$ for some $p \in [1, \mathit{ar}(R)]$. Henceforth, D is inconsistent with respect to Σ if there is a symbol $R \in \mathbf{S}$ and facts $R(\bar{a}), R(\bar{b}) \in D$ such that

$$\mathit{key}(R) = A \quad \text{and} \quad \pi_A(\bar{a}) = \pi_A(\bar{b}).$$

In this case we call the pair $(R(\bar{a}), R(\bar{b}))$ to be *key-violating*.

When D is inconsistent with respect to a set Σ of primary keys, it is always possible to restore consistency by deleting some of the tuples from D. The idea, of course, is to do so without deleting more information than is needed. As an example, consider the relational table **SOURCES** shown in Table 42.1, and assume that **News Id** is the key for the table. Then clearly **SOURCES** is inconsistent with respect to this key. However, the consistency can be restored by deleting one tuple with key N1 and one tuple with key N4. We could also delete all tuples with key N1 and all tuples with key N4, which would yield a database that satisfies the key, but this would violate our second requirement: we would be deleting more information than is needed to restore consistency.

We can generalize this process as follows. Let us define a *block* in a database D with respect to a set Σ of primary keys to be any maximal set B of facts from D such that the facts in B are pairwise key-violating. For instance, in the previous example the table **SOURCES** contains precisely the five blocks listed below:

- $B_1 = \{(\text{N1}, \text{Internet Research Agency}, \text{300K})\}$.
- $B_2 = \{(\text{N2}, \text{Gotham Globe}, \text{27K}),\ (\text{N2}, \text{The Chippewa Buggle}, \text{950K})\}$.
- $B_3 = \{(\text{N3}, \text{New York Daily Enquirer}, \text{2M})\}$.

SOURCES		
News Id	**Media**	**Shares**
N1	Internet Research Agency	300K
N2	Gotham Globe	27K
N2	The Chippewa Bugle	950K
N3	New York Daily Inquirer	2M
N4	La Cuarta	125K
N4	Fortín Mapocho	500K
N5	Twin Peaks Gazette	350K

Table 42.1: A table **SOURCES**, which is inconsistent with respect to the key constraint that establishes that no two tuples can have the same value for **News Id**.

- $B_4 = \{(\text{N4}, \text{La Cuarta}, \text{125K}),\ (\text{N4}, \text{Fortín Mapocho}, \text{500K})\}$.
- $B_5 = \{(\text{N5},\ \text{Twin Peaks Gazette}, \text{350K})\}$.

It is clear that to restore consistency from D with respect to Σ we need to delete facts from D until leaving at most one tuple in each block B of D. In addition, in order to do it in a minimal way we have to leave at least one fact in each such a block. This gives rise to the notion of a *repair* of D with respect to Σ. This is a subset D' of D that is obtained by choosing exactly one tuple from each block of D with respect to Σ. The notion of repair thus satisfies the following important property, which establish that they are precisely the maximally consistent subsets of D with respect to Σ.

Proposition 42.1. *Let D be a database and Σ a set of primary keys. The repairs of D with respect to Σ are precisely the D' with $D' \subseteq D$ such that:*

- *$D' \models \Sigma$, and*
- *there is no D'' with $D' \subsetneq D'' \subseteq D$ such that $D'' \models \Sigma$.*

Notice that repairs always exist and, if $D \models \Sigma$, then the only repair of D with respect to Σ is D itself. Moreover, the number of repairs can be exponential in the number of tuples of D (e.g., if D has n blocks, and each block has two facts, then D has 2^n repairs).

When querying a database D that is inconsistent with respect to a set Σ of primary keys, we are interested in obtaining those results that are true regardless of how consistency is restored. These are the *consistent answers*, which are those answers that are true in every possible repair of D with respect to Σ. Formally, given a query $q(\bar{x})$, a database D, and a tuple $\bar{a}$ of elements in $\text{Dom}(D)$ of the same arity as $\bar{x}$, we call $\bar{a}$ a consistent answer to q over (D, Σ) if $\bar{a} \in q(D')$, for every repair D' of D with respect to Σ. We then define

$$\text{CQA}(q, D, \Sigma) \ = \ \{\bar{a} \mid \bar{a} \text{ is a consistent answer to } q \text{ over } (D, \Sigma)\}.$$

When q is boolean, we abuse notation write $\mathrm{CQA}(q, D, \Sigma) = \mathsf{true}$ to denote that the empty tuple () belongs to $\mathrm{CQA}(q, D, \Sigma)$; i.e., $q(D') = \mathsf{true}$ for every repair D' of D with respect to Σ.

For instance, in our example we have that the set of consistent answers to the CQ $q(x) = \exists y \exists z\, \mathbf{Sources}(x, y, z)$ is the set $\{\mathrm{N1}, \mathrm{N2}, \mathrm{N3}, \mathrm{N4}, \mathrm{N5}\}$, and to the CQ $q(x, y) = \exists z\, \mathbf{Sources}(x, y, z)$ is the set

$$\{(\mathrm{N1}, \text{Internet Research Agency}), (\mathrm{N3}, \text{New York Daily Enquirer}),$$
$$(\mathrm{N5},\ \text{Twin Peaks Gazette})\}.$$

Complexity of consistent query answering

Given a query language $\mathcal{L}$, we are interested in the decision problem $\mathcal{L}$-ConsEvaluation which is defined as follows.

PROBLEM:	$\mathcal{L}$-ConsEvaluation
INPUT:	a query q from $\mathcal{L}$, a set Σ of primary keys, a database D, a tuple $\bar{a}$ over $\mathrm{Dom}(D)$
QUESTION:	is $\bar{a} \in \mathrm{CQA}(q, D, \Sigma)$?

As for the case of probabilistic query evaluation, we can show that the problem $\mathcal{L}$-ConsEvaluation is computationally hard even for the class of CQs and in data complexity (where we assume both the actual query and the set of keys to be fixed). For simplicity, if q is a fixed CQ and Σ a fixed set of constraints we denote by ConsEvaluation(q, Σ) the problem of checking if $\bar{a} \in \mathrm{CQA}(q, D, \Sigma)$ on an input given by a database D and a tuple $\bar{a}$ over $\mathrm{Dom}(D)$. We can then show the following result.

Theorem 42.2. *The problem CQ-*ConsEvaluation *is* CONP*-hard even in data complexity. More precisely, there are a fixed Boolean CQ q and a fixed set Σ of primary keys such that* ConsEvaluation(q, Σ) *is* CONP*-hard.*

Proof. We define the boolean CQ q to be $\exists x \exists y (\mathsf{Col}(x, x') \wedge \mathsf{edge}(x, y) \wedge \mathsf{Col}(y, y) \wedge \mathsf{Bad}(x', y'))$ and Σ to consist exclusively of the constraint that establishes the first attribute of col to be a key; that is, $\Sigma = \{key(\mathsf{col}) = 1\}$. We show that there is a polynomial time reduction from 3-Col, the problem of checking if a graph is 3-colorable, to the complement of the problem of checking whether $\mathrm{CQA}(q, D, \Sigma) = \mathsf{true}$, given a database D.

Let $G = (V, E)$ be a graph. We construct a database D over the schema of q as follows. The domain of D is defined as the disjoint union of V and three fresh constants 0, 1, and 2. The relation edge^D contains all edges $(v, w) \in E$. Second, the relation col^D contains all pairs $(v, 0)$, $(v, 1)$, and $(v, 2)$, for $v \in V$. Finally, the relation Bad^D contains the pairs $(0, 0)$, $(1, 1)$, and $(2, 2)$. Since the first component of col is the key of this relation, each repair of D defines an assignment of a color in $\{0, 1, 2\}$ to each node $v \in V$. By definition, any such an assignment corresponds to a proper 3-coloring of the graph G if there are no two nodes v, w that are adjacent in G which are assigned the same color. It is easy to see then that G is 3-colorable if and only if there is a repair D' of D that does not satisfy q, i.e., it is not the case that $\mathrm{CQA}(q, D, \Sigma) = \mathsf{true}$. $\square$

Hence, consistent query answering resembles query evaluation over probabilistic databases in terms of the complexity of evaluation for CQs: Both problems are intractable even in data complexity. As we did for CQ evaluation over probabilistic databases, we present next two lines of research that have been developed to tackle this issue. The first one corresponds to identifying classes of CQs for which the problem can be solved efficiently. The second one corresponds to the design of an FPRAS for computing the "degree of certainty" with which a CQ holds over an inconsistent database. This degree of certainty refers to the number of repairs that satisfy the given CQ.

A tractable class of CQs

We focus on the class of self-join free CQs (sjfCQs), which was studied in the previous chapter, since this case is much better understood in the literature and results for it are much easier to explain than for the general case. Recall that a sjfCQ q is a CQ in which no two distinct atoms use the same relation symbol. We identify a syntactically restricted class of pairs (q, Σ), where q is a sjfCQ and Σ is a set of primary keys, for which $\mathsf{ConsEvaluation}(q, \Sigma)$ can be solved in polynomial time. This class is defined in terms of the *key-join* graphs of pairs of the form (q, Σ), a notion that we introduce below. To simplify the presentation, we consider Boolean sjfCQs only.

Let $R(\bar{x})$ be an atom over a schema $\mathbf{S}$ with $\mathit{key}(R) = A$. The tuple of variables in *key positions* of $R(\bar{x})$ is defined as $\emptyset$, if $A = \emptyset$, and as $(x_1, \dots, x_p)$, if $A = [1, p]$ for $p \in [1, \mathit{ar}(R)]$. We often write $R(\bar{x})$ as $R(\bar{y}\,;\bar{z})$ to denote that $\bar{y}$ and $\bar{z}$ are the tuples of variables in key and non-key positions of $R(\bar{x})$, respectively. A Boolean sjfCQ $R_1(\bar{x}_1), \dots, R_m(\bar{x}_m)$ together with a set Σ of primary keys can thus be represented in a single expression:

$$R_1(\bar{y}_1\,;\,\bar{z}_1), \dots, R_m(\bar{y}_m\,;\,\bar{z}_m). \tag{42.1}$$

We call these expressions *constrained* sjfCQs and often denote them $\hat{q}$. We write $\mathsf{ConsEvaluation}(\hat{q})$ as a shorthand for the problem $\mathsf{ConsEvaluation}(q, \Sigma)$, where q and Σ are the CQ and set of primary keys, respectively, that are univocally associated with $\hat{q}$.

Let $\hat{q}$ be a Boolean constrained sjfCQ of the form (42.1). By definition of self-join freeness, the R_is are pairwise different. The key-join free graph of $\hat{q}$ is a directed graph whose nodes are the atoms $R_i(\bar{y}_i\,;\,\bar{z}_i)$ of $\hat{q}$ and there is a directed edge from $R_i(\bar{y}_i\,;\,\bar{z}_i)$ to $R_j(\bar{y}_j\,;\,\bar{z}_j)$, for $i \neq j$, if there is a variable in $(\bar{y}_j\,;\,\bar{z}_j)$ that appears in $\bar{z}_i$, i.e., in a non-key position of $R_i(\bar{y}_i\,;\,\bar{z}_i)$.

Example 42.3. Consider the constrained sjfCQ $\hat{q}_1$:– $R(\underline{x}\,;z), S(\underline{z}\,;x)$, where we have underlined key positions for clarity. The key-join graph of $\hat{q}_1$ is shown below:

$$\begin{tikzcd} R(\underline{x}\,;z) \arrow[r, bend right, "z"'] & S(\underline{z}\,;x) \arrow[l, bend right, "x"'] \end{tikzcd}$$

The labels in the edges represent the non-key positions in the source that appear in the target.

Consider also the CQ $\hat{q}_2$:– $R(\underline{x}\,;z), S(\underline{z,v}\,;w)$. The key-join graph of $\hat{q}_2$ is shown in the following figure:

$$\begin{tikzcd} R(\underline{x}\,;z) \arrow[r, "z"'] & S(\underline{z,v}\,;w) \end{tikzcd}$$

□

Following with the previous example, we have that both ConsEvaluation$(\hat{q}_1)$ and ConsEvaluation$(\hat{q}_2)$ are coNP-complete problems (this fact is left as an exercise to the reader). Notice that the key-join graph of $\hat{q}_1$ contains a directed cycle, and hence for the tractable family of CQs we aim to define in this section we can only admit constrained sjfCQs with an acyclic key-join graph. Yet, this is not sufficient. In fact, the key-join graph of $\hat{q}_2$ is acyclic. In this case, the reason why ConsEvaluation$(\hat{q}_2)$ is coNP-complete is found in the label of the edge that goes from $R(x\,;z)$ to $S(z,v\,;w)$: this does not include all variables that appear in key positions in the second atom, in particular, the variable v is not present in such a label. In technical terms, it is said that the join from the key of $S(z,v\,;w)$ to $R(x\,;z)$ is *non-full*. Henceforth, our tractable class neither can admit non-full joins in edges of key-join graphs.

As we establish below, these two restrictions suffice to guarantee tractability for the problem ConsEvaluation$(\hat{q})$. Formally, let us define $\mathcal{C}_{\text{ac-full}}$ as the set of Boolean constrained sjfCQs $\hat{q}$ for which the following statements hold:

- the key-join graph of $\hat{q}$ contains no directed cycles, and
- every edge of the key-join graph of $\hat{q}$ is *full*, that is, each such an edge of the form $R(\bar{x}\,;\bar{y}) \to S(\bar{z}\,;\bar{w})$ satisfies that every variable that appears in the tuple $\bar{z}$ of key positions in $S(\bar{z}\,;\bar{w})$ also appears in the tuple $\bar{y}$ of non-key positions in $R(\bar{x}\,;\bar{y})$.

Example 42.4. Consider the constrained sjfCQ

$$\hat{q} :\!-\, R(\underline{x,z}\,;u,x), S(\underline{u,x}\,;v,w), M(\underline{u}\,;t), L(\underline{t}\,;y).$$

The key-join graph of $\hat{q}$ is shown in the following figure:

$$\begin{tikzcd}
R(\underline{x, z}\,; u, x) \arrow[r, "\{u, x\}"] \arrow[dr, "\{u\}"'] & S(\underline{u, x}\,; v, w) & \\
& M(\underline{u}\,; t) \arrow[r, "\{t\}"] & L(\underline{t}\,; y)
\end{tikzcd}$$

It can be observed that the key-join graph of $\hat{q}$ is acyclic and all edges are full. Hence, $\hat{q}$ in $\mathcal{C}_{\text{ac-full}}$. □

The following is an important observation regarding the class $\mathcal{C}_{\text{ac-full}}$. We leave the proof of this fact to the reader.

Lemma 42.5. *For every $\hat{q} \in \mathcal{C}_{ac\text{-}full}$ we have that the key-join graph of $\hat{q}$ must be a directed forest, i.e., every node has at most one incoming edge.*

The main result of this section is the following, which establishes the tractability of computing certain answers for the constrained CQs in $\mathcal{C}_{\text{ac-full}}$. It should be noted that this is also a meaningful result from a practical point of view, as it has been observed that constrained sjfCQs in the class $\mathcal{C}_{\text{ac-full}}$ occur often in real world scenarios.

Theorem 42.6. *Let $\hat{q}$ be a fixed Boolean constrained* sjf*CQ in $\mathcal{C}_{ac\text{-}full}$. The problem* ConsEvaluation($\hat{q}$) *can be solved in* PTIME.

Proof. We know from Lemma 42.5 that the key-join graph of $\hat{q}$ is a directed forest. For simplicity, we assume that it actually consists of a single directed tree T; the general case is slightly more involved and left as an exercise. Each node t of T is thus associated with an atom $R_t(\bar{x}_t\,; \bar{y}_t)$ of $\hat{q}$. We denote by $\hat{q}_t$ the constrained sjfCQ that is induced in $\hat{q}$ by all the atoms of the form $R_u(\bar{x}_u\,; \bar{y}_u)$, for u a (not necessarily proper) descendant of t in T. We proceed by showing that each such a constrained sjfCQ $\hat{q}_t$ admits an FO-*rewriting*. More formally there is an FO sentence ψ_t such that for every database D we have that:

$$\text{CQA}(\hat{q}_t, D) = \mathsf{true} \qquad \Longleftrightarrow \qquad D \models \psi_t.$$

Here $\text{CQA}(\hat{q}_t, D)$ is a shortening for $\text{CQA}(q_t, D, \Sigma_t)$, where q_t and Σ_t are the sjfCQ and set of primary keys, respectively, that are univocally associated with $\hat{q}_t$. That is, the certain answer to q_t over D with respect to Σ_t can be obtained by directly evaluating the rewriting ψ_t over D. In particular, if r is the root of T then $\hat{q}_r = \hat{q}$, and hence $\hat{q}$ itself admits an FO-rewriting. Theorem 42.6 then follows since FO sentences can be evaluated in DLOGSPACE in data complexity from Theorem 7.3.

We start by proving the following claim. Here, $q'_t(\bar{x}_t)$ denotes the sjfCQ which is obtained from q_t by removing the existential quantification on $\bar{x}_t$.

Claim 42.7. *For every node t of T there is an FO formula $\psi'_t(\bar{x}_t)$ such that, for every database D and tuple $\bar{a}$ over Dom(D) with $|\bar{a}| = |\bar{x}_t|$, the following statements are equivalent:*

- $D \models \psi'_t(\bar{a})$.
- *For every repair D' of D with respect to Σ_t it is the case that $\bar{a} \in q'_t(D')$.*

We prove the claim by induction on the height of t. If t has height 0, then it is a leaf. We illustrate the proof with an example. Suppose that the atom $R_t(\bar{x}_t\,;\bar{y}_t)$ is $R(x, y\,; x, v)$. Then $\psi'_t(x, y)$ can be defined as:

$$\exists v\, R(x, y, x, v) \,\wedge\, \forall u \forall v\, \big(R(x, y, u, v) \,\rightarrow\, u = x\big).$$

In fact, it is easy to see that for every database D and elements $a, b \in \mathrm{Dom}(D)$ the following holds: for every repair D' of D with respect to Σ_t we have that $(a, b) \in q'_t(D')$ if and only if there is a fact of the form $R(a, b, a, d) \in D$, for some $d \in \mathrm{Dom}(D)$, and each fact in the same block than $R(a, b, a, d)$ with respect to Σ_t is of the form $R(a, b, a, d')$, for some $d' \in \mathrm{Dom}(D)$. Clearly, the latter holds if and only if $D \models \psi'_t(a, b)$. The proof of the general case, in which t is associated with an arbitrary atom of the form $R_t(\bar{x}_t\,;\bar{y}_t)$, is left to the reader.

Assume now that t has height $h+1$, for $h \geq 0$. Let $t_1, \ldots, t_k$ be the children of t in T. We then define $\psi'_t(\bar{x}_t)$ as:

$$\exists \bar{y}'_t\, R_t(\bar{x}_t, \bar{y}_t) \,\wedge\, \forall \bar{z}_t\, \Big(R_t(\bar{x}_t, \bar{z}_t) \,\rightarrow\, \chi(\bar{x}_t, \bar{z}_t) \wedge \bigwedge_{i \in [1,k]} \psi'_{t_i}(\bar{z}_{t_i})\Big),$$

where $\bar{y}'_t$ is the tuple of all variables in $\bar{y}_t$ that do not appear in $\bar{x}_t$; the tuple $\bar{z}_t$ consists exclusively of fresh variables and satisfies $|\bar{z}_t| = |\bar{y}_t|$; for each t_i with $i \in [1, k]$ the formulae ψ'_i are obtained by induction hypothesis; the tuple $\bar{z}_{t_i}$ is the one that is obtained from $\bar{z}_t$ by selecting the positions corresponding to key positions from $\bar{x}_{t_i}$ (this is well-defined as, by definition of the class $\mathcal{C}_{\text{ac-full}}$, each variable in the $\bar{x}_{t_i}$ appears in $\bar{y}_t$); and, finally, $\chi(\bar{x}_t, \bar{z}_t)$ contains each equality of the form $x = z$ such that $x \in \bar{x}_t$, $z \in \bar{z}_t$, and the variable corresponding to z in $\bar{y}_t$ is precisely x. As an example of the construction, suppose that t is associated with atom $R(x, y\,; x, v)$ and has two children t_1 and t_2 which are respectively associated with atoms $S(x\,; u)$ and $T(x, v\,; w)$. Then $\psi'_t(x, y)$ is defined as:

$$\exists v\, R(x, y, x, v) \,\wedge\, \forall u \forall v \left(R(x, y, u, v) \,\rightarrow\, \big(u = x \wedge \psi'_{t_1}(u) \wedge \psi'_{t_2}(u, v)\big)\right).$$

We start by proving the first part of Claim 42.7. Assume first that $D \models \psi'_t(\bar{a})$. Then, by definition, there is a fact of the form $R_t(\bar{a}, \bar{b}) \in D$, for some tuple $\bar{b}$ over $\mathrm{Dom}(D)$, and each fact in the same block than $R_t(\bar{a}, \bar{b})$ with respect to Σ_t is of the form $R_t(\bar{a}, \bar{b}')$, for some tuple $\bar{b}'$ over $\mathrm{Dom}(D)$ that satisfies $\chi(\bar{a}, \bar{b}')$ and, in addition, it holds that $D \models \psi'_{t_i}(\bar{b}'_i)$, where $\bar{b}'_i$ is the restriction of $\bar{b}'$ to the variables in $\bar{z}_{t_i}$. Take an arbitrary repair D' of D with respect to Σ_t. Then D' must contain some fact of the form $R_t(\bar{a}, \bar{b})$, for some tuple $\bar{b}$ over $\mathrm{Dom}(D)$. By induction hypothesis, for each $i \in [1, k]$ we have that $\bar{b}_i \in q'_{t_i}(D')$, where $\bar{b}_i$ is the restriction of $\bar{b}$ to the variables in $\bar{z}_{t_i}$. That is, there is a homomorphism h_i from q_{t_i} to D' with $h_i(\bar{x}_{t_i}) = \bar{b}_i$. Let h be a mapping defined as $h(\bar{x}_t) = \bar{a}$, $h(\bar{y}_t) = \bar{b}$, and $h(z) = h_i(z)$, for each variable z that appears in q_{t_i} but not in $(\bar{x}_t, \bar{y}_t)$ This mapping is well-defined for the following reason: Every variable that appears in q_{t_i} and q_{t_j}, for $i \neq j$, also appears in $(\bar{x}_t, \bar{y}_t)$. In fact, by definition of $\mathcal{C}_{\text{ac-full}}$ each variable z that appears in q_{t_i} but not in $(\bar{x}_t, \bar{y}_t)$ must have been introduced in some non-key position of an atom A from q_{t_i}. If z was also mentioned in an atom A' of q_{t_j}, then there would be an edge from A to A'; henceforth, T would not be a directed tree, a contradiction. We prove now that $\bar{a} \in q'_t(D')$. We do so by showing that h is a homomorphism from q_t to D'. The result then follows since $h(\bar{x}_t) = \bar{a}$. We only have to prove that h is consistent with the h_is, i.e., $h(z) = h_i(z)$ if z appears in both q_{t_i} and $(\bar{x}_t, \bar{y}_t)$. But this follows from the fact that the only variables appearing in q_{t_i} and $(\bar{x}_t, \bar{y}_t)$ are those in $\bar{x}_{t_i}$ (which we know appear also in $\bar{y}_t$); indeed, in any other case we could prove the existence of an undesired edge destroying the fact that T is a directed tree (left as an exercise). By definition we have that $h(\bar{x}_{t_i}) = \bar{b}_i = h_i(\bar{x}_{t_i})$, and hence the claim holds.

Assume now that for every repair D' of D with respect to Σ_t we have that $\bar{a} \in q'_t(D')$. Henceforth, $D \models \exists \bar{y}'_t R_t(\bar{a}, \bar{y}_t)$. Consider now an arbitrary fact $R_t(\bar{a}, \bar{b}) \in D$. First, $D \models \chi(\bar{a}, \bar{b})$. Otherwise, $\bar{a}$ does not belong to the evaluation of $\exists \bar{y} R_t(\bar{x}_t, \bar{y}_t)$ over D'', where D'' is an arbitrary repair of D that contains $R_t(\bar{a}, \bar{b})$. This contradicts the fact that $\bar{a} \in q'_t(D'')$. We show next that $D \models \psi'_{t_i}(\bar{b}_i)$, for each $i \in [1,k]$, assuming that $\bar{b}_i$ is the restriction of $\bar{b}$ to the variables in $\bar{z}_{t_i}$. By induction hypothesis, it suffices to show that for every repair D_i of D with respect to Σ_t we have that $\bar{b}_i \in q'_{t_i}(D_i)$, i.e., there is a homomorphism h from q_{t_i} to D_i with $h(\bar{x}_{t_i}) = \bar{b}_i$. Take an arbitrary such a repair D_i and let us assume that $R_t(\bar{a}, \bar{b}')$ is the only fact in the block of $R(\bar{a}, \bar{b})$ that belongs to D_i. Let D'_i be the repair defined as $(D_i - \{R_t(\bar{a}, \bar{b}')\}) \cup \{R_t(\bar{a}, \bar{b})\}$. We know that there is a homomorphism h from q_t to D'_i with $h(\bar{x}_t) = \bar{a}$. By definition, $h(\bar{y}_t) = \bar{b}$ and $h(\bar{x}_{t_i}) = \bar{b}_i$. Consider the set $h(q_{t_i})$ of facts in the image of q_{t_i} under h, for $i \in [1,k]$. None of these facts uses the relation symbol R_t because q_t is self-join free, and hence $h(q_{t_i}) \subseteq D_i$. We conclude that h is also a homomorphism from q_{t_i} to D_i with $h(\bar{x}_{t_i}) = \bar{b}_i$, and hence $\bar{b}_i \in q'_{t_i}(D_i)$. Summing up, we have that $D \models \psi'_t(\bar{a})$.

We finish by proving the following claim, which establishes that ψ_t can be defined as $\exists \bar{x}_t \psi'_t(\bar{x}_t)$ for each node t of T.

Claim 42.8. *It is the case that* $\mathrm{CQA}(\hat{q}_t, D) = \mathsf{true} \Leftrightarrow D \models \exists \bar{x}_t \psi'_t(\bar{x}_t)$*, for every database* D*.*

The fact that $D \models \exists \bar{x}_t \psi'_t(\bar{x}_t)$ implies $\mathrm{CQA}(\hat{q}_t, D) = \mathsf{true}$ easily follows from Claim 42.7, so we omit it here. Assume then that $\mathrm{CQA}(\hat{q}_t, D) = \mathsf{true}$. We show that there must exist a tuple $\bar{a}$ over $\mathrm{Dom}(D)$ such that, for every repair D' of D with respect to Σ_t, it is the case that $\bar{a} \in q'_t(D')$. From Claim

42.7 this implies that $D \models \psi'_t(\bar{a})$, and hence $D \models \exists \bar{x}_t \psi'_t(\bar{x}_t)$. We actually prove the following claim.

Claim 42.9. *Given two repairs D_1 and D_2 of D with respect to Σ_t, there is a repair D^* of D with respect to Σ_t such that $q'_t(D^*) \subseteq q'_t(D_1) \cap q'_t(D_2)$.*

Notice that from Claim 42.9 we obtain our desired result. In fact, by repeated application of this claim we have that there is a repair D^* of D with respect to Σ_t such that $q'_t(D^*) \subseteq q'_t(D')$, for each repair D' of D with respect to Σ_t. Since $\mathrm{CQA}(\hat{q}_t, D) = \mathsf{true}$, there is a homomorphism h from q_t to D^*. Therefore, $h(\bar{x}_t) \in q'_t(D')$ for each repair D' of D with respect to Σ_t.

The proof of Claim 42.9 is by induction on the height of t. When t has height 0, we have that $q'_t = \exists \bar{y}_t R_t(\bar{x}_t, \bar{y}_t)$. Take two repairs D_1 and D_2 of D with respect to Σ_t. We define D^* by choosing one fact for each block over R_t^D. Take an arbitrary such a block B. Both D_1 and D_2 must contain a fact in B, which we call $R_t(\bar{a}, \bar{b})$ and $R_t(\bar{a}, \bar{b}')$, respectively. If $\bar{a} \in q'_t(D_1) \cap q'_t(D_2)$, we add to D^* either $R_t(\bar{a}, \bar{b})$ or $R_t(\bar{a}, \bar{b}')$. Otherwise, if $\bar{a} \notin q'_t(D_1)$, we add $R_t(\bar{a}, \bar{b})$ to D^*; else, we add $R_t(\bar{a}, \bar{b}')$ to D^*. It is clear then that $q'_t(D^*) \subseteq q'_t(D_1) \cap q'_t(D_2)$.

Assume now that t has height $h+1$, for $h \geq 0$. Let $t_1, \dots, t_k$ be the children of t in T. Take two repairs D_1 and D_2 of D with respect to Σ_t, and let D^i, D_1^i and D_2^i be the restrictions of D, D_1 and D_2, respectively, to the relation symbols in q_{t_i}, for each $i \in [1, k]$. By definition, D_1^i and D_2^i are repairs of D^i with respect to Σ_{t_i}. Hence, by induction hypothesis, there is a repair D_*^i of D^i with respect to Σ_{t_i} such that $q'_{t_i}(D_*^i) \subseteq q'_{t_i}(D_1^i) \cap q'_{t_i}(D_2^i)$. We define a repair D^* of D with respect to Σ_t by taking the disjoint union of the D_*^is, for $i \in [1, k]$, plus one fact for each block B over R_t^D. We explain next how to choose such a fact.

