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

Principles of Databases

January 20, 2021

Santiago Paris
Bayreuth Edinburgh

Contents

1

Introduction

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

The general philosophy of the book is the following:

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

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

We are also open to exercise suggestions. We have generated some initial ideas for exercises, but we are aware that the exercises for the currently released parts still needs work. We also plan to accompany each part with bibliographic remarks. (These are not implemented yet, even for Parts I and II.)

The finished product will consist of the following parts:

(I) The Relational Model: The Classics

(II) Conjunctive Queries

(III) Fast Conjunctive Query Evaluation

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

(IV) Expressive Languages

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

(V) Uncertainty

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

(VI) Query Answering Paradigms

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

(VII) Mappings and Views

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

(VIII) Tree-Structured Data

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

(IX) Graph-Structured Data

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

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

2

Background

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

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

Basic Notions and Notation

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

Sets

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

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

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

We further say that

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

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

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

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

Sequences and Tuples

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

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

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

Functions

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

We say that a function f is

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

A useful notion is that of composition of functions. Given two functions $f : S \to T$ and $g : T \to U$, the *composition* of f and g, denoted $g \circ f$, is the function from S to U defined as follows: $g \circ f(a) = g(f(a))$ for every $a \in S$.

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

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 tuples in the named perspective, and denote tuples as $t = (A_1 \colon a_1, \ldots, A_k \colon a_k)$, meaning that $t(A_i) = a_i$ for every $i \in [k]$. Notice that, according to this notation, the tuples $(A_1 \colon a_1, A_2 \colon a_2)$ and $(A_2 \colon a_2, A_1 \colon a_1)$ represent the same function t. A *relation instance* in the named perspective is a *finite* set S of database tuples of the same sort U, which we also call the sort of the relation instance S and denote by $\mathrm{sort}(S)$. By nRI (for *named* relational instances) we denote the set of all such relation instances. A *possibly infinite relation instance* in the named perspective is defined as the notion of relation instance, but without forcing it to be finite. We write nRI^∞ for the set of all possibly infinite relation instances in the named perspective.

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

Definition 2.1: Named Database Schema

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

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

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

In other words, a named database schema $\mathbf{S}$ provides a finite set of relations 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. We are now ready to 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 $\text{sort}(D(R)) = \mathbf{S}(R)$, for every $R \in \text{Dom}(\mathbf{S})$. This means that D is either finite as in Definition 2.2, where each relation name of $\text{Dom}(\mathbf{S})$ is mapped to a finite relation instance, or infinite in the sense that at least one relation name of $\text{Dom}(\mathbf{S})$ is mapped to an infinite relation instance. Infinite database instances are obviously not a real-life concept, and we are not interested in studying them per se. Having said that, they are a very useful mathematical tool as they allow us to prove some results in a more elegant way. In other words, infinite database instances are considered for purely technical reasons, which will be revealed later in the book.

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

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

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

Unnamed Perspective. Under the *unnamed perspective*, a *database tuple* is an element of Const^k for some $k \in \mathbb{N}$. We denote such tuples using lowercase letters from the beginning of the alphabet, that is, as $(a_1, \ldots, a_k)$, $(b_1, \ldots, b_k)$, etc., or even more succinctly as $\bar{a}, \bar{b}$, etc. A *relation instance* in the unnamed perspective is a *finite* set S of database tuples of the same arity k. We say that k is the *arity* of S, denoted by $\text{ar}(S)$. 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 $\text{Dom}(\mathbf{S})$ is finite. For a relation name $R \in \text{Dom}(\mathbf{S})$, the *arity of R under $\mathbf{S}$*, denoted $\text{ar}_{\mathbf{S}}(R)$, is defined as $\mathbf{S}(R)$.

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

Definition 2.4: Database Instance (The Unnamed Case)

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

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

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

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

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

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

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

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

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

Simplified Terminology and Notation

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

Although database schemas are formally defined as partial functions, with their domain being a finite subset of Rel, it is often convenient to tread them as sets of relation names. Thus, we will usually tread a schema $\mathbf{S}$ as the finite set $\mathrm{Dom}(\mathbf{S})$. This means that whenever we write $\mathbf{S} = \{R_1, \ldots, R_n\}$, we actually mean that $\mathrm{Dom}(\mathbf{S}) = \{R_1, \ldots, R_n\}$. In the unnamed case, we may also write

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

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

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

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

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 $R \in \mathrm{Dom}(\mathbf{S}')$,

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

From Unnamed to Named. Consider an unnamed database schema $\mathbf{S}$. We assume that Att contains an attribute name $\#_i$ for each $i \geq 1$. We define

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

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

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

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

Queries and Query Languages

Queries will appear throughout the book as both *semantic* and *syntactic* objects. As a semantic object, a query is a function that maps databases of some schema $\mathbf{S}$ to databases of some schema $\mathbf{S}'$. Consistently with what happens in real-life query languages, where queries return tables, we assume that the schema $\mathbf{S}'$ always consists of a single relation. We call such a schema a *relation schema*, that is, a database schema that contains only one relation. Since the name of the relation defined by $\mathbf{S}'$ is often not important, we do not explicitly state it, and we assume that $\text{Dom}(\mathbf{S}') = \{\text{Answer}\}$, where Answer is a default name for the output table, and its arity is determined by the considered query.

Definition 2.5: Queries and Query Languages

Consider a database schema $\mathbf{S}$, and a relation schema $\mathbf{S}'$. A *query* from $\mathbf{S}$ to $\mathbf{S}'$ is a function of the form

$$q \;:\; \text{Inst}(\mathbf{S}) \to \text{Inst}(\mathbf{S}').$$

A *query language* is a set of queries.

[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.

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

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

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

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

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

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 relations).

The *arity* of a query is defined as the arity of the relation that it produces. A query of arity k is called *k-ary*. If it produces a 0-ary relation, the query is also called *Boolean*. In this case, there are only two possible outputs, namely the empty set $\{\}$, and the singleton set $\{()\}$ containing the empty tuple. We interpret $\{()\}$ as the Boolean value `true`, and $\{\}$ as `false`. For readability, we write $q(D) = \texttt{true}$ in place of $q(D) = \{()\}$, and $q(D) = \texttt{false}$ in place of $q(D) = \{\}$. When denoting Boolean queries, we will often omit the empty tuple $()$ from the notation. We will simply write $q = \varphi$ rather than $q = \varphi()$.

A useful notion that we will use throughout the book, and, in particular, for defining the syntax of query languages that are based on logic, is that of relational atom. When R is a k-ary relation symbol and $\bar{u} \in (\mathsf{Const} \cup \mathsf{Var})^k$, $R(\bar{u})$ is a *relational atom*. Observe that the only difference between a fact and a relational atom is that the former mentions only constants, whereas the latter can mention both constants and variables. As for facts, since a relational

atom is always a statement about a single tuple, we simply write $R(u_1, \ldots, u_k)$ instead of $R((u_1, \ldots, u_k))$. Given a set of atoms S, we write $\mathrm{Dom}(S)$ for the set of constants and variables in S. For example, $\mathrm{Dom}(\{R(a, x, b), R(x, a, y)\}) = \{a, b, x, y\}$. We also write R^S for the set of tuples $\{\bar{u} \mid R(\bar{u}) \in S\}$.

Key Problems: Query Evaluation and Query Analysis

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

Query Evaluation

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

Problem: $\mathcal{L}$-Evaluation

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

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

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

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

Problem: q-Evaluation

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

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

Definition 2.6: Data Complexity

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

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

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

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

Data Complexity refers to the complexity of $\mathcal{L}$-Evaluation when its input consists only of D and $\bar{a}$, whereas q on the other hand 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. This assumes that queries return sets, and 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.

To reason about the computational complexity of problems, such as the ones introduced above, we need to represent their inputs as inputs to Turing Machines, that is, as words over some finite alphabet. Details on how databases and queries are encoded as words over a finite alphabet can be found in Appendix C. Henceforth, for an object o, e.g., a database or a query, we write $\|o\|$ for the size of its encoding, that is, the size of the word that encodes o.

Further Background Reading

Should the reader find himself/herself in a situation "that he/she does not have the prerequisites for reading the prerequisites" [5], rather than being discouraged he/she 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, [3, 11, 13]. 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 [12]. For additional information about computability theory, we provide a primer in Appendix B. Furthermore, we refer the reader to [6, 8, 14]; standard texts on complexity theory are [2, 10, 15]. For the foundations of finite model theory and descriptive complexity, the reader is referred to [4, 7, 9].

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 see 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 ($\forall$). The formal definition follows.

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.

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

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

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

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

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

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

Semantics of First-Order Logic

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

Definition 3.3: Semantics of First-Order Logic

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

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

An assignment η for such a sentence φ has an empty domain (since the domain of η is $\text{FV}(\varphi)$), 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 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.

Note that we defined the semantics of FO in a way that is well-suited for database applications, but slightly departs from the logic literature where one would allow η to associate arbitrary elements of Const to variables, whereas we only allow elements of $\mathrm{Dom}(D) \cup \mathrm{Dom}(\varphi)$. 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. This semantics ensures a key property, called *safety*, which we discuss at the end of the chapter. A crucial property of the active domain semantics is that each FO formula has a *finite* number of satisfying assignments.

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 k-ary relation $q(D) \subseteq \mathsf{Const}^k$, 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 k-tuple of free variables of φ such that each free variable of φ occurs in $\bar{x}$ at least once.

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 $\mathrm{Dom}(D) \cup \mathrm{Dom}(\varphi)$, 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.

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.

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

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

It is clear that $q(D) \subseteq (\mathrm{Dom}(D) \cup \mathrm{Dom}(\varphi))^k$. Therefore, q defines a k-ary query from $\mathbf{S}$ to $\mathbf{S}' = \{\mathrm{Answer}[k]\}$ in the sense of Definition 2.5.

Example 3.7: 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

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

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

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

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

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

Boolean First-Order queries

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

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

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

Safety of First-Order Queries

Finally, we come back to the *safety* property that we touched upon earlier. By definition, each assignment for an FO formula φ over a database D maps the free variables of φ to the finite set $\mathrm{Dom}(D) \cup \mathrm{Dom}(\varphi)$. This means that the output of a query $q = \varphi(\bar{x})$ on a database D is finite, which is important for q being a *query*. The term *safety* refers to the fact that we safely remain in the realm of finite relations. Observe that, if we would use the semantics of FO from logic textbooks, namely use assignments η that map to Const instead of $\mathrm{Dom}(D) \cup \mathrm{Dom}(\varphi)$, the set $q(D) = \{\bar{a} \mid D \models \varphi(\bar{a})\}$ could be infinite and therefore unsafe. For example, under such semantics, $\neg R(x)$ is satisfied over a database D by every η with $\eta(x) \in (\mathsf{Const} - R^D)$, which is an infinite set.

4

Relational Algebra

Queries expressed in FO are declarative and tell us what the output of a query should be. An alternative procedural language prescribes a way to implement such a declarative query, and obtain its output by a sequence of operations on the data. The standard such language is *relational algebra*, abbreviated RA. We present the definitions of relational algebra in the unnamed and the named perspectives. The following table gives a quick overview of the operators in the named and unnamed relational algebra.

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

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

Syntax of the Unnamed Relational Algebra

Under the unnamed perspective, RA consists of five primitive operations: selection, projection, Cartesian product, union, and difference. Before giving the formal definitions of those operations, we first introduce the notion of condition over a set of integers that is needed for defining the selection operation. A *condition* θ over $\{1, \ldots, k\}$, for some $k \geq 0$, is a Boolean combination of

statements of the form $i \doteq j$, $i \doteq a$, $i \neq\!\!\!\doteq j$, and $i \neq\!\!\!\doteq a$, 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 $i \neq\!\!\!\doteq j$ 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 $i \neq\!\!\!\doteq a$ is used to indicate that this value must be different than a. Let us clarify that we use the symbols $\doteq$ and $\neq\!\!\!\doteq$, 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 $i \neq\!\!\!\doteq j$, i.e., Boolean combinations using only conjunction $\wedge$ and disjunction $\vee$. For example, $\neg\big((1 \doteq 2) \vee (2 \neq\!\!\!\doteq 3)\big)$ is equivalent to $(1 \neq\!\!\!\doteq 2) \wedge (2 \doteq 3)$.

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.

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 $a_i = a$
$\bar{a} \models i \not\doteq j$	if $a_i \neq a_j$	$\bar{a} \models i \not\doteq a$	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:

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

[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.

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

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.

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 as in the unnamed case. 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 that map databases D to the output $e(D)$. In the remainder of the book, we will therefore sometimes also refer to e as a *query*.

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

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

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

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

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 $t = (A_1 \colon a_1, \ldots, A_k \colon a_k)$ in $D(R)$, for a relation name R of sort $\{A_1, \ldots, A_k\}$ (with $(R, A_1) \lessdot \cdots \lessdot (R, A_k)$), into a tuple $t' = (a_1, \ldots, a_k)$ in $f(D)(R)$. Let q_{n} be a named RA query, q_{u} an unnamed RA query, and $\mathbf{S}$ a named database schema. We say that q_{n} *is equivalent to* q_{u} *under* $\boldsymbol{S}$ if, for every database D of $\mathbf{S}$, $f(q_{\mathsf{n}}(D)) = q_{\mathsf{u}}(f(D))$. Note that two queries can be equivalent under one schema and non-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 $\lessdot$). Thus, renaming some attributes in the named perspective will change the order inside the unnamed tuples.

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}. Recall from Chapter 2 that q_{n} maps a database of $\mathbf{S}$ always to a single relation with name "Answer". Assume that q_{n} has sort $\{A_1, \ldots, A_k\}$ with $(\text{Answer}, A_1) \lessdot \cdots \lessdot (\text{Answer}, A_k)$. We proceed to explain how to obtain an unnamed RA query q_{u} that is equivalent to q_{n}, which means that the i-th attribute in the output of q_{u} corresponds to the A_i-attribute in the output of q_{n}. In the remainder of the proof, whenever we write a set of attributes as a set $\{A_1, \ldots, A_k\}$, we assume that $(\text{Answer}, A_1) \lessdot \cdots \lessdot (\text{Answer}, A_k)$. The base cases are the following:

- 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 $j = |\{i \mid (\text{Answer}, A'_i) \lessdot (\text{Answer}, B)\}|$. Then $q_{\mathsf{u}} = \pi_\alpha(q'_{\mathsf{u}})$, where α is the list obtained from $(1, \ldots, k)$ by deleting i and reinserting it right after j if $j > 0$, and at the beginning of the list if $j = 0$.
- Finally, if $q_{\mathsf{n}} = q'_{\mathsf{n}} \cup q''_{\mathsf{n}}$, then $q_{\mathsf{u}} = q'_{\mathsf{u}} \cup q''_{\mathsf{u}}$, where q'_{u} and q''_{u} are the unnamed RA expressions that are obtained by the induction hypothesis for q'_{n} and q''_{n}, respectively. The case when $q_{\mathsf{n}} = q'_{\mathsf{n}} - q''_{\mathsf{n}}$ is analogous. □

5

Relational Algebra and SQL

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

A Core of SQL

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

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

and we can form more complex queries by using expressions

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

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

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

```
RelationWithAVeryLongName AS ShortName
```

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

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

The keyword `DISTINCT` is to instruct the query to perform duplicate elimination. In general, SQL tables and query results are allowed to contain duplicates. For example, in a database containing two facts, $R(a, b)$ and $R(a, c)$, projecting on the first column would result in *two* copies of a. We shall discuss duplicates in Chapter 37. In this Chapter, we will always assume that SQL queries only return sets, and omit `DISTINCT` from queries used in examples.

