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

Principles of Databases

November 12, 2020

Santiago Paris
Bayreuth Edinburgh

Contents

Conventions (Just for the Authors)

The purpose of this chapter is to remind the authors of the conventions we will use throughout the book.

Structuring

We will not use section or subsection environments. Instead we use only one kind of title. The macro is called \heading and looks like this:

Heading

Here is some text below the heading.

Decision Problems

Format decision problems as it's done in background.tex. Here is an example:

PROBLEM:	CQ-Evaluation
INPUT:	a conjunctive query q,
	a database D, a tuple $\bar{a}$ over $\mathrm{Dom}(D)$
OUTPUT:	yes if $\bar{a} \in q(D)$ and no otherwise

We have PROBLEM, ... in small caps, the Problem-Name in textsf, with a dash in it, and we can use multiple lines to nicely structure the input and the output. At the moment we don't have a "." at the end of the lines / sentences in the block.

The formal notation of the problem is in line with how you read it in english, so you can just say that CQ-Containment is NP-complete.

Terminology

1. **Static analysis / optimization / containment / minimization.** *Static analysis* is a very general term and may include problems such as transferability, semantic acylicity, etc. *Optimization* is a special kind of static analysis which includes containment, satisfiabilitym, and minimization.
2. **Function**, not mapping or map. By default a function is total. If it's not, we call is a partial function.
3. **Formulae**. Not formulas.
4. Queries have **output** (not a result).
5. Graphs and trees have **nodes**, not vertices.
6. **Words / strings.** We say *words.* Not strings.
7. Elements from Const are called **values**.

Formatting and Notation

1. Natural numbers, following Bourbaki, start with 0, notation $\mathbb{N}$.
2. Try to use $O(\cdot)$ in prose, if necessary then $f \in O(\cdot)$.
3. The set $\{1, \dots, n\}$ is written as $[1, n]$.
4. **Size.** $|D|$ is the cardinality of a set, so it's the number of tuples in database D. The bit-size is $\|D\|$. There is a macro `\size{}` for this bit-size.
5. **Set difference** is written as $S_1 - S_2$. Not $S_1 \setminus S_2$, which is 19th-century.
6. **For every... statements**. We write "for every $i \in [1, n]$". Or, for every $x \in \{a_1, \dots, a_n\}$. Not "for every $1 \leq i \leq n$" or "for every $i = 1, \dots, n$". "For every $i \in \{1, \dots, n\}$" may be acceptable.
7. **SF fonts**. We only use $\mathsf{SF\ fonts}$ for the sets Const, Var, Rel, Att, and for computational problems, like FO-$\mathsf{Satisfiability}$. All other uses are in RM. Not sure about Dom. We usually use that as $\mathrm{Dom}(D)$, which is a different kind of notation.
8. **FO.** We can use $x \neq y$ to abbreviate $\neg(x = y)$.
9. **Substitution.** When we write $\varphi[x/y]$, it means "substitute x with y". We decided on Feb 4th, 2020 to use a 21st century notation.
10. **Queries** will be lower case, such as q. For tree patterns, Wim sometimes also uses p.
11. **Formulae vs Queries.** φ is a *formula* and $\varphi(\bar{x})$ is a *query*, since it specifies the output. We are more liberal when it comes to q. Here, q is a query, and so is $q(\bar{x})$.
12. **Spacing** between quantifiers and matrix of formula $\exists y_1 \exists y_2 \dots \exists y_m \, \varphi$ and $\exists \bar{x} \, \varphi$. We use backslash comma.

13. Use ldots for comma-separated things and cdots for sequences. Examples: The set $\{a_1, \ldots, a_n\}$. The word $a_1 \cdots a_n$. Tuples $(a_1, \ldots, a_n)$. Ordering $a_1 < \cdots < a_n$.
14. **Tuples in the named perspective** are denoted $(\mathit{att}_1\colon a_1, \ldots, \mathit{att}_k\colon a_k)$. We use `\colon` instead of ":". It looks nicer because it produces less space. Good: $(\mathit{att}_1\colon a_1, \ldots, \mathit{att}_k\colon a_k)$. Bad: $(\mathit{att}_1 : a_1, \ldots, \mathit{att}_k : a_k)$.
15. Equality in relational algebra is a macro `\raeq`, to avoid things like 1=2. Currently set to $\doteq$
16. **Composition of functions.** If we have $X \xrightarrow{f} Y \xrightarrow{g} Z$ then we write $g \circ f$ for the composition, which maps X to Z
17. **Assignment in algorithms** is denoted :=
18. Trees are captial T, because databases are capital D.

Conventions

1. Environments: We use example environment. We don't use the remark environment.
2. When we talk about complexity classes, it's Turing machines. If it's big O, we use RAM.
3. Titles Should Be Capitalized. This includes heading titles.
4. We say "We leave the proof as an exercise" then the exercise doesn't need a hint. If we say "We leave the proof as Exercise 5.3", then the exercise should have a hint. (Not just say "Prove Lemma 5.1")

Notifications and Things to Discuss

1. WIM (11.5.2020): Ordering of attributes in named perspective in examples.
2. WIM (7.3.2020): I noticed a notation clash for attributes and attribute names in the named perspective. We wrote A as set of attributes (background chapter) but then called attributes A, B etc. I now use U for subsets of Att. So now we can write $U = \{A, B\} \subseteq \mathsf{Att}$. Since this is similar to $\mathbf{S} = \{R_1, R_2\} \subseteq \mathsf{Rel}$, I'm OK with it.
3. WIM (10.3.2020): We now introduce FO over the active domain in the FO / RA chapter. For clarity, I'm denoting it with $\mathrm{FO}_{\mathrm{act}}$. From chapter 7 on, however, we denote $\mathrm{FO}_{\mathrm{act}}$ with FO. Should we say in Chapter 6 that, from now on, we only consider FO with active domain semantics? (And perhaps add a note in Chapter 3 that warns the reader that, while this is the well-known semantics of FO from mathematical logic, we will make a slight change to it in Chapter 6? Namely active domain?) DECISION: Let's do FO active immediately.

4. WIM (24.3.2020): Does anyone remember why we only have two-string TMs in the Appendix? Probably just for brevity, right? (Two-string TMs are less convenient than one-string TMs for the undecidability reduction of finite FO satisfiability.) DECISION: Let's start from one string TMs in the Appendix.
5. WIM: As discussed on 28.01.2020, I added a placeholder and initial dump for a new chapter on RA and SQL – Chapter 5.
6. MARCELO: I am using notation $\bar{u}$, $\bar{v}$ for tuples that include variables and constants, and $\bar{x}$, $\bar{y}$, $\bar{z}$ for tuples that only consist of constants. Is this OK?
7. MARCELO: we use $\square$ (command `\qed`) at the end of each example.
8. MARCELO: I would prefer to use "A query q over a schema $\mathbf{S}$" instead of "A query q over databases of schema $\mathbf{S}$".
9. WIM: I think that it will be reader friendly to add a list of notation symbols somewhere. I mean: Const, $\mathbf{S}$, etc.

Examples

In this book we talk about a few things that happened after [1]. To start with, we have a running example inspired by Wim's Wikidata example:

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

with the obvious keys and foreign keys. Use either as-is, or add or remove relations, such as

```
SubclassOff [ prname1, prname2 ]
```

to introduce hierarchies in professions. This is useful to say that a singer is some kind of artist, for example. Later to be used with graphs as well.

Marcelo:
Can we use ℓ instead of l? WIM: +1!

Marcelo:
We need to standardize the notation. For example, why do we use different fonts for sort(R) and $\mathsf{SR}(q, A)$? Notice that before we were using $\mathcal{R}(a, A)$ instead of $\mathsf{SR}(q, A)$.

Marcelo:
We need to standardize the notation for summations with conditions. I would like to use

$$\sum_{i\,:\,\varphi(i)} f(i).$$

In the document, we use different notations for such a summation. For example, we use:

$$\sum_{\{i\,|\,\varphi(i)\}} f(i).$$

Marcelo:
Do we denote a tuple in a database as t or $\bar{t}$? In the Background section we use t, but we also use $\bar{t}$ in other sections. I would prefer to use t.

1 Introduction

Wim's comments:

This should be some general introductory text of the book. What do we want to present here? What's the philosophy? What kind of questions are we interested in? Can you teach one chapter in one 2-hour lecture?

Maybe a dependency graph of chapters, as in AHV?

2

Background

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

- the relational model,
- queries and query languages, and
- key computational problems considered in the study of foundations of databases.

First we fix some basic notation. By $\mathbb{N}$ we denote the set $\{0, 1, 2, \ldots\}$ of natural numbers. For $i, j \in \mathbb{N}$, we denote the set $\{k \mid k \in \mathbb{N} \text{ and } i \leq k \leq j\}$ by $[i, j]$. We furthermore denote $[1, j]$ by $[j]$. For a finite set S, we denote by $|S|$ its cardinality, i.e., the number of elements of S. In the study of databases we primarily deal with finite sets; if we need to indicate that a set is infinite, we write $|S| = \infty$.

If $(a_1, \ldots, a_k) \in S^k$ then it is a *tuple* and $k \in \mathbb{N}$ is its *arity*. We also call $(a_1, \ldots, a_k)$ a *k-ary tuple*. We often abbreviate tuples $(a_1, \ldots, a_k) \in S^k$ as $\bar{a}$. Also, if $\bar{a}$ is a k-ary tuple, then we assume that its components are $(a_1, \ldots, a_k)$.

Let S and T be sets and $f : S \to T$ be a function. By default, we assume functions to be total. Throughout the book, we will also use the letter f to denote extensions of f on more complex objects (such as sets of elements of S, tuples 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.

We write $\mathcal{P}(S)$ for the powerset of S, and $\mathcal{P}_{\text{fin}}(S)$ for the finite powerset of S, i.e., the collection of all finite subsets of S.

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. Thus, to model databases, we need two disjoint sets

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

We assume that these sets are countably infinite, to ensure we never run out of ways to name new tables and their attributes. In practice, of course, these sets are 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 infinite.

Notice that in `create table` declarations, one specifies types of attributes as well, for examples, integers, Booleans, strings. In the study of 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 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 one to borrow many tools from it, and it also streamlines notations significantly.

Each relation name is associated with a set of attributes. We model this by means of a function $\mathrm{sort} : \mathsf{Rel} \to \mathcal{P}_{\mathrm{fin}}(\mathsf{Att})$, that assigns such a set of attributes to a relation name. We require that, for each finite set U of attribute names, there exists an infinite set of relation names R such that $\mathrm{sort}(R) = U$. This condition simply ensures that we can use as many relation names of a given sort as we need.

If R is a relation name and $U = \{A_1, \dots, A_k\}$ is a finite set of attribute names (that is, $U \in \mathcal{P}_{\mathrm{fin}}(\mathsf{Att})$), we write $R[A_1, \dots, A_k]$ when $\mathrm{sort}(R) = U$. For example, we write `City[city_id, name, country]` when the attributes associated with relation name `City` are `city_id`, `name`, and `country`. The *arity* of relation name $R[A_1, \dots, A_k]$ is denoted by $\mathrm{ar}(R)$ and is defined as the number of its attributes, i.e., $k = |\{A_1, \dots, A_k\}|$. For $k \in \mathbb{N}$, we denote by $R[k]$ that $\mathrm{ar}(R) = k$. For arities 1, 2, and 3, we speak of unary, binary, and ternary relations. Finally, a *database schema* is a finite set $\mathbf{S} = \{R_1, \dots, R_n\}$ of relation names.

Note that these definitions, as stated, imply that if we use a relation name R, its sort and arity should be the same throughout the entire book. However, due to the limited number of symbols we will use to denote relations, we will not carry this through consistently. That is, in different examples, we may use the same relation symbol to have different sets of attributes and arities. This will never lead to clashes however; such discrepancies only occur in different non-overlapping contexts.

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 is therefore more natural in examples, the unnamed perspective provides a very clean mathematical model that is easier to use for the study of the key computational properties. It is easy to go back and forth between the two, since their expressiveness is exactly the same.

Under the *named perspective*, attribute names are viewed as an explicit part of a database. Here, a *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 it 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, using this notation, the tuple $(A_1\colon a_1, A_2\colon a_2)$ and $(A_2\colon a_2, A_1\colon a_1)$ are the same function.

A *relation instance* in the named perspective is a set S of tuples of the same sort U, which we also call the sort of the relation instance S and which we denote by $\mathrm{sort}(S)$.[1] By nRI (for *named* relational instances) we denote the set of all such relation instances. In the named perspective, a *database instance* D of schema $\mathbf{S} = \{R_1, \ldots, R_n\}$ is a function

$$D : \mathbf{S} \to \mathsf{nRI},$$

where we require that $\mathrm{sort}(D(R_i)) = \mathrm{sort}(R_i)$ for every $i \in [n]$.

Under the *unnamed perspective*, a *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. In the unnamed perspective, a *relation instance* is a set of tuples of the same arity k. We say that k is the arity of the relation instance. By uRI (for *unnamed* relational instances) we denote the set of all such relation instances, and then a *database instance* D of schema $\mathbf{S} = \{R_1, \ldots, R_n\}$ is a function

$$D : \mathbf{S} \to \mathsf{uRI},$$

where we require that the arity of $D(R_i)$ equals $\mathrm{ar}(R_i)$ for every $i \in [n]$.

For brevity, we sometimes refer to a database instance as a *database.* When the exact perspective is clear from the context, or not relevant, we refer to the set RI of all relation instances. In both the named and 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, writing just R_i instead of R_i^D. Of course this means that we will effectively use the same notation for relation names and

[1] Notice that in this definition we allow infinite relation instances, which are obviously not used in practice, but play a very useful role as a conceptual tool (for example, they allows us to prove some results more elegantly).

for relation instances. This is nonetheless 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.

We write $\text{Inst}(\mathbf{S})$ for the set of all database instances of schema $\mathbf{S}$. The *domain* of a database instance D, denoted by

$$\text{Dom}(D),$$

is the set of all elements of Const that occur in it.

There is clearly a close connection between the named and unnamed perspectives, which is not surprising because both are mathematical abstractions of the same thing. It is instructive to make this connection formal and we will use it throughout the entire book. To do this, we need to assume that there exists an ordering $\lessdot$ on pairs of relation- and attribute names.

More precisely, consider a tuple $(A_1 : a_1, \ldots, A_k : a_k)$ of a relation instance R in the named perspective, where $(R, A_1) \lessdot (R, A_2) \lessdot \cdots \lessdot (R, A_k)$. In the unnamed perspective, we interpret this tuple as $(a_1, \ldots, a_k)$.

Conversely, to go from the unnamed to the named perspective, we assume that Att contains an attribute name $\#_i$ for each $i \geq 1$. These will be used as attribute names for positions in a tuple. That is, we interpret a tuple $(a_1, \ldots, a_k) \in \mathsf{Const}^k$, under the named perspective, as the function from $\{\#_1, \ldots, \#_k\}$ to Const that maps $\#_i$ to a_i for every $i \in [k]$.

Since this connection between the two perspectives is useful in many places throughout the book, we will assume from now on that, whenever the named perspective is used, the ordering $\lessdot$ is always available.

Since the unnamed perspective is often mathematically more elegant and since the named perspective is closer to practice, 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 as $R[A, B, \ldots]$ in an example, then we assume that the ordering of attributes 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.

Databases, Sets of Facts, and Sets of Atoms

There is another way of thinking about a database, namely as a set of facts. In the unnamed perspective, when R is a k-ary relation symbol and $\bar{a}$ is a k-ary tuple from Const^k, we say that $R(\bar{a})$ is a *fact*. The fact $R(\bar{a})$ simply states that $\bar{a}$ is in R^D. Since a fact is always a statement about a single tuple, we simplify the notation $R((a_1, \ldots, a_k))$ to $R(a_1, \ldots, a_k)$, which is less heavy. For instance, we can write $D = \{R_1(a, b), R_1(b, c), R_2(a, c, d)\}$ as a shorthand for stating that $R_1^D = \{(a, b), (b, c)\}$ and $R_2^D = \{(a, c, d)\}$.

Another useful notion that we will use throughout the book 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 $R(u_1, \dots, u_k)$ instead of $R((u_1, \dots, u_k))$. Given a set of atoms S, we write $\mathrm{Dom}(S)$ for the set of constants and variables in S. For example, $\mathrm{Dom}(\{R(a,x,b), R(x,a,y)\}) = \{a,b,x,y\}$. We also write R^S for the set of tuples $\{\bar{u} \mid R(\bar{u}) \in S\}$.

Queries and Query Languages

Queries will appear throughout the book as both *semantic* and *syntactic* objects. As a semantic object, a query is a partial function $q : \mathrm{Inst}(\mathbf{S}) \to \mathrm{Inst}(\mathbf{S}')$ that maps database instances over some schema $\mathbf{S}$ to database instances over 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. For brevity, we say that a *relation schema* is a database schema that contains only a single relation.

The key properties of queries are that they are *computable* and *generic.* The latter means that queries behave in the same way regardless what data one stores in the database. Formally, a partial function q on databases over $\mathbf{S}$ is *generic* if for each database instance D over $\mathbf{S}$ and each bijection $\rho : \mathsf{Const} \to \mathsf{Const}$, we have that $q(\rho(D)) = \rho(q(D))$. This is visualized by the following diagram.

$$\begin{tikzcd}
D \arrow[r, "q"] \arrow[d, "\rho"'] & q(D) \arrow[d, "\rho"] \\
\rho(D) \arrow[r, "q"] & q(\rho(D)) = \rho(q(D))
\end{tikzcd}$$

For example, if q asks whether there are two different pairs in relation R whose first components are the same, then it will be true in $D = \{R(1,2), R(1,3)\}$ but it will also be true in $D' = \{R(1,4), R(1,5)\}$ and $D'' = \{R(6,7), R(6,8)\}$, etc.; any bijection $\rho : \mathsf{Const} \to \mathsf{Const}$ applied to D does not change the truth of this query. This reflects the fact that query languages manipulate data.

The reason genericity is true in the query languages we consider is that they, by and large, are based on logical languages, and properties definable in logical languages are closed under isomorphism. Applying a bijection $\rho : \mathsf{Const} \to \mathsf{Const}$ to a database instance D results, of course, in an isomorphic database.

Sometimes, however, logical formulae can mention constants. For instance, we may ask whether there exists a tuple whose first component is 1. Considering the database instances D, D', D'' mentioned before, we see that this query is true in D and D', but false in D''. This is because D'' is obtained from D by applying a bijection ρ that does not preserve this constant value 1, while D' is obtained by applying a bijection that preserves 1. Thus, for queries that mention constants, we shall rely on the more general notion of C-genericity, where C is a subset of Const. It requires that $q(\rho(D)) = \rho(q(D))$ for every bijection ρ such that $\rho(c) = c$ for all $c \in C$. Notice that a query is generic if and only if it is C-generic for $C = \emptyset$ (that is, we do not fixed any constants when asking a query to be generic).

To define computability of queries, we will use the notion of *Turing Machine with Output (TMO)*, for which we recall the fundamental definitions in Appendix A. We say that a partial function $q : \mathrm{Inst}(\mathbf{S}) \to \mathrm{Inst}(\mathbf{S}')$ is *computable* if there exists a TMO M that, for each database instance $D \in \mathrm{Inst}(\mathbf{S})$,

(1) if $q(D)$ is undefined, then M does not halt on input D, and
(2) if $q(D)$ is defined, M halts on input D with $q(D)$ on its output tape.

Definition 2.1 (Queries and query languages). *Let $\boldsymbol{S}$ be a database schema and $\boldsymbol{S}'$ be a relation schema. A* query *from $\boldsymbol{S}$ to $\boldsymbol{S}'$ is a partial function* ***(Marcelo)***[2]

$$q : \mathit{Inst}(\boldsymbol{S}) \to \mathit{Inst}(\boldsymbol{S}')$$

that is generic and computable. A query language *is a set of queries.*

Of course queries as semantic objects must be given in some syntax. In order to clarify this, we assume a countably infinite set

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

disjoint from Const, Rel, and Att. The syntax of queries could be relational algebra, SQL, first-order logic, Datalog, or other languages. In many of these languages (notably, languages based on logic) we will denote these syntactic objects as $q(\bar{x})$, where $\bar{x} = (x_1, \ldots, x_k) \in \mathsf{Var}^k$ is a tuple of variables. The purpose of $\bar{x}$ is to make clear what is the *output* of the query. More precisely, we will always define for a database instance D and tuple $\bar{a} = (a_1, \ldots, a_k)$ in Const^k whether D *satisfies q using the values $\bar{a}$*, denoted by $D \models q(\bar{a})$. Then with the syntactic object $q(\bar{x})$, we associate a semantic object that produces a k-ary relation over Const for each database D, defined as:

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

[2] **Marcelo:** Why do we need to define a query as a partial function? Notice that if we consider partial functions, then we need to add an additional condition to the definition of genericity.

This semantic object will always be a query in the sense of Definition 2.1.

We call a query *k-ary* if it produces a k-ary relation. If it produces a 0-ary relation, we also call the query *Boolean*. In this case, there are only two possible outputs, namely the empty set $\{\}$ and the singleton set $\{()\}$ containing the empty tuple. We will 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 $q(D) = \{\}$.

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, to be able to substitute a query with a better behaved one that returns the same result.

In their most common forms that we see here, these problems are parameterized by a query language $\mathcal{L}$. We usually present them as *decision* problems, i.e., problems whose answers could be yes or no. This is because the standard complexity classes, which are given in Appendix A, and which are commonly used to describe computational resources that a problem requires, are defined for decision problems.

The *query evaluation problem*, or simply the *evaluation problem* is of the following form:

PROBLEM:	$\mathcal{L}$-Evaluation
INPUT:	a query q from $\mathcal{L}$,
	a database D, a tuple $\bar{a}$ over $\text{Dom}(D)$
OUTPUT:	yes if $\bar{a} \in q(D)$ and no otherwise

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

Very often we shall deal with a different kind of complexity of query evaluation, when query q is *fixed*. This is referred to as *data complexity* since we measure 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.

Wim:
Say how we see the input here. The size of the input is size of D plus size of q (or may be different depending on the problem). We say here that, in order to do a more refined analysis, we sometimes analyse complexity in terms of D and q.

(Marcelo)[3]

[3] **Marcelo:** The only time when we need to restrict to finite databases.

To reiterate, as we shall use these concepts many times throughout the 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 of query evaluation refers to the complexity of the $\mathcal{L}$-Evaluation problem when its inputs contain only D and $\bar{a}$, and q on the other hand is fixed.

Thus, when we talk of data complexity, then $\mathcal{L}$-Evaluation is really a family of problems, one for each $q \in \mathcal{L}$. Nonetheless, we shall apply the standard notions of complexity theory, such as membership in a class, or hardness and completeness for a class, to data complexity. We now explain precisely what we mean by that. If we say that the "data complexity of $\mathcal{L}$-Evaluation" is in some complexity class $\mathcal{C}$, it means that for every query $q \in \mathcal{L}$, the complexity of deciding if $\bar{a} \in q(D)$ is in $\mathcal{C}$. If we say that the "data complexity of $\mathcal{L}$-Evaluation" is hard for $\mathcal{C}$, it means that there exists a query $q \in \mathcal{L}$ for which deciding if $\bar{a} \in q(D)$, on the input that consists of D and $\bar{a}$, is hard for $\mathcal{C}$. And as usual, a problem that is in the complexity class $\mathcal{C}$ and is hard for it is $\mathcal{C}$-complete.

The basis of the static analysis of queries is the *containment* problem. A query q is *contained* in query q', written as $q \subseteq q'$, if $q(D) \subseteq q'(D)$ for every database instance D. This assumes of course that queries return sets and the notion of subset is applicable to query outputs. This is a most basic task of reasoning about queries: note that containment is one half of the *equivalence* problem. Indeed, two queries q and q' are *equivalent*, denoted $q \equiv q'$, if both $q \subseteq q'$ and $q' \subseteq q$ are true. 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, we consider the following problem, again parameterized by a query language $\mathcal{L}$.**(Andreas)**[4]

PROBLEM:	$\mathcal{L}$-Containment
INPUT:	queries q and q' from $\mathcal{L}$
OUTPUT:	yes if $q \subseteq q'$ and no otherwise

(Marcelo)[5]

Note that for $\mathcal{L}$-Containment, the input to the problem consists of two queries. Most of the time, queries are much smaller objects than databases.

[4] **Andreas:** I suggest here to properly define also the equivalence problem. I think we should treat containment and equivalence equally, and only the satisfiability problem is defined locally.

[5] **Marcelo:** With the current version, this will be defined for finite and infinite database (recall finite controllability).

Thus, 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 inputs.

Further Background Reading

Should the reader find himself/herself in a situation "that he/she does not have the prerequisites for reading the prerequisites" [14], rather than being discouraged the reader 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, [11, 27, 30]. For additional information about computability theory, we provide a primer in Appendix A. Furthermore, we refer the reader to [17, 21, 31]; standard texts on complexity theory are [2, 26, 35]. For foundations of finite model theory and descriptive complexity, the reader is referred to [12, 18, 24].

Table of Notation

Hard table:

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

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

Symbol(s)	Usual meaning
D	database
$R, S, \dots$	relation names
S	sometimes a set
$A, B, \dots$	attribute names
$U, V, \dots$	sets of attribute names
$\mathbf{S}$	database schema

Wim:
This is just an attempt and a first try. Let's see. We may first fill the second table as some sort of inventory. Then we can see if we want to make conventions.

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 most commonly are based on *first-order logic*, that often appears under the name *relational calculus* in database literature. In a procedural language, on the other hand, one specifies how 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, abbreviated FO.

Syntax of First-Order Logic

Recall that a schema $\mathbf{S}$ is a set of relation names, and each relation has an arity. Assume a countably infinite set of variables. Variables will be typically denoted by $x, y, z, \ldots$, with subscripts and superscripts. Formulae of *first-order logic* (FO) are inductively defined using variables, conjunction ($\wedge$), disjunction ($\vee$), negation ($\neg$), existential quantification ($\exists$), and universal quantification ($\forall$).

We inductively define terms and formulae of FO over schema $\mathbf{S}$ as follows:

- Each variable $x \in \mathsf{Var}$ is a term.
- Each value $a \in \mathsf{Const}$ is a term, called a *constant.*
- If x and y are variables and $a \in \mathsf{Const}$, then $x = y$ and $x = a$ are atomic formulae.
- If $u_1, \ldots, u_k$ are terms (not necessarily distinct) and R is a k-ary relation symbol 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 is a variable, then $(\exists x\, \varphi)$ and $(\forall x\, \varphi)$ are formulae.

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

The set of *free* variables of a formula φ, denoted $\mathrm{FV}(\varphi)$, is defined as follows:

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

If $x \in \mathrm{FV}(\varphi)$, we call it a *free variable (of φ).* Variables that are not free are called *bound*.

Example 3.1. Let us assume that we have the following (named) relational 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`).

We give some examples of FO formulae over this schema. Consider first the FO formula:

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

The set of free variables of this formula is $\{x\}$. Consider now the FO formula

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

The set of free variables of this formula is $\{x, y\}$. Finally, consider the formula

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

The set of free variables of this formula is $\{x, y, z\}$. □

Semantics of First-Order Logic

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

If the formula φ is satisfied in D under assignment η, we write $(D, \eta) \models \varphi$. This is now defined inductively:

- If φ is $x = y$, then $(D, \eta) \models \varphi$ iff $\eta(x) = \eta(y)$.
- If φ is $x = a$, then $(D, \eta) \models \varphi$ iff $\eta(x) = a$.
- If φ is $R(u_1, \ldots, u_k)$, then $(D, \eta) \models \varphi$ iff $R(\eta(\bar{u}))$ is a fact of D or, equivalently, $\eta(\bar{u}) \in R^D$.
- $(D, \eta) \models \neg\varphi$ iff $(D, \eta) \models \varphi$ does not hold.
- If $\varphi = \varphi_1 \wedge \varphi_2$, then $(D, \eta) \models \varphi$ iff $(D, \eta) \models \varphi_1$ and $(D, \eta) \models \varphi_2$.
- If $\varphi = \varphi_1 \vee \varphi_2$, then $(D, \eta) \models \varphi$ iff $(D, \eta) \models \varphi_1$ or $(D, \eta) \models \varphi_2$.
- If φ is $\exists x\, \psi$, then $(D, \eta) \models \varphi$ iff there exists some $a \in \text{Dom}(D) \cup \text{Dom}(\varphi)$ such that $(D, \eta[x/a]) \models \psi$.
- If φ is $\forall x\, \psi$, then $(D, \eta) \models \varphi$ iff $(D, \eta[x/a]) \models \psi$ for each $a \in \text{Dom}(D) \cup \text{Dom}(\varphi)$.

An FO formula φ without free variables is called a *sentence*. For such a sentence φ, the function η has an empty domain (it is defined for the free variables of ψ) and therefore it is unique. For this unique η, it is either the case that $(D, \eta) \models \varphi$ or not. If the former is the case, we simply write $D \models \varphi$ and say that D *satisfies* φ.

Example 3.2. We provide an intuitive description of the semantics of the formulae in Example 3.1:

- Formula (3.1) is satisfied by all x where 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. □

We 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: For the latter one would allow η to associate arbitrary elements of Const to variables, whereas we only allow elements of $\mathrm{Dom}(D) \cup \mathrm{Dom}(\varphi)$. What we defined is the so-called *active domain semantics*, which is standard in the database literature as it ensures an important property known as *safety*, which we discuss in more detail at the end of the chapter. A crucial property of the active domain semantics is that each FO formula only has a *finite* number of satisfying assignments.

Notation

Since conjunction is associative, we will omit brackets in long conjunctions and write, e.g., $x_1 \wedge x_2 \wedge x_3 \wedge x_4$ 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. Additionally, 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 $\forall \bar{x}$.

Also, we assume that $\wedge$ binds the strongest, followed by $\vee$, then $\neg$, and finally quantifiers. This means that, when we write $\exists x\, \neg R(x) \wedge S(x)$, 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. That is, 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

The way FO is commonly defined, some constructors are redundant. For instance, we know that by De Morgan's laws $\neg(\varphi \vee \psi)$ 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 use just $\vee, \neg$, 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 it will be convenient to assume that constants do not appear in relational atoms. We can always rewrite FO formulae to such a form by adding 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 is a syntactic object that produces a k-ary relation $q(D) \subseteq \mathrm{Dom}(D)^k$, 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 $R(x, x, y)$. It has two free variables, x and y, but we may view it as a query with output (x, x, y) to indicate that it will produce results consisting of triples from relation R, where the first and the second component coincide.

More generally, an *FO query* over 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 *some* tuple that mentions precisely the free variables in φ (perhaps with repetitions). More precisely, $\bar{x}$ is a tuple that uses only variables from $\mathrm{FV}(\varphi)$ and uses each such variable at least once. Given a tuple $\bar{a}$ of elements from $\mathrm{Dom}(D) \cup \mathrm{Dom}(\varphi)$, for a database D, we say that *D satisfies the query $\varphi(\bar{x})$ using the values $\bar{a}$*, denoted by $D \models \varphi(\bar{a})$, when there exists an assignment η such that $\eta(\bar{x}) = \bar{a}$ and $(D, \eta) \models \varphi$. When $\bar{x}$ is clear from the context, we sometimes also denote the query $\varphi(\bar{x})$ as φ. Assuming that the arity of $\bar{x}$ is k, the *output* of φ on D is then defined as

$$\varphi(D) \; = \; \{\bar{a} \mid D \models \varphi(\bar{a})\}\,.$$

By definition, $\varphi(D) \subseteq (\mathrm{Dom}(D) \cup \mathrm{Dom}(\varphi))^k$, and hence $\varphi(\bar{x})$ defines a k-ary query over $\mathbf{S}$. We leave the proof that $\varphi(\bar{x})$ defines a query as an exercise. As an example, consider the FO query $\varphi(x, y)$ where $\varphi = R(x, x, y)$. On a database with facts $R(1, 1, 2)$ and $R(3, 4, 5)$ it will return tuple $(1, 2)$. As a second example, consider the FO query $\varphi(x, x, y)$ where $\varphi = \exists z \; R(x, y, z)$. Even though φ has two free variables, x and y, we allow the repetition of x to form the output of the query. On a database with facts $R(1, 2, 3)$ it will therefore return tuple $(1, 1, 2)$ in which value 1, corresponding to variable x, will be repeated.

To emphasize that these formulae define database queries, we shall often use notation $q(\bar{x})$ for them. Again, when $\bar{x}$ is clear from the context, we sometimes omit it. In particular, we may write that $q = \varphi(\bar{x})$ is a query.

Example 3.3. Figure 3.1 contains a database instance D of the schema in Example 3.1.

- Consider the query $\varphi_1(x)$, where φ_1 is formula (3.1). The output of φ_1 on D is $\varphi_1(D) = \{\text{'1'}, \text{'3'}, \text{'4'}\}$.
- Consider the query $\varphi_2(x, y)$, where φ_2 is formula (3.2). The output of φ_2 on D is $\varphi_2(D) = \{(\text{'1'}, \text{'Billie'})\}$.
- Consider the query $\varphi_3(y)$, where φ_3 is formula (3.3). The output of φ_3 on D is $\varphi_3(D) = \{\text{'Aretha'}, \text{'Bob'}\}$. □

`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 small database instance of the schema in Example 3.1.

Boolean FO queries

FO sentences, i.e., FO formulae without free variables, are used to define Boolean queries, i.e., queries that return `true` or `false`. For such a reason they are called *Boolean FO queries*.

By definition, the output of a query q over a database D corresponds to a set of tuples of elements in $\mathrm{Dom}(D)$. Boolean FO queries will be no exception to this. In fact, we can consider them to be of the form $q()$, where $()$ denotes the empty tuple, as they have no free variables. Thus, there are only two possibilities for $q(D)$:

- either it consists precisely of the empty tuple, 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

$$q(D) = \texttt{true} \quad \Longleftrightarrow \quad D \models q \quad \Longleftrightarrow \quad q(D) = \{()\},$$

and $q(D) = \texttt{false}$ otherwise.

Safety of FO 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 free variables in φ to the finite set $\mathrm{Dom}(D) \cup \mathrm{Dom}(\varphi)$. This means that the output of φ on a database D is always finite, which is important for $\varphi(\bar{x})$ 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, i.e., use assignments η that map to Const instead of to $\mathrm{Dom}(D)$, the set $\varphi(D) = \{\bar{a} \mid D \models \varphi(\bar{a})\}$ could be infinite and therefore unsafe. For instance, under such semantics the formula $\neg R(x)$ 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 result of a query should be. An alternative procedural language prescribes a way to implement such a declarative query and obtain the result of a query by a sequence of operations on data. The standard such language is *relational algebra*, abbreviated RA. We present definitions for the 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 RA* and *named RA*. Since we will usually be working over the unnamed perspective in this book, we will often abbreviate "unnamed RA" as "RA".

Syntax of the Unnamed Relational Algebra

Under the unnamed perspective, the relational algebra consists of five primitive operations: selection, projection, Cartesian product, union, and difference. We inductively define RA expressions and their associated arities over a schema **S** as follows:

Base Expressions: If R is a k-ary relation symbol 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 with arity 1.

Selection: If e is an RA expression of arity $k \geq 0$ and θ is a *condition over* $\{1, \ldots, k\}$, then $\sigma_\theta(e)$ is an RA expression of arity k. Conditions over $\{1, \ldots, k\}$ are defined as Boolean combinations of statements $i \doteq j$, $i \doteq a$, $i \neq j$, and $i \neq a$ for $a \in \mathsf{Const}$ and $1 \leq i, j \leq k$.

Projection: If e is an RA expression of arity $k \geq 0$ and $\alpha = (i_1, \ldots, i_m)$ is a list of numbers from $\{1, \ldots, k\}$, then $\pi_\alpha(e)$ is an RA expression of arity m. We allow $m = 0$, in which case α is the empty list (). This will be useful for expressing Boolean queries.

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

Union: If e_1 and e_2 are RA expressions of the same arity $k \geq 0$, then their union $(e_1 \cup e_2)$ is an RA expression of arity k.

Difference: If e_1 and e_2 are RA expressions of the same arity $k \geq 0$, then their difference $(e_1 - e_2)$ is an RA expression of arity k.

A condition of the form $i \doteq j$ 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 $i \neq j$ is used to indicate that these values must be different. Notice that we use symbols $\doteq$, $\neq$ instead of $=$, $\neq$ to avoid writing statements such as "1 = 2", which are likely to confuse readers. 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 $i \neq j$, i.e., Boolean combinations using only conjunction $\wedge$ and disjunction $\vee$. For example, $\neg\big((1 \doteq 2) \vee (2 \neq 3)\big)$ is equivalent to $(1 \neq 2) \wedge (2 \doteq 3)$.

Semantics of Unnamed Relational Algebra

We now define the semantics of RA expressions. As a preparatory step, we define the operation of projection and the notion of satisfaction of conditions over tuples.

If $\bar{a} = (a_1, \ldots, a_k)$ is a tuple of values from Const and $\alpha = (i_1, \ldots, i_m)$ is a list of numbers from $\{1, \ldots, k\}$, then the projection $\pi_\alpha(\bar{a})$ is defined to be $(a_{i_1}, a_{i_2}, \ldots, a_{i_m})$. For example, $\pi_{(1,3)}(a, b, c, d) = (a, c)$, $\pi_{(1,3,3)}(a, b, c, d) = (a, c, c)$, and $\pi_{()}(a, b, c, d) = ()$.[1]

We inductively define that a tuple $\bar{a}$ *satisfies the condition* θ, denoted $\bar{a} \models \theta$, as follows:

[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.

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

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

- If $e = R$, where R is a relation symbol 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 $\{1, \dots, k\}$, then $e(D)$ is the k-ary relation $\{\bar{a} \mid \bar{a} \in e_1(D) \text{ and } \bar{a} \models \theta\}$.
- If $e = \pi_\alpha(e_1)$, where e_1 is an RA expression of arity $k \geq 0$ and $\alpha = (i_1, \dots, i_m)$ is a list of numbers from $\{1, \dots, k\}$, then $e(D)$ is the m-ary relation $\{\pi_\alpha(\bar{a}) \mid \bar{a} \in e_1(D)\}$.
- If $e = (e_1 \times e_2)$, for RA expressions e_1 and e_2 of arity $k \geq 0$ and $\ell \geq 0$, respectively, then $e(D) = e_1(D) \times e_2(D)$, where the latter represents the $(k + \ell)$-ary relation defined as

$$\{(a_1, \dots, a_k, b_1, \dots, b_\ell) \mid (a_1, \dots, a_k) \in e_1(D) \text{ and } (b_1, \dots, b_\ell) \in e_2(D)\}.$$

- 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 $e = (e_1 - e_2)$, 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 $\{1, \dots, k+m\}$, the *θ-join* of e_1 and e_2 is denoted $e_1 \bowtie_\theta e_2$. Its evaluation over a database D is defined as

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

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

Example 4.1. Consider again the (named) relational schema:

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

The RA expression

$$\pi_{(1)}\big(\sigma_{5\not\doteq 7}\big((\text{Person} \bowtie_{1\doteq 4} \text{Profession}) \bowtie_{1\doteq 6} \text{Profession}\big)\big)$$

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 have the profession author or actor. □

Syntax of the Named Relational Algebra

Under the named perspective, the presentation changes a bit. Formally, we inductively define named RA expressions and their associated sorts over a schema $\mathbf{S}$ as follows:

Base Expressions: If R is a relation symbol from $\mathbf{S}$, then R is an atomic named RA expression over $\mathbf{S}$ of sort $\text{sort}(R)$. If furthermore $a \in \mathsf{Const}$ 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. Conditions over U are defined as Boolean combinations of statement $A \doteq B$, $A \doteq a$, $A \not\doteq B$, and $A \not\doteq a$, where a is from Const and A and B are from 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. Notice the contrast with the unnamed perspective, where α is a list in which we allow repetitions.

Join: If e_1 and 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 with attribute $A \in U$ but attribute $B \notin U$, then $\rho_{A\to B}(e)$ is a named RA expression of sort $(U - \{A\}) \cup \{B\}$.

Union: If e_1 and 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 and 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.

Semantics of the Named Relational Algebra

The semantics of named RA expressions is defined similarly as in the unnamed case. For this reason, we only discuss rename and join, and leave the others as exercises.

- If $e = \rho_{A\to B}(e_1)$, where e_1 is a named RA expression of sort U, with $A \in U$ and $B \notin U$, 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 data at all, it only changes names of attributes of a relations. 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 $\big(R \cup \rho_{B\to A}(S)\big)$.

The other new primitive operator in the named perpective is *join* (sometimes also called *natural join* in the literature). It is simply a join of two relations on the condition that their common attributes are the same. Formally, it is defined as follows:

- 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 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 4.2. We provide named RA versions for the expressions in Example 4.1. The expression
$$\pi_{\{\text{pid}\}}\big(\sigma_{\text{prname}\,\dot{\neq}\,\text{prname2}}\big((\text{Person} \bowtie \text{Profession}) \bowtie \rho_{\text{prname}\to\text{prname2}}(\text{Profession})\big)\big)$$
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 have the profession author or actor. □

Expressiveness of Named and Unnamed Relational Algebra

We often use the named version of RA in examples, because it is closer to how we think about real-life databases. On the other hand, many results are easier to state and prove in the unnamed RA. This comes at no cost as every query that be expressed in named RA can also be expressed in unnamed RA and vice versa. More formally, let f be the function that converts a database instance D from the named to the unnamed perspective, as presented in the Background chapter. Recall that this converts each tuple $t = (A_1 : a_1, \ldots, A_k : a_k)$ in $D(R)$, for a relation symbol R of sort $\{A_1, \ldots, A_k\}$ (with $A_1 <_{\mathsf{Att}} \cdots <_{\mathsf{Att}} A_k$), into a tuple $t' = (a_1, \ldots, a_k)$ in $f(D)(R)$.

Let q_{n} be a named relational algebra query and q_{u} an unnamed relational algebra query. We call q_{n} *equivalent to* q_{u} *under* $\boldsymbol{S}$ if, for every database instance D of $\mathbf{S}$ it holds that

$$f(q_{\mathsf{n}}(D)) \;=\; q_{\mathsf{u}}(f(D)).$$

Notice that two queries can be equivalent under one schema and inequivalent under another one. This cannot be avoided, as the order inside an unnamed tuple depends on the names of the attributes (the order is defined by $\leq_{\mathsf{Att}}$). Therefore 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 to an equivalent unnamed RA query. We leave the statement of the reverse direction and its proof as an exercise.

Theorem 4.3. *Given a named relational schema* $\boldsymbol{S}$ *and a named RA query* q_{n}*, there exists an unnamed RA query* q_{u} *that is equivalent to* q_{n} *under* $\boldsymbol{S}$*.*

Proof. We prove this by induction on q_{n}. Assume that q_{n} has sort $\{A_1, \ldots, A_k\}$ with $A_1 <_{\mathsf{Att}} \cdots <_{\mathsf{Att}} A_k$. We will show 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}.

We now proceed with the induction. Whenever we write a set of attributes as a set $\{A_1, \ldots, A_k\}$, we assume that $A_1 <_{\mathsf{Att}} \cdots <_{\mathsf{Att}} A_k$.

- If $q_{\mathsf{n}} = R$, for a relation symbol R in $\mathbf{S}$ of sort $\{A_1, \ldots, A_k\}$, then $q_{\mathsf{u}} = R$. If $q_{\mathsf{n}} = \{(A : a)\}$ then $q_{\mathsf{u}} = \{a\}$.
- Let $q_{\mathsf{n}} = \sigma_\theta(q'_{\mathsf{n}})$, where q'_{n} is a named RA expression of sort $U = \{A_1, \ldots, A_k\}$. Suppose that (q'_{u}, f') is the unnamed RA expression of arity k and the function that corresponds to the translation of q'_{n} obtained by the induction hypothesis. 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 $i \in [1, k]$. For instance, if θ is $(A_1 \doteq A_3) \wedge (A_2 \neq b)$, then θ' is $(1 \doteq 3) \wedge (2 \neq b)$.

- Let $q_{\mathsf{n}} = \pi_{\alpha}(q'_{\mathsf{n}})$, where q'_{n} is a named RA expression of sort $U = \{A_1, \ldots, A_k\}$ and $\alpha \subseteq U$. Then $q_{\mathsf{u}} = \pi_{\alpha'}(q'_{\mathsf{u}})$, where q'_{u} is the unnamed RA expression of arity k that corresponds to the translation of q'_{n} obtained by induction hypothesis, and α' is the list of all $i \in [1, k]$ with $A_i \in \alpha$.
- Let $q_{\mathsf{n}} = (q'_{\mathsf{n}} \bowtie q''_{\mathsf{n}})$. Suppose that q'_{n} and q''_{n} are of sort $U' = \{A'_1, \ldots, A'_k\}$ and $U'' = \{A''_1, \ldots, A''_\ell\}$, respectively, and that q'_{u} and q''_{u} are the unnamed RA expressions that are obtained by induction hypothesis for q'_{n} and q''_{n}, respectively. 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 $i \in [1, k]$ and $j \in [1, \ell]$. Finally, let $\{A_1, \ldots, A_m\} = U' \cup U''$ and let $f : [1, m] \to [1, k + \ell]$ be such that

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

  We now define $\alpha = (f(1), \ldots, f(m))$. That is, 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 $<_{\mathsf{Att}}$.
- Let $q_{\mathsf{n}} = \rho_{A \to B}(q'_{\mathsf{n}})$, where q'_{n} is a named RA expression of sort $U = \{A_1, \ldots, A_k\}$ and $A = A_i$ for some $i \in [1, k]$. Let q'_{u} be the query obtained from the induction hypothesis and let j be the cardinality of $\{i \mid A_i <_{\mathsf{Att}} B\}$. 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 $j = 0$.
- If $q_{\mathsf{n}} = q'_{\mathsf{n}} \cup q''_{\mathsf{n}}$, then $q_{\mathsf{u}} = q'_{\mathsf{u}} \cup q''_{\mathsf{u}}$. Analogously, for $q_{\mathsf{u}} = q'_{\mathsf{u}} - q''_{\mathsf{u}}$. □

5

Relational Algebra and SQL

In this chapter, we shed some light on the relationship between the 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 database theory already 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, starting with the very core. 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 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 relations 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. For a constant, one needs to provide the name of attribute: for example, 5 AS B will output 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 35. 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 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.

Returning to queries in Example 3.1, the first one of them returns people who have two different professions. This will be written as

```
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)
```

This faithfully translates the expression in 3.1. It mentions the relation Person once and the relation Profession twice, and so does the 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 (3.1); the last condition corresponds to $\neg(u_1 = u_2)$.

Moving to the next query (3.2) mb which asked for people whose cities of birth were not recorded in the City relation, it will be expressed as follows:

```
SELECT Person.pid
FROM Person
  EXCEPT
SELECT Person.pid
FROM Person, City
WHERE Person.cid = City.cid
```

The first subquery asks for all people, and the second for those that have a city of birth recorded, and EXCEPT is their difference.

Relational Algebra to Core SQL

We now show that (named) relational algebra queries can be encoded in SQL. Let e be a named RA expression. We inductively define an equivalent SQL query Q_e as follows.

If $e = R$, and relation R has attributes $A_1, \ldots, A_n$, then Q_e is `SELECT` $\mathtt{A}_1, \ldots, \mathtt{A}_n$ `FROM R`. In fact, SQL has a shorthand `*` for listing all attributes of a relation, and the above query can be written as `SELECT * FROM R`.

If $e = \{(A : a)\}$ then Q_e is simply `SELECT` a `AS A`.

Next, assume that Q_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 C_θ
```

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  A_{i_1}, ..., A_{i_k}
FROM  R₁, ..., Rₘ
WHERE  condition
```

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

Next assume that Q_e is translated into

```
SELECT  ..., Rᵢ.Aⱼ AS A, ...
FROM  R₁, ..., Rₘ
WHERE  condition
```

Then $\delta_{A \to B}(e)$ translated into

```
SELECT  ..., Rᵢ.Aⱼ AS A, ...
FROM  R₁, ..., Rₘ
WHERE  condition
```

To handle the join operator, assume that e is translated into

```
SELECT  A₁, ..., Aₖ, B₁, ..., Bₚ,
FROM  R₁, ..., Rₘ
WHERE  condition
```

and that e' is translated into

```
SELECT A$_1$, ..., A$_k$, C$_1$, ..., C$_s$,
FROM S$_1$, ..., S$_\ell$
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. Then $e \bowtie e'$ is translated into

```
SELECT $n_e$(A$_1$) AS A$_1$, ..., $n_e$(A$_k$) AS A$_k$,
       $n_e$(B$_1$) AS B$_1$, ..., $n_e$(B$_p$) AS B$_p$,
       $n_{e'}$(C$_1$) AS C$_1$, ..., $n_{e'}$(C$_s$) AS C$_s$,
FROM   R$_1$, ..., R$_m$, S$_1$, ..., S$_\ell$
WHERE  condition AND condition$'$
       AND $n_e$(A$_1$) = $n_{e'}$(A$_1$) AND ...  AND $n_e$(A$_k$) = $n_{e'}$(A$_k$)
```

where $n_e(A_i)$ is the name of the attribute that was renamed as A_i in the translation of e. That is, if we had `R.A AS A`$_i$ in that query, then $n_e(\mathtt{A}_i) = \mathtt{R.A}$, and the definition is similar for e'.

This can be easily seen in example. Consider relations $R(A, B, D)$, $S(B, C)$, and $T(A, C, D)$ and two queries

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

Finally, if $e = e_1 - e_2$ then $Q_e = (Q_{e_1})$ `EXCEPT` (Q_{e_2}), and if $e = e_1 \cup e_2$ then $Q_e = (Q_{e_1})$ `UNION` (Q_{e_2}).

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 a SQL query is executed on a DBMS. A declarative query is then 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  $\alpha_1$ AS B$_1$, ..., $\alpha_n$ AS B$_n$
FROM  R$_1$ AS S$_1$, ..., R$_m$ AS S$_m$
WHERE  condition
```

where all relations in FROM have been renamed so their names are different and each α_i is of the form $\texttt{S}_j.\texttt{A}_p$, i.e., one of the attributes of relations in the FROM clause. Let $\boldsymbol{\delta}_i$ be the sequence of renaming operators that rename each attribute A of $\texttt{R}_i$ to $\texttt{S}_i.A$. Let $\boldsymbol{\delta}_{\text{out}}$ 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 the relational algebra is

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

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 different of two relations R and S with one attribute A. Of course 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 express the difference of course. Query (3.2), which asked for people whose cities of birth were not recorded in the City relation, can also be written as:

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

These two forms of subqueries, using NOT IN and NOT EXISTS, correspond to adding two types of selection conditions to RA:

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

- empty(e), checking if the result of the evaluation of e is empty.

These selection conditions do not increase the expressiveness of RA; see Exercise 11.5.

In general, subqueries can be used in other clauses, and in fact they are very commonly used in FROM. For example, the query (3.1) in Example 3.1 asking for people who have two different professions can also be written as:

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

Here the join of Person and Profession occurs in the subquery in FROM, and then 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 11.6) 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 result 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 35.

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 25.

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 from the handling of values from Const, and even the notion of query answers changes in this case. These issues are discussed in detail in Chapters 30 and 31.

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 25) this is a subject that we do not delve into in this book.

6

Equivalence of Logic and Algebra

We will prove that the power of the declarative language FO and the procedural query language RA is the same, 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. *Let $\boldsymbol{S}$ be a database schema.*

(a) For every RA expression e over schema $\boldsymbol{S}$, there exists an FO query $\varphi_e(\bar{x})$ such that $\varphi_e(D) = e(D)$ for every database D of $\boldsymbol{S}$.
(b) For every FO query $\varphi(\bar{x})$ over $\boldsymbol{S}$, there exists an RA expression e_φ such that $e_\varphi(D) = \varphi(D)$ for every database D of $\boldsymbol{S}$.

In the proof, we need to substitute variables in formulas. 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 $\{x_1, \ldots, x_n\}$. The notation abbreviates $\exists x_1 \cdots \exists x_n \varphi$. Notice that the ordering of quantification is irrelevant for the semantics of this formula.

From RA to FO

We show part (a) of Theorem 6.1 by induction on the structure of e. The base cases are the following:

- If e is $R \in \mathbf{S}$, then the FO query is $\varphi_e(x_1, \ldots, x_{\mathrm{ar}(R)})$, where φ_e is the formula
$$R(x_1, \ldots, x_{\mathrm{ar}(R)})$$
and all variables in $x_1, \ldots, x_{\mathrm{ar}R}$ are different.
- If e is $\{a\}$ with $a \in \mathsf{Const}$, then the FO query is $\varphi_e(x)$, where φ_e is the formula
$$(x = a).$$

We now do the induction step. Assume that e and e' are RA expressions over schema $\mathbf{S}$ 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 define a formula $\theta[\bar{x}]$ as follows:
  - if θ is $i \doteq j$, $i \doteq a$, $i \not\doteq j$, or $i \not\doteq a$, then $\theta[\bar{x}]$ is $x_i = x_j$, $x_i = a$, $x_i \neq x_j$, or $x_i \neq a$, respectively;
  - if $\theta = \theta_1 \wedge \theta_2$, then $\theta[\bar{x}] = \theta_1[\bar{x}] \wedge \theta_2[\bar{x}]$; and
  - if $\theta = \theta_1 \vee \theta_2$, then $\theta[\bar{x}] = \theta_1[\bar{x}] \vee \theta_2[\bar{x}]$.

  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 $[1,k]$. 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

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

  Notice that, if α has repetitions, then $(x_{i_1},\ldots,x_{i_p})$ has repeated variables. For instance, 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 this construction, the inductive proof of the equivalence of the queries e and $\varphi_e(\bar{x})$, to the reader. This concludes the proof of part (a).

From FO to RA

For proving part (b) of Theorem 6.1, we assume that relational atoms do not contain constants, which we observed in Chapter 3 is always possible. We also consider a slight generalization of FO queries that will simplify the induction: we will also allow $\varphi(x_1,\ldots,x_n)$ to be an FO query if the free variables of φ are a subset of $\{x_1,\ldots,x_n\}$. The semantics of such a formula $\varphi(x_1,\ldots,x_n)$ 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 φ 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*, i.e., it is of the form $Q_k \cdots Q_1 \varphi'$ where

- φ' is quantifier-free and has (free) variables $y_1, \ldots, y_m$,
- each Q_j is of the form $\exists y_j$ or $\neg\exists y_j$,
- $x_1, \ldots, x_n = y_{k+1}, \ldots, y_m$, and
- φ' 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 the i-fold Cartesian product $\mathrm{Adom} \times \cdots \times \mathrm{Adom}$, for $i \in \mathbb{N}$.

We construct for each subformula ψ of φ an RA query e_ψ. The induction hypothesis is that, for each subformula ψ of φ', the expression e_ψ has arity m and is equivalent to the FO query $\psi(y_1, \ldots, y_m)$. For all other subformulae of φ, it holds that $\psi(y_{j+1}, \ldots, y_m) = Q_j \cdots Q_1 \varphi'$ for some j, expression e_ψ has arity $m - j$, and e_ψ is equivalent to the FO query $Q_j \cdots Q_1 \varphi'(y_{j+1}, \ldots, y_m)$. The inductive construction defines

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

Finally, the RA query equivalent to $\varphi(x_1, \ldots, x_n)$ is e_φ. We leave the the inductive proof of the equivalence of the queries to the reader.

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 verifying $\bar{a} \in q(D)$ for an FO query q, a database D, and a tuple $\bar{a}$ over $\mathrm{Dom}(D)$.

Combined Complexity

We first look at the combined complexity of the problem, that is, when the input consists of the query q, the database D, and the candidate output $\bar{a}$.

Theorem 7.1. FO-Evaluation *is* PSPACE-*complete.*

Proof. We start with the upper bound. Since FO formulae can have atomic formulae of the form $x = a$, we can assume that relational atoms in queries do not contain constants (see also Chapter 3). Consider an FO query $q = \varphi(\bar{x})$, a database D, and a tuple $\bar{a}$ over $\mathrm{Dom}(D)$. We can assume without loss of generality that q and $\bar{a}$ have the same arity and that φ uses only $\neg$, $\vee$, and $\exists$ (Exercise 11.1).