Take an arbitrary block B over R_t^D. Both D_1 and D_2 must contain a fact in B, which we call $R_t(\bar{a}, \bar{b})$ and $R_t(\bar{a}, \bar{b}')$, respectively. If $\bar{a} \in q'_t(D_1) \cap q'_t(D_2)$, we add to D^* either $R_t(\bar{a}, \bar{b})$ or $R_t(\bar{a}, \bar{b}')$. Otherwise, if $\bar{a} \notin q'_t(D_1)$, we add $R_t(\bar{a}, \bar{b})$ to D^*; else, we add $R_t(\bar{a}, \bar{b}')$ to D^*. We claim that $q'_t(D^*) \subseteq q'_t(D_1) \cap q'_t(D_2)$. In fact, take a tuple $\bar{a} \in q'_t(D^*)$. Hence, there is a homomorphism h from q_t to D^* with $h(\bar{x}_t) = \bar{a}$. In particular, D^* contains a (unique) fact of the form $R_t(\bar{a}, \bar{b})$, for $\bar{b}$ a tuple over $\mathrm{Dom}(D)$. By definition, h is also a homomorphism from q_{t_i} to D_*^i, for each $i \in [1, k]$. By hypothesis, then, $\bar{b}_i \in q'_{t_i}(D_1^i) \cap q'_{t_i}(D_2^i)$, where $\bar{b}_i = h(\bar{x}_{t_i})$. Now, suppose for the sake of contradiction that $\bar{a} \notin q'_t(D_1)$; the other case, when $\bar{a} \in q'_t(D_1)$ but $\bar{a} \notin q'_t(D_2)$, is treated analogously. By the way in which D^* is constructed, it is the case that D_1 also contains the fact $R_t(\bar{a}, \bar{b})$. In addition, $\bar{b}_i \in q'_{t_i}(D_1^i)$ for each $i \in [1, k]$. It is easy to observe then that $\bar{a} \in q'_t(D_1)$, which is a contradiction. □

An approximation algorithm

Although evaluating consistent answers to CQs is an intractable problem, even in data complexity, analogously to the case of CQ evaluation over probabilistic databases we can show the existence of an FPRAS for computing the ratio between the number of repairs that satisfy the query and the total number of repairs. This can serve as a good measure of how "consistent" an answer is, especially in those cases in which exact evaluation of consistent answers is infeasible.

Formally, given a CQ $q(\bar{x})$, a database D, and a tuple $\bar{a}$ of elements in $\mathrm{Dom}(D)$ of the same arity as $\bar{x}$, we define

$$\mathsf{ConsLevel}(q, \bar{a}, D, \Sigma) \ := \ \frac{|\{D' \mid D' \text{ is a repair of } D \text{ wrt } \Sigma \text{ with } \bar{a} \in q'(D)\}|}{|\{D' \mid D' \text{ is a repair of } D \text{ wrt } \Sigma\}|}.$$

The next result establishes that this value can be approximated with an FPRAS when q and Σ are fixed.

Theorem 42.10. *Let $q(\bar{x})$ and Σ be a fixed CQ and set Σ of primary keys, respectively. There is an FPRAS for the problem of computing the value $\mathsf{ConsLevel}(q, \bar{a}, D, \Sigma)$ on a given database D and tuple $\bar{a}$ in Dom(D).*

The construction is very similar to the one shown in Theorem 41.9 for evaluating CQs over probabilistic databases. We leave the proof to the reader.

Bibliographic Comments

To be done.

55

Background For Tree and Graph Structured Data

Conceptually, tree- and graph-structured data is fundamentally different from relational data. The reason is that tree- and graph-structured data does not necessarily need to adhere to a schema. The reader does not need to be familiar with the entire background chapter (Chapter 2) to start with Parts VIII and X. We assume here that the reader is familiar with the *Basic Notions and Notation* from Chapter 2 and introduce everything else here or give the reader an explicit pointer to Chapter 2.

Graphs

Throughout Parts VIII and X, we assume that we have a countably infinite set

$$\mathsf{Nodes} \text{ of } \textit{nodes}$$

available, from which we will take the nodes in trees and graphs. We also assume that we have a countably infinite set

$$\mathsf{Lab} \text{ of } \textit{labels.}$$

Furthermore, we assume the existence of a special symbol $*$ that does not occur in any of the aforementioned sets and which we will use as a "wildcard" in query languages over trees.

We next define node- and edge-labled directed graphs, which will be the basis of different data models that we will use throughouth Parts VIII and X. In particular, we will use special cases of the following definition. The main reason why we introduce this general definition here is because we want to define *queries* and some of their computational problems in this chapter.

Definition 55.1: Node- and Edge-Labeled, Directed Graph

Let $k \in \mathbb{N}$. A *k-sorted node- and edge-labeled, directed graph with wildcards* is a tuple

$$G = (V, E_1, \dots, E_k, \text{lab}) ,$$

where

- $V \subseteq \mathsf{Nodes}$ is a finite, nonempty set of *nodes*,
- $E_i \subseteq V \times V$ is a set of directed *edges* for every $i \in [k]$, and
- $\text{lab}\colon V \cup E_1 \cup \dots \cup E_k \to \mathsf{Lab} \cup \{*\}$ is a partial function that assigns to every node and edge its label from Lab or the wildcard symbol.

We say that G is *node-labeled* (resp., *edge-labeled*) if $\text{Dom}(\text{lab}) = V$ (resp., $\text{Dom}(\text{lab}) = E_1 \cup \dots \cup E_k$).

We will use the k different sorts of edges to be able to assign different roles to edges in the graph. For example, in Part VIII we will use this feature to define *child* edges and *next-sibling* edges in trees.

Usually, we will use a restricted labeling function that does not assign wildcards, that is, $\text{lab}\colon V \cup E_1 \cup \dots \cup E_k \to \mathsf{Lab}$. If this is the case, we will omit "with wildcards" when we refer to G, that is, we will talk about a *k-sorted node-labeled, directed graph*, etc. Furthermore, we will often leave "k-sorted" implicit, since the number k can be inferred from the tuple $(V, E_1, \dots, E_k, \text{lab})$.

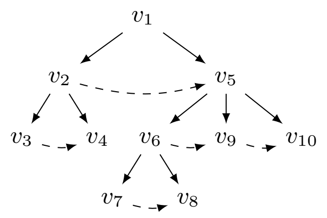
Example 55.2

Figure 55.1 has a graphical representation of a 2-sorted node- and edge-labeled directed graph $G = (V, E_1, E_2, \text{lab})$ where

- $V = \{v_1, \dots, v_{10}\}$,
- $E_1 = \{(v_1, v_2), (v_2, v_3), (v_2, v_4), (v_1, v_5), (v_5, v_6), (v_6, v_7), (v_6, v_8), (v_5, v_9), (v_5, v_{10})\}$,
- $E_2 = \{(v_2, v_5), (v_3, v_4), (v_6, v_9), (v_7, v_8), (v_9, v_{10})\}$, and
- lab is defined in the Figure.

The two edge sorts are represented with solid and dashed arrows, respectively.

Since presenting graphs as in Figure 55.1 is very verbose, we will leave the elements of V implicit whenever we can. For example, the graph of Figure 55.1 can be represented more compactly as in Figure 55.2. Notice that, for the sake of brevity, we omitted some of the elements of V. (We only mention v_4, v_5, v_8, and v_{10}.) We may do this in cases where only some of the elements of V are relevant. For instance, if we view



with

$$\begin{aligned} \mathrm{lab}(v_1) &= a & \mathrm{lab}(v_6) &= b\\ \mathrm{lab}(v_2) &= b & \mathrm{lab}(v_7) &= c\\ \mathrm{lab}(v_3) &= c & \mathrm{lab}(v_8) &= d\\ \mathrm{lab}(v_4) &= d & \mathrm{lab}(v_9) &= a\\ \mathrm{lab}(v_5) &= e & \mathrm{lab}(v_{10}) &= b \end{aligned}$$

$$\begin{aligned} \mathrm{lab}(v_1,v_2) &= \mathrm{lab}(v_2,v_3) = \mathrm{lab}(v_2,v_4) = \mathrm{lab}(v_1,v_5) = \mathrm{lab}(v_5,v_6)\\ &= \mathrm{lab}(v_6,v_7) = \mathrm{lab}(v_6,v_8) = \mathrm{lab}(v_5,v_9) = \mathrm{lab}(v_5,v_{10}) = \mathrm{c} \end{aligned}$$

$$\mathrm{lab}(v_2,v_5) = \mathrm{lab}(v_3,v_4) = \mathrm{lab}(v_6,v_9) = \mathrm{lab}(v_7,v_8) = \mathrm{lab}(v_9,v_{10}) = \mathrm{ns}$$

Fig. 55.1: Example of a Node- and Edge-Labeled Graph

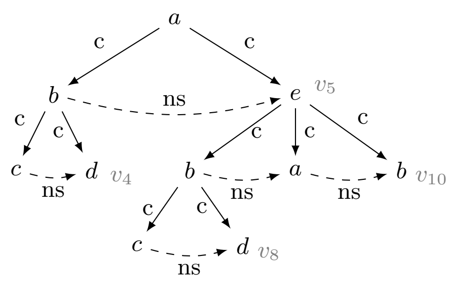


Fig. 55.2: Less verbose representation of the graph in Figure 55.1, leaving some of the elements of Nodes implicit

Figure 55.2 as a tree-like structure, we may want to say that v_4, v_5, v_8, and v_{10} are "rightmost children of their respective parents". The nodes we omitted are irrelevant for this statement.

In Parts VIII–X, we will use the letters u and v for referring to nodes. Let $G = (V, E_1, \ldots, E_k, \mathrm{lab})$ and $E = E_1 \cup \cdots \cup E_k$. An *undirected path* in G is a non-empty sequence of nodes

$$\pi = v_0 v_1 \cdots v_n$$

where $(v_{i-1}, v_i) \in E$ or $(v_i, v_{i-1}) \in E$ for every $i \in [n]$. We say that π is *from* v_0 *to* v_n and has *length* n. (The path of length zero is from v_0 to v_0.) A *directed path* in G is an undirected path where $(v_{i-1}, v_i) \in E$ for every $i \in [n]$. We assume in Parts VIII–X that paths are directed unless stated otherwise. A graph is *connected* if, for every pair of nodes $u, v \in V$, there is an undirected path from u to v. Notice that these definitions are consistent with the corresponding definitions for directed graphs in Chapter 2.

Data Graphs

For the following definition, we recall from Chapter 2 that we use Const throughout the book to denote the set of *values* in databases.

Definition 55.3: k-Sorted Data Graph

Let $k \in \mathbb{N}$. A *k-sorted data graph* is a tuple

$$G = (V, E_1, \ldots, E_k, \text{lab}, d) \ ,$$

where:

- $(V, E_1, \ldots, E_k, \text{lab})$ is a k-sorted node- and edge-labeled directed graph, and
- $d \colon V \cup E_1 \cup \cdots \cup E_k \to \mathsf{Const}$ is a partial function that assigns to every node and edge its *data value*.

We write $\mathcal{G}$ for the set of all k-sorted data graphs for some $k \in \mathbb{N} - \{0\}$.

When the number k of sorts is not important, we may use the term *data graph* to refer to a k-sorted data graph for some $k \in \mathbb{N} - \{0\}$.

Notice that we assume a single data value per node. This is just for notational simplicity and is not a restriction at all, since a node u with $k > 1$ attributes can be modeled by a node u with k outgoing edges, leading to k new nodes containing the data values.

Queries

Next we define queries over data graphs. Since Parts VIII–X will sometimes consider strict subsets of the data graphs (e.g., node-labeled trees, edge-labeled graphs), we define queries for subsets $\mathcal{G}'$ of $\mathcal{G}$.

Definition 55.4: Queries and Query Languages over Graphs

Let $\mathcal{G}' \subseteq \mathcal{G}$. A k-ary *data graph query over $\mathcal{G}'$* is a function of the form

$$q \ : \ \mathcal{G}' \to (\mathsf{Nodes} \cup \mathsf{Const})^k \ .$$

A *data graph query language* is a set of data graph queries.

Key Problems: Query Evaluation and Query Analysis

We now define some key problems concerning tree and graph-structured data. In their most common form, these problems are parameterized by a data graph query language $\mathcal{L}$. Such query languages in Parts VIII–X will always be associated to a *data model*, i.e., a subset $\mathcal{G}_{\mathcal{L}} \subseteq \mathcal{G}$ onto which the semantics of $\mathcal{L}$ is defined. The intention is that queries from $\mathcal{L}$ will always be evaluated over elements of $\mathcal{G}_{\mathcal{L}}$.

Query Evaluation

Problem: $\mathcal{L}$-Evaluation

Input: A query q from $\mathcal{L}$, a data graph $G \in \mathcal{G}_{\mathcal{L}}$, a tuple $\bar{a}$ over $\mathsf{Nodes} \cup \mathsf{Const}$

Output: `true` if $\bar{a} \in q(G)$, and `false` otherwise

Query Containment and Equivalence

Let $\mathcal{L}$ be a data graph query language. We say that a query $q \in \mathcal{L}$ is *contained* in a query $q' \in \mathcal{L}$, written as $q \subseteq q'$, if $q(G) \subseteq q'(G)$ for every data graph $G \in \mathcal{G}_{\mathcal{L}}$. Query q is *equivalent* to q', written as $q \equiv q'$, if $q(G) = q'(G)$ for every data graph $G \in \mathcal{G}_{\mathcal{L}}$. Since, by definition, queries return sets of tuples, the notions of set inclusion and equality can be applied.

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

Part VIII

Tree-Structured Data

Tree-structured data came to the attention of the data management community with the introduction of the Extensible Markup Language (XML) in 1998, which has since then become one of the most widespread formats for exchanging data on the Web. The other popular data interchange format, the JavaScript Data Interchange Standard (JSON), was defined in 1999 and is similar to XML in the sense that it also treats data in a tree structured manner. Tree-structured data is therefore an important part of data management and we devote this part to discuss some of its fundamental aspects.

Part VIII is organized as follows. In Chapter 56, we discuss the data models that we will use throughout this part. [TODO add explaining sentence.] We then define *tree pattern queries* in Chapter 61. Tree pattern queries are a fragment of the language XPath, and can navigate through trees using the *child* and *descendant* relations. As such, they are among the most fundamental languages one can devise for querying trees. They arguably play a similar role for tree-structured data as conjunctive queries do for relational data. In Chapters 62 and 63 we treat the basic static analysis and optimization problems for tree pattern queries, namely *containment* and *minimization.*

Chapters 61 and 63 are presented in terms of unordered trees, that is, trees in which siblings are unordered. However, since tree pattern queries are agnostic of the sibling ordering in the data, all the results in Chapters 61 and 63 remain to hold in the setting where sibling ordering is present in the data.

56

Data Model

In this chapter we introduce several abstractions of tree-structured data. We focus on abstractions that are simple and elegant, yet powerful enough to prove results that help us understand the nature of querying and reasoning about tree-structured data in practice. Before we dive into our mathematical models, we take a look at how tree-structured data is represented in practice.

Tree-Structured Data in Practice

The most widespread formats for storing tree-structured data are XML (eXtensible Markup Language) and JSON (JavaScript Object Notation). We briefly discuss the nuts and bolts of these formats by example.

XML

Consider Figure 56.1(a), which describes a simple XML document that contains IDs, names, and birthplaces of three persons. The first line states that the document uses version 1.0 of the XML standard and is encoded in UTF-8. The actual data starts in the second line and is hierarchically structured, much like HTML.[1] In XML, the tags such as `persons` and `person` are called *element names*, whereas `pers_id`, `name`, and `birthplace` are *attribute names*. In the document, the top-level element `person` contains three elements, labeled `person`, each of which has attributes with names `pers_id`, `name`, and `birthplace`. The values of the respective attributes are given as strings inside double quotes. For instance, the birthplace of Jimi is Seattle.

Figure 56.1(a), however, does not significantly use the capabilities of XML to structure data as a tree. So let us consider Figure 56.1(b), which contains similar information than Figure 56.1(a), but with more detail. Compared to Figure 56.1(a), we made two significant changes. First, instead of writing the

[1] For brevity, we assume that the reader has basic familiarity with HTML, but at the same time encourage her/him to continue reading even if this is not the case.

```
<?xml version="1.0" encoding="UTF-8"?>
<persons>
  <person pers_id="1" name="Jimi" birthplace="Seattle"/>
  <person pers_id="2" name="Saul" birthplace="Stoke-on-Trent"/>
  <person pers_id="3" name="Mark" birthplace="Glasgow"/>
</persons>
```

(a)

```
<?xml version="1.0" encoding="UTF-8"?>
<persons>
  <person pers_id="1">
    <name> Jimi </name>
    <birthplace>
      <city> Seattle </city>
      <state> Washington </state>
      <country> United States </country>
    </birthplace>
  </person>
  <person pers_id="2">
    <name> Saul </name>
    <birthplace>
      <city> Stoke-on-Trent </city>
      <country> United Kingdom </country>
    </birthplace>
  </person>
  <person pers_id="3">
    <name> Mark </name>
    <birthplace>
      <city> Glasgow </city>
      <country> United Kingdom </country>
    </birthplace>
  </person>
</persons>
```

(b)

Fig. 56.1: Two XML documents

names of persons as XML attributes, we wrote them inside `name`-tags which are nested inside `person`-tags. (Notice that the values of names are no longer enclosed in double quotes.) Second, we did the same with birthplaces, but additionally provided more detail in terms of the state and country of the city of birth. Furthermore, we did not provide state information for every city of birth.

We now discuss some principled differences between the two documents. First of all, the ordering between attributes in XML is irrelevant. That is, the element

```
<person pers_id="1" name="Jimi" birthplace="Seattle"/>
```

conveys the same information than

```
<person name="Jimi" birthplace="Seattle" pers_id="1"/> .
```

Intuitively, XML considers both fragments as encoding an element with name `person` for which the attribute with name `pers_id` has value “1”, `name` has value “Jimi”, and `birthplace` has value “Seattle”.

Ordering between elements, however is important. That is, the fragment

```
<person> <pers_id> 3 </pers_id> <name> Mark </name> </person>
```

is considered to be different from

```
<person> <name> Mark </name> <pers_id> 3 </pers_id> </person>
```

in XML. The reason is that XML treats the hierarchical nesting of elements as a tree where, in the first fragment, the *first child* of `person` is a `pers_id`-element, whereas in the second fragment, the *first child* of `person` is a `name`-element, which is different.

JSON

Let us now consider how JSON represents tree-structured data. In JSON, a natural way to represent the information similar to that in Figure 56.1(a) would be the document in Figure 56.2(a). Another way to represent this information would be in Figure 56.2(b), which is closer to our XML representation in Figure 56.1(a).

Before we dive into detail on these examples of JSON documents, let us give a quick overview of how JSON documents are constructed, focusing on aspects that are relevant to this book. *JSON documents* can be inductively defined as follows. The simplest JSON documents are *string values*, which are strings enclosed in double quotes, such as `"persons"`, `"person"`, `"pers_id"`, `"1"`, etc. If $k_1, \ldots, k_n$ are pairwise distinct string values and $v_1, \ldots, v_n$ are JSON documents, then

$$\texttt{\{}\ k_1 : v_1 \texttt{,}\ \ldots \texttt{,}\ k_n : v_n\ \texttt{\}}$$

is a JSON document, called a *JSON object*. For each $i \in [n]$, we refer to $k_i : v_i$ as a *key-value* pair of the object. Furthermore,

$$\texttt{[}\ v_1, \ldots, v_n\ \texttt{]}$$

is a JSON document, called a *JSON array*. It is important to note that, inside JSON objects and arrays, the v_i can again be objects and arrays, which gives rise to the hierarchical or tree-structured nature of JSON documents.

Objects and arrays are used in JSON to represent ordered and unordered information, respectively. That is,

```
{"pers_id":"3", "name":"Mark", "birthplace":"Glasgow"}
```

and

```
{"name":"Mark", "birthplace":"Glasgow", "pers_id":"3"}
```

represent the same information, but

```
["pers_id":"3", "name":"Mark", "birthplace":"Glasgow"]
```

and

```
["name":"Mark", "birthplace":"Glasgow", "pers_id":"3"]
```

do not, since the order inside arrays is important.

Let us now discuss the two JSON documents in Figure 56.2. We can now see that the document in Figure 56.2(b) indeed represents a similar structure than the XML document in Figure 56.1(a). We have a `persons` entity, which contains an ordered list of three `person` entities. Each of these contains a JSON object that describes their "attribute names" `pers_id`, `name`, and `birthplace`, and their respective values. The data in Figure 56.1(a) could be described in exactly the same way. Notice that, in this example, the requirement that keys inside JSON objects are pairwise disjoint corresponds to the requirement that attribute names in XML elements are pairwise disjoint.

The JSON document in Figure 56.2(a) represents the information a bit differently, since it omits the `person`-keys and immediately packs the information on the three persons in an array that contains three objects.

Finally, we show in Figure 56.3 how tree-structured information as in Figure 56.1b can be modeled in JSON. Notice that we needed to add a string `data` to give a name to the array that contains the ordered name and birthplace information of persons. The ordering between the person ID and this array is irrelevant.

Labeled Unordered Trees

We now turn to mathematical abstractions of tree-structured data. The simplest such abstraction is the one of *labeled unordered trees*, which is based on node-labeled directed graphs.

```
{"persons": [
  {"pers_id":"1", "name":"Jimi", "birthplace":"Seattle"},
  {"pers_id":"2", "name":"Saul", "birthplace":"Stoke-on-Trent"},
  {"pers_id":"3", "name":"Mark", "birthplace":"Glasgow"}
 ]}
```

(a)

```
{"persons": [
  {"person":
    {"pers_id":"1", "name":"Jimi", "birthplace":"Seattle"}},
  {"person":
    {"pers_id":"2", "name":"Saul", "birthplace":"Stoke-on-Trent"}},
  {"person":
    {"pers_id":"3", "name":"Mark", "birthplace":"Glasgow"}}
 ]}
```

(b)

Fig. 56.2: Two similar JSON documents

Definition 56.1: Labeled Unordered Tree

A connected, node-labeled, directed graph $T = (V, E, \text{lab})$ is a *labeled unordered tree* if,

- for every node v, there is at most one node u with $(u, v) \in E$ and
- there is exactly one node v (called the *root* of T) without an incoming edge (u, v).

If T is a labeled unordered tree, we denote its root by $\text{Root}(T)$. Furthermore, if $(u, v) \in E$ then we call v a *child of* u and u the *parent of* v. Likewise, we refer to E as the set of *child edges* of T. We call v a *descendant* of u if there exists a non-empty path from u to v in T. In this case, we also call u and *ancestor* of v. A *leaf* is a node without a child, that is, a node u such that $\{v \mid (u, v) \in E\}$ is empty. Two nodes are *siblings* if they have the same parent. We extend this terminology to sets of nodes and say that S is a set of siblings if all pairs of elements from S are siblings. A set of siblings S is *maximal* if there does not exist a node $u \in V - S$ such that $S \cup \{u\}$ is a set of siblings. By $T_{|u}$ we denote the subtree of T rooted at node u, formally defined as $(V_{|u}, E_{|u}, \text{lab}_{|u})$, where $V_{|u}$ consists of u and all its descendants in T, the relation $E_{|u}$ is $E \cap V_{|u} \times V_{|u}$, and $\text{lab}_{|u}$ is the restriction of lab to $V_{|u}$.

We denote the empty labeled unordered tree by ε. For $a \in \mathsf{Lab}$, we use the notation $a(T_1, \ldots, T_n)$ to denote the labeled unordered tree in which the root has label a and has n children $u_1, \ldots, u_n$. Furthermore, T_i is its subtree at node u_i for each $i \in [n]$. That is, for each $i \in [n]$, we have that T_i is a labeled unordered tree and $a(T_1, \ldots, T_n)_{|u_i} = T_i$. Notice that this definition implies that T_i is never empty.

```
{"persons": [
  {"person":
    {"pers_id": "1",
     "data": [
       {"name": "Jimi"},
       {"birthplace": [
         {"city": "Seattle" },
         {"state": "Washington"},
         {"country": "United States"}]
       }]
    }
  },
  {"person":
    {"pers_id": "2",
     "data": [
       {"name": "Saul"},
       {"birthplace": [
         {"city": "Stoke-on-Trent"},
         {"country": "United Kingdom"}]
       }]
    }
  },
  {"person":
    {"pers_id": "3",
     "data": [
       {"name": "Mark"},
       {"birthplace": [
         {"city": "Glasgow"},
         {"country": "United Kingdom"}]
       }]
    }
  }]
}
```

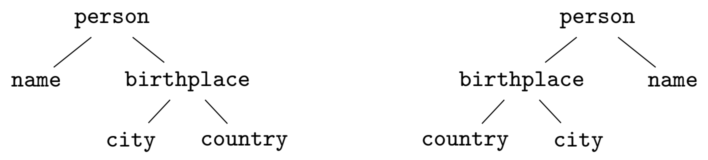
Fig. 56.3: A JSON document that describes ordered and unordered information in a similar way as the XML document in Figure 56.1b

The *degree* of a node is the number of its children. The degree of a tree is the maximum degree of any of its nodes. The *depth* of the empty tree ε is zero, the depth of a tree $T = a$ is one, and the depth of $T = a(T_1, \ldots, T_n)$ is one plus the maximal depth of $T_1, \ldots, T_n$.

We depict trees in the usual way as in Figure 56.4(b), with the root node on top and the leafs at the bottom. Since all edges are directed downward (or away from the root) in such pictures, we omit arrows. The trees we defined until now are *unordered*, which means that we consider the children of a node as an unordered set. Therefore, the trees in Figures 56.4(b) and 56.4(c)

```
{"person":
  {"name": "Jimi",
   "birthplace": { "city": "Seattle",
                   "country": "United States"}
  }
}
```

(a)



(b) (c)

Fig. 56.4: A JSON document and two illustrations of the unordered tree that captures the nesting of its keys

formally represent the same unordered tree, which corresponds to the nesting structure of the keys in the JSON document in Figure 56.4(a).

Recall, however, that there is a fundamental difference between the nesting of keys in JSON objects and labeled unordered trees. Labeled unordered trees can have siblings with the same label, whereas key values on the same level of a JSON object need to be unique. We choose not to impose this restriction on labeled unordered trees, because

(1) this allows us to present results that are relevant to both JSON and XML in Chapters 61–63 and
(2) the results in Chapters 61–63 are not affected by this choice.

Labeled Ordered Trees

In many situations, the ordering between siblings is important. For instance, if we want to describe an address, we may want to start with a street name and number, followed by a city, followed by a country. We will model such behavior by labeled trees in which the siblings are ordered. To this end, let S be a finite set. We say that a binary relation R *is a successor relation on* S if the restriction of R to S, that is, $\{(a,b) \in R \mid a \in S, b \in S\}$ is isomorphic to $\{(1,2),(2,3),\ldots,(|S|-1,|S|)\}$.

Definition 56.2: Labeled Ordered Tree

A *labeled ordered tree* is a tuple

$$T = (V, E^{\mathrm{c}}, E^{\mathrm{ns}}, \mathrm{lab}) \ ,$$

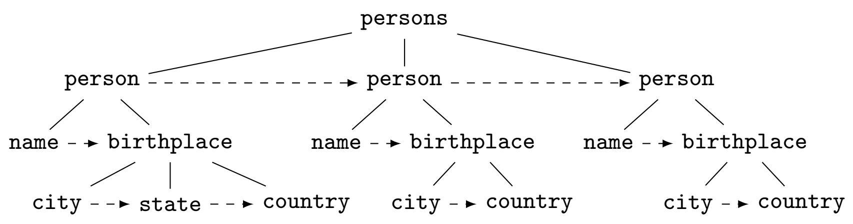


Fig. 56.5: A labeled ordered tree, capturing the nesting of the element names of the XML document in Figure 56.1(b).

where $(V, E^{\mathrm{c}}, \mathrm{lab})$ is a labeled unordered tree and E^{ns} is a successor relation on every maximal set of siblings of $(V, E^{\mathrm{c}}, \mathrm{lab})$.

Notice that a labeled ordered tree is a two-sorted node-labeled directed graph according to Definition 55.1. We will use all the terminology that we introduced for labeled unordered trees (such as *root*, *child*, *degree*, etc.) in the context of labeled ordered trees. We refer to E^{ns} as the *next sibling* relation. The *first child* (resp., *last child*) *of* u *in* T is only defined if u has at least one child and is the smallest (resp., largest) element among all the children of u according to E^{ns}. We say that v is a *following sibling* of u if there exists a non-empty path from u to v in T whose edges are all in E^{ns}. Whenever T is clear from the context, we may omit "in T" when we discuss nodes an the relationships between them. A set of labeled ordered trees is also called a *tree language*.

Similarly to unordered labeled trees, we denote by $T_{|u}$ the subtree of T rooted at node u, formally defined as $(V_{|u}, E^{\mathrm{c}}_{|u}, E^{\mathrm{ns}}_{|u}, \mathrm{lab}_{|u})$, where $V_{|u}$ consists of u and all its descendants, $\mathrm{lab}_{|u}$ is the restriction of lab to $V_{|u}$ and the relations $E^{\mathrm{c}}_{|u}$ and $E^{\mathrm{ns}}_{|u}$ are $E^{\mathrm{c}} \cap V_{|u} \times V_{|u}$, and $E^{\mathrm{ns}} \cap V_{|u} \times V_{|u}$, respectively.