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

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

Example 5.1: SQL Queries

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

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

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

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

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

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

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

Relational Algebra to Core SQL

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

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

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

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

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

```
SELECT a AS A
```

Selection and Projection. Assume that e is translated into

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

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

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

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

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

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

Rename. Assume now that e is translated into

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

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

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

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

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

and that e_2 is translated into

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Core SQL to Relational Algebra

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

Assume that we start with the query

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

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

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

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

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

Other SQL Features Captured by RA

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

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

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

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

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

Example 5.2: Subqueries in SQL

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

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

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

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

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

Example 5.3: Subqueries in FROM

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

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

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

Other SQL Features Not Captured by RA

Bag Semantics. As mentioned already, SQL's data model is based on bags, i.e., the same tuple may occur multiple times in a database or output of a query. Here we tacitly assumed that all relations are sets and each `SELECT` is followed by `DISTINCT` to ensure that duplicates are eliminated. To see how RA operations change in the presence of duplicates, see Chapter 37.

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 27.

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 32 and 33.

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 27), this is a subject that we do study in this book.

6

Equivalence of Logic and Algebra

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

Theorem 6.1

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

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

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

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

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

From RA to FO

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

From FO to RA

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

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

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

where

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

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

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

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

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

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

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

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

The inductive construction defines the expression

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

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

7

First-Order Query Evaluation

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

Combined Complexity

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

Theorem 7.1

FO-Evaluation is PSPACE-complete.

Proof. We start with the upper bound. We prove the theorem for the case where $\bar{a}$ is a tuple over $\mathrm{Dom}(D)$ and leave the extension to arbitrary tuples of constants as an exercise.

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

By Definition 3.6, $\bar{a} \in q(D)$ if and only if $(D, \eta) \models \varphi$ with η being the assignment for φ over D such that $\eta(\bar{x}) = \bar{a}$. Therefore, to establish Theorem 7.1, it suffices to show that the problem of checking whether $(D, \eta) \models \varphi$ is in PSPACE. This is done by exploiting the recursive procedure EVALUATION, depicted in Algorithm 7.1. 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 EVALUATION$(\varphi, D, \eta) =$ `true`. It remains to argue that EVALUATION(φ, D, η) uses polynomial space.

Algorithm 7.1 EVALUATION(φ, D, η)

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

Lemma 7.2. EVALUATION*(φ, D, η) runs in space $O(\|\varphi\|^2 \cdot \log \|D\|)$.*

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

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

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

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

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

and the Boolean FO query

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

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

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

Data Complexity

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

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

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

Theorem 7.3

FO-Evaluation is in DLogSpace in data complexity.

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

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

8

Static Analysis

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

Satisfiability

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

Problem: $\mathcal{L}$-Satisfiability

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

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

We are asking this question on finite databases. Had it been asked over possibly infinite databases, a classical result in logic from the 1930s, known as Church's theorem (sometimes Church-Turing theorem) would have told us that it is undecidable. But over finite databases, the problem is still undecidable; this was proved by Trakhtenbrot a number of years after the Church-Turing theorem. This is what we show next.

Theorem 8.1: Trakhtenbrot's Theorem

FO-Satisfiability is undecidable.

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

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

The Boolean FO query q_M will be over the schema

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

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

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

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

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

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

The Sentence $\varphi_{\prec}$

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

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

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 $\prec$:

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

The Sentence φ_{succ}

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

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

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

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

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

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

and the machine M is in state s

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

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

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

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

each tape cell y contains exactly one symbol

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

and the head points at exactly one cell

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

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

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

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 φ_{halt}. Finally, this formula checks whether M has reached an accepting or a rejecting configuration

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

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

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

The proof of Theorem 8.1 relies on the fact that databases are *finite*. For possibly infinite databases, the proof given above does not work. Indeed, 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, the Church-Turing theorem shows 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.

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.

[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).

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. In this chapter, we define homomorphisms and provide some simple examples that illustrate this notion.

Definition of Homomorphism

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

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

Definition 9.1: Homomorphism

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

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

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

Example 9.2: Homomorphism

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

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

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

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

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

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

Grounding Sets of Atoms

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

Definition 9.3: Grounding

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

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

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

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

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

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

- *the* grounding of S, denoted $S^{\downarrow}$, and
- *the* unique bijective homomorphism $\mathsf{G}_S : S \to S^{\downarrow}$.

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

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

10

Functional Dependencies

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

One of the most important classes of dependencies supported by relational systems is the class of *functional dependencies*, which can express that the values of some attributes of a tuple uniquely (or functionally) determine the values of other attributes of that tuple. For example, considering the 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 essentially states that whenever two tuples of the relation Person agree on the first attribute, i.e., the id, they should also agree on all the other attributes. In fact, this form of dependency, 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 the Person relation.

Syntax and Semantics

We start with the syntax of functional dependencies.

Definition 10.1: Syntax of Functional Dependencies

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

$$R : U \to V$$

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

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

Definition 10.2: Semantics of FDs

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

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

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

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

Satisfaction of Functional Dependencies

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

Problem: FD-Satisfaction

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

It is not difficult to show the following result:

Theorem 10.3

FD-Satisfaction is in PTIME.

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

The Chase for Functional Dependencies

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

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

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

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

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

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

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

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

Definition 10.4: The Chase for FDs

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

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

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

Lemma 10.5. *There is no infinite chase sequence of S under Σ.*

[2] Formal proofs are omitted since in Chapter 45 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 10.6. *Let $S_0, \ldots, S_n$ and $S'_0, \ldots, S'_m$ be two finite chase sequences of S under Σ. Then it holds that $S_n = S'_m$.*

The above lemma allows us to refer to *the* result of the chase of S under Σ, denoted by $\text{Chase}(S, \Sigma)$, which is defined as S_n for some (any) finite chase sequence $S_0, \ldots, S_n$ of S under Σ. Notice that we do not need to define the result of infinite chase sequences under FDs since, by Lemma 10.5, they do not exist. Hence, $\text{Chase}(S, \Sigma)$ is either the symbol $\bot$, or a finite set of relational atoms. It is not difficult to verify that in the latter case, $\text{Chase}(S, \Sigma)$ satisfies Σ. Actually, this follows from the definition of successful chase sequences.

Lemma 10.7. *If* $\text{Chase}(S, \Sigma) \neq \bot$, *then* $\text{Chase}(S, \Sigma) \models \Sigma$.

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

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

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

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

It is clear that $h_s(S_0) = h_s(S) = S_n$. Since, by Lemma 10.6, different finite chase sequences have the same result, we get the following.

Lemma 10.8. *Let s and s' be successful finite chase sequences of S under Σ. It holds that $h_s(S) = h_{s'}(S)$.*

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

By Lemma 10.5, $\text{Chase}(S, \Sigma)$ can be computed after finitely many steps. Furthermore, assuming that $\text{Chase}(S, \Sigma) \neq \bot$, also the chase homomorphism $h_{S,\Sigma}$ can be computed after finitely many steps. In fact, as the next lemma states, this is even possible after polynomially many steps.

Lemma 10.9. $\text{Chase}(S, \Sigma)$ *can be computed in polynomial time. Furthermore, if* $\text{Chase}(S, \Sigma) \neq \bot$, *then $h_{S,\Sigma}$ can be computed in polynomial time.*

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

Lemma 10.10. *Let S' be a set of atoms over $\boldsymbol{S}$ such that $(S, \bar{u}) \to (S', \bar{v})$ and $S' \models \Sigma$. If* $\text{Chase}(S, \Sigma) \neq \bot$, *then* $(\text{Chase}(S, \Sigma), h_{S,\Sigma}(\bar{u})) \to (S', \bar{v})$.

Note that the definition of the chase for FDs, as well as its main properties, would be technically simpler if we focus on sets of constant-free atoms since in this case there are no failing chase sequences. As we shall see, this suffices for studying the implication problem for FDs. Nevertheless, we consider sets of atoms with constants since the chase is also used in Chapter 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 10.11

FD-Implication is in PTIME.

To show Theorem 10.11, we first show how implication of FDs can be characterized via the chase for FDs. This is done by showing that checking whether a set of FDs Σ implies an FD σ boils down to checking whether the result of the chase of the prototypical set of relational atoms S_σ that violates σ is a set of atoms that satisfies σ. Given an FD σ of the form $R : U \to V$, the set S_σ is defined as $\{R(x_1, \ldots, x_{\text{ar}(R)}), R(y_1, \ldots, y_{\text{ar}(R)})\}$, where

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

We can now show the following useful characterization:

Proposition 10.12

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 10.7, we have that $\text{Chase}(S_\sigma, \Sigma) \models \Sigma$. Since, by Lemma 10.5, $\text{Chase}(S_\sigma, \Sigma)$ is finite, we get that $\text{Chase}(S_\sigma, \Sigma) \models \sigma$.

($\Leftarrow$) Consider now a database D of $\mathbf{S}$ such that $D \models \Sigma$, and with σ being of the form $R : \{i_1, \ldots, i_k\} \to \{j_1, \ldots, j_\ell\}$, assume that there are tuples $(a_1, \ldots, a_{\text{ar}(R)}), (b_1, \ldots, b_{\text{ar}(R)}) \in R^D$ such that $(a_{i_1}, \ldots, a_{i_k}) = (b_{i_1}, \ldots, b_{i_k})$. Recall also that S_σ is of the form $\{R(x_1, \ldots, x_{\text{ar}(R)}), R(y_1, \ldots, y_{\text{ar}(R)})\}$. Let $\bar{z} = (x_{j_1}, \ldots, x_{j_\ell}, y_{j_1}, \ldots, y_{j_\ell})$ and $\bar{c} = (a_{j_1}, \ldots, a_{j_\ell}, b_{j_1}, \ldots, b_{j_\ell})$. It is clear that $(S_\sigma, \bar{z}) \to (D, \bar{c})$. Since $D \models \Sigma$ and $\text{Chase}(S_\sigma, \Sigma) \neq \bot$, by Lemma 10.10

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

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

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

which in turn implies that

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

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

By Proposition 10.12, we get a simple procedure for checking whether a set Σ of FDs implies an FD σ that runs in polynomial time:

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

We know that the set of atoms $\text{Chase}(S_\sigma, \Sigma)$ can be constructed in polynomial time (Lemma 10.9), and we also know that $\text{Chase}(S_\sigma, \Sigma) \models \sigma$ can be checked in polynomial time (Theorem 10.3), and Theorem 10.11 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. For example, having the 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 11.1: Syntax of Inclusion Dependencies

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

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

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

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

Definition 11.2: Semantics of INDs

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

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

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

Satisfaction of Inclusion Dependencies

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

Problem: IND-Satisfaction

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

It is not difficult to show the following result:

Theorem 11.3

IND-Satisfaction is in PTIME.

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

The Chase for Inclusion Dependencies

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

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

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

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

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

Definition 11.4: 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 $\mathsf{new}(\sigma, \bar{u}) \in S_j$.

  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 11.4 simply says that S_{i+1} is obtained from S_i by applying σ to S_i with $\bar{u}$, while

[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.

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 11.4, 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 11.5. *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 $S_n = S'_m$.*
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 11.5 allows us to refer to *the* unique result of the chase of S under Σ, denoted $\mathrm{Chase}(S, \Sigma)$, which is defined as the result of some (any) finite or

[2] Formal proofs are omitted since in Chapter 36 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.

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 11.2. Nevertheless, Definition 11.2 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 11.6. *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 11.7. *Let* S' *be a set of atoms over* $\boldsymbol{S}$ *such that* $(S, \bar{u}) \to (S', \bar{v})$ *and* $S' \models \Sigma$. *It holds that* $(\text{Chase}(S, \Sigma), \bar{u}) \to (S', \bar{v})$.

Implication of Inclusion Dependencies

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

Problem: IND-Implication

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

Although for FDs the implication problem is tractable (Theorem 10.11), for INDs it turns out to be an intractable problem:

Theorem 11.8

IND-Implication is PSPACE-complete.

We first concentrate on the upper bound. We are going to establish a result, analogous to Proposition 10.12 for FDs, that characterizes implication of INDs via the chase. However, since the chase for INDs may build an infinite set of

atoms, we can only characterize implication under possibly infinite databases. Given a set Σ of INDs over a schema $\mathbf{S}$ and a single IND σ over $\mathbf{S}$, we say that Σ *implies without restriction* σ, denoted $\Sigma \models^{\infty} \sigma$, if, for every possibly infinite database D of $\mathbf{S}$, we have that $D \models \Sigma$ implies $D \models \sigma$.

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

Lemma 11.9. *Consider a set Σ of INDs over schema $\boldsymbol{S}$, and an IND σ over $\boldsymbol{S}$. It holds that $\Sigma \models^{\infty} \sigma$ if and only if* $\text{Chase}(S_\sigma, \Sigma) \models \sigma$.

Proof. ($\Rightarrow$) By hypothesis, for every possibly infinite set of relational atoms S, it holds that $S \models \Sigma$ implies $S \models \sigma$. By Lemma 11.6, $\text{Chase}(S_\sigma, \Sigma) \models \Sigma$, and therefore, $\text{Chase}(S_\sigma, \Sigma) \models \sigma$.

($\Leftarrow$) Consider now a possibly infinite database D of $\mathbf{S}$ such that $D \models \Sigma$, and with σ being of the form $R[i_1, \ldots, i_k] \subseteq P[j_1, \ldots, j_k]$, assume that there exists a tuple $(a_1, \ldots, a_{\text{ar}(R)}) \in R^D$. Recall also that S_σ is of the form $\{R(x_1, \ldots, x_{\text{ar}(R)})\}$. Let $\bar{y} = (x_{i_1}, \ldots, x_{i_k})$ and $\bar{b} = (a_{i_1}, \ldots, a_{i_k})$. It is clear that $(S_\sigma, \bar{y}) \to (D, \bar{b})$. Since $D \models \Sigma$, by Lemma 11.7

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

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

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

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

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

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

Lemma 11.9 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 11.10: 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 11.9 and Theorem 11.10 is the following:

Corollary 11.11

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 11.11, the reader may think that the procedure for checking whether $\Sigma \models \sigma$, which will lead to the PSPACE upper bound claimed in Theorem 11.8, 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 11.3. However, it should not be forgotten that $\text{Chase}(S_\sigma, \Sigma)$ may be infinite. Therefore, we need to rely on a finer procedure that avoids the explicit construction of $\text{Chase}(S_\sigma, \Sigma)$. We proceed to present a technical lemma that is the building block of this refined procedure, but first we need some terminology.