Let $\bar{x} = (x_1, \dots, x_n)$ and $\bar{a} = (a_1, \dots, a_n)$. We first check if the variable assignment η defined by setting $\eta(x_i) = a_i$ for all $i \in [n]$ is well-defined (e.g., if $x_i = x_j$, then $a_i = a_j$ for every $i, j \in [n]$). If this is not the case, then we declare $\bar{a} \notin \varphi(D)$. Otherwise we run the recursive procedure $\textsc{Evaluation}(\varphi, D, \eta)$ defined in Algorithm 1 for $\eta(\bar{x}) = \bar{a}$, and declare $\bar{a} \in q(D)$ if and only if it returns `true`. In the description of the procedure, we assume that statements such as $R(\eta(\bar{x})) \in D$ evaluate to `true` or `false` and that $\max\{\texttt{true}, \texttt{false}\} = \texttt{true}$.

Clearly, the procedure is sound and complete. We will argue why it can be implemented to run in polynomial space via a recursive algorithm. Assume that Algorithm 1 is started with formula φ_0. We will argue that it can be implemented to run in space $O(\log(\|D\|) \cdot \|\varphi_0\|^2)$. It is easy to see that the recursion depth of the algorithm is $O(\|\varphi_0\|)$ and that $\|\eta\| \leq \log(\|D\|) \cdot \|\varphi_0\|$ at each call of $\textsc{Evaluation}(\varphi, D, \eta)$. We can store η in the work tape as an array

Algorithm 1 EVALUATION(φ, D, η)

```
Input: An FO formula $\varphi$, a database $D$, and an assignment $\eta$ for the free variables
    of $\varphi$ over $\mathrm{Dom}(D)$.
Output: true if $(D, \eta) \models \varphi$ and false otherwise.
1: if $\varphi$ is of the form $R(\bar{x})$ then return $R(\eta(\bar{x})) \in D$
2: else if $\varphi$ is of the form $(x_i = x_j)$ then return $\eta(x_i) = \eta(x_j)$
3: else if $\varphi$ is of the form $(x_i = a)$ then return $\eta(x_i) = a$
4: else if $\varphi$ is of the form $\neg\varphi'$ then return $\neg$EVALUATION($\varphi', D, \eta$)
5: else if $\varphi$ is of the form $\varphi' \vee \varphi''$ then
6:     return max {EVALUATION($\varphi', D, \eta$), EVALUATION($\varphi'', D, \eta$)}
7: else if $\varphi$ is of the form $\exists x\, \varphi'$ then
8:     return $\max_{a \in \mathrm{Dom}(D)}$ {EVALUATION($\varphi', D, \eta[x/a]$)}
9:                                  ▷ $\eta[x/a]$ extends $\eta$ by setting $\eta(x) = a$.
```

of at most $\|\varphi_0\|$ values in $\mathrm{Dom}(D)$, in the same order as their appearance in φ_0. **(Wim)**[1]

We now prove by induction on formulae that the space we use in a single recursive call is at most $2(\log(\|D\|) \cdot \|\varphi_0\|)$.

- When $\varphi = R(\bar{x})$, $\varphi = (x_i = x_j)$, or when $\varphi = (x_i = a)$, we argue that we can use space $\log(\|D\|) \cdot \|\varphi_0\| + \|\eta\| \leq 2\log(\|D\|) \cdot \|\varphi_0\|$. Indeed, to check whether $R(\eta(\bar{x})) \in D$ it suffices to transfer from the input tape to the work tape each fact in D of the form $R(\bar{a})$, one-by-one. This uses extra space $r \cdot \log(\|D\|)$ in addition to the space $\log(\|D\|) \cdot \|\varphi_0\|$ that is used for storing η, where r is the maximum arity of a relation symbol in φ_0. Clearly, $r \leq \|\varphi_0\|$ and thus the extra space needed is at most $\log(\|D\|) \cdot \|\varphi_0\|$. For each such a fact we only have to check whether $\bar{a} = \eta(\bar{x})$, which can be done directly in the work tape without using any extra space. Checking $\eta(x_i) = \eta(x_j)$ or $\eta(x_i) = a$ is even simpler, as we do not need to transfer any extra information from the input tape.

We now assume that checking EVALUATION(φ', D, η) and EVALUATION(φ'', D, η) can both be done in space $2(\log(\|D\|) \cdot \|\varphi_0\|)$ by induction hypothesis.

- Checking if EVALUATION(φ, D, η) = `true` when $\varphi = \neg\varphi'$ can clearly be done in the same space than EVALUATION(φ', D, η) = `false`, which is $2(\log(\|D\|) \cdot \|\varphi_0\|)$.
- When $\varphi = \varphi' \vee \varphi''$, we can first check whether EVALUATION(φ', D, η) = `true` and, if this is not the case, reuse the space to check whether EVALUATION(φ'', D, η) = `true`. This procedure uses the maximum of the space of the two subprocedures, which is $2(\log(\|D\|) \cdot \|\varphi_0\|)$.

[1] **Wim:** Furthermore, we don't really store the values, do we? We store natural numbers where i encodes the i-th value we see on the input tape.

- Finally, when $\varphi = \exists x\, \varphi'$, we check for each $a \in \mathrm{Dom}(D)$ – one-by-one, and following some predetermined order – whether EVALUATION$(\varphi', D, \eta[x/a])$ holds. Again, the recursion reuses the space needed to check this for each $a \in \mathrm{Dom}(D)$. To this end, we need to remember the current such value, so that we can correctly move on to the next. However, this current value is always stored in the array representing $\eta[x/a]$ that we used in the deeper recursive call, so we don't need to store it separately. Therefore, we use space $2(\log(\|D\|) \cdot \|\varphi_0\|)$.

The total space used then is the recursion depth times the space used by each recursive call, which corresponds to $O(\log(\|D\|) \cdot \|\varphi_0\|^2)$. This is polynomial in the size of the input. The recursion depth is important as any instantiation of the algorithm requires a stack of size $\log(\|\varphi_0\|)$ in order to preserve local information across recursive calls. **(Wim)**[2]

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

$$\theta := \exists \bar{x}_1 \forall \bar{x}_2 \exists \bar{x}_3 \ldots Q_n \bar{x}_n\, \alpha(\bar{x}_1, \ldots, \bar{x}_n),$$

where Q_n is $\forall$ if n is even and $\exists$ if n is odd. We assume without loss of generality that α is given in negation normal form, i.e., negations only occur immediately in front of variables. We construct a database $D = \{\mathsf{Zero}(0), \mathsf{One}(1)\}$ and a Boolean FO query

$$\varphi := \exists \bar{x}_1 \forall \bar{x}_2 \exists \bar{x}_3 \ldots Q_n \bar{x}_n\, \alpha'(\bar{x}_1, \ldots, \bar{x}_n),$$

where α' is obtained from α by changing each occurrence of the literal x by $\mathsf{One}(x)$ and each occurrence of the literal $\neg x$ by $\mathsf{Zero}(x)$. It is not hard to see that θ is true if and only if $D \models \varphi$ (we leave the proof as Exercise 11.7). □

Wim:
Exercise suggestion: Prove that evaluation is also in PSPACE if the input tuple $\bar{a}$ is over Const instead of over $\mathrm{Dom}(D)$. I didn't think this through but it would surprise me if this would not hold. I tried to adapt our proof immediately, but it became uglier, so I let it alone. The issue is encoding the database values and other constants in the input, which takes extra care. You get these ugly cases like: what happens if the DB is really small, etc.

Theorem 7.1 implies that the set $q(D)$, for an FO query q and database D, can also be computed in polynomial space: We can iterate over all tuples $\bar{a}$ of elements in $\mathrm{Dom}(D)$ that are of the same arity than q, and for each one of them run the procedure EVALUATION(q, D, η). Each such tuple for which the procedure returns `true` can then be written to the output. Notice

[2] **Wim:** What is meant here? I don't understand this. Our current bound is precisely stack depth times local function call size.

that this argument does not take the size of the output into account, which is the standard complexity-theoretic way of measuring the space usage of computable functions, see Appendix A.

Data Complexity

How can it be that databases are successful in practice, even though Theorem 7.1 proves that their most essential computational problem is PSpace-complete, a class that we usually 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 database query. In fact, the database $D = \{\mathsf{Zero}(0), \mathsf{One}(1)\}$ only has two tuples, whereas the query q can become arbitrarily large.

This is in strong contrast to how we usually experience databases in practice, where the data is orders of magnitude smaller than queries and, therefore, data and queries play different roles for the complexity of evaluation. In this light, it makes sense to study the *data complexity* of evaluation, that is, consider the query q to be fixed and only take the database D and the tuple $\bar{a}$ as inputs.

As we see in the complexity analysis of the algorithm Evaluation(φ, D, η), fixing the query φ indeed has a big impact in the complexity of evaluation, which becomes solvable in space $O(\log(\|D\|))$, that is, in a low-complexity parallelizable class.

Theorem 7.2. FO-Evaluation *is in* DLogSpace *in data complexity.*

The exact data complexity of evaluation for FO lies in the complexity class AC_0, which is properly contained in DLogSpace. The class AC_0 consists of those languages which are accepted by polynomial-size circuits of constant depth and unbounded fan-in. This is 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 semantically not meaningful since its evaluation 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:	query q from $\mathcal{L}$
OUTPUT:	yes if there is a database D such that $q(D) \neq \emptyset$ and no otherwise

The following theorem, known as Trakhtenbrot's Theorem, is due to Boris Trakhtenbrot and goes back to the 1950s.

Theorem 8.1. FO-Satisfiability *is undecidable.*

Proof. The proof is by reduction from the halting problem for 1-tape Turing machines; details on Turing Machines can be found in Appendix A. Let $M = (Q, \Sigma, \delta, s)$ be a (deterministic) 1-tape Turing machine. The problem of deciding whether M 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 relations 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

$$q_{\prec} \;\wedge\; q_{\text{first}} \;\wedge\; q_{\text{succ}} \;\wedge\; q_{\text{comp}},$$

where $q_{\prec}$, q_{first} and q_{succ} are Boolean FO queries that are responsible for defining the relations $\prec$, First and Succ, respectively, while q_{comp} is a Boolean FO query responsible for simulating the computation of M on the empty word. The definitions of the above queries follow. For the sake of readability, we write $x \prec y$ instead of the formal $\prec(x,y)$.

The Query $q_{\prec}$

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

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

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 Query q_{first}

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

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

The Query q_{succ}

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

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

The Query q_{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 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, q_{comp} is defined as the Boolean query

$$\exists x_1 \cdots \exists x_k \Bigg(\textstyle\bigwedge_{1 \le i < j \le k} \neg(x_i = x_j) \;\wedge\; q_{\text{start}}(x_1) \;\wedge\; q_{\text{consistent}}(x_1, \ldots, x_k) \;\wedge$$
$$q_\delta(x_1, \ldots, x_k) \;\wedge\; q_{\text{halt}}(x_2, x_3) \Bigg),$$

where

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

The definitions of the above queries follow.

The Query q_{start}. It is defined as the conjunction of the following FO queries, expressing that the first tape cell contains the left marker

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

the rest of tape cells contain the blank symbol

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

the head points to the first cell

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

and the machine M is in state s

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

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

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

$$\forall x \left(\left(\bigvee_{1 \le i \le k} \text{State}(x, x_i) \right) \ \wedge \bigwedge_{1 \le i < j \le k} \neg\big(\text{State}(x, x_i) \wedge \text{State}(x, x_j)\big) \right),$$

each tape cell y contains exactly one symbol

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

and the head points at exactly one cell

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

The Query q_δ. It is defined as the conjunction of the following FO queries: for each pair $(p_i, a) \in (Q - \{\text{"yes"}, \text{"no"}\}) \times \Sigma$ with $\delta(p_i, a) = (p_j, b, \text{dir})$,

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

where

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

The Query q_{halt}. Finally, this query 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. □

It is important to clarify that the proof of Theorem 8.1 relies on the fact that databases are *finite*. If we would allow a database to be infinite, i.e., contain infinitely many facts, the proof given above does not work, although the satisfiability problem for FO queries remains undecidable in this case. **(Wim)**[1] Indeed, assuming that the Turing machine M does not halt on the empty word, it is possible to construct an infinite database D such that $q_M(D) = \texttt{true}$ (we leave this as Exercise 11.9).

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

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 the fact that the satisfiability problem for FO queries is undecidable (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 now the FO query

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

which is trivially unsatisfiable. It is not hard 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 corollary of the fact that FO and RA have the same expressive power and Theorem 8.3.

Corollary 8.4. RA-Containment *and* RA-Equivalence *are undecidable.*

[1] **Wim:** @author: Discuss the undecidability in the infinite case in the bibliographic remarks of Part I and refer to it?

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 both 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 the notion of domain applies to sets S of relational atoms as well: $\mathrm{Dom}(S)$ is thus the set of constants and variables occurring in a set of atoms S; e.g., $\mathrm{Dom}(\{R(a, x, b), R(x, a, y)\}) = \{a, b, x, y\}$.

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 and S' be sets of relational atoms over the same schema. A* homomorphism from S to S' *is a function $h : Dom(S) \to Dom(S')$ such that the following hold:*

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

If, in addition, $h(\bar{u}) = \bar{v}$, where $\bar{u}$ and $\bar{v}$ are two tuples of the same length over $Dom(S)$ and $Dom(S')$, 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 $(S', \bar{v})$.*

Examples

We proceed to give four examples, assuming that both sets S and S' contain relational atoms over a single binary relation 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 that occur in the 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 $S' = \{R(z, z)\}$. Then 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 is a constant value. Then, as opposed to the previous example, there is no homomorphism h from S to S', as $h(a)$ 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 a third element z to $\mathrm{Dom}(S')$, and let S' be $\{R(x, y), R(y, x), R(x, z), R(z, x), R(y, z), R(z, y)\}$. 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 some of the following chapters, it is technically convenient to have a machinery in place that allows us to convert a set of atoms S into a possibly infinite database by converting the variables occurring in S into new constants not already in S.[1] This process is called *grounding* and can be easily achieved by exploiting the notion of homomorphism.

Definition 9.2 (Grounding). *Let S be a set of relational atoms over a schema $\boldsymbol{S}$. A possibly infinite database D over $\boldsymbol{S}$ is called a* grounding *of S if there exists a homomorphism from S to D that is a bijection.*

It is not difficult to verify that 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 we write G_S for *the* unique bijective homomorphism from S to $S^{\downarrow}$.

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

[1] Converting a database into a set of atoms by replacing constants with variables is needed less often; this is discussed in Chapter 13.

10

Functional Dependencies

In a relational databse system, it is possible to specify semantic properties that should be satisfied by all database instances 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 relation

```
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

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

$$R \ : \ U \ \rightarrow \ V$$

where $R \in \mathbf{S}$ and $U, V \subseteq \{1, \ldots, \mathrm{ar}(R)\}$. A key dependency is a special case of an FD, which states that some attributes functionally determine *all* the other attributes of the relation in question. Formally, the FD σ above is a *key dependency* if $V = \{1, \ldots, \mathrm{ar}(R)\}$. In this case, for notational convenience, we may also write $\mathit{key}(R) = U$.

A possibly infinite database D over $\mathbf{S}$ *satisfies* $\sigma = R : U \rightarrow V$, denoted $D \models \sigma$, if for each pair of tuples $\bar{a}, \bar{b} \in R^D$, it holds that

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

Note that, by abuse of notation, we write U and V in the projection expressions $\pi_U(\cdot)$ and $\pi_V(\cdot)$ above for the lists consisting of the elements of U and V, respectively, in ascending order. We say that D *satisfies* a set Σ of FDs, written $D \models \Sigma$, if $D \models \sigma$ for each $\sigma \in \Sigma$. The notion of satisfaction for FDs can be transferred to sets of atoms by exploiting the notion of grounding of sets of atoms. In particular, a 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 over a schema $\mathbf{S}$ and a set Σ of FDs over $\mathbf{S}$
OUTPUT:	yes if $D \models \Sigma$, and no otherwise

It is not difficult to show the following result:

Theorem 10.1. FD-Satisfaction *is in* PTIME.

Proof. Consider a database D over a schema $\mathbf{S}$ and a set Σ of FDs over $\mathbf{S}$. Let σ be an FD from Σ of the form $R : U \rightarrow V$. To check whether $D \models \sigma$ we need to check that, for every $\bar{a}, \bar{b} \in R^D$, if $\pi_U(\bar{a}) = \pi_U(\bar{b})$, then $\pi_V(\bar{a}) = \pi_V(\bar{b})$. 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 seen as a family of algorithms since, depending on the class of dependencies in question, we may get a different variant. On the other hand, all the chase variants have the same objective: given a set of relational atoms S and a set Σ of dependencies, to transform S as dictated by the dependencies of Σ into a set of relational atoms that satisfies Σ. We proceed to introduce the chase for FDs.

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

$$h_{\bar{u},\bar{v}}(w) \;=\; \begin{cases} u_i & \text{if } w = v_i \text{ and } u_i < v_i, \text{ for some } i \in [1,k], \\[2ex] v_i & \text{if } w = u_i \text{ and } v_i < u_i, \text{ for some } i \in [1,k], \\[2ex] w & \text{otherwise.} \end{cases}$$

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

$$S' \;=\; \begin{cases} \perp & \text{if there is an } i \in [1,k] \text{ with } u_i \neq v_i \text{ and } u_i, v_i \in \mathsf{Const}, \\[2ex] h_{\bar{u},\bar{v}}(S) & \text{otherwise.} \end{cases}$$

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 each $i \in [1,k]$. 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 now introduce the notion of chase sequence of S under a finite set Σ of FDs. We distinguish the two cases of finite and infinite chase sequences:

- A *finite chase sequence* of S under Σ is a finite sequence $s = S_0, \ldots, S_n$ where S_i is a set of atoms for each $i \in [1, n-1]$, the first element S_0 is S itself, and S_n is either the symbol $\perp$ or a set of atoms, such that:
  - for each $i \in [0, n-1]$, there exist $\sigma = R : U \rightarrow V$ in Σ and $R(\bar{u}), R(\bar{v}) \in S_i$ with $S_i \xrightarrow{\sigma,(\bar{u},\bar{v})} S_{i+1}$, which simply means that S_{i+1} is obtained by applying σ to S_i with $(\bar{u}, \bar{v})$, and
  - either

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

  · $S_n = \bot$, in which case we say that s is *failing*, or,
  · for every FD $\sigma = R : U \to V$ in Σ and $R(\bar{u}), R(\bar{v}) \in S_n$, 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 atoms, where $S_0 = S$, such that, for each $i \geq 0$, there exist $\sigma = R : U \to V$ in Σ and $R(\bar{u}), R(\bar{v}) \in S_i$ with $S_i \xrightarrow{\sigma,(\bar{u},\bar{v})} S_{i+1}$.

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

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 10.3. *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 $S_n = S'_m$.*

This 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 10.2, 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 10.4. *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

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

[2] Formal proofs are omitted since in Chapter 43 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.

Recall that $S_i = h_{\bar{u}_{i-1},\bar{v}_{i-1}}(S_{i-1})$, for each $i \in [1,n]$. 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 10.3, different finite chase sequences have the same result, we get the following.

Lemma 10.5. *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 $\text{Chase}(S,\Sigma) \neq \perp$, we can refer to *the* chase homomorphism of S under Σ, denoted $h_{S,\Sigma}$. It should be clear that $\text{Chase}(S,\Sigma) \neq \perp$ implies $h_{S,\Sigma}(S) = \text{Chase}(S,\Sigma)$.

By Lemma 10.2, $\text{Chase}(S,\Sigma)$ can be computed after finitely many steps. Furthermore, assuming that $\text{Chase}(S,\Sigma) \neq \perp$, also the chase homomorphism $h_{S,\Sigma}$ can be computed after finitely many steps. In fact, as the next result shows, this is even possible after polynomially many steps.

Lemma 10.6. $\text{Chase}(S,\Sigma)$ *can be computed in polynomial time. Furthermore, if* $\text{Chase}(S,\Sigma) \neq \perp$*, then $h_{S,\Sigma}$ can be computed in polynomial time.*

The last main property of the chase states that, if $\text{Chase}(S,\Sigma) \neq \perp$, 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 10.7. *Let S' be a set of atoms over $\mathbf{S}$ such that $(S,\bar{u}) \to (S',\bar{v})$ and $S' \models \Sigma$. If* $\text{Chase}(S,\Sigma) \neq \perp$*, 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 16 for studying a different problem for which the proper treatment of constants is crucial.

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 over $\mathbf{S}$, 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:	yes if $\Sigma \models \sigma$, and no otherwise

We proceed to show the following result.

Theorem 10.8. FD-Implication *is in* PTIME.

To show Theorem 10.8, 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 $\{R(x_1, \dots, x_{\mathrm{ar}(R)}), R(y_1, \dots, y_{\mathrm{ar}(R)})\}$, where

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

We can now show the following auxiliary result.

Lemma 10.9. *Consider a set Σ of FDs over as schema $\boldsymbol{S}$, and an FD σ over $\boldsymbol{S}$. It holds that $\Sigma \models \sigma$ if and only if* $\mathrm{Chase}(S_\sigma, \Sigma) \models \sigma$.

Proof. ($\Rightarrow$) By hypothesis, for every finite set of relational atoms S, it holds that $S \models \Sigma$ implies $S \models \sigma$. Observe that $\mathrm{Chase}(S_\sigma, \Sigma) \neq \perp$ since S_σ contains only variables. Therefore, by Lemma 10.4, we have that $\mathrm{Chase}(S_\sigma, \Sigma) \models \Sigma$. Since, by Lemma 10.2, $\mathrm{Chase}(S_\sigma, \Sigma)$ is finite, we get that $\mathrm{Chase}(S_\sigma, \Sigma) \models \sigma$.

($\Leftarrow$) Consider now a database D over $\mathbf{S}$ such that $D \models \Sigma$, and with σ being of the form $R : \{i_1, \dots, i_k\} \to \{j_1, \dots, j_\ell\}$, assume that there are tuples $(a_1, \dots, a_{\mathrm{ar}(R)}), (b_1, \dots, b_{\mathrm{ar}(R)}) \in R^D$ such that $(a_{i_1}, \dots, a_{i_k}) = (b_{i_1}, \dots, b_{i_k})$. Recall also that S_σ is of the form $\{R(x_1, \dots, x_{\mathrm{ar}(R)}), R(y_1, \dots, y_{\mathrm{ar}(R)})\}$. Let $\bar{z} = (x_{j_1}, \dots, x_{j_\ell}, y_{j_1}, \dots, y_{j_\ell})$ and $\bar{c} = (a_{j_1}, \dots, a_{j_\ell}, b_{j_1}, \dots, b_{j_\ell})$. It is clear that $(S_\sigma, \bar{z}) \to (D, \bar{c})$. Since $D \models \Sigma$ and $\mathrm{Chase}(S_\sigma, \Sigma) \neq \perp$, by Lemma 10.7

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

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

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

which in turn implies that

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

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

By Lemma 10.9, we get a simple procedure for checking whether a set Σ of FDs implies an FD σ that runs in polynomial time:

$$\text{if } \mathrm{Chase}(S_\sigma, \Sigma) \models \sigma, \text{then return yes; otherwise, return no.}$$

Indeed, the set of atoms $\mathrm{Chase}(S_\sigma, \Sigma)$ can be constructed in polynomial time (by Lemma 10.6), and $\mathrm{Chase}(S_\sigma, \Sigma) \models \sigma$ can be checked in polynomial time (by Theorem 10.1), and Theorem 10.8 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 relations

```
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

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

A possibly infinite database D over $\mathbf{S}$ *satisfies* σ, denoted $D \models \sigma$, if, for every tuple $\bar{a} \in R^D$, there exists a tuple $\bar{b} \in P^D$ such that

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

For a set Σ of INDs, we say that D *satisfies* Σ, denoted $D \models \Sigma$, if $D \models \sigma$ for each $\sigma \in \Sigma$. The notion of satisfaction for INDs can be transferred to sets of atoms via the grounding of sets of atoms. 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 $S^{\downarrow} \models \Sigma$.

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:	yes if $D \models \Sigma$, and no otherwise

It is not difficult to show the following result:

Theorem 11.1. IND-Satisfaction *is in* PTIME.

Proof. Consider a database D over 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 if $D \models \sigma$ we need to check that, for every tuple $(a_1, \ldots, a_{\mathrm{ar}(R)}) \in R^D$, there exists a tuple $(b_1, \ldots, b_{\mathrm{ar}(S)}) \in P^D$ such that $(a_{i_1}, \ldots, a_{i_k}) = (b_{j_1}, \ldots, b_{j_k})$. 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 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 $\ell \in [1, \mathrm{ar}(P)]$,

$$v_\ell \;=\; \begin{cases} u_{i_k} & \text{if } \ell = j_k, \text{ for } k \in [1, m], \\ x_\ell^{\sigma, \pi_{(i_1, \ldots, i_m)}(\bar{u})} & \text{otherwise,} \end{cases}$$

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 now introduce the notion of chase sequence of S under a finite set Σ of INDs. We distinguish the two cases of finite and infinite chase sequences:

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

- A *finite chase sequence* of S under Σ is a finite sequence $s = S_0, \dots, S_n$ of sets of atoms, with $S_0 = S$, such that:

  1. for each $i \in [1, n-1]$, there exists $\sigma = R[\alpha] \subseteq P[\beta] \in \Sigma$ and $\bar{u} \in R^{S_i}$ with $\mathsf{new}(\sigma, \bar{u}) \notin S_i$ and $S_i \xrightarrow{\sigma, \bar{u}} S_{i+1}$, and
  2. for every IND $\sigma = R[\alpha] \subseteq P[\beta] \in \Sigma$ and $\bar{u} \in R^{S_n}$, $\mathsf{new}(\sigma, \bar{u}) \in S_n$.

  The first condition simply says that S_{i+1} is obtained from S_i by applying σ to S_i with $\bar{u}$, while σ has not been already applied to some S_j, for $j < i$, with $\bar{u}$. The second condition states that no new atom, which is not already in S_n, can be derived by applying an IND of Σ to S_n. 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 $s = S_0, S_1, \dots$ of sets of atoms, with $S_0 = S$, such that:

  1. for each $i \geq 0$, there exists $\sigma = R[\alpha] \subseteq P[\beta] \in \Sigma$ and $\bar{u} \in R^{S_i}$ with $\mathsf{new}(\sigma, \bar{u}) \notin S_i$ and $S_i \xrightarrow{\sigma, \bar{u}} S_{i+1}$, and
  2. for each $i \geq 0$, and for each $\sigma = R[\alpha] \subseteq P[\beta] \in \Sigma$ and $\bar{u} \in R^{S_i}$ such that σ is applicable to S_i with $\bar{u}$, there is a $j > i$ with $\mathsf{new}(\sigma, \bar{u}) \in S_j$.

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

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 identifies 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 may happen even for simple sets of atoms and INDs. For example, it is not hard to verify that the chase sequence of $\{R(a, b)\}$ under $\{R[2] \subseteq R[1]\}$ is infinite.[3]

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

[2] Formal proofs are omitted since in Chapter 34 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] In this simple case we have only one chase sequence.

the same result. At this point, let us clarify that the result of an infinite chase sequence $s = S_0, S_1, \ldots$ of S under Σ always exists.[4] 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 $(\text{Inst}(\mathbf{S}), \subseteq)$, which we know that always exists by Knaster-Tarski's Theorem (see Exercise 11.13). We can now state the announced result.

Lemma 11.2. *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 11.2 allows us to refer to *the* unique result of the chase of S under Σ, denoted $\text{Chase}(S, \Sigma)$, which is defined as the result of some (any) finite or infinite chase sequence of S under Σ. It is not difficult to show that $\text{Chase}(S, \Sigma)$ satisfies Σ. Let us stress, though, that in the case where only infinite chase sequences exist, this result heavily relies on the fairness condition.

Lemma 11.3. *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 11.4. *Let S' be a set of atoms over $\mathbf{S}$ such that $(S, \bar{u}) \to (S', \bar{v})$ and $S' \models \Sigma$. It holds that* $(\text{Chase}(S, \Sigma), \bar{u}) \to (S', \bar{v})$.

Implication of Inclusion Dependencies

We now proceed to study the implication problem for INDs. The notion of implication for INDs is defined in the same way as for functional dependencies. More precisely, given a set Σ of INDs over a schema $\mathbf{S}$ and a single IND σ over $\mathbf{S}$, we say that Σ *implies* σ, denoted $\Sigma \models \sigma$, if, for every database D over $\mathbf{S}$, 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:	yes if $\Sigma \models \sigma$ and no otherwise

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

Although for FDs the implication problem is tractable (Theorem 10.8), for INDs it turns out to be an intractable problem:

Theorem 11.5. IND-Implication *is* PSPACE-*complete.*

We first concentrate on the upper bound. We are going to establish a result, analogous to Lemma 10.9 for FDs, that characterizes implication of INDs via the chase. However, since the chase for INDs might 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 σ without restriction*, denoted $\Sigma \models_\infty \sigma$, if, for every possibly infinite database D over $\mathbf{S}$, we have that $D \models \Sigma$ implies $D \models \sigma$.

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

Lemma 11.6. *Consider a set Σ of INDs over schema $\boldsymbol{S}$, and an IND σ over $\boldsymbol{S}$. It holds that $\Sigma \models_\infty \sigma$ if and only if* $\mathrm{Chase}(S_\sigma, \Sigma) \models \sigma$.

Proof. ($\Rightarrow$) By hypothesis, for every possibly infinite set of relational atoms S, it holds that $S \models \Sigma$ implies $S \models \sigma$. By Lemma 11.3, $\mathrm{Chase}(S_\sigma, \Sigma) \models \Sigma$, and therefore, $\mathrm{Chase}(S_\sigma, \Sigma) \models \sigma$.

($\Leftarrow$) Consider now a possibly infinite database D over $\mathbf{S}$ such that $D \models \Sigma$, and with σ being of the form $R[i_1, \dots, i_k] \subseteq P[j_1, \dots, j_k]$, assume that there exists a tuple $(a_1, \dots, a_{\mathrm{ar}(R)}) \in R^D$. Recall also that S_σ is of the form $\{R(x_1, \dots, x_{\mathrm{ar}(R)})\}$. Let $\bar{y} = (x_{i_1}, \dots, x_{i_k})$ and $\bar{b} = (a_{i_1}, \dots, a_{i_k})$. It is clear that $(S_\sigma, \bar{y}) \to (D, \bar{b})$. Since $D \models \Sigma$, by Lemma 11.4

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

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

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

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

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

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

Lemma 11.6 alone is of little use since it characterizes implication of INDs under possibly infinite databases, whereas in practice we are interested only in databases that are finite. 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 11.7. *Consider a set Σ of INDs over as schema $\boldsymbol{S}$, and an IND σ over $\boldsymbol{S}$. It holds that $\Sigma \models \sigma$ if and only if $\Sigma \models_\infty \sigma$.*

Although the above theorem is extremely useful for our analysis, we do not discuss its proof since it is out of the scope of this book. An immediate consequence of Lemma 11.6 and Theorem 11.7 is the following:

Corollary 11.8. *Consider a set Σ of INDs over a schema $\boldsymbol{S}$, and an IND σ over $\boldsymbol{S}$. It holds that $\Sigma \models \sigma$ if and only if* $\text{Chase}(S_\sigma, \Sigma) \models \sigma$.

Having Corollary 11.8 in place, one may think that the procedure for checking whether $\Sigma \models \sigma$, which will lead to the PSpace upper bound claimed in Theorem 11.5, is simply to construct the set of atoms $\text{Chase}(S_\sigma, \Sigma)$, and then check whether it satisfies σ, which can be done due to Theorem 11.1. However, it should not be forgotten that $\text{Chase}(S_\sigma, \Sigma)$ is, in general, infinite. Therefore, we need to rely on a more refined procedure that avoids the explicit construction of $\text{Chase}(S_\sigma, \Sigma)$. We proceed to present a technical lemma that is the building block underlying this refined procedure.

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 each $\ell \in [1, \text{ar}(P)]$,

$$y_\ell \;=\; \begin{cases} x_{i_k} & \text{if } \ell = j_k, \text{ for } k \in [1, m], \\ \\ \star & \text{otherwise.} \end{cases}$$

We can then show the following technical result; see Exercise 11.14.

Lemma 11.9. *Consider a set Σ of INDs over as schema $\boldsymbol{S}$, and an IND $\sigma = R[i_1, \ldots, i_k] \subseteq P[j_1, \ldots, j_k]$ over $\boldsymbol{S}$. The following are equivalent:*

1. $\text{Chase}(S_\sigma, \Sigma) \models \sigma$.
2. *There is a sequence of atoms $R_1(\bar{x}_1), \ldots, R_n(\bar{x}_n)$, for $n \geq 1$, such that:*
   a) $S_\sigma = \{R_1(\bar{x}_1)\}$,
   b) *for each $i \in [2, n]$, there is an IND $\sigma_i = R_{i-1}[\alpha_{i-1}] \subseteq R_i[\alpha_i]$ in Σ that is applicable to $\{R_{i-1}(\bar{x}_{i-1})\}$ with $\bar{x}_{i-1}$ such that $R_i(\bar{x}_i) = \mathsf{new}^\star(\sigma_i, \bar{x}_{i-1})$,*
   c) *$R_n = P$, and*
   d) $\pi_{(i_1,\ldots,i_k)}(\bar{x}_1) = \pi_{(j_1,\ldots,j_k)}(\bar{x}_n)$.

Lemma 11.9 leads to a simple nondeterministic procedure for checking whether a set Σ of INDs over a schema $\mathbf{S}$ implies a single IND σ of the form $R[i_1, \ldots, i_k] \subseteq P[j_1, \ldots, j_k]$ over $\mathbf{S}$ (see Algorithm 2). The algorithm first checks whether $R[i_1, \ldots, i_k]$ and $P[j_1, \ldots, j_k]$ are the same, in which case

Algorithm 2 IND-IMPLICATION(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: If $\Sigma \models \sigma$, then yes; otherwise, no.

```
 1: if $R = P$ and $(i_1, \ldots, i_k) = (j_1, \ldots, j_k)$ then
 2:     return yes
 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:         $T'(\bar{z}) := \mathsf{new}^\star(\sigma', \bar{y})$
 8:         $S_\triangleright := \{T'(\bar{z})\}$
 9:     if $S_\triangleright = \emptyset$ then
10:         return no
11:     $S_\triangledown := S_\triangleright$
12:     $S_\triangleright := \emptyset$
13:     $Check := b$, where $b \in \{0, 1\}$
14: until $Check = 1$
15: if $T' = P$ and $\pi_{(i_1,\ldots,i_k)}(\bar{x}) = \pi_{(j_1,\ldots,j_k)}(\bar{z})$ then
16:     return yes
17: else
18:     return no
```

$\Sigma \models \sigma$ holds trivially, and returns yes. Otherwise, it proceeds to construct the sequence of atoms $R_1(\bar{x}_1), \ldots, R_n(\bar{x}_n)$ as in Lemma 11.9 (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. In particular, 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 nondeterministically chooses to exit the loop by setting *Check* to 1, and check whether $S_\triangleright$ consists of an atom of the form $T'(\bar{z})$ 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 yes; otherwise, it returns no.

It is easy to verify that Algorithm 2 uses polynomial space. This heavily relies on the fact that the atoms generated during its computation contain only variables from $\{x_1, \ldots, x_{\mathrm{ar}(R)}\}$ and the special variable $\star$, which in turn implies that $S_\triangledown$ and $S_\triangleright$ can be stored in polynomial space. It also takes polynomial space to check if $R[i_1, \ldots, i_k] = P[j_1, \ldots, j_k]$ (see line 1), to check if an IND is applicable to $S_\triangledown$ with $\bar{y}$ (see line 6), and to check if $T' = P$ and $\pi_{(i_1,\ldots,i_k)}(\bar{x}) = \pi_{(j_1,\ldots,j_k)}(\bar{z})$ (see line 15). Therefore, IND-Implication is in NPSPACE, and thus in PSPACE since, by Savitch's theorem, NPSPACE = PSPACE.

A PSPACE lower bound for IND-Implication can be shown via a reduction from the following PSPACE-hard problem: given a linear space bounded Turing machine M and a word x, decide whether M accepts x; see Exercise 11.15.

Comments and Exercises for Part I

Many SQL textbooks, an overview of the Standard in [5]. A formal semantics and detailed translation of core SQL into RA in [13].

Exercise 11.1. Let q be an FO query. Prove that one can compute in polynomial time an equivalent FO query q' that uses only $\neg$, $\vee$, and $\exists$.

Exercise 11.2. Let $\varphi(\bar{x})$ be such that φ is an FO formula over schema $\mathbf{S}$ and $\bar{x}$ a k-ary tuple that mentions precisely the free variables in φ. In Chapter 3, we defined the *output* of φ on D as $\varphi(D) = \{\bar{a} \mid D \models \varphi(\bar{a})\}$. Prove that $\varphi(\bar{x})$ is a k-ary query over $\mathbf{S}$, i.e., that $\varphi(\bar{x})$ is C-generic.

Exercise 11.3. In Chapter 4, we formally defined the semantics of rename and join in the named relational algebra. Provide formal definitions of the other operators.

Exercise 11.4. State and prove the converse of Theorem 4.3.

Exercise 11.5. Prove that adding conditions $\bar{a} \in e$ and $\text{empty}(e)$ to selection conditions of RA does not increase its expressiveness, by showing how selections with these conditions can be translated in the standard operations of RA.

Exercise 11.6. Prove that adding nested subqueries in the `FROM` clause clause does not icnrease expressiveness. That is, extend the translation from basic SQL to RA that handles nested subqueries in `FROM`.

Exercise 11.7. Let θ be the formula constructed in the proof of Theorem 7.1. Prove that θ is true if and only if $D \models \varphi$.

Exercise 11.8. Prove that the complexity of evaluation for FO_k is in polynomial time.

Exercise 11.9. Let q_M be the Boolean FO query constructed in the proof of Theorem 8.1. Prove that if the Turing machine M on the empty word does not halt, then there exists an infinite database D such that $q(D) = \texttt{true}$.

Exercise 11.10. The existence of such an infinite database (see the above exercise) implies that the proof of Theorem 8.1 does not show that FO-Satisfiability is undecidable if we consider arbitrary (possibly infinite) databases. Show that FO-Satisfiability is undecidable even if we consider arbitrary databases by adapting the proof of Theorem 8.1.

Exercise 11.11. Prove that FO-Containment is undecidable even if the left hand-side query is a Boolean query of the form $\exists \bar{x}\, \varphi(\bar{x})$, where φ is a conjunction of relational atoms or the negation of relational atoms.

Exercise 11.12.

Wim:
Just a suggestion. An implementer would at least sort tuples according to U

The algorithms in Theorems 10.1 and 11.1 were chosen with simplicity in mind instead of efficiency. How would you implement the algorithm if time efficiency is important?

Exercise 11.13. Prove that the result of an infinite chase sequence of a finite set of relational atoms under a set of INDs always exists.

Exercise 11.14. Prove Lemma 11.9.

Andreas:
Further details will be given concerning the proof strategy.

Exercise 11.15. Prove a PSPACE lower bound for IND-Implication (Theorem 11.5). Use a reduction from the following PSPACE-hard problem: given a linear space bounded Turing machine M and a word x, decide whether M accepts x.

Wim:
@Andreas: This is just my quick formulation of the exercise. Improve if needed. We also agreed that, if we refer to an exercise using a number in the chapter ("see Exercise 1.13") then the exercise has a hint on how to solve it.

Part II

Conjunctive Queries

Conjunctive queries (CQs) are of special importance in databases. They express relational joins, the operation that is most commonly performed by relational database engines. This is due to the fact that in the process of designing a database schema, information is spread over multiple relations; to obtain answers to queries one thus needs to join such relations. More specifically, CQs have the power of select-project-join queries and, thus, correspond to a very common type of queries written in SQL. The special shape of CQs makes static analysis easier for them. In fact, there is a simple algorithm to check containment, which leads to the widespread use of CQs in applications where good properties of both query evaluation and reasoning about queries are important (such as answering queries using views and ontology-based data access). In this part, we study some fundamental problems associated with CQs, namely evaluation of these queries, and static analysis tasks such as containment and minimization.

12

Syntax and Semantics

Syntax of CQs

Conjunctive queries (CQs) are defined as FO queries of the form:

$$\varphi(\bar{x}) \;=\; \exists \bar{y} \left(R_1(\bar{u}_1) \wedge \cdots \wedge R_n(\bar{u}_n) \right), \tag{12.1}$$

where each $R_i(\bar{u}_i)$, for $i \in [1, n]$, is an atomic formula. That is, R_i is a k-ary relation symbol in the schema, and $\bar{u}_i$ is a k-ary tuple of constants and variables, where elements can be repeated (i.e., we can write $R(x, x, y)$ and $S(a, x, x, a)$). The tuple $\bar{x}$ consists only of variables, and it is composed of the output variables of the formula, which are those variables in the $\bar{u}_i$s that do not occur in the tuple $\bar{y}$. Again such variables can repeat, i.e., we can write $\varphi(z, x, z) = \exists y \; R(x, y, y, a) \wedge S(b, z, x)$.

It is very common to write CQs as *rules*. The CQ $\varphi(\bar{x})$ of the form (12.1), written as a rule, appears as follows:

$$q(\bar{x}) \;\text{:-}\; R_1(\bar{u}_1), \ldots, R_n(\bar{u}_n)\,. \tag{12.2}$$

Here q is a fresh predicate symbol of the same arity than the tuple $\bar{x}$ in $\varphi(\bar{x})$, and it defines the output of the query (hence notation q). It is referred to as the *head* of the rule. What appears on the right of the :– symbol – i.e., $R_1(\bar{u}_1), \ldots, R_n(\bar{u}_n)$ – is referred to as *body* of the rule.

Example 12.1. Consider again the relational schema from Chapter 3:

```
Person [ person_id, name, city_of_birth ]
Profession [ person_id, profession_name ]
City [ city_id, name, country ]
```

Then 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:

$$\begin{aligned} \varphi(y) \;=\; & \exists x \exists z \big(\text{Person}(x, y, z) \wedge \\ & \quad \text{Profession}(x, \text{‘computer scientist’}) \wedge \text{City}(z, \text{‘Athens’}, \text{‘Greece’}) \big). \end{aligned}$$

In the rule-based notation, this query is expressed as follows:

$$\begin{aligned} q(y) \coloneq\ & \mathrm{Person}(x, y, z), \\ & \mathrm{Profession}(x, \text{'computer scientist'}), \mathrm{City}(z, \text{'Athens'}, \text{'Greece'}). \quad \square \end{aligned}$$

As shown in the example, to go from the formula (12.1) to the rule-based notation, we simply omit the existential quantifiers $\exists \bar{y}$ and replace conjunctions $\wedge$ with commas. For the opposite direction, to go from the rule (12.2) to the FO presentation (12.1), we replace commas in the body with conjunctions, and existentially quantify all variables that appear in the body but not in the head.

We often write simply $q(\bar{x})$ for a CQ of the form (12.2). We also write q if the output variables are not important, or are understood from the context. A CQ q is *Boolean* if it has no output variables, i.e., $\bar{x}$ is the empty tuple. In that case, we simply write q as the head, instead of $q()$. As an example, the following Boolean CQ checks whether there exists any computer scientist that was born in the city of Putú in Chile:

$$q \coloneq \mathrm{Person}(x, y, z), \mathrm{Profession}(x, \text{'computer scientist'}), \mathrm{City}(z, \text{'Putú'}, \text{'Chile'}).$$

While we generally stick to the rule-based notation throughout the book, for convenience we freely interpret a CQ as a rule of the form (12.2) or as a FO formula of the form (12.1).

Semantics of CQs

Consider a CQ, presented either as a rule $q(\bar{x}) \coloneq R_1(\bar{u}_1), \ldots, R_n(\bar{u}_n)$, or as a formula $\varphi(\bar{x}) = \exists \bar{y}\, (R_1(\bar{u}_1) \wedge \cdots \wedge R_n(\bar{u}_n))$. Let k be the arity of q, that is, the length of the tuple $\bar{x}$. The simplest way to define the *output* $q(D)$ of evaluating q over a database D, is to view q as an FO formula and appeal to the definition of satisfaction of FO formulae; then

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

However, there is a more intuitive way of defining the semantics of CQs when they are viewed as rules. Let X be the set of all variables that occur in CQ q of the form (12.2). Recall than an assignment of variables, over a database D, is a mapping $\eta : X \to \mathrm{Dom}(D)$. An assignment is *consistent* with D if

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

where $\eta(\bar{u}_i)$ $(1 \leq i \leq n)$ is a tuple obtained from $\bar{u}_i$ by replacing each variable x occurring in it by $\eta(x)$, and by leaving the constants unchanged. Hence, the previous inclusion tell us that applying η to each tuple of variables $\bar{u}_i$ produces a tuple $\eta(\bar{u}_i)$ that is in the relation R_i of D, for all atoms $R_1(\bar{u}_1), \ldots, R_n(\bar{u}_n)$ mentioned in the query. Then

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

We can think of a CQ given as a rule as a *pattern*. If an assignment applied to this pattern produces only facts that occur in D, it means that the pattern itself occurs in D. If the pattern occurs in D under an assignment η, then we output the tuple $\eta(\bar{x})$.

Example 12.2. Consider the following CQ to retrieve the identifiers and names of computer scientists:

$$q(x, y) \coloneq \text{Person}(x, y, z), \text{Profession}(x, \text{`computer scientist'}).$$

and a database D with facts

$$\begin{aligned}\{\ &\text{Person}(\text{`p1'}, \text{`John Smith'}, \text{`c1'}), \text{Profession}(\text{`p1'}, \text{`computer scientist'})\\ &\text{Person}(\text{`p2'}, \text{`Paul Brown'}, \text{`c1'}), \text{Profession}(\text{`p2'}, \text{`lawyer'}),\\ &\text{City}(\text{`c1'}, \text{`Chicago'}, \text{`USA'})\ \}.\end{aligned}$$

Then the assignment $\eta = \{x \mapsto \text{`p1'}, y \mapsto \text{`John Smith'}, z \mapsto \text{`c1'}\}$ is consistent with $D$; when applied to the body of $q$, it generates Person(‘p1’, ‘John Smith’, ‘c1’) and Profession(‘p1’, ‘computer scientist’), both of which are facts of $D$. On the other hand, the assignment $\eta' = \{x \mapsto \text{`p2'}, y \mapsto \text{`Paul Brown'}, z \mapsto \text{`c1'}\}$ is not consistent with D; when applied to the body of q, it generates the fact Profession(‘p2’, ‘computer scientist’) that is not in D. It is straightforward to verify that η is the only assignment that is consistent with D, so that the output of q over D consists of the tuple $\eta\big((x, y)\big)$, that is, $q(D) = \{(\text{`p1'}, \text{`John Smith'})\}$. □

If q is a Boolean query, then $q(D) = \texttt{true}$ if and only if there is an assignment consistent with D. In other words, $q(D)$ is `true` if and only if the pattern in the body of the CQ occurs in the database at least once. For instance, if in Example 12.2, we change the head to $q_1()$, then $q_1(D)$ evaluates to `true`, since assignments η is consistent with D. On the other hand, query

$$q_2 \coloneq \text{Person}(x, y, z), \text{Profession}(x, \text{`nurse'})$$

evaluates to `false` over D, since there is no assignment η such that Person($\eta(x)$, $\eta(y), \eta(z)$) and Profession($\eta(x)$, ‘nurse’) are both facts of D.

CQs as a Fragment of FO

To write CQs in FO, we only needed relational atoms $R_i(\bar{u}_i)$, conjunction $\wedge$, and existential quantification $\exists$. Thus, every CQ can be expressed in the fragment of FO that is the closure of relational atoms under $\exists$ and $\wedge$.

In fact, the converse is true too. Take an arbitrary query $\varphi(\bar{x})$ in this fragment of FO, i.e., the closure of relational atoms under $\exists, \wedge$. Then $\varphi(\bar{x})$ is equivalent to a CQ. This can be seen by first renaming variables so they do not repeat (which can always be done for FO formulae) and then by pushing existential quantifiers outside. For example, to express $\varphi(x) = (\exists y\, R(x,a,y)) \wedge (\exists y\, S(y,x,b))$, we first ensure that bound variables do not repeat and express $\varphi(x)$ as $(\exists y\, R(x,a,y)) \wedge (\exists z\, S(z,x,b))$. After that we push all the quantifiers outside: $\varphi(x) = \exists y \exists z\, \big(R(x,a,y) \wedge S(z,x,b)\big)$. The result of course is a CQ.

Note that the fragment that we described is *not* the same as the $\exists, \wedge$-fragment of FO. Fragments given by listing a set of features of FO are assumed to be the closure of all atomic formulae under those features. Thus, the $\exists, \wedge$-fragment would also allow equality atoms, and thus formulae like $\varphi(x,y) = (x = y)$. However, this formula is not even equivalent to a CQ, which will be proved in the next chapter.

CQs as a Fragment of Relational Algebra

CQs have the same expressive power than the fragment of RA that contains the following operations: selection, where conditions in selections are conjunctions of equalities; projection; and Cartesian product. This is usually referred to as the SPJ fragment of RA, for *select-project-join.* Indeed, recall that the join is a selection from the Cartesian product on a condition which is a conjunction of equalities.

We now show how to translate CQs into SPJ queries. As before, start with a CQ $q(\bar{x}) \text{ :- } R_1(\bar{u}_1), \ldots, R_n(\bar{u}_n)$ as in (12.2). Then we can express $q(\bar{x})$ as

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

where conditions in selections and the list of positions in the projection operator are defined by the rules below.

1. For a tuple $\bar{u}$ of constants and variables, the condition $\theta(\bar{u})$ is the conjunction of equalities $j \doteq a$, where the jth component of $\bar{u}$ is the constant a, and equalities $j \doteq k$, where the jth and the kth component of $\bar{u}$ are the same variable. If no constant occurs in $\bar{u}$ and all the components of $\bar{u}$ are distinct, then the condition is true and the selection operator can be omitted.
2. The condition $\theta(q)$ is the conjunction of all equalities among variables imposed by q. That is, if in the sequence $\bar{u}_1\bar{u}_2\ldots\bar{u}_n$ of constants and variables, two positions j and k from different tuples are occupied by the same variable, then $j \doteq k$ is added to $\theta(q)$.
3. 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 $\bar{x}$.

Example 12.3. Consider the CQ

$$q(x,x,y) \text{ :- } R_1(x,z,z,a,x), R_2(a,y,z,a,b), R_3(x,y,z)\,. \tag{12.3}$$

Then for the tuple $\bar{u}_1 = (x, z, z, a, x)$, we have that $\theta(\bar{u}_1) = (4 \doteq a) \wedge (1 \doteq 5) \wedge (2 \doteq 3)$. For the tuple $\bar{u}_2 = (a, y, z, a, b)$, we have that $\theta(\bar{u}_2) = (1 \doteq a) \wedge (4 \doteq a) \wedge (5 \doteq b)$. Moreover, neither a constant nor a repetition of variables occurs in $u_3 = (x, y, z)$, so $\theta(\bar{u}_3)$ is omitted as this condition is true.

To specify the condition $\theta(q)$, we list all constants and variables of relational atoms, i.e., $\bar{u}_1\bar{u}_2\bar{u}_3$, which is $(x, z, z, a, x, a, y, z, a, b, x, y, z)$. The condition has to equate x from $\bar{u}_1$ with x in $\bar{u}_3$, as well z from $\bar{u}_1$ with z in both $\bar{u}_2$ and $\bar{u}_3$, and y from $\bar{u}_2$ with y in $\bar{u}_3$. This results in

$$\begin{aligned}\theta(q) \;=\; & (1 \doteq 11) \wedge (5 \doteq 11) \wedge (2 \doteq 8) \wedge (2 \doteq 13) \wedge \\ & \qquad (3 \doteq 8) \wedge (3 \doteq 13) \wedge (8 \doteq 13) \wedge (7 \doteq 12).\end{aligned}$$

Finally, α corresponds to variable x repeated twice and variable y, i.e., $\alpha = (1, 1, 7)$. Summing up, the query (12.3) is expressed as

$$\begin{aligned} q \;=\; & \pi_{(1,1,7)}\Big(\sigma_{(1\doteq 11)\wedge(2\doteq 8)\wedge(2\doteq 13)\wedge(7\doteq 12)}\big(\\ & \qquad \sigma_{(4\doteq a)\wedge(1\doteq 5)\wedge(2\doteq 3)}(R_1) \times \sigma_{(1\doteq a)\wedge(4\doteq a)\wedge(5\doteq b)}(R_2) \times R_3\big)\Big), \end{aligned} \tag{12.4}$$

where, for the sake of readability, we have eliminated condition $(5 \doteq 11)$ from $\theta(q)$ since it can be derived from conditions $(1 \doteq 11)$ 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 $\theta(q)$. □

Notice that when translating a CQ into RA, instead of using condition $\theta(q)$, one can replace Cartesian products by θ-joins (recall that the θ-join of relations R and S is defined as $R \bowtie_\theta S = \sigma_\theta(R \times S)$).

For the converse, we can show that every SPJ query can be defined by a CQ by induction on the structure of SPJ expressions. We can assume that in SPJ expressions all selections are of the form either $\sigma_{i\doteq a}$ 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 $\pi_{\bar{\imath}}$ that exclude the ith 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).

- An SPJ expression R, where R is a k-ary relation, is equivalent to the CQ $\varphi(x_1, \ldots, x_k) = R(x_1, \ldots, x_k)$, where $(x_1, \ldots, x_k)$ is a tuple of pairwise distinct and fresh variables.
- If e is an SPJ expression of arity k equivalent to a CQ $\varphi(x_1, \ldots, x_k)$, where the x_is are not necessarily distinct, then
  - $\sigma_{i\doteq a}(e)$ is equivalent to φ in which variable x_i is replaced by the constant a;
  - $\sigma_{i\doteq j}(e)$ is equivalent to φ in which variable x_j is replaced everywhere by variable x_i; and

  – $\pi_{\bar{i}}(e)$ is replaced by $\varphi(x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_k)$ if x_i occurs among the x_js with $j \neq i$, and by $\exists x_i\, \varphi(x_1, \ldots, x_k)$ otherwise.

- If e_1 is an SPJ expression of arity k equivalent to a CQ $\varphi_1(x_1, \ldots, x_k)$, and e_2 is an SPJ expression of arity m equivalent to a CQ $\varphi_2(y_1, \ldots, y_m)$, then $(e_1 \times e_2)$ is equivalent to CQ $\varphi(x_1, \ldots, x_k, y_1, \ldots, y_m) = \varphi_1(x_1, \ldots, x_k) \wedge \varphi_2(y_1, \ldots, y_m)$.

To explain the difference between the two cases of handling projection, consider 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 SPJ expression U has to be translated as a relational atom of the form $U(z)$ where the variable z is fresh, and likewise for SPJ 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 $\psi(x) = U(x) \wedge V(x)$. Instead, if we consider $e' = \pi_{\bar{2}}(U \times V)$, then the correct way to define it as a CQ is to existentially quantify over y in $\varphi(x, y)$ that defines $U \times V$; that is, the CQ defining e' is $\psi'(x) = \exists y\, (U(x) \wedge V(y))$.

The correctness of these translations are left as exercises to the reader. To sum up, we have the following equivalence result.