We denote the empty labeled ordered tree by ε. For $a \in \mathsf{Lab}$, we use the notation $a(T_1, \dots, T_n)$ to denote the labeled ordered tree in which the root has label a and has n children $u_1, \dots, u_n$. Furthermore, T_i is its subtree at node u_i for each $i \in [n]$. That is, for each $i \in [n]$, we have that T_i is a labeled ordered tree and $a(T_1, \dots, T_n)_{|u_i} = T_i$.

Figure 56.5 depicts the ordered tree that captures the nested structure of the element names in the XML document of Figure 56.1(b). The edges in E^{c} are solid, whereas the edges in E^{ns} are dashed. As in Figure 56.4(c), we omit the arrows on the edges in E^{c}, since they all point downward. Furthermore, notice that all the edges in E^{ns} can be inferred from the figure, once we know that the tree is supposed to be a labeled ordered tree. (They just go from left to right among the children of nodes.) For this reason, we will therefore also omit the arrows in E^{ns} in illustrations of labeled ordered trees. To avoid confusion, we will always make clear in the context whether a figure illustrates a labeled unordered tree or a labeled ordered tree.

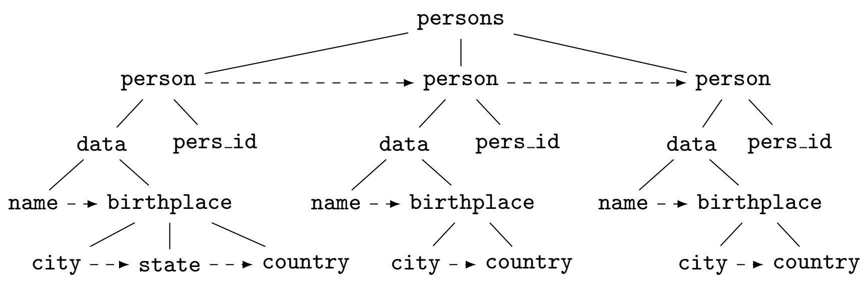


Fig. 56.6: A two-sorted node-labeled graph, capturing the nesting and ordering of the element keys of the JSON document in Figure 56.3.

Theory versus Practice

We have now seen two fundamental theoretical abstractions of tree-structured data, namely *labeled unordered trees* and *labeled ordered trees*, together with a few examples of how tree-structured data can be represented in practice, using JSON and XML. As the careful reader undoubtedly realizes, our two abstractions are not even powerful enough to precisely capture our practical examples. For instance, for representing the nesting of keys in the JSON document in Figure 56.3, we would arguably require a structure as in Figure 56.6, which is neither a labeled ordered tree, nor a labeled unordered tree.

So why do we work with labeled ordered or unordered trees? This is not because we are we are lazy or uninterested in the gritty details of reality. Our aim is rather to work with elegant and simple abstractions that still allow us to study practically relevant issues. In fact, simple abstractions are often very helpful to distill practical challenges to their mathematical core. They enable us to discover the mathematical questions whose solution would imply a solution to the practical challenge. Good abstractions help us to focus our attention on the fundamental underlying issues.

In this spirit, all results in Part VIII are presented in terms of labeled ordered or unordered trees. As such, they are immediately applicable to those parts of XML or JSON that can be understood as labeled ordered or unordered trees, respectively. In many cases, the results can even be trivially generalized to abstractions that are closer to reality, e.g., abstractions that consider a mixture of ordered and unordered siblings as in Figure 56.6.

We finally argue that there is no "best" way to abstract tree-structured data as a tree. For example, the JSON document in Figure 56.4(a) can also be represented as the labeled unordered tree in Figure 56.7, in which we treat both the JSON keys and the *data values* 'Jimi', 'Seattle', and 'United States' as node labels. This aspect is important for understanding many results in Part VIII. Although these results are about labeled ordered/unordered trees, this does not mean that the results are only about the nesting structure of element names in XML document or keys in JSON documents and therefore

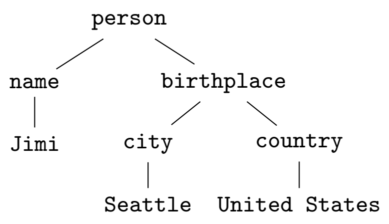


Fig. 56.7: Another way of seeing the JSON document in Figure 56.4(a) as a labeled unordered tree

do not concern the actual data. In fact, when it comes to interpreting the results in Part VIII for managing XML or JSON data, it is more instructive to consider the interaction between *data* and *queries*. In Part VIII, we will consider query languages that allow to compare labels to fixed values that we specify in the query. An example of such a query is

Select all nodes with label 'Seattle',

which compares node labels to the fixed value 'Seattle'. Therefore, such results also generalize to query languages that can compare data values to constants. Later in the book, we will consider query languages that can compare data values to each other, which is fundamentally different. An example of a query that compares values to each other is

Select all node pairs (x, y) *such that* x *and* y *have different labels.*

First-Order Logic over Trees

In this chapter we introduce first-order logic over labeled ordered trees, to which we refer as "first order logic over trees" for the sake of brevity. This logic is similar to first-order logic (Chapter 3), but is tuned towards expressing properties over labeled ordered trees. Just like its relational counterpart, first-order logic over trees is an important yardstick for the expressiveness of query languages over trees.

Formulae in first-order logic over trees will use relation names from the set

$$\{\mathrm{Lab}_a \mid a \in \mathsf{Lab}\} \cup \{E^{\mathrm{fc}}, E^{\mathrm{ns}}, E^{\mathrm{d}}, E^{\mathrm{fs}}\} \ .$$

Here, Lab_a is called the *a-label relation*, E^{fc} is called the *first-child* relation, E^{ns} the *next-sibling* relation, E^{d} the *descendant* relation, and E^{fs} the *following-sibling* relation, respectively.[1]

We will follow the Background chapter (Chapter 2) in the sense that we assume that we have a countably infinite set Var of *variables*, disjoint from Nodes and Lab. As in Chapter 3, variables will be typically denoted by $x, y, z, \ldots$ (possibly with subscripts and superscripts). Differently from Chapter 3, the symbols $a, b, c, \ldots$ are now used to to denote *labels* from Lab. Formulae of first-order logic are inductively defined using terms, conjunction ($\wedge$), disjunction ($\vee$), negation ($\neg$), existential quantification ($\exists$), and universal quantification ($\forall$).

Definition 57.1: Syntax of First-Order Logic over Trees

We define *formulae of first-order logic* (FO) over trees as follows:

- If x is a variable, then $\mathrm{Lab}_a(x)$, for every $a \in \mathsf{Lab}$ is an atomic formula.
- If x and y are variables from Var, then $x = y$ is an atomic formula.

[1] The reader may wonder why we don't use the child relation or the last-child relation, but we will see that these can be expressed using the available relations.

- If x and y are variables from Var, then $E^{\mathrm{fc}}(x, y)$, $E^{\mathrm{ns}}(x, y)$, $E^{\mathrm{d}}(x, y)$, $E^{\mathrm{fs}}(x, y)$, and are atomic formulae.
- 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.

Analogously to the first-order formulae for relational databases in Chapter 3, the *size* $\|\varphi\|$ of φ is defined to be the total number of labels, variables, and symbols from $\{\wedge, \vee, \neg, =, \exists, \forall\}$ occurring in φ. The *free variables* $\mathrm{FV}(\varphi)$ *of* φ are defined analogously as in first-order logic over relational databases. We also omit brackets in the same manner as for first order formulae for relational databases to avoid notional clutter.

Example 57.2: First-Order Formulae

Consider the following FO-formula over trees:

$$\mathrm{Lab}_{\text{`person'}}(x) \wedge \mathrm{Lab}_{\text{`name'}}(y) \wedge E^{\mathrm{fc}}(x, y)\ . \tag{57.1}$$

The free variables of this formula are x and y. The three formulae

$$\mathrm{Lab}_{\text{`person'}}(x) \wedge \neg\exists y \Big(\mathrm{Lab}_{\text{`state'}}(y) \wedge \big(\exists z \exists z_1 \exists z_2\ \mathrm{Lab}_{\text{`birthplace'}}(z) \wedge E^{\mathrm{fc}}(x, z_1) \wedge E^{\mathrm{ns}}(z_1, z) \wedge E^{\mathrm{fc}}(z, z_2) \wedge E^{\mathrm{ns}}(z_2, y)\big)\Big)\ , \tag{57.2}$$

$$\neg\exists y\ E^{\mathrm{ns}}(x, y)\ , \tag{57.3}$$

and

$$\neg\exists y\ \big(E^{\mathrm{fc}}(y, x) \vee E^{\mathrm{ns}}(y, x)\big) \tag{57.4}$$

have only one free variable, namely x. The formula

$$E^{\mathrm{fc}}(x, y) \vee \big(\exists z\ E^{\mathrm{fc}}(x, z) \wedge E^{\mathrm{fs}}(z, y)\big) \tag{57.5}$$

has two free variables, namely x and y.

Given a labeled ordered tree $T = (V, E^{\mathrm{c}}, E^{\mathrm{ns}}, \mathrm{lab})$, we define the notion of satisfaction of a formula φ with respect to an *assignment* η *for* φ *over* T. Such an assignment is a function from $\mathrm{FV}(\varphi)$ to V. If x is a variable and $v \in V$ a node of T, we use $\eta[x/v]$ to denote the assignment that modifies η by setting $\eta(x) = v$. We are now ready for defining the semantics of first-order formulae over trees.

Definition 57.3: Semantics of First-Order Logic over Trees

Given a tree $T = (V, E^{\mathrm{c}}, E^{\mathrm{ns}}, \mathrm{lab})$, a formula φ, and an assignment η for φ over T, we inductively define when φ is *satisfied* in T under η, written $(T, \eta) \models \varphi$, as follows:

- If φ is $\mathrm{Lab}_a(x)$, then $(T, \eta) \models \varphi$ if $\mathrm{lab}(\eta(x)) = a$.
- If φ is $x = y$, then $(T, \eta) \models \varphi$ if $\eta(x) = \eta(y)$.
- If φ is $E^{\mathrm{fc}}(x, y)$, then $(T, \eta) \models \varphi$ if $\eta(y)$ is the first child of $\eta(x)$ in T.
- If φ is $E^{\mathrm{ns}}(x, y)$, then $(T, \eta) \models \varphi$ if $\eta(y)$ is a next sibling of $\eta(x)$ in T.
- If φ is $E^{\mathrm{d}}(x, y)$, then $(T, \eta) \models \varphi$ if $\eta(y)$ is a descendant of $\eta(x)$ in T.
- If φ is $E^{\mathrm{fs}}(x, y)$, then $(T, \eta) \models \varphi$ if $\eta(y)$ is a following sibling of $\eta(x)$ in T.
- If $\varphi = \varphi_1 \wedge \varphi_2$, then $(T, \eta) \models \varphi$ if $(T, \eta) \models \varphi_1$ and $(T, \eta) \models \varphi_2$.
- If $\varphi = \varphi_1 \vee \varphi_2$, then $(T, \eta) \models \varphi$ if $(T, \eta) \models \varphi_1$ or $(T, \eta) \models \varphi_2$.
- If $\varphi = \neg\psi$, then $(T, \eta) \models \varphi$ if $(T, \eta) \models \psi$ does not hold.
- If $\varphi = \exists x\, \psi$, then $(T, \eta) \models \varphi$ if $(T, \eta[x/v]) \models \psi$ for *some* node $v \in V$.
- If $\varphi = \forall x\, \psi$, then $(T, \eta) \models \varphi$ if $(T, \eta[x/v]) \models \psi$ for *every* node $v \in V$.

Formulae of FO over trees are therefore very similar to formulae of FO over relations (Chapter 3) "over $\{\mathrm{Lab}_a \mid a \in \mathsf{Lab}\} \cup \{E^{\mathrm{fc}}, E^{\mathrm{ns}}, E^{\mathrm{d}}, E^{\mathrm{fs}}\}$", with the following differences:

(1) Formally, assignments for formulae of FO over relations map variables to Const, whereas they map variables to Nodes for formulae of FO over trees. This means that formulae in FO over relations are interpreted over Const, whereas formulae in FO over trees are interpreted over Nodes.

(2) Formulae in FO over relations have atomic formulae of the form "$x = a$", which allow equality tests between variables and constants. We do not have a similar construct here (which would be an equality test between a variable and a node).

(3) Whereas a schema $\mathbf{S}$ in Chapter 3 is always finite, the set $\{\mathrm{Lab}_a \mid a \in \mathsf{Lab}\} \cup \{E^{\mathrm{fc}}, E^{\mathrm{ns}}, E^{\mathrm{d}}, E^{\mathrm{fs}}\}$ is infinite, becaues Lab is infinite. This is only a minor difference in the sense that it is similar to "omitting" the schema in the relational world. (Each FO formula "over Rel" is an FO formula over some schema $\mathbf{S}$.)

Example 57.4: Semantics of First-Order Formulae

We give an intuitive description of the semantic meaning of the formulae in Example 57.2:

- Formula (57.1) is satisfied by all nodes x and y such that x has label 'person', y has label 'name', and y is the first child of x.
- Formula (57.2) is satisfied by all nodes x with label 'person', such that there does not exist a node y which is labeled 'state', and y is the second child of a node labeled 'birthplace', which is in turn the second child of x. In Figure 56.5, such nodes x would be the second and third children of the root, i.e., nodes that do not have a 'state' child of their 'birthplace' child.
- Formula (57.3) is satisfied by all nodes x that do not have next siblings. That is, x is a last sibling (or the root).
- Formula (57.4) is satisfied by all nodes x such that there does not exist a node y from which x is the first child or the next sibling. That is, x is the root of the tree.
- Formula (57.5) is satisfied by all nodes x and y such that y is a child of x.

First-Order Queries over Trees

Recall that a k-ary data graph query is a function q that takes a data graph G as input and produces a set $q(G) \subseteq (\mathsf{Nodes} \cup \mathsf{Const})^k$. First-order queries over trees are a special case of such queries, since their inputs are labeled ordered trees and they produce a set of k-tuples of nodes, that is, a set in Nodes^k. We first define the syntax of first-order queries over trees.

Definition 57.5: First-Order Queries over Trees

A *first-order query over trees* is an expression of the form $\varphi(\bar{x})$, where φ is an FO formula over trees, 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 $\|\varphi\| + \|\bar{x}\|$.

We now define the semantics of first-order queries over trees. Let $\varphi(\bar{x})$ be an FO query over trees. Given a labeled ordered tree $T = (V, E^{\mathrm{c}}, E^{\mathrm{ns}}, \mathrm{lab})$ and a tuple $\bar{a}$ of elements from Nodes, we say that T *satisfies the query* $\varphi(\bar{x})$ *using the values* $\bar{a}$, denoted by $T \models \varphi(\bar{a})$, if there exists an assignment η for φ over T such that $\eta(\bar{x}) = \bar{a}$ and $(D, \eta) \models \varphi$. We can now define the data model and the output of FO queries over trees. The data model defines the possible

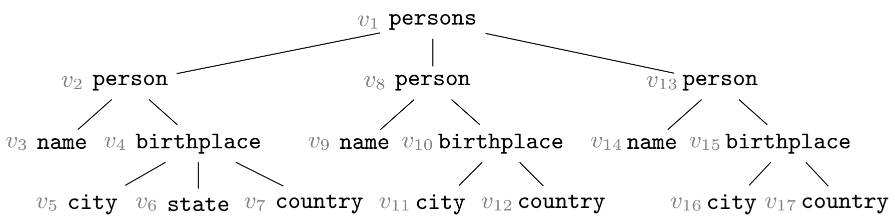


Fig. 57.1: A labeled ordered tree

inputs of the queries. For each data graph in the data model, we define the output of the query.

Definition 57.6: Evaluation of First-Order Queries over Trees

Given a labeled ordered tree $T = (V, E^{\mathrm{c}}, E^{\mathrm{ns}}, \mathrm{lab})$ and an FO query over trees $q = \varphi(x_1, \dots, x_k)$, where $k \geq 0$, the *output* of q on T is defined as the set of tuples

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

The *data model* of FO queries over trees is the set of labeled ordered trees.

We conclude the chapter with a few examples of first-order queries over trees.

Example 57.7: First-Order Queries over Trees

Let T be the tree in Figure 57.1, where the nodes are depicted in grey.

- Let q_1 be the query $\varphi_1(x, y)$, where φ_1 is the formula (57.1). Then

$$q_1(T) = \{(v_1, v_3), (v_8, v_9), (v_{13}, v_{14})\} \;.$$

- Let q_2 be the query $\varphi_2(x)$, where φ_2 is the formula (57.2). Then

$$q_2(T) = \{v_8, v_9\} \;.$$

- Let q_3 be the query $\varphi_3(x)$, where φ_3 is the formula (57.3). Then

$$q_3(T) = \{v_1, v_4, v_7, v_{10}, v_{12}, v_{13}, v_{15}, v_{17}\} \;.$$

- Let q_4 be the query $\varphi_4(x)$, where φ_4 is the formula (57.4). Then

$$q_4(T) = \{v_1\} \;.$$

- Let q_5 be the query $\varphi_5(y, x)$, where φ_5 is the formula (57.5). Then

$$\begin{aligned} q_5(T) = \{&(v_2, v_1), (v_3, v_2), (v_4, v_2), (v_5, v_4), (v_6, v_4), (v_7, v_4), \\ &(v_8, v_1), (v_9, v_8), (v_{10}, v_8), (v_{11}, v_{10}), \\ &(v_{12}, v_{10}), (v_{13}, v_1), (v_{14}, v_{13}), (v_{15}, v_{13}), (v_{16}, v_{15}), (v_{17}, v_{15})\} \end{aligned}$$

XPath

XPath is a powerful language that is designed for navigation and node-selection in tree-structured data. In this chapter, we offer a principled view on XPath, which means that we describe and formally define an elegant and mathematically clean language for navigating through labeled ordered trees. We will point out differences with the official XPath standard whenever relevant.

XPath Axes

XPath uses so-called *axes* as primitive operations for navigating in trees. We use the following axes in this chapter:

```
self    descendant            next-sibling        following
child   descendant-or-self    following-sibling   preceding
parent  ancestor              previous-sibling
        ancestor-or-self      preceding-sibling
```

We note that the XPath standard does not consider the axes `next-sibling` and `previous-sibling`, but we consider them here for completeness. We now explain the meaning of each axis x by associating it to a binary relation R_x over Nodes. To this end, if R and R' are two binary relations, then

- we denote by $R \circ R'$ the *composition* of R and R', which is defined as $\{(a,c) \mid (a,b) \in R, (b,c) \in R'\}$,
- we denote by R^+ the *transitive closure* of R, which is defined as $\{(a_1, a_n) \mid \exists a_2, \ldots, a_{n-1}$ such that $(a_i, a_{i+1}) \in R$ for every $i \in [n-1]\}$,
- we denote by R^* the *transitive reflexive closure* of R, which is defined as $R^+ \cup \{(a,a) \mid \exists b$ such that $(a,b) \in R$ or $(b,a) \in R\}$, and
- we denote by R^{-1} the *inverse* of R, which is defined as $\{(b,a) \mid (a,b) \in R\}$.

Now consider a labeled ordered tree $T = (V, E^{\mathrm{c}}, E^{\mathrm{ns}}, \mathrm{lab})$. Then, we define

$$R_{\texttt{self}} = \{(u,u) \mid u \in V\},\ R_{\texttt{child}} = E^{\mathrm{c}},\ \text{and } R_{\texttt{parent}} = (R_{\texttt{child}})^{-1}\ .$$

Furthermore, we define

$$\begin{aligned}
R_{\texttt{descendant}} &= (R_{\texttt{child}})^{+}\ , & R_{\texttt{descendant-or-self}} &= (R_{\texttt{child}})^{*}\ ,\\
R_{\texttt{ancestor}} &= (R_{\texttt{parent}})^{+}\ , & R_{\texttt{ancestor-or-self}} &= (R_{\texttt{parent}})^{*}\ ,\\
R_{\texttt{next-sibling}} &= E^{\mathrm{ns}}\ , & R_{\texttt{following-sibling}} &= (R_{\texttt{next-sibling}})^{+}\ ,\\
R_{\texttt{previous-sibling}} &= (R_{\texttt{next-sibling}})^{-1}\ , & R_{\texttt{preceding-sibling}} &= (R_{\texttt{previous-sibling}})^{+}\ ,
\end{aligned}$$

$$\begin{aligned}
R_{\texttt{following}} &= R_{\texttt{ancestor-or-self}} \circ R_{\texttt{following-sibling}} \circ R_{\texttt{descendant-or-self}}\ , \text{ and}\\
R_{\texttt{preceding}} &= (R_{\texttt{following}})^{-1}\ .
\end{aligned}$$

Core XPath and Conditional XPath

We are now ready to define the syntax of *Core XPath* and *Conditional XPath*, which are two languages that have received considerable attention in the research literature.

Definition 58.1: Syntax of Core- and Conditional XPath

The syntax of Core XPath expressions is defined by the grammar

$$\begin{aligned}
\texttt{pe} &::= \texttt{step} \mid (\texttt{pe}/\texttt{pe}) \mid (\texttt{pe} \cup \texttt{pe})\\
\texttt{step} &::= \texttt{axis} \mid \texttt{step}[\texttt{ne}]\\
\texttt{ne} &::= ?\texttt{pe} \mid (\texttt{lab} = a) \mid (\texttt{lab} = *) \mid (\texttt{ne} \wedge \texttt{ne}) \mid (\texttt{ne} \vee \texttt{ne}) \mid \neg\texttt{ne}
\end{aligned}$$

Here, `axis` stands for one of the aforementioned axes and a is a label from Lab. The non-terminal `pe` defines the syntax of Core XPath *path expressions* and `ne` defines the syntax of Core XPath *node expressions*.

The syntax of *Conditional XPath expressions* is obtained from the above definition by replacing the rule for `step` with

$$\texttt{step} ::= \texttt{axis} \mid \texttt{step}[\texttt{ne}] \mid (\texttt{step}[\texttt{ne}])^{*}$$

In the grammar for Conditional XPath expressions, the non-terminal `pe` defines the syntax of Conditional XPath *path expressions* and `ne` defines the syntax of Conditional XPath *node expressions*.

In the remainder of the chapter, we abbreviate Core XPath with CoreXPath and Conditional XPath with CondXPath. We now define the semantics of these languages.

Definition 58.2: Semantics of Core- and Conditional XPath

Let $T = (V, E^{\mathrm{c}}, E^{\mathrm{ns}}, \mathrm{lab})$ be a labeled ordered tree. We define the semantics of CoreXPath and CondXPath expressions on T using a mutual induction that defines the semantics of path expressions and node expressions simultaneously. For every path expression e_{p}, its semantics $[\![e_{\mathsf{p}}]\!]$ is a subset of V^2 and, for every node expression e_{n}, its semantics $[\![e_{\mathsf{n}}]\!]$ is a subset of V. More precisely, we have

$$\begin{aligned}
[\![\texttt{axis}]\!]_T &:= R_{\texttt{axis}} \\
[\![\texttt{step}[e_{\mathsf{n}}]]\!]_T &:= \{(u,v) \in [\![\texttt{step}]\!]_T \mid v \in [\![e_{\mathsf{n}}]\!]_T\} \\
[\![(\texttt{step}[e_{\mathsf{n}}])^*]\!]_T &:= [\![\texttt{step}[e_{\mathsf{n}}]]\!]_T^* \\
[\![(e_{\mathsf{p}}/e'_{\mathsf{p}})]\!]_T &:= [\![e_{\mathsf{p}}]\!]_T \circ [\![e'_{\mathsf{p}}]\!]_T \\
[\![(e_{\mathsf{p}} \cup e'_{\mathsf{p}})]\!]_T &:= [\![e_{\mathsf{p}}]\!]_T \cup [\![e'_{\mathsf{p}}]\!]_T
\end{aligned}$$

and

$$\begin{aligned}
[\![(\texttt{lab} = a)]\!]_T &:= \{v \mid \mathrm{lab}(v) = a\} \\
[\![(\texttt{lab} = *)]\!]_T &:= V \\
[\![?e_{\mathsf{p}}]\!]_T &:= \{u \mid \exists v \text{ with } (u,v) \in [\![e_{\mathsf{p}}]\!]_T\} \\
[\![(e_{\mathsf{n}} \wedge e'_{\mathsf{n}})]\!]_T &:= [\![e_{\mathsf{n}}]\!]_T \cap [\![e'_{\mathsf{n}}]\!]_T \\
[\![(e_{\mathsf{n}} \vee e'_{\mathsf{n}})]\!]_T &:= [\![e_{\mathsf{n}}]\!]_T \cup [\![e'_{\mathsf{n}}]\!]_T \\
[\![\neg e_{\mathsf{n}}]\!]_T &:= V - [\![e_{\mathsf{n}}]\!]_T
\end{aligned}$$

We note that subexpressions of the form $(\texttt{lab} = *)$, i.e., wildcard tests, are syntactic sugar, since they can be written as $((\texttt{lab} = a) \vee \neg(\texttt{lab} = a))$ for an arbitrary $a \in \mathsf{Lab}$. Our main reasons for having $(\texttt{lab} = *)$ in our definition are that the XPath standard has this construct and it allows us to have a clean correspondence between XPath and tree pattern queries in Chapter 61. On the other hand, since it is syntactic sugar, we do not need to consider the case $(\texttt{lab} = *)$ in several proofs.

Example 58.3

Consider the labeled ordered tree in Figure 57.1. The CoreXPath path expression

$$(\texttt{self}[(\texttt{lab} = \texttt{person})]/\texttt{child}[(\texttt{lab} = \texttt{name})]) \tag{58.1}$$

selects pairs of nodes (x, y) where x is labeled 'person', y is labeled 'name', and y is a child of x.

The CoreXPath node expression

$$?\big(\texttt{descendant}[((\texttt{lab} = \texttt{person}) \wedge \neg?\texttt{child}[(\mathrm{lab} = \texttt{birthplace}))]]/ \\ \texttt{child}[(\texttt{lab} = \texttt{name})]\big)$$

selects the nodes x that are labeled ‘person’ and have a child with label ‘name’, but do not have a child with label ‘birthplace’.

The CondXPath path expression

$$(\texttt{next-sibling}[(\texttt{lab} = \texttt{person})])^* \tag{58.2}$$

selects node pairs (x, y) such that x and y are siblings with label ‘person’ and every sibling between them also has label ‘person’.

Notice that our definition of the semantics of node and path expressions allows us to interpret them as queries over the set $\mathcal{T}$ of labeled ordered trees. Indeed, we can associate to each node expression e_n a query q_{e_n} and to each path expression e_p a query q_{e_p} such that, on each labeled ordered tree T,

- the *output of* q_{e_n} *on* T is the set of nodes

  $$q_{e_\mathsf{n}}(T) = [\![e_\mathsf{n}]\!]_T$$

  and
- the *output of* q_{e_p} *on* T is the set of node pairs

  $$q_{e_\mathsf{p}}(T) = [\![e_\mathsf{p}]\!]_T \ .$$

From now on, we will therefore also treat e_n and e_p as unary and binary queries over $\mathcal{T}$, respectively.

Complexity of Evaluation

We show that unary CoreXPath and CondXPath queries can be evaluated in linear time combined complexity.

Theorem 58.4

Let T be a labeled ordered tree and q be a unary CoreXPath or CondXPath query. Then we can compute $q(T)$ in time $O(\|q\|\|T\|)$.

Proof. Let $T = (V, E^\mathrm{c}, E^\mathrm{ns}, \mathrm{lab})$ be a labeled ordered tree. We assume without loss of generality that $V = [k]$, where $k = |V|$. Since q is a unary query, we have that $q = e_\mathsf{n}$ for some node expression e_n. Since node and path expressions are defined by mutual induction, we will prove the following by induction on subexpressions e of q:

(a) if $e = e_\mathsf{n}$ for some node expression e_n, then the set $[\![e_\mathsf{n}]\!]_T \subseteq V$ can be computed in time $O(\|e_\mathsf{n}\|\|T\|)$ and

(b) if $S \subseteq V$ and $e = e_{\mathsf{p}}$ for some path expression e_{p}, then the set $[\![e_{\mathsf{p}}]\!]_{T,S} := \{u \mid \exists v \in S \text{ with } (u,v) \in [\![e_{\mathsf{p}}]\!]_T\} \subseteq V$ can be computed in time $O(\|e_{\mathsf{p}}\|\|T\|)$.

Notice that the theorem immediately follows from the first bullet. The algorithm recursively computes and stores the sets $[\![e_{\mathsf{n}}]\!]_T$ and $[\![e_{\mathsf{p}}]\!]_{T,S}$, for a relevant set of nodes $S \subseteq V$, for every subexpression e_{n} and e_{p} of q. Each such set can be stored as an array of $|V|$ bits.

Let e be a subexpression of q. We first consider case (a) where e is a node expression. The first base case is when e is of the form $(\mathtt{lab} = a)$. Since we can indeed compute the set of a-labeled nodes in T in time $O(\|T\|)$, the induction hypothesis holds. The second base case, where e is of the form $(\mathtt{lab} = \mathtt{*})$ is analogous.