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

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

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

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

A witness of σ relative to Σ is essentially a compact representation, which uses only $\text{ar}(R) + 1$ variables, of a sequence of atoms of $\text{Chase}(S_\sigma, \Sigma)$ that witnesses the following: starting from $S_\sigma = \{R(x_1, \ldots, x_{\text{ar}(R)})\}$, an atom $P(y_1, \ldots, y_{\text{ar}(P)})$ with $\pi_{(i_1,\ldots,i_m)}(x_1, \ldots, x_{\text{ar}(R)}) = \pi_{(j_1,\ldots,j_m)}(y_1, \ldots, y_{\text{ar}(P)})$ can be obtained via chase applications, which means that $\text{Chase}(S_\sigma, \Sigma) \models \sigma$. It is also not difficult to see that if $\text{Chase}(S_\sigma, \Sigma) \models \sigma$, then a witness of σ relative to Σ can be extracted from $\text{Chase}(S_\sigma, \Sigma)$. This discussion is summarized in the following technical lemma, whose proof is left as an exercise.

Lemma 11.12. *Consider a set Σ of INDs over a schema $\mathbf{S}$, and an IND σ over $\mathbf{S}$. Then,* $\text{Chase}(S_\sigma, \Sigma) \models \sigma$ *iff there is a witness of σ relative to Σ.*

By Corollary 11.11 and Lemma 11.12, 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 11.2. 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`.

Algorithm 11.2 ImplicationWitness(Σ, σ)

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

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

It is easy to verify that Algorithm 11.2 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_∇ with $\bar{y}$ (see line 6), and to check if $T' = P$ and $\pi_{(i_1,\ldots,i_k)}(\bar{x}) = \pi_{(j_1,\ldots,j_k)}(\bar{z})$ (see line 14). Therefore, IND-Implication is in NPSPACE, and thus in PSPACE since NPSPACE = PSPACE.

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

12

Exercises for Part I

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

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

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

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

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

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

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

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

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

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

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

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 10.3 and 11.3 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 11.10. 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 11.9, 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 11.12.

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.

Part II

Conjunctive Queries

13

Syntax and Semantics

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

Syntax of Conjunctive Queries

We start with the syntax of conjunctive queries.

Definition 13.1: Syntax of Conjunctive Queries

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

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

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

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

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

Recall from Chapter 2 that a query q maps a database to a single relation, which by default is called Answer and its arity is equal to the arity of q. This

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

Example 13.2: Conjunctive Queries

Consider again the relational schema from Example 3.2:

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

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

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

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

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

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

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

Semantics of Conjunctive Queries

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

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

Notice that CQs only use constants inside relational atoms and, in particular, do not use equational atoms. For this reason, the output of CQs only consists of tuples of constants from Dom(D).

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

Consider a database D and a CQ q of the form

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

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

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

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

Definition 13.3: Evaluation of CQs

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

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

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

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

Example 13.4: Evaluation of CQs

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

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

Observe that the assignment η for q over D such that

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

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

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

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

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

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

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

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

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

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

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

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

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

Conjunctive Queries as a Fragment of FO

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

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

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

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

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

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

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

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

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

From the above discussion, we immediately get that:

Theorem 13.6

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

Notice that $\text{FO}^{\text{rel}}[\wedge, \exists]$ is *not* the same as $\text{FO}[\wedge, \exists]$, 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 $\text{FO}[\wedge, \exists]$ allows also for equational atoms, which means that the query $\varphi(x, y)$ with $\varphi = (x = y)$ is an $\text{FO}[\wedge, \exists]$ 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 allows for selection, where conditions in selections are conjunctions of equalities, projection, and Cartesian product. This is usually called the *select-project-join* (SPJ) fragment of RA; henceforth, we simply refer to SPJ queries. Recall that the join operation is actually a selection from the Cartesian product on a condition that is a conjunction of equalities. We proceed to show that:

Theorem 13.7

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

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

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

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

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

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

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

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

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

Example 13.8: From CQs to SPJ Queries

Consider the CQ q defined as

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

It is easy to verify that

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

14

Homomorphisms and Expressiveness

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

CQ Evaluation and Homomorphisms

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

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

we define the set of relational atoms

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

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

Definition 14.1: Homomorphisms from CQs to Databases

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

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

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

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

Theorem 14.2

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

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

Here is a simple example that illustrates the above characterization.

Example 14.3: CQ Evaluation via Homomorphisms

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

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

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

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

It is clear that the following facts belong to D:

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

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

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

Preservation Results for CQs

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

Preservation under Homomorphisms

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

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

Example 14.4: Homomorphisms Among Databases

Consider the databases

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

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

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

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

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

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

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

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

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

We can now define the notion of preservation under homomorphisms.

Definition 14.5: Preservation under Homomorphisms

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

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

We then show the following for CQs.

Proposition 14.6

Every CQ is preserved under homomorphisms.

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

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

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

We show that homomorphism preservation implies monotonicity of CQs.

Corollary 14.7

Every CQ is monotone.

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

Preservation under Direct Products

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

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

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

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

We proceed to define the notion of preservation under direct products. We do this for Boolean queries without constants, as this suffices to understand the limitations of CQs. Exercises 2.5 and 2.6 explain how these results can be extended to queries with constants and free variables, respectively.

Definition 14.8: Preservation under Direct Products

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

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

We then show the following for CQs.

Proposition 14.9

Every Boolean CQ is preserved under direct products.

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

Exercise 2.5). Let q be a Boolean CQ without constants over a schema $\mathbf{S}$, and let D, D' be databases of $\mathbf{S}$ such that $D \models q$ and $D' \models q$. By Theorem 14.2, there are homomorphisms h, g witnessing $q \to D$ and $q \to D'$, respectively. Define now $f(x) = \tau\big(h(x), g(x)\big)$. Assume that $R(u_1, \ldots, u_n)$ is an atom in q. Then $R(h(u_1), \ldots, h(u_n)) \in D$ and $R(g(u_1), \ldots, g(u_n)) \in D'$. Hence,

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

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

Expressiveness of CQs

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

CQs cannot express inequality. This is because CQs with inequalities are not preserved under homomorphisms. Consider, for example, the FO query

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

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

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

where a is a constant. Given $D = \{U(b)\}$ and $D' = \{U(a)\}$, 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.

15

Conjunctive Query Evaluation

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

Theorem 15.1

CQ-Evaluation is NP-complete.

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

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

Problem: Clique

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

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

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

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

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

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

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

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

Corollary 15.2

CQ-Evaluation is in DLogSpace in data complexity.

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

Parameterized Complexity

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

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

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

Background on Parameterized Complexity

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

Definition 15.3: Parameterized Problem

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

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

Example 15.4: Parameterized Clique

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

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

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

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

Problem: p-Clique

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

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

Problem: p-CQ-Evaluation

Input: A CQ $q(\bar{x})$, a database D, and a tuple $\bar{a}$ over $\mathrm{Dom}(D)$
Parameter: $\|q\|$
Output: `true` if $\bar{a} \in q(D)$, and `false` otherwise

Recall that the motivation underlying parameterized complexity is to have a finer notion of complexity that allows us to classify algorithms as practical or impractical. But when an algorithm in the realm of parameterized complexity is considered to be practical? This brings us to *fixed-parameter tractability*.

Definition 15.5: Fixed-Parameter Tractability

Consider a finite alphabet Σ, and a parametarization $\kappa : \Sigma^* \to \mathbb{N}$ of Σ^*. An algorithm A with input alphabet Σ is an *fpt-algorithm with respect to* κ if there exists a computable function $f : \mathbb{N} \to \mathbb{R}_0^+$, and a polynomial $p(\cdot)$ such that, for every $w \in \Sigma^*$, A on input w runs in time

$$O\big(p(|w|) \cdot f(\kappa(w))\big).$$

A parameterized problem (L, κ) is *fixed-parameter tractable* if there is an fpt-algorithm with respect to κ that decides L. We write FPT for the class of all fixed-parameter tractable problems.

In simple words, (L, κ) is fixed-parameter tractable if there is an algorithm that decides whether $w \in L$ in time arbitrarily large in the parameter $\kappa(w)$, but polynomial in the size of the input w. This reflects the assumption that $\kappa(w)$ is much smaller than $|w|$, and thus, an algorithm that runs, e.g., in time $O(|w| \cdot 2^{\kappa(w)})$ is preferable than one that runs in time $O(|w|^{\kappa(w)})$.

Whenever we deal with an intractable problem, e.g., the problem of concern of this chapter, i.e., CQ-Evaluation, it would be ideal to be able to show that its parameterized version is in FPT. The reader may be tempted to think that p-CQ-Evaluation is in FPT, and that this can be easily shown by exploiting the algorithm for proving that CQ-Evaluation is in NP. It turns out that this is not true. Consider a CQ $q(\bar{x})$, a database D, and a tuple $\bar{a}$ over $\mathrm{Dom}(D)$.

To check if $\bar{a} \in q(D)$, we can iterate over all functions $h : \mathrm{Dom}(S_q) \to \mathrm{Dom}(D)$ until we find one that is a homomorphism from $(S_q, \bar{x})$ to $(D, \bar{a})$, in which case we return `true`; otherwise, we return `false`. Since there are $|\mathrm{Dom}(D)|^{|\mathrm{Dom}(S_q)|}$ such functions, we conclude that this algorithm runs in time

$$O\big(\|D\|^{\|q\|} \cdot r(\|D\| + \|q\|)\big)$$

for some polynomial $r(\cdot)$; note that the size of $\bar{a}$ is not included in the bound since it is polynomially bounded by $\|D\|$ and $\|q\|$. Therefore, we cannot conclude that p-CQ-Evaluation is in FPT since the expression that describes the running time of the above algorithm is not of the form $O(p(\|D\|) \cdot f(\|q\|))$, for some polynomial $p(\cdot)$ and computable function $f : \mathbb{N} \to \mathbb{R}_0^+$, as required by fixed-parameter tractability in Definition 15.5.

It is widely believed that there is no fpt-algorithm that decides the parameterized version of CQ-Evaluation. But then the natural question that comes up is the following: how can we prove that a parameterized problem is not in FPT? Several complexity classes have been defined in the context of parameterized complexity in order to prove that a parameterized problem is not in FPT. Such classes are widely believed to properly contain FPT. This means that if a parameterized problem is complete for one of those classes, then this is a strong indication that the problem in question is not in FPT. Notice here the analogy with classes such as NP and PSpace: it is not known whether these classes properly contain Ptime, but if a problem is complete for any of them, then this is considered as a strong evidence that the problem is not tractable. We proceed to define one of such classes, namely W[1], which will allow us to pinpoint the exact complexity of p-CQ-Evaluation.

To define the class W[1], we need to introduce some auxiliary terminology. Consider a schema $\mathbf{S}$. Let X be a relation name of arity $m \geq 0$ that does not belong to $\mathbf{S}$, and φ an FO sentence over $\mathbf{S} \cup \{X\}$. For a database D of $\mathbf{S}$, and a relation $S \subseteq \mathrm{Dom}(D)^m$, we write $D \models \varphi(S)$ to indicate that $D' \models \varphi$, where $D' = D \cup \{X(\bar{a}) \mid \bar{a} \in S\}$. We further define the problem p-WD$_\varphi$ as follows:

Problem: p-WD$_\varphi$

Input:	A database D of the schema $\mathbf{S}$, and $k \in \mathbb{N}$
Parameter:	k
Output:	`true` if there exists $S \subseteq \mathrm{Dom}(D)^m$ such that $\lvert S\rvert = k$ and $D \models \varphi(S)$, and `false` otherwise

Notice that the sentence φ is fixed in the definition of p-WD$_\varphi$. Therefore, a different FO sentence ψ of the form described above gives rise to a different parameterized problem, dubbed p-WD$_\psi$. The last notion that we need before introducing the class W[1] is that of FPT-reduction.

An FPT-*reduction* from a parameterized problem (L_1, κ_1) over Σ_1 to a parameterized problem (L_2, κ_2) over Σ_2 is a function $\Phi : \Sigma_1^* \to \Sigma_2^*$ such that

the following holds: there are computable functions $f, g : \mathbb{N} \to \mathbb{R}_0^+$, and a polynomial $p(\cdot)$, such that, for every word $w \in \Sigma_1^*$:

1. $w \in L_1$ if and only if $\Phi(w) \in L_2$,
2. $\Phi(w)$ can be computed in time $p(|w|) \cdot f(\kappa_1(w))$, and
3. $\kappa_2(\Phi(w)) \leq g(\kappa_1(w))$.

The first and the second conditions are natural. The third condition is needed to ensure the crucial property that FPT is closed under FPT-reductions: if there exists an FPT-reduction from (L_1, κ_1) to (L_2, κ_2), and $(L_2, \kappa_2) \in \text{FPT}$, then $(L_1, \kappa_1) \in \text{FPT}$; the proof is left as an exercise.

We now have all the ingredients needed for introducing the class W[1]. Recall that *universal* FO sentences are FO sentences of the form $\forall x_1 \cdots \forall x_n \, \psi$, where ψ is quantifier free and $\text{FV}(\psi) = \{x_1, \ldots, x_n\}$.

Definition 15.6: The Class W[1]

A parameterized problem (L, κ) is in W[1] if there exists a schema $\mathbf{S}$, a relation name X not in $\mathbf{S}$, and a universal FO sentence φ over $\mathbf{S} \cup \{X\}$, such that there exists an FPT-reduction from (L, κ) to p-WD_φ.

To give some intuition about the definition of W[1], we show that p-Clique is in W[1]. We first define a universal FO sentence φ, and then show that there exists an FPT-reduction from p-Clique to p-WD_φ. Assume that $\mathbf{S}$ consists of the relation names Node[1] and Edge[2]. Let also Elem[1] be a relation name not in $\mathbf{S}$. We define the universal FO sentence φ over $\mathbf{S} \cup \{\text{Elem}\}$

$$\forall x \forall y \big((\text{Elem}(x) \wedge \text{Elem}(y) \wedge x \neq y) \rightarrow \text{Edge}(x, y) \big).$$

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

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

The sentence φ checks whether the nodes in the relation Elem form a clique. Thus, G has a clique of size k if and only if there exists $S \subseteq \text{Dom}(D)$ such that $|S| = k$ and $D \models \varphi(S)$. It is also clear that (D, k) can be computed in polynomial time. Therefore, the above reduction from p-Clique to p-WD_φ is an FPT-reduction, which in turn implies that p-Clique $\in$ W[1].

Before we proceed with the parameterized complexity of CQ-Evaluation, let us comment on the nomenclature of W[1]. The class W[1] is the first level of a hierarchy of complexity classes W[t], for each $t \geq 1$; hence the number 1. More specifically, the class W[t] is defined in the same way as the class W[1], but allowing the FO sentence φ in p-WD_φ to be of the form $\forall \bar{x}_1 \exists \bar{x}_2 \cdots Q \bar{x}_t \, \psi$, where ψ is quantifier free, $Q = \exists$ if t is even, and $Q = \forall$ if t is odd. The W-hierarchy is defined as the union of all the classes W[t], that is, $\bigcup_{t \geq 1} \text{W}[t]$.