Theorem 12.4. *The classes of* CQ*s, of* SPJ *queries, and of FO queries in the fragment that is the* $\exists, \wedge$*-closure of relational atoms are equally expressive.*

13

Homomorphisms and Expressiveness

In this chapter we show that homomorphisms provide an alternative way to describe the evaluation of CQs, and 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. For a CQ presented as a rule

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

we define

$$S_q \;=\; \{R_1(\bar{u}_1), \dots, R_n(\bar{u}_n)\}. \tag{13.1}$$

Thus, we can talk about homomorphisms that map a CQ q to a database D. In particular, whenever we say that there is a homomorphism from q to D, we mean that there is a homomorphism from S_q to D, and we write $q \rightarrow D$ instead of $S_q \rightarrow D$. We also say that there is a homomorphism from $(q, \bar{x})$ to $(D, \bar{a})$ when there is a homomorphism from $(S_q, \bar{x}) \rightarrow (D, \bar{a})$, and write $(q, \bar{x}) \rightarrow (D, \bar{a})$ instead of $(S_q, \bar{x}) \rightarrow (D, \bar{a})$.

Let us now recall how we defined the evaluation of CQs in Chapter 12. Consider a database D, and a CQ q. To define the evaluation of q over D, we used the notion of assignment η of the variables of q to elements of $\mathrm{Dom}(D)$, asking that for every atom $R(\bar{u})$ in q, the atom $R(\eta(\bar{u}))$ is a fact in D. As $\eta(\bar{u})$ is defined by leaving constants unchanged, such an assignment corresponds to a function $h : \mathrm{Dom}(S_q) \rightarrow \mathrm{Dom}(D)$, which is the identity on the constants occurring in q, such that $R(h(\bar{u})) = R(\eta(\bar{u}))$ for every atom $R(\bar{u})$ in q. But, of course, this is the same as saying that h is a homomorphism from q to D. Therefore, the output of q on D is the set of all tuples $h(\bar{x})$ where h is a homomorphism from q to D, that is, the set of all tuples $\bar{a}$ over $\mathrm{Dom}(D)$ with $(q, \bar{x}) \rightarrow (D, \bar{a})$. This gives an alternative characterization of CQ evaluation in terms of homomorphisms.

Theorem 13.1. *If q is a* CQ *of the form (12.2) with k variables $\bar{x}$ in the head, and D is a database, then*

$$q(D) = \{\bar{a} \in Dom(D)^k \mid (q, \bar{x}) \to (D, \bar{a})\}. \tag{13.2}$$

Here is a simple example that illustrates the above characterization.

Example 13.2. Consider the CQ q and the database D given in Example 12.2, which satisfy that $q(D) = \{(\text{‘p1’}, \text{‘John Smith’})\}$. By the characterization given in Theorem 13.1, we know that $\big(q, (x, y)\big) \to \big(D, (\text{‘p1’}, \text{‘John Smith’})\big)$. To see why this is the case, recall that we need to check whether $\big(S_q, (x, y)\big) \to \big(D, (\text{‘p1’}, \text{‘John Smith’})\big)$, where:

$$S_q \;=\; \{\text{Person}(x, y, z), \text{Profession}(x, \text{‘computer scientist’})\}. \tag{13.3}$$

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

$$\begin{aligned} &h(x) = \text{‘p1’} \\ &h(y) = \text{‘John Smith’} \\ &h(z) = \text{‘c1’} \\ &h(\text{‘computer scientist’}) = \text{‘computer scientist’}. \end{aligned}$$

Then we have that both facts $\text{Person}(h(x), h(y), h(z))$ and $\text{Profession}(h(x), h(\text{‘computer scientist’}))$ are elements of D, so that h is a homomorphism from S_q to D. Moreover, $h\big((x, y)\big) = (\text{‘p1’}, \text{‘John Smith’})$, so that h is also a homomorphism from $\big(S_q, (x, y)\big)$ to $\big(D, (\text{‘p1’}, \text{‘John Smith’})\big)$, thus witnessing the fact that $\big(S_q, (x, y)\big) \to \big(D, (\text{‘p1’}, \text{‘John Smith’})\big)$. □

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. We proceed to explain those properties.

By saying that an n-query q is preserved under homomorphisms, we essentially mean that if a tuple $\bar{a}$ is an answer to q on a database D, and $(D, \bar{a})$ can be homomorphically mapped to $(D', \bar{b})$, then $\bar{b}$ should be an answer to 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 should map constant values to themselves. Since $\text{Dom}(D) \subseteq \mathsf{Const}$ for every database D, it follows that $D \to D'$ if and only if $D \subseteq D'$. Thus, the notion of homomorphism among databases, rather than sets of atoms, is nothing other than subset inclusion. However, the real intention underlying the notion of homomorphism is to preserve the structure, possibly by leaving some constants unchanged.

To overcome this mismatch, we need a machinery that allows us to convert a database into a set of 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 13.3. Consider databases $D_1 = \{R(a,b), R(b,a)\}$ and $D_2 = \{R(c,c)\}$. If $C_1 = \emptyset$, then we have that $\mathsf{V}_{C_1}(a)$ and $\mathsf{V}_{C_1}(b)$ are distinct elements of Var, say $\mathsf{V}_{C_1}(a) = x$ and $\mathsf{V}_{C_1}(b) = y$. Hence, $\mathsf{V}_{C_1}(D_1) = \{R(x,y), R(y,x)\}$ and $\mathsf{V}_{C_1}((a,b)) = (x,y)$, 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 $\mathsf{V}_{C_2}(D_1) = \{R(a,b), R(b,a)\}$ and $\mathsf{V}_{C_2}((a,b)) = (a,b)$. Therefore, it does not hold that $(D_1, (a,b)) \to_{C_2} (D_2, (c,c))$, since it does not hold that $\big(\mathsf{V}_{C_2}(D_1), \mathsf{V}_{C_2}((a,b))\big) \to \big(D_2, (c,c)\big)$. □

Definition 13.4 (Preservation under homomorphisms). *Let q be an n-ary query over a schema $\boldsymbol{S}$, and let C be the set of constants occurring in q. We say that q is* preserved under homomorphisms *if, for every two databases D and D' over $\boldsymbol{S}$, and tuples $\bar{a} \in Dom(D)^n$ and $\bar{b} \in Dom(D')^n$, it holds that*

$$\text{if } (D, \bar{a}) \to_C (D', \bar{b}) \text{ and } \bar{a} \in q(D), \text{ then } \bar{b} \in q(D').$$

We can then show the following for CQs.

Proposition 13.5. *Every* CQ *q is preserved under homomorphisms.*

Proof. Let $q(\bar{x})$ be a CQ over a schema $\mathbf{S}$, and let C be the set of constants mentioned in q. Assume that $(D, \bar{a}) \to_C (D', \bar{b})$, and let $\bar{a} \in q(D)$. Let h be a homomorphism witnessing $(\mathsf{V}_C(D), \mathsf{V}_C(\bar{a})) \to (D', \bar{b})$. By Theorem 13.1, $(q, \bar{x}) \to (D, \bar{a})$ via some homomorphism h'. Then $h_q = \mathsf{V}_C \circ h'$ is a homomorphism witnessing $(q, \bar{x}) \to (\mathsf{V}_C(D), \mathsf{V}_C(\bar{a}))$, since h_q preserves constants in q: if a is a constant in C, then $\mathsf{V}_C(h'(a)) = a$ by definition. Since homomorphisms compose, $h \circ h_q$ is a homomorphism from $(q, \bar{x})$ to $(D', \bar{b})$, and thus, again by Theorem 13.1, $\bar{b} \in q(D')$, as needed. □

A query q over a schema $\mathbf{S}$ is said to be *monotone* if for every two databases D and D' over $\mathbf{S}$, we have that

$$\text{if } D \subseteq D', \text{ then } q(D) \subseteq q(D').$$

Homomorphism preservation implies monotonicity of CQs.

Corollary 13.6. *Every* CQ *is monotone.*

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

Proof. Let q be a CQ over $\mathbf{S}$, with C the set of constants occurring in it, and let D and D' be databases over $\mathbf{S}$. If $D \subseteq D'$, then V_C^{-1} is a homomorphism from $(\mathsf{V}_C(D), \mathsf{V}_C(\bar{a}))$ to $(D', \bar{a})$ and, thus, $(D, \bar{a}) \rightarrow_C (D', \bar{a})$. Assume now that $\bar{a} \in q(D)$. By Proposition 13.5, we get that $\bar{a} \in q(D')$, as needed. $\square$

The second preservation result stated here concerns *direct products*. First we 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. That is, 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 direct product is different from the Cartesian product: the latter, of two binary relations, would produce a relation of arity four, while their direct product is still a binary relation.

The definition of direct products for databases is essentially the same, modulo one small technical detail. Elements of databases come from the set 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 $\tau : \mathsf{Const} \times \mathsf{Const} \rightarrow \mathsf{Const}$. A typical example, if Const is enumerated as $\{c_0, c_1, c_2, \dots\}$ 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' over the same schema $\mathbf{S}$, their direct product $D \otimes D'$ is a database over $\mathbf{S}$ that, for each n-ary relation R in $\mathbf{S}$, contains the following facts:

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

Technically speaking, this definition depends on the choice of a pairing function, but this choice is irrelevant as far as FO queries are concerned (see Exercise 5).

We state preservation under direct products for Boolean queries without constants, as this is what we need to understand the limitations of CQs. Exercises 6 and 7 explain how these results can be extended to queries with constants and free variables, respectively.

Definition 13.7 (Preservation under direct products). *A Boolean query q over a schema $\boldsymbol{S}$ is* preserved under direct products *if, for every two databases D and D' over $\boldsymbol{S}$, $D \models q$ and $D' \models q$ imply $D \otimes D' \models q$.*

We can then show the following for CQs.

Proposition 13.8. *Every Boolean* CQ *is preserved under direct products.*

Proof. As stated earlier, in this simple proof we only consider CQs that do not mention constants. Let q be such a Boolean CQ over a schema $\mathbf{S}$, and let D and D' be databases over $\mathbf{S}$ such that $D \models q$ and $D' \models q$. Then, by Theorem 13.1, we have homomorphisms h, g witnessing $q \rightarrow D$ and $q \rightarrow D'$, 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 $R(h(u_1), \ldots, h(u_n))$ is a fact of D and $R(g(u_1), \ldots, g(u_n))$ is a fact of D'. Hence,

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

is a fact of $D \otimes D'$, proving that f is a homomorphism from q to $D \otimes D'$. Thus, by Theorem 13.1, $D \otimes D' \models q$. □

Expressiveness of CQs

The above preservation results allow us to delineate the expressiveness boundaries of CQs. Recall that CQs have the same expressiveness as SPJ queries, i.e., RA queries in which we do *not* have inequality in selections, union (and disjunction in selection conditions), and difference. We now formally prove that none of these is expressible as a CQ. Also notice that in defining CQs, we disallowed explicit equality: CQs correspond to the FO queries in the fragment that is the $\exists, \wedge$-closure of relational atoms. Implicit equality of course is allowed, by reusing variables. We can now show that by adding explicit equality one obtains queries that cannot be expressed as CQs.

- CQ*s cannot express inequality.* This is because CQs with inequalities are not preserved under homomorphisms. Indeed, consider a query $\varphi_1 = \exists x \exists y \big(R(x, y) \wedge x \neq y\big)$. Given databases $D = \{R(a, b)\}$ and $D' = \{R(c, c)\}$, we have that $D \rightarrow_{\emptyset} D'$. However, $D \models \varphi_1$ while $D' \not\models \varphi_1$. As a second example, consider the query $\varphi_2 = \exists x\, (U(x) \wedge x \neq a)$, where a is a constant. Given $D = \{U(b)\}$ and $D' = \{U(a)\}$, we have that $D \rightarrow_{\{a\}} D'$. However, $D \models \varphi_2$ while $D' \not\models \varphi_2$.
- CQ*s cannot express negative relational atoms.* This is because such queries are not monotone. Consider a Boolean query $\varphi = \neg P(a)$, where a is a constant. If we take $D = \{\,\}$ and $D' = \{P(a)\}$, then $D \subseteq D'$ but $D \models \varphi$ while $D' \not\models \varphi$.
- CQ*s cannot express difference.* This is because difference is not monotone. Consider a query $\varphi = \exists x (P(x) \wedge \neg Q(x))$. If we take $D = \{P(a)\}$ and $D' = \{P(a), Q(a)\}$, then $D \subseteq D'$ but $D \models \varphi$ while $D' \not\models \varphi$.
- CQ*s cannot express union.* Consider a Boolean query $\varphi = \exists x\, (R(x) \vee S(x))$. Let $D = \{R(a)\}$ and $D' = \{S(a)\}$. Then $D \models \varphi$ and $D' \models \varphi$, but $D \otimes D'$ has no facts and, thus, it does not make φ true. Hence, by Proposition 13.8, φ cannot be expressed by a CQ.
- CQ*s cannot express explicit equality.* Consider a Boolean query $\varphi_1 = \exists x \exists y\, (x = y)$, and take databases D, D' as in the previous union example. Then $D \models \varphi_1$ and $D' \models \varphi_1$, but $D \otimes D' \not\models \varphi_1$.

14

Conjunctive Query Evaluation

Here we study the complexity of CQ-Evaluation. Recall that this is the problem of verifying, given a database D over a schema $\mathbf{S}$, a CQ $q(\bar{x})$ over $\mathbf{S}$, and a tuple $\bar{a}$ over the elements of $\mathrm{Dom}(D)$ of the same length as $\bar{x}$, whether it is the case that $\bar{a} \in q(D)$. In Chapter 7, we study the evaluation problem for the case of FO showing that it is PSPACE-complete. For the case of conjunctive queries, the complexity is considerably lower (under widely-held complexity assumptions).

Theorem 14.1. CQ-Evaluation *is* NP*-complete. Moreover, this problem remains* NP*-hard even for the case of Boolean CQs over a schema that consists of a single binary relation symbol.*

Proof. For the upper bound, recall that Theorem 13.1 tell us that checking whether $\bar{a} \in q(D)$ is equivalent to checking if there is a homomorphism from $(q, \bar{x})$ to $(D, \bar{a})$. Thus, to verify whether $\bar{a} \in q(D)$, one can guess a function $h : \mathrm{Dom}(D_q) \to \mathrm{Dom}(D)$ and then verify in polynomial time if h is a homomorphism from $(q, \bar{x})$ to $(D, \bar{a})$. This shows that CQ-Evaluation is in NP.

For the lower bound, we reduce from Clique. Recall that this is the problem whose input consists of a graph $G = (V, E)$ without loops and an integer $k \geq 1$, and the question to answer is whether G contains a *clique* of size k, i.e., a set of k nodes in V such that any two different nodes in the set are adjacent. Consider an input to Clique given by $G = (V, E)$ and $k \geq 1$. From this input, we construct as follows a database D and a Boolean CQ q over a schema that consists of a unary relation symbol node and a binary relation symbol edge:

1. The interpretation of node and edge over D are V and E, respectively; that is, $\mathsf{node}^D = V$ and $\mathsf{edge}^D = E$.
2. The boolean CQ q is defined as

$$\exists x_1 \cdots \exists x_k \left(\bigwedge_{i=1}^{k} \mathsf{node}(x_i) \wedge \bigwedge_{i,j \in [1,k] \,:\, i \neq j} \mathsf{edge}(x_i, x_j) \right).$$

Clearly, D and q can be constructed in polynomial time from G. Moreover, it is easy to see that G has a clique of size k if and only if $q(D) = \texttt{true}$. $\square$

For the data complexity of CQ-Evaluation, this is in DLogSpace as for the full language of FO queries (see Theorem 7.2).

The parameterized complexity of CQ-Evaluation

As mentioned before, queries are much smaller than databases in practice. This motivates the idea of data complexity, in which the cost of evaluation is measured only in terms of the size of the data assuming the query to be fixed. However, an algorithm whose running time is $O(||D||^{||q||})$, which is tractable in data complexity, cannot be considered to be practical if the database D is large, even if q is small.

A finer complexity notion comes in handy at this point. This notion is *parameterized complexity*, which is relevant in any case in which we need to classify the complexity of a problem depending on some relevant parameters. In our context, it makes sense to consider the size of a database and the size of a query as separate parameters when developing query answering algorithms, and to prefer algorithms that take less time on the former parameter. For example, a query evaluation algorithm that runs in time $O(\|D\| \cdot \|q\|^2)$ would be better in practice than an algorithm that runs in time $O(\|D\|^2 \cdot \|q\|)$. Moreover, if the difference between $\|D\|$ and $\|q\|$ is very large, as it usually happens in practice, then even an algorithm running in time $O(\|D\| \cdot 2^{\|q\|})$ could be better in practice than an algorithm running in time $O(\|D\|^2 \cdot \|q\|)$.

Next we define some fundamental notions of parameterized complexity, with the goal of using them to improve our understanding of CQ-Evaluation when considering the size of a database and the size of a query as separate parameters. Given an alphabet Σ, a *parameter* is a function $\kappa : \Sigma^* \to \mathbb{N}$, and a *parameterized language* is a pair (L, κ) where L is a language over Σ. A parameterized language (L, κ), or simply language (L, κ) from now on, is said to be *fixed-parameter tractable* if there exists a polynomial $p(n)$, a computable function $f : \mathbb{N} \to \mathbb{R}_0^+$, and an algorithm that decides L in time

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

That is, (L, κ) is fixed-parameter tractable if there exists an algorithm which is allowed to decide whether $w \in L$ in time arbitrarily large in the parameter $k(w)$, but polynomial in the size of the input w, and thus an algorithm designed with the idea in mind that $k(w)$ is much smaller than w. The class FPT consists of all languages (L, κ) which are fixed-parameter tractable.

The motivation of this section is to study CQ-Evaluation when the size of the query is considered as a parameter, which is formalized as the following parameterized language.

PROBLEM:	p-CQ-Evaluation
INPUT:	a database D, a CQ $q(\bar{x})$, and a tuple $\bar{a}$ over $\text{Dom}(D)$
PARAMETER:	$\|q\|$
OUTPUT:	yes if $\bar{a} \in q(D)$, and no otherwise

In particular, we would like to answer this question: is p-CQ-Evaluation in FPT? To answer such a question, consider first the algorithm for CQ-Evaluation that can be derived from the proof that this problem is in NP. Assume given a database D over a schema $\mathbf{S}$, a CQ $q(\bar{x})$ over $\mathbf{S}$, and a tuple $\bar{a}$ over the elements of $\text{Dom}(D)$ of the same length as $\bar{x}$. Then to verify if $\bar{a} \in q(D)$, we can check for every function $h : \text{Dom}(D_q) \to \text{Dom}(D)$, whether it is a homomorphism from $(q, \bar{x})$ to $(D, \bar{a})$; if at least one of these functions is a homomorphism then we answer yes, otherwise we answer no. Given that the number of such functions is $|\text{Dom}(D)|^{|\text{Dom}(D_q)|}$, we have that this algorithm runs in time

$$O(\|D\|^{\|q\|} \cdot r(\|D\| + \|q\|)), \tag{14.1}$$

where $r(n)$ is a polynomial (notice that the size of $\bar{a}$ is not included in (14.1) as it is polynomially bounded by $\|D\|$ and $\|q\|$). Thus, we cannot conclude that p-CQ-Evaluation is in FPT from the existence of this algorithm, as expression (14.1) is not of the form $O(p(\|D\|) \cdot f(\|q\|))$ for some polynomial p and arbitrary computable function f.

It is widely believed that no fixed-parameter tractable algorithm for evaluating CQs exists. But then the natural question is: how can we prove that p-CQ-Evaluation is not in FPT? Some complexity classes have been defined in the parameterized complexity area to prove that a parameterized language is not FPT. Such classes are widely believed to properly contain FPT, so if a parameterized language is complete for one of them, then this is considered as a strong evidence that such language is not in FPT. Notice here the analogy with classes such as NP and PSPACE: it is not known whether such 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. Next we define one of such classes, namely W[1], which allows us to pinpoint the exact complexity of p-CQ-Evaluation.

To define W[1], we need to introduce some terminology. Let $\mathbf{S}$ be a database schema, X be a relation name of arity m not mentioned in $\mathbf{S}$ and φ an FO-sentence over $\mathbf{S} \cup \{X\}$. Given a database D over $\mathbf{S}$ and a relation $S \subseteq \text{Dom}(D)^m$, we use notation $D \models \varphi(S)$ to indicate that $D' \models \varphi$, where D' is a database over $\mathbf{S} \cup \{X\}$ obtained by extending D with $X^{D'} = S$ (in particular, $\text{Dom}(D') = \text{Dom}(D)$ and $R^{D'} = R^D$ for every $R \in \mathbf{S}$). Moreover, we define p-WD$_\varphi$ as the following parameterized language.

PROBLEM:	p-WD$_\varphi$
INPUT:	a database D over $\mathbf{S}$ and $k \in \mathbb{N}$
PARAMETER:	k
OUTPUT:	yes if there exists $S \subseteq \mathrm{Dom}(D)^m$ such that $|S| = k$ and $D \models \varphi(S)$, and no otherwise

Notice that φ is fixed in the definition of p-WD$_\varphi$, so a different FO-sentence ψ of the form previously described gives rise to a different parameterized language p-WD$_\psi$. Now, a parameterized language (L, κ) is said to be in W[1] if there exists a database schema $\mathbf{S}$, a relation name X not mentioned in $\mathbf{S}$, and a *universal* FO-sentence φ over $\mathbf{S} \cup \{X\}$, such that (L, κ) can be *reduced* to p-WD$_\varphi$. Recall that an FO-sentence is universal if it is of the form $\forall \bar{x}\, \psi(\bar{x})$, where ψ is quantifier free. We do not formally define here the notion of reduction used for parameterized languages (see Exercise 10 for a formal definition of this notion), we just ask the reader to keep in mind that FPT is closed under this notion of reduction: if (L_1, κ_1) can be reduced to (L_2, κ_2) and $(L_2, \kappa_2) \in$ FPT, then $(L_1, \kappa_1) \in$ FPT.

To give some intuition about the definition of W[1], consider the following parameterized language over graphs.

PROBLEM:	p-Clique
INPUT:	a graph G without loops and an integer $k \geq 1$
PARAMETER:	k
OUTPUT:	yes if G has a clique of k nodes, and no otherwise

The parameterized language p-Clique is in W[1]. To prove this, consider an input of p-Clique consisting of a graph $G = (V, E)$ without loops and an integer $k \geq 1$. Then assume that $\mathbf{S}$ is a schema consisting of a unary relation symbol node and a binary relation symbol edge, and that elem is a unary relation symbol (not mentioned in $\mathbf{S}$), and define φ as the following universal FO-sentence over $\mathbf{S} \cup \{\mathsf{elem}\}$:

$$\forall x \forall y\, \big((\mathsf{elem}(x) \wedge \mathsf{elem}(y) \wedge x \neq y) \rightarrow \mathsf{edge}(x, y)\big).$$

Moreover, define an input of p-WD$_\varphi$ consisting of a database D over the schema $\mathbf{S}$ and the value k, where $\mathsf{node}^D = V$ and $\mathsf{edge}^D = E$. The FO-sentence φ is used to check whether the nodes stored in the relation elem form a clique and, thus, it holds that G has a clique with k nodes if and only if there exists $S \subseteq \mathrm{Dom}(D)$ such that $|S| = k$ and $D \models \varphi(S)$. In other words, $(G, k) \in$ p-Clique if and only if $(D, k) \in$ p-WD$_\varphi$, which shows that p-Clique can be reduced to p-WD$_\varphi$ and, therefore, p-Clique belongs to W[1]. As we mentioned before, we do not provide the formal definition of reduction for parameterized languages, but it is clear that the previous reduction can be computed in polynomial time, and that the existence of this reduction shows that if p-WD$_\varphi \in$ FPT, then p-Clique $\in$ FPT.

It is known that FPT $\subseteq$ W[1] and p-Clique is W[1]-complete. Moreover, it is widely believed that FPT is properly contained in W[1] and, thus, it is widely believed that p-Clique $\notin$ FPT (as FPT is closed under the notion of reduction used for parameterized languages). We use this result to prove that the same holds for p-CQ-Evaluation, thus providing strong evidence that this problem is not fixed-parameter tractable.

Theorem 14.2. p-CQ-Evaluation *is* W[1]*-complete.*

Proof. For the lower bound, we reduce from p-Clique. We use exactly the same reduction as for the lower bound in Theorem 14.1. It is important to reflect on why from the existence of this reduction, one can conclude that if p-CQ-Evaluation $\in$ FPT, then p-Clique $\in$ FPT. Notice that the parameter k for the input (G, k) of p-Clique is related with the parameter $\|q\|$ for the input (D, q) of p-CQ-Evaluation. In fact, we have that $\|q\| \leq c \cdot \log k \cdot k^2$ for some fixed constant $c \in \mathbb{R}^+$. Such a relationship is fundamental in the definition of reduction for parameterized languages, as it allows to prove that FPT is closed under this notion of reduction. In our particular case, notice that if there exists an algorithm for p-CQ-Evaluation that runs in time $O(p(\|D\|) \cdot f(\|q\|))$, where p is a polynomial and f is an arbitrary computable function, then one can conclude that p-Clique $\in$ FPT by using the reduction and the fact that $f(\|q\|) \leq f(c \cdot \log k \cdot k^2) = g(k)$ (we assume without loss of generality that f is nondecreasing). Hence, we have that p-Clique can be reduced to p-CQ-Evaluation, from which we conclude that p-CQ-Evaluation is W[1]-hard.

For the upper bound, we focus on the proof that p-CQ-Evaluation is in W[1] when restricted to Boolean conjunctive queries over a schema consisting of a binary relation symbol edge. We leave as an exercise for the reader to prove that p-CQ-Evaluation is in W[1] when no restrictions are imposed on queries and their schemata. Consider an input of p-CQ-Evaluation consisting of a database D and a Boolean conjunctive query q, both over the schema consisting of the binary relation symbol edge. Then define a schema $\mathbf{S}$ consisting of unary relation symbols var and const, and binary relation symbols edge_1 and edge_2. Moreover, consider a binary relation symbol hom (not mentioned in $\mathbf{S}$) and define a universal FO-sentence φ over $\mathbf{S} \cup \{\mathsf{hom}\}$ as follows:

$$\begin{aligned}
&\forall x \forall y \forall z \left((\mathsf{hom}(x, y) \wedge \mathsf{hom}(x, z)) \rightarrow y = z \right) \wedge \\
&\forall x \forall y \left(\mathsf{hom}(x, y) \rightarrow \left(\mathsf{var}(x) \wedge \mathsf{const}(y) \right) \right) \wedge \\
&\forall x_1 \forall y_1 \forall x_2 \forall y_2 \left((\mathsf{edge}_1(x_1, y_1) \wedge \mathsf{hom}(x_1, x_2) \wedge \mathsf{hom}(y_1, y_2)) \rightarrow \mathsf{edge}_2(x_2, y_2) \right).
\end{aligned}$$

Consider then an input of $\mathsf{p\text{-}WD}_\varphi$ consisting of a database D' over the schema $\mathbf{S}$ and a value $k \in \mathbb{N}$ defined as follows. Assuming that the set of variables occurring in q is $\{x_1, \ldots, x_n\}$, we have that

$$\begin{aligned}
\mathsf{var}^{D'} &= \{x_1, \ldots, x_n\}, \\
\mathsf{cons}^{D'} &= \mathrm{Dom}(D).
\end{aligned}$$

Thus, unary predicate var is used to store the variables occurring in q, while unary predicate const is used to store the constants occurring in D. Moreover, we have that

$$\begin{aligned} \mathsf{edge}_1^{D'} &= \{(x_i, x_j) \mid \mathsf{edge}(x_i, x_j) \text{ occurs in } q\}, \\ \mathsf{edge}_2^{D'} &= \mathsf{edge}^D, \end{aligned}$$

and, hence, binary predicate edge_1 is used to store the relational atoms occurring in q, while binary predicate edge_2 is used to store the facts of D. Finally, we have that $k = n$, that is, k is equal to the number of variables occurring in q.

With the definitions of D' and k, we can now explain the roles of the conjuncts in FO-sentence φ. The first conjunct $\forall x \forall y \forall z\, ((\mathsf{hom}(x,y) \wedge \mathsf{hom}(x,z)) \rightarrow y = z)$ indicates that hom represents a function, as only one value can be associated to x. The second conjunct indicates that hom maps variables of q into constants of D. Finally, the third conjunct indicates that hom represents a homomorphism from q to D. But notice that φ does not impose the restriction that every variable occurring q has to be mapped to a constants of D, as this requires of a non-universal FO-sentence of the form

$$\forall x\, (\mathsf{var}(x) \rightarrow \exists y\, (\mathsf{const}(y) \wedge \mathsf{hom}(x,y))).$$

Instead, the parameter $k = n$ is used to force hom to map every variable occurring in q, as n is the number of variables occurring in this query.

Summing up, we have that $q(D) = \texttt{true}$ if and only if there exists $S \subseteq \mathrm{Dom}(D')^2$ such that $|S| = k$ and $D' \models \varphi(S)$. Thus, we have provided a reduction from p-CQ-Evaluation to p-WD$_\varphi$, from which we conclude that p-CQ-Evaluation is in W[1] (when restricted to Boolean conjunctive queries over a single binary relation). Again, it is important to notice that the reduction we have provided shows that if p-WD$_\varphi \in$ FPT, then p-CQ-Evaluation $\in$ FPT. □

The reader may wonder why the number 1 is included in the name W[1] of the class considered in this section. The reason is that W[1] is the first level of a hierarchy of classes W[t] for every $t \geq 1$. More specifically, the class W[t] is defined exactly as the class W[1] but allowing the FO-sentence φ in p-WD$_\varphi$ to be of the form $\forall \bar{x}_1 \exists \bar{x}_2 \cdots Q\bar{x}_t\, \psi$, where ψ is quantifier free and $Q = \exists$ if t is even, and $Q = \forall$ if t is odd. The W-hierarchy consists of the union of all the classes W[t]. The parameterized complexity of other important problems in database theory have been established by using classes in this fundamental hierarchy.

15

Containment, Minimization, and Cores

The main goal of query optimization is to transform a query into an equivalent one that can be evaluated more efficiently. To do so, we need static analysis tools that let us check for containment and equivalence of queries. We have seen in Chapter 8 that for FO, or RA queries, these problems are undecidable. For CQs, on the other hand, there are algorithms for checking containment, equivalence, and finding the most efficient equivalent queries, where efficiency is measured as the number of joins a query performs.

Containment

Recall that containment is a basic static analysis problem: given two queries q and q' over a schema $\mathbf{S}$, q is contained in q', denoted by $q \subseteq q'$, if $q(D) \subseteq q'(D)$ for every database D over the schema $\mathbf{S}$.

Besides, recall our notational conventions about conjunctive queries detailed in Chapter 12. In particular, given a CQ $q(\bar{x}) \coloneq R_1(\bar{u}_1), \ldots, R_n(\bar{u}_n)$, we write S_q for the set of atoms in its body, that is, for $\{R_1(\bar{u}_1), \ldots, R_n(\bar{u}_n)\}$. In this way, given two CQs $q(\bar{x})$ and $q'(\bar{x}')$, we can use the notation $(q, \bar{x}) \to (q', \bar{x}')$ to indicate that there exists a homomorphism from $(S_q, \bar{x})$ to $(S_{q'}, \bar{x}')$. Recall also that for a set of atoms S, we write $S^{\downarrow}$ for the grounding of S, and G_S for the bijective homomorphism from S to $S^{\downarrow}$. Observe that, by definition, $\mathsf{G}_S(S) = S^{\downarrow}$. We proceed to show that the notion of homomorphism between queries plays a crucial role when deciding containment among queries.

Theorem 15.1 (Homomorphism Theorem). *Let $q(\bar{x})$ and $q'(\bar{x}')$ be CQs over the same schema such that $\bar{x}$ and $\bar{x}'$ are of the same length. Then:*

$$q \subseteq q' \quad \textit{if and only if} \quad (q', \bar{x}') \to (q, \bar{x}).$$

Proof. ($\Rightarrow$) Assume that $q \subseteq q'$. It is clear that $\mathsf{G}_{S_q}(\bar{x}) \in q(\mathsf{G}_{S_q}(S_q))$, and therefore, by hypothesis, $\mathsf{G}_{S_q}(\bar{x}) \in q'(\mathsf{G}_{S_q}(S_q))$. By Theorem 13.1, there exists a homomorphism h from $(S_{q'}, \bar{x}')$ to $(\mathsf{G}_{S_q}(S_q), \mathsf{G}_{S_q}(\bar{x}))$. Clearly, $\mathsf{G}_{S_q}^{-1} \circ h$ is a homomorphism from $(S_{q'}, \bar{x}')$ to $(S_q, \bar{x})$, as needed.

($\Leftarrow$) Conversely, assume that $(q', \bar{x}') \to (q, \bar{x})$, that is, that there exists a homomorphism h from $(S_{q'}, \bar{x}')$ to $(S_q, \bar{x})$. Consider an arbitrary database D, and assume that $\bar{a} \in q(D)$. By Theorem 13.1, 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 13.1. Therefore, we have that $q(D) \subseteq q'(D)$, from which we conclude that $q \subseteq q'$. $\square$

Here is a simple example that illustrates the usefulness of the Homomorphism Theorem.

Example 15.2. Consider again the relational schema from Chapter 3:

```
Person [ person_id, name, city_of_birth ]
Profession [ person_id, profession_name ]
```

and consider the following CQs over this schema:

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

Query q_1 is used to retrieve the list of names of computer scientists in a database D, while q_2 is used to retrieve the list of names of people who have a profession in D. As q_1 is more specific than q_2, we expect that $q_1(D) \subseteq q_2(D)$. In fact, we expect this containment to hold in general, so that $q_1 \subseteq q_2$. To verify this, we can use the Homomorphism Theorem and check whether the condition $(q_2, y_2) \to (q_1, y_1)$ holds, that is, whether $(S_{q_2}, y_2) \to (S_{q_1}, y_1)$ holds. In fact, this is the case since the function $h : \text{Dom}(S_{q_2}) \to \text{Dom}(S_{q_1})$ defined as $h(x_2) = x_1$, $h(y_2) = y_1$, $h(z_2) = z_1$, and $h(w_2) = \text{‘computer scientist’}$ is a homomorphism from (S_{q_2}, y_2) to (S_{q_1}, y_1). $\square$

As an immediate consequence of the Homomorphism Theorem, we can see that CQ evaluation and CQ containment are equivalent problems.

Corollary 15.3. *Let $q(\bar{x})$ and $q'(\bar{x}')$ be* CQ*s over the same schema such that $\bar{x}$ and $\bar{x}'$ are of the same length. Then*

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

Proof. By Theorem 15.1, $q \subseteq q'$ if and only if $(S_{q'}, \bar{x}') \to (S_q, \bar{x})$. The latter is equivalent to $(S_{q'}, \bar{x}') \to (\mathsf{G}_{S_q}(S_q), \mathsf{G}_{S_q}(\bar{x}))$. Indeed, assuming that $(S_{q'}, \bar{x}') \to (S_q, \bar{x})$ is witnessed via h, since homomorphisms compose, $\mathsf{G}_{S_q} \circ h$ 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 $(S_q, \bar{x})$. The claim follows by Theorem 13.1, which states that $(S_{q'}, \bar{x}') \to (\mathsf{G}_{S_q}(S_q), \mathsf{G}_{S_q}(\bar{x}))$ if and only if $\mathsf{G}_{S_q}(\bar{x}) \in q'(\mathsf{G}_{S_q}(S_q))$. $\square$

This, together with the complexity analysis of Chapter 14, establishes the complexity of the $\mathsf{CQ\text{-}Containment}$ problem, that is, the problem of checking, given two CQs q and q', whether $q \subseteq q'$.

Corollary 15.4. $\mathsf{CQ\text{-}Containment}$ *is* NP*-complete.*

Equivalence

Given two queries q and q' over a schema $\mathbf{S}$, q and q' are equivalent, denoted by $q \equiv q'$, if $q(D) = q'(D)$ for every database D over the schema $\mathbf{S}$. This is of course the same as having both containments: $q \subseteq q'$ and $q' \subseteq q$. We say that two CQs $q(\bar{x})$ and $q'(\bar{x}')$ are *homomorphically equivalent* if both conditions $(q', \bar{x}') \to (q, \bar{x})$ and $(q, \bar{x}) \to (q', \bar{x}')$ hold. As an immediate consequence of the Homomorphism Theorem, we obtain the following characterization for equivalence of CQs.

Corollary 15.5. *Two* CQ*s are equivalent if and only if they are homomorphically equivalent.*

Query optimization amounts to transforming a query into an equivalent one that is easier to evaluate. To evaluate a CQ $q(\bar{x}) \coloneq R_1(\bar{x}_1), \dots, R_n(\bar{x}_n)$, one has to perform $n - 1$ joins. Since joins are expensive operations, it is reasonable to consider their number as an indicator of the complexity of the CQ, and, thus, to attempt to find a CQ $q'(\bar{x}')$ that is equivalent to $q(\bar{x})$ and has the minimum number of atoms (and, hence, the minimum number of joins to be evaluated).

Definition 15.6 (Minimization of CQs). *Given a* CQ q*, a* CQ q' *is called a* minimization *of* q *if it is equivalent to* q *and has the smallest number of atoms among all the* CQ*s that are equivalent to* q.

It is clear that *a* minimization of a CQ q always exists, but then several questions arise: could there be multiple minimizations of q, and if so, how are they related to each other? For CQ optimization, these questions have a neat answer: there is essentially (up to renaming of variables) exactly one minimization of a CQ q. Moreover, such a minimization can be obtained by deleting some of the atoms of q. In fact, this minimization is called the core of q, a concept that we study next.

The core of a CQ

Consider a CQ $q(\bar{x}) \coloneq R_1(\bar{u}_1), \dots, R_n(\bar{u}_n)$ and $i \in \{1, \dots, n\}$. The CQ obtained from q by removing the atom $R_i(\bar{u}_i)$ is defined as

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

where $\bar{x}'$ is obtained from $\bar{x}$ by removing every variable occurring in $\bar{x}$ that is only mentioned in the atom $R_i(\bar{u}_i)$. For example, if we remove the atom $R(x)$ from the CQ $q_1(x, y) \coloneq R(x), S(y)$, then we obtain the CQ $q_1'(y) \coloneq S(y)$ as the variable x is only mentioned in the atom $R(x)$. On the other hand, if we remove the atom $R(x)$ from the CQ $q_2(x, y) \coloneq R(x), T(x, y)$, then we obtain the CQ $q_2'(x, y) \coloneq T(x, y)$. If we have that the condition $(q, \bar{x}) \to (q', \bar{x}')$ holds, then we can conclude that q and q' are homomorphically equivalent, as $\bar{x}$ and

$\bar{x}'$ must be equal and, thus, the identity is a homomorphism witnessing the other direction $(q', \bar{x}') \to (q, \bar{x})$. Hence, if $(q, \bar{x}) \to (q', \bar{x}')$ holds, then we can conclude from Corollary 15.5 that q and q' are equivalent CQs. Continuing this process of removing an atom from a CQ while the obtained query is equivalent, we end up with a CQ that is obtained from q by removing some atoms, and which is equivalent to q. This is the intuitive idea behind the minimization process of a CQ. In particular, the resulting query is called the *core* of q, which is a notion that is formalized in the following definition.

Definition 15.7 (Core of a CQ**).** *Let $q(\bar{x})$ be a* CQ*. A core of $q(\bar{x})$ is a* CQ *$q'(\bar{x})$ obtained by removing zero or more atoms from q and such that:*

1. *q and q' are homomorphically equivalent, and*
2. *removing any atom from q' results in a query that is no longer homomorphically equivalent to q.*

Here is a simple example that illustrates the notion of core of a CQ.

Example 15.8. Consider the Boolean CQ q_1 :- $R(x, y), R(x, z)$. Then the function h defined as $h(x) = x$, $h(y) = y$, $h(z) = y$ is a homomorphism from $\{R(x, y), R(x, z)\}$ to $\{R(x, y)\}$. Thus, q_1 is homomorphically equivalent to the CQ q_1' :- $R(x, y)$ obtained by removing the atom $R(x, z)$ from q_1. We conclude that q_1' is a core of q_1, since removing the single atom in q_1' does not produce a CQ homomorphically equivalent to q_1 (in fact, deleting the single atom in q_1' does not produce a CQ). Notice that in this case q_1'' :- $R(x, z)$ is also a core of q_1. On the other hand, consider the Boolean CQ q_2 :- $R(x, y), R(y, z)$. Then we cannot remove any atom, as 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 $\{R(y, z)\}$. Hence, q_2 is its own core.

Finally, consider the CQ $q_3(x, y, z)$:- $R(x, y), R(x, z)$, which is defined as q_1 but having all variables in the output. Then by removing the atom $R(x, z)$ from q_3, we obtain the CQ $q_3'(x, y)$:- $R(x, y)$. In this case, there is no homomorphism from $(S_{q_3}, (x, y, z))$ to $(S_{q_3'}, (x, y))$, as every mapping applied to (x, y, z) will produce a tuple with three elements, which then cannot be equal to (x, y). Hence, q_3' is not homomorphically equivalent to q_3. In the same way, we can conclude that the CQ obtained from q_3 by removing atom $R(x, y)$ is not homomorphically equivalent to q_3, so that q_3 is its own core. □

Notice that in the previous example, query q_1 has two distinct cores q_1' :- $R(x, y)$ and q_1'' :- $R(x, z)$. However, these two cores are essentially the same query: one can be obtained from the other by a simple renaming of variables, namely by replacing y by z or vice versa. This is not an isolated phenomenon; in fact, we show in the following theorem that all cores of a CQ are the same up to renaming of variables. We say that two CQs $q(\bar{x})$ and $q'(\bar{x}')$ are *isomorphic* if one can be turned into the other by renaming of variables, that is, if there exists a function $f : \text{Dom}(S_q) \to \text{Dom}(S_{q'})$ such that f is the identity on the constants, f is a bijection, $f(\bar{x}) = \bar{x}'$ and $f(S_q) = S_{q'}$, where $f(S_q)$ is the set resulting from replacing each atom $R(\bar{u}) \in S_q$ by $R(f(\bar{u}))$.

Theorem 15.9. *Every* CQ *has has at least one core, and all cores of a* CQ *are isomorphic.*

Proof. To see that every CQ q has a core, note that if q is not its own core, then it is possible to remove at least one atom from q and obtain a CQ q' that is homomorphically equivalent to q. If q' is its own core, then it also a core of q. Otherwise, iteratively continue with this process until reaching a core of q.

Now we prove that all cores of $q(\bar{x})$ are isomorphic. Let $q_1(\bar{x})$ and $q_2(\bar{x})$ be two distinct cores of q, and let g be a homomorphism from $(q, \bar{x})$ to $(q_1, \bar{x})$. Since both q_1 and q_2 are homomorphically equivalent to q, we have that q_1 and q_2 are homomorphically equivalent (it is not difficult to prove that *being homomorphically equivalent* is an equivalence relation on the set of CQs). Hence, we have homomorphisms h_1, h_2 witnessing $(q_1, \bar{x}) \to (q_2, \bar{x})$ and $(q_2, \bar{x}) \to (q_1, \bar{x})$, respectively. Next we show that h_1 is a function witnessing the fact that $q_1(\bar{x})$ and $q_2(\bar{x})$ are isomorphic.

Given that h_1 is a homomorphism from $(q_1, \bar{x})$ to $(q_2, \bar{x})$, we know that $h_1 : \mathrm{Dom}(S_{q_1}) \to \mathrm{Dom}(S_{q_2})$ is the identity on the constants and $h_1(\bar{x}) = \bar{x}$. Moreover, h_1 is a bijective function. To prove this, assume first that h_1 is not surjective, that is, there exists a variable $z \in \mathrm{Dom}(S_{q_2})$ that is not the image of any variable in $\mathrm{Dom}(S_{q_1})$ under h_1. Let $R(\bar{u}) \in S_{q_2}$ be an atom that mentions variable z. Then we have that $R(\bar{u})$ is not the image of any atom in S_{q_1} under h_1 and, thus, $h_1 \circ g$ is a homomorphism from $(q, \bar{x})$ to $(q_2, \bar{x})$ whose image is a proper subquery of q_2, contradicting the fact that $q_2(\bar{x})$ is a core of $q(\bar{x})$ since such a subquery is homomorphically equivalent to q. Second, assume that h_1 is not injective. Then we have $u, v \in \mathrm{Dom}(S_{q_1})$ such that $u \neq v$ and $h_1(u) = h_1(v)$. Hence, $h_2 \circ h_1(u) = h_2 \circ h_1(v)$, and we conclude that $h_2 \circ h_1 : \mathrm{Dom}(S_{q_1}) \to \mathrm{Dom}(S_{q_1})$ is a non-surjective homomorphism from $(q_1, \bar{x})$ to $(q_1, \bar{x})$, from which we conclude as before that $h_2 \circ h_1$ maps q_1 to one of its proper subqueries. Therefore, $h_2 \circ h_1 \circ g$ is a homomorphism from $q(\bar{x})$ to a proper subquery of $q_1(\bar{x})$, contradicting the fact that $q_1(\bar{x})$ is a core of $q(\bar{x})$ as such a subquery is homomorphically equivalent to q. Finally, we need to show that $h_1(S_{q_1}) = S_{q_2}$. Given that h_1 is a homomorphism from $(q_1, \bar{x})$ to $(q_2, \bar{x})$, we have that $h_1(S_{q_1}) \subseteq S_{q_2}$. If we assume that $h_1(S_{q_1}) \subsetneq S_{q_2}$, then again we conclude that $h_1 \circ g$ is a homomorphism from $(q, \bar{x})$ to $(q_2, \bar{x})$ whose image is a proper subquery of q_2, contradicting the fact that $q_2(\bar{x})$ is a core of $q(\bar{x})$. Therefore, we have that $h_1(S_{q_1}) = S_{q_2}$, and the claim follows. □

Since all cores of a CQ are isomorphic, we can unambiguously refer to *the* core of a CQ.

Minimization of CQs

Recall the notion of minimization of a CQ formalized in Definition 15.6. Next we show that the cores of CQs are precisely their minimizations.

Theorem 15.10. *Let $q(\bar{x})$ be a* CQ *and $q'(\bar{x})$ be the core of $q(\bar{x})$. Then $q'(\bar{x})$ is a minimization of $q(\bar{x})$, and each minimization of $q(\bar{x})$ is isomorphic to $q'(\bar{x})$.*

The proof of this theorem follows the same ideas as in the proof of Theorem 15.9, so it is left as an exercise for the reader. We conclude that to minimize a CQ, one simply has to find its core. This can be done by applying the simple algorithm ComputeCore shown below, which checks the atoms of a query, one-by-one, to see if their removal results in an equivalent query, and stops when no atoms can be deleted.

Algorithm 3 ComputeCore(q)

```
Input: A CQ $q(\bar{x})$
Output: A query $q^*(\bar{x})$ that is the 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 $q^*(\bar{x})$ :- $R_1(\bar{u}_1), \ldots, R_n(\bar{u}_n)$,  where $S = \{R_1(\bar{u}_1), \ldots, R_n(\bar{u}_n)\}$
```

Proposition 15.11. *Algorithm* ComputeCore(q) *computes the core of $q(\bar{x})$.*

Proof. At every stage of the algorithm, if $S = \{T_1(\bar{v}_1), \ldots, T_m(\bar{v}_m)\}$, then the CQ $q(\bar{x})$:- $T_1(\bar{v}_1), \ldots, T_m(\bar{v}_m)$ is homomorphically equivalent to $q(\bar{x})$. In addition, the algorithm stops with a CQ $q^*(\bar{x})$ that is a core of itself, as it cannot be homomorphically mapped into any of its proper subqueries. Therefore, $q^*(\bar{x})$ is also a core of $q(\bar{x})$. □

Note that algorithm ComputeCore(q) is non-deterministic: there could be several atoms $R(\bar{y}) \in S$ satisfying the condition of the while loop (in particular, the condition $(S, \bar{x}) \to (S - \{R(\bar{y})\}, \bar{x})$), and we do not indicate how to choose one. However, every choice will result in a core of $q(\bar{x})$, and since all these cores are isomorphic by Theorem 15.9, it does not matter which atom is chosen by the algorithm as the result will be the same (up to renaming of variables).

The complexity of this algorithm is exponential since we need to test whether there exists a homomorphism between two CQs. More specifically, the algorithm computes the core of q in polynomial time if we grant access to an NP-oracle that checks for the existence of homomorphisms. This implies that core computation belongs to the complexity class $\mathrm{FP}^{\mathrm{NP}}$ of functions that can be computed in polynomial time if one has access to an oracle that solves an NP-complete problem. However, the input q to this problem is relatively small, so this exponential time algorithm can be used oftenly in practice to compute the core of q.

16

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 database instances of a certain schema. This is achieved via integrity constraints, also called dependencies. It is not difficult to see that the presence of constraints does not affect the complexity of the evaluation problem for CQs. But how containment of CQs, studied in Chapter 15, is affected if we focus on databases that satisfy a given set of dependencies. In this chapter, we study this question in the case of functional dependencies (FDs) and inclusion dependencies (INDs).

Functional Dependencies

We first concentrate on FDs, and illustrate via an example that containment of CQs is affected if we focus on databases that satisfy a given set of FDs.

Example 16.1. Consider the CQs

$$\begin{aligned} q_1(x_1, y_1) &\text{ :- } R(x_1, y_1), R(y_1, z_1), R(x_1, z_1), \\ q_2(x_2, y_2) &\text{ :- } R(x_2, y_2), R(y_2, y_2). \end{aligned}$$

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. In fact, if we consider the database $D_1 = \{R(1,2), R(2,3), R(1,3)\}$, then it holds that $q_1(D_1) = \{(1,2)\}$ and $q_2(D_1) = \emptyset$, so that $q_1(D_1) \not\subseteq q_2(D_1)$.

Suppose now that q_1 and q_2 will be evaluated only over the databases that satisfy the functional dependency

$$\sigma \; = \; R : \{1\} \to \{2\}.$$

In particular, q_1 and q_2 will not be evaluated over the database D_1 since it does not satisfy σ. We can show that, for every database D such that $D \models \sigma$, it holds that $q_1(D) \subseteq q_2(D)$. To see that this is the case, consider an arbitrary database D satisfying σ, and assume that $(a, b) \in q_1(D)$, or, equivalently by Theorem 13.1, $(q_1, (x_1, y_1)) \rightarrow (D, (a, b))$ via a homomorphism h_1. Since $D \models \sigma$ and $R(h_1(x_1), h_1(y_1)), R(h_1(x_1), h_1(z_1)) \in D$, we have that $h_1(y_1) = h_1(z_1)$. Hence, given that $R(h_1(y_1), h_1(z_1)) \in D$, we conclude $(q_2, (x_2, y_2)) \rightarrow (D, (a, b))$ via h_2 such that $h_2(x_2) = h_1(x_1)$ and $h_2(y_2) = h_1(y_1) = h_1(z_1)$. $\square$

Our goal is to revisit the problem of containment of 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:	yes if $q \subseteq_\Sigma q'$, and no otherwise

We proceed to show the following result:

Theorem 16.2. CQ-Containment-FD *is* NP*-complete.*

It is clear that the NP-hardness is inherited from CQ containment without constraints (see Corollary 15.4). Let us now show the upper bound. 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 $(q', \bar{x}') \rightarrow (q, \bar{x})$. Although 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 $(q', \bar{x}') \rightarrow (S, \bar{u})$. This simple idea has been already illustrated by Example 16.1. 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.

Recall that the body of a CQ, written as a rule, can be seen as a set of atoms. As usual, for a CQ $q(\bar{x}) \text{ :- } R_1(\bar{u}_1), \ldots, R_n(\bar{u}_n)$, we write S_q for the set of atoms $\{R_1(\bar{u}_1), \ldots, R_n(\bar{u}_n)\}$. For brevity, we simply write $\mathrm{Chase}(q, \Sigma)$ instead of $\mathrm{Chase}(S_q, \Sigma)$ 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 16.3. *Let* $q(\bar{x})$ *and* $q'(\bar{x}')$ *be CQs over a schema* $\boldsymbol{S}$*, and* Σ *a set of FDs over* $\boldsymbol{S}$*. The following are equivalent:*

1. $q \subseteq_\Sigma q'$.
2. $\mathrm{Chase}(q, \Sigma) \neq \bot$ *implies* $(q', \bar{x}') \rightarrow (\mathrm{Chase}(q, \Sigma), h_{q,\Sigma}(\bar{x}))$.

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

((1) $\Rightarrow$ (2)) Assume that $q \subseteq_\Sigma q'$. It is clear that, if $S \neq \bot$, then $\mathsf{G}_S(\bar{u}) \in q(\mathsf{G}_S(S))$. Since, by Lemma 10.4, $S \models \Sigma$, which means that $\mathsf{G}_S(S) \models \Sigma$, we have that $\mathsf{G}_S(\bar{u}) \in q'(\mathsf{G}_S(S))$. By Theorem 13.1, 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.

((2) $\Rightarrow$ (1)) For this direction, we proceed by case analysis:

- Assume first that $S = \bot$. This implies that, for every database D over **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 over **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 $(q', \bar{x}') \rightarrow (S, \bar{u})$ via a homomorphism h. Let D be an arbitrary database over **S** such that $D \models \Sigma$, and assume that $\bar{a} \in q(D)$. By Theorem 13.1, $(q, \bar{x}) \rightarrow (D, \bar{a})$. Since $D \models \Sigma$, Lemma 10.7 implies that $(S, \bar{u}) \rightarrow (D, \bar{a})$ via a homomorphism g. Since homomorphisms compose, $g \circ h$ is a homomorphism from $(q', \bar{x}')$ to $(D, \bar{a})$. By Theorem 13.1, $\bar{a} \in q'(D)$, 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 16.3 and Theorem 13.1.

Corollary 16.4. *Let $q(\bar{x})$ and $q'(\bar{x}')$ be CQs over a schema **S**, and Σ a set of FDs over **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 10.6, Chase(q, Σ) can be computed in polynomial time. Moreover, if Chase$(q, \Sigma) \neq \bot$, then the chase homomorphism $h_{q,\Sigma}$ can be also computed in polynomial time. Since CQ-Evaluation is in NP (see Theorem 14.1), we conclude that CQ-Containment-FD is also in NP, and Theorem 16.2 follows.

Inclusion Dependencies

We now concentrate on INDs, and start by showing via a simple example that also in this case containment of CQs is affected.

Example 16.5. Consider the CQs

$$\begin{aligned} q_1(x_1, y_1) &\ \text{:-}\ R(x_1, y_1), R(y_1, z_1), P(z_1, y_1) \\ q2(x_2, y_2) &\ \text{:-}\ R(x_2, y_2), R(y_2, z_2), S(x_2, y_2, z_2). \end{aligned}$$

It is clear that $(q_2, (x_2, y_2)) \rightarrow (q_1, (x_1, y_1))$ 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 the inclusion dependencies

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

We can show that, for every such database D, $q_1(D) \subseteq q_2(D)$. 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 can also conclude that D contains the atom $R(h_1(y_1), c)$. Summing up, we can conclude that

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

This immediately 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.,

$$q_1'(x_1, y_1) \;\text{:-}\; R(x_1, y_1), R(y_1, z_1), P(z_1, y_1), S(x_1, y_1, w_1), R(y_1, w_1),$$

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

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:	yes if $q \subseteq_\Sigma q'$, and no otherwise

Although the complexity of CQ containment in the presence of FDs remains NP-complete (Theorem 16.2), this is not true for INDs:

Theorem 16.6. 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 $(q', \bar{x}') \to (S, \bar{x})$. This simple idea has been already illustrated by Example 16.5. As expected, the transformation of S_q into S can be done by exploiting the chase for INDs, which has been introduced in Chapter 11.

We are going to establish a result analogous to Theorem 16.3 for functional dependencies. However, since the chase for INDs might build an infinite set of atoms, we can only characterize CQ containment under possibly infinite databases. Given two CQs q and q', and a set Σ of INDs, q *is contained without restriction in* q' *under* Σ, denoted $q \subseteq_{\Sigma}^{\infty} q'$, if for every possibly infinite database D that satisfies Σ, it holds that $q(D) \subseteq q'(D)$.