Moving to the induction, if e is one of $(e_{\mathsf{n}} \wedge e'_{\mathsf{n}})$ or $(e_{\mathsf{n}} \vee e'_{\mathsf{n}})$, then $[\![e_{\mathsf{n}}]\!]_T$ can be computed by iterating over every node v of T and testing if $v \in [\![e_{\mathsf{n}}]\!]_T \cap [\![e'_{\mathsf{n}}]\!]_T$. The total time for doing this, including the computation of $[\![e_{\mathsf{n}}]\!]_T$ and $[\![e'_{\mathsf{n}}]\!]_T$ is $2|V| + O(\|e_{\mathsf{n}}\|\|T\|) + O(\|e'_{\mathsf{n}}\|\|T\|)$, which is in $O(\|e\|\|T\|)$. The proof where $e = \neg e_{\mathsf{n}}$ is analogous. The final case, where $e = ?e_{\mathsf{p}}$, is immediate from the induction hypothesis (b), taking $S = V$.

We now consider case (b). To this end, let $S \subseteq V$ and e be a path expression. If $e = \mathtt{axis}$, then $[\![e]\!]_{T,S}$ can be computed for each possibility of $\mathtt{axis}$ in time $O(\|T\|)$. E.g., when $\mathtt{axis} = \mathtt{descendant}$, then $[\![e]\!]_{T,S}$ is the set of nodes $\{u \mid \exists v \in S \text{ with } (u,v) \in R_{\mathtt{descendant}}\}$, which are the ancestors of nodes in S. (We leave the other cases as an exercise.) If $e = (e_{\mathsf{p}}/e'_{\mathsf{p}})$, then we first compute $S' = [\![e'_{\mathsf{p}}]\!]_{T,S}$ and then $[\![e_{\mathsf{p}}]\!]_{T,S'}$. For the case $e = \mathtt{step}[e_{\mathsf{n}}]$, we use the induction hypothesis to compute $S' = [\![e_{\mathsf{n}}]\!]_T$ and then $[\![\mathtt{step}]\!]_{T,S'}$. The final case $e = (e_{\mathsf{p}} \cup e'_{\mathsf{p}})$ is also immediate from the induction, by computing $[\![e]\!]_{T,S}$ as $[\![e_{\mathsf{p}}]\!]_{T,S} \cup [\![e'_{\mathsf{p}}]\!]_{T,S}$. This concludes the proof for CoreXPath.

In the case of CondXPath, the only extra case we need to deal with is $e = (\mathtt{step}[e_{\mathsf{n}}])^*$. According to the definition of CondXPath, e can either be of the form $(\mathtt{axis}[e_{\mathsf{n}}])^*$, $(\mathtt{step}'[e'_{\mathsf{n}}][e_{\mathsf{n}}])^*$, or $((\mathtt{step}'[e'_{\mathsf{n}}])^*[e_{\mathsf{n}}])^*$. We will show how $[\![e]\!]_{T,S}$ can be computed in time $O(\|q\|\|T\|)$ in all these cases.

In the first case, if $[\![\mathtt{axis}]\!]_T$ is a transitive relation (e.g., for $\mathtt{axis} = \mathtt{descendant}$), then e is equivalent to $\mathtt{self} \cup \mathtt{axis}[e_{\mathsf{n}}]$, which we have already solved. If $\mathtt{axis} = \mathtt{child}$, then we need to compute all ancestors u of nodes in S such that all nodes on the path from u to S are in $[\![e_{\mathsf{n}}]\!]$, which can obviously be done in the required time. The other cases where $[\![\mathtt{axis}]\!]_T$ is not transitive are analogous.

In the second case, e is of the form $(\mathtt{step}'[e'_{\mathsf{n}}][e_{\mathsf{n}}])^*$ which is equivalent to $(\mathtt{step}'[e'_{\mathsf{n}} \wedge e_{\mathsf{n}}])^*$ and therefore reduces to the first or third case.

In the third case, e is equivalent to $\mathtt{self} \cup (\mathtt{step}'[e'_{\mathsf{n}}])^*[e_{\mathsf{n}}]$, which also reduces to cases that we already dealt with (union and $\mathtt{step}[e_{\mathsf{n}}]$). This concludes the proof. □

59

Expressiveness of XPath

In this chapter we compare the expressiveness of CoreXPath and CondXPath queries to first-order queries over trees (Definition 57.5). In a nutshell, we will see that unary CoreXPath queries are equally expressive as the unary FO queries over trees that can be defined using formulae that use two variables. CondXPath on the other hand can express precisely the unary and binary FO queries over trees. In this chapter we present some of the main ideas behind these correspondences. We first focus on unary queries and discuss binary queries at the end of the chapter.

Unary CoreXPath and FO^2

For our first result, we consider the *two-variable fragment* of FO over trees, which is the set of all FO formulae over trees that use at most two variables (repetitions are allowed). We denote this subset of formulae as FO^2. We define *FO^2 queries over trees* analogously to FO queries over trees. We illustrate such a query in the next example.

Example 59.1: FO^2 Query over Trees

Consider the query $\varphi(x)$, where φ is the formula

$$\exists y \left(E^{\mathrm{ns}}(x,y) \wedge \big(\exists x\, E^{\mathrm{ns}}(y,x) \wedge \exists y\, E^{\mathrm{ns}}(x,y)\big)\right).$$

This query selects all nodes x that have at least three siblings to the right. Notice that the existence of the second and third sibling to the right is expressed by re-using the variables x and y respectively.

It turns out that the unary CoreXPath queries are precisely the same as the unary FO^2 queries over trees.

Theorem 59.2

Unary CoreXPath is equally expressive as unary FO^2 over trees.

Proof (Sketch). For simplicity, we will only prove the result in *one dimension*, that is, we only consider the CoreXPath queries for which the set of axes is restricted to `self`, `next-sibling`, `previous-sibling`, `following-sibling`, and `preceding-sibling`. We will prove that these queries are equally expressive as those FO^2 queries in which formulae use the predicates $\mathrm{Lab}_a(x)$, $E^{\mathrm{ns}}(x_1, x_2)$, and $E^{\mathrm{fs}}(x_1, x_2)$. The general result can be proved using the same technique.

For the translation from CoreXPath to FO^2, we only need to show that CoreXPath queries defined by node expressions can be translated to FO^2 queries. To this end, we will define

(a) for every CoreXPath node expression e an FO^2 query $\varphi_e(x)$ that is equivalent to the query e and
(b) for every CoreXPath path expression e an FO^2 query $\varphi_e(x)$ that is equivalent to the query $?e$.

If e is a node expression, the definition of φ_e is as follows:

- If e is of the form $(\texttt{lab} = a)$, then $\varphi_e = \mathrm{Lab}_a(x)$.
- If e is of the form $(e_1 \wedge e_2)$, $(e_1 \vee e_2)$, or $\neg e_1$, then $\varphi_e = (\varphi_{e_1} \wedge \varphi_{e_2})$, $\varphi_e = (\varphi_{e_1} \vee \varphi_{e_2})$, or $\varphi_e = (\neg\varphi_{e_1})$, respectively.

If e is a path expression, the definition of φ_e is as follows:

- If e is of the form `self`, `next-sibling` or `preceding-sibling`, then $\varphi_e = x$, $\varphi_e = \exists y\ E^{\mathrm{ns}}(x, y)$ or $\varphi_e(x) = \exists y\ E^{\mathrm{ns}}(y, x)$, respectively.
- If e is of the form `following-sibling` or `preceding-sibling`, then $\varphi_e = \exists y\ E^{\mathrm{fs}}(x, y)$ or $\varphi_e = \exists y\ E^{\mathrm{fs}}(y, x)$, respectively.
- If e is of the form (e_1/e_2), then $\varphi_e = \varphi_{e_1} \wedge (\varphi_{e_2}[x/y, y/x])$.
- If e is of the form $(e_1 \cup e_2)$, then $\varphi_e = \varphi_{e_1} \vee \varphi_{e_2}$.
- If e is of the form $e_1[e_2]$, where e_2 is a node expression, then $\varphi_e = \varphi_{e_1} \wedge \varphi_{e_2}[x/y, y/x]$.

Recall that $\varphi_{e_2}[x/y, y/x]$ is the formula obtained from φ_{e_2} by swapping x and y, that is, simultaneously replacing every occurrence of x with y and vice versa. We leave the correctness of the translation as an exercise.

Conversely, let $q = \varphi(x)$ be a unary FO^2 query. We show a recursive translation procedure to turn $\varphi(x)$ into an equivalent CoreXPath node expression e_φ. We can assume without loss of generality that $\varphi(x)$ only uses the Boolean connectives $\vee$ and $\neg$ and the quantifier $\exists$.

If the formula φ is atomic, that is, of the form $\mathrm{Lab}_a(x)$, then $e_\varphi = (\texttt{lab} = a)$. If φ is of the form $\psi_1 \vee \psi_2$ or $\neg\psi$, then we recursively compute $e_{\psi_1} \vee e_{\psi_2}$ or $\neg e_\psi$, respectively. Now, there is one remaining case, which is that φ is of the form $\exists y\ \psi$. Since $\varphi(x)$ is an FO^2 query, the variable x must be free in φ, which means that either $\mathrm{FV}(\psi) = \{x\}$ or $\mathrm{FV}(\psi) = \{x, y\}$.

In the first case, the formula φ is equivalent to

$$\exists y\ (\psi \wedge (\mathrm{lab}(y) = a \vee \neg\mathrm{lab}(y) = a)),$$

which therefore reduces to the case where $\mathrm{FV}(\psi) = \{x, y\}$. So the only remaining case is $\varphi = \exists y\ \psi$ with $\mathrm{FV}(\psi) = \{x, y\}$. In the remainder of the proof, for a Boolean formula β over variables $\{x_1, \ldots, x_k\}$, and for FO^2 formulae $\psi_1, \ldots, \psi_k$, we write $\beta[\psi_1, \ldots, \psi_k]$ for the formula obtained from β by replacing each x_i with ψ_i. Using this notation, and since ψ only mentions variables x and y, we can write ψ as

$$\beta[\chi_1, \ldots, \chi_r, \xi_1, \ldots, \xi_s, \zeta_1, \ldots, \zeta_t],$$

where

- β is a Boolean formula,
- each χ_i is an atomic FO^2 formula with $\mathrm{FV}(\chi_i) = \{x, y\}$,
- each ξ_i is an atomic or existential FO^2 formula with $\mathrm{FV}(\xi_i) = \{x\}$, and
- each ζ_i is an atomic or existential FO^2 formula with $\mathrm{FV}(\zeta_i) = \{y\}$.

In order to be able to recurse on subformulas of $\varphi(x)$, we have to separate the ξ_i's from the ζ_i's. We first introduce a case distinction on which of the subformulas ξ_i's hold or not and obtain that φ is equivalent to

$$\bigvee_{\bar\gamma \in \{\texttt{true},\texttt{false}\}^s} \left(\bigwedge_{i \in [s]} (\xi_i \leftrightarrow \gamma_i) \wedge \exists y\ \beta(\chi_1, \ldots, \chi_r, \gamma_1, \ldots, \gamma_s, \zeta_1, \ldots, \zeta_t) \right).$$

We proceed with a case distinction on which order relation holds between x and y. These are five mutually exclusive cases, determined by the following formulas, which we call *order types*: $x = y$, $E^{\mathrm{ns}}(x, y)$, $E^{\mathrm{ns}}(y, x)$, $E^{\mathrm{fs}}(x, y) \wedge \neg E^{\mathrm{ns}}(x, y)$, or $E^{\mathrm{fs}}(y, x) \wedge \neg E^{\mathrm{ns}}(y, x)$. When we assume that one of these order types τ is true, each atomic order formula evaluates to either `true` or `false`. In particular, each of the χ_i's evaluates to either `true` or `false`, and we will denote this truth value by χ_i^τ. Taking Υ as the set containing the five order types, we can now obtain that φ is equivalent to

$$\bigvee_{\bar\gamma \in \{\texttt{true},\texttt{false}\}^s} \left(\bigwedge_{i \in [s]} (\xi_i \leftrightarrow \gamma_i) \wedge \bigvee_{\tau \in \Upsilon} \exists y\ \left(\tau \wedge \beta(\chi_1^\tau, \ldots, \chi_r^\tau, \bar\gamma, \bar\zeta)\right) \right).$$

If τ is an order type, $\eta(x)$ an FO^2 query, and e_η an equivalent CoreXPath formula, there is an obvious way to obtain a CoreXPath expression $e\langle\tau, \eta\rangle$, as shown in the following table.

τ	$e\langle\tau,\eta\rangle$
$x = y$	e_η
$E^{\text{ns}}(x,y)$	$\texttt{next-sibling}[e_\eta]$
$E^{\text{ns}}(y,x)$	$\texttt{previous-sibling}[e_\eta]$
$E^{\text{fs}}(x,y) \wedge \neg E^{\text{ns}}(x,y)$	$\texttt{next-sibling/following-sibling}[e_\eta]$
$E^{\text{fs}}(y,x) \wedge \neg E^{\text{ns}}(y,x)$	$\texttt{previous-sibling/preceding-sibling}[e_\eta]$

Let us denote by $\zeta_i[x]$ the formula ζ_i in which we substituted the free occurrences of y with x. Our recursive procedure will then recursively compute e_{ξ_i} for every $i \in [s]$ and $e_{\zeta_i[x]}$ for every $i \in [t]$ and output

$$\bigvee_{\bar{\gamma}\in\{\texttt{true},\texttt{false}\}^s} \left(\bigwedge_{i\in[s]} (e_{\xi_i} \leftrightarrow \gamma_i) \wedge \bigvee_{\tau\in\Upsilon} e\langle\tau, \beta\big(\chi_1^\tau, \ldots, \chi_r^\tau, \bar{\gamma}, e_{\zeta_1[x]}, \ldots, e_{\zeta_t[x]}\big)\rangle \right).$$

This concludes the translation. We leave the proof of its correctness as an exercise. □

Unary CondXPath and FO

We now turn to CondXPath and FO. First, we shed some light on the difference between FO^2 and FO. By definition, every FO^2 query is also an FO query, but the converse does not hold. Indeed, one can show that the unary FO query $\varphi(x)$ with

$$\begin{aligned}\varphi = {} & \texttt{following-sibling}(x,y) \wedge \text{Lab}_a(x) \wedge \text{Lab}_b(y) \\ & \wedge \forall z \,\big((\texttt{following-sibling}(x,z) \wedge \texttt{following-sibling}(z,y)) \\ & \qquad\qquad\qquad\qquad\qquad\qquad\qquad \to \text{Lab}_c(z)\big)\end{aligned}$$

cannot be expressed as an FO^2 query over trees. The query returns nodes x that have a following-sibling y such that x is labeled a, y is labeled b and every node *between* x and y is labeled c. Intuitively, the two-variable fragment cannot express this condition, because one needs one variable for x and y each, and then one lacks the third variable to express the condition between x and y.

Therefore, concerning unary queries, FO over trees is strictly more expressive than CoreXPath and FO^2 over trees. However, it can be shown that the addition of $(\texttt{step}[\texttt{ne}])^*$, which we have in CondXPath, makes the language equally expressive as FO. In fact, this is one of the reasons why such a construct has been added to the XPath standard. We state the following result without proof.

Theorem 59.3

Unary CondXPath is equally expressive as unary FO over trees.

Binary Queries

It is natural to ask if Theorem 59.2 can be generalized to binary queries. It turns out that this is not the case, since the binary CoreXPath query

$$(\texttt{next-sibling}/\texttt{next-sibling})$$

cannot be defined as a binary FO^2 query (Exercise 8.3).

The connection between CondXPath and FO is stronger, though, since the binary FO queries over trees are indeed the same as the binary CondXPath queries.

Theorem 59.4

Binary CondXPath is equally expressive as binary FO over trees.

60

Static Analysis of XPath

Static analysis tasks such as satisfiability, containment, and equivalence for queries are useful for query optimization. Indeed, if a (sub)query is not satisfiable, then the empty result can be returned without looking at the data. In turn, if a query is satisfiable, it makes sense to test if it can be rewritten into an equivalent query that can be more efficiently evaluated on the data. In this chapter we study satisfiability, containment, and equivalence for CondXPath.

Satisfiability

We first investigate Satisfiability. From a complexity theory point of view, satisfiability for CoreXPath and CondXPath are rather complex, since both problems are ExpTime-complete (Exercise 8.5). Here, we present the main argument why the problems are in ExpTime.

Theorem 60.1

CondXPath-Satisfiability is in ExpTime.

Proof (Sketch). We prove the result only for CondXPath expressions that

- do not use unions of path expressions,
- only use $(\texttt{step}[\texttt{ne}])^*$ in the form $(\texttt{child}[\texttt{ne}])^*$, and
- only use the axes `child`, `descendant`, and `descendant-or-self`.

To this end, let q be a CondXPath query that satisfies these conditions. We will first rewrite q into an equivalent query that allows us to reduce the number of cases we need to consider later in the proof. Using De Morgan's law, we can rewrite q such that it only uses $\wedge$ and $\neg$. Using associativity, we can rewrite all subexpressions of the form $((\texttt{pe}/\texttt{pe})/\texttt{pe})$ as $(\texttt{pe}/(\texttt{pe}/\texttt{pe}))$. We can replace subexpressions of the form $\texttt{step}[e_1][e_2]$ by $\texttt{step}[e_1 \wedge e_2]$ until no such expression occurs anymore. Finally, we can rewrite `descendant` into `child/descendant-or-self` and then `descendant-or-self` into $(\texttt{child}[(\texttt{lab} = \texttt{*})])^*$, so that q only uses the `child` axis.

We will prove how we can iteratively compute sets $S_{\mathsf{n}}(T)$ and $S_{\mathsf{p}}(T)$ of subexpressions of q such that T is a tree and

- $S_{\mathsf{n}}(T) = \{e_{\mathsf{n}} \mid e_{\mathsf{n}}$ is a node subexpression of q with $\mathrm{Root}(T) \in [\![e_{\mathsf{n}}]\!]_T\}$ and
- $S_{\mathsf{p}}(T) = \{e_{\mathsf{p}} \mid e_{\mathsf{p}}$ is a path subexpression of q with $\mathrm{Root}(T) \in [\![?e_{\mathsf{p}}]\!]_T\}$.

Let $\mathcal{S}$ be the set of all such sets. Notice that $|\mathcal{S}|$, the number of sets in $\mathcal{S}$, is at most $2^{\|q\|}$.

The computation of $\mathcal{S}$ will be a *fixpoint computation* in a process that iterates over the increasing depth of trees T. More precisely, in a first iteration, we compute $S_{\mathsf{n}}(T)$ and $S_{\mathsf{p}}(T)$ for the empty tree and single-node trees. Later iterations consider trees $T = a(T_1, \ldots, T_n)$ of depth k, where we already computed $S_{\mathsf{n}}(T_i)$ and $S_{\mathsf{p}}(T_i)$ for all $i \in [n]$ in a previous iteration. We stop once we find an iteration that has not found any new set in $\mathcal{S}$. This would mean that, for every tree of T depth k, the set $S_{\mathsf{n}}(T)$ equals the set $S_{\mathsf{n}}(T')$ for some T' of depth smaller than k (similarly for $S_{\mathsf{p}}(T)$).

During the entire proof, we will use a set $\Sigma = \{a \mid (\texttt{lab} = a)$ is a subexpression of $q\} \cup \{\#\}$. (We assume without loss of generality that $(\texttt{lab} = \#)$ is not a subexpression of q. The symbol # plays the role of representing "every label that does not appear in q".) The first iteration of the fixpoint algorithm computes all the sets $S_{\mathsf{n}}(T)$ and $S_{\mathsf{p}}(T)$ for $T = \varepsilon$ and for $T = \sigma$ for every $\sigma \in \Sigma$. According to Theorem 58.4, these sets can be computed in polynomial time.

We now assume that we have computed $S_{\mathsf{n}}(T)$ and $S_{\mathsf{p}}(T)$ for all trees of depth $k-1$, degree at most $\|q\|$, and with labels in Σ. The next iteration now considers every $\sigma \in \Sigma$ and $\ell \leq \|q\|$. We then consider every tree $T = \sigma(T_1, \ldots, T_\ell)$, such that we have computed $S_{\mathsf{n}}(T_i)$ and $S_{\mathsf{p}}(T_i)$ for all $i \in [\ell]$. We will now decide, for every node subexpression e_{n} and every path subexpression e_{p} of q, if $e_{\mathsf{n}} \in S_{\mathsf{n}}(T)$ or not and if $e_{\mathsf{p}} \in S_{\mathsf{p}}(T)$ or not. We make these decisions inductively on the structure of the subexpressions.

- If e_{n} is of the form $(\texttt{lab} = a)$, then we decide $e_{\mathsf{n}} \in S_{\mathsf{n}}(T)$ if and only if $\sigma = a$.
- If e_{n} is of the form $(\texttt{lab} = \texttt{*})$, then we decide $e_{\mathsf{n}} \in S_{\mathsf{n}}(T)$.
- If e_{n} is of the form $(e_1 \wedge e_2)$, then we decide $e_{\mathsf{n}} \in S_{\mathsf{n}}(T)$ if and only if $e_1 \in S_{\mathsf{n}}(T)$ and $e_2 \in S_{\mathsf{n}}(T)$.
- If e_{n} is of the form $\neg e$, then we decide $e_{\mathsf{n}} \in S_{\mathsf{n}}(T)$ if and only if $e \notin S_{\mathsf{n}}(T)$.
- If e_{n} is of the form $?e$ for a path expression e, then we decide $e_{\mathsf{n}} \in S_{\mathsf{n}}(T)$ if and only if $e \in S_{\mathsf{p}}(T)$.

This concludes all cases for node subexpressions. We now turn to path subexpressions.

- If e_{p} is of the form `child`, then we decide that $e_{\mathsf{p}} \in S_{\mathsf{p}}(T)$.
- If e_{p} is of the form $\texttt{child}[e]$, then we decide that $e_{\mathsf{p}} \in S_{\mathsf{p}}(T)$ if and only if $e \in S_{\mathsf{n}}(T_i)$ for some $i \in [\ell]$.
- If e_{p} is of the form $(\texttt{child}[e])^*[f]$, then we decide that $e_{\mathsf{p}} \in S_{\mathsf{p}}(T)$ if and only if $f \in S_{\mathsf{n}}(T)$ or there exists some $i \in [\ell]$ such that $e \in S_{\mathsf{n}}(T_i)$ and $e_{\mathsf{p}} \in S_{\mathsf{p}}(T_i)$.
- If e_{p} is of the form e_1/e_2, then we do a case distinction on the structure of e_1.
  - If e_1 is of the form `child`, then we decide that $e_{\mathsf{p}} \in S_{\mathsf{p}}(T)$ if and only if $e_2 \in S_{\mathsf{p}}(T_i)$ for some $i \in [\ell]$.
  - If e_1 is of the form $(\texttt{child}[e])^*[f]$, then we decide that $e_{\mathsf{p}} \in S_{\mathsf{p}}(T)$ if and only if either (1) $f \in S_{\mathsf{n}}(T)$ and $e_2 \in S_{\mathsf{p}}(T)$ or (2) there exists some $i \in [\ell]$ such that $e \in S_{\mathsf{n}}(T_i)$ and $e_{\mathsf{p}} \in S_{\mathsf{p}}(T_i)$.

This concludes the computation of $\mathcal{S}$.

We argue why $\mathcal{S}$ can be computed in exponential time. This holds because $|\mathcal{S}| \leq 2^{\|q\|}$ at all times of the algorithm. Therefore, we can do at most $|\mathcal{S}| \leq 2^{\|q\|}$ iterations before we reach a fixpoint. Furthermore, in every iteration, we consider at most $|\Sigma| \cdot \|q\|^{\|q\|}$ trees.

We leave the proof of correctness as an exercise. (This requires proving that considering the types of trees that we have considered in this proof, i.e., trees that use labels from Σ and and have degree at moes $\|q\|$ is sufficient.) $\square$

Since every CoreXPath expression is also a CondXPath expression, we have the following corollary.

Corollary 60.2

CoreXPath-Satisfiability is in ExpTime.

Containment and Equivalence

Since CoreXPath and CondXPath node expressions are closed under the Boolean operations, we can reduce equivalence and containment to satisfiability.

Theorem 60.3

1. CoreXPath-Containment is in ExpTime.
2. CondXPath-Containment is in ExpTime.

Proof (Sketch). For unary queries, we have that e_{n} is contained in e'_{n} if and only if $(e_{\mathsf{n}} \wedge \neg e'_{\mathsf{n}})$ is not satisfiable. Concerning binary queries, there is a construction that reduces the containment problem of binary queries to the one of unary queries (Exercise 8.7).

Since equivalence can be decided by testing containment in both directions, we also have the following.

Corollary 60.4

1. CoreXPath-Equivalence is in ExpTime.
2. CondXPath-Equivalence is in ExpTime.

Tree Pattern Queries

Tree pattern queries are a simple query language that uses the child and descendant relation in trees in order to select tuples of nodes. They are important for querying tree-structured data because of two reasons. The first is that they closely correspond to a natural part of XPath and CoreXPath, namely the fragment that navigates with `child` and `descendant`, and uses conjunction, label tests, wildcards, and filter expressions (denoted $[\cdot]$). The second reason is that this XPath fragment is widely used in practice, which makes tree pattern queries important from a practical point of view.

In this sense, tree pattern queries play a similar fundamental role for tree-structured data as conjunctive queries for relational data. An important difference between tree pattern queries and conjunctive queries though is that tree pattern queries are always acyclic. This is due to the syntax of CoreXPath, which cannot specify cyclic queries.

Definition and Semantics

In order to define tree pattern queries, we first generalize the definition of labeled unordered trees to incorporate wildcards. We say that a connected, node-labeled, directed graph with wildcards $T = (V, E, \mathrm{lab})$ is a *labeled unordered tree with wildcards* if,

- for every node v, there is at most one node u with $(u, v) \in E$ and
- there is exactly one node v (called the *root* of T) without an incoming edge (u, v).

We are now ready to define tree pattern queries.

Definition 61.1: Tree Pattern Query

A *k-ary tree pattern query (TPQ)* is a tuple

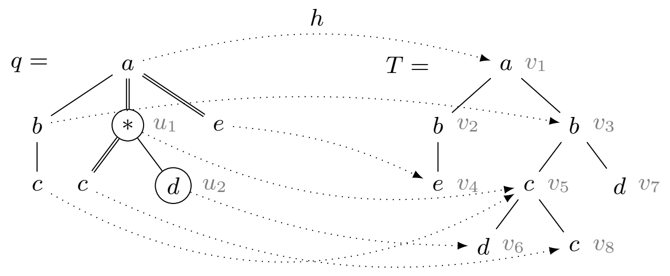


Fig. 61.1: A tree pattern query q (left), a tree T (right), and a tree pattern homomorphism from q to T.

$$q = (V, E^{\mathrm{c}}, E^{\mathrm{d}}, \mathrm{lab}, \bar{v})$$

where

- E^{c} and E^{d} are disjoint,
- $(V, E^{\mathrm{c}} \cup E^{\mathrm{d}}, \mathrm{lab})$ is a labeled unordered tree with wildcards, and
- $\bar{v} = (v_1, \ldots, v_k)$ is a k-tuple of *output nodes*

Similar to first-order queries over trees, we will sometimes denote a tree pattern query q as $q(\bar{v})$ to emphasize that $\bar{v}$ is the tuple of output nodes of q. We refer to E^{c} and E^{d} as the *child edges* and *descendant edges* of q, respectively. If a node is labeled "$*$", we call it a *wildcard node*. When we represent TPQs graphically, we draw child edges using single lines and descendant edges using double lines, see Figure 61.1.

We define $\|q\|$ to be $|V| + k$, i.e., the number of nodes of q plus the arity k of the tuple of output nodes. Notice that we do not require the k output nodes to be distinct. We call q a *k-ary* TPQ. If $k = 0$, we call the query *Boolean*. We sometimes write Boolean TPQs as $q = (V, E^{\mathrm{c}}, E^{\mathrm{d}}, \mathrm{lab})$ to simplify notation. We define BoolTPQ to be the set of Boolean TPQs.

Intuitively, a TPQ can be matched in a tree if there exists a function from the nodes of the TPQ to the nodes of tree that satisfies all constraints imposed by the query. We define this next.

Definition 61.2: Semantics of Tree Pattern Queries

Let $q = (V, E^{\mathrm{c}}, E^{\mathrm{d}}, \mathrm{lab}, \bar{v})$ be a TPQ and $T = (V_T, E^{\mathrm{c}}_T, \mathrm{lab}_T)$ be a labeled unordered tree. We say that a label $a \in \mathsf{Lab}$ *matches* a node $v \in V$ if $\mathrm{lab}(v) = a$ or $\mathrm{lab}(v) = *$. A function $h \colon V \to V_T$ is a *tree pattern homomorphism* from q to T if it fulfills all the following conditions:

- for every $v \in V$, the label $\mathrm{lab}_T(h(v))$ matches v;

- if $(u, v) \in E^{\mathrm{c}}$, then $h(v)$ is a child of $h(u)$ in T; and
- if $(u, v) \in E^{\mathrm{d}}$, then $h(v)$ is a descendant of $h(u)$ in T.