Parameterized Complexity of CQ-Evaluation

We know that p-Clique is W[1]-complete, which means that p-Clique $\in$ W[1] (this has been shown above), and every parameterized problem in W[1] can be reduced via an FPT-reduction to p-Clique. We also known that FPT $\subseteq$ W[1], and it is widely believed that this inclusion is strict (the status of the question whether FPT $\neq$ W[1] is comparable to that of PTIME $\neq$ NP). Thus, it is unlikely that p-Clique $\in$ FPT (as FPT is closed under FPT-reductions). We use this result to prove that the same holds for p-CQ-Evaluation, thus providing strong evidence that this problem is not fixed-parameter tractable.

Theorem 15.7

p-CQ-Evaluation is W[1]-complete.

Proof. For the lower bound, we show that there exists an FPT-reduction from p-Clique to p-CQ-Evaluation. We use the same reduction as for the lower bound in Theorem 15.1, which we recall here for the sake of readability. Consider an input to p-Clique given by $G = (V, E)$ and $k \geq 1$. The database is

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

and the Boolean CQ is

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

As discussed in the proof of Theorem 15.1, G has a clique of size k if and only if $D \models q$, and D and q can be constructed in polynomial time from G and k. To conclude that this is an FPT-reduction, it remains to show that the third condition in the definition of FPT-reductions holds, i.e., $\|q\| \leq g(k)$ for some computable function $g : \mathbb{N} \to \mathbb{R}_0^+$. It is easy to verify that $\|q\| \leq c \cdot \log k \cdot k^2$ for some constant $c \in \mathbb{R}^+$, and thus, p-CQ-Evaluation is W[1]-hard.

We now focus on the upper bound. For technical clarity, we consider only constant-free Boolean CQs over a schema consisting of a single binary relation name Edge. We leave the prove for the general case, where no restrictions are imposed to the query and its schema, as an exercise.

We first define a universal FO sentence φ, and then show that there exists an FPT-reduction from p-CQ-Evaluation to p-WD$_\varphi$. Consider the schema

$$\mathbf{S} \;=\; \{\mathrm{Const}[1],\, \mathrm{Var}[1],\, \mathrm{Edge}_1[2],\, \mathrm{Edge}_2[2]\}.$$

Consider also the relation name Hom[2] that does not belong to $\mathbf{S}$. We define the universal FO sentence φ over $\mathbf{S} \cup \{\mathrm{Hom}\}$ as follows:

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

We show that there is an FPT-reduction from p-CQ-Evaluation to $\mathsf{p\text{-}WD}_\varphi$. Consider an input to p-CQ-Evaluation given by a constant-free Boolean CQ q over the schema $\{\mathrm{Edge}[2]\}$, and a database D of $\{\mathrm{Edge}[2]\}$. Assuming that $\{x_1, \ldots, x_n\}$ are the variables occurring in q, we define the database D' as

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

Roughly, the relation Const stores the constants occurring in D, the relation Var stores the variables occurring in q, the relation Edge_1 stores the atoms of q, and the relation Edge_2 stores the facts of D. We further define $n = k$, that is, k is the number of variables occurring in q.

With the definitions of D' and k in place, we can now explain the meaning of the FO sentence φ. The first conjunct $\forall x \forall y \forall z \left((\mathrm{Hom}(x,y) \wedge \mathrm{Hom}(x,z)) \rightarrow y = z \right)$ states that Hom represents a function, as only one value can be associated to x. The second conjunct states that Hom maps variables of q to constants of D. Finally, the third conjunct states that Hom represents a homomorphism from q to D. Notice, however, that φ does not impose the restriction that every variable occurring q has to be mapped to a constant of D, as this requires a non-universal FO sentence of the form

$$\forall x \left(\mathrm{Var}(x) \rightarrow \exists y \left(\mathrm{Const}(y) \wedge \mathrm{Hom}(x,y) \right) \right).$$

Instead, the parameter $k = n$ is used to force Hom to map every variable in q to a constant of D, as n is the number of variables occurring in q.

Summing up, $q(D) = \texttt{true}$ if and only if there is $S \subseteq \mathrm{Dom}(D')^2$ with $|S| = k$ and $D' \models \varphi(S)$. It is also clear that D' and k can be constructed from D and q in polynomial time, and $k \leq \|q\|$. Thus, we have provided an FPT-reduction from p-CQ-Evaluation to $\mathsf{p\text{-}WD}_\varphi$, which shows that p-CQ-Evaluation $\in \mathrm{W}[1]$ (for constant-free Boolean CQs over a single binary relation). □

16

Static Analysis and Minimization

We have seen in Chapter 8 that for FO and RA queries, the static analysis tasks of containment and equivalence, which are key ingredients for query optimization, are undecidable. Interestingly, these tasks become decidable for CQs. There is also a way for computing the most efficient equivalent queries, where efficiency here is measured as the number of joins a query has to perform. The goal of this chapter is to analyze the complexity of CQ-Containment and CQ-Equivalence, and discuss how a CQ can be minimized, i.e., compute an equivalent version of it of minimal size.

Recall that in Chapter 8 we have also shown that checking for satisfiability, that is, whether a query has a non-empty output on at least one database, is undecidable for FO and RA queries. For CQs, though, checking for satisfiability is a trivial problem. This is because, given a CQ q, there exists always a database on which q has a non-empty output, and this is the database $S_q^{\downarrow}$, i.e., the grounding of S_q (see Definition 9.3).

Containment

We start with the problem of containment: given two CQs q, q', check whether $q \subseteq q'$, i.e., whether $q(D) \subseteq q'(D)$ for every database D. We show that:

Theorem 16.1

CQ-Containment is NP-complete.

The proof of Theorem 16.1 heavily relies on a fundamental result, known in the literature as the *Homomorphism Theorem*. 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 $(S_{q'}, \bar{x}')$ to $(S_q, \bar{x})$; we also write $q \to q'$ to indicate that $S_q \to S_{q'}$. Recall that S_q and $S_{q'}$ 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 makes it possible to view S as a database. Formally, such a grounding is given by a bijective mapping $\mathsf{G}_S : S \to S^{\downarrow}$ that replaces variables in S by new constants; in particular, $\mathsf{G}_S(S) = S^{\downarrow}$.

Theorem 16.2: 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_{S_q} is a homomorphism, Theorem 14.2 we obtain $\mathsf{G}_{S_q}(\bar{x}) \in q(\mathsf{G}_{S_q}(S_q))$. Since $q \subseteq q'$, we have $\mathsf{G}_{S_q}(\bar{x}) \in q'(\mathsf{G}_{S_q}(S_q))$. Applying Theorem 14.2 again, we conclude that there exists a homomorphism h from $(S_{q'}, \bar{x}')$ to $(\mathsf{G}_{S_q}(S_q), \mathsf{G}_{S_q}(\bar{x}))$. Since G_{S_q} is bijective, $\mathsf{G}_{S_q}^{-1} \circ h$ is a homomorphism from $(S_{q'}, \bar{x}')$ to $(S_q, \bar{x})$, as required.

($\Leftarrow$) Conversely, assume that $(q', \bar{x}') \to (q, \bar{x})$, and let h be a homomorphism from $(S_{q'}, \bar{x}')$ to $(S_q, \bar{x})$. Given a database D, assume that $\bar{a} \in q(D)$. By Theorem 14.2, there exists a homomorphism g from $(S_q, \bar{x})$ to $(D, \bar{a})$. Since homomorphisms compose, $g \circ h$ is a homomorphism from $(S_{q'}, \bar{x}')$ to $(D, \bar{a})$ and, thus, $\bar{a} \in q'(D)$ by Theorem 14.2. Therefore, we have that $q(D) \subseteq q'(D)$, from which we conclude that $q \subseteq q'$. □

The next example shows the usefulness of the Homomorphism Theorem.

Example 16.3: Homomorphism Theorem

Consider again the relational schema from Chapter 3

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

and the CQs over this schema

$$q_1 = \text{Answer}(y_1) \text{ :– } \text{Person}(x_1, y_1, z_1), \text{Profession}(x_1, \text{‘actor’}),$$
$$q_2 = \text{Answer}(y_2) \text{ :– } \text{Person}(x_2, y_2, z_2), \text{Profession}(x_2, w_2).$$

The first query asks for the names of actors, while the second one asks for the names of people with a profession. Observe that q_1 is more specific than q_2, and thus, we expect that q_1 is contained in q_2. This is confirmed by the Homomorphism Theorem since

$$(q_2, y_2) \to (q_1, y_1).$$

This is the case since the function $h : \mathrm{Dom}(S_{q_2}) \to \mathrm{Dom}(S_{q_1})$ 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 (S_{q_2}, y_2) to (S_{q_1}, y_1).

An easy consequence of the Homomorphism Theorem is that the problem CQ-Containment can be reduced to CQ-Evaluation.

Corollary 16.4

Let $q(\bar{x})$ and $q'(\bar{x}')$ be CQs. Then:

$$q \subseteq q' \text{ if and only if } \mathsf{G}_{S_q}(\bar{x}) \in q'(\mathsf{G}_{S_q}(S_q)).$$

Proof. By Theorem 16.2, we conclude that

$$q \subseteq q' \text{ if and only if } (S_{q'}, \bar{x}') \to (S_q, \bar{x}).$$

We can also show that

$$(S_{q'}, \bar{x}') \to (S_q, \bar{x}) \text{ if and only if } (S_{q'}, \bar{x}') \to (\mathsf{G}_{S_q}(S_q), \mathsf{G}_{S_q}(\bar{x})).$$

Indeed, if $(S_{q'}, \bar{x}') \to (S_q, \bar{x})$ is witnessed via h, then we have that $\mathsf{G}_{S_q} \circ h$ is a homomorphism from $(S_{q'}, \bar{x}')$ to $(\mathsf{G}_{S_q}(S_q), \mathsf{G}_{S_q}(\bar{x}))$. Conversely, assuming that $(S_{q'}, \bar{x}') \to (\mathsf{G}_{S_q}(S_q), \mathsf{G}_{S_q}(\bar{x}))$ is witnessed via g, $\mathsf{G}_{S_q}^{-1} \circ g$ is a homomorphism from $(S_{q'}, \bar{x}')$ to $(S_q, \bar{x})$. By Theorem 14.2, we get that

$$(S_{q'}, \bar{x}') \to (\mathsf{G}_{S_q}(S_q), \mathsf{G}_{S_q}(\bar{x})) \text{ if and only if } \mathsf{G}_{S_q}(\bar{x}) \in q'(\mathsf{G}_{S_q}(S_q)).$$

Consequently, we get that $q \subseteq q'$ if and only if $\mathsf{G}_{S_q}(\bar{x}) \in q'(\mathsf{G}_{S_q}(S_q))$. □

By exploiting the Homomorphism Theorem, we can further show that the problem CQ-Evaluation can be reduced to CQ-Containment, i.e., the opposite of what Corollary 16.4 shows. In the proof of Corollary 16.4, we essentially convert the CQ q into a database via the bijective homomorphism G_{S_q}. Now we are going to do the opposite, i.e., convert a database into a CQ. As discussed in Chapter 14, we can convert a database D into a set of relational atoms via the injective function $\mathsf{V}_C : \mathsf{Const} \to \mathsf{Const} \cup \mathsf{Var}$, where C is a finite set of constants. Recall that $\mathsf{V}_C(D)$ is the set of relational atoms obtained from D by replacing constants, except for those in C, with variables. The following corollary, which establishes that CQ-Evaluation can be reduced to CQ-Containment, is stated for Boolean CQs, as this suffices for the purpose of pinpointing the complexity of CQ-Containment, but it can be easily generalized to arbitrary CQs.

Corollary 16.5

Let q be a Boolean CQ, D a database, and q_D the Boolean CQ such that $S_{q_D} = \mathsf{V}_C(D)$, where $C = \mathrm{Dom}(S_q) \cap \mathsf{Const}$. Then:

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

Proof. By Theorem 14.2, we conclude that

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

It is easy to show that

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

Indeed, if $q \to D$ is witnessed via h, then we get that $\mathsf{V}_C \circ h$ is a homomorphism from q to q_D. Conversely, assuming that $q \to q_D$ is witnessed via g, $\mathsf{V}_C^{-1} \circ g$ is a homomorphism from q to D. By Theorem 16.2, we conclude that

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

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

By Theorem 16.1, CQ-Evaluation is in NP, and thus, Corollary 16.4 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 16.1 is Boolean), Corollary 16.5 implies that CQ-Containment is NP-hard. Therefore, CQ-Containment in NP-complete, and Theorem 16.1 follows.

Equivalence

We now focus on the equivalence problem: given two CQs q, q', check whether $q \equiv q'$, i.e., whether $q(D) = q'(D)$ for every database D. We show that:

Theorem 16.6

CQ-Equivalence is NP-complete.

Proof. Concerning the upper bound, it suffices to observe that

$$q \equiv q' \text{ if and only if } q \subseteq q' \text{ and } q' \subseteq q,$$

which implies that CQ-Equivalence is in NP since, by Theorem 16.1, the problem of deciding whether $q \subseteq q'$ and $q' \subseteq q$ is in NP.

Concerning the lower bound, we provide a reduction from CQ-Containment. In fact, CQ-Containment is NP-hard even if we consider Boolean CQs (this is a consequence of the proof of Theorem 16.1). Consider two Boolean CQs

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

We assume that q, q' do not share variables since we can always rename variables without affecting the semantics of a query. Let $q_\cap$ be the Boolean CQ

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

which essentially computes the intersection of q and q'. In other words, for every database D, $q(D) \cap q'(D) = q_\cap(D)$. It is straightforward to see that

$$q \subseteq q' \text{ if and only if } q \equiv q_\cap,$$

which in turn implies that CQ-Equivalence is NP-hard, as needed. □

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 (and thus, with fewer joins to perform) an optimization of q. 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 16.7: 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 $|S_{q'}| \leq |S_{q''}|$.

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)

$$M_q \;=\; \{q' \mid q' \text{ is a CQ over } \mathbf{S} \text{ and } |S_{q'}| \leq |S_q|\}$$

that collects all the CQs over $\mathbf{S}$ (up to variable renaming) with at most $|S_q|$ 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 more clever procedure for computing a minimization of q instead of naively iterating over the exponentially many CQs of M_q?
2. And even if such a refined procedure exists, 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, to find a minimization of a CQ q, we simply need to remove atoms from its body. Moreover, although q may have several minimizations, they are all the same (up to variable renaming). This implies that no matter in which order we remove atoms from the body of q, we will always compute the same minimization of q (up to variable renaming).