As usual, for a CQ written as a rule of the form $q(\bar{x}) \coloneq R_1(\bar{u}_1), \ldots, R_n(\bar{u}_n)$, we write S_q for the set of atoms $\{R_1(\bar{u}_1), \ldots, R_n(\bar{u}_n)\}$. Henceforth, for brevity, we simply write $\mathrm{Chase}(q, \Sigma)$ instead of $\mathrm{Chase}(S_q, \Sigma)$. By providing a proof similar to that of Theorem 16.3, we can show the following:

Theorem 16.7. *Let $q(\bar{x})$ and $q'(\bar{x}')$ be CQs over a schema $\boldsymbol{S}$, and Σ a set of INDs over $\boldsymbol{S}$. It holds that $q \subseteq_{\Sigma}^{\infty} q'$ if and only if $(q', \bar{x}') \rightarrow (\mathrm{Chase}(q, \Sigma), \bar{x})$.*

The above theorem alone is of little use since we only care about 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 16.8. *Let $q(\bar{x})$ and $q'(\bar{x}')$ be CQs over a schema $\boldsymbol{S}$, and Σ a set of INDs over $\boldsymbol{S}$. It holds that $q \subseteq_{\Sigma} q'$ if and only if $q \subseteq_{\Sigma}^{\infty} q'$.*

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 16.7 and 16.8, combined with Theorem 13.1, is the following:

Corollary 16.9. *Let $q(\bar{x})$ and $q'(\bar{x}')$ be CQs over a schema $\boldsymbol{S}$, and Σ a set of INDs over $\boldsymbol{S}$. With $S = \mathrm{Chase}(q, \Sigma)$, $q \subseteq_{\Sigma} q'$ if and only if $\mathsf{G}_S(\bar{x}) \in q'(\mathsf{G}_S(S))$.*

Due to Corollary 16.9, one may think that the procedure for checking whether $q \subseteq_{\Sigma} q'$, which in turn will lead to the PSPACE upper bound claimed in Theorem 16.6, is to check whether $\mathsf{G}_S(\bar{x})$ belongs to the evaluation of q' over S, where $S = \mathrm{Chase}(q, \Sigma)$. However, it should not be forgotten that $\mathrm{Chase}(q, \Sigma)$ is, in general, infinite. Therefore, we need to rely on a refined procedure that avoids the explicit construction of $\mathrm{Chase}(q, \Sigma)$. We proceed to present a technical lemma that is the building block underlying this procedure.

For an IND $\sigma = R[i_1, \ldots, i_m] \subseteq P[j_1, \ldots, j_m]$, a tuple $\bar{u} = (u_1, \ldots, u_{\mathrm{ar}(R)})$, 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, $\mathsf{new}^V(\sigma, \bar{u}) = P(v_1, \ldots, v_{\mathrm{ar}(P)})$, where, for each $\ell \in [1, \mathrm{ar}(P)]$,

$$v_\ell \;=\; \begin{cases} u_{i_k} & \text{if } \ell = j_k, \text{ for } k \in [1, m], \\ x \in V & \text{otherwise} \end{cases}$$

such that, for each $i, j \in [1, \mathrm{ar}(P)] - \{j_1, \ldots, j_m\}$, $i \neq j$ implies $v_i \neq v_j$.[1] We can then show the following technical result; see Exercise 16.1.

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

Lemma 16.10. *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 $S = \text{Chase}(q, \Sigma)$, the following are equivalent:*

1. $\mathsf{G}_S(\bar{x}) \in q'(\mathsf{G}_S(S))$.
2. *There is a sequence of sets of variables $V_2, \dots, V_n$, for $n \geq 1$, a sequence of sets of atoms $S_1, \dots, S_n$, and a set of atoms $S' \subseteq \bigcup_{i \in [1,n]} S_i$, such that:*
   a) $|\bigcup_{i \in [2,n]} V_i| \leq 3 \cdot |S_{q'}| \cdot \max_{R \in \boldsymbol{S}}\{ar(R)\}$,
   b) *for each* $i \in [2, n]$, $V_i \cap (Dom(S_{i-1}) \cup Dom(S')) = \emptyset$,
   c) *for each* $i \in [1, n]$, $|S_i| \leq |S_{q'}|$,
   d) $S_1 \subseteq S_q$,
   e) *for each* $i \in [2, n]$ *and* $P(\bar{v}) \in S_i$, *there is* $\sigma = R[\alpha] \subseteq P[\beta] \in \Sigma$ *that is applicable on* S_{i-1} *with some* $\bar{u} \in R^{S_{i-1}}$ *such that* $P(\bar{v}) = \mathsf{new}^{V_i}(\sigma, \bar{u})$,
   f) *for each* $i \in [2, n]$ *and variable* $x \in Dom(S_i)$, *if* $x \notin Dom(S_{i-1})$, *then there is only one occurrence of* x *in* S_i,
   g) $|S'| \leq |S_{q'}|$, *and*
   h) $\mathsf{G}_{S'}(\bar{x}) \in q'(\mathsf{G}_{S'}(S'))$.

Lemma 16.10 leads to a relatively simple nondeterministic procedure for checking whether a CQ $q(\bar{x})$ is contained in a CQ $q'(\bar{x}')$ under a set Σ of INDs (see Algorithm 4). It essentially constructs the sequence of sets of variables $V_2, \dots, V_n$ and the sequence of sets of atoms $S_1, \dots, S_n$ as in Lemma 16.10 one after the other (if they exist), without storing more than two consecutive sets during its computation. In other words, it constructs V_i and S_i from V_{i-1} and S_{i-1}, respectively, and then it discards V_{i-1} and S_{i-1}. It also constructs on the fly the set S'. This is done by storing some of the atoms of S_{i-1} (possibly none) into S' before discarding it. Finally, the algorithm checks whether $\mathsf{G}_{S'}(\bar{x}) \in q'(\mathsf{G}_{S'}(S'))$ in which case it returns yes; otherwise, it returns no. We proceed to give a bit more detailed description of Algorithm 4.

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 understood 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 S' (see line 3); this step is part of the "on the fly" construction of the set S'. It also collects $2 \cdot |S_{q'}| \cdot \max_{R \in \mathbf{S}}\{\text{ar}(R)\}$ variables not occurring in S_q in the set V (see line 4). 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

Algorithm 4 CQ-IND-CONTAINMENT(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: If $q \subseteq_\Sigma q'$, then yes; otherwise, no.

```
1: $S_\triangledown := A$, where $A \subseteq S_q$ with $|A| \leq |S_{q'}|$
2: $S_\triangleright := \emptyset$
3: $S' := A$, where $A \subseteq S_\triangledown$
4: $V := \{y_1, \ldots, y_m\} \subset \mathsf{Var}$, where $m \in [1, 2 \cdot |S_{q'}| \cdot \max_{R \in \mathbf{S}}\{\mathrm{ar}(R)\}]$ and $\mathrm{Dom}(S_q) \cap \{y_1, \ldots, y_m\} = \emptyset$
5: **repeat**
6:     **repeat**
7:         **if** $\sigma = R[\alpha] \subseteq P[\beta] \in \Sigma$ is applicable on $S_\triangledown$ with $\bar{u} \in \mathrm{Dom}(S_\triangledown)^{\mathrm{ar}(R)}$ **then**
8:             $N := \mathsf{new}^V(\sigma, \bar{u})$
9:             $V := V - \mathrm{Dom}(\{N\})$
10:            $S_\triangleright := S_\triangleright \cup \{N\}$
11:        **if** $|S_\triangleright| < |S_{q'}|$ **then**
12:            $Next := b$, where $b \in \{0, 1\}$
13:        **else**
14:            $Next := 1$
15:    **until** $Next = 1$
16:    **if** $S_\triangleright = \emptyset$ **then**
17:        **return** no
18:    $V := V \cup ((\mathrm{Dom}(S_\triangledown) \cap \mathsf{Var}) - (\mathrm{Dom}(S_\triangleright) \cup \mathrm{Dom}(S')))$
19:    $S_\triangledown := S_\triangleright$
20:    $S_\triangleright := \emptyset$
21:    $S' := S' \cup A$, where $A \subseteq S_\triangledown$
22:    **if** $|S'| < |S_{q'}|$ **then**
23:        $Evaluate := b$, where $b \in \{0, 1\}$
24:    **else**
25:        $Evaluate := 1$
26: **until** $Evaluate = 1$
27: **if** $\mathsf{G}_{S'}(\bar{x}) \in q'(\mathsf{G}_{S'}(S'))$ **then**
28:     **return** yes
29: **else**
30:     **return** no
```

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 S' (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 S' (see line 21); this step is part of the "on the fly" construction of S'. The above is repeated (the outer repeat-until loop) until S' 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 yes if $\mathsf{G}_{S'}(\bar{x}) \in q'(\mathsf{G}_{S'}(S'))$; otherwise, it returns no.

It is not difficult to verify that Algorithm 4 uses polynomial space, that is, the space needed to store the sets $S_\triangledown$, $S_\triangleright$, S' 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 $\mathsf{G}_{S'}(\bar{x}) \in q'(\mathsf{G}_{S'}(S'))$ (see line 27). This shows that CQ-Containment-IND is in NPSPACE, and thus in PSPACE since, by Savitch's theorem, NPSPACE = PSPACE.

The PSPACE-hardness of CQ-Containment-IND is shown via a reduction from IND-Implication, which is PSPACE-hard (see Theorem 11.5). 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

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

while the CQ q' is defined as

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

where, for each $m \in [1, \mathrm{ar}(P)]$,

$$f(m) \ = \ \begin{cases} i_\ell & \text{if } m = j_\ell, \text{ where } \ell \in [1,k], \\ \\ \mathrm{ar}(R) + m & \text{otherwise.} \end{cases}$$

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 $q \subseteq_\Sigma q'$.

Comments and Exercises for Part II

Exercises

1. Let q be the CQ (12.3) defined in Chapter 12. Show how q can be expressed in RA by using θ-joins instead of Cartesian products.
2. Prove the correctness of the translation of CQs into SPJ queries.
3. Prove the correctness of the translation of SPJ queries into CQs.
4. If the size of a CQ q, written as a rule, is $|q|$, what is the size of its translation into an SPJ query? And what about the converse: if we know the size of an SPJ query, what is the size of an equivalent CQ?
5. 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 $D \otimes_{\tau'} D' \models q$.
6. The goal of this exercise is to extend preservation under direct product to queries with constants. For the former, we adjust 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, and for a Boolean CQ q over that schema that only mentions 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.
7. The goal of this exercise is to extend the result of Exercise 6 to queries with free variables. Let τ_C be a pairing function defined as in Exercise 6. Then given two tuples $\bar{a} = (a_1, \ldots, a_n)$ and $\bar{b} = (b_1, \ldots, b_n)$, define an n-ary tuple $\bar{a} \otimes \bar{b}$ as $\big(\tau_C(a_1, b_1), \ldots, \tau_C(a_n, b_n)\big)$. Let $q(x_1, \ldots, x_n)$ be a CQ that only mentions constants from C. Prove 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.

8. Use Exercise 6 to prove that Boolean query $\exists x\,(x = a)$, where a is a constant, cannot be expressed as a CQ.
9. Use Exercise 7 to prove that query $\varphi(x, y) = (x = y)$ cannot be expressed as a CQ.
10. Given parameterized languages (L_1, κ_1), (L_2, κ_2) over the same alphabet Σ, a function $\Phi : \Sigma^* \to \Sigma^*$ is said to be an FPT-reduction from (L_1, κ_1) to (L_2, κ_2) if there exist computable functions $f, g : \mathbb{N} \to \mathbb{R}_0^+$ and a polynomial p such that for every $w \in \Sigma^*$:

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

    Prove that if there exists an FPT-reduction from (L_1, κ_1) to (L_2, κ_2) and $(L_2, \kappa_2) \in \mathrm{FPT}$, then $(L_1, \kappa_1) \in \mathrm{FPT}$.
11. Let Φ be the reduction from Clique to CQ-Evaluation given in the proof of Theorem 14.1. Show that Φ is also an FPT-reduction from p-Clique to p-CQ-Evaluation.
12. Prove that p-CQ-Evaluation is in W[1].
13. Consider the relations $\equiv$ and *being homomorphically equivalent* defined in Chapter 15. Prove that both are equivalence relations on the set of CQs.
14. Answer the following questions about CQs and their cores.

    a) Consider the following Boolean CQ over a schema consisting of a single binary relation E:

    $$q_1 :- E(x_1, y_1), E(y_1, z_1), E(z_1, w_1), E(w_1, x_1), E(x_2, y_2), E(y_2, x_2).$$

    If E is used to represent the edge relation of a graph, what is this Boolean CQ checking over this graph? Construct the core of q_1.

    b) Consider the following Boolean CQ over a schema consisting of two unary relations R, S:

    $$q_2 :- R(x), S(x), R(y), S(y).$$

    Construct the core of q_2.

    c) Consider the following CQ over a schema consisting of two unary relations R, S:

    $$q_3(x, y) :- R(x), S(x), R(y), S(y).$$

    Prove that q_3 is its own core.
15. Prove Theorem 15.10. That is, show that if $q(\bar{x})$ is a CQ and $q'(\bar{x})$ is the core of $q(\bar{x})$, then $q'(\bar{x})$ is a minimization of $q(\bar{x})$, and each minimization of $q(\bar{x})$ is isomorphic to $q'(\bar{x})$.

16. Let $q(\bar{x})$ be a CQ and $q'(\bar{x})$ be the core of $q(\bar{x})$. Prove that there exists a homomorphism from $(q, \bar{x})$ to $(q', \bar{x})$ that is the identity on $\mathrm{Dom}(S_{q'})$.
17. Prove Lemma 10.3.
18. Let D be a database and $S = \{\bar{a}_1, \ldots, \bar{a}_n\}$ a set of m-ary tuples over $\mathrm{Dom}(D)$, for $m > 0$. Show that there exists a CQ $q(\bar{x})$ such that $q(D) = S$ iff the following hold:

    a) $\prod_{1 \leq i \leq n} \bar{a}_i$ appears in $\prod_{1 \leq i \leq n} D$, and
    b) there is no tuple $\bar{b} \in \mathrm{Dom}(D)^m - S$ such that $\prod_{1 \leq i \leq n}(D, \bar{a}_i) \to (D, \bar{b})$.
19. Prove that the problem of checking whether there exists a CQ $q(\bar{x})$ such that $q(D) = S$, given a database D and a set $S = \{\bar{a}_1, \ldots, \bar{a}_n\}$ of m-ary tuples over $\mathrm{Dom}(D)$, for $m > 0$, is coNEXP-complete.
20. Prove that the problem of checking containment of an FO formula in a CQ, and the converse problem of checking containment of a CQ in an FO formula, are undecidable.

Exercise 16.1. Prove Lemma 16.10.

Andreas:
Further details will be given concerning the proof strategy.

Part III

Fast Conjunctive Query Evaluation

Motivation

Here we are interested in understanding when CQ evaluation can be solved efficiently in combined complexity. Let us start by recalling the proof that CQ evaluation is NP-complete given in Theorem 14.1: The reduction constructs from a graph G both a database D and a boolean CQ q such that:

$$G \text{ is 3-colorable} \quad \Longleftrightarrow \quad q(D) = \texttt{true}.$$

While the database D is very simple, it consists of three elements only, the CQ q is arbitrarily involved, as it corresponds to a direct encoding of the input graph G over a binary schema.

It is known, on the other hand, that several NP-complete problems over graphs, including 3-colorability, become tractable if graphs are restricted to be *nearly-acyclic*. As we show in this part of the book, similar ideas can be applied to prove that CQ evaluation is tractable when CQs are nearly-acyclic. This is relevant from a practical point of view, as such CQs appear often in real-world applications.

17 Acyclicity

We start by studying the notion of *acyclicity* for CQs, which has received considerable attention in the database literature since the early 80s. We will present two algorithms that show that acyclic CQs can be evaluated efficiently. The first one, known as Yannakakis's algorithm, makes use of a decomposition of an acyclic CQ known as *join tree*, while the second one is based on a simple *consistency* criterion. Yannakakis's algorithm achieves an optimal running time of $O(\|D\| \cdot \|q\|)$, i.e., it is linear on both the size of the database and the CQ. On the other hand, the algorithm based on the consistency criterion has the advantage that it does not require the CQ itself to be acyclic, only its core (as defined in Chapter 15).

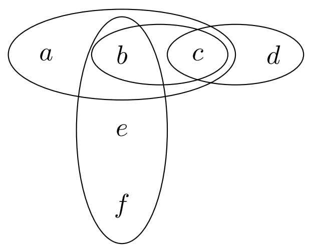
The notion of acyclicity

Defining the notion of acyclicity for CQs is not as simple as it is for graphs. This is because CQs are expressed over schemas of arbitrary arity, and, therefore, its underlying structure is more faithfully represented as a *hypergraph*. Unfortunately, for hypergraphs acyclicity is not a self-evident notion, and, in fact, several natural, non-equivalent notions of acyclicity for hypergraphs exist. We work here with one such a notion, often referred to as α-*acyclicity*, which has received considerable attention in database theory.

Recall that a hypergraph corresponds to a pair $H = (V, E)$, where V is a finite set of vertices and E corresponds to a set of *hyperedges* over V, i.e., nonempty subsets of V. Then H is α-*acyclic*, or simply *acyclic* from now on, if it admits a *join tree*. This means that its hyperedges can be arranged in the form of a tree, while preserving the connectivity of elements that occur in different hyperedges. Formally, a join tree of $H = (V, E)$ is a tree T such that:

- The nodes of T are precisely the hyperedges in E.
- For each node $v \in V$, it is the case that the set of nodes of T in which v is mentioned defines a connected subtree of T.

Example 17.1. Consider the following hypergraph $H_1 = (V_1, E_1)$:



Thus, we have that $V_1 = \{a, b, c, d, e, f\}$ and $E_1 = \{\{a, b, c\}, \{b, c\}, \{c, d\}, \{b, e, f\}\}$. It holds that H_1 is an acyclic hypergraph, as the following tree T_1 is a join tree for H_1:

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

In fact, we have that T_1 is a join tree of H_1 as the nodes of T_1 are precisely the hyperedges in E_1, and for each $v \in V_1$, the set of nodes of T_1 in which v occurs defines a connected subtree of T_1. As an example of this latter condition, if we consider $v = c$, then we obtained the following subtree of T_1 that is connected:

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

On the other hand, consider a hypergraph H_2 that extends H_1 with the hyperedge $\{c, e\}$. We have that H_2 is not acyclic, as it is not possible to construct a join tree for it. For instance, consider the extension T_2 of T_1 that is obtained by adding a node $\{c, e\}$ and connecting it with the node $\{a, b, c\}$ of T_1:

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

Then we have that T_2 is not a join tree for H_2 as the set of nodes of T_2 in which e occurs do not define a connected subtree of T_2:

$$\{b, e, f\} \qquad \{c, e\}$$

□

It is not hard to see that for undirected graphs, the notion of α-acyclicity coincides with the usual notion of acyclicity that stems from graph theory (i.e., tree-shaped graphs).

The notion of acyclic hypergraph is the key concept in the definition of acyclic CQs. Each CQ q is naturally associated with a hypergraph H_q that represents the structure of joins among its variables. In particular, if q is of the form:

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

then $H_q = (V, E)$ is the hypergraph such that:

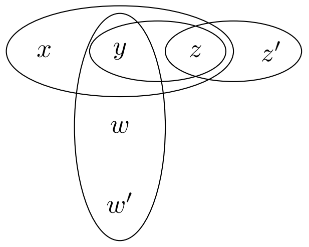
- Its set V of vertices contains all variables mentioned in q.
- The hyperedges in E are precisely the sets of variables mentioned in the atoms of q; i.e., $E = \{X_i \mid i \in [1, n]\}$, where X_i is the set of variables occurring in $\bar{x}_i$.

Definition 17.2 (Acyclicity of CQs). *A CQ q is acyclic if and only if its associated hypergraph H_q is acyclic (that is, it has a join tree). We denote by* AC *the class of acyclic CQs.*

Example 17.3. Consider the following CQ q_1:

$$q_1(x, y) :- R(x, y, z), T(y, z), S(y, w, w'), T(z, z').$$

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



We know from Example 17.1 that H_{q_1} is an acyclic hypergraph, as H_{q_1} can be obtained from the hypergraph H_1 in Example 17.1 by renaming the nodes of H_1. Therefore, we have that q_1 is an acyclic CQ. On the other hand, the following CQ q_2

$$q_2(x, y) :- R(x, y, z), T(y, z), S(y, w, w'), T(z, z'), T(z, w)$$

is not acyclic as the hypergraph H_{q_2} is not acyclic. Notice that this latter fact is also obtained from Example 17.1, as H_{q_2} can be obtained from the hypergraph H_2 in Example 17.1 by renaming the nodes of H_2. Finally, consider the following CQ q_3:

$$q_3(x, y) :- R(x, y, z), R(y, y, z), T(y, z), S(y, w, w'), T(z, z').$$

Then we have that q_3 is an acyclic CQ since $H_{q_3} = H_{q_1}$. Notice that this latter condition holds as the set of variable occurring in $R(y, y, z)$ is $\{y, z\}$, which is the same as the set of variables occurring in $T(y, z)$. $\square$

Acyclicity recognition

We will show that CQs in AC can be evaluated efficiently. But before doing so, it is important to explain why acyclicity itself can be efficiently recognized. This follows from the existence of an equivalent definition of acyclicity in terms of an iterative process described in the following proposition.

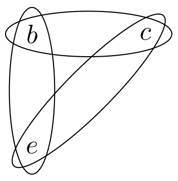
Proposition 17.4. *A hypergraph $H = (V, E)$ is acyclic if and only if all of its vertices can be deleted by repeatedly applying the following two operations (in no particular order):*

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

This characterization leads directly to a polynomial-time algorithm for checking acyclicity of hypergraphs, and thus of CQs: Given a CQ q, we apply operations (1) and (2) in the statement of Proposition 17.4 over H_q until a fixpoint is reached. If we are left with no vertices, then q is acyclic. Interestingly, a simple extension of this algorithm also constructs a join tree of H_q when q is acyclic (see Exercise 1). It is easy to see that the algorithm runs in quadratic time in the size of H_q, and, therefore, in the size of the CQ q.

Example 17.5. Consider the hypergraph H_1 in Example 17.1. As expected, by using the previous algorithm we obtain that H_1 is acyclic. In fact, all vertices of H_1 are deleted by applying the following sequence of operations: delete vertex d (that appears only in hyperedge $\{c, d\}$), delete vertices e and f, delete hyperedges $\{b\}$ and $\{c\}$ (that are contained in hyperedge $\{b, c\}$), delete hyperedge $\{b, c\}$ (that is contained in hyperedge $\{a, b, c\}$), and delete vertices a, b and c.

On the other hand, and also as expected, by applying the previous algorithm on hypergraph H_2 from Example 17.1, we obtain that H_2 is not acyclic. In fact, no matter what order is used when applying the two operations of the algorithm, we reach the following fixed point:



Notice that no operation can be applied to reduce this hypergraph, which is intuitively correct as this hypergraph represents the canonical example of an undirected graph that is not acyclic. □

It is important to mention that there are more sophisticated algorithms that check whether a CQ q is acyclic, and construct a join tree of H_q if the latter is the case, in linear time. We will exploit this fact later in the presentation of Yannakakis's algorithm.

Semijoins and acyclic CQs

The evaluation of acyclic CQs is tightly related to a particular relational algebra operation, known as *semijoin*, which we describe next. Let D be a database. Given CQs $q(\bar{x})$ and $q'(\bar{x}')$ and tuples $\bar{a} \in q(D)$ and $\bar{b} \in q'(D)$, we call $\bar{a}$ and $\bar{b}$ *consistent* if they have the same value on each position that corresponds to a variable shared by $\bar{x}$ and $\bar{x}'$. We then define the semijoin of $q(D)$ and $q'(D)$, denoted by $q(D) \ltimes q'(D)$, as the set of tuples $\bar{a} \in q(D)$ that are consistent with some tuple $\bar{b} \in q'(D)$.

Example 17.6. Assume that $D = \{R(a,b,c), R(d,d,d), S(c,b,e), S(d,e,e)\}$, and that q and q' are the following CQs:

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

Then we have that $q(D) = \{(a,b,c),(d,d,d)\}$, $q'(D) = \{(c,b,e),(d,e,e)\}$, and $q(D) \ltimes q'(D) = \{(a,b,c)\}$. In particular, we have that (a,b,c) is in $q(D) \ltimes q'(D)$ since (a,b,c) belongs to $q(D)$ and (a,b,c) is consistent with the tuple $(c,b,e) \in q'(D)$, as these two tuples have the same value b in the position that corresponds to the variable y shared by (x,y,z) and (z,y,w), and have the same value c in the position that corresponds to the variable z shared by (x,y,z) and (z,y,w). Moreover, we have that (d,d,d) is not in $q(D) \ltimes q'(D)$ as this tuple is not consistent with any tuple in $q'(D)$. Finally, notice that $q'(D) \ltimes q(D) = \{(c,b,e)\}$, which shows that, as opposed to the case of the join operator, $\ltimes$ is not a commutative operator. □

We now explain the relationship between the CQs in AC and the semijoin operator. Let $q \ {:}{-}\ R_1(\bar{x}_1), \dots, R_n(\bar{x}_n)$ be a Boolean CQ in AC, and consider an arbitrary join tree T of H_q. Recall that the set of nodes of T is $\{X_i \mid i \in [1,n]\}$, where each X_i is the set of variables occurring in $\bar{x}_i$. We see T as a rooted and directed tree by arbitrarily fixing a root r. Therefore, we can naturally talk about the *leaves* of T and about the *children* of a node in T. For every node s of T, we define the following CQs, assuming that s is associated with the set X_i of variables, for some $i \in [1,n]$, and assuming that $\bar{y}_i$ is a tuple of pairwise distinct variables consisting exactly of the variables in X_i:

- A CQ $q_s(\bar{y}_i) \ {:}{-}\ R_{j_1}(\bar{x}_{j_1}), \dots, R_{j_p}(\bar{x}_{j_p})$, where $\{R_{j_1}(\bar{x}_{j_1}), \dots, R_{j_p}(\bar{x}_{j_p})\}$ is the set of of atoms from q such that $X_{j_\ell} = X_i$ for each $\ell \in [1,p]$.
- A CQ $Q_s(\bar{y}_i)$ whose atoms are precisely those that appear in the CQs of the form $q_{s'}$, for s' a descendant of s in T (including s itself).

Notice that $Q_s \subseteq q_s$, for each node s of T. Moreover, if s is a (non-leaf) node of T with children $s_1, \dots, s_p$, then the atoms of Q_s correspond to the union of the atoms in Q_{s_1}, …, Q_{s_p}, plus the atoms from q_s.

In what follows, we present a fundamental connection between the evaluation of acyclic CQs and the semijoin operator. To understand this connection, assume that $q :- R_1(\bar{x}_1), \dots, R_n(\bar{x}_n)$ is a Boolean CQ in AC, and suppose that T is a rooted and directed join tree of H_q with root r, where r is associated with the set of variables X_ℓ, for some $\ell \in [1, n]$. Then we have that Q_r is a CQ of the form $Q_r(\bar{y}_\ell) :- R_1(\bar{x}_1), \dots, R_n(\bar{x}_n)$; in particular, Q_r has the same body as q. Thus, we have that for every database D, it holds that $q(D) = \texttt{true}$ if and only if $Q_r(D) \neq \emptyset$, and, therefore, an efficient algorithm for the evaluation of Q_r can also be used to evaluate q. As shown in the next section, the following proposition gives us such an algorithm, as it tell us that Q_r can be inductively evaluated by computing semijoins while traversing T in a bottom-up manner, provided that for every node s of T, CQ q_s has been previously evaluated.

Proposition 17.7. *Let q be a Boolean CQ in AC, T a rooted and directed join tree of H_q and D a database. Then for every node s of T, it holds that:*

- *If s is a leaf of T, then $Q_s(D) = q_s(D)$.*
- *Otherwise, if the children of s in T are $s_1, \dots, s_p$, then:*

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

Proof. If s is a leaf of T, then q_s and Q_s are the same CQ and, thus, $Q_s(D) = q_s(D)$. Let us assume then that s is a non-leaf node of T with children $s_1, \dots, s_p$. Moreover, assume that nodes s, s_1, …, s_p are associated with the set of variables X_ℓ, $X_{k_1}, \dots, X_{k_p}$, respectively, where $\ell, k_1, \dots, k_p$ are pairwise distinct numbers in the set $\{1, \dots, n\}$. Then we have that $\bar{y}_\ell$, $\bar{y}_{k_1}$, …, $\bar{y}_{k_p}$ are the tuples of free variables of CQs Q_s, Q_{s_1}, …, Q_{s_p}, respectively. Let us consider first an arbitrary tuple in $Q_s(D)$. By definition, such a tuple is of the form $h(\bar{y}_\ell)$ for some homomorphism h from Q_s to D. It is not hard to see that $h(\bar{y}_\ell) \in q_s(D) \ltimes Q_{s_i}(D)$, for every $i \in [1, p]$. In fact, $h(\bar{y}_\ell) \in q_s(D)$ since $Q_s \subseteq q_s$, and $h(\bar{y}_{k_i}) \in Q_{s_i}(D)$ since the atoms of Q_{s_i} are contained in those of Q_s. Moreover, $h(\bar{y}_\ell)$ and $h(\bar{y}_{k_i})$ are consistent by definition. We conclude that $h(\bar{y}_\ell) \in \bigcap_{i=1}^{p} \big(q_s(D) \ltimes Q_{s_i}(D)\big)$.

Let us consider now an arbitrary tuple in $\bigcap_{i=1}^{p} \big(q_s(D) \ltimes Q_{s_i}(D)\big)$. By definition, such a tuple is of the form $h(\bar{y}_\ell)$ for some homomorphism h from q_s to D. Moreover, for each $i \in [1, p]$, there is a homomorphism h_i from Q_{s_i} to D such that $h(\bar{y}_\ell)$ and $h_i(\bar{y}_{k_i})$ are consistent; i.e., they have the same value on positions that represent shared variables. We claim that $h' = h \cup h_1 \cup \dots \cup h_p$ defines a homomorphism from Q_s to D. Since $h'(\bar{y}_\ell) = h(\bar{y}_\ell)$, this shows that $h(\bar{y}_\ell) \in Q_s(D)$ as desired.

We first show that h' is well defined. Take an arbitrary variable y in Q_s. If y is mentioned only in q_s but not in any of the CQs Q_{s_i} (for $i \in [1, p]$), or if y is mentioned only in one of the CQs Q_{s_i} (for $i \in [1, p]$) but not in q_s, then clearly $h'(y)$ is well defined. There are two other possibilities: y is mentioned in q_s and in Q_{s_i}, for $i \in [1,p]$, or y is mentioned in Q_{s_i} and Q_{s_j}, for $i, j \in [1,p]$ with $i \neq j$. We only consider the latter case since the former can be handled analogously. By definition of the notion of join tree, the occurrences of y in T are connected. Hence, y is mentioned in s, s_i, and s_j. Therefore, we conclude that $h_i(y) = h_j(y) = h(y)$ since $h(\bar{y}_\ell)$ is consistent with both $h_i(\bar{y}_{k_i})$ and $h_j(\bar{y}_{k_j})$.

We now prove that h' is a homomorphism from Q_s to D. Take an arbitrary atom $R(\bar{z})$ in Q_s. Then, $R(h'(\bar{z})) = R(h(\bar{z}))$ or $R(h'(\bar{z})) = R(h_i(\bar{z}))$ for some $i \in [1,p]$. Thus, $R(h'(\bar{z})) \in D$ because h and h_i are homomorphisms. This concludes the proof of the proposition. □

17.1 Yannakakis's algorithm

Yannakakis's algorithm uses the conditions in Proposition 17.7 to evaluate a Boolean acyclic CQ, as shown in Algorithm 5. The soundness and completeness of the algorithm follows from Proposition 17.7 – which justifies the correctness of the inductive computation carried out in while loop – and the fact that the atoms of Q_r are precisely those of q, from which we conclude that $Q_r(D) \neq \emptyset$ if and only if $q(D) = \texttt{true}$.

Algorithm 5 YANNAKAKIS(q, D)

```
Input: A Boolean CQ $q$ in AC and a database $D$.
Output: If $q(D) = \texttt{true}$, then 1. Otherwise 0.
 1: Compute a rooted and directed join tree $T$ of $H_q$, and assume that $N$ is the set
    of nodes of $T$ and $r$ is the root of $T$
 2: while $N \neq \emptyset$ do
 3:     Choose $s \in N$ such that no children of $s$ is in $N$
 4:     Compute $q_s(D)$
 5:     if $s$ is a leaf of $T$ then
 6:         $Q_s(D) := q_s(D)$
 7:     else
 8:         Let $s_1, \dots, s_p$ be the children of $s$ in $T$
 9:         $Q_s(D) := \bigcap_{i=1}^{p} \big(q_s(D) \ltimes Q_{s_i}(D)\big)$
10:     $N := N - \{s\}$
11: if $Q_r(D) \neq \emptyset$ then return 1
12: elsereturn 0
```

We now analyze the complexity of the algorithm. From our previous remarks, a join tree T of H_q can be computed in time $O(\|H_q\|) = O(\|q\|)$. We show next that the second step can be implemented in time $O(\|D\| \cdot \|q\|)$. To see why this is the case, we need the following observation: the cost of computing $q(D) \ltimes q'(D)$, given $q(D)$ and $q'(D)$, is $O(\|q(D)\| + \|q'(D)\|)$. In particular, then, each $q_s(D)$s for s a node in T, can be computed in time $O(||D|| \cdot ||q_s||)$. Therefore, the collection of all $q_s(D)$s, for s a node in T, can be computed in time $O(||D|| \cdot ||q^{\bar{t}}||) = O(||D|| \cdot ||q||)$. This is because, by definition, each atom of $q^{\bar{t}}$ appears in exactly one q_s.

Now, if s is a node of T with children $s_1, \dots, s_p$, we can compute $Q_s(D) = \bigcap_{1 \leq i \leq p} q_s(D) \ltimes Q_{s_i}(D)$ in time $O(||D|| \cdot p)$. This follows from the fact that $||q_s(D)|| \leq ||D||$ and $||Q_{s_i}(D)|| \leq ||q_{s_i}(D)|| \leq ||D||$, for each $1 \leq i \leq p$. Therefore, we can inductively compute the collection of all Q_s's, for s a node in T, in time $O(||D|| \cdot ||T||) = O(||D|| \cdot ||q||)$.

In summary, we obtain the following result:

Theorem 17.8. AC-Evaluation *can be solved in time* $O(||D|| \cdot ||q||)$.

Proof. We already proved that the theorem holds for the fragment of AC consisting of Boolean queries.

17.2 The consistency algorithm

While Yannakakis's algorithm uses a join tree of an acyclic CQ q in order to evaluate q over a database D in linear time, if we only aim for tractability then there is no need for such a join tree to be computed. In fact, below we present an algorithm that evaluates q on D in polynomial time, only by holding the *promise* that q is acyclic (i.e., that a join tree of q exists). The design of such an algorithm is based on a simple consistency criterion, established in the following proposition, which characterizes when $q(D) = \texttt{true}$ for a Boolean CQ $q \in$ AC and a database D:

Proposition 17.9. *Let* q :- $R_1(\bar{u}_1), \dots, R_n(\bar{u}_n)$ *be a boolean CQ in* AC *and* D *a database. Let us define* $q_i(\bar{x}_i)$:- $R_i(\bar{u}_i)$*, for each* $1 \leq i \leq n$*, where* $\bar{x}_i$ *is the tuple that is obtained from* $\bar{u}_i$ *by removing constants. Then the following are equivalent:*

1. $q(D) = \texttt{true}$.
2. *There are nonempty sets* $\text{Cons}(q_1(D)), \dots, \text{Cons}(q_n(D))$ *such that, for each* $1 \leq i \leq n$ *it is the case that* $\text{Cons}(q_i(D)) \subseteq q_i(D)$ *and:*

$$\text{Cons}(q_i(D)) \ = \ \text{Cons}(q_i(D)) \ltimes \text{Cons}(q_j(D)), \quad \textit{for each } 1 \leq j \leq n.$$

That is, each tuple in $\text{Cons}(q_i(D))$ *is consistent with some tuple in* $\text{Cons}(q_j(D))$*, for each* $1 \leq i, j \leq n$.

Proof. Suppose first that $q(D) = \texttt{true}$; i.e., there is a homomorphism h from q to D. It is not hard to see then that we can choose $\text{Cons}(q_i(D))$ to be

$\{h(\bar{x}_i)\}$, for each $1 \leq i \leq n$. Suppose, on the other hand, that nonempty sets $\text{Cons}(q_1(D)), \ldots, \text{Cons}(q_n(D))$ as described in (2) exist. Let T be an arbitrary rooted and directed join tree of q. One can then prove by induction the following for each node s of T (see Exercise 4): If s corresponds to the set X_i of variables mentioned in $\bar{x}_i$, for $1 \leq i \leq n$, then $\text{Cons}(q_i(D)) \subseteq Q_s(D)$. In particular, if the root r of T corresponds to the set X_j of variables mentioned in $\bar{x}_j$, for $1 \leq j \leq n$, then $\text{Cons}(q_j(D)) \subseteq Q_r(D)$. Therefore, since $\text{Cons}(q_j(D))$ is nonempty we conclude that $Q_r(D)$ is also nonempty. This implies that there is at least one homomorphism from Q_r to D. But the atoms of Q_r and q are the same by definition, and thus $q(D) = \texttt{true}$. □

We are ready to present our consistency algorithm, which is nothing but a greatest fixed-point computation checking for the existence of nonempty sets $\text{Cons}(q_1(D)), \ldots, \text{Cons}(q_n(D))$ as described in item (2) of Proposition 17.9:

Algorithm 6 $\text{C}\textsc{onsistency}(q, D)$

Input: A Boolean CQ q :- $R_1(\bar{u}_1), \ldots, R_n(\bar{u}_n)$ in AC and a database D.
Output: If $q(D) = \texttt{true}$, then $\text{Cons}(q_i(D)) \neq \emptyset$ for each $1 \leq i \leq n$. Otherwise, `fail`.

```
1: $\text{Cons}(q_i(D)) := q_i(D)$, for each $1 \leq i \leq n$
2: **while** $\text{Cons}(q_i(D)) \neq \text{Cons}(q_i(D)) \ltimes \text{Cons}(q_j(D))$ for $1 \leq i, j \leq n$ **do**
3:     $\text{Cons}(q_i(D)) := \text{Cons}(q_i(D)) \ltimes \text{Cons}(q_j(D))$
4: **if** $\text{Cons}(q_i(D)) = \emptyset$ for some $1 \leq i \leq n$ **then**
5:     fail
```

That is, the algorithm initializes $\text{Cons}(q_i(D))$ to be $q_i(D)$, for each $1 \leq i \leq n$. Starting from this, it iteratively deletes every tuple in $\text{Cons}(q_i(D))$ that is not consistent with a tuple in $\text{Cons}(q_j(D))$, for some $1 \leq j \leq n$. If after a fixed-point is reached some $\text{Cons}(q_i(D))$ is empty, the algorithm declares $q(D) = \texttt{false}$. Otherwise, $q(D) = \texttt{true}$. It is not hard to see that the algorithm runs in polynomial time (but not in linear time as Yannakakis's algorithm).

We establish next the soundness and completeness of the consistency algorithm:

Proposition 17.10. *Let q be a boolean CQ in AC and D a database. Then:*

$$q(D) = \texttt{true} \iff \text{C}\textsc{onsistency}(q, D) \neq \texttt{fail}.$$

We leave the proof of Proposition 17.10 as an exercise for the reader (see Exercise 4).

Acyclicity of the core

It is known that there are boolean CQs that are not acyclic, yet its core is acyclic. (The reader is asked to prove this fact in Exercise 5). Interestingly, the consistency algorithm continues being sound and complete in terms of the evaluation problem for the class of such CQs. That is:

Proposition 17.11. *Let q be a boolean CQ whose core is acyclic and D a database. Then:*

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

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

$$\begin{aligned} q(D) = \texttt{true} &\Longleftrightarrow q'(D) = \texttt{true} \\ &\Longleftrightarrow \textsc{Consistency}(q', D) \neq \texttt{fail} \\ &\Longleftrightarrow \textsc{Consistency}(q, D) \neq \texttt{fail}. \end{aligned}$$

The first equivalence holds since $q \equiv q'$, the second one since q' is acyclic (based on Proposition 17.10), and the last one since the fact that there is a homomorphism from q to q' and $\textsc{Consistency}(q', D)$ does not fail implies that $\textsc{Consistency}(q, D)$ does not fail (see Exercise 7). □

As a corollary, we then obtain the following:

Theorem 17.12. *The evaluation problem for the class of CQs whose core is acyclic can be solved in polynomial time.*

18

Generalized Hypertreewidth

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

The notion of generalized hypertreewidth

The definition of generalized hypertreewidth is based on the important notions of *tree decompositions* and *generalized hypertree decompositions*. Let $H = (V, E)$ be a hypergraph. A *tree decomposition* of $H = (V, E)$ is a pair (T, χ), formed by a tree T and a mapping χ that assigns a subset of the nodes in V to each node $s \in T$, for which the following statements hold:

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

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

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

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

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

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

Definition 18.1 (Generalized hypertreewidth). *The generalized hypertreewidth of a CQ q corresponds to the generalized hypertreewidth of its associated hypergraph H_q.*

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

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

$$q(x, y) :\!-\; R(x, y, z), S(y, z), S(y, w, w'), T(z, z'), T(z, w),$$

which was introduced in Chapter 17. The CQ q is in $\mathsf{GHW}(2)$. This is witnessed by the generalized hypertree decomposition (T, χ, λ) such that T consists of two nodes r and t for which $\chi(r) = \{x, y, z, w, w'\}$, $\lambda(r) = \{\{x, y, z\}, \{y, w, w'\}\}$, $\chi(t) = \{z, z'\}$, and $\lambda(t) = \{\{z, z'\}\}$. On the other hand, $q(x, y)$ is not in $\mathsf{GHW}(1)$. This is because CQs in $\mathsf{GHW}(1)$ are acyclic (see Proposition 18.2 below) and we know from Chapter 17 that q is not acyclic.

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

Bounded generalized hypertreewidth and acyclicity

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

Proposition 18.2. $\mathsf{GHW}(1)$ *is the class of acyclic CQs.*

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

Tractable evaluation based on consistency

It is shown next that CQs of bounded generalized hypertreewidth can be evaluated in polynomial time by extending the consistency criterion for acyclic CQs developed in Proposition 17.9. For reasons explained in Chapter 17, we concentrate on Boolean CQs.

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

Proposition 18.3. *Fix $k \geq 1$. Let q :- $R_1(\bar{u}_1), \ldots, R_m(\bar{u}_m)$ be a Boolean CQ in $\mathsf{GHW}(k)$ and D a database. Then the following are equivalent:*

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

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

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

Proof. Suppose first that $q(D) = \texttt{true}$; i.e., there is a homomorphism h from q to D. It is not hard to see then that one can choose $\text{Cons}(q_S(D))$ to be $\{h(\bar{x}_S)\}$, for each $S \subseteq \{1, \ldots, m\}$ with at most k elements. Suppose, on the other hand, that for each $S \subseteq \{1, \ldots, m\}$ with at most k elements a nonempty set $\text{Cons}(q_S(D))$ as described in (2) exists. Let (T, χ, λ) be an arbitrary rooted and directed generalized hypertree decomposition of q of width k. This implies that $\text{Cons}(q_{\lambda(t)}(D))$ is well-defined for each node $s \in T$, as $|\lambda(s)| \leq k$ by definition. One can then prove by induction the following for each node $s \in T$ (see Exercise 10): Let X_s be the set of variables mentioned in sets of the form $\chi(s')$, for s' a descendant of s in T (including s itself) and $S(X_s)$ the set of all atoms in q that only mention variables in X_s. Then $\text{Cons}(q_{\lambda(s)}(D)) \subseteq q'_{S(X_s)}(D)$, where $q'_{S(X_s)}$ is the CQ that has the same set of atoms than $q_{S(X_s)}$ but its tuple of free variables is $\bar{x}_{\lambda(s)}$. In particular, for the root r of T it is the case that $\text{Cons}(q_{\lambda(r)}(D)) \subseteq q'_{S(X_r)}(D)$. Therefore, since $\text{Cons}(q_{\lambda(r)}(D))$ is nonempty one can conclude that $q'_{S(X_r)}(D)$ is also nonempty. This implies that there is at least one homomorphism from $q'_{S(X_r)}$ to D. But the atoms of $q'_{S(X_r)}$ and q are the same by definition, and thus $q(D) = \texttt{true}$. □

As for acyclic CQs, this result allows us to construct a simple greatest-fixed point procedure that checks for the existence of sets $\text{Cons}(q_S(D))$ as described in item (2) of Proposition 18.3. The algorithm, called k-Consistency, is presented next:

Algorithm 7 k-Consistency(q, D)

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

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

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

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

Proposition 18.4. *Let q be a Boolean CQ in $\mathsf{GHW}(k)$ and D a database. Then:*

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

The proof of Proposition 17.10 is left as an exercise for the reader (see Exercise 10). As a corollary, one obtains the following:

Theorem 18.5. *Fix $k \geq 1$. $\mathsf{GHW}(k)$-$\mathsf{Evaluation}$ can be solved in polynomial time $O(||D||^{2k} \cdot ||q||^{2k})$.*

Bounded generalized hypertreewidth of the core

Recall that the Consistency procedure for evaluating acyclic CQs, presented in Chapter 17, continues being sound for the class of CQs whose core is acyclic. The k-Consistency algorithm presented above preserves this good behavior, this time with respect to the class of CQs whose core is in $\mathsf{GHW}(k)$:

Proposition 18.6. *Fix $k \geq 1$. Let q be a Boolean CQ whose core is in $\mathsf{GHW}(k)$ and D a database. Then:*

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

The proof of this result is similar to the proof of Proposition 17.11. As a corollary, one then obtains the following:

Theorem 18.7. *Fix $k \geq 1$. The evaluation problem for the class of CQs whose core is in $\mathsf{GHW}(k)$ can be solved in polynomial time.*

Computing generalized hypertree decompositions for faster evaluation

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

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

Theorem 18.8. *Fix $k > 1$. Assuming* PTIME $\neq$ NP*, there is no polynomial time algorithm that given an input CQ q computes a generalized hypertree decomposition of width k whenever $q \in \mathsf{HW}(k)$.*

Proof. The proof relies on the following difficult result:

Proposition 18.9. *Fix $k > 1$. The problem of checking if a given CQ is in $\mathsf{GHW}(k)$ is* NP*-complete.*

In fact, assume for the sake of contradiction that for some $k > 1$ there is a polynomial time algorithm $\mathcal{A}$ that given an input CQ q computes a generalized hypertree decomposition of width k whenever $q \in \mathsf{HW}(k)$. We show then that there is a polynomial time algorithm $\mathcal{A}'$ that checks whether a given CQ is in $\mathsf{GHW}(k)$, thus contradicting Proposition 18.9. Take an arbitrary CQ q and run $\mathcal{A}$ on q. The algorithm $\mathcal{A}'$ accepts iff $\mathcal{A}$ outputs a generalized hypertree decomposition of q of width k (the latter can be checked in polynomial time). It is easy to see, then, that $\mathcal{A}'$ accepts iff q is in $\mathsf{HW}(k)$. □

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

We can then establish the following:

Theorem 18.10. *Fix $k \geq 1$. Then $\mathsf{GHW}(k)$-$\mathsf{Evaluation}$ can be solved in time $2^{||q||^c} + O(||D||^k \cdot ||q||)$, for some integer $c \geq 1$.*

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

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

This takes time $O(||D||^k \cdot ||q||)$ since $||q_{\lambda(s)}||$ is $O(||D||^k)$, for each node $s \in T$, and T has at most n nodes. It can be proved then (see Exercise 11) that for each node $s \in T$ we have that:

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

where $q'_{S(X_s)}$ is as defined in the proof of Proposition 18.3. In particular, the atoms of $q'_{S(X_s)}$ are precisely the atoms of q that only mention variables that appear in the subtree of T rooted in s, and the free variables of $q'_{S(X_s)}$ are those that are mentioned in $\lambda(s)$. The algorithm then accepts if $Q_{\lambda(r)}(D) \neq \emptyset$, where r is the root of T. In fact, from the previous observation it is the case that $Q_{\lambda(r)}(D) = q'_{S(X_r)}(D)$. Moreover, clearly the atoms of $q'_{S(X_r)}$ and q are the same. It follows then that $Q_{\lambda(r)}(D) \neq \emptyset$ if and only if there is a homomorphism from q to D, i.e., $q(D) = \texttt{true}$. $\square$

The bound for evaluation of CQs in $\mathsf{GHW}(k)$ obtained in Theorem 18.10 is better, in many practical situations, than the one offered by the consistency algorithm. In fact, from Theorem 18.10 we obtain that the problem can be solved in time $2^{||q||^c} + O(||D||^k \cdot ||q||)$. This is better than the $O(||D||^{2k} \cdot ||q||^{2k})$ obtained by applying the k-Consistency algorithm whenever $2^{||q||^c}$ is $O(||D||^{2k})$. But this is not uncommon, as often the database D is very large and the CQ q is orders of magnitude smaller.

19

The Necessity of Bounded Treewidth

If a class $\mathcal{C}$ of CQs has bounded generalized hypertreewidth, then the evaluation problem for $\mathcal{C}$ can be solved in polynomial time. Moreover, this positive behavior continues to hold even if $\mathcal{C}$ itself does not have bounded generalized hypertreewidth, but the class $\mathcal{C}_{\text{core}}$ of all cores of CQs in $\mathcal{C}$ does. More formally, it follows from Proposition 18.6 that if $\mathcal{C}_{\text{core}}$ satisfies that there is a $k \geq 1$ such that every CQ $q \in \mathcal{C}_{\text{core}}$ is in $\mathsf{GHW}(k)$, then the evaluation problem for $\mathcal{C}$ can be solved in polynomial time.

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

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

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

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

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

CQs of bounded treewidth

Over schemas of fixed arity, the notion of bounded generalized hypertreewidth collases to the notion of bounded *treewidth*. The concept of treewidth has received vast attention in the context of graph theory, where it is used as a measure for the degree of cyclicity of a graph. In particular, the class of trees corresponds precisely to the class of undirected graphs of treewidth one, while the class of all k-cliques, for $k \geq 1$, has unbounded treewidth (in particular, the k-clique has treewidth $k-1$ for each $k > 1$). The notion of treewidth of a CQ corresponds to a natural generalization of the notion of treewidth of a graph to the case when schemas are arbitrary. The main result this section will be presented in terms of bounded treewidth, instead of bounded generalized hypertreewidth, as the former notion more naturally fits in the scenario in which schemas are of fixed arity.

Recall that a tree decomposition of a hypergraph $H = (V, E)$ is a pair (T, χ), formed by a tree T and a mapping χ that assigns a subset of the nodes in V to each node $s \in T$, for which the following statements hold:

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

The *width* of (T, χ) is $(\max_{s \in T} |\chi(s)|) - 1$. The treewidth of H is the minimum width of its tree decompositions. The treewidth of a CQ q is the treewidth of its associated hypergraph H_q. We denote by $\mathsf{TW}(k)$ the set of all CQs of treewidth at most k, for $k \geq 1$.

The treewidth of a CQ can equivalently defined as the treewidth of the graph that represents which variables occur together in some atom of q. Formally, let q be a CQ and G_q the *Gaifman graph* of q; this is the simple graph whose nodes are the variables of q and there is an edge between distinct variables x and y if and only if there is some atom in q that mentions both x and y. As stated next, q and G_q have the same treewidth (the proof of this result is left as an exercise):

Lemma 19.1. *Let q be a CQ. The treewidth of q is precisely the treewidth of its Gaifman graph G_q.*

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

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

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

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

Lemma 19.2. *Let q be a CQ defined over a schema all of whose relation symbols have arity at most c, for $c \geq 1$. Then for every $k \geq 1$:*

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

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

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

The main result

The main result of this section, which is the characterization of the tractable classes of CQs over fixed arity schemas in terms of the notion of bounded treewidth, is stated next. Only the version for boolean CQs is presented, but there is a simple extension of the result that characterizes tractability for arbitrary CQs (see Exercise 15):

Theorem 19.3. *Assume that* FPT $\neq$ W[1]. *Let $\mathcal{C}$ be a recursively enumerable class of boolean CQs such that there is a bound on the arity of the relation symbols that are mentioned by CQs in $\mathcal{C}$. Then the following are equivalent:*

1. $\mathcal{C}$-$\mathsf{Evaluation}$ *can be solved in polynomial time.*
2. $\mathcal{C}$-$\mathsf{Evaluation}$ *is* FPT.

3. $\mathcal{C}_{\text{core}}$ *has bounded treewidth.*

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

Overall idea behind the proof of Theorem 19.3

The implication from (1) to (2) is straightforward. The implication from (3) to (1) follows from Proposition 18.6 and Lemma 19.2. The implication from (2) to (3) is proved next via the contrapositive. To do this, it is shown that if $\mathcal{C}_{\text{core}}$ does not have bounded treewidth, then the evaluation problem for $\mathcal{C}_{\text{core}}$ is W[1]-hard under fpt-reductions and, therefore, not in FPT based on the assumption that FPT $\neq$ W[1].

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

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

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

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

Theorem 19.4 (Excluded Grid Theorem). *There is a function* $t : \mathbb{N} \to \mathbb{N}$*, such that for every* $k \geq 1$ *and simple graph* G *of treewidth at least* $t(k)$ *it is the case that* G *contains a* $(k \times K)$*-grid as a minor, for* $K = \binom{k}{2}$*.*

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

(M1) If n is a node of H, then $\mu(n)$ is a connected and nonempty subset of the nodes of G.

(M2) The sets of the form $\mu(n)$, for n a node of H, are pairwise disjoint.

(M3) For every edge $(n_1, n_2) \in E'$, there exist nodes $n_1' \in \mu(n_1)$ and $n_2' \in \mu(n_2)$ such that $(n_1', n_2') \in E$.

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

The following is left as an easy exercise for the reader (see Exercise 14):

Lemma 19.5. *H is a minor of G if and only if there is a minor map μ from H to G. If, in addition, G is connected, then there exists an onto minor map μ from H to G.*

The reduction from p-CLIQUE to p-$\mathcal{C}$-Evaluation is explained next. Consider an input for p-CLIQUE given by the pair (G, k), where G is a simple graph and $k \geq 1$ is an integer. Since $\mathcal{C}_{\text{core}}$ is not of bounded treewidth, there is some CQ $q \in \mathcal{C}$ whose core q' has treewidth at least $t(k)$. This means that there is at least one connected component q^* of q' whose treewidth is at least $t(k)$. By Lemma 19.2 then, the Gaifman graph G_{q^*} of q^* has treewidth at least $t(k)$, and from the Excluded Grid Theorem it is the case that the $(k \times K)$-grid is a minor of G_{q^*}, where $K = \binom{k}{2}$. Notice that q^* is also a core. Moreover, since the $(k \times K)$-grid is a minor of G_{q^*} and G_{q^*} is connected, Lemma 19.5 implies that there exists an onto minor map μ from the $(k \times K)$-grid to G_{q^*}.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

The domain The domain of D consists of all tuples:

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

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

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

(C1) If $i = i'$ then $v = v'$.

(C2) If $p = p'$ then $e = e'$.

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

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

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

Correctness of the construction

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

Lemma 19.6. *If G contains a k-clique, then $q^*(D) = \texttt{true}$.*

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

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

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

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

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

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

Lemma 19.7. *If $q^*(D) = \texttt{true}$, then G contains a k-clique.*

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

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

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

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

Claim 19.8. *For each $i \in [k]$, $p \in [K]$, and $x, x' \in \mu(i, p)$, it is the case that $v_x = v_{x'}$ and $e_x = e_{x'}$.*

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

Claim 19.9. *For each $i, i' \in [k]$, $p \in [K]$, $x \in \mu(i,p)$, and $x' \in \mu(i',p)$, it is the case that $e_x = e_{x'}$.*

Proof. Let us assume without loss of generality that $i \leq i'$. If $i = i'$ then the result holds from Claim 19.8. Let us suppose then that $i < i'$. It is sufficient to establish the result for the case $i' = i + 1$, as all other cases follow easily by induction.