By $q \to T$ we denote that there exists a tree pattern homomorphism from q to T. Conversely, $q \nrightarrow T$ denotes that there does not exist such a tree pattern homomorphism. The *output* of $q(\bar{v})$ on T is defined as

$$q(T) = \{h(\bar{v}) \mid h \text{ is a homomorphism from } q \text{ to } T\} .$$

As such, the *data model* of TPQs is the set of labeled unordered trees.

Notice that we do not require a homomorphism to be injective.

Notice that, even though we define the data model of TPQs to be the set of labeled unordered trees, their semantics can also be defined on ordered trees $T = (V, E^{\mathrm{c}}, E^{\mathrm{ns}}, \mathrm{lab})$. (In fact, the definition is exactly the same.)

Example 61.3: Semantics of Tree Pattern Queries

Figure 61.1 depicts a TPQ q, a tree T, and a tree pattern homomorphism h from q to T. Assume that the tuple of output nodes of q is (u_1, u_2) — we circled these two nodes in the Figure. The homomorphism h produces the answer (v_5, v_6) in T. Another homomorphism can be obtained from h by mapping u_1 and u_2 to nodes v_3 and v_7, respectively. Furthermore, $q(T) = \{(v_3, v_7), (v_5, v_6)\}$. We note that there exist different tree pattern homomorphisms that produce (v_3, v_7) (by mapping the c-labeled sibling of node u_2 in q to node v_5 or to node v_8, respectively).

Relationship to XPath

Unary and binary tree pattern queries naturally correspond to a fragment of CoreXPath that uses the `child` and `descendant` axes, label tests, wildcards, conjunction, and the $[\cdot]$-operator. This fragment of CoreXPath turns out to be very widely used in practice. We will not prove this correspondence formally, but provide an illustrating example.

Example 61.4: XPath versus Tree Pattern Queries

The pattern q in Figure 61.1 can be defined in CoreXPath as

$$\texttt{self}\Big[(\texttt{lab} = *) \wedge ?\texttt{descendant}[(\texttt{lab} = c)]$$
$$\wedge ?\texttt{ancestor}\Big[(\texttt{lab} = a) \wedge ?\texttt{child}\big[(\texttt{lab} = b) \wedge ?\texttt{child}[(\texttt{lab} = c)]\big]$$
$$\wedge ?\texttt{descendant}[(\texttt{lab} = e)]\Big]\Big]/\texttt{child}[(\texttt{lab} = d)]$$

More generally, one can prove the following.

Proposition 61.5

For every unary or binary tree pattern query q, there exists an equivalent CoreXPath query q'. Furthermore,

1. q' can be constructed in linear time and
2. q' only uses `child`, `descendant`, $\wedge$, $(\texttt{lab} = a)$, $(\texttt{lab} = *)$, and $[\cdot]$.

Evaluation

Since TPQs have labeled unordered trees as their associated data model, the problem TPQ-Evaluation is defined as follows.

Problem: TPQ-Evaluation

Input: A TPQ q, a labeled unordered tree $T = (V, E^{\mathrm{c}}, \mathrm{lab})$, and a tuple $\bar{u} \in V^k$

Output: `true` if $\bar{u} \in q(T)$ and `false` otherwise

We will show that this problem can be solved in polynomial time. First we reduce it to a simpler, Boolean variant of the problem. We present a logarithmic space reduction, but it is easy to see that the reduction can also be carried out in linar time on a random-access machine.

Lemma 61.1. *Let q be a k-ary TPQ, T a tree, and $\bar{u}$ a k-tuple of nodes of T. Then there exists a Boolean TPQ q_b and a tree T_b such that $\bar{u} \in q(T)$ if and only if $q_b \to T_b$. Furthermore, q_b and T_b can be computed in logarithmic space.*

Proof. Let $q = (V_q, E^{\mathrm{c}}_q, E^{\mathrm{d}}_q, \mathrm{lab}_q, \bar{v})$ with $\bar{v} = (v_1, \dots, v_k)$, let $T = (V, E^{\mathrm{c}}, \mathrm{lab})$, and let $\bar{u} = (u_1, \dots, u_k)$. Let z and $o_1, \dots, o_k$ be elements of Lab not appearing in T or q. The tree T_b is obtained from T by attaching to each node u_i a new child u'_i with label o_i and to each leaf node $u \notin \{u_1, \dots, u_k\}$ a new child u'

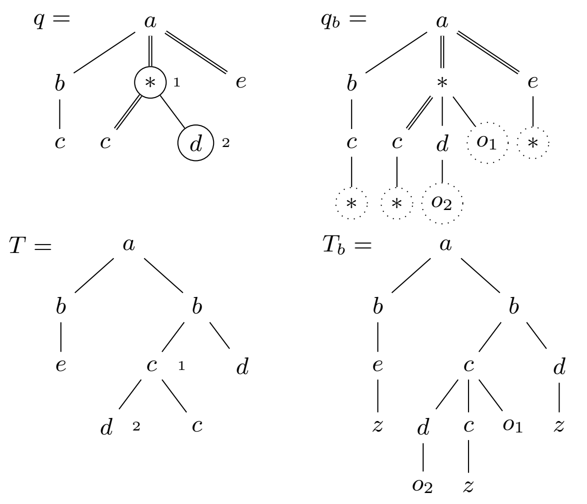


Fig. 61.2: Illustration of the reduction from TPQ-Evaluation to BoolTPQ-Evaluation

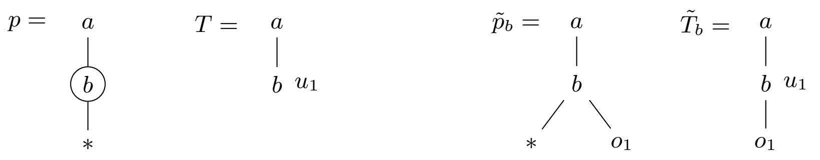


Fig. 61.3: Illustration why the reduction in Lemma 61.1 needs to attach the leaf nodes labeled z and $*$.

with label z. Similarly, pattern q_b is obtained from q by attaching to each node v_i a new child v_i' with label o_i and to each leaf node $v \notin \{v_1, \ldots, v_k\}$ a new child v' with label $*$. It is easy to show that $\bar{u} \in q(T)$ if and only if $q_b \to T_b$. □

Example 61.6: Construction in Lemma 61.1

We illustrate the construction in Lemma 61.1 on an example TPQ q and labeled unordered tree T in Figure 61.2. The construction essentially adds nodes with special labels to the output nodes of q and the nodes in T that are to be tested. Leaf nodes different from these special nodes receive an additional child.

One may wonder why it is necessary to attach new nodes (labeled $*$ and z, respectively) to the leaf nodes of q and T in the proof of Lemma 61.1. The reason is that this is necessary for the correctness of the construction, as we explain next. Consider the pattern q and tree T in Figure 61.3. Furthermore, $\tilde{q}_b$ and $\tilde{T}_b$ are obtained from q and T by the construction as in Lemma 61.1, but *without attaching the new nodes to the leaf nodes*. Assume that u_1 is the b-labeled node in T. Then $u_1 \notin q(T)$ because q requires that its output node is not a child. However, $\tilde{q}_b \rightarrow \tilde{T}_b$, which means that attaching new leaf nodes is indeed necessary.

Theorem 61.7

TPQ-Evaluation is in PTIME.

Proof. By Lemma 61.1, it suffices to show how to test $q \rightarrow T$ in PTIME for a Boolean pattern q. So, assume that $T = (V, E^{\mathrm{c}}, \mathrm{lab})$ and $q = (V_q, E^{\mathrm{c}}_q, E^{\mathrm{d}}_q, \mathrm{lab}_q)$. The idea is that we compute two sets of nodes of q for each node u of T, namely

- $\mathsf{match}(u) = \{v \in V_q \mid q_{|v} \rightarrow T_{|u}\}$ and
- $\mathsf{match}'(u) = \{v \in V_q \mid$ there exists a descendant u' of u such that $q_{|v} \rightarrow T_{|u'}\}$

We compute these sets in a bottom-up fashion. First, for each leaf u of T, we define $\mathsf{match}'(u) = \emptyset$ and $\mathsf{match}(u) = \{v \in V_q \mid v$ is a leaf in q and $\mathrm{lab}(u)$ matches $v\}$.

Now, assume that u is a node in T with children $\{u_1, \ldots, u_n\}$ such that we know all sets $\mathsf{match}(u_i)$ and $\mathsf{match}'(u_i)$. Then, $\mathsf{match}(u)$ is the set of nodes $v \in V_q$ that are matched by $\mathrm{lab}(u)$ and such that, for every edge (v, v') of q, there exists a child u_i of u for which one of the following holds:

- If (v, v') is a child edge, then $v' \in \mathsf{match}(u_i)$.
- If (v, v') is a descendant edge, $v' \in \mathsf{match}(u_i) \cup \mathsf{match}'(u_i)$.

The set $\mathsf{match}'(u)$ is simply the union of all sets $\mathsf{match}(u_i)$ and $\mathsf{match}'(u_i)$. Finally, the algorithm accepts if there exists a node $u \in V$ such that $\mathrm{Root}(q) \in \mathsf{match}(u)$. It is easy to see that the algorithm runs in polynomial time. □

The complexity in Theorem 61.7 can be improved to $O(\|q\| \cdot \|T\|)$ by a direct algorithm that does not use the reduction to Boolean patterns, see Exercise 8.8.

62

Tree Pattern Query Containment and Equivalence

Recall that the main static analysis problems for query languages are satisfiability, containment, equivalence, and minimization. We will study these problems in the present and the next chapter.

First, observe that satisfiability for tree pattern queries is trivial, since every tree pattern query is satisfiable by the tree obtained from the pattern by

1. replacing each descendant edge with a child edge and
2. replacing each wildcard by an arbitrary label.

Therefore, the first nontrivial problems we study are containment and equivalence.

Containment and Equivalence

Recall that tree pattern query q_1 is *contained* in tree pattern query q_2 (denoted $q_1 \subseteq q_2$), if $q_1(T) \subseteq q_2(T)$ for every labeled unordered tree T. Tree pattern query q_1 is *equivalent* to q_2 (denoted $q_1 \equiv q_2$), if $q_1 \subseteq q_2$ and $q_2 \subseteq q_1$.

We first prove that it suffices to focus on Boolean TPQs to study the complexity of containment and equivalence.

Lemma 62.1. *Let q_1 and q_2 be TPQs. Then there exist Boolean TPQs q_1^b and q_2^b such that $q_1 \subseteq q_2$ if and only if $q_1^b \subseteq q_2^b$. Furthermore, q_1^b and q_2^b can be computed in logarithmic space.*

Proof. The construction of q_i^b from q_i is the same as the construction of q_b from q in the proof of Lemma 61.1. □

Next, we prove that it suffices to focus on the containment problem, i.e., for Boolean TPQs, containment and equivalence are interreducible.

Assume that q_1 and q_2 are Boolean TPQs. On the one hand, $q_1 \equiv q_2$ if and only if $q_1 \subseteq q_2$ and $q_2 \subseteq q_1$. On the other hand, suppose that we want to test if $q_1 \subseteq q_2$. Let q_1' be the TPQ obtained from q_1 by adding a new root labeled $*$ and connecting it with a child edge to the root of q_1. Likewise, let $q_1 \cap q_2$ be the TPQ obtained from q_1 and q_2 by adding a new root labeled $*$ and attaching two child edges, one to the root of q_1 one to the root of q_2, respectively. Then, we have that $q_1 \subseteq q_2$ if and only if $q_1' \equiv q_1 \cap q_2$. Therefore, and by Lemma 62.1, we have now obtained the following.

Proposition 62.1

There exists logarithmic-space reductions between all of the following problems:

- TPQ-Containment
- TPQ-Equivalence
- BoolTPQ-Containment
- BoolTPQ-Equivalence

For this reason, our focus for studying containment and equivalence of tree pattern queries will be on the BoolTPQ-Containment problem.

Containment Via Homomorphisms

We proved in Theorem 15.4, the Homomorphism Theorem, that a conjunctive query q_1 is contained in q_2 if and only if there exists a homomorphism from q_2 to q_1. We now prove a similar result for TPQs. This time, however, the equivalence between containment and the existence of a homomorphism only holds for a fragment of TPQs. To this end, let $\text{BoolTPQ}_{\mathsf{Lab}}$ be the set of Boolean TPQs without wildcard nodes, i.e., every node has some label from Lab.

We generalize the definition of tree pattern homomorphisms (Definition 61.2) to include functions from tree pattern queries to tree pattern queries.

Definition 62.2: Homomorphism between Tree Pattern Queries

Let $q_1 = (V_1, E_1^{\mathrm{c}}, E_1^{\mathrm{d}}, \mathrm{lab}_1)$ and $q_2 = (V_2, E_2^{\mathrm{c}}, E_2^{\mathrm{d}}, \mathrm{lab}_2)$ be Boolean TPQs. A function $h : V_1 \to V_2$ is a *tree pattern homomorphism from* q_1 *to* q_2, if

- every $v \in V_{q_1}$ matches $\mathrm{lab}_2(h(v))$,
- if $(u, v) \in E_1^{\mathrm{c}}$ then $h(u, v) \in E_2^{\mathrm{c}}$, and
- if $(u, v) \in E_1^{\mathrm{d}}$ then $h(u)$ is an ancestor of $h(v)$ in q_2.

We write $q_1 \to q_2$ if there exists a tree pattern homomorphism from q_1 to q_2. Similarly, we write $q_1 \nrightarrow q_2$ if no such tree pattern homomorphism exists.

We are now ready to state a result for TPQs that is similar to the Homomorphism Theorem for conjunctive queries.

Theorem 62.3

If q_1 and q_2 are in $\text{BoolTPQ}_{\mathsf{Lab}}$, then

$$q_1 \subseteq q_2 \qquad \Longleftrightarrow \qquad q_2 \to q_1$$

Proof. Assume that h is a tree pattern homomorphism from q_2 to q_1. Then if h_1 is a tree pattern homomorphism from q_1 to a tree T, we have that $h_1 \circ h$ is a tree pattern homomorphism from q_2 to T.

Conversely, assume that $q_1 \subseteq q_2$. Let z be a label from Lab not appearing in q_2. Notice that z exists because Lab is infinite. Let T be the tree obtained from q_1 by replacing each descendant edge (u, v) with two child edges (u, u_{uv}) and (u_{uv}, v), where u_{uv} is a new node, labeled z. Since $q_1(T) = \texttt{true}$ and $q_1 \subseteq q_2$, we also have that $q_2(T) = \texttt{true}$. Therefore, there is a tree pattern homomorphism h from q_2 to T. Since q_2 does not have wildcard nodes, the image of h does not contain any new nodes of the form n_{uv} and, therefore, h is a tree pattern homomorphism from q_2 to q_1. □

We now turn to complexity. The following result is analogous to Theorem 61.7.

Proposition 62.4

If q_1 and q_2 are Boolean TPQs, then it can be tested in PTIME whether $q_1 \to q_2$.

The following corollary is immediate from Theorem 62.3 and Proposition 62.4.

Corollary 62.5

$\mathsf{BoolTPQ}_{\mathsf{Lab}}$-$\mathsf{Containment}$ and $\mathsf{BoolTPQ}_{\mathsf{Lab}}$-$\mathsf{Equivalence}$ are in PTIME.

We note that Corollary 62.5 also holds in the case where the queries are not Boolean, see Exercise 8.9.

We now discuss why we restricted the aforementioned results to tree pattern queries without wildcards. If the queries use wildcard nodes, there are obvious examples that show that the existence of a homomorphism is not necessary for containment, see Figure 62.1(a). Here, the left query is contained

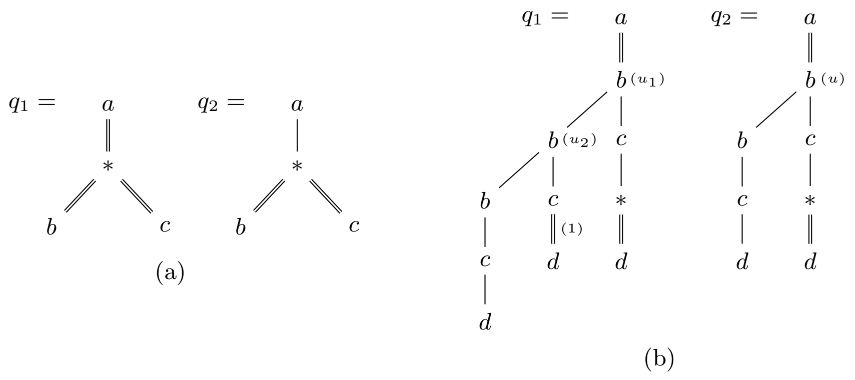


Fig. 62.1: Two examples where $q_1 \subseteq q_2$ even though $q_2 \not\rightarrow q_1$

in the right query (and is even equivalent), even though no homomorphism exists from right to left. Figure 62.1(b) illustrates a more complex example. Here one can see that the left query is contained in the right query by case distinction on the edge marked (1). Assume that $T \models q_1$ with match m. If (1) is mapped to a single edge in T, then node u of q_2 can be mapped to $m(u_1)$. Otherwise, it can be mapped to $m(u_2)$.

Star Extensions and Canonical Trees

Since testing the existence of a homomorphism is not sufficent to test containment of TPQs, the question arises if there exists an alternative method. Here we make a first step towards such a method, by showing that it is sufficient to consider *canonical trees*, which we define next.

Definition 62.6: Star Extensions and Canonical Trees

Let q be a Boolean TPQ.

- A *star extension* of q is a TPQ without descendant edges, obtained from q by replacing each descendant edge (u, v) by a path $uu_1^{uv} \cdots u_k^{uv} v$ where, for every $i \in [k]$ the node u_i^{uv} is new and labeled $*$.
- A tree T is a *canonical tree* of q if it can be obtained from a star extension q^* of q by labeling each wildcard node by some fixed label z that does not appear in q. By $T_{\mathsf{z}}[q^*]$ we denote the canonical tree that was obtained in this manner. Notice that canonical trees T of q always match q, since the identity function on the nodes of q is always a match of q in T.

The importance of canonical trees is captured in the following Lemma, which shows that it suffices to restrict our attention to canonical trees for deciding whether $q_1 \subseteq q_2$.

Lemma 62.2. *If q_1 and q_2 are Boolean TPQs. Then $q_1 \subseteq q_2$ if and only if $q_2(T) = \texttt{true}$ for every canononical tree T of q_1.*

Proof. The direction from left to right is trivial. We prove the other direction by contraposition. Assume that $q_1 \not\subseteq q_2$ and let z be a label that does not occur in q_2. Let T be a (not necessarily canonical) tree such that $q_1(T) = \texttt{true}$ and $q_2(T) = \texttt{false}$. Let h_1 be a homomorphism from q_1 to T. For each descendant edge (u, v) of q_1, let k_{uv} be the number of nodes between $h_1(u)$ and $h_1(v)$ in T. Let q_1^* be the star extension of q_1 where each descendant edge (u, v) is replaced by the path $uu_1^{uv} \cdots u_{k_{uv}}^{uv} v$. Let h^* be a homomorphism from q_1^* to T.

We claim that $q_2(T_{\mathsf{z}}[q_1^*]) = \texttt{false}$. Towards a contradiction, assume that h_2 is a tree pattern homomorphism from q_2 to $T_{\mathsf{z}}[q_1^*]$. Notice that h_2 can only map wildcard nodes of q_2 to z-labeled nodes of $T_{\mathsf{z}}[q_1^*]$, since z does not appear in q_2. But this means that $h = h^* \circ h_2$ is a tree pattern homomorphism[1] of q_2 in T, which contradicts that $q_2(T) = \texttt{false}$. □

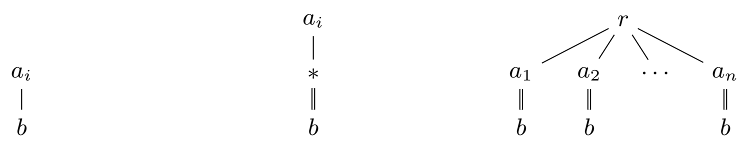
Next, we establish that it is even sufficient to consider canonical trees of polynomial size.

Lemma 62.3. *If q_1 and q_2 are Boolean TPQs such that $q_1 \not\subseteq q_2$, then there exists a tree T with $|T| \leq |q_1|(|q_2| + 1)$ such that $q_1(T) = \texttt{true}$ and $q_2(T) = \texttt{false}$.*

Proof. Let z be a label that does not occur in q_1 or q_2. By Lemma 62.2 we know that there exists a canonical tree $T_{\mathsf{long}} = T_{\mathsf{z}}[q_1^*]$ such that $q_1(T_{\mathsf{long}}) = \texttt{true}$ and $q_2(T_{\mathsf{long}}) = \texttt{false}$.

We will now prove that long paths of z-labeled nodes in T_{long} can be shortened. Assume that T_{long} has a path $\pi = uu_1^{uv} \cdots u_k^{uv} v$ with (u, v) a descendant edge in q_1 and $k > |q_2| + 1$, so $u_1^{uv}, \ldots, u_k^{uv}$ are new nodes in q_1^*. Let T_{short} be obtained from T_{long} by replacing π with $\pi' = uu_1^{uv} \cdots u_{|q_2|+1}^{uv} v$. Then clearly $q_1(T_{\mathsf{short}}) = \texttt{true}$, because it is a canonical tree. We will show that $q_2(T_{\mathsf{short}}) = \texttt{false}$. Assume the contrary. Then there is a tree pattern homomorphism $h : q_2 \to T_{\mathsf{short}}$. By the pigeon hole principle, there is at least one node v_j in $\{v_1^{uv}, \ldots, v_{|q_2|+1}^{uv}\}$ that is not in the image of h. But then h is also a homomorphism from q_2 to the tree T', obtained from T_{short} by replacing π' with $\pi'' = uv_1^{uv} \cdots v_{j-1}^{uv} v_{|q_2|+2}^{uv} \cdots v_k^{uv} v_j^{uv} \cdots v_{|q_2|+1}^{uv}$. However, T' is isomorphic with T_{long}, which would mean that $q_2 \to T_{\mathsf{long}}$ and therefore $q_2(T_{\mathsf{long}}) = \texttt{true}$. This is a contradiction. □

[1] We note that h^* and h_2 can indeed be composed, since $T_{\mathsf{z}}[q_1^*]$ has the same set of nodes as q_1^*.



(a) Encoding $x_i = \texttt{true}$ (b) Encoding $x_i = \texttt{false}$ (c) The TPQ p_A

Fig. 62.2: Encoding `true`, `false`, and truth assignments

The Complexity of TPQ Containment

We are now ready to settle the complexity of TPQ-Equivalence and TPQ-Containment.

Theorem 62.7

TPQ-Containment and TPQ-Equivalence are coNP-complete.

Proof. Lemma 62.3 implies that BoolTPQ-Containment is in coNP, since we can guess T and check deterministically (using Theorem 61.7) that $q_1 \to T$ and $q_2 \not\to T$. Using Proposition 62.1, this shows that TPQ-Containment and TPQ-Equivalence are in coNP.

We now prove that the problem is coNP-hard. In particular, we prove that BoolTPQ-Containment is coNP hard. Hardness for equivalence follows again from Proposition 62.1.

We reduce from the validity problem of 3DNF formulae, which is well known to be coNP-complete. To this end, let φ be a Boolean formula in 3DNF with variables $\{x_1, \dots, x_n\}$. We will construct Boolean TPQs p_1 and p_2 such that $p_1 \subseteq p_2$ if and only if φ is valid. Let

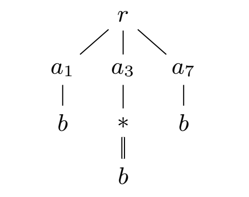
$$\varphi = c_1 \vee \cdots \vee c_k \ ,$$

where c_i is a clause of the form $(\ell_{i,1} \wedge \ell_{i,2} \wedge \ell_{i,3})$ for each $i \in [k]$. Here, each $\ell_{i,j}$ is called a *literal* and is either a variable or the negation thereof. We will first construct TPQs p_A and $p_{C_1}, \dots, p_{C_k}$ such that φ is valid if and only if, for every tree T,

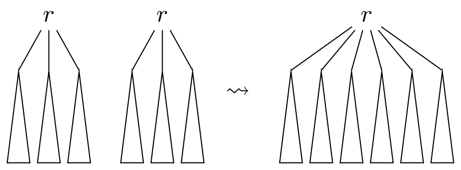
$$p_A \to T \quad \Longrightarrow \quad \exists i \in [k] \text{ such that } p_{C_i} \to T$$

The task of p_A is to "generate all the truth assignments". To this end, we interpret the tree in Figure 62.2a as "$x_i = \texttt{true}$" and every tree that matches the TPQ in Figure 62.2b as "$x_i = \texttt{false}$". As such, every tree that matches the TPQ p_A in Figure 62.2c can be understood as a truth assignment for $\{x_1, \dots, x_n\}$.

We now construct the patterns $p_{C_1}, \dots, p_{C_k}$. We first illustrate how to construct p_{C_i} by example. Assume that $c_i = (x_1 \wedge \neg x_3 \wedge x_7)$. Then p_{C_i} is



(a) Encoding a clause



(b) Merging roots

Fig. 62.3: Constructing clause patterns and all-match patterns

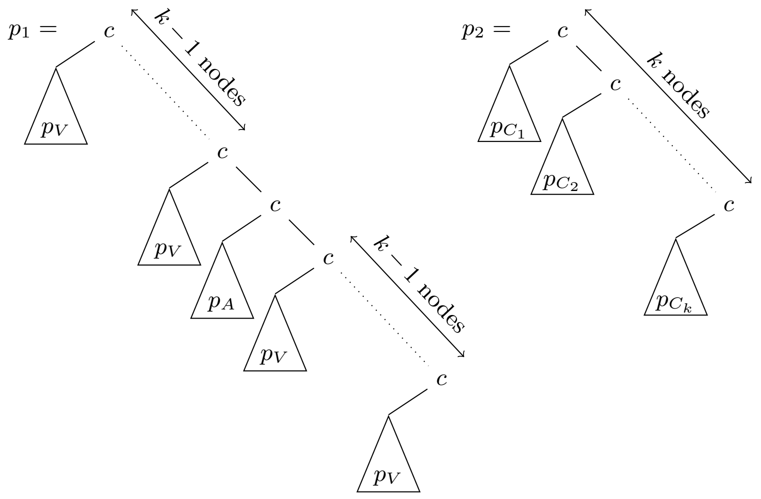


Fig. 62.4: Patterns p_1 and p_2 in Theorem 62.7

the TPQ in Figure 62.3a. More generally, assume that $c_i = (\ell_{i,1} \wedge \ell_{i,2} \wedge \ell_{i,3})$. Let $f(i,j) \in [n]$ be the number such that $x_{f(i,j)}$ is the variable mentioned in $\ell_{i,j}$ for every $j \in [3]$. Let $b_{i,j}$ be a Boolean value indicating whether $\ell_{i,j}$ is a positive or negative literal. That is, $b_{i,j} = \texttt{true}$ if $\ell_{i,j} = x_{i,j}$ and $b_{i,j} = \texttt{false}$ if $\ell_{i,j} = \neg x_{i,j}$. Then, p_{C_i} consists of a root, labeled r, below which we attach the three subpatterns encoding $x_{f(i,j)} = b_{i,j}$, for $j \in [3]$. It is easy to prove that φ is valid if and only if every tree that matches p_A matches at least one p_{C_i} for some $i \in [k]$.

In the next part of the proof, we turn p_A and $p_{C_1}, \ldots, p_{C_k}$ into two patterns p_1 and p_2 such that φ is valid if and only if $p_1 \subseteq p_2$. To this end, let V be the pattern obtained by merging the roots of $p_{C_1}, \ldots, p_{C_k}$ (see Figure 62.3b). We then define p_1 and p_2 as illustrated in Figure 62.4. We leave the proof that φ is valid if and only if $p_1 \subseteq p_2$ as an exercise. $\square$

63

Tree Pattern Query Minimization

Tree pattern query minimization amounts to transforming a given TPQ into an equivalent one that has as few nodes as possible. We introduce minimizations of TPQs in a similar way as we did for conjunctive queries.

Definition 63.1: Minimization of TPQs

Given a TPQ q, a TPQ q' is called a *minimization* of q if it is equivalent to q and has the smallest number of nodes among all the TPQs that are equivalent to q. A TPQ q is *minimal* if it is a minimization of itself.

In Chapter 16 we saw that, in the case of CQs, minimizations can be obtained by removing atoms from the query. We can now ask ourselves if something similar is true for TPQs. More precisely, we ask if TPQs can be minimized by *deleting nodes*. To this end, for a TPQ $q = (V_q, E_q, \text{lab}_q, \overline{v})$ and a leaf $u \in V_q$ that does not appear in $\overline{v}$, denote by $(q - u)$ the query obtained from q by removing u from V_q and every edge of the form (u', u) from E_q. Notice that every subpattern of q that contains the root and output nodes of q can be obtained by repeatedly deleting leaves.

In Algorithm 17, we describe a procedure TrimTPQ, which is similar to the ComputeCore algorithm for CQs, i.e., Algorithm 4 from Chapter 16. It repeatedly removes leaves from the pattern as long as the pattern stays equivalent. In some important cases, it is not very difficult to show that TrimTPQ indeed computes a minimization. We leave the proof as Exercise 8.13.