Minimization via Atom Removals

Consider a CQ q of the form $\text{Answer}(\bar{x}) \coloneq R_1(\bar{u}_1), \ldots, R_n(\bar{u}_n)$. The CQ q' obtained from q by removing the atom $R_i(\bar{u}_i)$, for some $i \in [n]$, is

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

where $\bar{x}'$ is obtained from $\bar{x}$ by removing every variable that is only mentioned in the atom $R_i(\bar{u}_i)$. For example, if we remove the atom $R(x)$ from the CQ $\text{Answer}(x, y) \coloneq R(x), S(y)$, then we obtain the CQ $\text{Answer}(y) \coloneq S(y)$ as the variable x is only mentioned in $R(x)$. On the other hand, if we remove the atom $R(x)$ from the CQ $\text{Answer}(x, y) \coloneq R(x), T(x, y)$, then we obtain the CQ $\text{Answer}(x, y) \coloneq T(x, y)$ since x occurs also in $T(x, y)$.

The building block of minimization via atom removals is as follows: given a CQ $q(\bar{x})$, construct a CQ $q'(\bar{x})$ by removing an atom $R(\bar{u})$ from the body of q such that $(q, \bar{x}) \to (q', \bar{x})$. Notice that the output tuple $\bar{x}$ remains the same, which means that the atom $R(\bar{u})$ either it does not contain a variable of $\bar{x}$, or it contains only variables of $\bar{x}$ that occur also in atoms of $S_q - \{R(\bar{u})\}$. In this way, we actually construct a CQ that is equivalent to q. Indeed, since $(q, \bar{x}) \to (q', \bar{x})$, we get that $q' \subseteq q$ (by Theorem 16.2). Moreover, $(q', \bar{x}) \to (q, \bar{x})$ holds trivially due to the identity homomorphism from $S_{q'}$ to S_q, and thus, $q \subseteq q'$ (again by Theorem 16.2). 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 16.8: 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. $S_{q'} \subseteq S_q$,
2. $(q, \bar{x}) \to (q', \bar{x})$, and
3. for every CQ $q''(\bar{x})$ with $S_{q''} \subsetneq S_{q'}$, $(q', \bar{x}) \to (q'', \bar{x})$ does not hold.

The first condition in Definition 16.8 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 16.9: 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 $(S_{q_3}, (x, y, z))$ to $(S_{q_3'}, (x, y))$ 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 16.10

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 16.8) is violated, which in turn implies that there is a CQ $q'(\bar{x})$ with $S_{q'} \subsetneq S_q$ such that $(q, \bar{x}) \to (q', \bar{x})$. If q' is a core of itself, then it is clear that q' is a core of q, and the claim follows. 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 16.11. *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 $|S_{q_2}| < |S_{q_1}|$. Then, there is a CQ $q_3(\bar{y}_1)$ such that*

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

Proof. By Theorem 16.2, we conclude that

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

Assume that these statements are witnessed via the homomorphisms h_1 and h_2, respectively. Let $q_3(\bar{y}_3)$ be the CQ such that

$$S_{q_3} = h_2(S_{q_2}) \quad \text{and} \quad \bar{y}_3 = h_2(\bar{y}_2).$$

It is clear that $\bar{y}_3 = \bar{y}_1$ and $S_{q_3} \subseteq S_{q_1}$. Furthermore, since $|S_{q_3}| \leq |S_{q_2}|$ and $|S_{q_2}| < |S_{q_1}|$, we conclude that $|S_{q_3}| < |S_{q_1}|$, and thus, $S_{q_3} \subsetneq S_{q_1}$. It remains to show that $(q_1, \bar{y}_1) \to (q_3, \bar{y}_1)$. Since homomorphisms compose, the latter is witnessed via the homomorphism $h_2 \circ h_1$. □

Consider now a CQ $q'(\bar{x})$ that is a core of $q(\bar{x})$. Towards a contradiction, assume that q' is not a minimization of q. This implies that there exists a CQ q'' such that $q' \equiv q''$ and $|S_{q''}| < |S_{q'}|$. By Lemma 16.11, we conclude that there exists a CQ $q'''(\bar{x})$ such that $(q', \bar{x}) \to (q''', \bar{x})$ and $S_{q'''} \subsetneq S_{q'}$. This contradicts our hypothesis that q' is a core of q, and the claim follows. □

By Proposition 16.10, 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 16.3. 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 16.12. *Given a CQ q,* ComputeCore(q) *is a core of q.*

Proof. At each iteration of the while-loop, the CQ $q'(\bar{x})$ with $S_{q'} = S$ (which is indeed a CQ since, by construction, every variable in $\bar{x}$ occurs in $S_{q'}$) is such that $S_{q'} \subseteq S_q$ and $(q, \bar{x}) \to (q', \bar{x})$. Therefore, the CQ $q^*(\bar{x})$ returned by the algorithm is such that $S_{q^*} \subseteq S_q$ and $(q, \bar{x}) \to (q^*, \bar{x})$. Furthermore, by construction, for every CQ $q''(\bar{x})$ with $S_{q''} \subsetneq S_{q^*}$, $(q^*, \bar{x}) \to (q'', \bar{x})$ does not hold. Therefore, q^* satisfies all the three conditions given in the definition of core (Definition 16.8), and thus, it is a core of q, as needed. □

Algorithm 16.3 ComputeCore(q)

Input: A CQ $q(\bar{x})$
Output: A query $q^*(\bar{x})$ that is a core of $q(\bar{x})$

1: $S := S_q$
2: **while** there exists $R(\bar{y}) \in S$ such that each variable in $\bar{x}$
3: occurs in $\mathrm{Dom}(S - \{R(\bar{y})\})$ and $(S, \bar{x}) \to (S - \{R(\bar{y})\}, \bar{x})$ **do**
4: $S := S - \{R(\bar{y})\}$
5: **return** $q^*(\bar{x}) \coloneq R_1(\bar{u}_1), \ldots, R_n(\bar{u}_n)$, where $S = \{R_1(\bar{u}_1), \ldots, R_n(\bar{u}_n)\}$

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 16.9 (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 $\rho : \mathrm{Dom}(S_q) \to \mathrm{Dom}(S_{q'})$ that is a homomorphism from $(S_q, \bar{x})$ to $(S_{q'}, \bar{x}')$, and its inverse ρ^{-1} is a homomorphism from $(S_{q'}, \bar{x}')$ to $(S_q, \bar{x})$. (Recall from Chapter 9 that homomorphisms between sets of atoms are always the identity on constants.)

Proposition 16.13

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 $\rho : \mathrm{Dom}(S_{q'}) \to \mathrm{Dom}(S_{q''})$ that is a homomorphism from $(S_{q'}, \bar{x}')$ to $(S_{q''}, \bar{x}'')$, and its inverse ρ^{-1} is a homomorphism from $(S_{q''}, \bar{x}'')$ to $(S_{q'}, \bar{x}')$. Since both q' and q'' are minimizations of q, we get that $q \equiv q'$ and $q \equiv q''$, and thus, $q' \equiv q''$. By Theorem 16.2,

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

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 16.14. *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 $z \in \mathrm{Dom}(S_{q''})$ such that there is no variable $y \in \mathrm{Dom}(S_{q'})$ with $h(y) = z$. Let $R(\bar{u}) \in S_{q''}$ be an atom that mentions z. We have that $R(\bar{u}) \notin h(S_{q'})$. We define $q'''(\bar{x}'')$ as the CQ with $S_{q'''} = h(S_{q'})$. It is clear that $(q', \bar{x}') \to (q''', \bar{x}'')$ via h, and $(q''', \bar{x}'') \to (q', \bar{x}')$ via g. Therefore, by Theorem 16.2, $q' \equiv q'''$. Since $q' \equiv q''$, we conclude that $q'' \equiv q'''$. Observe also that $S_{q'''} \subsetneq S_{q''}$, which implies that $|S_{q'''}| < |S_{q''}|$. 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 $y, z \in \mathrm{Dom}(S_{q'})$ 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 $u \in \mathrm{Dom}(S_{q'})$ such that there is no variable $v \in \mathrm{Dom}(S_{q'})$ with $g(h(v)) = u$. Let $R(\bar{u}) \in S_{q'}$ be an atom that mentions u. We have that $R(\bar{u}) \notin g(h(S_{q'}))$. We define $q'''(\bar{x}')$ as the CQ with $S_{q'''} = g(h(S_{q'}))$. It is clear that $(q', \bar{x}') \to (q''', \bar{x}')$ via $g \circ h$. Observe also that $S_{q'''} \subsetneq S_{q'}$. Hence, $(q''', \bar{x}') \to (q', \bar{x}')$ via the identity homomorphism, which means that $q' \equiv q'''$ due to Theorem 16.2, and $|S_{q'''}| < |S_{q'}|$. 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 $\rho : \mathrm{Dom}(S_{q'}) \to \mathrm{Dom}(S_{q''})$. Let $f = g \circ h$. It is clear that f is a homomorphism from $(S_{q'}, \bar{x}')$ to $(S_{q'}, \bar{x}')$. Since, by Lemma 16.14, 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 $(S_{q'}, \bar{x}')$ to $(S_{q'}, \bar{x}')$. 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 $(S_{q'}, \bar{x}')$ to $(S_{q''}, \bar{x}'')$. 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 $(S_{q''}, \bar{x}'')$ to $(S_{q'}, \bar{x}')$. Therefore, ρ witnesses the fact that q' and q'' are isomorphic, and the claim follows. □

From Proposition 16.10, which tells us that a core of CQ q is a minimization, and Proposition 16.13, we immediately get the following corollary:

Corollary 16.15

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 16.15 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 14.2, we have that

$$(q_1, (x_1, y_1)) \rightarrow (D', (a, b))$$

via a homomorphism h_1. Since $D' \models \sigma$ and

$$\{R(h_1(x_1), h_1(y_1)), R(h_1(x_1), h_1(z_1))\} \subseteq D',$$

it holds that $h_1(y_1) = h_1(z_1)$. Since $R(h_1(y_1), h_1(z_1)) \in D'$, we get that

$$(q_2, (x_2, y_2)) \rightarrow (D', (a, b))$$

via h_2 such that $h_2(x_2) = h_1(x_1)$ and $h_2(y_2) = h_1(y_1) = h_1(z_1)$.

Our goal is to revisit the problem of containment for CQs in the presence of FDs. More precisely, given two CQs q and q', and a set Σ of FDs, we say that q *is contained in* q' *under* Σ, denoted by $q \subseteq_\Sigma q'$, if for every database D that satisfies Σ, it holds that $q(D) \subseteq q'(D)$. The problem of interest follows:

Problem: CQ-Containment-FD

Input: Two CQs q and q', and a set Σ of FDs
Output: `true` if $q \subseteq_\Sigma q'$, and `false` otherwise

We proceed to show the following result:

Theorem 17.2

CQ-Containment-FD is NP-complete.

It is clear that the NP-hardness is inherited from CQ containment without constraints (see Theorem 16.1). 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 S_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 $(S_{q'}, \bar{x}') \rightarrow (S, \bar{u})$. This simple idea has been already

illustrated by Example 17.1. Unsurprisingly, the transformation of S_q and $\bar{x}$ into S and $\bar{u}$, respectively, can be done by exploiting the chase for FDs, which has been introduced in Chapter 10. For brevity, we simply write $\text{Chase}(q, \Sigma)$ instead of $\text{Chase}(S_q, \Sigma)$, and $h_{q,\Sigma}$ instead of $h_{S_q,\Sigma}$. 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. $\text{Chase}(q, \Sigma) \neq \bot$ implies $(S_{q'}, \bar{x}') \to (\text{Chase}(q, \Sigma), h_{q,\Sigma}(\bar{x}))$.

Proof. For brevity, let $S = \text{Chase}(q, \Sigma)$ and $\bar{u} = h_{q,\Sigma}(\bar{x})$.

We first show that (1) implies (2). By hypothesis, $q \subseteq_\Sigma q'$. It is clear that, if $S \neq \bot$, then $\mathsf{G}_S(\bar{u}) \in q(\mathsf{G}_S(S))$. Since, by Lemma 10.7, $S \models \Sigma$, which means that $\mathsf{G}_S(S) \models \Sigma$, we have that $\mathsf{G}_S(\bar{u}) \in q'(\mathsf{G}_S(S))$. By Theorem 14.2, there exists a homomorphism h from $(S_{q'}, \bar{x}')$ to $(\mathsf{G}_S(S), \mathsf{G}_S(\bar{u}))$. Clearly, $\mathsf{G}_S^{-1} \circ h$ is a homomorphism from $(S_{q'}, \bar{x}')$ 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 $(S_{q'}, \bar{x}') \to (S, \bar{u})$ via a homomorphism h. Let D be an arbitrary database of $\mathbf{S}$ such that $D \models \Sigma$, and assume that $\bar{a} \in q(D)$. By Theorem 14.2, $(q, \bar{x}) \to (D, \bar{a})$. Since $D \models \Sigma$, Lemma 10.10 implies that $(S, \bar{u}) \to (D, \bar{a})$ via a homomorphism g. Since homomorphisms compose, $g \circ h$ is a homomorphism from $(q', \bar{x}')$ to $(D, \bar{a})$. By Theorem 14.2, $\bar{a} \in q'(D)$, which implies that $q \subseteq_\Sigma q'$.

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

The following is an easy consequence of Theorem 17.3 and Theorem 14.2.

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 $S = \text{Chase}(q, \Sigma)$, the following are equivalent:

1. $q \subseteq_\Sigma q'$.

2. $S \neq \bot$ implies $\mathsf{G}_S(h_{q,\Sigma}(\bar{x})) \in q'(\mathsf{G}_S(S))$.

By Lemma 10.9, $\text{Chase}(q, \Sigma)$ can be computed in polynomial time. Moreover, if $\text{Chase}(q, \Sigma) \neq \bot$, then the chase homomorphism $h_{q,\Sigma}$ can be also computed in polynomial time. Since CQ-Evaluation is in NP (see Theorem 15.1), we conclude that CQ-Containment-FD is also in NP, and Theorem 17.2 follows.

Inclusion Dependencies

We now concentrate 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

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

respectively. It is clear that $(q_2, (x_2, y_2)) \to (q_1, (x_1, y_1))$ does not hold, and thus, we have that $q_1 \not\subseteq q_2$ by the Homomorphism Theorem. Suppose now that q_1 and q_2 will be evaluated only over databases that satisfy

$$\sigma_1 = R[1, 2] \subseteq S[1, 2] \quad \text{and} \quad \sigma_2 = S[2, 3] \subseteq R[1, 2].$$

We can show that, for every database D,

$$D \models \{\sigma_1, \sigma_2\} \text{ implies } q_1(D) \subseteq q_2(D).$$

Consider an arbitrary database D that satisfies $\{\sigma_1, \sigma_2\}$, and assume that $(a, b) \in q_1(D)$, or, equivalently, $(q_1, (x_1, y_1)) \to (D, (a, b))$ via a homomorphism h_1. This implies that $R(h_1(x_1), h_1(y_1)) \in D$. Since $D \models \sigma_1$, we get that D contains an atom of the form $S(h_1(x_1), h(y_1), c)$. But since $D \models \sigma_2$, we also get that D contains the atom $R(h_1(y_1), c)$. Hence,

$$\{R(h_1(x_1), h_1(y_1)), R(h_1(y_1), c), S(h_1(x_1), h_1(y_1), c)\} \subseteq D.$$

This implies that $(q_1', (x_1, y_1)) \to (D, (a, b))$, where q_1' is obtained from q_1 by adding certain atoms according to σ_1 and σ_2, i.e., q_1' is defined as

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

where w_1 is a new variable not in q_1. Now observe that $(q_2, (x_2, y_2)) \to (q_1', (x_1, y_1))$, which implies that $(q_2, (x_2, y_2)) \to (D, (a, b))$. By the Homomorphism Theorem, $(a, b) \in q_2(D)$, and thus, $q_1(D) \subseteq q_2(D)$.