Since there is an edge between $\{i, p\}$ and $\{i+1, p\} = \{i', p\}$ in the $(k \times K)$-grid and μ is a minor map, there is an edge between an element $y \in \mu(i,p)$ and an element $y' \in \mu(i',p)$ in G_{q^*}. Thus, there is a fact $R(\bar{y})$ in q^* such that both y and y' are mentioned in $\bar{y}$. Let us assume without loss of generality that $\bar{y} = (y, y', \dots)$. Since $R(h(\bar{y})) \in D$, it must be the case that $e_y = e_{y'}$ by condition (C2). But by Claim 19.8, it is the case that $e_y = e_x$ and $e_{y'} = e_{x'}$. This finishes the proof of the claim. ☐

Claim 19.10. *For each $i \in [k]$, $p, p' \in [K]$, $x \in \mu(i,p)$, and $x' \in \mu(i,p')$, it is the case that $v_x = v_{x'}$.*

Proof. Analogous to the proof of Claim 19.9 and left as an exercise for the reader. ☐

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

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

Fix an arbitrary pair $\{i, j\}$ with $1 \leq i < j \leq k$ and consider the pair $p \in [K]$ such that $\varphi(p) = \{i, j\}$. It is possible to prove that $e_p = \{v_i, v_j\}$, and, therefore, that $\{v_1, \dots, v_k\}$ defines a k-clique in G. In fact, since μ is a minor map, there are elements $x \in \mu(i,p)$ and $x' \in \mu(j,p)$. From the previous remarks, $h(x) = (v_i, e_p, i, p, x)$ and $h(x') = (v_j, e_p, j, p, x)$. Now, since $v_i \in e_p \iff i \in \varphi(p)$ and $v_j \in e_p \iff j \in \varphi(p)$ by definition, we conclude that $e_p = \{v_i, v_j\}$. This finishes the proof of Lemma 19.7.

20

Approximations of Conjunctive Queries

Since CQs of bounded generalized hypertreewidth can be evaluated in polynomial time, it is meaningful to study the process of approximating a CQ as one in a class of small generalized hypertreewidth. In fact, this process provides a certificate of efficiency for the cost of evaluating such an approximation.

Approximations of this kind are *static*, in the sense that they depend only on the CQ q and not on the underlying database D. This has benefits in terms of the cost of the approximation process, as q is often orders of magnitude smaller than D and an approximation that has been computed once can be used for all databases. Moreover, it allows to develop a principled approach to CQ approximation based on the notion of CQ containment.

In this section we study $\mathsf{GHW}(k)$-*underapproximations*, which correspond to those CQs in $\mathsf{GHW}(k)$ that are "maximally" contained in a given CQ q, In particular, $\mathsf{GHW}(k)$-underapproximations of q (or simply $\mathsf{GHW}(k)$-*approximations* from now on) return sound but not necessarily complete answers. Moreover, if q' is a $\mathsf{GHW}(k)$-approximation of q, then the maximality condition expressed above ensures that no CQ q'' in $\mathsf{GHW}(k)$ with $q'' \subseteq q$ approximates q "better" than q'.

Approximations of bounded generalized hypertreewidth

We define approximations of bounded generalized hypertreewidth as follows.

Definition 20.1 ($\mathsf{GHW}(k)$-approximations). *Fix $k \geq 1$. Let q be a CQ. A* $\mathsf{GHW}(k)$-approximation *of q is a CQ $q' \in \mathsf{GHW}(k)$ that satisfies the following two conditions:*

- *Soundness: The CQ q' only retrieves* sound *answers with respect to q over every database D; in other words, $q' \subseteq q$.*
- *Maximality: The CQ q is* maximally *contained in q; that is, there is no CQ q'' in $\mathsf{GHW}(k)$ with*
$$q' \subsetneq q'' \subseteq q.$$

Intuitively, this means that no CQ q'' in $\mathsf{GHW}(k)$ *approximates q "better" than q' in terms of containment.*

Notice that whenever there is a CQ $q' \in \mathsf{GHW}(k)$ such that $q \equiv q'$ – which happens precisely when the core of q is in $\mathsf{GHW}(k)$ – then the unique $\mathsf{GHW}(k)$-approximation of q is q' itself. That is, the notion of $\mathsf{GHW}(k)$-approximation provides a suitable extension of the notion of being equivalent to a CQ in $\mathsf{GHW}(k)$. The following example shows, on the other hand, that computing an approximation might be useful when exact reformulation in $\mathsf{GHW}(k)$ is impossible.

Example 20.2. An *oriented path* $P = (u_0, \dots, u_n)$ is a directed graph with nodes $u_0, \dots, u_n$ and n edges such that either (u_i, u_{i+1}) or (u_{i+1}, u_i) is an edge, for each $0 \leq i < n$. The oriented path $P = (u_0, \dots, u_n)$ will be represented as a word $b_0 \dots b_{n-1}$ over alphabet $\{0, 1\}$ in such a way that $b_i = 1$ if (u_i, u_{i+1}) is an edge of P, and $b_i = 0$ otherwise.

Consider the oriented paths $P_1 = 001000$ and $P_2 = 000100$. Based on P_1 and P_2, Figure 20.1 depicts boolean CQs q, q_1 and q_2 over the schema that consists of a single binary relation E. Nodes represent variables and an edge from node x to node y represents the presence of a binary atom $E(x, y)$ in the CQ. In addition, an edge from x to y labeled P_1 or P_2 states that there is an oriented P_1 or P_2, respectively, from x to y. It can be shown that all of q, q_1 and q_2 are cores. Moreover, it is straightforward to see that $q \in \mathsf{GHW}(2) \setminus \mathsf{GHW}(1)$ while both q_1 and q_2 are in $\mathsf{GHW}(1)$.

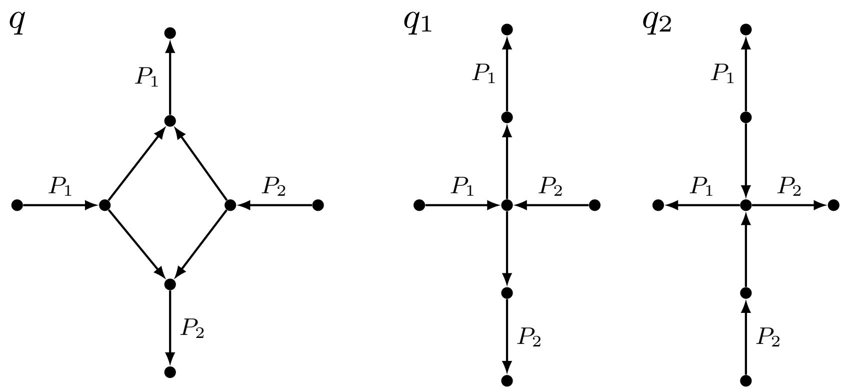


Fig. 20.1: The CQs q, q_1 and q_2 from Example 20.2.

It is possible to prove that q_1 and q_2 are $\mathsf{GHW}(1)$-approximations of q. In fact, it is not hard to see that $q \to q_1$ and $q \to q_2$, thus implying that both $q_1 \subseteq q$ and $q_2 \subseteq q$. Some extra work is required to show that q_1 and q_2 also satisfy the maximality condition; that is, that there is no CQ $q' \in \mathsf{GHW}(1)$ such that $q_1 \subsetneq q' \subseteq q$ or $q_2 \subsetneq q' \subseteq q$. In addition, it can be shown that $q_1 \not\equiv q_2$, from which we can conclude that $\mathsf{GHW}(1)$-approximations are not necessarily unique. □

Existence, size, and computation

$\mathsf{GHW}(k)$-approximations have good properties that justify its application in some contexts: they always exist, are of polynomial size, and the set of all $\mathsf{GHW}(k)$-approximations of a CQ q can be computed in single-exponential time. This is formalized in the theorem below.

Theorem 20.3. *Fix $k \geq 1$:*

1. *Every CQ q has a $\mathsf{GHW}(k)$-approximation.*
2. *Each $\mathsf{GHW}(k)$-approximation of q is of polynomial size (up to equivalence).*
3. *There is an exponential time algorithm that on input a CQ q computes the set of all its $\mathsf{GHW}(k)$-approximations.*

The proof of Theorem 20.3 relies on a crucial *interpolation lemma* presented next:

Lemma 20.4 (Interpolation Lemma). *Let q be a CQ with n atoms ($n \geq 1$) and q' a CQ in $\mathsf{GHW}(k)$ such that $q' \subseteq q$. There is a CQ $q'' \in \mathsf{GHW}(k)$ such that $q' \subseteq q'' \subseteq q$ and q'' has at most $n \cdot (2k+1)$ atoms.*

Before proving Lemma 20.4 we explain how Theorem 20.3 follows from it. First, observe that for every CQ $q(\bar{x})$ there is at least one CQ $q_k(\bar{y})$ in $\mathsf{GHW}(k)$ such that $q_k \subseteq q$ and q_k has no more atoms than q; simply take a single variable y and add an atom $R(y, \ldots, y)$ to q_k for each relation symbol R that is mentioned in q. Then set $\bar{y} = (y, \ldots, y)$, where $\bar{y}$ is a tuple of the same arity than $\bar{x}$. The resulting CQ q_k is in $\mathsf{GHW}(1)$, and thus in $\mathsf{GHW}(k)$ for each $k \geq 1$. Furthermore, it is possible to define a homomorphism from q to q_k by simply mapping each variable of q to z. Thus, $q_k \subseteq q$ from Theorem 15.1.

Let q be a CQ with n atoms and consider the set $\mathsf{Cont}_{\mathsf{GHW}(k)}(q)$ of CQs q' in $\mathsf{GHW}(k)$ with at most $n \cdot (2k+1)$ atoms such that $q' \subseteq q$. From the above discussion, at least q_k belongs to $\mathsf{Cont}_{\mathsf{GHW}(k)}(q)$. Let us then consider the set $\mathsf{Maximal}_{\mathsf{GHW}(k)}(q)$ consisting of the $\subseteq$-maximal elements of $\mathsf{Cont}_{\mathsf{GHW}(k)}(q)$. It is possible to show that $\mathsf{Maximal}_{\mathsf{GHW}(k)}(q)$ is precisely the set of $\mathsf{GHW}(k)$-approximations of q (up to equivalence). In fact, consider first a $\mathsf{GHW}(k)$-approximation q' of q. By definition, $q' \in \mathsf{GHW}(k)$ and $q' \subseteq q$, and, thus, from Lemma 20.4 there is a CQ $q^* \in \mathsf{GHW}(k)$ such that $q' \subseteq q^* \subseteq q$ and q^* has at most $n \cdot (2k+1)$ atoms. Therefore, $q^* \in \mathsf{Cont}_{\mathsf{GHW}(k)}(q)$, and there is a CQ $q'' \in \mathsf{Maximal}_{\mathsf{GHW}(k)}(q)$ such that $q' \subseteq q^* \subseteq q'' \subseteq q$. By definition, $q'' \in \mathsf{GHW}(k)$ and, hence, $q' \equiv q''$ since q' is a $\mathsf{GHW}(k)$-approximation of q. The proof that each CQ $q' \in \mathsf{Maximal}_{\mathsf{GHW}(k)}(q)$ is a $\mathsf{GHW}(k)$-approximation of q follows a similar reasoning.

Note that $\mathsf{Maximal}_{\mathsf{GHW}(k)}(q)$ contains at least one CQ (since $\mathsf{Cont}_{\mathsf{GHW}(k)}(q)$ is nonempty). Thus, each CQ q has at least one $\mathsf{GHW}(k)$-approximation. This yields item (1) of Theorem 20.3. Item (2) follows from the aforementioned observation that the elements of $\mathsf{Maximal}_{\mathsf{GHW}(k)}(q)$ are precisely the $\mathsf{GHW}(k)$-approximations of q (up to equivalence). For item (3), it is sufficient to observe that the set $\mathsf{Maximal}_{\mathsf{GHW}(k)}(q)$ can be computed in single-exponential time. This is done by enumerating all CQs q' with at most $n \cdot (2k+1)$ atoms, and for each one of them checking the following: (a) $q' \in \mathsf{GHW}(k)$, (b) $q' \subseteq q$, and (c) there is no $q'' \in \mathsf{GHW}(k)$ with $n \cdot (2k+1)$ atoms such that $q' \subsetneq q'' \subseteq q$. Each one of these steps can be carried out in single-exponential time.

Proof (of Lemma 20.4). Since, by hypothesis, $q' \in \mathsf{GHW}(k)$, the CQ q' (or, strictly speaking, its hypergraph $H_{q'}$) admits a generalized hypertree decomposition (T, χ, λ) of width k. The goal is, by exploiting the existence of (T, χ, λ) and the fact that $q' \subseteq q$, to first construct a set of at most $n \cdot (2k+1)$ atoms, where n is the number of atoms in q, and then show that this set can be turn into a CQ $q'' \in \mathsf{GHW}(k)$ with the same number of atoms such that $q' \subseteq q'' \subseteq q$.

<u>The construction of the set of atoms:</u> Let us assume that q is of the form $q(\bar{x}) \coloneq R_1(\bar{x}_1), \ldots, R_m(\bar{x}_m)$. By hypothesis, $q'(\bar{x}') \subseteq q(\bar{x})$, and thus Theorem 15.1 implies the existence a homomorphism h from q to q' such that $h(\bar{x}) = \bar{x}'$. By definition, for each $1 \leq i \leq m$ there exists a node v_i in T such that $\chi(v_i)$ contains all the variables occurring in $R_i(h(\bar{u}_i))$. Consider now the subtree T_q of T consisting of $v_1, \ldots, v_m$ and their ancestors in T. From T_q one extracts the tree $F = (V, E)$ defined as follows:

- V consists of all the root nodes and leaf nodes of T_q, and all the inner nodes of T_q with at least two children.
- For $v, u \in V$, it is the case that $(v, u) \in E$ if and only if u is descendant of v in T_q, and the only nodes of V that occur on the unique shortest path from v to u in T_q is v and u.

The construction of F is illustrated in Figure 20.2. The generalized hypertree decomposition (T, χ, λ) of q' is shown in Figure 20.2(a), where the shaded part are the nodes that cover the atoms in the image of the CQ q according to the homomorphism h. The subtree T_q of T is depicted in Figure 20.2(b), where the red nodes form the node set V of F. Finally, the tree $F = (V, E)$ is shown in Figure 20.2(c), which is essentially obtained from T_q by replacing the unique shortest path between two red nodes with a single edge.

Let $S' = \bigcup_{v \in V} \lambda(v)$, i.e., S' is the set consisting of all atoms that "cover" the nodes of F. At this point, one may be tempted to think that (F, χ', λ'), where χ' (resp., λ') is the restriction of χ (resp., λ) over the nodes of V, is

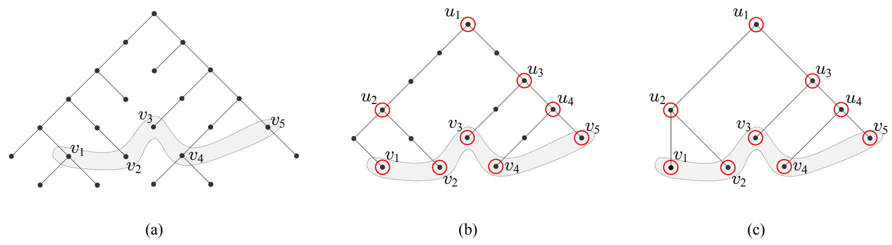


Fig. 20.2: The construction of the tree $F = (V, E)$ in the proof of Lemma 20.4.

a generalized hypertree decomposition of the hypergraph induced by the set of atoms in S'. However, this is not the case; it is easy to verify that one may have an atom $R(\bar{u}) \in S'$, but no $v \in V$ such that $\chi'(v)$ contains all the variables occurring in $\bar{u}$.

Nevertheless, from (F, χ', λ') one can construct a triple (F, χ'', λ'') that is a generalized hypertree decomposition with width k of the hypergraph induced by the set of atoms

$$S'' \;=\; \bigcup_{1 \leq i \leq m} R_i(h(\bar{u}_i)) \;\cup\; \bigcup_{v \in V} \lambda''(v).$$

The construction of (F, χ'', λ'') from (F, χ', λ') follows. Consider an arbitrary node $v \in V$ such that $\chi'(v) = \{x_1, \ldots, x_\ell\}$, where $\ell \geq 1$, and the variables in $\lambda'(v)$ are $\{x_1, \ldots, x_\ell\} \cup \{x_{\ell+1}, \ldots, x_{\ell+p}\}$, where $p \geq 0$; notice that $p = 0$ implies $\{x_{\ell+1}, \ldots, x_{\ell+p}\} = \emptyset$. Then

$$\chi''(v) \;=\; \{x_1, \ldots, x_\ell, z_{\ell+1}, \ldots, z_{\ell+p}\},$$

where $z_{\ell+1}, \ldots, z_{\ell+p}$ are fresh variables occurring only in $\chi''(v)$, i.e., there is no node $u \neq v$ such that $\chi''(u)$ contains any of those variables, and $\lambda''(v)$ is obtained from $\lambda'(v)$ by replacing each variable $x_{\ell+i}$, for $1 \leq i \leq p$, with $z_{\ell+i}$. For example, assume that $\chi'(v) = \{x_1, x_2\}$ and $\lambda'(v) = \{R(x_1, x_3), P(x_2, x_3, x_4)\}$. Then $\chi''(v) = \{x_1, x_2, z_3, z_4\}$ and $\lambda''(v) = \{R(x_1, z_3), P(x_2, z_3, z_4)\}$. It is easy to verify that (F, λ'', χ'') is indeed a generalized hypertree decomposition with width k of the hypergraph induced by the atoms in S''.

It is shown next that S'' has at most $n \cdot (2k+1)$ atoms, which then implies that the desired set of atoms is precisely S''. By construction, F has at most $2n$ nodes, while, for each node $v \in V$ it is the case that $|\chi''(v)| \leq k$. Therefore, $|S''| \leq (2n \cdot k + n) = n \cdot (2k+1)$.

Converting S'' into a CQ q'': We first observe that, by construction, each $x \in \bar{x}'$ occurs in S''. Let $q''(\bar{x}')$ be the CQ obtained by considering the conjunction of atoms in S''. It is clear that $q'' \in \mathsf{GHW}(k)$ and the number of atoms in q'' is at most $n \cdot (2k+1)$. Moreover, h is a homomorphism from q to q'' such that $h(\bar{x}) = \bar{x}'$ and thus, by Theorem 15.1, $q'' \subseteq q$. In addition, there exists a homomorphism μ from q'' to q' such that $\mu(\bar{x}') = \bar{x}'$, and hence, $q' \subseteq q''$. The homomorphism μ is obtained by simply reversing the renaming substitutions applied during the construction of (F, λ'', χ'') from (F, λ', χ'). Since $q' \subseteq q'' \subseteq q$, the claim follows. □

Evaluation of approximations

Let us look at the problem of evaluating the $\mathsf{GHW}(k)$-approximations of q, i.e., given a CQ q, a database D, and a tuple $\bar{t}$ in D, checking whether $\bar{t} \in q'(D)$ for some $\mathsf{GHW}(k)$-approximation q' of q. Since each such a q' is contained in q, we can then be sure that $\bar{t}$ also belongs to $q(D)$. As stated next, this problem can be solved by a fixed-parameter tractable algorithm:

Proposition 20.5. *Fix $k \geq 1$. Given a CQ $q(\bar{x})$, a database D, and a tuple $\bar{t}$ of elements in $Dom(D)$ with $|\bar{t}| = |\bar{x}|$, checking if $\bar{t} \in q'(D)$ for some* $\mathsf{GHW}(k)$*-approximation q' of q can be solved in time*

$$2^{r(||q||)} \ + \ ||D||^k \cdot 2^{r'(||q||)},$$

for polynomials $r, r' : \mathbb{N} \to \mathbb{N}$.

Proof. First, compute the set of all $\mathsf{GHW}(k)$-approximations of q. This can be done in time $2^{s(||q||)}$, for some polynomial $s : \mathbb{N} \to \mathbb{N}$. Then check whether $\bar{t} \in q'(D)$, for at least one such a $\mathsf{GHW}(k)$-approximation q'. From Theorem 18.10, this can be done in time $2^{s'(||q||)} \ + \ ||D||^k \cdot ||q||$, for some polynomial $s' : \mathbb{N} \to \mathbb{N}$. The result follows directly. □

This explains why it is convenient, in some cases, to evaluate the approximations of a CQ q as as way to obtain quick answers when exact evaluation is infeasible or is taking too long. Suppose, in particular, that q is not equivalent to a CQ in $\mathsf{GHW}(k)$. Hence, it must be the case that $q \in \mathsf{GHW}(k')$ for some $k' > k$. Let us assume that a generalized hypertree decomposition of q of width k' is available. One can then use this decomposition to solve the exact evaluation problem for q over D in time $O(||D||^{k'} \cdot ||q||)$. Still, in the realistic case in which D is too large – in particular, when $2^{r(||q||)} + 2^{r'(||q||)} \ll ||D||$ – evaluating the $\mathsf{GHW}(k)$-approximations of q over D in time $2^{r(||q||)} \ + \ ||D||^k \cdot 2^{r'(||q||)}$ can be considerably faster than evaluating q itself in time $O(||D||^{k'} \cdot ||q||)$.

Proposition 20.5 states that the problem of evaluating the $\mathsf{GHW}(k)$-approximations of q is fixed-parameter tractable. If PTIME $\neq$ NP, on the other hand, this problem cannot be solved in polynomial time.

Proposition 20.6. *Consider an arbitrary $k \geq 1$. The following problem is* NP*-hard: Given a database D and a boolean CQ q, determine whether for some* $\mathsf{GHW}(k)$*-approximation q' of q it is the case that $q'(D) = \texttt{true}$.*

Proof. Recall that it is NP-complete to check whether a boolean CQ q is equivalent to some CQ in $\mathsf{GHW}(k)$. It is an easy exercise to prove that the latter holds if and only if for some $\mathsf{GHW}(k)$-approximation q' of q it is the case that $q'(D_q) = \texttt{true}$. The proof is left to the reader. □

21

Bounding the Join Size

Wim:
A note after hearing Ke Yi's talk at ICDT18: He said that no tree-structured join plan can match the efficiency of the AGM bound. I guess he means the example of pairwise joins, but I'm not sure. It's an easy and insightful example. Can we incorporate this in the story somewhere? (Perhaps in the beginning of the next chapter?)

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

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

The AGM bound

We consider in this chapter the named perspective for relational algebra. Recall that under this perspective, as defined in Chapter 4, the natural join $R_1 \bowtie R_2$ of relations R_1, R_2 is defined as the join of R_1 with R_2 on the condition that their common attributes are the same, and with the additional requirement that one copy of each common attribute is dropped.

For the sake of presentation, this section only deals with the class of CQs that represent natural joins $R_1 \bowtie \cdots \bowtie R_n$. Notice that we can assume the relation symbols $R_1, \ldots, R_n$ are pairwise different, since the natural join operator $\bowtie$ is commutative, associative and idempotent (that is,

$(R \bowtie R)(D) = R(D)$ for every database D). Moreover, CQs of this form are projection free. For several of the results presented next this restriction is inessential, and in fact such results continue to hold for arbitrary CQs at the cost of more complicated proofs.

Let $q = R_1 \bowtie \cdots \bowtie R_n$ be a query of the form described before, and assume that $\{A_1, \ldots, A_m\}$ is the set of all attributes occurring in q. The set of relation symbols in $\{R_1, \ldots, R_n\}$ that mention attribute A_j, for $j \in [1, m]$, is denoted by $\mathsf{SR}(q, A_j)$. Then a *fractional cover* of q is any real solution $(r_1, \ldots, r_n)$ to the following system of equations:

$$\begin{aligned} \sum_{R_i \in \mathsf{SR}(q,A_j)} r_i \ &\geq\ 1, \quad \text{for each } j \in [1, m], \\ r_i \ &\geq\ 0, \quad \text{for each } i \in [1, n]. \end{aligned}$$

That is, with each relation symbol R_i, we associate a variable r_i whose value must be non-negative. In addition, for each attribute A_j, the sum of the values of the r_i's, for those R_i's that “cover” A_j in the sense that A_j is an attribute of R_i, must be at least 1.

Recall that R^D is the interpretation of the relation symbol R in the database D, i.e., the set of all tuple $\bar{t}$ such that $R(\bar{t}) \in D$. With this notation, the AGM bound is stated next.

Theorem 21.1 (AGM bound). *Consider a natural join query $q = R_1 \bowtie \cdots \bowtie R_n$ and a fractional cover $(r_1, \ldots, r_n)$ of q. Then for every database D:*

$$|q(D)| \ \leq\ \prod_{i=1}^{n} |R_i^D|^{r_i}.$$

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

$$q(A, B, C) \ \text{:-}\ R(A, B), S(B, C), T(C, A),$$

which defines the set of “triangles” RST in a database D. The system of equations associated to this query is as follows:

$$r + t \ \geq\ 1, \quad r + s \ \geq\ 1, \quad s + t \ \geq\ 1, \quad r, s, t \ \geq\ 0,$$

where the variables associated to R, S, T are r, s and t, respectively. Thus, for example, the first equation above is obtained by considering attribute A and the fact that $\mathsf{SR}(q, A) = \{R, T\}$. One solution to this system of equations is $r = s = t = 1/2$. Therefore, from the AGM bound one obtains that over every database D:

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

Another solution is $r = s = 1$ and $t = 0$. Applying the AGM bound one then obtains that:

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

Which bound is better depends on the underlying database D.

- Consider first the case when $R^D = S^D = T^D$, i.e., when the interpretations of all three relations over D coincide. Moreover, assume that R^D does not contain any tuple of the form (a, a). Then D can be naturally seen as a graph without loops and with $M = |R^D|$ edges, and, thus, $|q(D)|$ is equal to three times the number of directed triangles in such a graph.[1] In this case, the first bound is better, as it implies that $|q(D)| \leq M^{3/2}$, while the second one implies that $|q(D)| \leq M^2$.
  The fact that the maximum number of directed triangles in a graph with M edges is bounded by
  $$\frac{M^{3/2}}{3}$$
  is non-trivial, which illustrates the power of Theorem 21.1 as a tool for obtaining meaningful bounds on the size of the evaluation of a natural join query.
- Consider now the case when $|R^D| = |S^D| = 1$ and $|T^D| = M$. Then the second bound is tighter, as it implies that $|q(D)| = 1$ while the first one implies that $|q(D)| \leq \sqrt{M}$.

Consider a natural join query $q = R_1 \bowtie \cdots \bowtie R_n$ as before, and a database D. How can one find a fractional cover of q that minimizes the value of the AGM bound for q over D? Assuming that $|R_i^D| \geq 1$ for each $i \in [1, n]$, it is not hard to see that it suffices to solve the following linear program, which is obtained by minimizing the value of $\log_2(\prod_{i=1}^{n} |R_i^D|^{r_i})$, instead of directly minimizing the value of the AGM bound:

$$\begin{array}{lllr}
\text{minimize} & \displaystyle\sum_{i=1}^{n} (\log_2 |R_i^D|) \cdot r_i & & \\
\text{subject to} & \displaystyle\sum_{R_i \in \mathsf{SR}(q, A_j)} r_i \;\geq\; 1, & \text{for each } j \in [1, m], & (21.1) \\
& r_i \geq 0, & \text{for each } i \in [1, n]. &
\end{array}$$

This linear program depends on q and D; in particular, $\log_2 |R_i^D|$ is a non-negative constant for each $i \in [1, n]$. Let $\rho(q, D)$ be the optimal value of this

[1] Notice that each directed triangle is counted three times as a tuple $(A\colon a, B\colon b, C\colon c)$ is considered to be different from a tuple $(A\colon b, B\colon c, C\colon a)$.

linear program, which is known as the *optimal fractional cover of q over D*.[2] Then the following corollary is obtained from Theorem 21.1.

Corollary 21.2. *Consider a natural join query $q = R_1 \bowtie \cdots \bowtie R_n$ and a database D. If $|R_i^D| \geq 1$ for every $i \in [1, n]$, then*

$$|q(D)| \;\leq\; 2^{\rho(q,D)}.$$

Proof of the AGM bound

The proof makes use of a key *query decomposition lemma*, which is in turn proved by applying the following version of Hölder's inequality.

Claim 21.3 (Hölder's inequality). *Let p, r be positive integers, $y_1, \ldots, y_r$ be non-negative real numbers such that $y_1 + \cdots + y_r \geq 1$, and $a_{i,j}$ be a non-negative real number, for every $i \in [1, p]$ and $j \in [1, r]$. Then, it holds that:*

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

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

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

Recall that $\mathrm{sort}(R)$ is the set of attributes associated to a relation symbol R. Thus, query q_Y defines the natural join of the projection over Y of those R_i's that mention at least some attribute in Y (notice that $\pi_Y(R_i)$ may not be well defined if Y contains an attribute not occurring in $\mathrm{sort}(R_i)$, so we need to include $\pi_{Y \cap \mathrm{sort}(R_i)}(R_i)$ in q_Y instead of $\pi_Y(R_i)$).

Finally, to state the query decomposition lemma, recall the definition of the semijoin operator $\ltimes$ from Chapter 17. In particular, recall that two tuples $\bar{a}$ and $\bar{b}$ are said to be consistent if they have the same value in each shared attribute. Thus, the term $R \ltimes \{\bar{t}\}$ is used in the lemma to denote the set of tuples in R that are consistent with $\bar{t}$.

[2] Given the definition of linear program (21.1) and, in particular, given that $\log_2 |R_i^D| \geq 0$ for each $i \in [1, n]$, it is not hard to see that the same optimal value as for the linear program (21.1) can be obtained if the additional restriction $r_i \leq 1$ is imposed for each $i \in [1, n]$. Therefore, we have that $\rho(q, D)$ is well defined, as we are minimizing a continuous function over a closed and bounded subset of $\mathbb{R}^n$.

Lemma 21.4 (Query decomposition lemma). *Assume that* $(r_1, \ldots, r_n)$ *is a fractional cover for* q*, and consider an arbitrary partition* Y*,* Z *of* X*. Then for every database* D*, it holds that*

$$\sum_{\bar{t} \in q_Y(D)} \prod_{R_i \in \mathsf{SR}(q,Z)} |R_i^D \ltimes \{\bar{t}\}|^{r_i} \;\leq\; \prod_{i=1}^{n} |R_i^D|^{r_i}.$$

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

$$\sum_{\bar{t} \in q_Y(D)} \prod_{R_i \in \mathsf{SR}(q,Z)} |R_i^D \ltimes \{\bar{t}\}|^{r_i} \;\leq\; \sum_{\bar{t}' \in q_{Y'}(D)} \prod_{R_i \in \mathsf{SR}(q,Z')} |R_i^D \ltimes \{\bar{t}'\}|^{r_i}. \quad (21.2)$$

The lemma is then obtained by repeatedly applying (21.2) until Y' is empty, in which case the right-hand side of the inequality is precisely $\prod_{i=1}^{n} |R_i^D|^{r_i}$.

Each tuple $\bar{t} \in q_Y(D)$ can be decomposed as a pair $\bar{t}', v$ such that $\bar{t}' \in q_{Y'}(D)$ and v is the value of $\bar{t}$ for attribute A. In what follows, we use term $(\bar{t}', v)$ as an alternative notation for tuple $\bar{t}$. Then the left-hand side of equation (21.2) can be expressed as

$$\sum_{\bar{t}' \in q_{Y'}(D)} \sum_{v \,:\, (\bar{t}',v) \in q_Y(D)} \prod_{R_i \in \mathsf{SR}(q,Z)} |R_i^D \ltimes \{(\bar{t}', v)\}|^{r_i},$$

which in turn can be rewritten as

$$\sum_{\bar{t}' \in q_{Y'}(D)} \sum_{v \,:\, (\bar{t}',v) \in q_Y(D)} \Bigg(\bigg(\prod_{R_i \in \mathsf{SR}(q,Z)} |R_i^D \ltimes \{(\bar{t}', v)\}|^{r_i} \bigg) \cdot \bigg(\prod_{R_i \in \mathsf{SR}(q,Z') - \mathsf{SR}(q,Z)} 1^{r_i} \bigg) \Bigg). \quad (21.3)$$

Notice that equation (21.3) is equivalent to

$$\sum_{\bar{t}' \in q_{Y'}(D)} \sum_{v \,:\, (\bar{t}',v) \in q_Y(D)} \prod_{R_i \in \mathsf{SR}(q,Z')} |R_i^D \ltimes \{(\bar{t}', v)\}|^{r_i}, \quad (21.4)$$

since for those R_i in $\mathsf{SR}(q,Z') - \mathsf{SR}(q,Z)$, it holds that the set of attributes of R_i is contained in Y and, thus, $|R_i^D \ltimes \{(\bar{t}', v)\}| = 1$. Moreover, given that $R_i^D \ltimes \{(\bar{t}', v)\} = R_i^D \ltimes \{\bar{t}'\}$ for each $R_i \in \mathsf{SR}(q,Z') - \mathsf{SR}(q,A)$, equation (21.4) can be expressed as

$$\sum_{\bar{t}' \in q_{Y'}(D)} \prod_{R_i \in \mathsf{SR}(q,Z') - \mathsf{SR}(q,A)} |R_i^D \ltimes \{\bar{t}'\}|^{r_i} \sum_{v \,:\, (\bar{t}',v) \in q_Y(D)} \prod_{R_i \in \mathsf{SR}(q,A)} |R_i^D \ltimes \{(\bar{t}', v)\}|^{r_i}. \quad (21.5)$$

This is the moment in which Hölder's inequality stated in Claim 21.3 is applied to obtain that the value expressed in equation (21.5) is bounded by

$$\sum_{\bar{t}' \in q_{Y'}(D)} \prod_{R_i \in \mathsf{SR}(q,Z') - \mathsf{SR}(q,A)} |R_i^D \ltimes \{\bar{t}'\}|^{r_i} \\ \prod_{R_i \in \mathsf{SR}(q,A)} \Bigg(\sum_{v \,:\, (\bar{t}', v) \in q_Y(D)} |R_i^D \ltimes \{(\bar{t}', v)\}| \Bigg)^{r_i}. \quad (21.6)$$

Observe that it is possible to apply Claim 21.3 because: (i) $(r_1, \ldots, r_n)$ is a fractional cover of q and, thus, $\sum_{R_i \in \mathsf{SR}(q,A)} r_i \geq 1$; and (ii) $|R_i^D \ltimes \{(\bar{t}', v)\}| \geq 0$ for every $R_i \in \mathsf{SR}(q, A)$ and v such that $(\bar{t}', v) \in q_Y(D)$.

To conclude, the value expressed in Equation (21.6) is bounded by

$$\sum_{\bar{t}' \in q_{Y'}(D)} \prod_{R_i \in \mathsf{SR}(q,Z') - \mathsf{SR}(q,A)} |R_i^D \ltimes \{\bar{t}'\}|^{r_i} \prod_{R_i \in \mathsf{SR}(q,A)} |R_i^D \ltimes \{\bar{t}'\}|^{r_i}. \quad (21.7)$$

This is because for every $R_i \in \mathsf{SR}(q, A)$: (i) $R_i^D \ltimes \{(\bar{t}', v)\} \subseteq R_i^D \ltimes \{\bar{t}'\}$; and (ii) if $(\bar{t}', v_1)$ and $(\bar{t}', v_2)$ are in $q_Y(D)$, with $v_1 \neq v_2$, then $R_i^D \ltimes \{(\bar{t}', v_1)\}$ and $R_i^D \ltimes \{(\bar{t}', v_2)\}$ are disjoint given that $A \in \text{sort}(R_i)$. Finally, equation (21.7) can be simplified as

$$\sum_{\bar{t}' \in q_{Y'}(D)} \prod_{R_i \in \mathsf{SR}(q,Z')} |R_i^D \ltimes \{\bar{t}'\}|^{r_i},$$

which finishes the proof of the lemma. □

We now move to the proof of Theorem 21.1, which is done by induction on the size of the set $X = \{A_1, \ldots, A_m\}$ of attributes occurring in q. Notice that this theorem trivially holds if $|R_i^D| = 0$ for some $i \in [1, n]$. Thus, we assume in the following proof that $|R_i^D| \geq 1$ for every $i \in [1, n]$.

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

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

  where the second inequality holds given that $\min_{\ell \in [1,n]} |R_\ell^D| \geq 1$ and that $\sum_{i=1}^n r_i \geq 1$.

- **Inductive case.** We have that $|X| = m$ with $m > 1$. Consider an arbitrary partition $Y \uplus Z$ of X. Recall that then we have that $1 \leq |Y| < m$. For each $\bar{t} \in q_Y(D)$, let us define a database $D_{\bar{t}}$ over the schema $\mathsf{SR}(q, Z)$ such that for each $R \in \mathsf{SR}(q, Z)$, it is the case that $R^{D_{\bar{t}}} = R^D \ltimes \{\bar{t}\}$. It is easy to see that

$$q(D) \;=\; \bigcup_{\bar{t} \in q_Y(D)} q_Z(D_{\bar{t}}) \bowtie \{\bar{t}\},$$

and, therefore,

$$|q(D)| \;\leq\; \sum_{\bar{t} \in q_Y(D)} |q_Z(D_{\bar{t}})|.$$

Observe that the restriction of $(r_1, \dots, r_n)$ to those r_i's such that $R_i \in \mathsf{SR}(q, Z)$ is a fractional cover of q_Z. Given that $|Z| < m$, we obtain by induction hypothesis that

$$|q_Z(D_{\bar{t}})| \;\leq\; \prod_{R_i \in \mathsf{SR}(q,Z)} |R_i^{D_{\bar{t}}}|^{r_i} \;=\; \prod_{R_i \in \mathsf{SR}(q,Z)} |R_i^D \ltimes \{\bar{t}\}|^{r_i}.$$

It is possible then to conclude that

$$|q(D)| \;\leq\; \sum_{\bar{t} \in q_Y(D)} |q_Z(D_{\bar{t}})| \;\leq\; \sum_{\bar{t} \in q_Y(D)} \prod_{R_i \in \mathsf{SR}(q,Z)} |R_i^D \ltimes \{\bar{t}\}|^{r_i} \;\leq\; \prod_{i=1}^{n} |R_i^D|^{r_i},$$

where the last inequality holds by Lemma 21.4.

This finishes the proof of Theorem 21.1.

Optimality of the AGM bound

As shown next, the bound stated in Theorem 21.1 is strict. This is formalized in the following proposition.

Proposition 21.5. *Let $q = R_1 \bowtie \cdots \bowtie R_n$ be a natural join query. Then there exist arbitrarily large databases D such that $|R_i^D| \geq 1$ for each $i \in [1, n]$ and*

$$|q(D)| \;=\; 2^{\rho(q,D)}.$$

Proof. Assume that $X = \{A_1, \dots, A_m\}$ is the set of attributes occurring in q. Moreover, let $K_1, \dots, K_n \in \mathbb{N}$ such that $K_i = 2^{\ell_i}$ with $\ell_i \in \mathbb{N}$, for every $i \in [1, n]$. Finally let $B(q, K_1, \dots, K_n)$ be the following linear program:

$$\begin{array}{lll} \text{minimize} & \displaystyle\sum_{i=1}^{n} (\log_2 K_i) \cdot r_i & \\ \text{subject to} & \displaystyle\sum_{R_i \in \mathsf{SR}(q, A_j)} r_i \ \geq \ 1, & \text{for each } j \in [1, m], \\ & r_i \ \geq \ 0, & \text{for each } i \in [1, n]. \end{array}$$

In particular, when $K_i = |R_i^D|$, for some database D, then $B(q, K_1, \ldots, K_n)$ is the linear program in (21.1).

Consider now the *dual* $S(q, K_1, \ldots, K_n)$ of $B(q, K_1, \ldots, K_n)$, which is the following linear program:

$$\begin{array}{lll} \text{maximize} & \displaystyle\sum_{j=1}^{m} p_j & \\ \text{subject to} & \displaystyle\sum_{j \,:\, A_j \in \mathrm{sort}(R_i)} p_j \ \leq \ \log_2 K_i, & \text{for each } i \in [1, n], \\ & p_j \geq 0, & \text{for each } j \in [1, m]. \end{array}$$

The strong duality theorem implies that if $(r_1, \ldots, r_n)$ and $(p_1, \ldots, p_m)$ are optimal solutions of $B(q, K_1, \ldots, K_n)$ and $S(q, K_1, \ldots, K_n)$, respectively, then

$$\sum_{i=1}^{n} (\log_2 K_i) \cdot r_i \ = \ \sum_{j=1}^{m} p_j.$$

Therefore, when $K_i = |R_i^D|$, for some database D, it follows by definition that

$$\rho(q, D) \ = \ \sum_{i=1}^{n} (\log_2 |R_i^D|) \cdot r_i \ = \ \sum_{j=1}^{m} p_j.$$

Given that all the K_i's are powers of two, the coefficients of $S(q, K_1, \ldots, K_n)$ are integers, which implies that this linear program has an optimal *rational* solution $(p_1, \ldots, p_m) \in \mathbb{Q}^m$. Let $s_1, \ldots, s_m$ and t be integers such that $p_j = s_j / t$ for each $j \in [1, m]$. For arbitrary such $K_1, \ldots, K_n$, it is shown next how to construct a database D with $|R_i^D| = K_i^t \geq 1$ such that

$$|q(D)| \ \geq \ 2^{s_1 + \cdots + s_m}.$$

This implies that $|q(D)| \geq 2^{\rho(q,D)}$, as desired. In fact, it is easy to see that $(s_1, \ldots, s_m)$ is an optimal solution of $S(q, K_1^t, \ldots, K_n^t)$. Hence, $s_1 + \cdots + s_m = \rho(q, D)$ by duality.

The database D is defined as follows. Recall that $\{A_1, \ldots, A_m\}$ is the set of attributes occurring in q. First, an auxiliary relation R_q with attributes $A_1, \ldots, A_m$ is defined over D, in such a way that R_q^D corresponds to the set

$$\{1, \ldots, 2^{s_1}\} \times \cdots \times \{1, \ldots, 2^{s_m}\}.$$

From this, auxiliary relations $T_1^D, \dots, T_n^D$ are defined, in such a way that T_i^D corresponds precisely to the projection of R_q^D over the attributes that are mentioned by R_i. Finally, an arbitrary superset R_i^D of T_i^D with $|R_i^D| = K_i^t$ is chosen, for every $i \in [1, n]$. Notice that this construction is well defined since $|T_i^D| \leq K_i^t$. To see why this is the case, notice that by definition

$$|T_i^D| \;=\; 2^{s_{j_1} + \cdots + s_{j_p}},$$

assuming that $\mathrm{sort}(R_i) = \{A_{j_1}, \dots, A_{j_p}\}$. But $2^{s_{j_1} + \cdots + s_{j_p}} \leq K_i^t$, given that $s_{j_1} + \cdots + s_{j_p} \leq \log_2 K_i^t$ as $(s_1, \dots, s_m)$ is a solution of $S(q, K_1^t, \dots, K_n^t)$.

Clearly, $|q(D)| \geq |R_q(D)| = 2^{s_1 + \cdots + s_m}$ by construction. This concludes the proof of the proposition. $\square$

22

The Leapfrog Algorithm

Marcelo / Pablo

Wim:
Dump of my course notes about this (with Matthias Niewerth). May already be close to what we need?

In this chapter we inspect a worst-case optimal algorithm for computing join queries. In Section 22.2, we introduce the Leapfrog Algorithm for computing natural joins of unary relations. In the next section, we introduce the trie datastructure for relations. Finally, we see how the Leafrog Algorithm can be used on Tries to compute arbitrary natural joins.

In this chapter, all relations have their attributes ordered according to some global total order. This is necessary for the Leapfrog Triejoin algorithm to work.

22.1 Linear Iterator Interface

The linear iterator interface has 3 methods:

key()	returns $\begin{cases} \bot & \text{if the iterator is at the beginning} \\ \top & \text{if the iterator is at the end} \\ \text{the value at the current position} & \text{otherwise} \end{cases}$
next()	move iterator to next tuple
seek(key)	Moves the iterator to the smallest position with a value greater or equal to key. Moves the iterator to the end if no such value exists.

The special values $\bot$ and $\top$ encode that the iterator is just before the first or just after the last tuple, respectively. For the algorithm to work, we assume that $\bot$ is smaller than any actual value and $\top$ is larger than any actual value.

22.2 Leapfrog Algorithm

The Leapfrog Algorithm is depicted as Algorithm 8. It uses the iterator interface to implement the natural join of unary relations. The result is again given by the iterator interface. In fact, Algorithm 8 can be seen as a class that implements an iterator for the output relation. To actually compute a join of unary relations, we need to call the leapfrog function with iterators of these relations and then call next() and key() alternatingly until key() returns $\top$.

Algorithm 8 Leapfrog-Join

```
Input: linear iterators $J_0, \dots, J_{k-1}$ for unary relations $R_0, \dots, R_{k-1}$
Output: linear iterator for $R_0 \bowtie \dots \bowtie R_{k-1}$
  function search()
      max := $I_{p-1 \bmod k}$.key()
      min := $I_p$.key()
      while min $\neq$ max do
          $I_p$.seek(max)
          max := $I_p$.key()
          $p := p + 1 \mod k$
          min := $I_p$.key()
  function next()
      $I_p$.next()
      $p := p + 1 \mod k$
      search()
  function seek(key)
      $I_p$.seek(key)
      $p := p + 1 \mod k$
      search()
  function key()
      return $I_p$.key()
  function leapfrog($J_0, \dots, J_{k-1}$)          ▷ Constructor for leapfrog iterator
      $(I_0, \dots, I_{k-1}) := (J_0, \dots, J_{k-1})$
      $p := 0$
      return this
```

We skip a formal correctness proof and just take a look at the runtime. The function seek() can be called at most $|R_i|$ times on relation i. As the algorithm calls the function seek() equally often on all relations, the total number of calls to seek() is bounded by $k \cdot \min_i |R_i|$. As the total runtime is dominated by calls to seek(), we have a runtime of $O(k \cdot \log(\max_i |R_i|) \cdot \min_i |R_i|)$. If we use hash-indexes with amortized constant time seeks, we can improve the runtime to $O(k \cdot \min_i |R_i|)$.

22.3 Tries

A trie is a datastructure to store relations. It is a rooted unraked node labeled tree with one layer for each attribute. Each leaf of represents a tuple that is constructed by reading the node labels from the root to the leaf. In Figure 22.1 we depicted an example relation together with its trie representation.

A_1	A_2	A_3	A_4
1	1	2	3
1	1	2	5
1	1	4	1
1	1	4	7
1	2	3	1
1	2	4	7
2	4	3	7

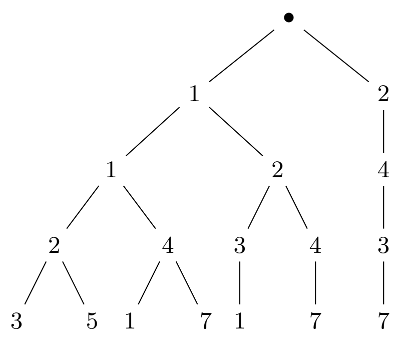


Fig. 22.1: Relation with Trie representation

22.4 Leapfrog Triejoin

In this section, we introduce the Leapfrog Triejoin algorithm. As for the leapfrog algorithm, we will use an iterator interface. The trie iterator interface extends the linear iterator interface with two additional methods for navigating up- and downwards in the trie.

The trie iterator interface has 5 methods:

$$\left.\begin{array}{l}\textbf{key()}\\ \textbf{next()}\\ \textbf{seek(key)}\end{array}\right\}\text{ as in linear iterator (see explanation in text)}$$

open()	move iterator before the first child of the current node
up()	go to the parent of the current node

Intuitively, the functions open and up move the iterator down and up in the trie, while the operations next and seek move the iterator to the right. After open has been called on a node v, the functions, key, next, and seek work like a linear iterator for the unary relation

$$\pi_{A_\ell}(\sigma_{A_1=a_1,\ldots,A_{\ell-1}=a_{\ell-1}}(R)) ,$$

where ℓ is the level in the trie after calling open, $A_1, \ldots, A_\ell$ are the attributes represented at levels $1, \ldots, \ell$ and $a_1, \ldots, a_{\ell-1}$ are the labels on the path from the root to v. Put differently, next, seek, and key work like a linear operator that iterates over the labels of the children of v.

The Leapfrog Triejoin algorithm is depicted as Algorithm 9. Again, the algorithm is implicit by giving an implementation for the iterator interface.

Again, we skip the correctness proof and only look at the runtime. We assume that the operations key, next(), open(), and up() on the underlying tries are implemented to use constant time and seek() takes time at most logarithmic in the size of the relation. These assumptions are reasonable if an appropriate index exists in the database system.

One can show that the worst case running time is $O(M \cdot \log(\max_i |R_i|))$, where M is the AGM bound. If one uses hash tables with amortized constant time for seek, the runtime drops to $O(M)$, which is worst case optimal, as the output trie can have size $\Omega(M)$ by Theorem AGM.

The idea for the proof on the runtime is to look at each node in the output trie. To compute the children, one has to do a leapfrog search on the iterators of the input relations that have the attribute corresponding to the current level in the output trie. Summing up the number of calls to seek() done in all those leapfrog searches, yields the AGM bound in the worst case.

Algorithm 9 Leapfrog Triejoin

```
Input: trie iterators $J_0, \dots, J_{k-1}$ for relations $R_0, \dots, R_{k-1}$ that all have the at-
  tributes ordered according to the same total order $A_1 < A_2 < \cdots < A_m$
Output: linear iterator for $R_0 \bowtie \dots \bowtie R_{k-1}$
  function open()
     $\ell :- \ell + 1$
     $\mathcal{I}_{A_\ell} :- \{I_i \mid A_\ell \in \mathrm{sort}(R_i)\}$        ▷ Iterators for all relations containing $A_\ell$
     for $I \in \mathcal{I}_{A_\ell}$ do
        $I$.open()
     $J_{A_\ell} :-$ leapfrog$(\mathcal{I}_{A_\ell})$        ▷ Iterator for the join of unary relations
  function up()
     for $I \in \mathcal{I}_{A_\ell}$ do
        $I$.up()
     $\ell :- \ell - 1$
  function next()
     $J_{A_\ell}$.next()
  function seek(value)
     $J_{A_\ell}$.seek(value)
  function key()
     $J_{A_\ell}$.key()
  function leapfrog-triejoin$(J_0, \dots, J_{k-1})$        ▷ Constructor for leapfrog-triejoin
  iterator
     $(I_0, \dots, I_{k-1}) :- (J_0, \dots, J_{k-1})$
     $\ell :- 0$
     return this
```

Comments and Exercises for Part III

Comments

The consistency algorithm for evaluating acyclic CQs is called in such a way due to its close relationship with the family of consistency algorithms often applied for solving constraint satisfaction problems. In the database literature similar algorithms are often referred to by the name of *full reducers*.

Exercises

1. Based on Proposition 17.4, design a polynomial time algorithm that computes a join tree of an acyclic CQ q.
2. Prove that $q(D) \ltimes q'(D)$, given $q(D)$ and $q'(D)$, can be computed in time $O(|q(D)| + |q'(D)|)$.
3. Extend Yannakakis's algorithm to show that the set $q(D)$, for $q(\bar{x})$ an acyclic CQ and D a database, can be computed in time $O(|D| \cdot |q| \cdot |q(D)|)$. In addition, if the set of free variables of q is contained in at least one node of the join tree of q, then the latter can be improved to $O(|D| \cdot |q|)$.
4. Complete the proofs of Propositions 17.9 and 17.10.
5. Prove that there are CQs with arbitrarily many atoms that are not acyclic, yet its core is acyclic.
6. While the class of CQs whose core is acyclic defines an "island of tractability" for CQ evaluation, checking membership into such an island is an intractable problem. In particular, checking whether a CQ has an acyclic core is NP-complete (this is in stark contrast with acyclicity recognition, which can be solved in linear time). You are asked to prove this fact.
7. Let q, q' be boolean CQs and D a database. Prove that:

$$q \to q' \text{ and } \textsc{Consistency}(q', D) \neq \texttt{fail} \implies \textsc{Consistency}(q, D) \neq \texttt{fail}.$$

8. Acyclicity and LOGCFL.
9. Consider a restricted version of the notion of hypertree decomposition in which $\lambda(t)$ always corresponds to the set of variables in q that are mentioned in atoms $\chi(t)$. Show that the width of a CQ under this restricted notion can be larger than its generalized hypertreewidth.
10. Complete the proofs of Propositions 18.3 and 18.4.
11. In the proof of Theorem 18.10, show that $Q_{\lambda(s)}(D) = q'_{S(X_s)}(D)$ for each node $s \in T$.

12. *Hypertreewidth*

    There is a slight restriction of the notion of generalized hypertreewidth, known simply as *hypertreewidth*, for which hypertree decompositions can be computed in polynomial time. Formally, a *hypertree decomposition* is a generalized hypertree decomposition (T, χ, λ) that in addition satisfies the following technical condition for each node $s \in T$:

$$\chi(T_s) \cap \text{vars}(\lambda(s)) \subseteq \chi(s),$$

    where $\chi(T_s)$ denotes the set of variables that appear in the sets $\chi(s')$, for s' a descendant of s in T, and vars$(\lambda(s))$ is the set of variables mentioned in the atoms in $\lambda(s)$. Intuitively, this expresses that the variables mentioned in $\lambda(s)$ but not in $\chi(s)$ do not reappear when we walk down the tree in the sets of the form $\chi(s')$.
    The width of a CQ q is then the minimum width of its hypertree decompositions. We define $\mathsf{HW}(k)$, for $k \geq 1$, as the set of CQs whose hypertreewidth is bounded by k. The following result establishes the good properties of the notion of bounded hypertreewidth:

    Theorem 22.1. *Fix $k \geq 1$. There is a polynomial time algorithm that takes as input a CQ q and does the following:*

    a) It checks whether $q \in \mathsf{HW}(k)$.
    b) If the latter is the case, it computes a hypertree decomposition of q of width k. The number of nodes in this hypertree decomposition is bounded by the number of variables in q.

    More in particular, this algorithm runs in time $O(||q||^{2k})$.

    This shows a stark difference with the notion of bounded generalized hypertreewidth, as based on Theorem 18.8 no analogous efficient algorithm can exist for such a notion (under complexity assumptions). We leave the proof of Theorem 22.1 as an interesting exercise for the reader (see Exercise 12).

The drawback, on the other hand, of the notion of hypertree with respect to generalize hypertreewidth is that for a given CQ q the former might be larger than the former. Still, not arbitrarily larger, as the next result establishes:

Theorem 22.2. *Fix $k \geq 1$. If a CQ q has generalized hypertreewidth k, then it has hypertreewidth at most $3k+1$.*

For CQs that appear in practice, however, the values of their hypertreewidth and generalized hypertreewidth usually coincide.
Prove Theorem 22.1.

13. Fractional hyperteewidth, submodular width?
14. Prove Lemma 19.5.
15. Let $q(\bar{x})$ be a CQ and $q^{\downarrow}(\bar{x})$ the CQ that is obtained from $q(\bar{x})$ by adding a new atom $R(\bar{x})$ that contains all its free variables, where R is a fresh relation symbol. For a class $\mathcal{C}$ of CQs we denote by $\mathcal{C}^{\downarrow}$ the class $\{q^f(\bar{x}) \mid q(\bar{x}) \in C\}$.
Assume that FPT $\neq$ W[1]. Prove that the following are equivalent for every recursively enumerable class $\mathcal{C}$ of (not necessarily boolean) CQs such that there is a bound on the arity of the relation symbols that are mentioned by CQs in $\mathcal{C}$:
    a) Evaluation for $\mathcal{C}$ can be solved in polynomial time.
    b) Evaluation for $\mathcal{C}$ is FPT.
    c) $(\mathcal{C}^{\downarrow})_{\text{core}}$ has bounded treewidth.
16. Complete the missing proofs of all statements in Example 20.2. In particular, prove the following: (1) q_1 and q_2 are cores, (2) $q \to q_1$ and $q \to q_2$, (3) both q_1 and q_2 are $\mathsf{GHW}(1)$-approximations of q, and (4) $q_1 \not\equiv q_2$.
17. Lack of quantitative guarantees for approximations.
18. Overapproximations.
19. AGM bound on other CQs: Cliques, Loomis Whitney.
20. AGM bound on arbitrary CQs.