Theorem 63.2

Let q be a TPQ. Then TrimTPQ(q) computes a minimization of q if either

Algorithm 17 TrimTPQ(q)

```
Input: A tree pattern query $q$
Output: A subpattern $q^*$ of $q$ that is equivalent to $q$
1: $q^* := q$
2: while there is a leaf $u$ of $q^*$ such that $(q^* - u) \equiv q^*$ do
3:     $q^* := (q^* - u)$
   return $q^*$
```

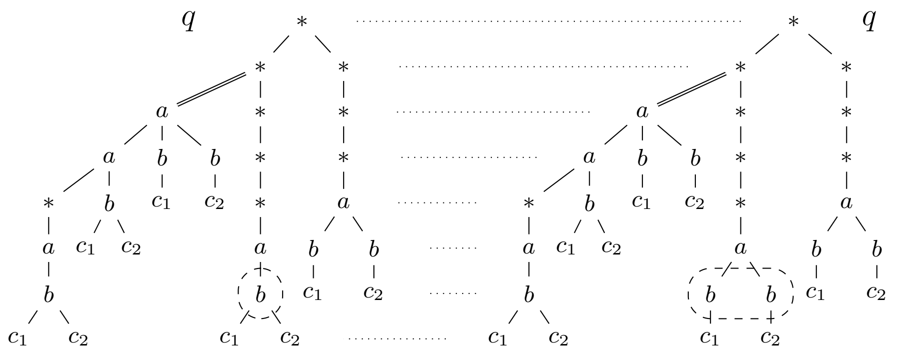


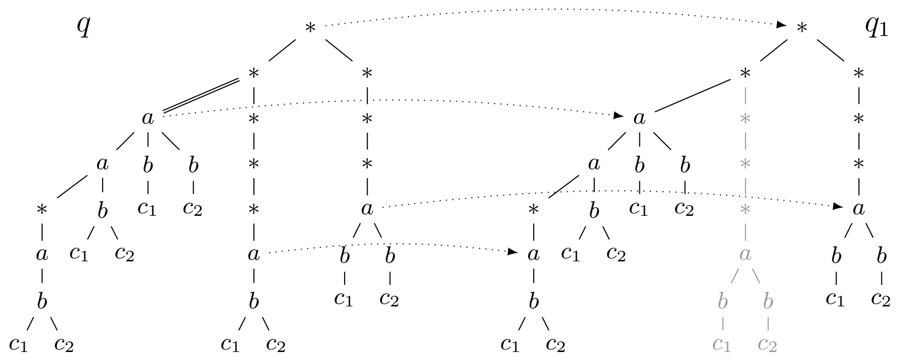
Fig. 63.1: A tree pattern query q that has no equivalent proper subpattern (right) and a tree pattern query q that is equivalent and smaller (left). The only difference between the two patterns is in the dashed lines.

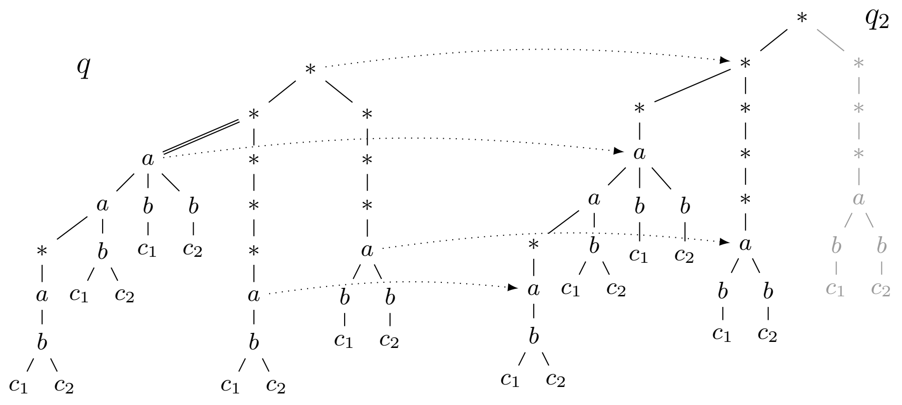
(1) q does not contain descendant edges, or

(2) q does not have wildcard nodes.

Perhaps surprisingly, Theorem 63.2 does not hold for TPQs in general. Figure 63.1 contains on the right a tree pattern query q that has no equivalent proper subpattern, that is, there is no node v with $q \equiv (q - v)$. However, it is not minimal, because the pattern on the left is equivalent and smaller. We prove that q and q are equivalent and leave the proof that q has no equivalent proper subpattern as an exercise (Exercise 8.14).

Consider the tree pattern queries $q_1, \ldots, q_5$, depicted on the right hand sides of Figures 63.2 and 63.3. Observe that q is equivalent to $q_1 \cup q_2 \cup q_3 \cup q_4 \cup q_5$, since the only difference between these patterns and q is that they replace the lowermost descendant edge in q by paths of increasing length, where q_5 has a descendant edge to deal with paths of length at least five. Figures 63.2 and 63.3 then show how homomorphisms from q to $q_1, \ldots, q_5$ can be constructed. This shows that $q \subseteq q$. The inclusion $q \subseteq q$ is easy to see, because there exists a homomorphism from q to q. The homomorphism maps the two nodes in the dashed lines in q to the single such node in q.


(a) Homomorphism from q to q_1


(b) Homomorphism from q to q_2

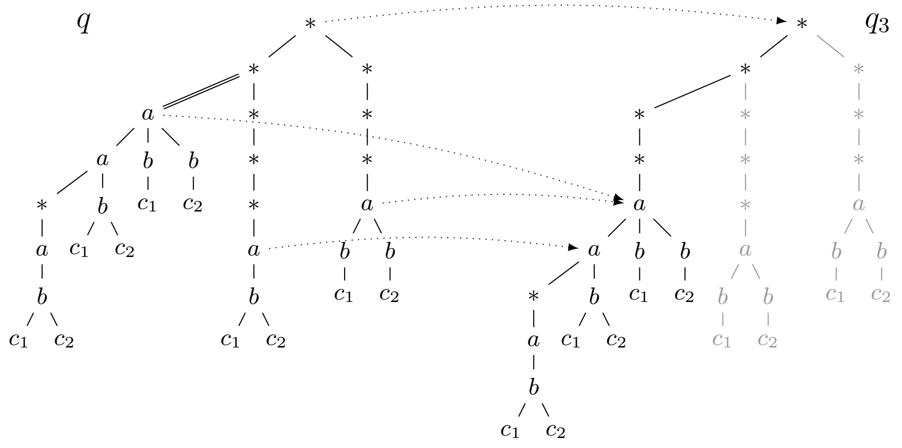
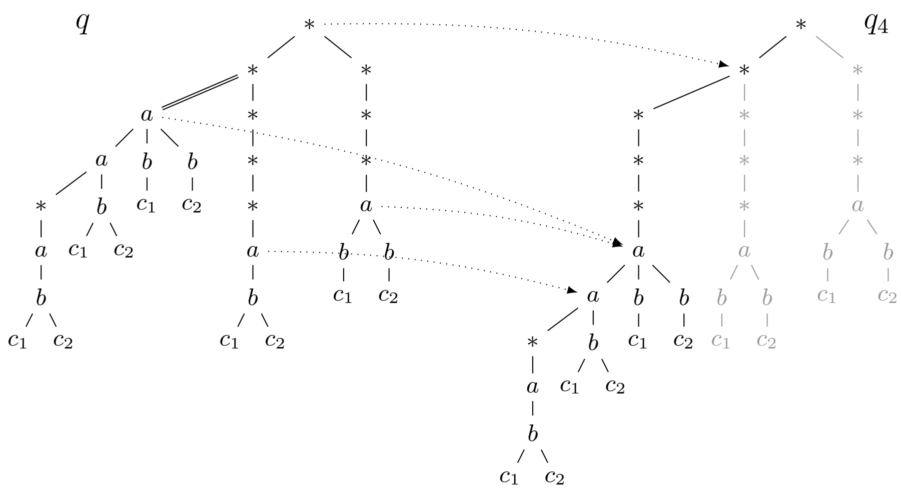
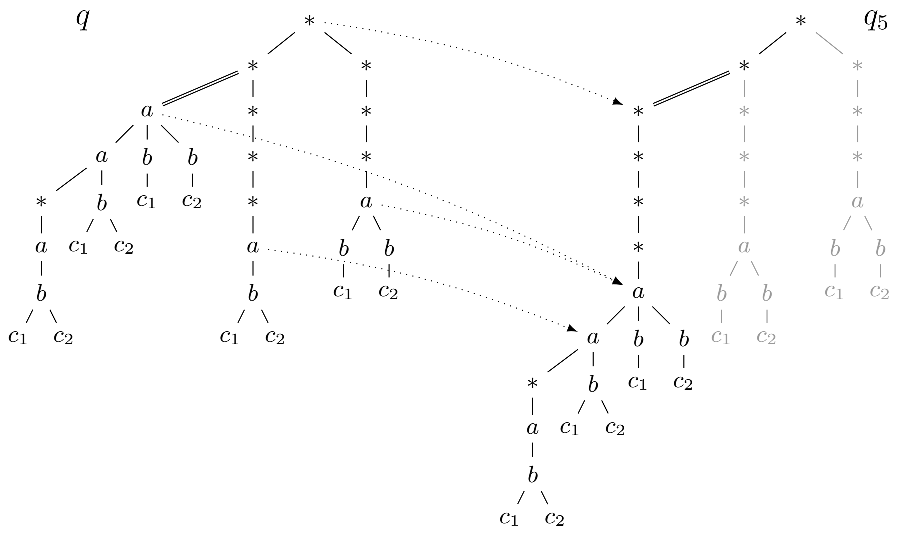

(c) Homomorphism from q to q_3

Fig. 63.2: Showing that patterns q and q in Figure 63.1 are equivalent



(a) Homomorphism from q to q_4



(b) Homomorphism from q to q_5

Fig. 63.3: Showing that patterns q and q in Figure 63.1 are equivalent

We conclude this chapter with a note on the complexity of computing minimizations of TPQs. Minimizations of a TPQ q can be computed by a naive algorithm that iterates through all TPQs that are smaller than q and tests whether they are equivalent, but such an algorithm hardly seems satisfactory. Surprisingly, we do not know an algorithm that is significantly better! From the example in Figure 63.1, we know that deleting nodes is not sufficient for minimizing TPQs since, in this case, nodes also need to be merged. But there are examples that show that deleting and merging nodes still is not sufficient: in some cases, nodes also need to be split. As such minimization is a more complex problem for TPQs than for CQs, for example. Indeed, one can show that, whereas it is coNP-complete to test if a given CQ is minimal (Exercise 2.17), this problem is Π_2^p-complete for TPQs (Exercise 8.15). The latter means that the "obvious algorithm" that tests if, for a given TPQ q, every smaller TPQ is not equivalent to q is worst-case optimal.

Exercises

Exercise 8.1. Let $T = (V, E^{\mathrm{c}}, E^{\mathrm{ns}}, \mathrm{lab})$ be a labeled ordered tree. In the proof of Theorem 58.4, show that, for every `axis` in the set

$$\{\texttt{self}, \texttt{child}, \texttt{parent}, \texttt{descendant}, \texttt{descendant-or-self}, \texttt{ancestor}, \\ \texttt{ancestor-or-self}, \texttt{next-sibling}, \texttt{following-sibling}, \\ \texttt{previous-sibling}, \texttt{preceding-sibling}, \texttt{following}, \texttt{preceding}\}$$

and every set $S \subseteq V$, it is possible to compute $[\![\texttt{axis}]\!]_{T,S}$ in time $O(\|T\|)$.

Exercise 8.2 (*). What is the precise complexity of evaluation for downward XPath (child,descendant,filter,and,or,not)? For which complexity class is the evaluation problem complete under logspace reductions? (Open problem.)

Exercise 8.3. Show that the binary CoreXPath query

$$(\texttt{next-sibling/next-sibling}) \tag{63.1}$$

cannot be defined as a binary FO^2 query.

Hint: It will be convenient to use a pebble characterization for FO^2.

Exercise 8.4. Show that every FO query over trees can be defined as an FO^3 query, that is, an FO query that only uses three variables.

Hint: Show that every binary CondXPath query can be defined in FO^3 over trees.

Hint: Look into the paper Immerman & Kozen. Definability by a bounded number of variables.

Exercise 8.5. Show that CoreXPath-Satisfiability and CondXPath-Satisfiability are EXPTIME-hard.

Hint: Reduce from two-player corridor tiling. For checking vertical constraints, an idea of encoding their violations in unary as in Bjorklund Martens Schwentick MFCS 13 will be useful.

Exercise 8.6. Show that the algorithm in the proof of Theorem 60.1 is correct.

Exercise 8.7. Show that CoreXPath-Containment and CondXPath-Containment for binary queries can be reduced in polynomial time to CoreXPath-Containment and CondXPath-Containment for unary queries, respectively

Hint: The construction is the proof of Theorem 6 in [**Marx-edbt04**]. It consists of rewriting the queries in such a way that some axes are reversed.

Exercise 8.8. We note that the complexity in Theorem 61.7 can be improved to $O(|p||T|)$. *Hint:* Let the input to TPQ Evaluation be p, T, and $(u_1, \ldots, u_k)$. Let $(v_1, \ldots, v_k)$ be the output nodes of p. Then the overall idea of the direct algorithm is the same as in Theorem 61.7, but it treats the output nodes of p differently. More precisely, it only allows an output node v_i to be in a set $\mathsf{match}(u)$ if $u = u_i$.

Exercise 8.9. Prove that the restriction to Boolean queries in Corollary 62.5 can be lifted. *Hint:* Notice that one cannot just apply Proposition 62.1 since the reduction to Boolean queries introduces wildcard nodes. Instead, generalize Definition 62.2 and Theorem 62.3.

Exercise 8.10. Prove that φ is valid if and only if $p_1 \subseteq p_2$ in the proof of Theorem 62.7.

Exercise 8.11. Consider *conjunctive queries over trees*, i.e., conjunctive queries evaluated over tree structures, where the built-in relations are *child* and *descendant*. What is the complexity of their evaluation problem?

Exercise 8.12. Show that the output of ComputeCore(p) (Algorithm 17 on page 512) is not unique up to isomorphism.

Exercise 8.13. Prove Theorem 63.2.

Exercise 8.14. Show that the pattern p in Figure 63.1 has no equivalent proper subpattern.

Exercise 8.15. (a) Let TPQ-Minimization be the problem where, given a Boolean TPQ q and integer $k \in \mathbb{N}$, the question is if there exists a TPQ q' such that $q' \equiv q$ and $|q'| \leq k$. Prove that TPQ-Minimization is Σ_2^p-complete.

(b) Let TPQ-Minimality be the problem where, given a Boolean TPQ q, the question is to answer `true` if q is minimal and `false` otherwise. Prove that TPQ-Minimality is Π_2^p-complete.

Bibliographic Comments

To be done.

JSON formal model [5].

Say why we don't have a PTIME lower bound for THM 59.4 (TPQ eval in PTIME).

Our definitions of Core XPath and Conditional XPath have differences from XPath 1.0 in the W3C. XPath leashed has a paragraph about this. XPath 1.0 does not have the axes `next-sibling` or `previous-sibling`. It also does not allow nested union. I have no idea yet about other versions of XPath. The paper [Navigational XPath: calculus and algebra (ten Cate / Marx) defines Core XPath similar to how we do. There doesn't seem to be a unique definition of Core XPath in the literature. First definition was in the Gottlob/Koch/Pichler VLDB 2002 paper?]

The proof of Theorem 59.2 comes from [12, Theorem 1]. The proof for trees is by Carsten Lutz / Ulrike Sattler / Frank Wolter: Modal Logic and the Two-Variable Fragment. It's actually very similar.

The specifications for DTD and XML Schema require content models to be *deterministic*, see [**dtdspec**] and [**xmlschemaspec**]. Formally, this constraint can be abstracted as follows. Let r be a regular expression. Let $\bar{r}$ stand for the regular expression obtained from r by replacing, for every integer i and alphabet symbol a, the i-th occurrence of a in r by a_i (counting occurrences from left to right). For example, for $r = b^*a(b^*a)^*$ we have $\bar{r} = b_1^*a_1(b_2^*a_2)^*$.

Definition 63.1. *A regular expression r is* deterministic *if there are no words wa_iv and wa_jv' in $L(\bar{r})$ such that $a \in \Sigma$ and $i \neq j$.*

Notice that the expression $(a+b)^*a$ is not deterministic since both words a_2 and a_1a_2 are in $L((a_1+b_1)^*a_2)$. The equivalent expression $b^*a(b^*a)^*$ is deterministic. Brüggemann-Klein and Wood showed that not every regular expression is equivalent to a deterministic one and, therefore, that the set of deterministic regular expressions are strictly less expressive than the regular expressions. The canonical iexample for a language that is not DRE-definable is $(a+b)^*b(a+b)$ [**BrueggemannKleinW-inc98**]. Czerwinski et al. showed that it is PSPACE-complete in general to decide if the language of a given regular expression is definable by a deterministic one.

Mention TATA book.

Part IX

Expressive Languages for Tree-Structured Data

In this part we move to more expressive languages for querying and specifying tree-structured data. In Chapter 64, we study *monadic second-order logic (MSO)* on trees. Boolean MSO formulas can define precisely the *regular tree languages* which, similarly to word languages, are also characterized by finite automata. We will see a third characterization, namely through *monadic Datalog*.

The idea of storing data as trees has led to the development of schema languages for trees, based on ideas from extended context-free grammars and regular tree languages. In Chapter 67 we define the structural core of the three most widespread schema languages for XML, namely DTD, XML Schema, and Relax NG and give some insights in their expressiveness.

We study queries and schema together in Chapter 68, where we look into query optimization *under schema information*. Queries can indeed be optimized more aggressively when schema information is taken into account, but the computational complexity for problems such as satisfiability and containment with respect to schemas also increases.

64

Monadic Second Order Logic

In this chapter, we consider a significant extension of first-order logic over trees with quantification over sets. This means that we will be able to write formulae that say "there exists a set of nodes X such that ..." or "for every set X of nodes ...", which is not possible in first-order logic. The logic we obtain in this manner is called *monadic second-order logic* or simply *MSO*.

In this chapter, we will define the syntax and semantics of MSO queries over trees. In the next chapters, we study their connections to *finite automata over trees* and *monadic Datalog*.

Monadic Second Order Logic over Trees

We assume a countably infinite set

$$\mathsf{SVar}$$

of *set variables*, disjoint from Var and Nodes. Set variables will be denoted by X, Y, Z, etc. Analogously to first-order logic over trees, formulae of monadic second-order logic over trees will use relation symbols from the set

$$\{\mathrm{Lab}_a \mid a \in \mathsf{Lab}\} \cup \{E^{\mathrm{fc}}, E^{\mathrm{ns}}, E^{\mathrm{d}}, E^{\mathrm{fs}}\} \ ,$$

where Lab_a is the *a-label relation*. In the following, we abbreviate *monadic second-order logic* as MSO.

Definition 64.1: Syntax of MSO over Trees

We define *formulae of MSO over trees* as follows:

- Every first-order formula over trees is an MSO formula over trees.
- If $X \in \mathsf{SVar}$ and $y \in \mathsf{Var}$, i.e., X is a set variable and y is a first-order variable, then $X(y)$ is an MSO formula over trees.

- If φ_1 and φ_2 are MSO formulae over trees, then $(\varphi_1 \wedge \varphi_2)$, $(\varphi_1 \vee \varphi_2)$, and $(\neg\varphi_1)$ are MSO formulae over trees.
- If φ is an MSO formula over trees and X is a set variable, then $(\exists X\ \varphi)$ and $(\forall X\ \varphi)$ are MSO formulae over trees.

To avoid notional clutter, we will omit brackets in the same manner as for FO formulae over relational databases (Chapter 3). The set of *free* variables of a formula φ, denoted $\mathrm{FV}(\varphi)$, is defined as follows:

- $\mathrm{FV}(\mathrm{Lab}_a(x)) = \{x\}$ for every $a \in \mathsf{Lab}$.
- $\mathrm{FV}(x = y) = \mathrm{FV}(R(x, y)) = \{x, y\}$ for every $R \in \{E^{\mathrm{fc}}, E^{\mathrm{ns}}, E^{\mathrm{d}}, E^{\mathrm{fs}}\}$.
- $\mathrm{FV}(X(y)) = \{X, y\}$.
- $\mathrm{FV}(\neg\varphi) = \mathrm{FV}(\varphi)$.
- $\mathrm{FV}(\varphi_1 \vee \varphi_2) = \mathrm{FV}(\varphi_1 \wedge \varphi_2) = \mathrm{FV}(\varphi_1) \cup \mathrm{FV}(\varphi_2)$.
- $\mathrm{FV}(\exists x\ \varphi) = \mathrm{FV}(\forall\ \varphi) = \mathrm{FV}(\varphi) - \{x\}$.
- $\mathrm{FV}(\exists X\ \varphi) = \mathrm{FV}(\forall X\ \varphi) = \mathrm{FV}(\varphi) = \{X\}$.

An MSO formula without free variables is called an MSO *sentence*.

Example 64.2: Monadic Second-Order Formulae

Consider the MSO formula over trees

$$\exists X\ \big(\forall z_1 \forall z_2\ (E^{\mathrm{ns}}(z_1, z_2) \rightarrow X(z_1) \leftrightarrow X(z_2))\big) \wedge X(x) \wedge \neg X(y)\ . \quad (64.1)$$

The free variables of this formula are x and y.

Consider the FO-formula over trees

$$\varphi_c(x, y) = E^{\mathrm{fc}}(x, y) \vee (\exists z\ E^{\mathrm{fc}}(x, z) \wedge E^{\mathrm{fs}}(z, y))\ ,$$

which is satisfied by nodes x and y if y is a child of x. The formulae

$$\varphi_{\mathrm{root}}(x) = \neg\exists y\ E^{\mathrm{fc}}(y, x) \text{ and}$$

$$\varphi_{\mathrm{leaf}}(x) = \neg\exists y\ E^{\mathrm{fc}}(x, y)$$

are satisfied by x if x is the root or a leaf of a tree, respectively. Consider the following MSO-formula over trees:

$$\exists x \exists y\ \Big(\varphi_{\mathrm{root}}(x) \wedge \varphi_{\mathrm{leaf}}(y) \wedge$$
$$\exists X \big(\forall z_1 \forall z_2\ (\varphi_c(z_1, z_2) \rightarrow (X(z_1) \leftrightarrow \neg X(z_2)))\big) \wedge X(x) \wedge X(y)\Big) \quad (64.2)$$

This formula has no free variables.

Given a tree $T = (V, E^{\mathrm{c}}, E^{\mathrm{ns}}, \mathrm{lab})$, we define the notion of satisfaction of a formula φ with respect to an *assignment* η *for* φ *over* T. Such an assignment is a function from $\mathrm{FV}(\varphi)$ to $V \cup 2^V$ such that $\eta(x) \in V$ for every $x \in \mathsf{Var}$ and $\eta(X) \subseteq V$ for every $X \in \mathsf{SVar}$. If X is a set variable and $S \subseteq V$ a set of nodes of T, we use $\eta[X/S]$ to denote the assignment that modifies η by setting $\eta(X) = S$. We are now ready for defining the semantics of MSO formulae over trees.

Definition 64.3: Semantics of MSO over Trees

Given a tree $T = (V, E^{\mathrm{c}}, E^{\mathrm{ns}}, \mathrm{lab})$, an MSO formula φ, and an assignment η for φ over T, we inductively define when φ is *satisfied* in T under η, written $(T, \eta) \models \varphi$, as follows. We only highlight the parts of the definition that are different from the definition of the semantics of FO over trees:

- If $\varphi = X(y)$ then $(T, \eta) \models \varphi$ if and only if $\eta(y) \in \eta(X)$.
- If $\varphi = (\exists X\ \psi)$, then $(T, \eta) \models \varphi$ if and only if $(T, \eta[S/X]) \models \psi$ for some $S \subseteq V$.
- If $\varphi = (\forall X\ \psi)$, then $(T, \eta) \models \varphi$ if and only if $(T, \eta[S/X]) \models \psi$ for every $S \subseteq V$.

Example 64.4: Monadic Second-Order Formulae

We give an intuitive description of the semantic meaning of the formulae in Example 64.2:

- Formula (64.1) is satisfied by all nodes x and y that are siblings and have an even number of nodes between them.
- Formula (64.2) is satisfied in every nonempty tree that has a path from root to leaf of even length.

MSO Queries over Trees

Just like first-order queries over trees, monadic second-order queries over trees are k-ary data graph queries, that is, they take a data graph G as input and produce a set $q(G) \subseteq (\mathsf{Nodes} \cup \mathsf{Const})^k$. We first define the syntax of first-order queries over trees.

Definition 64.5: Monadic Second-Order Queries over Trees

A *monadic second-order query over trees* or *MSO query over trees* for short is an expression of the form $\varphi(\bar{x})$, where φ is an MSO formula over trees without free set variables, and $\bar{x}$ is a tuple of free first-order variables of φ such that each free variable of φ occurs in $\bar{x}$ at least once.

We define *size* $\|\varphi(\bar{x})\|$ of an MSO query over trees $\varphi(\bar{x})$ as $\|\varphi\| + \|\bar{x}\|$.

We now define the semantics of MSO queries over trees. Let $\varphi(\bar{x})$ be an MSO query over trees. Given a labeled ordered tree $T = (V, E^{\mathrm{c}}, E^{\mathrm{ns}}, \mathrm{lab})$ and a tuple $\bar{a}$ of elements from Nodes, we say that T *satisfies the query* $\varphi(\bar{x})$ *using the values* $\bar{a}$, denoted by $T \models \varphi(\bar{a})$, if there exists an assignment η for φ over T such that $\eta(\bar{x}) = \bar{a}$ and $(D, \eta) \models \varphi$. We can now define the data model and the output of MSO queries over trees. The data model defines the possible inputs of the queries. For each data graph in the data model, we define the output of the query.

Definition 64.6: Evaluation of MSO Queries over Trees

Given a labeled ordered tree $T = (V, E^{\mathrm{c}}, E^{\mathrm{ns}}, \mathrm{lab})$ and an MSO query over trees $q = \varphi(x_1, \ldots, x_k)$, where $k \geq 0$, the *output* of q on T is defined as the set of tuples

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

The *data model* of FO queries over trees is the set of labeled ordered trees.

65

Tree Automata

We now define *tree automata*, which are an extension of finite state automata, geared towards trees. We assume that readers are familiar with the basic theory of finite automata, regular expressions, and regular word languages. We provide an introduction and a reminder of the standard notation in Appendix E.

Definition 65.1: Nondeterministic Unranked Tree Automaton

A *nondeterministic unranked tree automaton (NTA)* is a quadruple

$$A = (Q, \Sigma, \delta, F) ,$$

where

- Q is a finite set of *states*,
- $\Sigma \subseteq \mathsf{Lab}$ is a finite, non-empty set of labels,
- $F \subseteq Q$ is a set of *accepting states*, and
- $\delta : Q \times \Sigma \to 2^{Q^*}$ is its *transition function*, which is such that $\delta(q, a)$ is a regular word language over Q for every $a \in \Sigma$ and $q \in Q$.

Unless we say otherwise, we always assume that $\delta(q, a)$ is represented by a nondeterministic finite automaton (NFA) over words.

The term *unranked* comes from *unranked trees*, which are trees in which every node may have an arbitrary number of children. This is opposed to *ranked trees* in which the number of children of a node is determined by the node's label. This number of children is called the *rank* of the label in the literature.

The *size* of an NTA A, denoted $\|A\|$ is defined as $|Q| + \sum_{(q,a) \in Q \times \Sigma} \|N_{q,a}\|$, where $\|N_{q,a}\|$ is the size of the NFA representing $\delta(q, a)$ (we discuss $\|N_{q,a}\|$ in Appendix E).

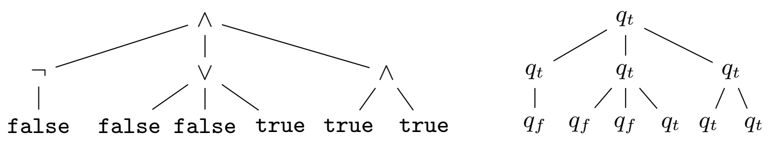


Fig. 65.1: A tree (left) and a run of the tree automaton of Example 65.2 over the tree.

A *run* of A on a tree $T = (V, E^{\mathrm{c}}, E^{\mathrm{ns}}, \mathrm{lab})$ is a mapping $\lambda : V \to Q$ such that, for every $u \in V$ with ordered sequence of children $u_1, \ldots, u_n$, we have that

$$\lambda(u_1) \cdots \lambda(u_n) \in \delta(\lambda(u), \mathrm{lab}(u)) \ .$$

Notice that, when u has no children, this condition reduces to $\varepsilon \in \delta(\lambda(u), \mathrm{lab}(u))$. A run is *accepting* if $\lambda(\mathrm{Root}(T)) \in F$, that is, it maps the root to an accepting state. A tree T is *accepted by* A if there exists an accepting run of A on T. The set of all accepted trees is denoted by $L(A)$ and is called the *language of* A.

We extend the definition of δ to trees by defining a function δ^* as follows:

- $\delta^*(a) := \{q \mid \varepsilon \in \delta(q, a)\}$; and
- $\delta^*(a(T_1, \ldots, T_n)) := \{q \mid \exists q_1 \in \delta^*(T_1), \ldots, \exists q_n \in \delta^*(T_n) \text{ with } q_1 \cdots q_n \in \delta(q, a)\}$.