Our goal is to revisit the problem of CQ containment in the presence of INDs. Given two CQs q and q', and a set Σ of INDs, q *is contained in* q' *under* Σ, denoted $q \subseteq_\Sigma q'$, if for every database D that satisfies Σ, $q(D) \subseteq q'(D)$. The problem of interest is defined as expected:

Problem: CQ-Containment-IND

Input: Two CQs q and q', and a set Σ of INDs
Output: `true` if $q \subseteq_\Sigma q'$, and `false` otherwise

Although the complexity of CQ containment in the presence of FDs remains NP-complete (Theorem 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 S_q occurring in q into a new set of atoms S that satisfies the INDs, and then check whether $(S_{q'}, \bar{x}') \to (S, \bar{x})$. This simple idea has been already illustrated by Example 17.5. As expected, the transformation of S_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 13.3), we proceed to give the formal definition for the sake of completeness.

Consider a possibly infinite database D and a CQ q of the form

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

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

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

where, for $i \in [n]$, $R_i(\eta(\bar{u}_i))$ is the fact obtained after replacing each variable x in $\bar{u}_i$ with $\eta(x)$, and leave the constants in $\bar{u}_i$ untouched. Having this notion, we can define what is the output of a CQ on a possibly infinite database.

Definition 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 14.1 merely extends to possibly infinite databases, which allows us to state a result analogous to Theorem 14.2:

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 $\mathrm{Chase}(S_q, \Sigma)$. 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:

$$q \subseteq^{\infty}_{\Sigma} q' \quad \text{if and only if} \quad (S_{q'}, \bar{x}') \to (\mathrm{Chase}(q, \Sigma), \bar{x}).$$

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:

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

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 $\sigma = R[i_1, \ldots, i_m] \subseteq P[j_1, \ldots, j_m]$, a tuple $\bar{u} = (u_1, \ldots, u_{\text{ar}(R)})$, and a set of variables V, $\mathsf{new}^V(\sigma, \bar{u})$ is the atom obtained from $\mathsf{new}(\sigma, \bar{u})$ after replacing each newly introduced variable with a distinct variable from V. Formally, $\mathsf{new}^V(\sigma, \bar{u}) = P(v_1, \ldots, v_{\text{ar}(P)})$, where, for each $\ell \in [\text{ar}(P)]$,

$$v_\ell = \begin{cases} u_{i_k} & \text{if } \ell = j_k, \text{ for } k \in [m], \\ x \in V & \text{otherwise,} \end{cases}$$

such that, for each $i, j \in [\text{ar}(P)] - \{j_1, \ldots, j_m\}$, $i \neq j$ implies $v_i \neq v_j$.[1] Given two CQs $q(\bar{x}), q'(\bar{x}')$ over a schema $\mathbf{S}$, and a set Σ of INDs over $\mathbf{S}$, *a witness of q' relative to q and Σ* is a triple $(\mathcal{V}, \mathcal{S}, Q)$, where $\mathcal{V}$ is a sequence of (not necessarily disjoint) sets of variables $V_1, \ldots, V_n$, for $n \geq 0$, $\mathcal{S}$ is a sequence of disjoint sets of relational atoms $S_0, \ldots, S_n$, and $Q \subseteq \bigcup_{i \in [0,n]} S_i$, such that:

- $|\bigcup_{i \in [n]} V_i| \leq 3 \cdot |S_{q'}| \cdot \max_{R \in \mathbf{S}}\{\text{ar}(R)\}$,
- for each $i \in [n]$, $V_i \cap (\text{Dom}(S_{i-1}) \cup \text{Dom}(S)) = \emptyset$,
- for each $i \in [0, n]$, $|S_i| \leq |S_{q'}|$,
- $S_0 \subseteq S_q$,
- 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,
- $|Q| \leq |S_{q'}|$, and
- $\mathsf{G}_Q(\bar{x}) \in q'(\mathsf{G}_Q(Q))$.

[1] We assume some fixed mechanism that chooses the variable v_ℓ from the set V whenever $\ell \in [\text{ar}(P)] - \{j_1, \ldots, j_m\}$.

Let $S = \text{Chase}(q, \Sigma)$. Notice that $\mathsf{G}_S(\bar{x}) \in q'(\mathsf{G}_S(S))$ holds due to the existence of a set $A \subseteq \text{Chase}(q, \Sigma)$ such that $(S_{q'}, \bar{x}') \to (A, \bar{x})$. It is also not difficult to see that the construction of A can be witnessed via a sequence $A_0, A_1, \ldots, A_n$ of disjoint subsets of $\text{Chase}(q, \Sigma)$, where each such set consists of at most $|S_{q'}|$ atoms, $A_0 \subseteq S_{q'}$, $A_n = A$, 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 17.4 ContainmentWitness(q, q', Σ)

Input: Two CQs $q(\bar{x})$ and $q'(\bar{x}')$ over $\mathbf{S}$, and a set Σ of INDs over $\mathbf{S}$.
Output: `true` if there is a witness for q' relative to q and Σ, and `false` otherwise.

$S_\triangledown := A$, where $A \subseteq S_q$ with $|A| \leq |S_{q'}|$
$S_\triangleright := \emptyset$
$Q := A$, where $A \subseteq S_\triangledown$
$V := \{y_1, \ldots, y_m\} \subset \mathsf{Var} - \text{Dom}(S_q)$ for some $m \in [3 \cdot |S_{q'}| \cdot \max_{R \in \mathbf{S}}\{\text{ar}(R)\}]$
repeat
    repeat
        if $\sigma = R[\alpha] \subseteq P[\beta] \in \Sigma$ is applicable on $S_\triangledown$ with $\bar{u} \in \text{Dom}(S_\triangledown)^{\text{ar}(R)}$ **then**
            $N := \mathsf{new}^V(\sigma, \bar{u})$
            $V := V - \text{Dom}(\{N\})$
            $S_\triangleright := S_\triangleright \cup \{N\}$
        if $|S_\triangleright| < |S_{q'}|$ **then**
            Next $:= b$, where $b \in \{0, 1\}$
        else
            Next $:= 1$
    until *Next* $= 1$
    if $S_\triangleright = \emptyset$ **then**
        return `false`
    $V := V \cup ((\text{Dom}(S_\triangledown) \cap \mathsf{Var}) - (\text{Dom}(S_\triangleright) \cup \text{Dom}(Q)))$
    $S_\triangledown := S_\triangleright$
    $S_\triangleright := \emptyset$
    $Q := Q \cup A$, where $A \subseteq S_\triangledown$
    if $|Q| < |S_{q'}|$ **then**
        Evaluate $:= b$, where $b \in \{0, 1\}$
    else
        Evaluate $:= 1$
until *Evaluate* $= 1$
return $\mathsf{G}_Q(\bar{x}) \in q'(\mathsf{G}_Q(Q))$

Lemma 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 $S = \text{Chase}(q, \Sigma)$, it holds that $\mathsf{G}_S(\bar{x}) \in q'(\mathsf{G}_S(S))$ if and only if there exists a witness of q' relative to q and Σ.*

By Corollary 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 17.4. 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 17.4:

Initialization. The algorithm starts by guessing a subset of S_q with at most $|S_{q'}|$ atoms, which is stored in $S_\triangledown$ (see line 1); $S_\triangledown$ should be seen as the "current set" from which we construct the "next set" $S_\triangleright$ in the sequence. It also guesses a subset of $S_\triangledown$ 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 $3 \cdot |S_{q'}| \cdot \max_{R \in \mathbf{S}}\{\text{ar}(R)\}$ variables not occurring in S_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 $S_\triangleright$ from $S_\triangledown$. This is done by guessing an IND $\sigma \in \Sigma$ and a tuple $\bar{u}$ over $\text{Dom}(S_\triangledown)$, and adding to $S_\triangleright$ the atom $\mathsf{new}^V(\sigma, \bar{u})$ if σ is applicable on the current set $S_\triangledown$ 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 $S_\triangleright$ that will be generated by a subsequent iteration. This is repeated until $S_\triangleright$ contains exactly $|S_{q'}|$ 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 $|S_{q'}|$ atoms, by setting *Next* to 1. Once $S_\triangleright$ is in place, the algorithm updates V by adding to it the variables that occur in the current set $S_\triangledown$, but have not been propagated to $S_\triangleright$ 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 $S_\triangledown$ is not needed further, and we can reuse the space that it occupies. The set $S_\triangleright$ becomes the current set $S_\triangledown$ (see line 19), while $S_\triangleright$ becomes empty (see line 20). Then the algorithm guesses a subset of $S_\triangledown$ 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 $|S_{q'}|$ atoms (in the worst-case, $2 \cdot |S_{q'}|$ 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 $|S_{q'}|$ 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 17.4 uses polynomial space, which is actually the space needed to represent the sets $S_\triangledown$, $S_\triangleright$, Q and V, as well as the space needed to check whether an IND is applicable on $S_\triangledown$ with some tuple $\bar{u} \in \mathrm{Dom}(S_\triangledown)^{\mathrm{ar}(R)}$ (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 11.8). Recall that the IND-Implication problem takes as input a set Σ of INDs over a schema $\mathbf{S}$, and an IND σ over $\mathbf{S}$, and asks whether $\Sigma \models \sigma$, i.e., whether for every database over $\mathbf{S}$, $D \models \Sigma$ implies $D \models \sigma$. We are going to construct two CQs q and q' such that $\Sigma \models \sigma$ if and only if $q \subseteq_\Sigma q'$.

Assume that $\sigma = R[i_1, \ldots, i_k] \subseteq P[j_1, \ldots, j_k]$. The CQ q is defined as

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

while the CQ q' is defined as

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

where, for each $m \in [\mathrm{ar}(P)]$,

$$f(m) \ = \ \begin{cases} i_\ell & \text{if } m = j_\ell, \text{ where } \ell \in [k], \\ \mathrm{ar}(R) + m & \text{otherwise.} \end{cases}$$

The function f ensures that the variable at position j_ℓ in the P-atom of q' is x_{i_ℓ}, i.e., the same as the one at position i_ℓ in the R-atom of q', while all the variables in the P-atom occurring at a position not in $\{j_1, \ldots, j_k\}$ are new variables occurring only once in the P-atom, and not occurring in the R-atom. It is an easy exercise to show that indeed $\Sigma \models \sigma$ if and only if $q \subseteq_\Sigma q'$.

18

Exercises for Part II

Exercise 2.1. Let q be the CQ given in Example 13.8. Express q as an RA query using θ-joins instead of Cartesian product.

Exercise 2.2. Prove the correctness of the translation of a CQ into an SPJ query, and the translation of an SPJ query into a CQ, given in the proof of Theorem 13.7, which establishes that the languages of CQs and of SPJ queries are equally expressive.

Exercise 2.3. For a CQ q, let e_q be the equivalent SPJ query obtained by applying the translation in the proof of Theorem 13.7. What is the size of e_q with respect to the size of q? Conversely, assuming that q_e is the CQ obtained after translating an SPJ query e into a CQ according to the translation in the proof of Theorem 13.7, what is the size of q_e with respect to the size of e?

Exercise 2.4. Prove that the choice of a pairing function in the definition of direct product does not matter. More precisely, let $\otimes_\tau$ be the direct product defined using a pairing function τ. Then, for every Boolean FO query q, every two databases D and D', and every two pairing functions τ and τ', show that $D \otimes_\tau D' \models q$ if and only if $D \otimes_{\tau'} D' \models q$.

Exercise 2.5. 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 2.6. 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 2.5.

Then, given two tuples $\bar{a} = (a_1, \ldots, a_n)$ and $\bar{b} = (b_1, \ldots, b_n)$, define the n-ary tuple $\bar{a} \otimes \bar{b}$ as $\big(\tau_C(a_1, b_1), \ldots, \tau_C(a_n, b_n)\big)$. Consider now an n-ary CQ $q(\bar{x})$ that mentions only constants from C. Show that if $\bar{a} \in q(D)$ and $\bar{b} \in q(D')$, then $\bar{a} \otimes \bar{b} \in q(D \otimes D')$, where $\otimes$ is defined with the pairing function τ_C.

Exercise 2.7. Use Exercise 2.5 to prove that the Boolean query $q = \exists x\,(x = a)$, where a is a constant, cannot be expressed as a CQ.

Exercise 2.8. Use Exercise 2.6 to prove that the query $q = \varphi(x, y)$, where φ is the equational atom $(x = y)$, cannot be expressed as a CQ.

Exercise 2.9. Consider a parameterized problem (L_1, κ_1) over Σ_1, and a parameterized problem (L_2, κ_2) over Σ_2. Show that if there is an FPT-reduction from (L_1, κ_1) to (L_2, κ_2), and $(L_2, \kappa_2) \in \mathrm{FPT}$, then $(L_1, \kappa_1) \in \mathrm{FPT}$.

Exercise 2.10. Recall that in the proof of the fact that p-CQ-Evaluation is in W[1] (see Theorem 15.7), for technical clarity, we consider only constant-free Boolean CQs over a schema consisting of a single binary relation name. Prove that p-CQ-Evaluation is in W[1] even for arbitrary CQs.

Exercise 2.11. Show Corollary 16.5 for arbitrary (non-Boolean) CQs.

Exercise 2.12. 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 2.13. Answer the following questions about CQs and their cores.

(i) Consider the Boolean CQ q_1 over the schema $\{E[2]\}$ defined as

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

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

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

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

Compute the core of q_2.

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

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

Prove that q_3 is a core of itself.

Exercise 2.14. 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 $\mathrm{Dom}(S_{q'})$.

Exercise 2.15. Recall that ComputeCore (see Algorithm 16.3) 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 2.16. Let D be a database, and $T = \{\bar{a}_1, \ldots, \bar{a}_n\}$ a set of m-ary tuples over $\mathrm{Dom}(D)$, 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 $\bar{b} \in \mathrm{Dom}(D)^m - T$ such that $\prod_{i\in[n]}(D, \bar{a}_i) \to (D, \bar{b})$.

Exercise 2.17. 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 $\mathrm{Dom}(D)$, for $m > 0$, check whether there exists a CQ q such that $q(D) = T$.

Exercise 2.18. Prove that FO-Containment remains undecidable even if one of the two input queries is a CQ.

Exercise 2.19. Prove Lemma 17.12.

Exercise 2.20. 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.