Part IV

Expressive Languages

23

Adding Union

The first, and simplest, addition to conjunctive queries is *union*. Queries from the class UCQ (unions of conjunctive queries) are queries of the form $q_1 \cup \ldots \cup q_n$, where $q_1, \ldots, q_n$ are all CQs, of the same arity. We first look at equivalent ways of defining them.

Existential positive formulae

Recall that the class $\exists\text{FO}^+$ of existential positive formulae is defined as the restriction of FO to conjunction, disjunction, and existential quantification. Both relational atoms $R(\bar{x})$ and equational atoms $x = y$ are permitted.

Queries defined by formulae from this class have precisely the power of UCQs. Of course every UCQ can be defined by an $\exists\text{FO}^+$ formula: simply express every CQ with $\exists$ and $\wedge$ and use disjunction to express their union. To prove the opposite, we transform an $\exists\text{FO}^+$ formula to a UCQ in a number of steps. First, we propagate disjunction by using simple rules:

- $\chi \wedge (\varphi \vee \psi) \rightsquigarrow (\chi \wedge \varphi) \vee (\chi \wedge \psi)$
- $\exists x\, (\varphi \vee \psi) \rightsquigarrow \exists x\, \varphi \vee \exists x\, \psi$.

As the result, starting with an $\exists\text{FO}^+$ formula $\varphi(\bar{x})$, we have a formula which is a disjunction $\varphi_1(\bar{x}_1) \vee \ldots \vee \varphi_n(\bar{x}_n)$, where each $\varphi_i(\bar{x}_i)$ is a formula that is built up from atomic formulae using $\exists$ and $\wedge$. Note that free variables of the disjuncts need not be $\bar{x}$. We now – temporarily – add a new predicate $\mathsf{dom}(x)$ which is true, in a database D, for every element of the domain $\text{Dom}(D)$. With this predicate we can view each formula $\varphi_i(\bar{x}_i)$ as a formula $\varphi_i(\bar{x})$, simply by adding conjunctions with all $\mathsf{dom}(y)$ for variables y which are in $\bar{x}$ but not in $\bar{y}$. After this transformation, the output of φ is then the same as the union of outputs of the φ_is.

To give an example, consider an $\exists\text{FO}^+$ formula $\varphi(y, z) = \exists x\, \big(R(x, y) \vee (S(x, z) \wedge x = z)\big)$. Propagating disjunction, we obtain $\exists x\, R(x, y) \vee \exists x\, (S(x, z) \wedge x = z)$. These disjuncts can be viewed as formulae in free variables y, z:

$$\varphi_1(y,z) = \exists x\,\big(R(x,y) \wedge \mathsf{dom}(z)\big) \qquad \varphi_2(y,z) = \exists x\,\big(S(x,z) \wedge x = z \wedge \mathsf{dom}(y)\big)\,.$$

This, we need to show that each formula $\varphi_i(\bar{x})$ is equivalent to a UCQ. These formulae start with an existential quantifier $\exists\bar{y}$ that is followed by a conjunction of relational atoms, equality atoms, and formulae $\mathsf{dom}(u)$. First, note that with the help of $\mathsf{dom}(\cdot)$, we can eliminate equality atoms. Indeed, for each atom $u = v$, we replace v by u everywhere, and add $\mathsf{dom}(u)$ to the conjunction. To see why the latter is needed, consider, for example, $\psi(u,v) = (u = v)$. We cannot just throw away the equality; instead this formula is equivalent to $\psi(u,u) = \mathsf{dom}(u)$.

The final step is to eliminate $\mathsf{dom}(\cdot)$ predicates, with the help of union. Let $\varphi(\bar{x}) = \exists\bar{y}\,\alpha(\bar{x},\bar{y}) \wedge \mathsf{dom}(u)$, where α is an arbitrary formula. Then it is equivalent to the following disjunction:

$$\bigvee_{R\in\mathbf{S}}\ \bigvee_{i\le \mathrm{ar}(R)} \exists\bar{y}\,\exists z_1 \ldots \exists z_{\mathrm{ar}(R)-1}\,\alpha(\bar{x},\bar{y}) \wedge R\big(z_1,\ldots,z_{i-1},u,z_i,\ldots,z_{\mathrm{ar}(R)-1}\big)$$

since, if a an element is in $\mathrm{Dom}(D)$, it must appear in a tuple in some relation R, in some position $i \le \mathrm{ar}(R)$. Using this transformation, we eventually get rid of all the formulae $\mathsf{dom}(u)$ and obtain a disjunction of formulae where, after existential quantifiers, we only have a conjunction of relational atoms, i.e.., CQs.

Thus, every $\exists\mathrm{FO}^+$ formula is equivalent to a disjunction (or union) of CQs. The transformation from an $\exists\mathrm{FO}^+$ formula to a union of conjunctive queries can be costly however. Consider, for instance, an $\exists\mathrm{FO}^+$ formula $(q_1 \vee q_i') \wedge \ldots \wedge (q_n \vee q_n')$, where all the q_is and q_i's are CQs, for $1 \le i \le n$. Representing it as a UCQ requires transforming a conjunctive normal form formula into disjunctive normal form, resulting in a disjunction of 2^n CQs. Consequently, even though $\exists\mathrm{FO}^+$ and UCQs have the same power, some problems related to them will have different complexity (in case when the size of the query matters).

Positive relational algebra

Recall that each CQ can be written as an SPJ query of relational algebra, i.e., only with operations of selection, projection, and Cartesian product (or join). Thus, their unions can be written in *positive relational algebra* RA^+, which contains all operations of RA except negation, i.e., selection, projection, Cartesian product, and union. The converse is true too: every RA^+ query is a union of conjunctive queries. This is because, as for $\exists\mathrm{FO}^+$, union can be propagated through other operations to become the outermost operation:

- $\sigma_\theta(e_1 \cup e_2) \rightsquigarrow \sigma_\theta(e_1) \cup \sigma_\theta(e_2)$;
- $\pi_\alpha(e_1 \cup e_2) \rightsquigarrow \pi_\alpha(e_1) \cup \pi_\alpha(e_2)$;
- $e_1 \times (e_2 \cup e_3) \rightsquigarrow (e_1 \times e_2) \cup (e_1 \times e_3)$.

The translation again can blow up the query exponentially: for example, if all q_i, q_i's are CQ s, for $1 \leq i \leq n$, then $(q_1 \cup q_1') \times \cdots \times (q_n \cup q_n')$ is the union of 2^n CQs.

Summing up, we have

Theorem 23.1. *Unions of conjunctive queries, existential positive queries of first-order logic, and positive relational algebra queries are all equally expressive.*

At the same time, transforming a query in one of those classes into another can increase the size of the query exponentially, so not all complexity results about one class will automatically apply to the others.

Query evaluation

A general rule of thumb is that whatever is true about evaluation of CQs, is true about evaluation of their unions. We consider two examples of it.

First, evaluation of UCQs is NP-complete in combined complexity. Indeed, we know that even evaluation of CQ s can be NP-hard, so we need to show the upper bound. Let $q = q_1 \cup \ldots \cup q_n$, and assume we are given a database D and a tuple $\bar{a}$ of the same arity as q. To check if $\bar{a} \in q(D)$, we simply check that there is an $i \in \{1, \ldots, n\}$ such that $\bar{a} \in q_i(D)$. For this, it suffices to guess such an $i \in \{1, \ldots, n\}$ and a homomorphism from q_i to $(D, \bar{a})$, and thus the whole algorithm has NP complexity.

Second, a Boolean acyclic UCQ q can be evalued in time $O\big(||D|| \cdot ||q||\big)$ on a database D. By acyclic UCQ we mean a union $q = q_1 \cup \ldots \cup q_n$, where each q_i is an acyclic CQ. Indeed, as we know, each q_i can be evaluated in time $O\big(||D|| \cdot ||q_i||\big)$, and thus q can be evaluated in time

$$O\Big(||D|| \cdot \sum_{i=1}^{n} ||q_i||)|\Big) \;=\; O\big(||D|| \cdot ||q||\big)\,.$$

Query containment and minimization

For UCQs, testing query containment can be reduced to testing containment of CQs, that we know how to do (see Chapter 15). Recall that a query q is contained in q', written $q \subseteq q'$, iff $q(D) \subseteq q(D')$ for every database D. For UCQs, the general principle of testing containment is the same for all their presentations – as UCQs, or $\exists\mathrm{FO}^+$, or RA^+ queries – but the complexity depends on how queries are given. We start with UCQs.

Theorem 23.2. *Let $q = q_1 \cup \ldots \cup q_n$ and $q' = q_1' \cup \ldots \cup q_m'$ be two* UCQ*s, where all of q_is and q_j's are* CQ*s. Then $q \subseteq q'$ iff for every $i \in [1, n]$, there is $j \in [1, m]$ such that $q_i \subseteq q_j'$.*

Proof. It is clear that if the condition of the theorem is satisfied, then $q \subseteq q'$; thus we prove the opposite direction. Suppose that $\bar{x}_i$ and $\bar{x}'_j$ are the tuple of free variables of q_i and q'_j, respectively, for each $i \in [1, n]$ and $j \in [1, m]$. Assume $q \subseteq q'$. Choose an arbitrary $i \in [1, n]$ and consider D_{q_i}, the canonical database of q_i. Then $\bar{x}_i \in q_i(D_{q_i})$, and by containment, $\bar{x} \in q'(D_{q_i})$ as $q_i \subseteq q'$. Therefore, there is some $j \in [1, m]$ so that $\bar{x}_i \in q'_j(D_{q_i})$, and thus there is a homomorphism $(q'_j, \bar{x}'_j) \to (q_i, \bar{x}_i)$ (cf. the description of evaluation of CQs via homomorphisms in Chapter 26, in particular (13.2)). By the containment criterion for CQs this tells us that $q_i \subseteq q'_j$. □

As a corollary we can now pinpoint the complexity of containment for UCQs.

Corollary 23.3. *The containment problem for* UCQ*s is* NP*-complete.*

Proof. The NP-hardness of containment is already known for CQs (see Chapter 15). To show that the problem is in NP, assume as in Theorem 23.2, that $q = q_1 \cup \ldots \cup q_n$ and $q' = q'_1 \cup \ldots \cup q'_m$. Then, to check whether $q \subseteq q'$, by Theorem 23.2, we need to guess pairs $(1, j_1), \ldots, (n, j_n)$, with $j_1, \ldots, j_n \in [1, m]$, and homomorphisms $(q'_{j_1}, \bar{y}_{j_1}) \to (q_1, \bar{x}_1)$, …, $(q'_{j_n}, \bar{y}_{j_n}) \to (q_n, \bar{x}_n)$. The size of the guess is polynomial, and it takes polynomial time to check its correctness, showing that containment is in NP. □

Another corollary of Theorem 23.2 gives us a minimization procedure for UCQs. Given a UCQ q, its *minimization* is a UCQ q' that is equivalent to q, and that has the smallest number of atoms among all UCQs equivalent to q. The algorithm to construct it is shown below.

Algorithm 10 MinimizeUCQ(q)

```
Input: A UCQ $q = q_1 \cup \ldots \cup q_n$
Output: $q'$ is a minimization of $q$
1: for $i = 1, \ldots, n$ do
2:     compute the core $q_i^*$ of $q_i$
3: $\mathcal{Q} = \{q_1^*, \ldots, q_n^*\}$
4: while there are $i \neq j$ such that $q_i^* \subseteq q_j^*$ do
5:     remove $q_i^*$ from $\mathcal{Q}$
   return $q' = \bigcup_{q^* \in \mathcal{Q}} q^*$
```

Corollary 23.4. *Every minimization of a* UCQ Q *is equivalent to the output of* MinimizeUCQ(Q).

Proof. Let $q'_1 \cup \ldots \cup q'_m$ be the output of MinimizeUCQ(Q), and assume that we have another minimization $q''_1 \cup \ldots \cup q''_k$. Then each q''_i is a core (if not, by taking its core we obtain a smaller equivalent query), and $q''_i \subseteq q''_j$ for no distinct $i, j \in [1, k]$ (otherwise removing q''_i we get a smaller equivalent query). Consider an arbitrary q'_i; by Theorem 23.2 we have $q'_i \subseteq q''_j$ for some $j \in [1, k]$, and, $q''_j \subseteq q'_l$ for some $l \in [1, m]$ again by Theorem 23.2. Algorithm MINIMIZEUCQ(Q) then ensures $l = i$, and thus $q'_i \equiv q''_j$. Hence, for each $i \in [1, m]$, there is $j \in [1, k]$ such that $q'_i \equiv q''_j$. Applying the same argument to $q''_1 \cup \ldots \cup q''_k$ we conclude that for each $j \in [1, k]$, there is $i \in [1, m]$ such that $q''_j \equiv q'_i$. Thus, the sets $\{q'_1, \ldots, q'_m\}$ and $\{q''_1, \ldots, q''_k\}$ are the same, since queries in them are pairwise incomparable, and since all queries in them are cores, it means they are the same up to renaming of variables. $\square$

When a different syntactic presentation of UCQs is used, the basic principle behind query containment remains the same, but the complexity may actually go up. We now illustrate this with RA$^+$ queries; the treatment for $\exists$FO$^+$ is essentially the same as is left as an exercise (see Exercise 3).

With each RA$^+$ query q, we associate a set of queries CQ(q) such that each $q' \in$ CQ(q) is an SPJ query. This is done by induction on the structure of q; essentially for every union $q_1 \cup q_2$ that occurs, we look at possible ways of resolving this union, i.e., choosing q_1 or q_2. Formally,

- $\mathrm{CQ}(R) = \{R\}$, if R is a relation name;
- $\mathrm{CQ}(\sigma_\theta(q)) = \{\sigma_\theta(q') \mid q' \in \mathrm{CQ}(q)\}$;
- $\mathrm{CQ}(\pi_\alpha(q)) = \{\pi_\alpha(q') \mid q' \in \mathrm{CQ}(q)\}$;
- $\mathrm{CQ}(q_1 \times q_2) = \{q'_1 \times q'_2 \mid q'_1 \in \mathrm{CQ}(q_1) \text{ and } q'_2 \in \mathrm{CQ}(q_2)\}$;
- $\mathrm{CQ}(q_1 \cup q_2) = \mathrm{CQ}(q_1) \cup \mathrm{CQ}(q_2)$.

The following is easily shown by induction on RA$^+$ queries.

Proposition 23.5. *If q is an RA$^+$ query, then every $q' \in \mathrm{CQ}(q)$ is an* SPJ *query, and $q = \bigcup_{q' \in \mathrm{CQ}(q)} q'$.*

This gives us an algorithm for testing $q_1 \subseteq q_2$, when q_1, q_2 are RA$^+$ queries: we construct sets CQ(q_1) and CQ(q_2), and then for each $q'_1 \in$ CQ(q_1) check whether exists $q'_2 \in$ CQ(q_2) such that $q'_1 \subseteq q'_2$. The latter test is of course a containment test for CQs; thus, by Theorem 23.2, this provides a test for containment of RA$^+$ queries.

What is the complexity of such an algorithm? The way it is naïvely presented above, we need to construct sets CQ(q_1) and CQ(q_2) of exponential size, and then run an NP algorithm testing $q'_1 \subseteq q'_2$ exponentially many times, for each $q'_1 \in$ CQ(q_1). This gives an exponential time upper bound. While such a bound is unavoidable under complexity assumptions (to start with, the test for CQ containment is already NP-complete), we can pinpoint the complexity of the problem with more accuracy.

Note that finding a query $q' \in$ CQ(q) amounts to "resolving" each union in q. That is, think of a parse-tree of the expression for q, and every time a union node occurs, with two subtrees under it, drop one of the subtrees of that union node. If union occurs k times in q, this gives us 2^k expressions that can result from resolving those union nodes, as each one gives us two choices. Now, if q_1, q_2 are RA$^+$ queries, then we need to check whether for each way of resolving unions in the parse tree of q giving us an SPJ query q_1', there exists a way of resolving unions in q_2 producing a query q_2', together with a witness of $q_1' \subseteq q_2'$. Such a witness is simply a homomorphism $(q_2', \bar{x}) \to (q_1', \bar{x})$ if we write q_1' and q_2' as CQs. Thus, the structure of the problem is this: for every guess of polynomial size, there is another guess of polynomial size, such that a certain polynomial-time algorithm says yes. This is precisely the structure of Π_2^p problems. In fact, this complexity class reflects the exact complexity of containment for RA$^+$ queries.

Theorem 23.6. *Testing containment of RA$^+$ queries is Π_2^p-complete.*

Above we have explained how to obtain membership in Π_2^p; to prove Π_2^p-hardness, one can provide a reduction from the $\forall\exists$SAT problem; see Exercise 2 for details.

Theorem 23.6 also implies that equivalence of RA$^+$ is Π_2^p-complete. Indeed, $q_1 \equiv q_2$ iff both $q_1 \subseteq q_2$ and $q_2 \subseteq q_1$ hold, and $q_1 \subseteq q_2$ iff $\pi_\alpha(q_1 \bowtie_\theta q_2) \equiv q_1$, where, assuming that q_1, q_2 are of arity m, the condition θ is the conjunction of $1 = m+1$, $2 = m+2$, $\ldots$, $m = 2m$, and $\alpha = (1, \ldots, m)$.

Preservation under homomorphisms

We have seen already that CQs are preserved under homomorphisms: if $\bar{a} \in q(D)$ and $h : D \to D'$ is a homomorphism, then $h(\bar{a}) \in q(D')$. It is immediate that this extends to UCQs: if $\bar{a} \in q_1(D) \cup \ldots \cup q_n(D)$, where all the q_is are CQs, then $\bar{a}$ is in one some $q_i(D)$ for $i \leq n$, and thus $h(\bar{a}) \in q_i(D') \subseteq q_1(D') \cup \ldots \cup q_n(D')$. It is far more remarkable that the converse is true: FO queries preserved under homomorphisms are exactly the UCQs.

Theorem 23.7. *If q is a query expressible in FO that is preserved under homomorphisms, then it is equivalent to a* UCQ.

This is a very deep result whose proof is out of the scope of this book, but it is very important and found many applications in database theory. It is relevant to understand that in logic, there are two versions of homomorphism preservation: one applies to arbitrary structures (finite and infinite), and the other only to finite structures. In other words, if we have an arbitrary FO formulae $\varphi(\bar{x})$, then it is preserved under homomorphisms

- *on all structures* if, for every two structures D and D', finite or infinite, and a homomorphism $h : D \to D'$, we have $D' \models \varphi(h(\bar{a}))$ whenever $D \models \varphi(\bar{a})$;

- *on finite structures* if, for every two finite structures D and D' and a homomorphism $h : D \to D'$, we have $D' \models \varphi(h(\bar{a}))$ whenever $D \models \varphi(\bar{a})$.

Many preservation results known in logic are true on all structures, but fail on finite structures – which is of course what is of interest to us in database theory. Theorem 23.7 is a rare exception: the result is true for either flavor of preservation under homomorphisms.

24

Adding Negation

We have already seen that the class of UCQs has the same expressive power as the class of existential positive formulae, $\exists\mathrm{FO}^+$. Formulae of $\exists\mathrm{FO}^+$ are built up from atomic formulae using conjunction, disjunction, and existential quantification. Adding negation to them gives us the full power of first-order logic FO. Indeed, universal quantification can be expressed: $\forall x\, \varphi$ is equivalent to $\neg\exists x\, \neg\varphi$.

We already know a few facts about FO. In terms of expressive power, it is equivalent to the full relational algebra RA. Thus, adding negation in logic is the same as adding the difference operation in RA. FO query evaluation is PSpace-complete in combined complexity and is in DLogSpace in data complexity, the latter matching the complexity of UCQs. Unlike for CQs and UCQs, static analysis is undecidable for FO queries: we saw that the problem FO-Containment is undecidable.

We can also see, very easily, that adding negation (or difference) makes the language more powerful. Indeed, we know that UCQs are monotone, but FO queries need not be: for example, the query $\varphi(x) = R(x) \wedge \neg S(x)$ computing the difference of R and S is not monotone: adding elements to S can make the difference of R and S smaller. Thus, this query is not a equivalent to a UCQ.

The power of FO: an easy bound

Crucially, FO allows us to express queries with universal quantification, for example, "find students that take all courses offered in their year" or "find customers that shop at every branch in the city" – these are easily expressible with the availability of $\forall$. As we saw before, $\forall$ can be expressed with $\exists$ and two negations, and indeed practical languages, such as SQL, do not offer explicit universal quantification but instead express universal queries via negated existential statements.

We are now more interested in the limitations of the language, as they will justify what practical languages need to add on top of FO. Two results that we show next explain that FO cannot count: it cannot compare cardinalities of relations, nor can it express nontrivial statements about cardinalities of sets (e.g., is such cardinality even?).

We start with the simplest case, when the schema contains a single unary relation U, i.e., a set. Note that if we have two databases D_1 and D_2 with $|U^{D_1}| = |U^{D_2}|$, then $D_1 \models \varphi$ iff $D_2 \models \varphi$ for every FO sentence φ – this is because such databases are isomorphic. We can thus define

$$\mu_n(\varphi) = \begin{cases} 1 & \text{if } D \models \varphi, \text{ when } |U^D| = n \\ 0 & \text{otherwise}. \end{cases} \tag{24.1}$$

Theorem 24.1. *For every FO sentence φ over a single unary relation U, there exists $k \in \mathbb{N}$ such that either $\mu_n = 1$ for all $n \geq k$, or $\mu_n = 0$ for all $n \geq k$.*

To prove this, we need a few basic notions and facts from logic. A *theory* T is a set of sentences of the same vocabulary. A structure is a *model* of a theory if it satisfies every sentence of T. A theory is *consistent* if it has a model. If the vocabulary is finite (or even countable), then a consistent theory has a countable model (this is known as the Löwenheim-Skolem theorem). A sentence φ is a consequence of T if every model of T satisfies φ. If φ is a consequence of T, then it is a consequence of some finite subset $T_0 \subseteq T$ (this is compactness theorem for FO).

Proof. First notice that the contents of relation U^D are the same as $\text{Dom}(D)$. Hence every occurrence of $U(x)$ in a formula can be replaced with `true`, and thus we can assume that every FO query uses only equality atoms. Now consider a theory $T = \{\lambda_n \mid n \in \mathbb{N}\}$, where λ_n says that there are n distinct elements: $\exists x_1 \ldots x_n \bigwedge_{1 \leq i < j \leq n} \neg(x_i = x_j)$. Clearly T is consistent: any infinite set is a model. Now consider an arbitrary sentence φ. Then either φ or $\neg\varphi$ is a consequence of T. Indeed, if neither is, then $T \cup \{\varphi\}$ and $T \cup \{\neg\varphi\}$ are consistent theories, and thus have countable models. But there is only one countable model, up to isomorphism, just a countable set, and it cannot satisfy both φ and $\neg\varphi$.

Assume now that φ is a consequence of T. By compactness, there is a finite subset $T_0 \subset T$ such that φ is a consequence of T_0. Let λ_k be the sentence with the largest index k in T_0; then φ is a consequence of λ_k (since λ_m is a consequence of λ_k if $m \leq k$), and thus φ is true in D whenever $|U^D| \geq k$. Hence $\mu_n(\varphi) = 1$ for all $n \geq k$. If on the other hand $\neg\varphi$ is a consequence of T, then applying the same argument to $\neg\varphi$ shows that $\mu_n(\varphi) = 0$ for all n greater than some fixed $k \in \mathbb{N}$. $\square$

This result shows that only very simply properties of cardinalities of sets can be expressed in FO. Let $C \subseteq \mathbb{N}$. Define a Boolean query Q_C on databases with a single unary relation R as follows: Q_C is true in D iff $|U^D| \in C$.

Corollary 24.2. *The query Q_C is expressible in FO iff either C, or $\mathbb{N} - C$, is a finite set.*

For example, it is impossible to test in FO if the cardinality of a set is even, or more generally divisible by some number $n \in \mathbb{N}$.

Zero-One Law

Now consider again the definition (24.1) and reformulate it slightly. We assume that elements of the domain are numbers in $\mathbb{N}$, to be able to enumerate them. Then

$$\mu_n(\varphi) = \frac{|\{D \mid \mathrm{Dom}(D) = [1,n] \text{ and } D \models \varphi\}|}{|\{D \mid \mathrm{Dom}(D) = [1,n]\}|}\,. \tag{24.2}$$

In other words, $\mu_n(\varphi)$ is the proportion of databases with $\mathrm{Dom}(D) = [1, n]$ that satisfy φ. Indeed, there is only one D with the schema that consists of a single unary relation that satisfies $\mathrm{Dom}(D) = [1, n]$. Theorem 24.1 then says that this sequence eventually stabilizes, i.e.,

$$\lim_{n\to\infty} \mu_n(\varphi) \;\in\; \{0,1\}\,.$$

Notice that definition (24.2) applies to arbitrary schemas, not only to those with a single unary relation. The intuition behind the quantity $\mu_n(\varphi)$ can be described in purely probabilistic terms. Consider the finite set of databases with $\mathrm{Dom}(D) = [1, n]$. Then $\mu_n(\varphi)$ is the probability that a database one picks uniformly at random from this set satisfies φ. By taking the limit we describe the asymptotic behavior of this sequence; intuitively, it defines the probability that a randomly picked database satisfies φ.

Definition 24.3 (0–1 law). *If $\lim_{n\to\infty} \mu_n(\varphi) \in \{0,1\}$ for every sentence of a language, then it has the* 0–1 law.

If a logic has the 0–1 law, then every property expressible in it is either true for almost all databases, or false for almost all databases. Not all properties of course are such. For instance, in our previous example of a schema with one unary relation U, consider the property EVEN which is true iff $|U^D|$ is an even number. Then $\mu_n(\text{EVEN})$ is 1 when n is even and 0 when n is odd, and thus the limit does not even exist.

A fundamental result on the expressiveness of first-order logic states the following.

Theorem 24.4. *FO has the 0–1 law.*

Proof. We shall not prove the result in its full generality, but instead consider a special case when we have two unary relations, A and B. This case builds on the proof of Theorem 24.1, illustrates key elements in the proof of the 0–1 law, and allows us to derive corollaries about cardinality comparisons.

These key elements are the same in the general proof of the 0–1 law, namely, we construct a theory T such that:

- $\lim_{n\to\infty} \mu_n(\alpha) = 1$ for each $\alpha \in T$, and
- T has a unique, up to isomorphism, countable model.

The second condition, and the same argument as in the proof of Theorem 24.1 show that for every FO sentence φ, either φ or $\neg\varphi$ is a consequence of T.

With such a construction, the proof of the 0–1 law easily follows. Take any sentence φ, and assume that φ is a consequence of T. Then, by compactness, it is a consequence of a finite subset of $\{\alpha_1, \ldots, \alpha_m\}$ of sentences of T. Since $\lim_{n\to\infty} \mu_n(\alpha_i) = 1$ for all $i \in [1, m]$ by the first condition above, we have $\lim_{n\to\infty} \mu_n(\bigwedge_{i=1}^m \alpha_i) = 1$ and thus $\lim_{n\to\infty} \mu_n(\varphi) = 1$ since $\mu_n(\varphi) \geq \mu_n(\bigwedge_{i=1}^m \alpha_i)$. When $\neg\varphi$ is a consequence of T, we apply the above proof to $\neg\varphi$ and conclude $\lim_{n\to\infty} \mu_n(\neg\varphi) = 1$ and thus $\lim_{n\to\infty} \mu_n(\varphi) = 0$.

Now we construct a theory T satisfying propertes 1 and 2 above. It has sentences $\lambda_k(AB), \lambda_k(A\bar{B})$, and $\lambda_k(\bar{A}B)$ expressing that $A \cap B$, $A - B$, and $B - A$ respectively have at least k elements, for each $k > 0$. They are the same as sentences λ_k we saw in the proof of Theorem 24.1, but now we need to add conjuncts $A(x_i) \wedge B(x_i)$ for $\lambda_k(AB)$, and $A(x_i) \wedge \neg B(x_i)$ for $\lambda_k(A\bar{B})$, and $\neg A(x_i) \wedge B(x_i)$ for $\lambda_k(\bar{A}B)$. Furthermore, T has a sentence $\forall x\, A(x) \vee B(x)$. This is true in every finite database since quantification is over the domain $\text{Dom}(D)$ which is $A^D \cup B^D$, as these are the only relations in the schema. This theory is consistent. Indeed, it has a countable model in which sets A and B contain all elements of the universe, and all of $A \cap B$, $A - B$, and $B - A$ are countable (as an example, we can take $A = \{3n, 3n+1 \mid n \in \mathbb{N}\}$ and $B = \{3n, 3n+2 \mid n \in \mathbb{N}\}$). Notice that any two such structures are isomorphic, and thus up to isomorphism T has only one countable model, satisfying condition 2.

Now we prove that T satisfies the first condition. The sentence $\forall x\, A(x) \vee B(x)$ is true in all finite databases. Next we show that other three sentences are true in almost all databases. We do this as an illustration for $\lambda_k(AB)$. Consider instead its negation saying that $|A \cap B| < k$. Let the domain of the database, i.e., $A \cup B$, have n elements. How many ways are there to choose two sets A and B with $|A \cap B| < k$? For each $j < k$, we have $\binom{n}{j}$ ways to choose an intersection $A \cap B$ of this cardinality. Then we need to choose elements of $A - B$ from the remaining $n - j$ elements, and there are 2^{n-j} ways of doing so. The remaining elements must belong to $B - A$ since $\text{Dom}(D)$ is the union of A and B. Thus, there are $\binom{n}{j} \cdot 2^{n-j}$ ways to choose two sets from an n-element set whose intersection has exactly j elements and whose union contains all n elements. This means that there are at most

$$\sum_{j<k} \binom{n}{j} \cdot 2^{n-j} \;\leq\; n^k \cdot 2^n$$

ways of choosing two sets whose intersection has cardinality bounded by k. On the other hand, there are 3^n ways to choose a pair of sets A and B whose union contains all n elements, since for each element there are 3 possibilities: it can belong to $A \cap B$, or $A - B$, or $B - A$. Thus,

$$\mu_n(\neg\lambda_k(AB)) \ \leq \ n^k \left(\frac{2}{3}\right)^n$$

which means $\lim_{n\to\infty} \mu_n(\neg\lambda_k(AB)) = 0$, and therefore $\lim_{n\to\infty} \mu_n(\lambda_k(AB)) = 1$. Proofs for $\lambda_k(A\bar{B})$ and $\lambda_k(\bar{A}B)$ are very similar. This shows that T satisfies properties 1 and 2, and concludes the proof. □

Ideas behind extending this proof to graphs are explained in Exercise 9.

We now consider a cardinality comparison $|A| < |B|$. How many ways are there to pick a pair of sets satisfying the condition from an n-element set? For every pair of sets (A, B) satisfying the condition, the pair (B, A) will not satisfy it; from this, one concludes that $\lim_{n\to\infty}(|A| < |B|) = \frac{1}{2}$. A very similar argument works for $|A| \leq |B|$. Therefore,

Corollary 24.5. *Cardinality comparisons $|A| < |B|$ and $|A| \leq |B|$ are not expressible in FO.*

A 0–1 law argument can also be used to show that the condition $|A| = |B|$ is not expressible in FO either (see Exercise 7 at the end of this part). Thus, FO is too weak to express comparisons of cardinalities of sets, and nontrivial properties of cardinalities (e.g., being even).

Restricted static analysis

The containment problem is undecidable for FO. We know it is decidable for UCQs. Is there a class of queries in between, where a restricted form of negation can be used, and containment is still decidable? There are examples of such classes, the best known of them are CQ$^{\neq}$ and UCQ$^{\neq}$, extensions of CQ s and UCQs with disequality atoms $x \neq y$. For example, the CQ$^{\neq}$ query q :- $E(x, y), x \neq y$ checks if a graph has an edge that is not a loop, i.e., its source and target are different. These are still queries expressible in FO, and thus the formal definition of evaluating such queries uses the standard definition of evaluation of FO formulae.

Evaluation of queries with inequalities can be stated in terms of homomorphisms as well. A CQ$^{\neq}$ query, written as a rule, is of the form

$$q(\bar{y}) \ \text{:-} \ R_1(\bar{x}_1), \ldots, R_m(\bar{x}_m), \alpha(\bar{x})\,. \tag{24.3}$$

It is built from a usual CQ $\hat{q}(\bar{y})$:- $R_1(\bar{x}_1), \ldots, R_m(\bar{x}_m)$ by adding a list of inequalities $\alpha(\bar{x})$. Here $\bar{x} = (x_1, \ldots, x_n)$ is the tuple of all variables used in q, in both the body and the head, and α is the list of inequalities $x_i \neq x_j$

with indexes $i, j \in [1, n]$. Given a database D, a homomorphism h from $\hat{q}$ to $\mathrm{Dom}(D)$ *respects* α if for every inequality $x_i \neq x_j$ in α, we have $h(x_i) \neq h(x_j)$. Then the output of q on D consists of all the tuples $h(\bar{y})$, where h ranges over homomorphisms $\hat{q} \to D$ respecting α. The definition extends to UCQ$^{\neq}$ queries, just by taking unions.

This way of looking at evaluation of UCQ$^{\neq}$ queries lets us make two observations. First, UCQ$^{\neq}$ queries are monotone: if $D \subseteq D'$, then $q(D) \subseteq D'$. Second, they can be evaluated in NP in terms of combined complexity: one simply guesses a mapping h from $\hat{q}$ to $\mathrm{Dom}(D)$ and then checks in polynomial time that it is a homomorphism that respects α.

Furthermore, unlike for arbitrary FO queries, this tame way of adding negation preserves decidability of static analysis.

Theorem 24.6. *The problem* UCQ$^{\neq}$-Containment *is in* Π_2^p.

We prove the result for Boolean CQ$^{\neq}$ queries; it contains all the key ingredients, and the extension to UCQ$^{\neq}$ queries with free variables is left as an exercise (Exercise 10). Recall that for CQs we obtain from the Homomorphism Theorem that containment boils down to evaluation. More precisely, if q and q' are Boolean CQs, then $q \subseteq q'$ if and only if $q'(D[q]) = \mathsf{true}$, i.e., q' holds in the canonical database $D[q]$ of q. For CQ$^{\neq}$ queries, in turn, this reduction still holds but now we have to consider a set of canonical databases, not just a single one.

We now formalize this idea. Assume that a Boolean CQ$^{\neq}$ query q is given by (24.3), without variables $\bar{y}$ in the head. Let H_α^n be the set of all maps $h : \{x_1, \ldots, x_n\} \to \{x_1, \ldots, x_n\}$ that respect α, i.e., $h(x_i) \neq h(x_j)$ if the inequality $x_i \neq x_j$ belongs to α. Let q' be another CQ$^{\neq}$ query. We claim that

$$q \subseteq q' \iff \forall h \in H_\alpha^n : \; q'(D[h(\hat{q})]) = \mathsf{true}\,, \tag{24.4}$$

That is, $q \subseteq q'$ if and only if q' holds over every database of the form $h(\hat{q})$, for $h \in H_\alpha^n$. (Or, more precisely, over the canonical database of $h(\hat{q})$). In a certain sense, then, these are the “canonical” databases for the CQ$^{\neq}$ query q.

We now prove the claim. Assume that $q \subseteq q'$, and consider $h \in H_\alpha^n$. Then $q(h(\hat{q})) = \mathsf{true}$ since $h : \hat{q} \to h(\hat{q})$ is a homomorphism that respects α. By the $q \subseteq q'$ assumption, $q'(h(\hat{q})) = \mathsf{true}$, proving the left-to-right direction of (24.4). Conversely assume that the condition on the right-hand side of (24.4) holds. Let D be a database with $q(D) = \mathsf{true}$. Then there is a homomorphism $h : \hat{q} \to D$ that respects α. In particular, $h(\hat{q}) \subseteq D$ and there is an $h' \in H_\alpha^n$ such that $h(\hat{q})$ and $h'(\hat{q})$ are isomorphic. By assumption, $q'(h'(\hat{q})) = \mathsf{true}$, and hence $q'(h(\hat{q})) = \mathsf{true}$. By monotonicity of UCQ$^{\neq}$ queries, $q'(D) = \mathsf{true}$. This proves $q \subseteq q'$.

Finally it remains to see what the complexity of checking condition in (24.4) is. The first universal quantifier amounts to quantifying over polynomial-size objects (elements of H_α^n, given by n pairs), and the condition $h(\hat{q}) \models q'$

can be checked in NP, as we saw earlier, by guessing a homomorphism. This puts the complexity of the problem into Π_2^p.

The problem UCQ$^{\neq}$-Containment is in fact Π_2^p-complete; for the matching lower bound, see Exercise 11.

25

Aggregates and SQL

When the core of SQL was presented in Chapter 5, a key feature of it was ommited: *aggregation*. It is typically used together with grouping, to apply numerical functions to entire columns of a relation. Recall that one of the relations in the database schema we use in examples is `City [ city_id, name, country ]`. If we want to know how many cities each country has, we can write in SQL

```
SELECT country, COUNT(city) AS city_count
FROM City
GROUP BY country
```

For each value of the `country` attribute, this query groups together all the tuples having this value, and then counts the number of occurrences of `city` in such a group and outputs it as the value of the new attribute `city_count`. Here `COUNT` is an *aggregate function*: it applies to a collection, and produces a single numerical value. The standard aggregates of SQL, in addition to `COUNT`, are `SUM` and `AVG` that compute the sum and the average of a collection of *numbers*, as well as `MIN` and `MAX` that compute minimum and maximum numbers.

Queries with aggregates are extremely common: for example, “find the average grade for each class” or “find the total cost of products sold to each country”. The addition of aggregate functions however takes us out of the realm of first-order logic and relational algebra. To see this, recall the result of the previous section: cardinality comparisons are not expressible in FO. However, they are easily expressible with the help of aggregate functions:

```
SELECT DISTINCT 1 FROM R, S
WHERE (SELECT COUNT(*) FROM R) > (SELECT COUNT(*) FROM S)
```

outputs 1 if $|R| > |S|$ and nothing otherwise.

In this chapter, we look at languages with aggregates: both relational algebra, and logical languages. In the next chapter, we analyze queries that still cannot be expressed with this extremely power addition of aggregates.

Adding aggregates: issues

Previously, we assumed that database entries come from a countably infinite set Const of values. With the addition of aggregates, we can no longer make this assumption, as we need to distinguish attributes that are *numerical.* For example, we can only apply `AVG` over numbers. Thus, in the description of relational schemas, it is no longer sufficient to just state what the arity of each relation is. In addition, we need to provide information about attributes that are numerical. The standard approach to solving this is to consider databases as *two-sorted* structures: there will be columns populated by the usual values from Const, and columns populated by values from a numerical domain Num; it could be natural numbers $\mathbb{N}$, or integers $\mathbb{Z}$, or rationals $\mathbb{Q}$.

The second issue we must address manifests itself when we deal with logical languages for aggregates. Previously, we defined satisfaction of logical formulae only over the the domain $\mathrm{Dom}(D)$ of the database. However, now aggregates can produce new numerical values that did not occur in $\mathrm{Dom}(D)$. Thus, satisfaction of formulae must be defined with respect to the entire infinite numerical domain Num. This creates a possibility that a logical formula is not *safe*, that is, it may be satisfied by an infinite number of assignments of values to free variables. Safety is a known issue in database theory, and it is possible to come up with a logical language with aggregates that does not exhibit this problem. However, we shall take a different approach here. We present the language with aggregates at the level of relational algebra, where safety problems do not arise. We will define a logical language as well, and show how relational algebra translates into it, but the main goal of the logical language is to analyze the power of aggregates for querying relational data.

Two-sorted relational algebra with aggregates

First of all, we need to redefine database schemas to take into account the fact that relations can have ordinary and numerical attributes. Now each relation symbol S in schema $\mathbf{S}$ will come not simply with its arity k, but rather with a word w of length k over the alphabet $\{\mathsf{o}, \mathsf{n}\}$, where o indicates a column of ordinary type, and n indicates a column of numerical type. We shall write $S : w$; for example, $S : \mathsf{onn}$ means that the first attribute of S is of ordinary type taking values in Const, and the second and the third are of numerical type, taking values in Num.

Regarding the numerical domain Num, we assume that it comes with a number of predicates (like $=$ or $<$) and functions (like $+, \cdot$). It also comes with a number of aggregate functions. Formally, an *aggregate function* $\mathcal{F}$ is a function from bags, or multisets, of elements of Num, into Num. In a bag, unlike a set, an element can appear multiple types. For example, $\{\!| 1, 1, 2, 4 |\!\}$ is a bag that have two occurrences of 1, and one occurrence of 2 and 4. We shall use different brackets $\{\!| \; |\!\}$ to distinguish bags from sets. Aggregates must apply to bags rather than sets, since values in databases can repeat.

For example, consider a relation $R = \{(1,1),(2,1),(2,2),(4,4)\}$ whose second column is the above bag. If we ask to compute the average of the second column, applying the average aggregate to $\{\!|1,1,2,4|\!\}$ will correctly produce the value 2. However, if we apply it to the set $\{1,2,4\}$, we get an incorrect value $2.333\cdots$.

Now assume we have a collection Ω that contains predicates, functions, and aggregate functions over Num. Using it we define expressions of *relational algebra with aggregates*, $\mathrm{RA}_{\mathsf{Aggr}}(\Omega)$. Its expressions, unlike expressions of RA, will be *typed*; the type of an expression is again a word over $\{\mathsf{o},\mathsf{n}\}$ indicating which attributes of the output are numerical and which are not. For example, if we write $e : \mathsf{onn}$, it means that expression e returns ternary relations in which the second and the third attributes are numerical. In addition to the usual operations, we add operations of selection based on numerical predicates, applying numerical functions, and applying aggregates.

Schema Relation If $R : \boldsymbol{\tau}$ is in sch, then R is an expression of type $\boldsymbol{\tau}$.

Boolean Operations If e_1, e_2 are expressions of type $\boldsymbol{\tau}$, then so are $e_1 \cup e_2, e_1 \cap e_2, e_1 - e_2$.

Cartesian Product For $e_1 : \boldsymbol{\tau}_1$, $e_2 : \boldsymbol{\tau}_2$, then $e_1 \times e_2$ is an expression of type $\boldsymbol{\tau}_1 \cdot \boldsymbol{\tau}_2$.

Projection If e is of type $\boldsymbol{\tau} = \tau_1 \ldots \tau_k$, and $\alpha = (i_1, \ldots, i_m)$ is a list of numbers from $\{1, \ldots, k\}$, then $\pi_\alpha(e)$ is expression of type $\tau_{i_1} \ldots \tau_{i_m}$.

Selection If e is an expression of type $\boldsymbol{\tau}$, and θ is a condition, then $\sigma_\theta(e)$ is an expression of type $\boldsymbol{\tau}$. Conditions are defined as Boolean combination of statements

- $i \doteq j$ and $i \not\doteq j$ for $i, j \in [1, |\boldsymbol{\tau}|]$;
- $P(i_1, \ldots, i_m)$, where $i_1, \ldots, i_m \in [1, |\boldsymbol{\tau}|]$ and P is an m-ary predicate on Num from Ω.

Function Application If e is an expression of type $\boldsymbol{\tau} = \tau_1 \ldots \tau_k$ and $f : \mathsf{Num}^m \to \mathsf{Num}$ is a function from Ω, then for a sequence $i_1, \ldots, i_m \in [1, k]$ such that $\tau_{i_j} = \mathsf{n}$, for each $j \in [1, m]$, we have that $\mathsf{Apply}[f(i_1, \ldots, i_m)](e)$ is an expression of type $\boldsymbol{\tau} \cdot \mathsf{n}$. Note that m could be 0, in which case the function is just a constant in Num.

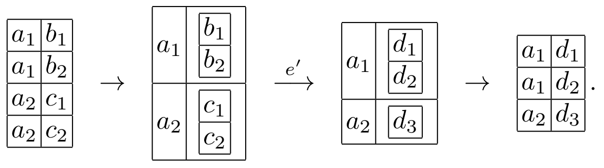
Aggregation If $e : \boldsymbol{\tau}$ is such that the ith position in $\boldsymbol{\tau}$ is n and $\mathcal{F}$ is an aggregate from Ω, then $\mathsf{Aggr}[\mathcal{F}(i)](e)$ is an expression of type $\boldsymbol{\tau} \cdot \mathsf{n}$.

Grouping Assume that e is an expression of type $\boldsymbol{\tau}$ and let $\alpha = (i_1, \ldots, i_m)$ be a list of distinct numbers from $[1, |\boldsymbol{\tau}|]$. By $\boldsymbol{\tau}[\alpha]$ and $\boldsymbol{\tau}[\bar{\alpha}]$ we denote words obtained from $\boldsymbol{\tau}$ by only keeping the positions in α, or by removing the positions in α, respectively. Let $e' : \boldsymbol{\tau}'$ be an expression of arity $|\boldsymbol{\tau}[\bar{\alpha}]|$. Then $\mathsf{Group}_\alpha[e'](e)$ is an expression of type $\boldsymbol{\tau}[\alpha] \cdot \boldsymbol{\tau}'$.

Semantics, examples, complexity

The semantics of the standard relational algebra operations is exactly the same as it was presented before, in Chapter 4. For numerical selection conditions, $P(i_1, \ldots, i_m)$ is true in a tuple $(a_1, \ldots, a_k)$ iff $i_1, \ldots, i_m$ correspond to columns of the numerical type and $P(a_{i_1}, \ldots, a_{i_m})$ is true. For example, $<(1,3)$ is true in a tuple (a_1, a_2, a_3), where the first and the third components are numerical, iff $a_1 < a_3$. Function application simply adds a column with function outputs: $\mathsf{Apply}[f(i_1, \ldots, i_m)]$ transforms each tuple $(a_1, \ldots, a_k)$ into $(a_1, \ldots, a_k, f(a_{i_1}, \ldots, a_{i_m}))$. Aggregation likewise adds the aggregate value as the last column. For example, given relation $R = \{(1,3),(2,3)\}$, the result of $\mathsf{Aggr}[\sum(2)](R)$ is $R = \{(1,3,6),(2,3,6)\}$, where $\sum$ is the sum aggregate. It produces the bag $\{\!|3,3|\!\}$ of second components of tuples, and sums them up. In general, for a relation with tuples $\{\bar{a}_1, \ldots, \bar{a}_n\}$, the result of applying $\mathsf{Aggr}[\mathcal{F}(i)]$ is $\{(\bar{a}_1, v) \ldots, (\bar{a}_n, v)\}$, where $v = \mathcal{F}(\{\!|\bar{a}_1[i], \ldots, \bar{a}_n[i]|\!\})$.

We explain the grouping operation by example first, applying it to a relation of type $\boldsymbol{\tau} = \tau_1\tau_2$, with α containing (1). Then $\boldsymbol{\tau}[\alpha] = (\tau_1)$ and $\boldsymbol{\tau}[\bar{\alpha}] = (\tau_2)$. The operation $\mathsf{Group}_\alpha[e']$, applied to this relation, works as shown in the figure below. First, it groups together tuples with the same value of the first component. Then it applies e' to sets that result from this grouping (in the example, we assume that e' returns $\{d_1, d_2\}$ on input $\{b_1, b_2\}$ and $\{d_3\}$ on input $\{c_1, c_2\}$). Finally, it flattens the result into a usual relation.



Formally, let e evaluate to a relation R. For each tuple $\bar{a} \in R$, define $S_{\bar{a}} = \{\pi_{\bar{\alpha}}(\bar{a}') \mid \bar{a}' \in R \text{ and } \pi_\alpha(\bar{a}) = \pi_\alpha(\bar{a}')\}$, where $\pi_{\bar{\alpha}}$ denotes projection on positions that are not in α. Then $\mathsf{Group}_\alpha[e'](e)$ produces the set $\{(\bar{a}, \bar{b}) \mid \bar{a} \in R \text{ and } \bar{b} \in e'(S_{\bar{a}})\}$.

We now provide a couple of examples of translation of SQL queries with aggregates into the algebra. First look at the query

```
SELECT R.A, SUM(R.B), AVG(R.C)
FROM R
GROUP BY R.A
HAVING SUM(R.B) > AVG(R.C)
```

Under the unnamed perspective, where A, B, C are attributes 1, 2, and 3, this query is expressed as

$$\mathsf{Group}_1\Big[\pi_{3,4}\Big(\sigma_{3>4}\big(\mathsf{Aggr}[\textstyle\sum(1)]\big(\mathsf{Aggr}[\mathsf{Avg}(2)]\big)\big)\Big)\Big](R).$$

The grouping operation does GROUP BY on the first attribute, and then to each group that consists of values of two attributes B and C it applies the function that does the following. First it computes aggregates SUM(R.B) and AVG(R.C) and appends them as values of two additional attributes, the third and the fourth one. The selection condition $\sigma_{3>4}$ then verifies the condition in HAVING, and the projection only keeps the values of aggregates attached to each value of the first attributes. Finally, the grouping operation flattens the relation, resulting in triples R.A, SUM(R.B), AVG(R.C).

As another example, consider a simple counting aggregate over a two-attribute relation T:

```
SELECT T.A, COUNT(T.B)
FROM T
GROUP BY T.A
```

To express this, we need a constant function $\underline{\mathbf{1}}$ that has no inputs and always returns 1. Then the query is expressed as

$$\mathsf{Group}_1\Big(\pi_2\Big(\mathsf{Aggr}[\textstyle\sum(2)]\big(\mathsf{Apply}[\underline{\mathbf{1}}]\big)\Big)\Big)(T)\,.$$

Again, after grouping on the A attribute, we attach a column containing value 1 to all values of B; summing them gives us counts of the numbers of Bs that appear with each A.

Finally, we look at the complexity of $\mathrm{RA}_{\mathsf{Aggr}}(\Omega)$. To check, for a query q defined in this language, whether $\bar{a} \in q(D)$, we actually need to compute $q(D)$. Indeed, the only way to check if a numerical value equals the output of an aggregate function is to compute the entire bag of values to which the aggregate is applied. Still, we can easily prove tractability of the algebra.

Proposition 25.1. *If every function, predicate, and aggregate in Ω is* Ptime*-computable, then data complexity of $RA_{\mathsf{Aggr}}(\Omega)$ is* Ptime.

Proof. This is done by a simple inspection of each operation of the algebra: each one can be computed in polynomial time.

An aggregate logic

We describe a logical language, an extension of FO, that allows numerical operators and aggregate functions. Similarly to relational algebra, the logic must be *two-sorted*, over the ordinary sort whose domain is Const, and the numerical domain Num. Each variable now comes with a sort: variables of the ordinary sort will be denoted by $x, y, z, \ldots$ and will be ranging over Const, and variables of the numerical sort, denoted by $\imath, \jmath, \ldots$ will be ranging over Num. A tuple of variables is thus typed: for example, the type of a tuple $(x, \imath, \jmath, y)$ is onno. The logic $\mathrm{FO}_{\mathsf{Aggr}}(\Omega)$ is defined by mutual induction on terms and formulae.

Terms The only terms of the ordinary sort are variables of ordinary sort $x, y, \ldots$. Terms of the numerical sort are defined as follows:

- every numerical sort variable $\imath$ is a term, whose only free variable is $\imath$;
- if $t_1, \ldots, t_k$ are numerical sort terms and f is a k-ary function in Ω, then $f(t_1, \ldots, t_k)$ is a numerical sort term, whose free variables are the free variables of the t_is;
- if φ is a formula and t is a term whose free variables are among $\bar{x}, \bar{\imath}$, and, in addition, $\bar{y}, \bar{\jmath}$ is a subtuple of $\bar{x}, \bar{\imath}$ and $\mathcal{F}$ is an aggregate from Ω, then $\mathsf{Aggr}_{\mathcal{F}}(\bar{y}, \bar{\jmath})\big(\varphi, t\big)$ is a term whose free variables are those in $\bar{x}, \bar{\imath}$ that do not occur in $\bar{y}, \bar{\jmath}$.

Formulae They are are formed using the standard FO connectives $\wedge, \vee, \neg$ and quantifiers $\exists, \forall$. Atomic formulae are $R(\bar{x}, \bar{\imath})$, where the tuple $(\bar{x}, \bar{\imath})$ has the same type as R, and $P(\imath)$, where P is a numerical predicate from Ω of arity $|\imath|$.

To define the semantics, we need to define satisfaction of formulae $(D, \eta) \models \varphi$, and values of terms $t^{D,\eta} \in \mathsf{Num}$, when η assigns values to free variables of φ and t. All the cases are as in the usual definition of FO; the only novelty here is the aggregate term. Let $t_0(\bar{x}, \bar{\imath}) = \mathsf{Aggr}_{\mathcal{F}}(\bar{y}, \bar{\jmath})\big(\varphi, t\big)$, where free variables of φ and t are among $\bar{x}, \bar{\imath}, \bar{y}, \bar{\jmath}$. Given an assignment η of values to $\bar{x}, \bar{\imath}$, consider all extensions $\eta_1, \ldots, \eta_m$ of η that assign values to $\bar{y}, \bar{\jmath}$ and for which it is the case that $(D, \eta_i) \models \varphi$ for all $i \in [1, m]$. Form the bag $B = \{\!|t^{D,\eta_1}, \ldots, t^{D,\eta_m}|\!\}$. It is a bag since it is possible that $t^{D,\eta_i} = t^{D,\eta_j}$ for different i and j. The value of $t_0^{D,\eta}$ is then $\mathcal{F}(B)$.

Since numerical variables range over the infinite domain Num, it is possible that there are infinitely many assignments η' extending η such that $(D, \eta') \models \varphi$. In this case, by convention, we let $t_0^{D,\eta} = \mathcal{F}(\{\!|\,|\!\})$.

There is a notable difference between this logic and others we have seen so far: formulae of $\mathrm{FO}_{\mathsf{Aggr}}(\Omega)$ can produce infinite outputs. Consider, for instance, $\varphi(x, \imath, \jmath) = R(x) \wedge (\jmath = \imath + 1)$. Then its output consists of all triples (a, i, j) such that $a \in R^D$ and $j = i + 1$, which is infinite whenever Num is infinite. The property of a query language that ensures finiteness of the output is called *safety*. Exercise 16 asks for a construction of a safe aggregate logic that can capture relational algebra with aggregates, but the syntax is much more cumbersome. We use this simple aggregate logic, as it is amenable to a simple analysis that will let us prove results about languages with aggregates in the next chapter.

Theorem 25.2. *Every query of $RA_{\mathsf{Aggr}}(\Omega)$ can be expressed in $FO_{\mathsf{Aggr}}(\Omega)$.*

Proof. The proof goes by induction on expressions of $\mathrm{RA}_{\mathsf{Aggr}}(\Omega)$, and most cases we have already seen in the proof of equivalence of FO and RA. We now concentrate on the three new cases that the aggregate relational algebra has: function application, aggregation, and grouping.

For function application and aggregation, suppose we have an expression e that was translated into a formula $\varphi(z_1,\ldots,z_k)$, where the z_is include variables of both sorts. Then $\mathsf{Apply}[f(i_1,\ldots,i_m)](e)$ is translated into $\psi(\bar z,\imath) = \varphi(\bar z) \wedge (\imath = f(z_{i_1},\ldots,z_{i_m}))$, and $\mathsf{Aggr}[\mathcal{F}(i)](e)$ is translated into $\psi(\bar z,\imath) = \varphi(\bar z) \wedge (\imath = \mathsf{Aggr}_{\mathcal{F}}(\bar z)\big(\varphi(\bar z), z_i\big))$.