Notice that a tree T is accepted by A if and only if $\delta^*(T) \cap F \neq \emptyset$.

Example 65.2: Unranked Tree Automaton

We give an example of a tree automaton A that recognizes tree representations of Boolean formulas that evaluate to `true` (as in Figure 65.1, left). The automaton has state set $Q = \{q_t, q_f\}$ and accepting states $F = \{q_t\}$. The transition function is defined as follows (we use regular expressions to denote the regular languages):

$$\begin{array}{llll} \delta(q_t, \wedge) = q_t^* & \delta(q_t, \vee) = q_f^* q_t (q_t + q_f)^* & \delta(q_t, \neg) = q_f & \delta(q_t, \texttt{true}) = \varepsilon \\ \delta(q_f, \vee) = q_f^* & \delta(q_f, \wedge) = q_t^* q_f (q_t + q_f)^* & \delta(q_f, \neg) = q_t & \delta(q_f, \texttt{false}) = \varepsilon \end{array}$$

A run of A on a tree T in Figure 65.1(left) is depicted in Figure 65.1(right). For every node u of T, the state $\lambda(u)$ appears on the position that corresponds to u in the tree on the right. For instance, if u is the rightmost leaf of T, then $\lambda(u) = q_t$.

The languages that can be accepted by unranked tree automata form a very robust class, known as the *regular tree languages*.

Definition 65.3: Regular Tree Language

A set S of trees is called a *tree language*. If there exists an unranked tree automaton A such that $S = L(A)$, then S is also called a *regular tree language*.

Algorithms on Tree Automata

We review some standard algorithms on tree automata. First we show that it can be tested in linear time combined complexity if a given tree is accepted by a given NTA.

Problem: NTA-Acceptance

Input: A labeled ordered tree T and an NTA A
Output: `true` if $T \in L(A)$ and `false` otherwise

Theorem 65.4

NTA-Acceptance can be decided in PTIME.

Proof. The algorithm computes the sets $\delta^*(T')$ for all subtrees T' of T in a bottom-up manner. For each leaf labeled a and each state q of A, it can be tested in constant time if $q \in \delta^*(a)$ by testing if $\varepsilon \in L(A_{q,a})$, where $A_{q,a}$ is the NFA for $\delta(q, a)$.

For each internal node, rooting a subtree $a(T_1, \ldots, T_n)$, we can test if $q \in \delta^*(a(T_1, \ldots, T_n))$ by testing if there is a word in $(\delta^*(T_1) \cdots \delta^*(T_n)) \cap L(A_{q,a})$.
□

Next, we show that unions and intersections of tree automata can be constructed in time proportional to the product of the sizes of the two respective automata.

Theorem 65.5

Let A_1 and A_2 be NTAs. Then NTAs for $L(A_1) \cap L(A_2)$ and $L(A_1) \cup L(A_2)$ can be constructed in time $O(\|A_1\|\|A_2\|)$.

Proof. The result follows from a "product construction" for the involved automata. Let $A_1 = (Q_1, \Sigma, \delta_1, F_1)$ and $A_2 = (Q_2, \Sigma, \delta_2, F_2)$. We first explain how to construct an automaton A for $L(A_1) \cap L(A_2)$. Here, $A = (Q_1 \times Q_2, \Sigma, \delta, F_1 \times F_2)$, where $\delta((q, q'), a) = \{(q_1, q_1') \cdots (q_n, q_n') \mid q_1 \cdots q_n \in \delta(q, a)$

Algorithm 18 Computing the set R of reachable states of an NTA

```
1: $\mathsf{R}_1 := \{q \in Q \mid \exists a \in \Sigma$ such that $\varepsilon \in \delta(q,a)\}$;
2: for $i := 2$ to $|Q|$ do
3:     $\mathsf{R}_i := \{q \in Q \mid$ there exists an $a \in \Sigma$ such that $\delta(q,a) \cap \mathsf{R}_{i-1}^* \neq \emptyset\}$;
4: $\mathsf{R} := \mathsf{R}_{|Q|}$;
```

and $q_1' \cdots q_n' \in \delta(q', a)\}$. If the NFAs for $\delta(q,a)$ and $\delta(q',a)$ use k_1 and k_2 states respectively, then an NFA for $\delta((q,q'),a)$ with k_1k_2 states can be constructed. The construction for $L(A_1) \cup L(A_2)$ is analogous but uses $(F_1 \times Q_2) \cup (Q_1 \times F_2)$ as accepting states. We leave the proof of correctness as an exercise. □

We now discuss the problem of testing whether a tree automaton accepts at least one tree. This problem is an automata-theoretic version of *satisfiability* in logic and is typically called *non-emptiness*, since it asks if the language of the automaton is non-empty.

Problem: NTA-Non-Emptiness

Input: A non-deterministic tree automaton A
Output: `true` if $L(A) \neq \emptyset$ and `false` otherwise

Theorem 65.6

NTA-Non-Emptiness is in PTIME.

Proof. Solving NTA-Non-Emptiness amounts to deciding if there exists a tree T such that $\delta^*(T)$ contains an accepting state. Let $A = (Q, \Sigma, \delta, F)$ be an NTA. Algorithm 18 computes the set of states $\mathsf{R} := \{q \mid \exists$ tree T such that $q \in \delta^*(T)\}$ in a bottom-up manner. Clearly, $L(A) \neq \emptyset$ if and only if $\mathsf{R} \cap F \neq \emptyset$. Note that $\mathsf{R}_i \subseteq \mathsf{R}_{i+1}$ and $\mathsf{R}_1 = \{\delta^*(a) \mid a \in \Sigma\}$. We argue that the algorithm is in PTIME. Clearly, R_1 can be computed in linear time. Further, the for-loop makes a linear number of iterations. Every iteration is a linear number of non-emptiness tests of the intersection of an NFA with R_{i-1}^* where $\mathsf{R}_{i-1} \subseteq Q$. As emptiness for NFAs is in linear time, the latter is also in linear time. We leave the proof of correctness as an exercise. □

Equivalence of MSO and Tree Automata

We will prove that MSO sentences and tree automata are equally expressive *over binary labeled ordered trees*, that is, labeled ordered trees in which every node has zero or two children. The result also holds for general labeled ordered trees, but the proof for the binary case is less technical. The difference between the proofs for the binary and the general case mainly lies in the following lemma, which shows that NTAs can be complemented. We denote the set of binary trees with labels in Σ by $\mathcal{T}^b_\Sigma$.

Lemma 65.7. *Let A be an NTA such that $L(A) \subseteq \mathcal{T}^b_\Sigma$. Then an NTA for $\mathcal{T}^b_\Sigma - L(A)$ can be constructed in time $O(2^{\|A\|})$.*

Proof. Let $A = (Q_A, \Sigma, \delta_A, F_A)$. The main idea of the proof is to compute a powerset automaton for A and to complement it. More precisely, we first construct an NTA $B = (Q_B, \Sigma, \delta_B, F_B)$ for $L(A)$ as follows. Define $Q_B = 2^{Q_A}$ and $F_B = \{S \mid S \cap F_A \neq \emptyset\}$. The transition function δ_B is defined such that, for each tree T, $\delta^*_B(T) = \{S\}$, where $S = \delta^*_A(T)$. That is, $\delta_B(S, a) = \{\varepsilon \mid \varepsilon \in \delta(q, a)$ for every $q \in S\} \cup \{S_1S_2 \mid \forall q \in S, \exists q_1 \in S_1, \exists q_2 \in S_2$ such that $q_1q_2 \in \delta_A(q, a)\}$. Finally, the NTA for $\mathcal{T}^b_\Sigma - L(A)$ is $\overline{B} = (Q_B, \Sigma, \delta_B, Q_B - F_B)$. We leave the proof of correctness as an exercise. □

Definition 65.8: MSO-definable tree languages

Let S be a tree language. We say that S is *MSO-definable* if there exists an MSO sentence φ such that $S = \{T \mid T \models \varphi\}$.

Theorem 65.9

Let L be a set of trees using labels from a nonempty finite set Σ. Then L is MSO-definable if and only if it is regular.

Proof. We will prove the theorem for binary trees only. Therefore, let L be a regular language of binary trees, which means that L is the language of an NTA $A = (Q, \Sigma, \delta, F)$. We can assume w.l.o.g. that $Q = [n]$. We construct an MSO formula φ such that $L = \{T \mid T \models \varphi\}$. Intuitively, φ expresses the existence of an accepting run. For every $i \in [n]$, it uses a set variable X_i for the set of nodes visited in state i. Consider the formulae

$$\begin{aligned}
\varphi_1 &= \big(\forall x\, (\vee_{i=1}^n X_i(x))\big) \wedge \big(\textstyle\bigvee_{i \neq j} \forall x\, (X_i(x) \rightarrow \neg X_j(x))\big),\\
\varphi_2 &= \forall x\, \big(\neg(\exists y\, E^{\mathrm{fc}}(y, x)) \rightarrow \textstyle\bigvee_{i \in F} X_i(x)\big),\\
\varphi_3 &= \textstyle\bigwedge_{a \in \Sigma} \forall x\, \big(\mathrm{Lab}_a(x) \wedge \neg(\exists y\, E^{\mathrm{fc}}(x, y))\big) \rightarrow \textstyle\bigvee_{\varepsilon \in \delta(i,a)} X_i(x)\big), \text{ and}\\
\varphi_4 &= \textstyle\bigwedge_{a \in \Sigma} \forall x \forall x_1 \forall x_2\, \big((\mathrm{Lab}_a(x) \wedge E^{\mathrm{fc}}(x, x_1) \wedge E^{\mathrm{ns}}(x_1, x_2))\\
&\qquad\qquad \rightarrow (\textstyle\bigvee_{(k,\ell) \in \delta(i,a)} (X_i(x) \wedge X_k(x_1) \wedge X_\ell(x_2)))\big).
\end{aligned}$$

Formula φ_1 expresses that every node is assigned exactly one state, φ_2 expresses that the root is assigned an accepting state, φ_3 expresses that the leafs are labeled in accordance with δ, and φ_4 expresses the internal nodes are labeled in accordance with δ. It is easy to show that A accepts a tree T

if and only if there exist sets $X_1, \ldots, X_n$ that satisfy φ_1, φ_2, φ_3, and φ_4. We conclude this direction by defining

$$\varphi = \exists X_1 \cdots \exists X_n \, (\varphi_1 \wedge \varphi_2 \wedge \varphi_3 \wedge \varphi_4) \, .$$

For the other direction, let φ be an MSO-formula. Since the formulae $E^{\mathrm{fs}}(x,y)$ and $E^{\mathrm{d}}(x,y)$ are expressible in MSO using the relations E^{fc} and E^{ns}, we assume that φ only uses the relations E^{fc} and E^{ns}. In fact, we can assume w.l.o.g. that φ is generated by the grammar

$$\varphi := \mathrm{Lab}_a(x) \mid E^{\mathrm{fc}}(x,y) \mid E^{\mathrm{ns}}(x,y) \mid x = y \mid \varphi \wedge \varphi \mid \neg\varphi \mid \exists x \, \varphi \mid \exists X \, \varphi \, ,$$

where $a \in \Sigma$.[1] We will further simplify the syntax of MSO-formulae by eliminating the first-order variables. That is, we introduce another logic with the same expressive power than MSO over trees that we call MSO_0. The idea is that MSO_0 introduces some syntactic sugar, but constrains formulae elsewhere. MSO_0-formulae φ over σ_{fcns} are generated over the grammar

$$\varphi := \mathsf{sing}(X) \mid \mathrm{Lab}_a(X) \mid X \subseteq Y \mid E^{\mathrm{fc}}(X,Y) \mid E^{\mathrm{ns}}(X,Y) \mid \varphi \wedge \varphi \mid \neg\varphi \mid \exists X \, \varphi \, ,$$

meaning respectively that X is a singleton; all variables in X are labeled a; the set X is a subset of Y; the sets X and Y are singletons $\{x\}$, $\{y\}$ with $E^{\mathrm{fc}}(x,y)$; the sets X and Y are singletons $\{x\}$, $\{y\}$ with $E^{\mathrm{ns}}(x,y)$; and $\exists X \varphi$. The semantics of these formulae can be defined in MSO over σ_{fcns}. For instance, $\mathsf{sing}(X)$ can be written as $\forall x \forall y \, (X(x) \wedge X(y)) \rightarrow (x = y)$ and $\mathrm{Lab}_a(X)$ as $\forall x (X(x) \rightarrow \mathrm{Lab}_a(x))$. Conversely, it is easy to show by structural induction that each MSO-formula can be tranlated into an equivalent MSO_0-formula.

We can assume w.l.o.g. that φ is an MSO_0 formula, that it does not re-use variables, and that its variables are $X_1, \ldots, X_n$. For a subformula ψ of φ with free variables S, we will construct an NTA that accepts trees for which the labels are pairs (a, f), where $a \in \Sigma$ and $f \in \{0,1\}^S$, i.e., f is a function from S to $\{0,1\}$. Intuitively, such a tree T represents a tree T' over Σ, together with an assigment of the free variables, and the task of the NTA is to accept precisely those that satisfy ψ.

More precisely, let T be a tree over $\Sigma \times \{0,1\}^S$. We denote by T_Σ the tree obtained from T by replacing each label (a, f) by a. We define η_T to be the assigment obtained from T by mapping each free variable X_i to $\eta(X_i) = \{u \mid \mathrm{lab}(u) = (a,f)$ and $f(X_i) = 1\}$.

We now inductively define NTAs A_ψ that accept precisely the trees T such that $(T_\Sigma, \eta_T) \models \psi$. An NTA for $\mathrm{Lab}_a(X_i)$ tests that the tree has no node labeled (b, f) with $f(X_i) = 1$ and $b \neq a$. An NTA for $X \subseteq Y$ is similar. An NTA for $E^{\mathrm{fc}}(X_i, X_j)$ tests that there is exactly one node u_1 labeled (a, f_1) with $f_1(X_i) = 1$, exactly one node u_2 labeled (b, f_2) with $f_2(X_j) = 1$, and that u_2 is a child of u_1. The NTAs for $\mathsf{sing}(X)$ and $E^{\mathrm{ns}}(x_i, x_j)$ are similar.

[1] For instance, subformulae $\mathrm{Lab}_b(x)$ with $b \notin \Sigma$ can be replaced by $\mathrm{Lab}_a(x) \wedge \neg\mathrm{Lab}_a(x)$ for some $a \in \Sigma$.

For the inductive step, in the cases $\psi = \neg\psi_1$ and $\psi = \psi_1 \wedge \psi_2$, we can use Theorem 65.5 and Lemma 65.7. In the case $\psi = \exists X_i\, \psi'$, assume that we have an NTA for ψ'. The NTA for ψ reads labels (a, f) for which $f(X_i)$ is undefined. It guesses labels of the form (a, f') for which f and f' agree on every free variable from ψ, and then runs as if it would be the NTA for ψ'. $\square$

Monadic Datalog

We now establish another characterization of MSO-definable queries, namely in terms of Datalog programs (see Chapter 35). We call a Datalog program Π *monadic* if all intensional relations are unary. In the following, we will consider monadic Datalog programs over the relations

$$\{\mathrm{Lab}_a \mid a \in \mathsf{Lab}\} \cup \{E^{\mathrm{fc}}, E^{\mathrm{ns}}, E^{\mathrm{d}}, E^{\mathrm{fs}}\} \cup \{\mathrm{Leaf}, \mathrm{LastSib}\} \ .$$

Given a tree $T = (V, E^{\mathrm{c}}, E^{\mathrm{ns}}, \mathrm{lab})$, we have that

- $\mathrm{Lab}_a = \{v \mid \mathrm{lab}(v) = a\}$,
- $E^{\mathrm{fc}} = \{(u,v) \mid v \text{ is the first child of } u\}$,
- $E^{\mathrm{ns}} = \{(u,v) \mid v \text{ is the next siblig of } u\}$,
- $E^{\mathrm{d}} = \{(u,v) \mid v \text{ is a descendant of } u\}$,
- $E^{\mathrm{fs}} = \{(u,v) \mid v \text{ is a following sibling of } u\}$,
- $\mathrm{Leaf} = \{v \mid v \text{ is a leaf}\}$, and
- $\mathrm{LastSib} = \{v \mid v \text{ is a last sibling}\}$.

A Boolean monadic Datalog program Π *recognizes* a set L of trees if $L = \{T \mid \Pi(T) = \mathtt{true}\}$. A set of trees L is *recognizable* by Boolean monadic Datalog if there exists a Boolean monadic Datalog program Π that recognizes L.

We will prove that the Boolean monadic Datalog queries are precisely the Boolean MSO-definable queries. The equivalence between unary MSO-definable queries and monadic Datalog queries is known to hold for unary queries as well, but the result presented here is easier to prove.

Theorem 66.1

A set L of binary trees is MSO-definable if and only if it is recognizable by a Boolean monadic Datalog program.

Proof. Let (Π, R_1) be a monadic Datalog query, where $\mathsf{edb}(\Pi) = \sigma_{\mathsf{tree}}$ and $\mathsf{idb}(\Pi) = \{R_1, \ldots, R_n\}$. Then, the MSO query can be defined as

$$\varphi := \forall R_1 \cdots \forall R_n \bigwedge_{\rho \in \Pi} \varphi_\rho \,,$$

where φ_ρ is the first-order sentence associated to Datalog rule ρ, that was used for defining the model-theoretic semantics of Datalog in Chapter 35.

Conversely, let $A = (Q, \Sigma, \delta, F)$ be an NTA. We define a Boolean monadic Datalog program (Π, R) that tests if there exists an accepting run of A on a given tree T. To this end, (Π, R) will have a unary predicate q for each $q \in Q$ that will be satisfied in node u of T if and only if $q \in \delta^*(T_{|u})$.

For every state $q \in Q$ and $a \in \Sigma$, let $N_{q,a} = (Q_{q,a}, Q, \delta_{q,a}, I_{q,a}, F_{q,a})$ be an NFA for the language $\delta(q, a)$. We can assume w.l.o.g. that Q and all state sets $Q_{q,a}$ are pairwise disjoint. In the following, we will always use q, q' to denote states from Q and p, p' to denote states from the sets $Q_{q,a}$.

Notice that we can view all $N_{q,a}$ as one automaton $(Q_N, Q, \delta_N, I_N, F_N)$, where $Q_N = \cup_{q \in Q} \cup_{a \in \Sigma} Q_{q,a}$, $I_N = \cup_{q,a} I_{q,a}$, $F_N = \cup_{q,a} F_{q,a}$, and δ_N is defined as follows. For a given $p \in \cup_{q,a} Q_{q,a}$, we define $\delta_N(p, q)$ to be the set $\delta_{q',a}(p, q)$ for the unique q' and a such that $p \in Q_{q',a}$. We define I_q to be the set of states reachable in N by reading q from an initial state, that is, $I_q := \{p \mid \exists p' \in I_N$ such that $p \in \delta_N(p', q)\}$.

For each transition such that $\varepsilon \in \delta(q, a)$, the program Π has the rule

$$q(x) :\!-\ \mathrm{Leaf}(x) \wedge \mathrm{Lab}_a(x).$$

Next, we define rules that capture the computation of the automata $N_{q,a}$. Therefore, the rules

$$\begin{array}{lll}
p(x) & :\!-\ E^{\mathrm{fc}}(y, x), \mathrm{Leaf}(x), q(x). & \text{for every } p \in I_q \\
p(x) & :\!-\ E^{\mathrm{fc}}(y, x), E^{\mathrm{fc}}(x, y'), q(x). & \text{for every } p \in I_q \\
p(x) & :\!-\ E^{\mathrm{ns}}(y, x), p'(y), \mathrm{Leaf}(x), q(x). & \text{for every } p \in \delta'(p', q) \\
p(x) & :\!-\ E^{\mathrm{ns}}(y, x), p'(y), E^{\mathrm{fc}}(x, y'), q(x). & \text{for every } p \in \delta'(p', q)
\end{array}$$

are added to Π. The rules

$$q(x) :\!-\ \mathrm{Lab}_a(x), E^{\mathrm{fc}}(x, y), \mathrm{LastSib}(y, y'), p(y') \qquad \text{for every } p \in F_{i,a}$$

test if the children of x can be visited in a sequence of states that is accepted by $N_{q,a}$. Finally, the rules $R() :\!-\ \mathrm{Root}(x), q(x)$ for every $q \in F$ tests if the root can be labeled with an accepting state. □

67

Schemas for XML

Database schemas for tree-structured data follow a different approach from those for relational data. The reason is that their task is more complex. Such schemas in practice do not only describe how attributes are associated to values, or which data types (integer, date, etc.) are allowed where, but they also describe *what is the admissible tree structure of the data*. Since one can follow different philosophies in how to specify all this information, there are different formalisms for specifying schemas for XML data — these are called *schema languages*. The most prominent schema languages are *Document Type Definition (DTD)*, *XML Schema*, and *Relax NG*. In this chapter, we explain how these three languages describe the structure of labeled ordered trees. The underlying ideas for all these languages come from *context-free grammars* (see Appendix E for a quick refresher) and from *tree automata*, which we defined in Chapter 65.

The first schema language for XML data is also the simplest and is called *Document Type Definition (DTD)*.

Definition 67.1: Document Type Definition

A *Document Type Definition (DTD)* over alphabet Σ is a triple

$$d = (\Sigma, \rho, S) ,$$

where

- $\Sigma \subseteq \mathsf{Lab}$ is a finite set of *labels*,
- ρ is a function from Σ to the set of regular expression over Σ, and
- $S \subseteq \Sigma$ is a set of *start labels*.

In practice, the labels in Σ are also called *element names*, which is a convention that we will also follow.

```
<?xml version="1.0" encoding="UTF-8"?>
<inventory>
  <category name="concert guitars">
    <item>
      <maker> Tandler </maker>
      <model> Advanced Student </model>
      <description> Spruce top,
                    Indian rosewood back and sides </description>
      <price> Please ask </price>
    </item>
    <item>
      <maker> Hanika </maker>
      <model> Grand Concert </model>
      <description> Spruce top,
                    Palo Escrito back and sides </description>
      <price> 4299 </price>
      <discount> 10 </discount>
    </item>
    [...]
  </category>
  [...]
</inventory>
```

Fig. 67.1: Fragment of an XML document representing the inventory of a store

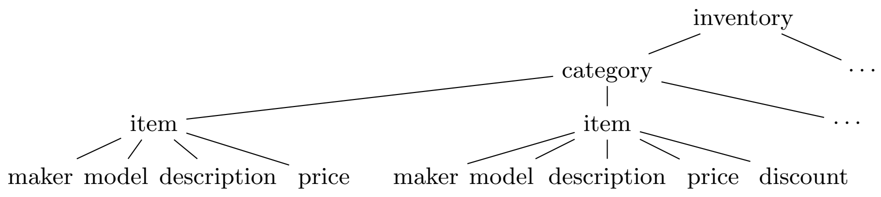


Fig. 67.2: The tag structure of the data in Figure 67.1 as a sibling-ordered tree

We now define the semantics of DTDs. Let $T = (V, E^{\mathrm{c}}, E^{\mathrm{ns}}, \mathrm{lab})$ be a labeled ordered tree. We say that T *satisfies* d if $\mathrm{lab}(\mathrm{Root}(T)) \in S$ and, for every $v \in V$ with ordered sequence of children $v_1, \ldots, v_n$, the word $\mathrm{lab}(v_1) \cdots \mathrm{lab}(v_n)$ is in $L(r)$, where $r = \rho(\mathrm{lab}(v))$. By $L(d)$ we denote the set of trees that satisfy d.

In real-world DTDs, ρ is usually written as a set of rules rather than a function. We will follow this convention and write $a \to r$ if $\rho(a) = r$. The *size* of a DTD d is $\sum_{a \in \Sigma} \|\rho(a)\|$.

Example 67.2

Throughout the chapter, we will illustrate schemas using a running example as depicted in Figure 67.1. The data is an XML document describing the inventory of a store, which sells items that are classified into categories. Each item has a maker, model, price, and optionally a description and a discount. For space reasons, we only list one category, in which we only list two items. (The reader can imagine further items and categories where we have put the placeholder `[...]`.)

Figure 67.2 depicts the tree structure induced by the nesting of the tags in the XML document. It is at this level of abstraction that DTDs operate.

A DTD describing such data can be defined with $\Sigma = \{$inventory, category, item, maker, model, description, price, discount$\}$ and $S = \{$inventory$\}$. The function ρ is defined as follows:

$$\begin{array}{lcl} \text{inventory} & \to & \text{category}^* \\ \text{category} & \to & \text{item}^* \\ \text{item} & \to & \text{maker model description? price discount?} \end{array}$$

It describes that inventory should have zero or more category children which, in turn, have zero or more item children. Each item has children labeled maker, model, price and, optionally description and discount, from left to right. The tree in Figure 67.2 satisfies this DTD.

Now assume that we want to express in our running example that each category contains at least one item with a discount child. DTDs cannot express this property,[1] since they only associate a single regular expression to item. Schema languages such as XML Schema and Relax NG solve this limitation by extending DTDs with *types*.

Definition 67.3: Extended DTD

An *extended DTD (EDTD)* is a tuple

$$D = (\Sigma, \Gamma, \rho, S, \mu) \ ,$$

where

- $\Sigma \subseteq \mathsf{Lab}$ is a finite set of *labels*,
- $\Gamma \subseteq \mathsf{Lab}$ is a finite set of *types*,

[1] In Theorem 67.9 we will learn a method to formally prove this.

- (Γ, ρ, S) is a DTD over alphabet Γ, and
- $\mu : \Gamma \to \Sigma$.

Intuitively, a labeled ordered tree $T = (V, E^{\mathrm{c}}, E^{\mathrm{ns}}, \mathrm{lab})$ satisfies an EDTD if there exists an assignment of types to the nodes in V such that the typed tree is a derivation tree of the underlying DTD (Γ, ρ, S).

Formally, for a tree T^{Γ} using labels in Γ, let us denote by $\mu(T^{\Gamma})$ the tree obtained from T^{Γ} by replacing each label $a \in \Gamma$ with $\mu(a) \in \Sigma$. Hence, $\mu(T^{\Gamma})$ only uses labels in Σ. We now say that T *satisfies* D if there exists a tree T^{Γ} such that $T^{\Gamma} \in L((\Gamma, \rho, S))$ and $\mu(T^{\Gamma}) = T$. We call T^{Γ} a *witness* for T. Again, we denote the set of trees satisfying D by $L(D)$.

The *size* of an EDTD D is the size of the underlying DTD (Γ, ρ, S). In examples, we denote EDTDs as sets of rules, just as we did before with DTDs.

Example 67.4

Continuing Example 67.2, we will define an EDTD that describes inventories in which each category has at least one item with a discount child. We take Σ as before and use $\Gamma = \{$inventory, category, normal, discounted, maker, model, description, price, discount$\}$. For defining ρ, we will use a simplified notation, much like how schemas are written in XML Schema or Relax NG: we use $a[b]$ to denote $a \in \Sigma$ and $b \in \Gamma$ with $\mu(b) = a$, and we abbreviate $a[a]$ by a. The mapping ρ can now be defined as follows.

$$\begin{array}{ll}
\text{inventory} & \to \text{category}^* \\
\text{category} & \to \text{item[normal]}^* \ \text{item[discounted]} \\
 & \qquad\qquad\qquad (\text{item[normal]} + \text{item[discounted]})^* \\
\text{item[normal]} & \to \text{maker model description? price} \\
\text{item[discounted]} & \to \text{maker model description? price discount}
\end{array}$$

The rule for category requires that at least one child is assigned the type discounted. As we can see in the rule for item[discounted], this means that the item must have a child with label discount. The (partial) tree in Figure 67.3 shows how a witness for the tree in Figure 67.2 w.r.t. the present EDTD can be found.

Figure 67.4a shows how the rule for category in Example 67.4 can be defined in Relax NG (with a partial definition of the rule for the type "normal"). In Relax NG, `element` blocks are used to define the elements from Σ (and their content), whereas `define` blocks are used to define types. Since we don't need a special type for category, we can define it immediately using an `element` block.

In Figure 67.4b, we try to mimic this behavior using XML Schema syntax. However, the code fragment is not syntactically correct because it violates

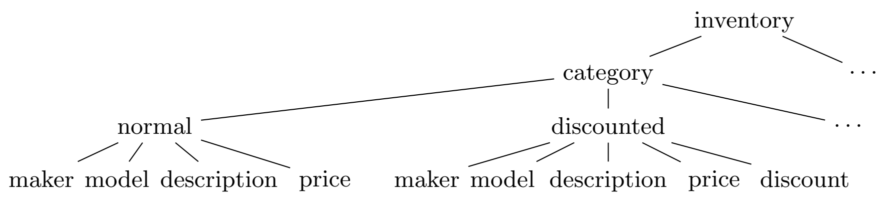


Fig. 67.3: A (partial) witness of the tree in Figure 67.2 for the EDTD in Example 67.4

XML Schema's *Element Declarations Consistent (EDC)* constraint, which forbids the occurrence of different types associated to the same element name in a regular expression $\rho(t)$. So, the occurrence of both types normal and discount associated to the same element name item is a violation of this constraint. We formalize this constraint next.