References

1. S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.
2. S. Arora and B. Barak. *Computational Complexity - A Modern Approach*. Cambridge University Press, 2009.
3. H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database Systems - the Complete Book*. Pearson Education, 2009.
4. E. Grädel, P. Kolaitis, L. Libkin, M. Marx, J. Spencer, M. Vardi, and S. Weinstein. *Finite Model Theory and its Applications*. Springer, 2008.
5. P. R. Halmos. *Measure Theory*. Springer, 1974.
6. J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 2003.
7. N. Immerman. *Descriptive Complexity*. Springer, 1999.
8. D. Kozen. *Automata and Computability*. Springer, 1997.
9. L. Libkin. *Elements of Finite Model Theory*. Springer, 2004.
10. C. H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.
11. R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw-Hill, 2003.
12. K. H. Rosen. *Discrete Mathematics and its Applications*. McGraw-Hill, 2006.
13. A. Silberschatz, H. F. Korth, and S. Sudarshan. *Database System Concepts*. McGraw-Hill Book Company, 2005.
14. M. Sipser. *Introduction to the Theory of Computation*. PWS Publishing Company, 1997.
15. I. Wegener. *Complexity Theory*. Springer, 2005.

Part X

Appendix: Theory of Computation

A

Big-O Notation

We write $\mathbb{R}_0^+$ for the set of non-negative real numbers, and $\mathbb{R}^+$ for the set of positive real numbers. We typically measure the performance of an algorithm, that is, the number of basic operations it performs, as a function of its input length. In other words, the performance of an algorithm can be captured by a function $f : \mathbb{N} \to \mathbb{R}_0^+$ such that $f(n)$ is the maximum number of basic operations that the algorithm performs on inputs of length n. However, since f may heavily depend on the details of the definition of basic operations, we usually concentrate on the overall and asymptotic behaviour of the algorithm. This is achieved via the well-known notion of *big-O notation*.

The big-O notation is typically defined for single variable functions such as f above. However, in the database setting, where the input to key problems usually consists of several different components, we generally have to deal with multiple variable functions. For example, the performance of a query evaluation algorithm, where the input consists of two distinct components, the *database* and the *query*, can be captured by a function $f : \mathbb{N}^2 \to \mathbb{R}_0^+$ such that $f(n, m)$ is the maximum number of basic operations that the algorithm performs on databases of size n and queries of size m. The notion of big-O notation for multiple variable functions follows:

Definition 1.1: Big-O Notation

Let $f, g : \mathbb{N}^\ell \to \mathbb{R}_0^+$, where $\ell \geq 1$. We say that

$$f(x_1, \ldots, x_\ell) \text{ is in } O(g(x_1, \ldots, x_\ell))$$

if there exist $k \in \mathbb{R}^+$ and $n_0 \in \mathbb{N}$ such that, for every $(x_1, \ldots, x_\ell)$ with $x_i \geq n_0$ for some $i \in [\ell]$, $f(x_1, \ldots, x_\ell) \leq k \cdot g(x_1, \ldots, x_\ell)$.

Notice that when $\ell = 1$, i.e., f, g are single variables function, Definition 1.1 coincides with the standard big-O notation for single variable functions.

B

Turing Machines and Complexity Classes

Many results in this book will provide bounds on computational resources (time and space), or key database problems such as query evaluation. These are often formulated in terms of membership in, or completeness for, complexity classes. Those, in turn, are defined using the basic model of computation, that is, Turing Machines. We now briefly recall basic concepts related to Turing Machines and complexity classes. For more details, the reader can consult standard textbooks on computability theory and computational complexity.

Turing Machines

Turing Machines can work in two modes: either as *acceptors*, for deciding whether an input string belongs to a given language (in which case we speak of *decision problems*), or as computational devices that compute the value of a function applied to its input. When a Turing Machine works as an acceptor, it typically contains a read-write tape, a model of computation that is convenient for defining time complexity classes, or a read-only input tape and a read-write working tape, a model that is convenient for defining space complexity classes. When a Turing Machine works as a computational device, it typically contains a read-only tape where the input is placed, a read-write working tape, and a write-only tape where the output computed by the Turing Machine is placed.

Turing Machines as Acceptors

We start with the definition of deterministic Turing Machines.

Definition 2.1: Deterministic Turing Machine

A *(deterministic) Turing Machine* (TM) is defined as a tuple

$$M \;=\; (Q, \Sigma, \delta, s)\,,$$

where

- Q is a finite set of states, including the *accepting state* "yes", and the *rejecting state* "no",
- Σ is a finite set of input symbols, called the *alphabet* of M, including the symbols $\sqcup$ (the *blank symbol*) and $\triangleright$ (the *left marker*),
- $\delta : (Q - \{\text{"yes"}, \text{"no"}\}) \times \Sigma \to Q \times \Sigma \times \{\rightarrow, \leftarrow, -\}$ is the *transition function* of M, and
- $s \in Q$ is the *start state* of M.

Accepting and rejecting states are needed for decision problems: they determine whether the input belongs to the language or not. Notice that, according to δ, the accepting and rejecting states do not have outgoing transitions.

A *configuration* of a TM $M = (Q, \Sigma, \delta, s)$ is a tuple

$$c \;=\; (q, u, v)\,,$$

where $q \in Q$, and u, v are words in Σ^* with u being always non-empty. If M is in configuration c, then the tape has content uv and the head is reading the last symbol of u. We use left markers, which means that u always starts with $\triangleright$. Moreover, the transition function δ is restricted in such a way that $\triangleright$ occurs exactly once in uv, and always as the first symbol of u.

Assume now that M is in a configuration $c = (q, ua, v)$, where $q \in Q - \{\text{"yes"}, \text{"no"}\}$, $a \in \Sigma$ and $u, v \in \Sigma^*$, and assume that $\delta(q, a) = (q', b, \text{dir})$, where $\text{dir} \in \{\rightarrow, \leftarrow, -\}$. Then, in one step, M enters the configuration $c' = (q', u', v')$, where u', v' is obtained from ua, v by replacing a with b and moving the head one step in the direction dir. By moving the head in the direction "$-$" we mean that the head stays in its place. Furthermore, the head cannot move left of the $\triangleright$ symbol (the transition function δ is restricted in such a way that this cannot happen: if $\delta(q, \triangleright) = (q', a, \text{dir})$, then $a = \triangleright$ and $\text{dir} \neq \leftarrow$). For example, if $c = (q, \triangleright 01, 100)$ and $\delta(q, 1) = (q', 0, \leftarrow)$, then $c' = (q', \triangleright 0, 0100)$. In this case, we write $c \to_M c'$, and we also write $c \to_M^m c'$ if c' can be reached from c in m steps, and $c \to_M^* c'$ if $c \to_M^m c'$ for some $m \geq 0$ (we assume that $c \to_M^0 c$). Finally, if $v = \varepsilon$ and $\text{dir} = \rightarrow$, then we insert an additional $\sqcup$-symbol in our configuration, that is $u' = ub\sqcup$ and $v' = \varepsilon$.

A TM M receives an input word $w = a_1 \cdots a_n$, where $n \geq 0$ and $a_i \in \Sigma - \{\sqcup, \triangleright\}$ for each $i \in [n]$. The *start configuration of M on input w* is $sc(w) = (s, \triangleright, w)$. We call a configuration c *accepting* if its state is "yes", and *rejecting* if its state is "no". The TM M *accepts* (respectively, *rejects*) input w if $sc(w) \to_M^* c$ for some accepting (respectively, rejecting) configuration c.

Nondeterministic Turing Machines as Acceptors

We also use nondeterministic Turing Machines as acceptors, which are defined similarly to deterministic ones, but with the key difference that the a state-symbol pair has more than one outgoing transitions.

Definition 2.2: Nondeterministic Turing Machine

A *nondeterministic Turing Machine* (NTM) is defined as a tuple

$$M = (Q, \Sigma, \delta, s),$$

where

- Q is a finite set of states, including the *accepting state* "yes", and the *rejecting state* "no",
- Σ is a finite set of input symbols, called the *alphabet* of M, including the symbols $\sqcup$ (the *blank symbol*) and $\triangleright$ (the *left marker*),
- $\delta : (Q - \{\text{"yes"}, \text{"no"}\}) \times \Sigma \to \mathcal{P}(Q \times \Sigma \times \{\rightarrow, \leftarrow, -\})$ is the *transition function* of M, and
- $s \in Q$ is the *start state* of M.

Observe that for a given configuration $c = (q, ua, v)$, where $q \in Q - \{\text{"yes"}, \text{"no"}\}$, $a \in \Sigma$ and $u \in \Sigma^*$, several alternatives (q', b, dir) can belong to $\delta(q, a)$, each one of which generates a successor configuration c' as in the case of (deterministic) TMs. If c' is a possible successor configuration of c, then we write $c \to_M c'$. Moreover, we write $c \to_M^m c'$ if there exists a sequence of configurations $c_1, \ldots, c_{m-1}$ such that $c \to_M c_1$, $c_1 \to_M c_2$, $\ldots$, $c_{m-1} \to_M c'$. In this case, notice that it is possible that $c \to_M^m c'$ and $c \to_M^n c'$ with $m \neq n$. Moreover, we write $c \to_M^* c'$ if there exists $m \geq 0$ such that $c \to_M^m c'$ (again, we assume that $c \to_M^0 c$).

Given an input word w for a NTM M, the start configuration $sc(w)$ of M, and accepting and rejecting configurations of M, are defined as in the deterministic case. Moreover, M accepts input w if there exists an accepting configuration c such that $sc(w) \to_M^* c$, and M rejects w otherwise (i.e., M rejects w if there is no accepting configuration c such that $sc(w) \to_M^* c$).

2-*Tape Turing Machines as Acceptors*

We now define Turing Machines that, apart from a read-write working tape, they also have a read-only input tape.

Definition 2.3: 2-Tape Deterministic Turing Machine

A 2-*tape (deterministic) Turing Machine* (2-TM) is defined as a tuple

$$M = (Q, \Sigma, \delta, s),$$

where

- Q is a finite set of states, including the *accepting state* "yes", and the *rejecting state* "no",
- Σ is a finite set of input symbols, called the *alphabet* of M, including the symbols $\sqcup$ (the *blank symbol*) and $\triangleright$ (the *left marker*),
- $\delta : (Q - \{\text{"yes"}, \text{"no"}\}) \times \Sigma \times \Sigma \to Q \times \{\rightarrow, \leftarrow, -\} \times \Sigma \times \{\rightarrow, \leftarrow, -\}$ is the *transition function* of M, and
- $s \in Q$ is the *start state* of M.

A *configuration* of a 2-TM is a tuple

$$c = (q, u_1, v_1, u_2, v_2),$$

where $q \in Q$ and, for every $i \in \{1, 2\}$, we have that $u_i, v_i \in \Sigma^*$ and u_i is not empty. If M is in configuration c, then the input tape has content $u_1 v_1$ and the head of this tape is reading the last symbol of u_1, while the working tape has content $u_2 v_2$ and the head of this tape is reading the last symbol of u_2. We use left markers, which means that u_i always starts with $\triangleright$. Besides, the transition function δ is restricted in such a way that $\triangleright$ occurs exactly once in $u_i v_i$, and always as the first symbol of u_i.

Assume that M is in a configuration $c = (q, u_1 a_1, v_1, u_2 a_2, v_2)$, where $q \in Q - \{\text{"yes"}, \text{"no"}\}$, $a_1, a_2 \in \Sigma$ and $u_1, v_1, u_2, v_2 \in \Sigma^*$, and assume that $\delta(q, a_1, a_2) = (q', \text{dir}_1, b, \text{dir}_2)$, where dir_i is a direction, i.e., one of $\{\rightarrow, \leftarrow, -\}$. Then in one step M enters configuration $c' = (q', u_1', v_1', u_2', v_2')$, where u_1', v_1' is obtained from $u_1 a_1, v_1$ by moving the head one step in the direction dir_1, and u_2', v_2' is obtained from $u_2 a_2, v_2$ by replacing a_2 with b and moving the head one step in the direction dir_2. Recall that by moving the head in the direction "$-$" we mean that the head stays in its place. Furthermore, the head cannot move left of the $\triangleright$ symbol (again, the transition function δ is restricted in such a way that this cannot happen). For example, if $c = (q, \triangleright 01, 100, \triangleright, \varepsilon)$ and $\delta(q, 1, \triangleright) = (q', \leftarrow, \triangleright, \rightarrow)$, then $c' = (q', \triangleright 0, 1100, \triangleright, \varepsilon)$. In this case, we write $c \to_M c'$. We also write $c \to_M^m c'$ if c' can be reached from c in m steps, and $c \to_M^* c'$ if $c \to_M^m c'$ for some $m \geq 0$ (we assume that $c \to_M^0 c$).

A 2-TM M receives an input word $w = a_1 \cdots a_n$, where $n \geq 0$ and $a_i \in \Sigma - \{\sqcup, \triangleright\}$ for each $i \in [n]$. The *start configuration of M on input w* is $sc(w) = (s, \triangleright, w, \triangleright, \sqcup)$. We call a configuration c *accepting* if its state is "yes", and

rejecting if its state is "no". The TM M *accepts* (respectively, *rejects*) input w if $sc(w) \rightarrow_M^* c$ for some accepting (respectively, rejecting) configuration c.

2-*Tape Nondeterministic Turing Machines as Acceptors*

As for TMs, we also have the nondeterministic version of 2-TMs.

Definition 2.4: 2-Tape Nondeterministic Turing Machine

A 2-*Tape Nondeterministic Turing Machine* (2-NTM) is a tuple

$$M \;=\; (Q, \Sigma, \delta, s)\,,$$

where

- Q is a finite set of states, including the *accepting state* "yes", and the *rejecting state* "no",
- Σ is a finite set of input symbols, called the *alphabet* of M, including the symbols $\sqcup$ (the *blank symbol*) and $\triangleright$ (the *left marker*),
- $\delta:\ (Q - \{\text{"yes"}, \text{"no"}\}) \times \Sigma \times \Sigma \rightarrow \mathcal{P}(Q \times \{\rightarrow, \leftarrow, -\} \times \Sigma \times \{\rightarrow, \leftarrow, -\})$ is the *transition function* of M, and
- $s \in Q$ is the *start state* of M.

It is clear that, for a configuration $c = (q, u_1, a_1v_1, u_2, a_2v_2)$, where $q \in Q - \{\text{"yes"}, \text{"no"}\}$, $a_1, a_2 \in \Sigma$ and $v_1, v_2 \in \Sigma^*$, several alternatives $(q', \mathrm{dir}_1, b, \mathrm{dir}_2)$ can belong to $\delta(q, a_1, a_2)$, each one of which generates a successor configuration c' as in the case of 2-TMs. If c' is a possible successor configuration of c, then we write $c \rightarrow_M c'$. Moreover, we write $c \rightarrow_M^m c'$ if there exists a sequence of configurations $c_1, \ldots, c_{m-1}$ such that $c \rightarrow_M c_1$, $c_1 \rightarrow_M c_2$, $\ldots$, $c_{m-1} \rightarrow_M c'$. In this case, notice that it is possible that $c \rightarrow_M^m c'$ and $c \rightarrow_M^n c'$ with $m \neq n$. Moreover, we write $c \rightarrow_M^* c'$ if there exists $m \geq 0$ such that $c \rightarrow_M^m c'$ (again, we assume that $c \rightarrow_M^0 c$).