For grouping, consider $\mathsf{Group}_\alpha[e'](e)$, and assume e produces a relation of arity k, and for the simplicity of notation, assume that α consists of $(1,\ldots,m)$. If e is translated into $\varphi(z_1,\ldots,z_k)$, and e' is translated into $\varphi'(\bar u)$ over a relation symbol S of arity $k-m$ (as e' takes such a relation as an input in the definition of the grouping operation), then $\mathsf{Group}_\alpha[e'](e)$ is translated into

$$\psi(z_1,\ldots,z_m,\bar u) = \exists \bar w\, \varphi(z_1,\ldots,z_m,\bar w) \ \wedge\ \varphi'(\bar u)\big[\varphi(z_1,\ldots,z_m,\bar v)/S(\bar v)\big]\,.$$

That is, whenever an atom $S(\bar v)$ appears in φ', with $|\bar v| = k-m$. it is replaced by $\varphi(z_1,\ldots,z_m,\bar v)$. To verify that the translation is correct is routine. $\square$

As an example, consider the expression $\mathsf{Group}_1\big(\pi_2\big(\mathsf{Aggr}[\sum(1)]\big)\big)(R)$, corresponding to the SQL query

```
SELECT A, SUM(B) FROM R GROUP BY A
```

for a binary relation R. The inner expression $\mathsf{Aggr}[\sum(1)]$ is over a vocabulary of a unary relation S; it produces a binary relation, by adding the value of the aggregate. This is expressed by a formula

$$S(u) \ \wedge\ \imath = \mathsf{Aggr}_{\sum}(w)\big(S(w), w\big)$$

with free variables u and $\imath$. The expression inside the grouping operator is then given by

$$\varphi'(\imath) = \exists u \Big(S(u) \ \wedge\ \imath = \mathsf{Aggr}_{\sum}(w)\big(S(w), w\big)\Big)\,,$$

and the entire query is obtained by substituting $R(x,w)$ for $S(w)$ everywhere:

$$\psi(x,\imath) \ = \ \big(\exists \jmath\, R(x,\jmath)\big) \ \wedge\ \exists u \Big(R(x,u) \wedge \imath = \mathsf{Aggr}_{\sum}(w)\big(R(x,w), w\big)\Big)\,.$$

26

Inexpressibility of Recursive Queries

Aggregation adds the ability to express powerful counting properties, but there are some common queries nonetheless that it cannot express. Essentially, these are queries requiring recursive computation. A canonical query of this type is reachability in directed graphs. It can be described by the following algorithm: a node y is reachable from a node x if

1. there is an edge from x to y, or
2. there is an edge from x to some node z such that y is reachable from z.

This description is recursive because the second clause defines reachability in terms of itself. Such a description allows us to build arbitrarily long paths. For example, in a graph with nodes $\{0, \ldots, n\}$ and edges $(i, i+1)$ for all $i \in [1, n-1]$, a node j is reachable from i iff $i < j$. Thus, the reachability query can construct paths of any length from 1 to n.

Our main goal is to show that relational algebra with aggregates cannot express this query.

Theorem 26.1. *Given an arbitrary numerical domain* Num*, and an arbitrary collection* Ω *of functions, predicates, and aggregate functions over* Num*, the language* $RA_{\mathsf{Aggr}}(\Omega)$ *cannot express the reachability query.*

The result we prove is in fact more general: we establish a certain locality property, showing that a query in $\mathrm{RA}_{\mathsf{Aggr}}(\Omega)$ can only "see" up to a fixed distance, determined by the query, and thus would not be able to construct paths of arbitrarily many different lengths as the reachability query does.

The proof is split into two parts. First we express $\mathrm{RA}_{\mathsf{Aggr}}(\Omega)$, via its translation into $\mathrm{FO}_{\mathsf{Aggr}}(\Omega)$, in a different logic. While it has features that make it unsuitable as the basis of a query language, it is easier to analyze mathematically. We then prove that the resulting logic has the desired locality properties.

A counting logic

Given a numerical domain, we now introduce a counting logic $\mathbf{L_C}$ and its sublogic $\mathbb{L}_\mathbf{C}$, which we prove to be equivalent to $\mathbf{L_C}$. The logics as before are two-sorted, over the ordinary sort with domain Const, and numerical sort over domain Num (which we assume to contain at least the set $\mathbb{N}$ of natural numbers). The definition is given by induction, and it also defines the *rank* of a formula, denoted by $\text{rank}(\varphi)$.

- Every variable is a term (of a respective sort), and every $c \in \mathsf{Num}$ is a term of the numerical sort; their rank is 0.
- Atomic formulae, whose rank is 0, are $R(\bar{u})$, where $\bar{u}$ is a tuple of terms, or $u_1 = u_2$, where u_1, u_2 are terms.
- If φ is a formula, then so is $\neg\varphi$, and $\text{rank}(\varphi) = \text{rank}(\neg\varphi)$.
- If $\varphi_i,\ i \in I$, is a collection of formulae (finite or infinite), whose free variables are among $\bar{z}$, then $\bigvee_{i\in I} \varphi_i$ and $\bigwedge_{i\in I} \varphi_i$ are formulae whose free variables are $\bar{z}$, and whose rank is $\sup_{i\in I} \text{rank}(\varphi_i)$. The usual conjunction and disjunction are special cases when $I = \{1, 2\}$.
- If φ is a formula with free variables $\bar{z}, \bar{u}$ (of both sorts), and $n \in \mathbb{N}$, then $\exists^{\geq n}\bar{u}\,\varphi$ is a formula whose free variables are $\bar{z}$, and whose rank is $\text{rank}(\varphi) + |\bar{u}|$.

The logic $\mathbf{L_C}$ consists of all the formulae where $\text{rank}(\varphi)$ is finite. Notice that the rank may be infinite because of the infinitary conjunctions and disjunctions $\bigwedge$ and $\bigvee$. For example, if $\text{rank}(\varphi_i) = i$ for all $i \in \mathbb{N}$, then $\text{rank}(\bigvee_{i\in I} \varphi_i)$ is infinite. The logic $\mathbb{L}_\mathbf{C}$ further restricts the definition by only allowing quantification $\exists^{\geq n} u$, where u is a single variable of the ordinary sort.

The semantics of $\psi(\bar{z}) = \exists^{\geq n}\bar{u}\,\varphi$ is that there exist at least n witnesses for $\bar{u}$. That is, $(D, \eta) \models \varphi$ if there are at least n extensions of η to η' defined on variables $\bar{u}$ such that $(D, \eta') \models \varphi$. We can also use the shorthand $\exists^{=n}\bar{u}\,\varphi$ to say that there are exactly n such extensions: this is equivalent to $\exists^{\geq n}\bar{u}\,\varphi \wedge \neg\exists^{\geq n+1}\bar{u}\,\varphi$. Note that this does not change the rank.

Lemma 26.2. *Every formula of $FO_{\mathsf{Aggr}}(\Omega)$ can be expressed in $\mathbf{L_C}$.*

Proof. We translate every $\text{FO}_{\mathsf{Aggr}}(\Omega)$ formula $\varphi(\bar{z})$ into a formula $\varphi^\circ(\bar{z})$ of $\mathbf{L_C}$, and every numerical term $t(\bar{z})$ into a formula $\alpha_t(\bar{z}, \imath)$ of $\mathbf{L_C}$, whose meaning is that $\imath$ is the value of $t(\bar{z})$. That is, $(D, \eta) \models \alpha_t$ iff $t^{D,\eta} = \eta(\imath)$. To ensure that these are formulae of $\mathbf{L_C}$, we must show that they have finite rank. Towards that, we extend the definition of rank to $\text{FO}_{\mathsf{Aggr}}(\Omega)$ formulae and terms. For terms that are variables, their rank is zero. For atomic formulae $R(t_1, \ldots, t_m)$ or $P(t_1, \ldots, t_m)$, and for a term $f(t_1, \ldots, t_m)$, where R is a database relation, P is a predicate, and f is a function from Ω, the rank is $\max_i \text{rank}(t_i)$. The rank of $\varphi \wedge \psi$ or $\varphi \vee \psi$ is $\max\{\text{rank}(\varphi), \text{rank}(\psi)\}$, and negation does not change the rank. Quantification increases the rank by 1, i.e., $\text{rank}(\exists x\,\varphi) = \text{rank}(\varphi) + 1$, and likewise for $\forall$. Finally, for aggregate terms, $\text{rank}(\mathsf{Aggr}_{\mathcal{F}}(\bar{u})\big(\varphi, t)\big) = \max\{\text{rank}(\varphi), \text{rank}(t)\} + |\bar{u}|$. We show, along with producing the translation, that it does not increase the rank of the formula.

A term which is a numerical sort variable $\imath$ is translated by a tautology $\imath = \imath$ (as the formula, that is true, needs a free variable). For a predicate P from Ω, the translation of $P(t_1, \ldots, t_m)$ is $\bigvee \big(\alpha_{t_1}(\bar{z}_1, c_1) \wedge \cdots \wedge \alpha_{t_m}(\bar{z}_m, c_m)\big)$, where each t_i is translated into $\alpha_{t_i}(\bar{z}_i, \jmath)$, and the disjunction is over all (potentially infinitely many) tuples of elements of Num such that $P(c_1, \ldots, c_m)$ holds. The translation of function terms is very similar. Note that the rank of this formula is $\max_i \text{rank}(\alpha_{t_i}) \leq \text{rank}(P(t_1, \ldots, t_m))$.

Translation of relational atoms and FO connectives preserves the structure, i.e., $(R(\bar{z}))^\circ = R(\bar{z})$, while $(\varphi \vee \psi)^\circ = \varphi^\circ \vee \psi^\circ$ and likewise for other Boolean operations, and $(\exists z\,\varphi)^\circ = \exists z\,\varphi^\circ$. These translations do not increase the rank.

It remains to translate aggregate terms $t'(\bar{z}) = \mathsf{Aggr}_{\mathcal{F}}(\bar{u})\big(\varphi(\bar{z}, \bar{u}), t(\bar{z}, \bar{u})\big)$, where $\bar{z}, \bar{u}$ can contain variables of both sorts. As we go over tuples $\bar{u}$ that make $\varphi(\bar{z}, \bar{u})$ true, the term $t(\bar{z}, \bar{u})$ takes numerical values. The idea is to consider all such bags of numerical values. For a bag B over Num, we write $\text{supp}(B)$ for its support, i.e., the set of elements that appear in it, and $\sharp(c, B)$ for the number of occurrences of c in B. Next, define

$$\chi_B(\bar{z}) \;=\; \bigwedge_{c \in \text{supp}(B)} \exists^{=\sharp(c,B)}\bar{u}\,\big(\varphi^\circ(\bar{z}, \bar{u}) \;\wedge\; \alpha_t(\bar{z}, \bar{u}, c)\big)$$

saying that the values of t, as $\bar{u}$ ranges over tuples satisfying φ, have exactly the same multiplicities as in B, and

$$\zeta_B(\bar{z}) \;=\; \forall \bar{u}\,\Big(\varphi^\circ(\bar{u}) \rightarrow \bigvee_{c \in \text{supp}(B)} \alpha_t(\bar{z}, \bar{u}, c)\Big)$$

saying that only elements of B occur as values of term t for tuples for which φ holds. Then the aggregate term is translated as

$$\alpha_{t'}(\bar{z}, \imath) \;=\; \bigvee_{B} \big(\chi_B(\bar{z}) \wedge \zeta_B(\bar{z}) \wedge (\imath = \mathcal{F}(B))\big)\,,$$

which is a disjunction over all bags B, finite and infinite, saying that the values of term t form exactly B, and $\imath$ equals the value of the aggregate $\mathcal{F}$ on B. The rank of this formula is $\max\{\text{rank}(\varphi^\circ), \text{rank}(\alpha_t)\} + |\bar{u}| \leq \max\{\text{rank}(\varphi), \text{rank}(t)\} + |\bar{u}| = \text{rank}(t')$. □

Lemma 26.3. *Every formula of* $\mathbf{L_C}$ *can be expressed in* $\mathbb{L}_{\mathbf{C}}$.

Proof. To prove this, we need to replace quantifiers $\exists^{\geq n}\bar{u}\,\varphi$ with $\exists^{\geq n}u\,\varphi$, without increasing the rank. We show how to do this when $|\bar{u}| = 2$; the general proof is then by induction on $|\bar{u}|$, using the case $|\bar{u}| = 2$ as the induction step. Consider a formula $\exists^{\geq n}(x, y)\,\varphi(x, y, \bar{z})$. The idea of replacing this by simpler quantifiers is to say that there are at least k_1 xs for which there exist exactly ℓ_1 ys satisfying φ, and there are exactly k_2 xs for which there exist exactly ℓ_2 ys satisfying φ, etc, with all the ℓ_is being distinct, so the same pair (x, y) is never counted twice. Formally, a collection of pairs $\{(k_1, \ell_1), \ldots, (k_s, \ell_s)\}$ is an n-witness if $\sum_{i=1}^{s} k_i \cdot \ell_i \geq n$ and all the ℓ_is are distinct. Then $\exists^{\geq n}(x, y)\, \varphi(x, y, \bar{z})$ is equivalent to the infinite disjunction:

$$\bigvee_{n\text{-witness}\{(k_1,\ell_1),\ldots,(k_s,\ell_s)\}} \quad \bigwedge_{i=1}^{s} \exists^{\geq k_i} x\, \exists^{=\ell_i} y\, \varphi(x, y, \bar{z})$$

whose rank equals $\mathrm{rank}(\varphi) + 2$, i.e., the rank of $\exists^{\geq n}(x, y)\, \varphi(x, y, \bar{z})$.

Finally, we get rid of quantifiers over the numerical sort using infinite disjunctions: $\exists^{\geq n}\imath\, \varphi(\bar{z}, \imath)$ is equivalent to $\bigvee_C \bigwedge_{c \in C} \varphi(\bar{z}, c)$, where C ranges over subsets of Num with at least n elements. Note that this actually reduces the rank by 1. $\square$

Combining this with Theorem 25.2, we see that every query in $\mathrm{RA}_{\mathsf{Aggr}}(\Omega)$ can be expressed in $\mathbb{L}_{\mathbf{C}}$.

Locality of queries and counting logic

We define the notion of locality assuming that the database D contains a single binary relation R, viewed as edges of a graph. Of course the definition, and the result, generalize to arbitrary databases; this is the subject of Exercise 19. Given such a database, the *distance* $d(a, b)$ between $a, b \in \mathrm{Dom}(D)$ is the minimal length of a path between a and b, where in the path direction does not matter. For example, if we have $R(a, c)$ and $R(b, c)$ in D, then $d(a, b) = 2$, by first traversing the edge from a to c forward, and then from c to b backwards. For a tuple $\bar{a} = (a_1, \ldots, a_n)$ of elements in $\mathrm{Dom}(D)$, we define $d(\bar{a}, b) = \min\{d(a_i, b) \mid i \in [1, n]\}$ for every $b \in \mathrm{Dom}(D)$. We further define the *r-neighborhood* $N_r^D(\bar{a})$ of $\bar{a}$ in D to consist of all the facts $R(b, c)$ in D such that both $d(\bar{a}, b) \leq r$ and $d(\bar{a}, c) \leq r$. We assume that $\bar{a}$ is a tuple of distinguished elements in $N_r^D(\bar{a})$. Similarly to what we did in Chapter when we characterized $\mathsf{CQ\text{-}Containment}$ via homomorphisms, such distinguished elements are to be preserved by mappings, in this case, isomorphisms. More precisely, we say that two r-neighborhoods $N_r^D(\bar{a})$ and $N_r^D(\bar{b})$ are isomorphic if there is a bijection (a one-to-one onto mapping) $h : \mathrm{Dom}(N_r^D(\bar{a})) \to \mathrm{Dom}(N_r^D(\bar{b}))$ such that $h(\bar{a}) = \bar{b}$ and $R(c, c')$ is a fact in D iff $R(h(c), h(c'))$ is.

A k-ary query q is called *r-local* if for every database D and every two k-tuples $\bar{a}$ and $\bar{b}$ such that $N_r^D(\bar{a})$ and $N_r^D(\bar{b})$ are isomorphic, we have that $\bar{a} \in q(D)$ iff $\bar{b} \in q(D)$. A query is *local* if it is r-local for some $r \in \mathbb{N}$.

An example of a query that is not local is reachability. Indeed, assume it were r-local, for some $r > 0$, as a query with two free variables $q(x, y)$. Consider then the database D with facts $R(i, i + 1)$ for $i \in [0, 5r]$. Then $N_r^D(r, 4r)$ and $N_r^D(4r, r)$ are isomorphic. Indeed, each of these neighborhoods is a disjoint union of two chains of length of $2r$, with distinguished elements in the middle of those chains. Thus, by locality, q could not distinguish $(r, 4r)$ from $(4r, r)$, while the reachability query does: $4r$ is reachable from r, but not conversely.

Lemma 26.4. *Over graphs of ordinary type, i.e., databases with a binary relation R : oo, every query defined by a $\mathbb{L}_{\mathbf{C}}$ formula of rank k with all free variables of non-numerical sort is $(3^k-1)/2$-local.*

This will complete the proof of Theorem 26.1, by showing that every $\mathrm{RA}_{\mathsf{Aggr}}(\Omega)$ query that produces relations in which columns are of ordinary type is r-local for some $r \geq 0$. This number r is $(3^k-1)/2$, where k is the rank of an $\mathbb{L}_{\mathbf{C}}$ into which the $\mathrm{RA}_{\mathsf{Aggr}}(\Omega)$ query is translated. Since reachability is not local, the result follows.

Thus, it remains to prove Lemma 26.4. We do it by induction on the rank of the formula. Note that for the sequence defined by $r_0 = 0$, $r_{i+1} = 3r_i + 1$ we have $r_k = (3^k-1)/2$. Thus, we need to show that, for $\mathbb{L}_{\mathbf{C}}$:

1. atomic formulae are 0-local;
2. if $\varphi_i(\bar{x})$, $i \in I$, are r-local, then $\bigvee_{i\in I} \varphi_i(\bar{x})$ and $\bigwedge_{i\in I} \varphi_i(\bar{x})$ are r-local, and if $\varphi(\bar{x})$ is r-local, then so is $\neg\varphi(\bar{x})$;
3. if $\varphi(\bar{x}, y)$ is r-local, then $\psi(\bar{x}) = \exists^{\geq n} y\, \varphi(\bar{x}, y)$ is $3r+1$-local.

The first two items follow straight from definitions (note that the isomorphism of $N_0^D(\bar{a})$ and $N_0^D(\bar{b})$ means that $\bar{a}$ and $\bar{b}$ witness the same atomic formulae $R(\bar{x})$ or $x = y$). Thus, we concentrate on the quantification case. Assume that h is an isomorphism from $N_{3r+1}^D(\bar{a})$ to $N_{3r+1}^D(\bar{b})$. We show that there is bijection $f : \mathrm{Dom}(D) \to \mathrm{Dom}(D)$ such that $N_r^D(\bar{a}, c)$ is isomorphic to $N_r^D(\bar{b}, f(c))$ for each $c \in \mathrm{Dom}(D)$. This will suffice: if $\psi(\bar{a})$ is true, we can find $c_1, \ldots, c_n \in \mathrm{Dom}(D)$ such that $\varphi(\bar{a}, c_i)$ is true for $i \in [1, n]$. But since $N_r^D(\bar{a}, c_i)$ is isomorphic to $N_r^D(\bar{b}, f(c_i))$ and f is a bijection, the distinct elements $f(c_1), \ldots, f(c_n)$ witness that $\psi(\bar{b})$ is true in D. Likewise, the truth of $\psi(\bar{b})$ implies the truth of $\psi(\bar{a})$, by applying f^{-1} instead of f.

To construct this bijection f, consider a pair (G, v), where G is a binary relation and v is an element such that every element of G is at the distance at most r from v. Consider $c \in N_{2r+1}^D(\bar{a})$ and suppose there is an isomorphism between $N_r^D(c)$ and (G, v) sending c to v. Since h is an isomorphism between the $3r+1$-neighborhoods of $\bar{a}$ and $\bar{b}$, there is also an isomorphism between $N_r^D(h(c))$ and (G, v) sending $h(c)$ to v. Thus, the number of elements c in $N_{2r+1}^D(\bar{a})$ and in $N_{2r+1}^D(\bar{b})$ such that $N_r^D(c)$ is isomorphic to (G, v) is the same. Therefore, the elements c in $\mathrm{Dom}(D) - N_{2r+1}^D(\bar{a})$ and in $\mathrm{Dom}(D) - N_{2r+1}^D(\bar{b})$ such that $N_r^D(c)$ is isomorphic to (G, v) is the same as well: together with the identical number of witnesses in $N_{2r+1}^D(\bar{a})$ and $N_{2r+1}^D(\bar{b})$ they add up to the same number, namely the number of elements of $\mathrm{Dom}(D)$ whose r-neighborhood is isomorphic to (G, v).

Since the graph (G, v) was chosen arbitrarily, this tells us that there is a bijection $f : \mathrm{Dom}(D) - N^D_{2r+1}(\bar{a}) \to \mathrm{Dom}(D) - N^D_{2r+1}(\bar{b})$ such that $N^D_r(c)$ and $N^D_r(f(c))$ are isomorphic. We extend it to all of $\mathrm{Dom}(D)$ by letting $f(c) = h(c)$ if $c \in N^D_{2r+1}(\bar{a})$ (and thus $f(c) \in N^D_{2r+1}(\bar{b})$). To conclude the proof, we note that if $c \in N^D_{2r+1}(\bar{a})$ then all elements of $N^D_r(c)$ are of distance at most $3r+1$ from $\bar{a}$, then $N^D_r(\bar{a}, c)$ is isomorphic to $N^D_r(\bar{b}, f(c))$ since f in this case is the isomorphism h. If on the other hand $c \notin N^D_{2r+1}(\bar{a})$, then $N^D_r(\bar{a}, c)$ is the disjoint union of $N^D_r(\bar{a})$ and $N^D_r(c)$. But then $N^D_r(\bar{a}, f(c))$ is the disjoint union of $N^D_r(\bar{b})$ and $N^D_r(f(c))$. Thus, $N^D_r(\bar{a}, c)$ and $N^D_r(\bar{b}, f(c))$ are isomorphic as disjoint unions of isomorphic neighborhoods. This completes the proof of Lemma 26.4. $\square$

Using Theorem 26.1 it is easy to prove that many other queries are not expressible with aggregation. For example, testing whether the graph is connected cannot be expressed. Indeed, assume it were expressible, and again look at the graph of the successor relation with edges $(k, k+1)$ over the domain $[1, n]$. Given two nodes i and j, we modify the graph by removing edges $(i, i+1)$ and $(j, j+1)$ and instead adding edges $(i, j+1)$ and $(n, i+1)$. It is a simple exercise to express this transformation in FO, and to show that the resulting graph is connected iff $i < j$. Thus, with connectivity test, one could define reachability, contradicting Theorem 26.1.

Note also that Theorem 26.1 only talks about inexpressibility over relations of ordinary type. What if instead relations stored elements of Num? Then there is no known analog of Theorem 26.1: proving such an analog would resolve deep open problems in complexity theory (see Exercise 21 for additional explanations).

27

Adding Recursion: Datalog

As discussed in the previous chapter, a serious limitation of SQL, as well as all the query languages encountered so far, is their inability to express recursive queries such as "compute the transitive closure of a graph". In this chapter, we study a query language that is powerful enough to express such queries. This language is called Datalog, and it can be seen as the extension of UCQs or SPJU (select-project-join-union) queries with the key feature of recursion.

Before delving into the details of Datalog, let us explain how the transitive closure of a graph can be computed via a Datalog query. We first assume that the given graph $G = (V, E)$ is stored in a database as follows:

$$D_G \;=\; \{\mathrm{Edge}(a, b) \mid (a, b) \in E\}.$$

We can now inductively compute, starting from D_G, the transitive closure of G, which will be stored in a binary relation called Answer, via the following set of Datalog rules (also called Datalog program):

$$\begin{aligned}
\mathrm{Reachable}(x, y) &\;\text{:–}\; \mathrm{Edge}(x, y)\\
\mathrm{Reachable}(x, y) &\;\text{:–}\; \mathrm{Reachable}(x, z), \mathrm{Edge}(z, y)\\
\mathrm{Answer}(x, y) &\;\text{:–}\; \mathrm{Reachable}(x, y).
\end{aligned}$$

The first rule, which is essentially the base step of the inductive definition, simply states that if there exists an edge from x to y, then y is reachable from x. The second rule, which corresponds to the inductive step of the definition, states that if z is reachable from x and there is an edge from z to y, then y is reachable from x. Notice that the second rule is recursive in the sense that the definition of Reachable depends on itself. Finally, the last rule computes the relation Answer by simply copying the relation Reachable. Eventually, the Datalog query that computes the transitive closure of a graph is the pair (Π, Answer), where Π is the set of Datalog rules given above. Such a query states the following: execute the Datalog program Π on the input database D, which will basically compute a new database $\Pi(D)$ that contains D, and then return as an answer the set of tuples $\{\bar{a} \mid \mathrm{Answer}(\bar{a}) \in \Pi(D)\}$.

Syntax of Datalog

For defining the syntax of Datalog, we typically use a rule-based syntax similar to that of CQs. A *Datalog rule* ρ is an expression of the form

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

where $n \geq 1$, $R_i(\bar{x}_i)$ is an atomic formula for each $i \in [1, n]$, and $P(\bar{x})$ is an atomic formula that mentions only variables, i.e., $\bar{x}$ consists only of variables. Furthermore, each variable in $\bar{x}$ must appear in at least one of $\bar{u}_1, \ldots, \bar{u}_n$; this is known as the *safety condition.* The atom $P(\bar{x})$ is the *head* of ρ, denoted $\mathsf{head}(\rho)$, while $R_1(\bar{u}_1), \ldots, R_n(\bar{u}_n)$ is the *body* of ρ, denoted $\mathsf{body}(\rho)$.

A *Datalog program* Π is a finite set of Datalog rules. A relation symbol mentioned in Π is *extensional* if it occurs only in the body of the rules of Π, and it is *intensional* if it appears in the head of at least one rule of Π. The *extensional (database) schema* of Π, denoted $\mathsf{edb}(\Pi)$, consists of the extensional relation symbols of Π, while the *intentional schema* of Π, denoted $\mathsf{idb}(\Pi)$, is the set of all intentional relation symbols of Π. Intuitively, the extensional relations are given as an input to Π via an extensional database, and the intentional relations are those computed by Π. The *schema* of Π, denoted $\mathsf{sch}(\Pi)$, is the set of relations $\mathsf{edb}(\Pi) \cup \mathsf{idb}(\Pi)$. Finally, A *Datalog query* is a pair (Π, R), where Π is a Datalog program and $R \in \mathsf{idb}(\Pi)$.

Semantics of Datalog

An interesting property of Datalog is the fact that we can define its semantics either declaratively by adopting a *model-theoretic* approach, or operationally by following a *fixpoint* approach. In the model-theoretic approach, the Datalog rules are considered as logical sentences asserting a property of the desired result, while in the fixpoint approach the semantics is defined as a particular solution of a fixpoint equation. Note that the informal explanation given above, based on the query that computes the transitive closure of a graph, follows the fixpoint approach. In what follows we present both approaches.

Model-Theoretic Semantics

The idea underlying the model-theoretic approach is to consider a Datalog program as a set of first-order sentences that describes the desired outcome of the program. In other words, the desired outcome is a particular database that satisfies the sentences. Such a database is also called a *model* of the sentences; hence the name model-theoretic semantics. However, there might be infinitely many databases that satisfy the sentences, which means that the sentences alone do not uniquely determine the desired outcome of the program. It is therefore crucial to specify which model is the intended outcome of the program. We proceed to formalize the above discussion. In particular, we are going to explain how a Datalog program is converted into a se of logical sentence, and which model of those sentences is the intended one.

A *Datalog rule* ρ of the form $P(\bar{x})$:– $R_1(\bar{u}_1), \ldots, R_n(\bar{u}_n)$ is converted into the first-order sentence φ_ρ defined as

$$\forall x_1 \cdots \forall x_m \, (R_1(\bar{u}_1) \wedge \cdots \wedge R_n(\bar{u}_n) \;\rightarrow\; P(\bar{x})),$$

where $x_1, \ldots, x_m$ are the variables in ρ. For a Datalog program Π, we write φ_Π for the set of sentences $\{\varphi_\rho \mid \rho \in \Pi\}$. Having φ_Π in place, we can talk about the models of Π. A database D over $\mathsf{sch}(\Pi)$ is a *model* of Π if D satisfies each sentence of φ_Π. Observe that, given a Datalog ρ as the one above (with $u_1, \ldots, u_k$ being the variables and constants occurring in ρ), and a database D, the following statements are equivalent:

1. D satisfies the sentence φ_ρ.
2. For every mapping $h : \{u_1, \ldots, u_k\} \rightarrow \mathrm{Dom}(D)$ that is the identity on Const, if $\{R_i(h(\bar{u}_i))\}_{i \in [1,n]} \subseteq D$ then $P(h(\bar{x})) \in D$.

This is a useful characterization of the notion of satisfaction of a sentence φ_ρ by a database D, which will be used in our later proofs.

It is clear that a Datalog program Π admits infinitely many models. We proceed to explain how we choose the intended one. The key idea is that the intended model should not contain more atoms than needed for satisfying the sentences of φ_Π. In other words, from all the models of φ_Π, we choose the $\subseteq$-minimal ones, i.e., those models D such that, for every atom $R(\bar{a}) \in D$, it is the case that $D - \{R(\bar{a})\}$ is not a model of φ_Π. Based on this simple idea, we proceed to define the semantics of a Datalog program on an input database.

Given a Datalog program Π and a database D over $\mathsf{edb}(\Pi)$, we define

$$\mathsf{MM}(\Pi, D) \;=\; \{D' \mid D' \text{ is a } \subseteq\text{-minimal model of } \Pi \text{ and } D \subseteq D'\}.$$

We proceed to show that $\mathsf{MM}(\Pi, D)$ contains *exactly one* database, which will give rise to the semantics of Π on D. But first we need to establish an auxiliary result. Let $\mathsf{B}(\Pi, D)$ be the union of D with the set of all atoms that can be formed using relations from $\mathsf{idb}(\Pi)$ and constants from $\mathrm{Dom}(D)$. Formally,

$$\mathsf{B}(\Pi, D) \;=\; D \cup \left\{ R(\bar{a}) \mid R \in \mathsf{idb}(\Pi) \text{ and } \bar{a} \in \mathrm{Dom}(D)^{\mathrm{ar}(R)} \right\}.$$

We can show the following:

Lemma 27.1. *Consider a Datalog program Π and a database D over $\mathsf{edb}(\Pi)$. It holds that $\mathsf{B}(\Pi, D)$ is a model of Π that contains D.*

Proof. The fact that $\mathsf{B}(\Pi, D)$ contains D follows by definition. It remains to show that $\mathsf{B}(\Pi, D)$ is a model of Π. Consider an arbitrary rule $\rho \in \Pi$ of the form $P(\bar{x})$:– $R_1(\bar{u}_1), \ldots, R_n(\bar{u}_n)$ with variables and constants $u_1, \ldots, u_k$, and assume that there exists $h : \{u_1, \dots, u_k\} \to \mathrm{Dom}(D)$, which is the identity on Const, such that $\{R_i(h(\bar{u}_i))\}_{i \in [1,n]} \subseteq D$. Due to the safety condition, every variable in $\bar{x}$ occurs in at least one of $\bar{u}_1, \dots, \bar{u}_n$. This implies that $h(\bar{x})$ contains only constants of $\mathrm{Dom}(D)$. Since $R \in \mathsf{idb}(\Pi)$ we conclude that $R(h(\bar{x})) \in \mathsf{B}(\Pi, D)$, and thus $\mathsf{B}(\Pi, D)$ satisfies the sentence φ_ρ. Consequently, $\mathsf{B}(\Pi, D)$ satisfies the set of sentences φ_Π, which implies that $\mathsf{B}(\Pi, D)$ is a model of Π, and the claim follows. □

We are now ready to show the claimed statement concerning $\mathsf{MM}(\Pi, D)$:

Lemma 27.2. *Consider a Datalog program Π and a database D over $\mathsf{edb}(\Pi)$. It holds that $\mathsf{MM}(\Pi, D)$ contains exactly one element.*

Proof. By Lemma 27.1, $\mathsf{B}(\Pi, D)$ is a model of Π that contains D. Thus, there is a subset of $\mathsf{B}(\Pi, D)$ that belongs to $\mathsf{MM}(\Pi, D)$, which implies that $\mathsf{MM}(\Pi, D) \neq \emptyset$. Assume now, towards a contradiction, that $\mathsf{MM}(\Pi, D)$ contains more than one element, and let $D_1, \dots, D_\ell$, for $\ell \geq 2$, be those elements. We proceed to show that $\bigcap_{i \in [1,\ell]} D_i$ is a model of Π that contains D, which contradicts the fact that $D_1, \dots, D_\ell$ are $\subseteq$-minimal models of Π that contain D. Clearly, $D \subseteq \bigcap_{i \in [1,\ell]} D_i$ since $D \subseteq D_i$, for each $i \in [1, \ell]$. Consider now an arbitrary rule $\rho \in \Pi$ of the form $P(\bar{x}) \text{ :- } R_1(\bar{u}_1), \dots, R_n(\bar{u}_n)$ with variables and constants $u_1, \dots, u_k$, and assume that there exists $h : \{u_1, \dots, u_k\} \to \mathrm{Dom}(D)$, which is the identity on Const, such that $\{R_i(h(\bar{u}_i))\}_{i \in [1,n]} \subseteq \bigcap_{i \in [1,\ell]} D_i$. For each $i \in [1, \ell]$, D_i is a model of Π, which in turn implies that $P(h(\bar{x})) \in D_i$. Therefore, $P(h(\bar{x})) \in \bigcap_{i \in [1,\ell]} D_i$, and the claim follows. □

Having the above result in place, we are now ready to define the semantics of a Datalog program on an input database, which will then allows us to define the semantics of Datalog queries.

Definition 27.3. *Consider a Datalog program Π and a database D over $\mathsf{edb}(\Pi)$. The output of Π on D, denoted $\Pi(D)$, is the $\subseteq$-minimal model of Π that contains D. The* evaluation *of a Datalog query $q = (\Pi, R)$ over D, denoted $q(D)$, is the set of tuples $\left\{\bar{a} \in Dom(D)^{ar(R)} \mid R(\bar{a}) \in \Pi(D)\right\}$.* □

The crucial question that comes up is whether we can devise an algorithm that computes the semantics of a Datalog program Π on a database D, which in turn provides an algorithm for evaluating Datalog queries. The next result provides such an algorithm. Let $\mathsf{M}(\Pi, D)$ be all the subsets of $\mathsf{B}(\Pi, D)$ that are models of Π and contain D. Formally,

$$\mathsf{M}(\Pi, D) \;=\; \{D' \mid D' \text{ is a model of } \Pi \text{ and } D \subseteq D' \subseteq \mathsf{B}(\Pi, D)\}.$$

Interestingly, the intersection of the databases occurring in $\mathsf{M}(\Pi, D)$ coincides with the $\subseteq$-minimal model of Π that contains D. By giving a proof similar to that of Lemma 27.2, we can show that $\bigcap \mathsf{M}(\Pi, D)$ is a model of Π that contains D, while the fact that is a $\subseteq$-minimal model follows by construction.

Theorem 27.4. *Consider a Datalog program Π and a database D over $\mathsf{edb}(\Pi)$. It holds that $\Pi(D) = \bigcap \mathsf{M}(\Pi, D)$.*

It is clear that the above result suggests the following procedure for computing the semantics of a Datalog program Π on a database D: construct all the possible subsets of $\mathsf{B}(\Pi, D)$ that are models of Π and contain D, and then compute their intersection. However, this is computationally a very expensive procedure. As we shall see in the next section, the fixpoint approach provides a more efficient algorithm for computing the database $\Pi(D)$.

Fixpoint Semantics

We present an alternative way to define the semantics of Datalog that relies on an operator called the *immediate consequence operator*. This operator is applied on a database in order to produce new atoms. The model-theoretic semantics presented above can be defined as the smallest solution of a fixpoint equation that involves the immediate consequence operator.

Consider a Datalog program Π and a database D over $\mathsf{sch}(\Pi)$. An atom $P(\bar{a})$ is an *immediate consequence* for Π and D if one of the following holds:

1. $P(\bar{a}) \in D$ and $P \in \mathsf{edb}(\Pi)$.
2. There is $\rho \in \Pi$ of the form $P(\bar{x}) \coloneq R_1(\bar{u}_1), \ldots, R_n(\bar{u}_n)$ with variables and constants $u_1, \ldots, u_k$, and a mapping $h : \{u_1, \ldots, u_k\} \to \mathrm{Dom}(D)$, which is the identity on Const, such that $\{R_i(h(\bar{x}_i))\}_{i \in [1,n]} \subseteq D$ and $h(\bar{x}) = \bar{a}$.

The *immediate consequence operator* of Π is defined as the mapping T_Π that takes as input a database D over $\mathsf{sch}(\Pi)$ and produces the database:

$$T_\Pi(D) \ = \ \{P(\bar{a}) \mid P(\bar{a}) \text{ is an immediate consequence for } \Pi \text{ and } D\}.$$

A database D is called a *fixpoint* of T_Π if $T_\Pi(D) = D$. The next lemma collects some useful properties of the T_Π operator that we are going to use below.

Lemma 27.5. *Consider a Datalog program Π. The following hold:*

1. *T_Π is monotone, i.e., for every two databases D, D' with $D \subseteq D'$, it holds that $T_\Pi(D) \subseteq T_\Pi(D')$.*
2. *A database D over $\mathsf{sch}(\Pi)$ is a model of Π if and only if $T_\Pi(D) \subseteq D$.*
3. *Every fixpoint of T_Π is a model of Π.*

We are now ready to establish the following crucial result, which states that the model-theoretic semantics of a Datalog program on a database D coincides with the $\subseteq$-minimal fixpoint of T_Π that contains D.

Theorem 27.6. *Consider a Datalog program Π and a database D over $\mathsf{edb}(\Pi)$. Then $\Pi(D)$ is the $\subseteq$-minimal fixpoint of T_Π that contains D.*

Proof. We first show that $\Pi(D)$ is a fixpoint of T_Π, i.e., $T_\Pi(\Pi(D)) = \Pi(D)$. Since $\Pi(D)$ is a model of Π, Lemma 27.5 implies that $T_\Pi(\Pi(D)) \subseteq \Pi(D)$. By Lemma 27.5, T_Π is monotone, and thus $T_\Pi(T_\Pi(\Pi(D))) \subseteq T_\Pi(\Pi(D))$. Therefore, by Lemma 27.5, $T_\Pi(\Pi(D))$ is a model of Π that contains D. But since $\Pi(D)$ is the $\subseteq$-minimal mode of Π that contains D, we immediately get that $\Pi(D) \subseteq T_\Pi(\Pi(D))$. Consequently, $T_\Pi(\Pi(D)) = \Pi(D)$. By Lemma 27.5, each fixpoint of T_Π that contains D is a model of Π that contains D. Therefore, $\Pi(D)$ is the $\subseteq$-minimal fixpoint of T_Π that contains D, as needed. □

It remains to explain how the $\subseteq$-minimal fixpoint of the T_Π operator that contains the database D is constructed. This is essentially done by iteratively applying the T_Π operator starting from D. We define

$$T_\Pi^0(D) \;=\; D \qquad \text{and} \qquad T_\Pi^{i+1}(D) \;=\; T_\Pi(T_\Pi^i(D)), \text{ for } i \geq 0,$$

and we let

$$T_\Pi^\infty(D) \;=\; \bigcup_{i \geq 0} T_\Pi^i(D).$$

At first glance, the construction of $T_\Pi^\infty(D)$ requires infinitely many iterations. However, since $T_\Pi^\infty(D) \subseteq \mathsf{B}(\Pi, D)$, it is the case that $T_\Pi^\infty(D)$ is obtained in at most $|\mathsf{B}(\Pi, D)|$ iterations. It is easy to verify that

$$T_\Pi^\infty(D) \;=\; T_\Pi^{|\mathsf{B}(\Pi,D)|}(D).$$

We now show the following result:

Theorem 27.7. *Consider a Datalog program Π and a database D over* $\mathsf{edb}(\Pi)$. *Then $T_\Pi^\infty(D)$ is the $\subseteq$-minimal fixpoint of T_Π that contains D.*

Proof. Let us first show that $T_\Pi^\infty(D)$ is a fixpoint of T_Π that contains D. Recall that $T_\Pi^\infty(D) = T_\Pi^{|\mathsf{B}(\Pi,D)|}(D)$, while $T_\Pi(T_\Pi^{|\mathsf{B}(\Pi,D)|}(D)) = T_\Pi^{|\mathsf{B}(\Pi,D)|}(D)$. Thus, $T_\Pi^\infty(D)$ is a fixpoint of T_Π that contains D. It remains to show that $T_\Pi^\infty(D)$ is $\subseteq$-minimal, or, equivalently, for every fixpoint D' of T_Π that contains D, $T_\Pi^\infty(D) \subseteq D'$. Fix such a fixpoint D'. We can show via an easy inductive argument that $T_\Pi^i(D) \subseteq D'$ for every $i \geq 0$, which implies that $T_\Pi^\infty(D) \subseteq D'$. In fact, $T_\Pi^0(D) \subseteq D'$ since $T_\Pi^0(D) = D$. Moreover, $T_\Pi^i(D) \subseteq D'$ implies $T_\Pi(T_\Pi^i(D)) = T_\Pi^{i+1}(D) \subseteq T_\Pi(D') = D'$ by monotonicity of T_Π. □

The next result is an immediate corollary of Theorems 27.6 and 27.7:

Corollary 27.8. *Consider a Datalog program Π and a database D over* $\mathsf{edb}(\Pi)$. *It holds that $\Pi(D) = T_\Pi^\infty(D)$.*

Datalog Query Evaluation

We proceed to study the complexity of Datalog-Evaluation: given a Datalog query $q = (\Pi, R)$, a database D over $\mathsf{edb}(\Pi)$, and a tuple $\bar{a}$ over the elements of $\mathrm{Dom}(D)$ of arity $\mathrm{ar}(R)$, is it the case that $\bar{a} \in q(D)$?

Combined Complexity

We first concentrate on the combined complexity of the problem, i.e., when the input consists of q, D, and $\bar{a}$. We proceed to show that:

Theorem 28.1. Datalog-Evaluation *is* EXPTIME-*complete.*

We first concentrate on the upper bound. Consider a Datalog query $q = (\Pi, R)$, a database D over $\mathsf{edb}(\Pi)$, and a tuple $\bar{a} \in \mathrm{Dom}(D)^{\mathrm{ar}(R)}$. Since, by Corollary 27.8, $\Pi(D) = T_{\Pi}^{\infty}(D)$, we get that

$$q(D) \;=\; \{\bar{b} \in \mathrm{Dom}(D)^{\mathrm{ar}(R)} \mid R(\bar{b}) \in T_{\Pi}^{\infty}(D)\}.$$

Therefore, to show that the problem of deciding whether $\bar{a}$ belongs to $q(D)$ is in EXPTIME, it suffices to show that $T_{\Pi}^{\infty}(D)$ can be computed in exponential time. Recall that for computing $T_{\Pi}^{\infty}(D)$ we need to apply the T_{Π} operator at most $|\mathsf{B}(\Pi, D)|$ times. Notice that $|\mathsf{B}(\Pi, D)| \leq |\mathsf{sch}(\Pi)| \cdot |\mathrm{Dom}(D)|^{\mathrm{ar}(\Pi)}$, where $\mathrm{ar}(\Pi)$ is the maximum arity over all relations of $\mathsf{sch}(\Pi)$.

We now analyze the complexity of the i-th application of T_{Π}. Let *maxvar* and *maxbody* be the maximum number of variables and body atoms, respectively, in a rule of Π. The i-th application of T_{Π} takes time

$$O(|\Pi| \cdot |\mathrm{Dom}(D)|^{\mathit{maxvar}} \cdot \mathit{maxbody} \cdot |T_{\Pi}^{i-1}(D)|)$$

since, for each $\rho \in \Pi$, we need to consider all the possible mappings h, which are the identity on Const, from the variables and constants in ρ to $\mathrm{Dom}(D)$, and then check whether h maps each atom of $\mathsf{body}(\rho)$ to $T_{\Pi}^{i-1}(D)$. Recall that $|T_{\Pi}^{i-1}(D)| \leq |\mathsf{B}(\Pi, D)|$. Hence, each application of T_{Π} takes exponential time.

Summing up, we need to apply the T_Π operator exponentially many times, while each application takes exponential time. Consequently, $T_\Pi^\infty(D)$ can be computed in exponential time, as needed.

For the lower bound we show that a language L in ExpTime is polynomial time reducible to Datalog-Evaluation. Let $M = (Q, \Sigma, \delta, s)$ be a (deterministic) 1-tape Turing machine that decides L in exponential time. The goal is, on input w, to construct in polynomial time in $|w|$ a database D, and a Datalog query $q = (\Pi, \text{Yes})$, where Yes is a 0-ary relation, such that M accepts w if and only if $q(D) = \texttt{true}$. Let us first discuss the high level idea of the reduction.

Consider a pair $(p, a) \in (Q - \{\text{"yes"}, \text{"no"}\}) \times \Sigma$. The transition rule $\delta(p, a) = (p', b, \text{dir})$ expresses the following if-then statement:

if at some time instant t of the computation of M on w, we have that M is in state p, the head points to the tape cell c, and c contains the symbol a

then at time instant $t+1$, we have that M is in state p', the cell c contains b, and the head points to the cell c', where c' is the cell right to c (respectively, the cell left to c, c itself) if dir $=\rightarrow$ (respectively, dir $=\leftarrow$, dir $= -$).

It is possible to encode such an if-then statement via Datalog rules since a Datalog rule is essentially an if-then statement. This in turn allows us to describe the complete evolution of M on input w from its start configuration $sc(w)$ to configuration c that can be reached in 2^m steps, where $m = |w|^k$ for some constant k. To achieve this, we need a way to refer to the i-th time instant of the computation of M on w, and the i-th tape cell of M, where $0 \leq i \leq 2^m - 1$. This can be done by representing the time instances and the tape cells from 0 to $2^m - 1$ by tuples of size m over $\{0, 1\}$, on which the functions "next time instant" and "next tape cell" are realized by means of a successor relation Succ^m from a linear order $\preceq^m$ on $\{0, 1\}^m$. We proceed to formalize the above informal description.

The Extensional and Intensional Schema

We begin by describing the extensional and intensional schema of Π. As we shall see, there will be relations Succ^i, First^i and Last^i, for each $i \in [1, m]$, which tell the successor, the first and the last element from a linear order $\preceq^i$ on $\{0, 1\}^i$, respectively, that will be inductively constructed by Π starting from Succ^1, First^1 and Last^1. The extensional schema $\mathsf{edb}(\Pi)$ is

$$\{\text{Succ}^1, \text{First}^1, \text{Last}^1\},$$

where Succ^1 is a binary relation symbol, and First^1, Last^1 are unary relation symbols. The intensional schema $\mathsf{idb}(\Pi)$ is defined as

$$\begin{gathered}\{\text{Symbol}_a \mid a \in \Sigma\} \ \cup\ \{\text{Head}\} \ \cup\ \{\text{State}_p \mid p \in Q\} \ \cup \\ \{\text{Yes}\} \ \cup\ \textstyle\bigcup_{i \in [2,m]} \{\text{Succ}^i, \text{First}^i, \text{Last}^i\} \ \cup\ \{\preceq^m\},\end{gathered}$$

where the arity of the relations symbols Symbol_a, Head, and $\preceq^m$ is $2m$, of State_p is m, of Succ^i is $2i$, of First^i and Last^i is i, and of Yes is 0.

The intuitive meaning of the relations of $\mathsf{idb}(\Pi)$, apart from Succ^i, First^i, Last^i and $\preceq^m$ that have been discussed above, is as follows:

- $\mathrm{Symbol}_a(t, c)$: at time instant t, the tape cell c contains the symbol a.
- $\mathrm{Head}(t, c)$: at time instant t, the head points at cell c.
- $\mathrm{State}_p(t)$: at time instant t, M is in state p.
- Yes: M has reached an accepting configuration.

Having $\mathsf{edb}(\Pi)$ and $\mathsf{idb}(\Pi)$ in place, we can now proceed with the definition of the database D and the Datalog program Π.

The Database D

We only need to store the relations Succ^1, First^1 and Last^1, which form the base case of the inductive definition of Succ^i, First^i and Last^i. In particular,

$$D \;=\; \{\mathrm{Succ}^1(0,1), \mathrm{First}^1(0), \mathrm{Last}^1(1)\}.$$

The Program Π

The program Π, which is responsible for faithfully describing the evolution of M on w starting from $sc(w)$, is the union of the following five programs:

1. $\Pi_{\preceq}$ that inductively constructs Succ^i, $\preceq^i$ and First^i, for $i \in [1, m]$.
2. Π_{start} that constructs the start configuration $sc(w) = (s, \triangleright, w, \sqcup, \ldots, \sqcup)$.
3. Π_{δ} that simulates the transition function of M.
4. Π_{inertia} that ensures that the tape cells that have not been changed at time instant t keep their values at time instant $t + 1$.
5. Π_{accept} that checks whether M has reached an accepting configuration.

The definitions of the above Datalog program follow. For notational convenience, we write $\bar{x}$ for $x_1, \ldots, x_m$, and $\bar{x}_i$ for $x_{i,1}, \ldots, x_{i,m}$.

The Program $\Pi_{\preceq}$. For each $i \in [1, m-1]$, we add the Datalog rules:

$$\begin{array}{rcl}
\mathrm{Succ}^{i+1}(z, \bar{x}, z, \bar{y}) & \text{:-} & \mathrm{Succ}^i(\bar{x}, \bar{y}), \mathrm{First}^1(z) \\
\mathrm{Succ}^{i+1}(z, \bar{x}, z, \bar{y}) & \text{:-} & \mathrm{Succ}^i(\bar{x}, \bar{y}), \mathrm{Last}^1(z) \\
\mathrm{Succ}^{i+1}(z, \bar{x}, v, \bar{y}) & \text{:-} & \mathrm{Succ}^1(z, v), \mathrm{Last}^i(\bar{x}), \mathrm{First}^i(\bar{y}) \\
\mathrm{First}^{i+1}(x, \bar{y}) & \text{:-} & \mathrm{First}^1(x), \mathrm{First}^i(\bar{y}) \\
\mathrm{Last}^{i+1}(x, \bar{y}) & \text{:-} & \mathrm{Last}^1(x), \mathrm{Last}^i(\bar{y}) \\
\preceq^m (\bar{x}, \bar{y}) & \text{:-} & \mathrm{Succ}^m(\bar{x}, \bar{y}) \\
\preceq^m (\bar{x}, \bar{z}) & \text{:-} & \preceq^m (\bar{x}, \bar{y}), \mathrm{Succ}^m(\bar{y}, \bar{z}).
\end{array}$$

The Program $\Pi_{\textbf{start}}$. Assuming that $w = a_0, \ldots, a_{|w|-1}$, we add the rules:

$$\begin{aligned}
\mathrm{State}_s(\bar{x}) &\text{ :- } \mathrm{First}^m(\bar{x}) \\
\mathrm{Symbol}_{a_0}(\bar{x}, \bar{x}) &\text{ :- } \mathrm{First}^m(\bar{x}) \\
\mathrm{Symbol}_{a_1}(\bar{x}_0, \bar{x}_1) &\text{ :- } \mathrm{First}^m(\bar{x}_0), \mathrm{Succ}^m(\bar{x}_0, \bar{x}_1) \\
&\vdots \\
\mathrm{Symbol}_{a_{|w|-1}}(\bar{x}_0, \bar{x}_i) &\text{ :- } \mathrm{First}^m(\bar{x}_0), \mathrm{Succ}^m(\bar{x}_0, \bar{x}_1), \ldots, \mathrm{Succ}^m(\bar{x}_{|w|-2}, \bar{x}_{|w|-1}) \\
\mathrm{Symbol}_{\sqcup}(\bar{x}_0, \bar{y}) &\text{ :- } \mathrm{First}^m(\bar{x}_0), \mathrm{Succ}^m(\bar{x}_0, \bar{x}_1), \ldots, \\
&\qquad\qquad \mathrm{Succ}^m(\bar{x}_{|w|-2}, \bar{x}_{|w|-1}), \preceq^m (\bar{x}_{|w|-1}, \bar{y}) \\
\mathrm{Head}(\bar{x}, \bar{x}) &\text{ :- } \mathrm{First}^m(\bar{x})
\end{aligned}$$

The Program Π_δ. For each pair $(p, a) \in (Q - \{\text{“yes”}, \text{“no”}\}) \times \Sigma$, with $\delta(p, a) = (p', b, \mathrm{dir})$, we add the following Datalog rules. For brevity, let

$$\Phi_{(p,a)}(\bar{x}, \bar{y}, \bar{z}) \;=\; \mathrm{State}_p(\bar{x}), \mathrm{Head}(\bar{x}, \bar{y}), \mathrm{Symbol}_a(\bar{x}, \bar{y}), \mathrm{Succ}^m(\bar{x}, \bar{z}).$$

The following rules change the state from p to p', and the symbol from a to b at the next time instant of the computation:

$$\begin{aligned}
\mathrm{State}_{p'}(\bar{z}) &\text{ :- } \Phi_{(p,a)}(\bar{x}, \bar{y}, \bar{z}) \\
\mathrm{Symbol}_b(\bar{z}, \bar{y}) &\text{ :- } \Phi_{(p,a)}(\bar{x}, \bar{y}, \bar{z}).
\end{aligned}$$

The next rule, which is responsible for moving the head, depends on the direction dir $\in \{\rightarrow, \leftarrow, -\}$. In particular, if dir $=\rightarrow$, then we add the rule

$$\mathrm{Head}(\bar{z}, \bar{v}) \text{ :- } \Phi_{(p,a)}(\bar{x}, \bar{y}, \bar{z}), \mathrm{Succ}^m(\bar{y}, \bar{v}).$$

If dir $=\leftarrow$, then we add the rule

$$\mathrm{Head}(\bar{z}, \bar{v}) \text{ :- } \Phi_{(p,a)}(\bar{x}, \bar{y}, \bar{z}), \mathrm{Succ}^m(\bar{v}, \bar{y}).$$

Finally, if dir $= -$, then we add the rule

$$\mathrm{Head}(\bar{z}, \bar{y}) \text{ :- } \Phi_{(p,a)}(\bar{x}, \bar{y}, \bar{z}).$$

The Program $\Pi_{\textbf{inertia}}$. Recall that this program is responsible for, essentially, copying the content of the tape cells that have not been affected during the transition from time instant t to time instant $t + 1$. The following rule achieves this for the tape cells coming before the current cell

$$\mathrm{Symbol}_a(\bar{v}, \bar{y}) \text{ :- } \mathrm{Symbol}_a(\bar{x}, \bar{y}), \mathrm{Head}(\bar{x}, \bar{z}), \preceq^m (\bar{y}, \bar{z}), \mathrm{Succ}^m(\bar{x}, \bar{v}).$$

The next rule does the same for the tape cells coming after the current cell

$$\mathrm{Symbol}_a(\bar{x}, \bar{y}) \text{ :- } \mathrm{Symbol}_a(\bar{x}, \bar{y}), \mathrm{Head}(\bar{x}, \bar{z}), \preceq^m (\bar{z}, \bar{y}), \mathrm{Succ}^m(\bar{x}, \bar{v}).$$

The Program $\Pi_{\mathbf{accept}}$. Finally, we check whether M has reached an accepting configuration via the Datalog rule

$$\text{Yes} \;\text{:-}\; \text{State}_{\text{“yes”}}(\bar{x}).$$

It is easy to verify that D and Π can be constructed from M and w in polynomial time. It is also not hard to see that Π faithfully describes the computation of M on input w. This means that, with $q = (\Pi, \text{Yes})$, M accepts w if and only if $q(D) = \texttt{true}$ (we leave the proof as an exercise).