Definition 67.5: Single-Type EDTDs

Let $D = (\Sigma, \Gamma, \rho, S, \mu)$ be an EDTD. A regular expression r over Γ is *single-type* if it does not contain distinct types t_1 and t_2 with $\mu(t_1) = \mu(t_2)$. We say that D is a *single-type EDTD (stEDTD)* when S does not contain distinct types t_1 and t_2 with $\mu(t_1) = \mu(t_2)$ and the regular expression $\rho(t)$ is single-type for every $t \in \Gamma$.

Although single-type EDTDs do not precisely capture the power of XML Schema on the level of labeled ordered trees,[2] they are a very useful theoretical abstraction.

EDTDs are Equivalent to Tree Automata

We prove that EDTDs have the same expressiveness of tree automata. Furthermore, EDTDs and tree automata can even be converted back and forth in polynomial time. To this end, if D and D' are tree automata, DTDs, or EDTDs, we say that D is *equivalent to* D' if $L(D) = L(D')$.

Theorem 67.6

Each EDTD D can be translated in time $O(\|D\|)$ to an equivalent NTA. Conversely, each NTA A over alphabet Σ can be translated in $O(\|A\|\|\Sigma\|^2)$ time to an equivalent EDTD.

[2] XML Schema has an additional restricion on the regular expressions in ρ, which we briefly discuss in the bibliographic notes.

```
<element name="category">
  <zeroOrMore><ref name="normal"/></zeroOrMore>
  <ref name="discounted"/>
  <zeroOrMore>
    <choice><ref name="normal"/><ref name="discounted"/></choice>
  </zeroOrMore>
</element>

<define name="normal">
  <element name="item"> ... </element>
</define>
```

(a) Relax NG code fragment corresponding to the EDTD rule for category

```
<element name="category">
  <complexType>
    <sequence>
      <element name="item" type="normal"
               minOccurs="0" maxOccurs="unbounded"/>
      <element name="item" type="discounted"/>
      <choice minOccurs="0" maxOccurs="unbounded">
        <element name="item" type="normal"/>
        <element name="item" type="discounted"/>
      </choice>
    </sequence>
  </complexType>
</element>

<complexType name="normal">
  <element name="item"> ... </element>
</complexType>
```

(b) Code fragment in XML Schema syntax (violating EDC), corresponding to the EDTD of Example 67.4

Fig. 67.4: Fragments of Relax NG and XML Schema code

Proof. Let $D = (\Sigma, \Gamma, \rho, S, \mu)$ be an EDTD. An equivalent NTA $A = (Q, \Sigma, \delta, F)$ is obtained by taking $Q = \Gamma$, $F = S$, and $\delta(t, \mu(t)) = \rho(t)$ for every type $t \in \Gamma$.

Conversely, let $A = (Q, \Sigma, \delta, F)$ be an NTA. An equivalent EDTD $D = (\Sigma, \Gamma, \rho, S, \mu)$ is obtained by taking $\Gamma = Q \times \Sigma$, $S = F \times \Sigma$, and $\mu((q, a)) = a$ for each type (q, a). Let $L_{(q,a)}$ denote the language obtained from $\delta(q, a)$ by replacing, in each word, every symbol $p \in Q$ by the disjunction $\sum_{b \in \Sigma}(p, b)$. Since $\delta(q, a)$ is a regular language over Q, the language $L_{(q,a)}$ is also regular.

For every $(q, a) \in \Gamma$, we define $\rho((q, a))$ to be a regular expression for $L_{(q,a)}$. We leave the proofs that D and A are equivalent as exercises to the reader. □

Complexity of Validation

We now consider the problem of testing if a tree satisfies a given schema. Since we have three types of schema languages (DTD, EDTD, and stEDTD), we define the problem in a general form.

Problem: $\mathcal{L}$-Validation

Input: A tree T and a schema D from the class $\mathcal{L}$
Output: `true` if $t \in L(D)$ and `false` otherwise

Theorem 67.7

EDTD-Validation is in PTIME.

Proof. This is an immediate consequence of Theorems 65.4 and 67.6. □

We have the following corollary since DTDs and stEDTDs are special cases of EDTDs.

Corollary 67.1. DTD-Validation *and* stEDTD-Validation *are in* PTIME.

Expressiveness

We now investigate the expressiveness of DTDs, stEDTDs, and EDTDs. The following is immediate from Theorem 67.6.

Theorem 67.8

EDTDs recognize precisely the regular tree languages.

If we want to characterize the languages that are definable by DTDs and stEDTDs, we need the notion of *subtree exchange*. Let $T = (V, E^{c}, E^{ns}, \text{lab})$ and $T' = (V', E'_{\downarrow}, E'_{\rightarrow}, \text{lab}')$ be two trees, and let $u \in V$. We assume w.l.o.g. that V and V' are disjoint (otherwise, the nodes in V' can be renamed). We denote by $T[u \leftarrow T']$ the tree obtained by replacing the subtree $T|_u$ of T rooted at u by the tree T'. (We omit a formal definition, but $T[u \leftarrow T']$ has none of the nodes in T_u, all the nodes of T' and has the edge $(v, \text{Root}(T'))$ instead of the edge (v, u).)

We say that a tree language L is *closed under node-guarded subtree exchange*, if the following holds. Whenever two trees $T_1, T_2 \in L$ have nodes u_1 and u_2 respectively, with $\text{lab}(u_1) = \text{lab}(u_2)$, then $T_1[u_1 \leftarrow T_2|_{u_2}] \in L$.

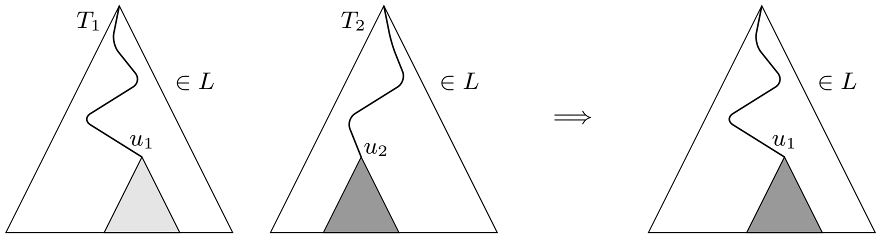


Fig. 67.5: Ancestor-guarded subtree exchange

Theorem 67.9

DTDs recognize precisely the regular tree languages that are closed under node-guarded subtree exchange.

Proof. Observe that, by definition, every language recognized by a DTD is closed under node-guarded subtree exchange. Conversely, let $D = (\Sigma, \Gamma, \rho, S, \mu)$ be an EDTD such that $L(D)$ is closed under node-guarded subtree exchange. We construct a DTD $d = (\Sigma, \rho_d, S_d)$ as follows. Define $S_d = \{\mu(t) \mid t \in S\}$. For a regular expression r over Γ, denote by $\mu(r)$ the expression obtained from r by replacing each symbol t with $\mu(t)$. Then we define $\rho_d(a)$ as the disjunction of all $\mu(\rho(t))$ over $\{t \mid \mu(t) = a\}$. It is now easy to show that $L(d) = L(D)$. □

We now provide a similar characterization for stEDTDs. For a tree $T = (V, E^{\mathrm{c}}, E^{\mathrm{ns}}, \mathrm{lab})$ and node $u \in V$, denote by $\mathsf{ancestorlab}^T(u)$ the word $\mathrm{lab}(u_1) \cdots \mathrm{lab}(u_n)$, where $u_1 \cdots u_n$ is the path from $\mathrm{Root}(T)$ to u in T. A tree language L is *closed under ancestor-guarded subtree exchange*, if the following holds. Whenever two trees $T_1, T_2 \in L$ have nodes u_1 and u_2 respectively, with $\mathsf{ancestorlab}^{T_1}(u_1) = \mathsf{ancestorlab}^{T_2}(u_2)$, then $T_1[u_1 \leftarrow T_2|_{u_2}] \in L$. We illustrate the concept in Figure 67.5.

Theorem 67.10

stEDTDs recognize precisely the regular tree languages that are closed under ancestor-guarded subtree exchange.

Proof. It is not difficult to prove that every language definable by an stEDTD is a regular language closed under ancestor-guarded subtree exchange, see Exercise 9.5.

Conversely, let $D = (\Sigma, \Gamma, \rho, S, \mu)$ be an EDTD such that $L(D)$ is closed under ancestor-guarded subtree exchange. We will construct a single-type EDTD E such that $L(E) = L(D)$. We will assume w.l.o.g. that D only uses *useful* types, that is, for every type $t \in \Gamma$, there exists a fixed tree

$T_t \in L((\Gamma, \rho, \{t\}))$. We will make use of the following general property, which immediately follows from the definition of EDTDs:

(†) If T_1, T_2 are trees in $L(D)$ with witnesses T_1', T_2', respectively, such that u_1 in T_1 and u_2 in T_2 have the same type in T_1' and T_2', respectively, then $T_1[u_1 \leftarrow T_2|_{u_2}] \in L(D)$.

For a word $w \in \Sigma^*$ and $a \in \Sigma$, let $\mathsf{types}(wa)$ be the set of all types t_a for which there is a tree $T \in L(D)$ with witness $T' \in L((\Gamma, \rho, S))$, and a node v such that $\mathsf{ancestorlab}^T(v) = wa$ and $\mathrm{lab}^{T'}(v) = t_a$. For each $a \in \Sigma$, let $\Gamma(D, a)$ be the set of all nonempty sets $\mathsf{types}(wa)$, with $w \in \Sigma^*$. Clearly, each $\Gamma(D, a)$ is finite.

We next define $E = (\Sigma, \Gamma_E, \rho_E, S_E, \mu_E)$. Its set of types Γ_E is $\bigcup_{a \in \Sigma} \Gamma(D, a)$ and its set of start types S_E is $\{\mathsf{types}(a) \mid a \in \Sigma\}$. For every type $\tau \in \Gamma(D, a)$, set $\mu_E(\tau) = a$. The mapping ρ_E maps each type $\mathsf{types}(wa)$ to the disjunction of all $\rho(t_a)$ for $t_a \in \mathsf{types}(wa)$, with each t_b in $\rho(t_a)$ with $\mu(t_b) = b$ replaced by $\mathsf{types}(wab)$. Notice that, by (†), the definition of the rules of ρ_E does not depend on the actual choice of wa.

Clearly, E is single-type and $L(D) \subseteq L(E)$. We show that $L(E) \subseteq L(D)$. To this end, let $T \in L(E)$ and let T' be a witness. We call a set N of nodes of T *well-formed*, if (1) for each node $v \in N$, all its ancestors are in N and (2) if a child u of a node v is in N then all children of v are in N. The singleton set N_0 containing the root is well-formed. We say that a tree T_1 *agrees* with T on a well-formed set N_1 of nodes of T_1, if N_1 can be mapped to a well-formed set of nodes N of T by a mapping m which respects the child-relationship, the order of siblings, and the labels of nodes. By definition, $\mathsf{ancestorlab}(v) = \mathsf{ancestorlab}(m(v))$ for every $v \in N$.

As the trees in $L(D)$ and $L(E)$ have the same possible root labels, there exists a tree $T_1 \in L(D)$ which agrees with T on N_0. To complete the proof, it is sufficient to prove the following.

Claim 67.2. *If there exists a tree $T_1 \in L(D)$ which agrees with $T = (V, E^c, E^{ns}, lab)$ on a well-formed set N with $m(N) \subsetneq V$ then there exists $T_2 \in L(D)$ which agrees with T on a well-formed, strict superset of N.*

For the proof of this claim, let T_1 be as stated and let T_1' be its witness. Let $v \in N$ be such that the children of $m(v)$ are not in $m(N)$ and let $wa = \mathsf{ancestorlab}^{T_1}(v) = \mathsf{ancestorlab}^{T}(m(v))$.

By construction of E, the regular expression $\rho_E(\mathsf{types}(wa))$ is a disjunction $\sum_{t \in \mathsf{types(wa)}} \rho'(t)$, where ρ' is defined as ρ but, for every $b \in \Sigma$, has the type $\mathsf{types}(wab) \in \Gamma_E$ instead of $t_b \in \Gamma$ if $\mu(t_b) = b$. Let t_a be such that the children of $m(v)$ are typed in T' according to a disjunct $\rho'(t_a)$, with $t_a \in \mathsf{types}(wa)$. Thus, there is a tree $T_3 \in L(D)$ with a node u such that $\mathsf{ancestorlab}^{T_3}(u) = wa$ and the type of u is t_a in the witness T_3' for T_3.

Let $v_1, \ldots, v_n$ be the ordered sequence of children of $m(v)$ in T and choose, for each $i \in [1, n]$, a type $f(v_i)$ such that $f(v_1) \cdots f(v_n)$ matches $\rho(t_a)$. Let T_4

be obtained from T_3 by (1) removing everything below u, (2) adding $v_1, \dots, v_n$ below u, and (3) adding for each child v_i the subtree $T_{f(v_i)}$ which exists because $f(v_i)$ is a useful type. Clearly, $T_4 \in L(D)$ by (†). Furthermore, by the ancestor-closed subtree exchange property, the tree T_2 resulting from T_1 by replacing the subtree rooted at v by the subtree of T_4 rooted at u is in $L(D)$, too. $\square$

Exercises

Exercise 9.1. (a) Prove that the construction in Theorem 65.5 is correct.

(b) Prove that the algorithm in Theorem 65.6 is correct.

(c) Prove that the construction in Lemma 65.7 is correct.

Exercise 9.2. Prove that Lemma 65.7 also holds for general labeled ordered trees. That is, let us denote by $\mathcal{T}_\Sigma$ the set of all labeled ordered trees with labels in Σ. Then, for every NTA A, we can construct an NTA for $\mathcal{T}_\Sigma - L(A)$ in time $O(2^{\|A\|})$.

Exercise 9.3. The cores of XML Schema and Relax NG, seen as grammars, are a bit more involved than what we described in Chapter **??**, even if we allow arbitrary regular expressions. For XML Schema, it can be more accurately defined as tuples (Σ, τ, R, S) where $S \subseteq \Sigma \cup (\Sigma \times \tau)$ and R maps $\Sigma \cup \tau$ to regular expressions over $\Sigma \cup (\Sigma \times \tau)$. For Relax NG, R maps $\Sigma \cup \tau$ to regular expressions over $\Sigma \cup \tau \cup (\Sigma \times \tau)$. Prove that this is equally expressive.

Exercise 9.4. Prove that the constructions in Theorem 67.6 are correct, that is, prove that the EDTD D and automaton A are indeed equivalent in both constructions.

Exercise 9.5. Prove that every language definable by an stEDTD is a regular language closed under ancestor-guarded subtree exchange.

Hint: Use a partial function $f : \Sigma^+ \to \Gamma$ that maps each "ancestor-string" to the relevant type, i.e.,

- $f(a) = t$, where $t \in S$ is unique such that $\mu(t) = a$ and
- $f(wa) = t$, where $f(w) = t'$ and t is the unique type occurring in $\rho(t')$ with $\mu(t) = a$.

Some ideas for exercises:

- Can we postpone canonical models to here?

- Prove that minimal patterns are not unique in a strong sense?
- Prove that you cannot define “trees for which the deepest node is on even depth” with a tree automaton.
- Prove that you cannot define “the tree has a unique b-labeled leaf” with XSD (single-type EDTD).
- Turn a regular expression into a deterministic one?

Exercise 9.6. Prove that, given a TPQ p and EDTD D, you can construct in polynomial time a Boolean TPQ p_b and EDTD D_b such that the following are equivalent:

(1) There exists a tree $T \in L(D)$ such that $p(T)$ is not empty.

(2) There exists a tree $T \in L(D_b)$ such that $T \models p_b$.

Hint: add children with unique labels to the output nodes of p.

Exercise 9.7. When reducing a DTD, it is important that steps 1 and 2 in the proof of Lemma 68.1 are performed in that order. Show that, if the order is reversed, the result is not always a reduced DTD.

Exercise 9.8. Prove that the NP-hardness of Theorem 68.2 also holds for DTDs and tree patterns without descendant edges or wildcards. *Hint: Try to “flatten” the trees defined by the DTD.*

Bibliographic Comments

To be done.

References

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases.* Addison-Wesley, 1995. ISBN: 0-201-53771-0. URL: http://webdam.inria.fr/Alice/ (cit. on p. 19).

[2] Sanjeev Arora and Boaz Barak. *Computational Complexity - A Modern Approach.* Cambridge University Press, 2009 (cit. on p. 20).

[3] Catriel Beeri et al. "On the Desirability of Acyclic Database Schemes". In: *J. ACM* 30.3 (1983), pp. 479–513. URL: https://doi.org/10.1145/2402.322389 (cit. on p. 247).

[4] Hans L. Bodlaender. "A Linear-Time Algorithm for Finding Tree-Decompositions of Small Treewidth". In: *SIAM J. Comput.* 25.6 (1996), pp. 1305–1317 (cit. on p. 247).

[5] Pierre Bourhis, Juan L. Reutter, and Domagoj Vrgoc. "JSON: Data model and query languages". In: *Inf. Syst.* 89 (2020), p. 101478 (cit. on pp. 520, 564).

[6] Diego Calvanese et al. "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.* 2000, pp. 176–185 (cit. on p. 593).

[7] Marco A. Casanova, Ronald Fagin, and Christos H. Papadimitriou. "Inclusion Dependencies and Their Interaction with Functional Dependencies". In: *J. Comput. Syst. Sci.* 28.1 (1984), pp. 29–59. DOI: 10.1016/0022-0000(84)90075-8. URL: https://doi.org/10.1016/0022-0000(84)90075-8 (cit. on p. 89).

[8] Ashok K. Chandra and Philip M. Merlin. "Optimal Implementation of Conjunctive Queries in Relational Data Bases". In: *Proceedings of the 9th Annual ACM Symposium on Theory of Computing, May 4-6, 1977, Boulder, Colorado, USA.* Ed. by John E. Hopcroft, Emily P. Friedman, and Michael A. Harrison. ACM, 1977, pp. 77–90. DOI: 10.1145/800105.803397. URL: https://doi.org/10.1145/800105.803397 (cit. on p. 145).

[9] Hubie Chen and Víctor Dalmau. "Beyond Hypertree Width: Decomposition Methods Without Decompositions". In: *Principles and Practice of Constraint Programming - CP 2005, 11th International Conference, CP 2005, Sitges, Spain, October 1-5, 2005, Proceedings.* Vol. 3709. Lecture Notes in Computer Science. Springer, 2005, pp. 167–181. URL: https://doi.org/10.1007/11564751%5C_15 (cit. on p. 247).

[10] E. F. Codd. "A Relational Model of Data for Large Shared Data Banks". In: *Commun. ACM* 13.6 (1970), pp. 377–387. DOI: 10.1145/362384.362685. URL: http://doi.acm.org/10.1145/362384.362685 (cit. on p. 89).

[11] E. F. Codd. "Relational Completeness of Data Base Sublanguages". In: *Research Report / RJ / IBM / San Jose, California* RJ987 (1972) (cit. on p. 89).

[12] Kousha Etessami, Moshe Y. Vardi, and Thomas Wilke. "First-Order Logic with Two Variables and Unary Temporal Logic". In: *Inf. Comput.* 179.2 (2002), pp. 279–295. DOI: 10.1006/inco.2001.2953. URL: https://doi.org/10.1006/inco.2001.2953 (cit. on pp. 520, 564).

[13] Ronald Fagin. "Degrees of Acyclicity for Hypergraphs and Relational Database Schemes". In: *J. ACM* 30.3 (1983), pp. 514–550. URL: https://doi.org/10.1145/2402.322390 (cit. on p. 247).

[14] Ronald Fagin, Alberto O. Mendelzon, and Jeffrey D. Ullman. "A Simplified Universal Relation Assumption and Its Properties". In: *ACM Trans. Database Syst.* 7.3 (1982), pp. 343–360. URL: https://doi.org/10.1145/319732.319735 (cit. on p. 247).

[15] Jörg Flum and Martin Grohe. *Parameterized Complexity Theory*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2006 (cit. on p. 247).

[16] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database Systems - the Complete Book*. Pearson Education, 2009. ISBN: 978-0-13-187325-4 (cit. on p. 19).

[17] Erich Grädel et al. *Finite Model Theory and its Applications*. Springer, 2008 (cit. on p. 20).

[18] Paul R. Halmos. *Measure Theory*. Springer, 1974 (cit. on p. 19).

[19] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 2003, pp. I–XIV, 1–521. ISBN: 978-0-321-21029-6 (cit. on p. 20).

[20] Neil Immerman. *Descriptive Complexity*. Springer, 1999, pp. I–XVI, 1–268. ISBN: 978-1-4612-6809-3, 978-0-387-98600-5 (cit. on p. 20).

[21] Dexter Kozen. *Automata and Computability*. Springer, 1997. ISBN: 978-0-387-94907-9 (cit. on p. 20).

[22] Leonid Libkin. *Elements of Finite Model Theory*. Springer, 2004. ISBN: 3-540-21202-7 (cit. on pp. 20, 358).

[23] Christos H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994. ISBN: 978-0-201-53082-7 (cit. on p. 20).

[24] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. McGraw-Hill, 2003. ISBN: 978-0-07-115110-8 (cit. on p. 19).

[25] Neil Robertson and Paul D. Seymour. "Graph minors. II. Algorithmic Aspects of Tree-Width". In: *Journal of Algorithms* 7.3 (1986), pp. 309–322 (cit. on p. 247).

[26] Kenneth H. Rosen. *Discrete Mathematics and its Applications*. McGraw-Hill, 2006 (cit. on p. 20).

[27] Abraham Silberschatz, Henry F. Korth, and S. Sudarshan. *Database System Concepts*. McGraw-Hill Book Company, 2005. ISBN: 978-0-07-295886-7 (cit. on p. 19).

[28] Michael Sipser. *Introduction to the Theory of Computation*. PWS Publishing Company, 1997. ISBN: 978-0-534-94728-6 (cit. on p. 20).

[29] Robert Endre Tarjan and Mihalis Yannakakis. "Simple Linear-Time Algorithms to Test Chordality of Graphs, Test Acyclicity of Hypergraphs, and Selectively Reduce Acyclic Hypergraphs". In: *SIAM J. Comput.* 13.3 (1984), pp. 566–579. DOI: `10.1137/0213035`. URL: `https://doi.org/10.1137/0213035` (cit. on p. 247).

[30] Moshe Y. Vardi. "The Complexity of Relational Query Languages (Extended Abstract)". In: *Proceedings of the 14th Annual ACM Symposium on Theory of Computing, May 5-7, 1982, San Francisco, California, USA*. Ed. by Harry R. Lewis et al. ACM, 1982, pp. 137–146. DOI: `10.1145/800070.802186`. URL: `https://doi.org/10.1145/800070.802186` (cit. on p. 89).

[31] Domagoj Vrgoč. "Querying graphs with data". PhD thesis. University of Edinburgh, 2014 (cit. on p. 593).

[32] Ingo Wegener. *Complexity Theory*. Springer, 2005. ISBN: 978-3-540-21045-0. DOI: `10.1007/3-540-27477-4`. URL: `https://doi.org/10.1007/3-540-27477-4` (cit. on p. 20).

[33] Mihalis Yannakakis. "Algorithms for Acyclic Database Schemes". In: *Very Large Data Bases, 7th International Conference, September 9-11, 1981, Cannes, France, Proceedings*. IEEE Computer Society, 1981, pp. 82–94 (cit. on p. 247).

Part XI

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

$$f(x_1, \ldots, x_\ell) \textit{ is in } O(g(x_1, \ldots, x_\ell))$$

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 $i \in [\ell]$, we have $f(x_1, \ldots, x_\ell) \leq k \cdot g(x_1, \ldots, x_\ell)$.

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.

Sometimes we will use a variant of the big-O notation that hides logarithmic factors. This variant will be denoted with $\tilde{O}$ instead of the usual letter O. We define this notation as follows.

Definition 1.2: Big-O Notation Without Logarithmic Factors

Let $f, g : \mathbb{N}^\ell \to \mathbb{R}_0^+$, where $\ell \geq 1$. We say that

$$f(x_1, \ldots, x_\ell) \textit{ is in } \tilde{O}(g(x_1, \ldots, x_\ell))$$

if $f(x_1, \ldots, x_\ell)$ is in $O\Big(g(x_1, \ldots, x_\ell) \cdot \log\big(g(x_1, \ldots, x_\ell)\big)\Big)$.

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*),
- $\delta : (Q - \{\text{"yes"}, \text{"no"}\}) \times \Sigma \rightarrow Q \times \Sigma \times \{\rightarrow, \leftarrow, -\}$ 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 $c \rightarrow_M c'$, and we also write $c \rightarrow_M^m c'$ if c' can be reached from c in m steps, and $c \rightarrow_M^* c'$ if $c \rightarrow_M^m c'$ for some $m \geq 0$ (we assume that $c \rightarrow_M^0 c$). 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 $sc(w) \rightarrow_M^* c$ 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, ..., c_{m-1} such that $c \to_M c_1$, $c_1 \to_M c_2$, ..., $c_{m-1} \to_M c'$. 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 $\delta(q, a_1, a_2) = (q', \mathrm{dir}_1, b, \mathrm{dir}_2)$, 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 $c \rightarrow_M c'$. We also write $c \rightarrow_M^m c'$ if c' can be reached from c in m steps, and $c \rightarrow_M^* c'$ if $c \rightarrow_M^m c'$ for some $m \geq 0$ (we assume that $c \rightarrow_M^0 c$).

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 $sc(w) \rightarrow_M^* (\text{"halt"}, u_1, v_1, u_2, v_2, \triangleright u, \varepsilon)$.

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, $sc(w) \rightarrow_M^m c$ 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

$\textsc{PTime} = \bigcup_{k\in\mathbb{N}} \textsc{Time}(n^k)$ $\qquad$ $\textsc{NP} = \bigcup_{k\in\mathbb{N}} \textsc{NTime}(n^k)$
$\textsc{ExpTime} = \bigcup_{k\in\mathbb{N}} \textsc{Time}(2^{n^k})$ $\qquad$ $\textsc{NExpTime} = \bigcup_{k\in\mathbb{N}} \textsc{NTime}(2^{n^k})$
$\textsc{2ExpTime} = \bigcup_{k\in\mathbb{N}} \textsc{Time}(2^{2^{n^k}})$

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

$$\begin{aligned}\textsc{Space}(f(n)) \;=\; \{L \mid & \text{ there exists a 2-TM that decides } L \\ & \text{ and runs in space } f(n)\}\end{aligned}$$

and

$$\begin{aligned}\textsc{NSpace}(f(n)) \;=\; \{L \mid & \text{ there exists a 2-NTM that decides } L \\ & \text{ and runs in space } f(n)\}.\end{aligned}$$

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

$\textsc{DLogSpace} = \textsc{Space}(\log n)$ $\qquad$ $\textsc{NLogSpace} = \textsc{NSpace}(\log n)$
$\textsc{PSpace} = \bigcup_{k\in\mathbb{N}} \textsc{Space}(n^k)$ $\qquad$ $\textsc{NPSpace} = \bigcup_{k\in\mathbb{N}} \textsc{NSpace}(n^k)$
$\textsc{ExpSpace} = \bigcup_{k\in\mathbb{N}} \textsc{Space}(2^{n^k})$ $\qquad$ $\textsc{NExpSpace} = \bigcup_{k\in\mathbb{N}} \textsc{NSpace}(2^{n^k})$

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 $\text{co}\mathcal{C}$ is defined as the set of complements of the problems in $\mathcal{C}$, that is, $\text{co}\mathcal{C} = \{\Sigma^* - L \mid L \in \mathcal{C}\}$. 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 $|u_2v_2| \leq f(|w|)$.

$$\begin{aligned}&\textsc{DLogSpace} \subseteq \textsc{NLogSpace} \subseteq \textsc{PTime} \subseteq \text{NP} \subseteq \textsc{PSpace} = \textsc{NPSpace}\\ &\subseteq \textsc{ExpTime} \subseteq \textsc{NExpTime} \subseteq \textsc{ExpSpace} = \textsc{NExpSpace} \subseteq \textsc{2ExpTime}\end{aligned}$$

$$\textsc{PTime} \subsetneq \textsc{ExpTime} \subsetneq \textsc{2ExpTime}$$

$$\text{NP} \subsetneq \textsc{NExpTime}$$

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 NP $\subseteq \mathcal{C}$ or 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 $f : V \to \{\texttt{true}, \texttt{false}\}$. 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

$$\begin{tikzcd}
 & \mathrm{NP} \arrow[dr, phantom, "\subseteq"] & & \Sigma_2^p \arrow[dr, phantom, "\subseteq"] & \\
\textsc{PTime} \arrow[ur, phantom, "\subseteq"] \arrow[dr, phantom, "\subseteq"] & & \mathrm{NP} \cup \mathrm{coNP} \arrow[ur, phantom, "\subseteq"] \arrow[dr, phantom, "\subseteq"] & & \textsc{PSpace} \\
 & \mathrm{coNP} \arrow[ur, phantom, "\subseteq"] & & \Pi_2^p \arrow[ur, phantom, "\subseteq"] &
\end{tikzcd}$$

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 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 $\Box$ 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 $\Box$ 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. $\Box$

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.

Encoding of Queries and Constraints

Queries q and constraints σ will most commonly be coming from a query language and a class of constraints, respectively, for which we define the *size* $\|q\|$ or $\|\sigma\|$ throughout the book. Since queries and constraints of size n can be encoded as words over a finite alphabet of length $n \log n$, we define the length of their Turing Machine encoding to be $\|q\| \log \|q\|$ and $\|\sigma\| \log \|\sigma\|$, respectively.