Given an input word w for a 2-NTM M, the start configuration $sc(w)$ of M, and accepting and rejecting configurations of M, are defined as in the deterministic case. Moreover, M accepts input w if there exists an accepting configuration c such that $sc(w) \rightarrow_M^* c$, and M rejects w otherwise (i.e., M rejects w if there is no accepting configuration c such that $sc(w) \rightarrow_M^* c$).

Turing Machines as Computational Devices

If a 2-TM acts not as a language acceptor but rather as a device for computing a function f, then a write-only output tape is added and the states "yes" and "no" are replaced with a *halting state* "halt"; once the computation enters the halting state, the output tape contains the value $f(w)$ for the input w.

Definition 2.5: Turing Machine with Output

A *Turing Machine with output* (TMO) is a tuple

$$M \;=\; (Q, \Sigma, \delta, s)\,,$$

where

- Q is a finite set of states, including the *halting state* "halt",
- Σ is a finite set of input symbols, called the *alphabet* of M, including the symbols $\sqcup$ (the *blank symbol*) and $\triangleright$ (the *left marker*),
- $\delta:\ (Q - \{\text{"halt"}\}) \times \Sigma \times \Sigma \to Q \times \{\rightarrow, \leftarrow, -\} \times \Sigma \times \{\rightarrow, \leftarrow, -\} \times \Sigma$ is the *transition function* of M, and
- $s \in Q$ is the *start state* of M.

If $\delta(q, a_1, a_2) = (q', \mathrm{dir}_1, b, \mathrm{dir}_2, c)$, then $q', \mathrm{dir}_1, b, \mathrm{dir}_2$ are used exactly as in the case of a 2-TM accepting a language. Moreover, if $c \neq \sqcup$, then c is written on the output tape and the head of this tape is moved one position to the right; otherwise, no changes are made on this tape. The start configuration of a TMO M on input w is $sc(w) = (s, \triangleright, w, \triangleright, \varepsilon, \triangleright, \varepsilon)$. The output of M on input w is the word u such that $sc(w) \to_M^* (\text{"halt"}, u_1, v_1, u_2, v_2, \triangleright u, \varepsilon)$.

Complexity Classes

We proceed to introduce some central complexity classes that are used in this book. Recall that $\mathbb{R}_0^+$ is the set of non-negative real numbers. Given a function $f : \mathbb{N} \to \mathbb{R}_0^+$, a TM (respectively, NTM) M is said to run in *time* $f(n)$ if, for every input w and configuration c, $sc(w) \to_M^m c$ implies $m \leq f(|w|)$.[1] We further say that M *decides* a language L if M accepts every word in L and rejects every word not in L. Notice that this implies that M's computation is finite on every input. We define the classes of decision problems

$$\mathrm{TIME}(f(n)) \;=\; \{L \mid \text{ there exists a TM that decides } L \text{ and runs in time } f(n)\}.$$

and

$$\mathrm{NTIME}(f(n)) \;=\; \{L \mid \text{ there exists a NTM that decides } L \text{ and runs in time } f(n)\}.$$

The following time complexity classes are used in this book:

[1] The running time of a TMO is defined in the same way.

Definition 2.6: Time Complexity Classes

$\textsc{Ptime} = \bigcup_{k\in\mathbb{N}} \text{TIME}(n^k)$ $\quad$ $\text{NP} = \bigcup_{k\in\mathbb{N}} \text{NTIME}(n^k)$

$\textsc{ExpTime} = \bigcup_{k\in\mathbb{N}} \text{TIME}(2^{n^k})$ $\quad$ $\textsc{NExpTime} = \bigcup_{k\in\mathbb{N}} \text{NTIME}(2^{n^k})$

$\textsc{2ExpTime} = \bigcup_{k\in\mathbb{N}} \text{TIME}(2^{2^{n^k}})$

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

$$\text{SPACE}(f(n)) = \{L \mid \text{there exists a 2-TM that decides } L \text{ and runs in space } f(n)\}.$$

and

$$\text{NSPACE}(f(n)) = \{L \mid \text{there exists a 2-NTM that decides } L \text{ and runs in space } f(n)\}.$$

The following space complexity classes are used in this book:

Definition 2.7: Space Complexity Classes

$\textsc{DLogSpace} = \text{SPACE}(\log n)$ $\quad$ $\textsc{NLogSpace} = \text{NSPACE}(\log n)$

$\textsc{PSpace} = \bigcup_{k\in\mathbb{N}} \text{SPACE}(n^k)$ $\quad$ $\textsc{NPSpace} = \bigcup_{k\in\mathbb{N}} \text{NSPACE}(n^k)$

$\textsc{ExpSpace} = \bigcup_{k\in\mathbb{N}} \text{SPACE}(2^{n^k})$ $\quad$ $\textsc{NExpSpace} = \bigcup_{k\in\mathbb{N}} \text{NSPACE}(2^{n^k})$

At this point, let us stress that we can always assume that the computation of a space-bounded 2-TM M is finite on every input word. Intuitively, since the space that M uses is bounded, the number of different configurations in which M can be is also bounded. Therefore, by maintaining a counter that "counts" the steps of M, we can guarantee that M will never fall in an unnecessarily long computation, which in turn allows us to assume that the computation of M is finite. Further details on this assumption can be found in any standard textbook on computational complexity.

For a complexity class $\mathcal{C}$, the class $\textsc{co}\mathcal{C}$ is defined as the set of complements of the problems in $\mathcal{C}$, that is, $\textsc{co}\mathcal{C} = \{\Sigma^* - L \mid L \in \mathcal{C}\}$. It is known that

$$\textsc{DLogSpace} \subseteq \textsc{NLogSpace} \subseteq \textsc{Ptime} \subseteq \text{NP} \subseteq \textsc{PSpace} = \textsc{NPSpace} \subseteq \textsc{ExpTime} \subseteq \textsc{NExpTime} \subseteq \textsc{ExpSpace} = \textsc{NExpSpace} \subseteq \textsc{2ExpTime}$$

[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 $|u_2v_2| \leq f(|w|)$.

$$\textsc{Ptime} \subsetneq \textsc{ExpTime} \subsetneq \textsc{2ExpTime}$$

$$\text{NP} \subsetneq \textsc{NExpTime}$$

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 $f \models \varphi$:

- 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 $f \models \varphi_2$.

[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 $\varphi = (\varphi_1 \vee \varphi_2)$, 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*mb*, 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:

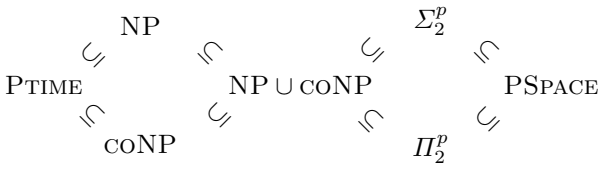
Problem: $\exists\forall$QSAT

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 $\exists\forall$QBFmb in polynomial time. Similarly, Π_2^p is the class of decision problems reducible to $\forall\exists$QBFmb in polynomial time. Recall that QSAT is PSPACE-complete. We know that



We finally remark that the smallest complexity class we consider here is DLOGSPACE. In database theory, and especially in its logical counterpart, that is, finite model theory, it is very common to consider parallel complexity classes, of which the smallest one is AC^0. These are circuit complexity classes, and the machinery needed to define them is not TMs but rather circuits, parameterized by their fan-in (the number of inputs to their gates), their size, and their depth. Due to the notational overhead this incurs, we shall not be using circuit-based classes in this book. The interested reader can consult books on finite model theory and descriptive complexity to understand the differences between DLOGSPACE and classes such as AC^0.

C

Input Encodings

To reason about the computational complexity of problems, we need to represent their inputs (such as databases, queries, and constraints) as inputs to Turing Machines, that is, as words over some finite alphabet.

Encoding of Queries and Constraints

Queries and constraints will most commonly be coming from a query language and a class of constraints, respectively, defined by a formal syntax. We thus associate a query and a set of constraints with its parse tree, which, of course, can be easily encoded as a word over a finite alphabet.

Encoding of Databases

For databases, the idea is that each value in the active domain can be encoded as a number in binary, and then use further separator symbols that allows us to faithfully encode the facts occurring in the database.

We assume a strict total order $<_{\mathsf{Rel}}$ on the elements of Rel, and a strict total order $<_{\mathsf{Const}}$ on the elements of Const. Consider a schema $\mathbf{S} = \{R_1, \ldots, R_n\}$ with $R_i <_{\mathsf{Rel}} R_{i+1}$ for each $i \in [n-1]$, and a database D of $\mathbf{S}$ with $\mathrm{Dom}(D) = \{a_1, \ldots, a_k\}$ and $a_i <_{\mathsf{Const}} a_{i+1}$ for each $i \in [k-1]$. We proceed to explain how D is encoded as a word over the alphabet

$$\Sigma \;=\; \{0, 1, \triangle, \#, \$, \square\}.$$

We first explain how constants, tuples, and relations are encoded:

- The constant $a_i \in \mathrm{Dom}(D)$, for $i \in [k]$, is encoded as the number i in binary, and we write $enc(a_i)$ for the obtained word over $\{0, 1\}$.
- A tuple $\bar{t} = (a_1, \ldots, a_\ell)$ over $\mathrm{Dom}(D)$, for $\ell \geq 0$, is encoded as the word

$$enc(\bar{t}) \;=\; \begin{cases} \square enc(a_1)\square \cdots \square enc(a_\ell)\square & \text{if } \ell > 0, \\ \square\square & \text{if } \ell = 0. \end{cases}$$

- A relation $R_i^D = \{\bar{t}_1, \ldots, \bar{t}_m\}$, for $i \in [n]$ and $m \geq 0$, is encoded as

$$enc(R_i^D) \;=\; \begin{cases} \$enc(\bar{t}_1)\$ \cdots \$enc(\bar{t}_m)\$ & \text{if } m > 0, \\ \$\$ & \text{if } m = 0. \end{cases}$$

We can now encode the database D as a word over Σ as follows:

$$enc(D) \;=\; \triangle enc(a_1) \triangle \cdots \triangle enc(a_k) \triangle \# enc(R_1^D) \# \cdots \# enc(R_n^D) \# \,.$$

The key property of the above encoding is that, for a database D of a schema $\mathbf{S}$, and a tuple $\bar{t}$, given as their encodings $enc(D)$ and $enc(\bar{t})$, respectively, we can check via a deterministic computation, which uses logarithmic space in the size fo $enc(D)$, whether $\bar{t} \in R^D$ for some $R \in \mathbf{S}$. In what follows, we write $enc(i)$ for the binary representation of an integer $i > 0$.

Lemma C.1. *Let $\boldsymbol{S}$ be the schema $\{R_1, \ldots, R_n\}$ with $R_1 <_{\mathsf{Rel}} \cdots <_{\mathsf{Rel}} R_n$. Consider a database D of $\boldsymbol{S}$, a tuple $\bar{t}$, and an integer $i \in [n]$, and let w be the word $\triangleright enc(D) \flat enc(\bar{t}) \flat enc(i)$ over $\Sigma \cup \{\triangleright, \sqcup, \flat\}$. There exists a 2-TM M with alphabet Σ such that the following hold:*

1. *M accepts w if and only if $\bar{t} \in R_i^D$, and*
2. *M runs in space $O(ar(R_i) \cdot \log |enc(D)|)$ if $ar(R_i) > 0$, and $O(\log |enc(D)|)$ if $ar(R_i) = 0$.*

Proof. We first give a high-level description of the 2-TM M; for brevity, we write *it* for the symbol read by the head of the input tape:

1. Let $ctr = 0$ – this is a counter maintained on the work tape in binary.
2. While $ctr \neq i$ do the following:
   a) If $it = \#$, then $ctr := ctr + 1$.
   b) Move the head of the input tape to the right so that it reads the first \$ symbol of $enc(R_i^D)$.
3. Move the head of the input tape to the right so that it reads the first $\square$ symbol of $enc(\bar{u})$, where $\bar{u}$ is the first tuple of R_i^D (i.e., $enc(R_i^D) = \$enc(\bar{u})\$ \cdots \$$), or the second \$ symbol of $enc(R_i^D)$ in case R_i^D is empty (which means that $enc(R_i^D) = \$\$$).
4. Erase the content of the work tape by replacing every symbol different than $\sqcup$ with $\sqcup$ (since ctr is not needed further), and move its head after the left marker $\triangleright$.
5. Repeat the following steps until $it = \#$ (which means that the relation R_i^D has been fully explored):
   a) While $it \neq \$$ do the following:

(i) Copy *it* to the work tape.

(ii) Move the head of both tapes to the right.

b) Assuming that $\triangleright u \sqcup$ is the content of the work tape, if $u = enc(\bar{t})$, then halt and accept; otherwise:

(i) Move the head of the input tape to the right so that it reads the first $\square$ symbol of the encoding of the next tuple of R_i^D, or the symbol # if the last tuple of R_i^D has just been explored. In other words, the head of the input tape reads the symbol to the right of the last \$ symbol read during the while loop of step (a).

(ii) Erase the content of the work tape by replacing every symbol different than $\sqcup$ with $\sqcup$ (since the copied tuple is not needed further), and move its head after the left marker $\triangleright$.

6. Halt and reject.

It is easy to verify that M accepts w if and only if $enc(R_i^D)$ is of the form $\$ \cdots \$ enc(\bar{t}) \$ \cdots \$$, or, equivalently, $\bar{t} \in R_i^D$. It remains to argue that M runs in the claimed space. At each step of the computation of M, the work tape holds either *ctr*, or the word $enc(\bar{t})$ for some $\bar{t} \in R_i^D$. The value of *ctr* (represented in binary) can be maintained using $O(|enc(i)|)$ bits. The encoding of a tuple of R_i^D takes space $O(\mathrm{ar}(R_i) \cdot \log |\mathrm{Dom}(D)|)$. Therefore, the space used is

$$O\left(\log |enc(i)| \ + \ \mathrm{ar}(R_i) \cdot \log |\mathrm{Dom}(D)|\right).$$

Since $|enc(i)| \leq |enc(D)|$ and $|\mathrm{Dom}(D)| \leq |enc(D)|$, we can conclude that the above 2-TM on input w runs in space $O(\mathrm{ar}(R_i) \cdot \log |enc(D)|)$ if $\mathrm{ar}(R_i) > 0$, and $O(\log |enc(D)|)$ if $\mathrm{ar}(R_i) = 0$, and the claim follows. $\square$

Note that the encoding described above is not the only way of encoding a database as a word over a finite alphabet. We could employ any other encoding as long as it enjoys the property established in Lemma C.1, without affecting the complexity results presented in this book.