Data Complexity

We now concentrate at the data complexity of Datalog-Evaluation, which corresponds to the complexity of the problem when the query q is fixed and the input consists of the database D and the tuple $\bar{a}$ only. As we show below, fixing the query has a huge impact on the complexity of the problem, namely Datalog-Evaluation, from provably intractable, becomes tractable.

Theorem 28.2. Datalog-Evaluation *is* PTIME*-complete in data complexity.*

Proof. The upper bound follows from the analysis performed in the proof of Theorem 28.1. Given a Datalog program Π and a database D over $\mathsf{edb}(\Pi)$, it is easy to see that when q is fixed so are $|\mathsf{sch}(\Pi)|$, $\text{ar}(\Pi)$, *maxvar* and *maxbody*. Thus, for computing $T_\Pi^\infty(D)$ we need to apply the T_Π operator polynomially many times in the size of D, while each application takes polynomial time in the size of D. Therefore, we can compute $T_\Pi^\infty(D)$ in polynomial time in the size of D, and the claimed upper bound follows.

For the lower bound we provide a reduction from a standard PTIME-hard problem, that is, the *monotone circuit value* problem (MCVP). An instance of MCVP is a set of gates $C = \{g_1, \ldots, g_n\}$, where each such a gate is either an $\wedge$-gate of the form $g_i = g_j \wedge g_k$, or an $\vee$-gate of the form $g_i = g_j \vee g_k$, or a constant value, i.e., $g_i = 1$ or $g_i = 0$. The question is whether the output gate, say g_n, evaluates to 1. We are going to encode such an instance C of MCVP in a database D_C, and construct a Datalog query $q = (\Pi, \text{Yes})$, where Yes is a 0-ary relation, that does depend on C, such that g_n evaluates to 1 if and only if $q(D_C) = \texttt{true}$.

The database D_C is defined as follows:

$$\begin{aligned}
&\{\text{True}(g_i) \mid g_i \in C \text{ and } g_i = 1\}\\
\cup\, &\{\text{False}(g_i) \mid g_i \in C \text{ and } g_i = 0\}\\
\cup\, &\{\text{And}(g_i, g_j, g_k) \mid g_i \in C \text{ and } g_i = g_j \wedge g_k\}\\
\cup\, &\{\text{Or}(g_i, g_j, g_k) \mid g_i \in C \text{ and } g_i = g_j \vee g_k\}\\
\cup\, &\{\text{Final}(g_n) \mid g_n \in C \text{ is the final gate}\}.
\end{aligned}$$

The Datalog program Π, which does not depend on C, that is responsible for evaluating the input circuit, is defined as follows:

$$\begin{array}{rcl}
\mathrm{True}(x) & \text{:–} & \mathrm{Or}(x, y, z), \mathrm{True}(y) \\
\mathrm{True}(x) & \text{:–} & \mathrm{Or}(x, y, z), \mathrm{True}(z) \\
\mathrm{True}(x) & \text{:–} & \mathrm{And}(x, y, z), \mathrm{True}(y), \mathrm{True}(z) \\
\mathrm{Yes} & \text{:–} & \mathrm{Final}(x), \mathrm{True}(x).
\end{array}$$

Clearly, g_n evaluates to 1 if and only if $q(D) = \texttt{true}$, and the claim follows. $\square$

29

Static Analysis of Datalog Queries

We proceed to discuss central static analysis tasks for Datalog queries that are important for query optimization purposes. In fact, we consider the three fundamental tasks that we have also studied for first-order and conjunctive queries, namely satisfiability, containment and equivalence. We also discuss a new static analysis task, known as boundedness, that is relevant for recursive query languages such as Datalog. In simple words, a Datalog query is bounded if it can be equivalently rewritten as a Datalog query without recursion.

As we shall see, we can easily check whether a Datalog query is satisfiable. On the other hand, the problem of checking whether a Datalog query is contained into (or is equivalent to) another Datalog query, as well as the problem of checking whether a Datalog query is bounded, are undecidable.

Satisfiability

We start by considering the satisfiability problem: given a Datalog query $q = (\Pi, R)$, is there a database D over $\mathsf{edb}(\Pi)$ such that $q(D) \neq \emptyset$? We can show that this can be checked efficiently:

Theorem 29.1. Datalog-Satisfiability *is in* PTIME.

We proceed to show Theorem 29.1 for constant-free Datalog queries, i.e., Datalog queries $q = (\Pi, R)$ where the rules occurring in Π do not mention constants. For the general case with constants see Exercise 36.

We first characterize the satisfiability of a constant-free Datalog query via a very simple database. Given a Datalog program Π, we define the database

$$D_\Pi \ = \{P(\star, \ldots, \star) \mid P \in \mathsf{edb}(\Pi)\},$$

where $\star$ is a value from Const. We can show the following:

Lemma 29.2. *Consider a constant-free Datalog query $q = (\Pi, R)$. It holds that q is satisfiable if and only if $q(D_\Pi) \neq \emptyset$.*

Proof. It is clear that if $q(D_{\Pi}) \neq \emptyset$, then q is satisfiable witnessed by the simple database D_{Π}. Assume now that q is satisfiable. This implies that there exists a database D over $\mathsf{edb}(\Pi)$ such that $q(D) \neq \emptyset$. Let $h : \mathrm{Dom}(D) \to \{\star\}$ be the mapping that maps each constant occurring in D to $\star$. Clearly, $h(D) \subseteq D_{\Pi}$, which in turn allows us to show that $h(\Pi(D)) \subseteq \Pi(D_{\Pi})$; the latter can be shown via an easy inductive argument. Therefore, $R(\star, \dots, \star) \in \Pi(D_{\Pi})$, which in turn implies that $\{\star\}^{\mathrm{ar}(R)} \in q(D_{\Pi})$, and the claim follows. □

Lemma 29.2 leads to the following simple procedure for checking whether a constant-free Datalog query $q = (\Pi, R)$ is satisfiable:

$$\text{if } (\underbrace{\star, \dots, \star}_{\mathrm{ar}(R)}) \in q(D_{\Pi}), \text{then yes; otherwise, no.}$$

Therefore, by Theorem 28.1, we get that Datalog-Satisfiability is decidable in exponential time. However, this can be reduced to polynomial time since D_{Π} contains only one value. The analysis in the proof of Theorem 28.1, together with the fact that $|\mathrm{Dom}(D_{\Pi})| = 1$, shows that $T_{\Pi}^{\infty}(D_{\Pi})$ can be computed in linear time, and Theorem 29.1 for constant-free Datalog queries follows.

Containment and Equivalence

We now concentrate on the containment problem for Datalog: given two Datalog queries $q_1 = (\Pi_1, R_1)$ and $q_2 = (\Pi_2, R_2)$, with $\mathsf{edb}(\Pi_1) = \mathsf{edb}(\Pi_2)$, is it the case that $q_1 \subseteq q_2$. We can show the following:

Theorem 29.3. Datalog-Containment *is undecidable.*

The above result is shown via a reduction from a known undecidable problem, namely containment for context-free grammars. A context-free grammar is a set of production rules that describe how to produce words over a certain alphabet. Consider, for example, the grammar G consisting of the rules

$$S \to AA \qquad A \to a \qquad A \to b.$$

The first rule states that we can replace S with AA, while the other two rules state that A can be replaced with a or b. Assuming that S is the starting point of the production, while $\{a, b\}$ is the underlying alphabet, the above grammar produces the words aa, ab, ba, bb.

A *context-free grammar* (CFG) is a tuple (N, T, P, S), where

- N is a finite set, the *non-terminal symbols*,
- T is a finite set disjoint from N, the *terminal symbols*,
- P is a finite subset of $N \times (N \cup T)^*$, the *production rules*, and
- $S \in N$, the *start symbol*.

For any two words $v, w \in (N \cup T)^*$, we say that v *directly yields* w, written $v \Rightarrow w$, if there exists $(x, y) \in P$ and $z_1, z_2 \in (N \cup T)^*$ such that $v = z_1 x z_2$ and $w = z_1 y z_2$. Now, for any two words $v, w \in (N \cup T)^*$, we say that v *yields* w, written as $v \Rightarrow^* w$, if there exists $k \geq 1$, and words $z_1, \ldots, z_k \in (N \cup T)^*$ such that $v = z_1 \Rightarrow z_2 \cdots \Rightarrow z_k = w$. Finally, the *language* of G, denoted $L(G)$, is the set of words $\{w \in T^* \mid S \Rightarrow^* w\}$, that is, all the words w over T that can be obtained starting from S and applying production rules of P.

Given two context-free grammars G_1 and G_2, we say that G_1 is *contained* in G_2, denoted $G_1 \subseteq G_2$, if $L(G_1) \subseteq L(G_2)$. The containment problem for context-free grammars, dubbed CFG-Containment, is defined as expected: given two context-free grammars G_1 and G_2, is it the case that $G_1 \subseteq G_2$? We know that this is an undecidable problem:

Theorem 29.4. CFG-Containment *is undecidable.*

Having Theorem 29.4 in place, to show Theorem 29.3 it suffices to reduce CFG-Containment to Datalog-Containment. In other words, given two context-free grammars G_1 and G_2, the goal is to construct two Datalog queries q_1 and q_2 such that $G_1 \subseteq G_2$ if and only if $q_1 \subseteq q_2$. We first explain how to transform a CFG into a Datalog query. Let us clarify that, in what follows, given a CFG $G = (N, T, P, S)$, we assume that P is a finite subset of $N \times ((N - \{S\}) \cup T)^* - \{\epsilon\}$, where ϵ denotes the empty string. In other words, there is no rule in P that produces the empty string, and the start symbol does not occur in the right-hand side of a rule. This will not affect our undecidability proof since Theorem 29.4 holds even with the above assumptions.

We proceed to define the Datalog program Π_G, where $G = (N, T, P, S)$ is CFG. The extensional schema $\mathsf{edb}(\Pi_G)$ and intensional schema $\mathsf{idb}(\Pi_G)$ are

$$\{\text{Symbol}_A \mid A \in T\} \qquad \text{and} \qquad \{\text{Symbol}_A \mid A \in N\},$$

respectively, where all the relations are binary. For each production rule in P

$$(A, A_1 \cdots A_n),$$

for $n \geq 1$, we add to Π_G the Datalog rule

$$\text{Symbol}_A(x_1, x_{n+1}) \ :\!-\ \text{Symbol}_{A_1}(x_1, x_2), \text{Symbol}_{A_2}(x_2, x_3), \ldots, \\ \text{Symbol}_{A_n}(x_n, x_{n+1}).$$

We finally define the Datalog query $q_G = (\Pi_G, \text{Symbol}_S)$.

Example 29.5. Consider the CFG $G = (N, T, P, S)$, where $N = \{A, S\}$, $T = \{a, b\}$, and $P = \{(S, Aa), (A, abA), (A, aa)\}$. The Datalog program Π_G is

$$\begin{aligned}
\text{Symbol}_S(x_1, x_3) \ &:\!-\ \text{Symbol}_A(x_1, x_2), \text{Symbol}_a(x_2, x_3) \\
\text{Symbol}_A(x_1, x_4) \ &:\!-\ \text{Symbol}_a(x_1, x_2), \text{Symbol}_b(x_2, x_3), \text{Symbol}_A(x_3, x_4) \\
\text{Symbol}_A(x_1, x_3) \ &:\!-\ \text{Symbol}_a(x_1, x_2), \text{Symbol}_a(x_2, x_3),
\end{aligned}$$

while the query $q_G = (\Pi_G, \text{Symbol}_S)$. □

To show the correctness of the above construction, we need to introduce the notion of proof tree of an atom from a Datalog program. Roughly speaking, such a proof tree explains how an atom can be derived from a Datalog program, i.e., it provides a proof for that atom. As we shall see below, this notion is closely related to the notion of derivation tree in context-free languages that essentially explains how a word can be derived from a CFG.

Consider an atom $R(\bar{a})$, with $\bar{a} \in \mathsf{Const}^{\mathrm{ar}(R)}$, and a Datalog program Π. A *proof tree of* $R(\bar{a})$ *from* Π is a labeled rooted tree $T = (V, E, \lambda)$, where λ is a function from V to the set of atoms that can be formed using relations from $\mathsf{sch}(\Pi)$ and constants from Const, such that

1. assuming that $v \in V$ is the root node of T, $\lambda(v) = R(\bar{a})$, and
2. for each internal node $v \in V$ with children $u_1, \dots, u_n$ for some $n \geq 1$, there exists a rule $\rho \in \Pi$ of the form $R_0(\bar{x}_0) \coloneq R_1(\bar{x}_1), \dots, R_n(\bar{x}_n)$, and a mapping h from the variables in ρ to Const such that $\lambda(v) = R_0(h(\bar{x}_0))$, and $\lambda(u_i) = R_i(h(\bar{x}_i))$ for each $i \in [1, n]$.

We say that the sequence of atoms $R_1(\bar{a}_1), \dots, R_n(\bar{a}_n)$ is *induced* by the proof tree T if, assuming that the leaf nodes of T are $v_1, \dots, v_n$ (in this order), then $\lambda(v_i) = R_i(\bar{a}_i)$ for each $i \in [1, n]$. Given a database D over $\mathsf{edb}(\Pi)$, *a proof tree of* $R(\bar{a})$ *from* Π *and* D is a proof tree $T = (V, E, \lambda)$ of $R(\bar{a})$ from Π such that, for each $v \in V$, $\lambda(v) \in \mathsf{B}(\Pi, D)$, i.e., $\lambda(v)$ is an atom with a relation from $\mathsf{sch}(\Pi)$ and constants from $\mathrm{Dom}(D)$, and for each leaf node v of T, $\lambda(v) \in D$.

Consider, for example, the Datalog program Π_G obtained from the CFG G as in Example 29.5. A proof tree of the atom $\mathrm{Symbol}_S(c_1, c_6)$ from Π_G and

$$D \supseteq \{\mathrm{Symbol}_a(c_1, c_2), \mathrm{Symbol}_b(c_2, c_3), \mathrm{Symbol}_a(c_3, c_4), \\ \mathrm{Symbol}_a(c_4, c_5), \mathrm{Symbol}_a(c_5, c_6)\}$$

is the following one

$$\begin{tikzcd}[arrows=dash]
 & & & \mathrm{Symbol}_S(c_1, c_6) \arrow[dll] \arrow[dr] & \\
 & \mathrm{Symbol}_A(c_1, c_5) \arrow[dl] \arrow[d] \arrow[drr] & & & \mathrm{Symbol}_a(c_5, c_6) \\
\mathrm{Symbol}_a(c_1, c_2) & \mathrm{Symbol}_b(c_2, c_3) & & \mathrm{Symbol}_A(c_3, c_5) \arrow[dl] \arrow[dr] & \\
 & & \mathrm{Symbol}_a(c_3, c_4) & & \mathrm{Symbol}_a(c_4, c_5)
\end{tikzcd}$$

Clearly, the above proof tree induces the sequence of atoms

$$\mathrm{Symbol}_a(c_1, c_2), \mathrm{Symbol}_b(c_2, c_3), \mathrm{Symbol}_a(c_3, c_4), \\ \mathrm{Symbol}_a(c_4, c_5), \mathrm{Symbol}_a(c_5, c_6).$$

It is an easy exercise to show the following lemma:

Lemma 29.6. *Consider a Datalog query $q = (\Pi, R)$, a database D over $\mathsf{edb}(\Pi)$, and a tuple $\bar{a} \in Dom(D)^{ar(R)}$. The following are equivalent:*

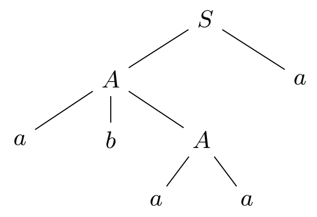
1. $\bar{a} \in q(D)$.
2. *There exists a proof tree of $R(\bar{a})$ from Π and D.*

The next technical result makes apparent the intention underlying the transformation of a CFG G to a Datalog program Π_G.

Lemma 29.7. *Consider a CFG $G = (N, T, P, S)$, and two words $a_1 \cdots a_n \in T^*$ and $c_1 \cdots c_{n+1} \in \mathsf{Const}^*$, for $n \geq 1$. The following are equivalent:*

1. $a_1 \cdots a_n \in L(G)$.
2. *There exists a proof tree of* $\mathrm{Symbol}_S(c_1, c_{n+1})$ *from Π_G that induces the sequence of atoms* $\mathrm{Symbol}_{a_1}(c_1, c_2), \ldots, \mathrm{Symbol}_{a_n}(c_n, c_{n+1})$.

The proof of the above lemma, which is left as an exercise, relies on the correspondence between proof trees and derivation trees in context-free languages. For example, the following tree is a derivation tree of the word *abaaa* from the CFG G given in Example 29.5



The correspondence between the proof tree of the atom $\mathrm{Symbol}_S(c_1, c_6)$ given above, and the derivation tree of the word *abaaa* should be apparent.

By exploiting Lemma 29.7, we can now show the following technical result, which immediately implies Theorem 29.3.

Lemma 29.8. *Consider the CFGs $G_i = (N_i, T_i, P_i, S_i)$ for $i \in \{1, 2\}$. It holds that $G_1 \subseteq G_2$ if and only if $q_{G_1} \subseteq q_{G_2}$.*

Proof. We show the 'if' direction; the 'only if' direction is shown analogously. Consider a database D over $\mathsf{edb}(\Pi_{G_1})$, and assume that $(c, d) \in q_{G_1}(D)$. By

Lemma 29.6, there exists a proof tree of $\text{Symbol}_{S_1}(c,d)$ from Π_{G_1} and D. Assume that this proof tree induces the sequence of atoms

$$s \;=\; \text{Symbol}_{a_1}(c_1,c_2), \text{Symbol}_{a_2}(c_2,c_3), \ldots, \text{Symbol}_{a_n}(c_n,c_{n+1}),$$

where $c = c_1$, $d = c_{n+1}$, and $a_1 \cdots a_n \in T_1^*$. By Lemma 29.7, we get that $a_1 \cdots a_n \in L(G_1)$. Since, by hypothesis, $G_1 \subseteq G_2$, we get that $a_1 \cdots a_n \in L(G_2)$. By Lemma 29.7, there exists a proof tree of $\text{Symbol}_{S_2}(c_1, c_{n+1})$ from Π_{G_2} that induces the sequence of atoms s. Since the atoms in s are atoms of D, Lemma 29.6 implies that $(c,d) \in q_{G_2}(D)$, and the claim follows. □

Let us now turn our attention on the equivalence problem: given two Datalog queries q and q', is it the case that $q \equiv q'$. By exploiting the fact that the containment problem is undecidable, we can easily show the following:

Theorem 29.9. Datalog-Equivalence *is undecidable.*

Proof. It suffices to reduce Datalog-Containment to Datalog-Equivalence. Consider two Datalog queries $q_1 = (\Pi_1, R_1)$ and $q_2 = (\Pi_2, R_2)$, where $\mathsf{edb}(\Pi_1) = \mathsf{edb}(\Pi_2)$. We assume, without loss of generality, that $\mathsf{idb}(\Pi_1) \cap \mathsf{idb}(\Pi_2) = \emptyset$. We define the Datalog query

$$q_{12} \;=\; (\Pi_1 \cup \Pi_2 \cup \{R_{12}(\bar{x}) \coloneq R_1(\bar{x}), R_{12}(\bar{x}) \coloneq R_1(\bar{x})\}, R_{12}),$$

where R_{12} is a new relation not occurring in $\mathsf{sch}(\Pi_1) \cup \mathsf{sch}(\Pi_2)$. It is easy to verify that $q_1 \subseteq q_2$ if and only if $q_{12} \equiv q_2$, and the claim follows. □

Interestingly, there are fragments of Datalog for which containment and equivalence become decidable. Notably, those problems are decidable for:

- *Linear Datalog queries*, where each rule can mention at most on intensional predicate in its body.
- *Guarded Datalog queries*, where each rule should have an atom in its body with an extensional predicate that mentions all the variables in the rule.

Boundedness

As discussed in Chapter 27, given a Datalog program Π and a database D over $\mathsf{edb}(\Pi)$, the semantics of Π on D, i.e., the database $\Pi(D)$, can be computed by repeatedly applying the immediate consequence operator T_Π of Π starting from D until a fixpoint is reached; in fact, by Corollary 27.8, $\Pi(D) = T_\Pi^\infty(D)$. We have also seen that the construction of $T_\Pi^\infty(D)$ does not require infinitely many iterations. Actually, there exists an integer $k \leq |\mathsf{B}(\Pi, D)|$ such that $T_\Pi^\infty(D) = T_\Pi^k(D)$. The smallest integer k such that $T_\Pi^\infty(D) = T_\Pi^k(D)$ is called the *stage* of Π and D, denoted $\mathsf{stage}(\Pi, D)$.

Given a Datalog program Π, it is generally the case that, for some arbitrary database D over $\mathsf{edb}(\Pi)$, the integer $\mathsf{stage}(\Pi, D)$ depends on both Π and D.

This essentially means that the program Π is inherently recursive, or, in other words, the depth of recursion of Π is unbounded. On the other hand, if there is a uniform upper bound (i.e., a bound that depends only on Π) for $\mathsf{stage}(\Pi, D)$, then the recursion of Π is bounded, which essentially means that Π is non-recursive despite the fact that syntactically may look recursive. The following example illustrates that a bounded (seemingly recursive) Datalog program can be replaced by an equivalent non-recursive Datalog program.

Example 29.10. Consider the Datalog program Π consisting of the rules

$$P(x,y) \ {:}\!-\ R(x), P(z,y) \qquad P(x,y) \ {:}\!-\ S(x,y).$$

It is clear that Π is syntactically recursive due to the first rule (P depends on itself). However, Π is bounded, and equivalent to the program consisting of

$$P(x,y) \ {:}\!-\ R(x), S(z,y) \qquad P(x,y) \ {:}\!-\ S(x,y),$$

that is non-recursive. The key reason why we can safely replace the atom $P(z,y)$ in the first rule with the atom $S(z,y)$, which leads to a non-recursive program, is because it does not share any variable with the atom $R(x)$. □

Boundedness for Datalog programs and queries is defined as expected:

Definition 29.11. *A Datalog program Π is* bounded *if there exists a constant $k \geq 0$ such that, for every database D over* $\mathsf{edb}(\Pi)$, $\mathsf{stage}(\Pi, D) \leq k$. *Furthermore, a Datalog query $q = (\Pi, R)$ is bounded if Π is bounded.* □

It should be clear that checking whether a Datalog query is bounded is an important static analysis task that is relevant for optimization purposes. The formal definition of this new static analysis task follows:

PROBLEM:	Datalog-Boundedness
INPUT:	Datalog query q
OUTPUT:	yes if q is bounded and no otherwise

It turned out that the above problem is undecidable.

Theorem 29.12. Datalog-Boundedness *is undecidable.*

This is shown by reducing an undecidable problem called Post Correspondence Problem to Datalog-Boundedness. Interestingly, Datalog-Boundedness is decidable for Linear and Guarded Datalog queries.

Comments and Exercises for Part IV

Comments

Results in Section 23 on unions of conjunctive queries mainly come from [29]. The homomorphism preservation theorem is from [28].

In Section 24 we show inexpressibility of counting properties using the 0-1 law for FO sentences. Our proof follows the proof of [8], which applies to arbitrary schemas. Here we chose to restrict to schemas with unary relations to simplify the presentation and still show the results we need. For more on 0-1 laws, see book treatments of the subject [24, 32]. Results on inequalities (Theorem 24.6 and Exercise 11) are from [33, 20]. For model theory background used in the proof of Theorem 24.1, see any standard logic text, e.g., [6].

Aggregate algebras presented in Section 25 have a long history; see, e.g., [19, 25]. In particular, the algebra in Exercise 15 is from [19]. We follow the presentation of [23] and use the aggregate logic from [16].

The notion of locality used in Section 26 originates in [10] (a different notion from Exercise 20, is from [15]). For textbook treatment of locality, see [24]. The counting logics into which aggregate logic is translated are from [22], which proved their locality. The equivalence of the complexity class TC^0 used in Exercise 21 and the simple counting logic is from [3]; for more on circuit complexity classes and their separation see [34].

Andreas writes about Section 27.

Andreas writes about Section 28.

Exercises

1. Prove Proposition 23.5.
2. Prove that containment of RA^+ queries is Π_2^p-hard. For this, provide a reduction from problem ∀∃3SAT, which is known to be Π_2^p-complete. The input to this problem is a Boolean formula $\alpha(\bar{x}, \bar{y})$ in 3CNF, i.e., in CNF where each clause has three literals. The question is whether the formula $\forall \bar{x} \exists \bar{y}\ \alpha(\bar{x}, \bar{y})$ is true.

3. Prove that containment of $\exists\text{FO}^+$ queries is Π_2^p-complete, following the same ideas as for RA^+ queries.
4. Recall the usual mathematical definition of cartesian product of two relational structures D and D': its domain contains pairs (a, a'), where $a \in \text{Dom}(D)$ and $a' \in \text{Dom}(D')$, and a k-ary relation R in it is interpreted as a set of k-tuples $\big((a_1, a_1'), \ldots, (a_k, a_k')\big)$ such that $(a_1, \ldots, a_k) \in R^D$ and $(a_1', \ldots, a_k') \in R^{D'}$. Prove the following characterization of CQ s: an FO sentence is equivalent to a CQ iff it is preserved under homomorphisms and cartesian product (the latter of course means that if D and D' satisfy φ, then their cartesian product satisfies φ).
5. Give a detailed proof of the fact that if φ and ψ are true in almost all databases, then so is $\varphi \wedge \psi$ (which we used in the proof of Theorem 24.4).
6. Consider databases D over the schema with two unary relations A and B satisfying $B^D \subseteq A^D$. Adapt the proof of Theorem 24.4 given in Chapter 24 to show that relative to such databases, the 0–1 law holds. Use it to show that the property PARITY testing if $|B|$ is even is not expressible in FO over such databases.
7. Use exercise 6 (not just the result but the proof you produced) to infer the following: for every FO sentence φ over the schema with two unary relations A and B, there exists numbers k and m such that knowing $|B^D| \geq k$ and $|A^D - B^D| \geq m$ tells us whether D satisfies φ (that is, any two databases satisfying these conditions will agree on φ).
   Use this to derive that in general checking whether two sets have the same cardinality cannot be done in FO.
8. Without any restrictions on A and B (like in the previous two exercises), give a formal proof that $\lim_{n\to\infty} \mu_n(|A| \geq |B|) = 1/2$, and likewise for $|A| > |B|$. Also show that $\lim_{n\to\infty} \mu_n(|A| = |B|) = 0$.
9. Extend the proof of the 0–1 law to unordered graphs. Recall that you need to construct a theory T which has a unique, up to isomorphism, countable model, and whose sentences are true in almost all graphs. Such a theory T has sentences $\alpha_{k,m}$ that say the following: for every two disjoint sets X and Y of nodes of cardinalities k and m respectively, there exists a node z such that there are edges (z, x) for each $x \in X$ and there is no edge (z, y) for $y \in Y$. While the proof of condition 2 of Theorem 24.4 follows the same ideas as those we saw, the proof of condition 1 is more elaborate and requires ideas not seen in this book; the interested reader is advised to consult [24].
10. Complete the proof of Theorem 24.6, by showing how to handle a) free variables and b) unions.
11. Prove a matching lower bound for Theorem 24.6, by showing that the containment problem is Π_2^p-complete.

12. Instead of disequalities $x \neq y$, now consider CQ s and UCQs with inequalities $x < y$, assuming that there is an order on the domain from which database entries are drawn. Prove that containment remains decidable for such queries.
13. Consider the class of *positive* FO queries: these come from the fragment of FO that is built up from relational atoms $R(\bar{x})$, equality atoms $x = y$, their negations $x \neq y$, by using $\wedge, \vee, \exists, \forall$, but not using negation. Prove that containment of a UCQ$^{\neq}$ query in a positive query is decidable.
14. The class BCCQ consists of Boolean combinations of CQ s, i.e., queries obtained by repeatedly applying operations $Q_1 \cup Q_2$, $Q_1 \cap Q_2$, and $Q_1 - Q_2$, to CQ s. Of course CQ s have to be of the same arity for these operations to apply. Prove that containment of BCCQs is decidable.
15. Consider a different algebra with aggregation where operations of aggregation and grouping are combined into one as follows. Suppose e is of type $\boldsymbol{\tau} = \tau_1 \ldots \tau_k$, and $\alpha = (i_1, \ldots, i_m)$ and $\beta = (j_1, \ldots, j_\ell)$ be two disjoint lists of numbers from $[1, k]$ so that $t_{i_s} = \mathsf{n}$ for $s \leq m$. Let $\mathcal{F}_1, \ldots, \mathcal{F}_m$ be aggregates from Ω. Then $\mathsf{Aggr}_\beta[\mathcal{F}_1(i_1), \ldots, \mathcal{F}_m(i_m)]$ is an expression of type $\boldsymbol{\tau} \cdot \mathsf{n} \ldots \mathsf{n}$, with m of ns added at the end. The semantics is explained by SQL:

    SELECT $\#1, \ldots, \#k, \mathcal{F}_1(\#i_1), \ldots, \mathcal{F}_k(\#i_k)$
    FROM E
    GROUPBY $\#j_1, \ldots, \#j_\ell$

    where E is the result of the expression e. Prove that this construct can be expressed in $\mathrm{RA}_{\mathsf{Aggr}}(\Omega)$.
16. Use the translation of Theorem 25.2 to find syntactic restrictions on $\mathrm{FO}_{\mathsf{Aggr}}(\Omega)$ to make the logic *safe*, i.e., to ensure that no formula is satisfied by the infinite number of tuples.
17. In the translation of the aggregate relational algebra into the aggregate logic (see Theorem 25.2), aggregate terms were of the simple form $\mathsf{Aggr}_{\mathcal{F}}(\bar{z})\big(\varphi, \imath\big)$, where $\imath$ is a variable, i.e., the simplest possible term of the numerical sort. Prove that restricting the definition of $\mathrm{FO}_{\mathsf{Aggr}}(\Omega)$ in such a way does not change its expressive power.
18. In the definition of logic $\mathbf{L_C}$ we assumed, for simplicity, that $\mathbb{N} \subseteq \mathsf{Num}$. Show that this assumption, although applicable to any numerical type that might occur in practice, is not technically necessary.
19. The notion of locality can be extended to databases over arbitrary schemas. Given a database D, its *Gaifman graph* is an undirected graph whose nodes are elements of $\mathrm{Dom}(D)$, and whose undirected edges are pairs $\{a, b\}$ such that a and b appear together in some fact of D, i.e., there is a fact of the form $R(\ldots, a, \ldots, b, \ldots)$ in D. Given a tuple $\bar{a}$, its radius-r ball $B_r^D(\bar{a})$ consists of all $b \in \mathrm{Dom}(D)$ such that there is a path of length at most r from b to an element of $\bar{a}$ in the Gaifman graph. The

radius-r neighborhood of $\bar{a}$ is then the set of all facts of D restricted to elements of $B_r^D(\bar{a})$, with $\bar{a}$ as the tuple of distinguished elements. Show that under this definition, every $\mathbf{L_C}$ query with all free variables of the ordinary sort is local.

20. There is a different way of defining locality, known in the literature as *Hanf-locality* (as opposed to *Gaifman-locality* that we were using in Chapter 26). Given two databases D and D', and tuples $\bar{a}$ and $\bar{a}'$ of the same length over $\mathrm{Dom}(D)$ and $\mathrm{Dom}(D')$, we write $(D, \bar{a}) \sim_r (D', \bar{a}')$ if there is a bijection $f : \mathrm{Dom}(D) \to \mathrm{Dom}(D')$ such that $N_r^D(\bar{a}, a)$ is isomorphic to $N_r^{D'}(\bar{a}', f(a))$. Note that in particular $(D, \bar{a}) \sim_r (D', \bar{a}')$ means that $|\mathrm{Dom}(D)| = |\mathrm{Dom}(D')|$. A query $q(\bar{x})$ is Hanf-local if there exists $r \geq 0$ such that $(D, \bar{a}) \sim_r (D', \bar{a}')$ implies that $\bar{a} \in q(D)$ iff $\bar{a}' \in q(D')$.
    Prove that over databases whose domains contain only elements of the ordinary type, every $\mathrm{RA}_{\mathsf{Aggr}}(\Omega)$ query is Hanf-local. To do so, prove that every query expressible by an $\mathbb{L}_{\mathbf{C}}$ formula without free numerical variables is Hanf-local, by extending the argument in the proof of Lemma 26.4.
21. Consider a restriction of $\mathbf{L_C}$ where infinitary connectives are not allowed, numerical variables can only range over $\{0, \ldots, n-1\}$, for $n = |\mathrm{Dom}(D)|$, the only numerical predicates are $<$, $+$, and $\cdot$ (for addition, we have access to the predicate interpreted as $\{(i, j, k) \mid i, j, k < n \text{ and } i + j = k\}$ and likewise for multiplication), we have quantification over both sorts, and counting terms $\sharp x\, \varphi(x, \bar{y})$ counting the number of elements of $\mathrm{Dom}(D)$ satisfying φ.
    Prove that all sentences of this logic are expressible in $\mathrm{RA}_{\mathsf{Aggr}}(\Omega)$, where Ω has $<, +, \cdot$ and the summation aggregate $\sum$, in two different scenarios. In the first, we have a database where all elements are of the numerical sort. In the second, we have a database where all elements are of the ordinary sort and we have access to an order relation $<$ on the elements of the ordinary sort.
    The importance of these results stems from the fact the restricted logic we defined captures a uniform version of a complexity class called TC^0 (this stands for threshold circuits of constant depth). TC^0 has not been separated from others above it such as PTIME or NLOGSPACE. In particular, this means that bounds on the expressivity of $\mathrm{RA}_{\mathsf{Aggr}}(\Omega)$ cannot be proved either over ordered non-numerical domains, or numerical domains, without resolving deep problems in complexity theory.
22. A *path system* is a tuple $P = (V, R, S, T)$, where V is a finite set of nodes, $R \subseteq V \times V \times V$, $S \subseteq V$ and $T \subseteq V$. A node $v \in V$ is *admissible* if $v \in T$, or there are admissible nodes $u, w \in V$ such that $(v, u, w) \in R$.
    Show that the set of admissible nodes for P can be computed via a Datalog query. In other words, there exists a Datalog query $q = (\Pi, A)$, where A is unary relation, such that, for every path system P, $q(D_P)$ is the set of admissible nodes for P, where D_P stores P in the obvious way, i.e., V, S and T via unary relations, and R via a ternary relation.

23. An undirected graph $G = (V, E)$ is *2-colorable* if there exists a function $f : V \to \{0, 1\}$ such that $(v, u) \in E$ implies $f(v) \neq f(u)$.
    Show that *non-2-colorability* is expressible via a Datalog query, i.e., there exists a Datalog query $q = (\Pi, \text{Yes})$, where Yes is a 0-ary relation, such that, for every undirected graph G, $q(D_G) = \texttt{true}$ if and only if G is *not* 2-colorable, where D_G represents G via the binary relation $\text{Edge}(\cdot, \cdot)$.
24. Prove Theorem 27.4.
25. Prove Lemma 27.5.
26. Prove that every Datalog query $q = (\Pi, R)$ is monotone, i.e., for every two databases D, D' over $\mathsf{edb}(\Pi)$, $D \subseteq D'$ implies $q(D) \subseteq q(D')$.
    Use this to show that the query that asks whether an undirected graph is 2-colorable is not expressible via a Datalog query.
27. Prove that there exists a monotone query that is not expressible via a Datalog query. The proof should not rely on any complexity-theoretic assumption. (It is very easy to show that this holds under the assumption that PTIME $\neq$ NP.)
28. A Datalog program Π is called *linear* if, for every rule $\rho \in \Pi$, $\mathsf{body}(\rho)$ mentions at most one relation from $\mathsf{idb}(\Pi)$. A Datalog query (Π, R) is linear if Π is linear.
    Show that there is no linear Datalog query that computes the set of admissible nodes for a path system P. The proof should not rely on any complexity-theoretic assumption. (It is easier to show this statement if we assume that NLOGSPACE $\neq$ PTIME.)
29. The *predicate graph* of a program Π is the directed graph $G_\Pi = (V, E)$, where V consists of the relations of $\mathsf{sch}(\Pi)$, and $(P, R) \in E$ if and only if there exists a rule $\rho \in \Pi$ of the form
    $$R(\bar{x}) :- \ \ldots, P(\bar{y}), \ldots$$
    We call Π *non-recursive* if G_Π is acyclic. A Datalog query (Π, R) is non-recursive if Π is non-recursive.
    Show that, for every non-recursive Datalog query $q = (\Pi, R)$, there exists a finite UCQ q' such that, for every D over $\mathsf{edb}(\Pi)$, $q(D) = q'(D)$.
30. Let D and q be the database and the Datalog query, respectively, constructed from the Turing machine M and the input word w in the proof of Theorem 28.1. Show that M accepts w if and only if $q(D) = \texttt{true}$.
31. A Datalog program Π is called *guarded* if, for every rule $\rho \in \Pi$, $\mathsf{body}(\rho)$ has an atom that contains (or "guards") all the variables occurring in ρ. A Datalog query (Π, R) is guarded if Π is guarded.
    Prove that the EXPTIME-hardness shown in Theorem 28.1 holds even if we focus on guarded Datalog queries.
32. Consider a Datalog query $q = (\Pi, R)$, where the arity of the relations of $\mathsf{sch}(\Pi)$ is bounded by some integer constant, a database D over $\mathsf{edb}(\Pi)$,

and a tuple $\bar{a}$ over the elements of $\mathrm{Dom}(D)$ of arity $\mathrm{ar}(R)$. Show that the problem of deciding whether $\bar{a} \in q(D)$ is NP-complete.

33. Consider a Datalog query $q = (\Pi, R)$, where $\mathsf{sch}(\Pi)$ consists only of 0-ary relations, and a database D over $\mathsf{edb}(\Pi)$. Show that the problem of deciding whether $q(D) = \mathtt{true}$ is PTIME-complete.
34. Show that evaluation of linear Datalog queries is PSPACE-complete in combined complexity, and NLOGSPACE-complete in data complexity.
35. Show that evaluation of non-recursive Datalog queries is PSPACE-complete in combined complexity, and in DLOGSPACE in data complexity.
36. Show that satisfiability of Datalog queries (with constants) is in PTIME.

Part V

Uncertainty

30

Incomplete Information and Certain Answers

So far we assumed all information in databases was known: we have a set Const from which all entries in databases come, and for every fact $R(\bar{a})$ in a database, every element of the tuple $\bar{a}$ comes from Const.

Plan: model, semantics, complexity of membership

Then query answering as GLB. Example where it's bad, example where it's good. Cert answers with nulls.

INITIAL DUMP

Incomplete data and nulls

A formal model of nulls

We consider incomplete databases with nulls interpreted as missing information. Below we recall definitions that are standard in the literature [**?**, **?**, **?**]. Databases are populated by two types of elements: *constants* and *nulls*, coming from countably infinite sets denoted by Const and Null, respectively. Nulls are denoted by $\perp$, sometimes with sub- or superscripts. If nulls can repeat in a database, they are referred to as *marked*, or labeled, nulls; otherwise one speaks of *Codd nulls*, which are the usual way of modeling SQL's nulls. Marked nulls are standard in applications such as data integration and exchange and OBDA [**?**, **?**, **?**], and are more general than Codd nulls; hence we use them here.

A relational schema is a set of relation names with associated arities. With each k-ary relation symbol R from the vocabulary, an incomplete relational instance D associates a k-ary relation over $\mathsf{Const} \cup \mathsf{Null}$, that is, a finite subset of $(\mathsf{Const} \cup \mathsf{Null})^k$. Slightly abusing notation (as it will never lead to confusion here) we will call it R as well.

The sets of constants and nulls that occur in a database D are denoted by $\mathsf{Const}(D)$ and $\mathsf{Null}(D)$, respectively. The *active domain* of D is $\mathrm{Dom}(D) = \mathsf{Const}(D) \cup \mathsf{Null}(D)$. If D has no nulls, we say that it is *complete*.

A *valuation* of nulls on an incomplete database D is a map $v : \mathsf{Null}(D) \to \mathsf{Const}$ assigning a constant value to each null. It naturally extends to databases, so we can write $v(D)$ as well. The standard semantics of incompleteness in relational databases are defined in terms of valuations, see [**?**, **?**]. These are the *closed world assumption*, or CWA semantics:

$$[\![D]\!]_{\text{CWA}} \;=\; \{v(D) \mid v \text{ is a valuation}\},$$

and the *open-world assumption*, or OWA semantics:

$$[\![D]\!]_{\text{OWA}} \;=\; \left\{ D' \;\middle|\; \begin{array}{l} D' \text{ is complete and} \\ v(D) \subseteq D' \text{ for some valuation } v \end{array} \right\}.$$

We shall also consider the *weak* CWA, or WCWA semantics, inspired by [**?**], given by

$$[\![D]\!]_{\text{WCWA}} \;=\; \left\{ v(D) \cup D' \;\middle|\; \begin{array}{l} v \text{ is a valuation and} \\ \mathrm{Dom}(D') \subseteq \mathrm{Dom}(v(D)) \end{array} \right\}.$$

That is, under CWA, we simply instantiate nulls by constants; under OWA, we can also add arbitrary tuples, and under WCWA, we can only add tuples formed by elements already present. For instance, if $D_0 = \{(\bot, \bot')\}$, then $[\![D]\!]_{\text{CWA}}$ only has instances $\{(c, c')\}$ for $c, c' \in \mathsf{Const}$, while $[\![D]\!]_{\text{WCWA}}$ can have in addition instances that add to (c, c') tuples $(c, c), (c', c')$, and (c', c), and $[\![D]\!]_{\text{OWA}}$ has all instances containing at least one tuple.

The membership question

Certain answers

Complexity of certain answers

31
Tractable Query Answering in Incomplete Databases

INITIAL DUMP

How to overcome coNP

Naive evaluation

Approximating certain answers

The translation is $\varphi(\bar{x}) \mapsto (\varphi^{\mathbf{t}}(\bar{x}), \varphi^{\mathbf{f}}(\bar{x}))$. The conditions it ensures are:

$$\varphi^{\mathbf{t}}(D) \subseteq \mathsf{cert}(\varphi, D) \qquad \varphi^{\mathbf{f}}(D) \subseteq \mathsf{cert}(\neg\varphi, D)$$

where cert is certain answers with nulls.

We write $\bar{x} \Uparrow \bar{y}$ for unification as before.

Translation $\varphi^{\mathbf{t}}$

- $(R(\bar{x}))^{\mathbf{t}} = R(\bar{x})$
- $(x = y)^{\mathbf{t}} = (x = y)$
- $(\varphi \wedge \psi)^{\mathbf{t}} = \varphi^{\mathbf{t}} \wedge \psi^{\mathbf{t}}$
- $(\neg\varphi)^{\mathbf{t}} = \varphi^{\mathbf{f}}$
- $(\exists x\, \varphi)^{\mathbf{t}} = \exists x\, \varphi^{\mathbf{t}}$

Translation $\varphi^{\mathbf{f}}$

- $(R(\bar{x}))^{\mathbf{f}} = \neg\exists\bar{y}\,\big(R(\bar{y}) \wedge \bar{x} \Uparrow \bar{y}\big)$
- $(x = y)^{\mathbf{f}} = \neg(x = y) \wedge \neg\mathsf{null}(x) \wedge \neg\mathsf{null}(y)$
- $(\varphi \wedge \psi)^{\mathbf{f}} = \varphi^{\mathbf{f}} \vee \psi^{\mathbf{f}}$
- $(\neg\varphi)^{\mathbf{f}} = \varphi^{\mathbf{t}}$
- $(\exists x\, \varphi)^{\mathbf{f}} = \forall x\, \varphi^{\mathbf{f}}$

Exercise: if we change $\bar{x} \Uparrow \bar{y}$ to "no x_i and y_i could be different constants" it should work too.

Exercise: rules for $x \neq y$:

$$\begin{aligned} (x \neq y)^{\mathbf{t}} &= \\ (\neg(x = y))^{\mathbf{t}} &= \\ (x = y)^{\mathbf{f}} &= \\ (x \neq y) \wedge \neg \mathsf{null}(x) \wedge \neg \mathsf{null}(y) \end{aligned}$$

and $(x \neq y)^{\mathbf{f}} = (\neg(x = y))^{\mathbf{f}} = (x = y)$ just as expected.

POSSIBLY big exercise

The translation is $\varphi(\bar{x}) \mapsto (\varphi^{+}(\bar{x}), \varphi^{?}(\bar{x}))$. The conditions it ensures are:

$$v(\varphi^{+}(D)) \subseteq \varphi(v(D)) \subseteq v(\varphi^{?}(D))$$

which ensures $\varphi^{+}(D) \subseteq \mathsf{cert}(\varphi, D)$.

Translation φ^{+}

- $(R(\bar{x}))^{+} = R(\bar{x})$
- $(x = y)^{+} = (x = y)$
- $(\varphi \wedge \psi)^{+} = \varphi^{+} \wedge \psi^{+}$
- $(\neg \varphi)^{+} = \neg \exists \bar{y} \left(\varphi^{?}(\bar{y}) \wedge \bar{x} \Uparrow \bar{y} \right)$
- $(\exists x\, \varphi)^{+} = \exists x\, \varphi^{+}$

Translation $\varphi^{?}$

- $(R(\bar{x}))^{?} = R(\bar{x})$
- $(x = y)^{?} = (x = y) \vee \mathsf{null}(x) \vee \mathsf{null}(y)$
- $(\varphi \wedge \psi)^{?} = \exists \bar{u} \exists \bar{w} \begin{pmatrix} \varphi^{?}(\bar{u}) \wedge \psi^{?}(\bar{w}) \\ \wedge\ \bar{u} \Uparrow \bar{w} \\ \wedge\ (\bar{x} = \bar{u} \vee \bar{x} = \bar{w}) \end{pmatrix}$
- $(\neg \varphi)^{?} = \varphi^{+}$
- $(\exists x\, \varphi)^{?} = \exists x\, \varphi^{?}$

Another remark: disequality. Here things get trickier. We can introduce the rules for it explicitly

- $(x \neq y)^{+} = (x \neq y) \wedge \neg \mathsf{null}(x) \wedge \neg \mathsf{null}(y)$
- $(x \neq y)^{?} = (x \neq y)$

Unlike in the previous translation, propagating existing rules does not give these formulas. In fact $(\neg(x = y))^{+}$ is

$$\neg \exists x', y' \left(\left(x' = y' \vee \mathsf{null}(x) \vee \mathsf{null}(y) \right) \wedge (x, y) \Uparrow (x', y') \right)$$

which is equivalent to false on any databases with two nulls.

Good vs perfect

reminder: what to do relax notions, one is approx, the other is prob guarantees

32

Probabilistic Databases

Probabilistic databases represent uncertainty in the data by assigning probabilities to certain aspects of it. In this section we focus on one of the simplest models of probabilistic databases in which each fact is assigned a certain value in the interval $[0, 1]$, representing the probability that such a fact belongs to the actual database. In the literature these are called *tuple-independent probabilistic databases*, but we call them simply probabilistic databases as in our case there is no room for confusion with other models of probabilistic data.

Probabilistic databases: semantics and query evaluation

A *probabilistic database* is a pair $T = (D, p)$, where D is a database over some schema $\mathbf{S}$ and p is a function that assigns a probability $p(R(\bar{a})) \in [0, 1]$ to each fact $R(\bar{a}) \in D$. In other words, the fact $R(\bar{a}) \in D$ is an independent Bernoulli random variable with probability $p(R(\bar{a}))$. Table 32.1 depicts an example of a probabilistic database T which represents a repository of news.

A probabilistic database might be in one of many states depending on which facts are assumed to be part of the actual data. Such states are known as the *possible worlds* of the probabilistic database $T = (D, p)$, and correspond to the databases D' with $D' \subseteq D$. Therefore, if D has n facts then T has exactly 2^n possible worlds. A probabilistic database $T = (D, p)$ thus defines a probability distribution Pr over the set of its possible worlds. In particular, we have that the probability of the possible world D' under P is

$$Pr(D') := \prod_{R(\bar{a}) \in D'} p(R(\bar{a})) \cdot \prod_{R(\bar{a}) \notin D'} (1 - p(R(\bar{a}))).$$

Notice that Pr is in fact a probability distribution over the set of possible worlds for T as $Pr(D') \in [0, 1]$, for each $D' \subseteq D$, and $\sum_{D' \subseteq D} Pr(D') = 1$. Continuing with our example, we have that the possible world

$$D' = \{\mathbf{News}(\text{N1},\text{'...'}),\mathbf{News}(\text{N2},\text{'...'}),\mathbf{News}(\text{N5},\text{'...'}), \\ \mathbf{Source}(\text{N1},\text{'...'}),\mathbf{Source}(\text{N4},\text{'...'})\}$$

NEWS		
News Id	**Text**	**p**
N1	'Ligers are not sterile'	0.9
N2	'Cannibalism reappears in Chile'	0.8
N3	'Habitable planet discovered'	0.7
N4	'Cyprus sinks completely'	0.2
N5	'Severe shortage of beer'	1

SOURCES			
News Id	**Media**	**Shares**	**p**
N1	Internet Research Agency	300K	0.6
N2	Gotham Globe	27K	0.9
N2	The Chippewa Bugle	950K	0.9
N3	New York Daily Inquirer	2M	0.6
N4	La Cuarta	125K	0.9
N5	Twin Peaks Gazette	350K	0.6

Table 32.1: A probabilistic table **NEWS**, which shows news with different degrees of uncertainty represented by their probability p. This value might depend on external aspects. A probabilistic table **SOURCES**, which lists news, the media where published, the number of times it has been shared, and their probability p. This value might be associated with the dataset from where the information comes from.

satisfies $Pr(D') = \prod_{R(\bar{a})\in D'} p(R(\bar{a})) \cdot \prod_{R(\bar{a})\notin D'}(1 - p(R(\bar{a})))$, where for the first term we have

$$\prod_{R(\bar{a})\in D'} p(R(\bar{a})) \;=\; 0.9 \times 0.8 \times 1 \times 0.6 \times 0.9$$

and for the second one we have

$$\prod_{R(\bar{a})\notin D'} (1 - p(R(\bar{a}))) \;=\; 0.3 \times 0.8 \times 0.1 \times 0.1 \times 0.4 \times 0.4.$$

When querying a probabilistic database we are interested in determining the probability under which a tuple belongs to the output of a certain query. More formally, given a query $q(\bar{x})$, a probabilistic database $T = (D, p)$, and a tuple $\bar{a}$ of elements in $\text{Dom}(D)$ of the same arity as $\bar{x}$, we have that the probability that $\bar{a}$ belongs to the result of q over the possible worlds of $T = (D, p)$ is

$$Pr(q, T, \bar{a}) \;:=\; \sum_{D'\subseteq D \text{ with } \bar{a}\in q(D')} Pr(D').$$

That is, $Pr(q, D, \bar{a})$ is the weighted sum (under Pr) of all possible worlds represented by T. For instance, in our example we have that the evaluation of the CQ

$$q(x,y) \;=\; \exists z\,\big(\mathbf{News}(x,y) \wedge \mathbf{Source}(x, \text{‘La Cuarta’}, z)\big),$$

which selects news and texts that have been published in ‘La Cuarta’, outputs tuple (N4,‘Cyprus sinks completely’) with probability 0.2×0.9.

Complexity of probabilistic query evaluation

Given a query language $\mathcal{L}$, we are interested in the computation problem $\mathcal{L}$-ProbEvaluation which is defined as follows.

PROBLEM:	$\mathcal{L}$-ProbEvaluation
INPUT:	a query q from $\mathcal{L}$, a probabilistic database $T = (D, p)$, a tuple $\bar{a}$ over $\mathrm{Dom}(D)$
OUTPUT:	the value $Pr(q, T, \bar{a})$

Solving $\mathcal{L}$-ProbEvaluation is, in general, a difficult task because it requires reasoning over the exponentially-sized set of possible worlds represented by T. In fact, we show next that the problem is not solvable in polynomial time (under widely held complexity theoretical assumptions) even for a *fixed* boolean CQ. In other words, not even in data complexity CQ-ProbEvaluation can be solved efficiently.

The result is stated in terms of the class #P, which consists of those computation problems whose solution corresponds to the number of accepting witnesses for an NP problem. Clearly, NP is contained in #P, and thus if a problem is #P-hard it cannot be solved in polynomial time under the assumption that PTIME $\neq$ NP. We will show that CQ-ProbEvaluation is #P-hard in data complexity (under polynomial time *Turing reductions*), thus showing that if PTIME $\neq$ NP then this problem cannot be solved efficiently.

Of course, if a decision problem is NP-hard then its associated computation problem “compute the number of accepting witnesses on a given instance” will be #P-hard. These include, for instance, counting the number of satisfying assignments of a propositional formula in CNF or counting the number of 3-colorings of a graph. But, more interestingly, there are decision problems which can be solved in PTIME such that its associated computation problem is #P-hard (and thus it is not efficiently solvable under complexity assumptions). These include, e.g., the satisfiability problem for propositional formulas in DNF or the existence of non-independent sets in bipartite graphs. While these problems can be solved in PTIME, both counting the number of satisfying assignments of a DNF formula and the number of non-independent sets in a bipartite graph are #P-hard problems.

Next we establish the intractability of the computation problem CQ-ProbEvaluation in data complexity. For simplicity, if q is a fixed CQ we denote by ProbEvaluation(q) the problem of computing $Pr(q, T, \bar{a})$ on an input given by a probabilistic database $T = (D, p)$ and a tuple $\bar{a}$ over $\mathrm{Dom}(D)$.

Theorem 32.1. *The problem* CQ-ProbEvaluation *is* #P*-hard even in data complexity. More precisely, there is a fixed Boolean CQ q such that the problem* ProbEvaluation(q) *is* #P*-hard.*

Proof. We define the boolean CQ q to be $\exists x \exists y(\mathsf{Part}_1(x) \wedge \mathsf{edge}(x,y) \wedge \mathsf{Part}_2(y))$. We show that the problem of computing $Pr(q,T)$, given a probabilistic database T, is #P-hard. To do this we provide a polynomial time Turing reduction from the problem of counting non-independent sets in bipartite graphs. In other words, we construct a polynomial time procedure that, given a bipartite graph $G = (V,E)$, it first constructs a database D and then computes the number of non-independent in G by having access to an oracle for $Pr(q,D)$.

Let us assume that the bipartiteness of G is witnessed by the partition $U \cup W = V$, where $U = \{u_1, \ldots, u_n\}$, $W = \{w_1, \ldots, w_m\}$, and $U \cap W = \emptyset$. In particular, $E \subseteq U \times W$. From G we construct in polynomial time a database D containing the following facts:

- $\mathsf{Part}_1(u_i)$, for each $i \in [1,n]$.
- $\mathsf{Part}_2(w_i)$, for each $i \in [1,m]$.
- $\mathsf{edge}(u_i, w_j)$, for each $i \in [1,n]$ and $j \in [1,m]$ with $(u_i, w_j) \in E$.

We then define a probabilistic database $T = (D,p)$, where $p(\mathsf{Part}_1(u_i)) = p(\mathsf{Part}_2(w_j)) = 1/2$, for each $i \in [1,n]$ and $j \in [1,m]$, and $p(\mathsf{edge}(u_i,w_j)) = 1$, for each $(u_i,w_j) \in E$.

Notice that each possible world D' of $T = (D,p)$ satisfies $\mathsf{Part}_1^{D'} \subseteq \mathsf{Part}_1^{D}$, $\mathsf{Part}_2^{D'} = \mathsf{Part}_2^{D}$, and $\mathsf{edge}^{D'} = \mathsf{edge}^{D}$. Hence, there is a one-to-one correspondence between the possible worlds of T and the ways in which both a subset of U and a subset of W can be chosen. Moreover, the set $\mathsf{Part}_1^{D'} \cup \mathsf{Part}_2^{D'}$ defines an independent set in G if and only if $q(D') = \mathsf{false}$. It follows that the number of non-independent sets in G is precisely the number of possible worlds D' such that $q(D') = \mathsf{true}$. Therefore, the number of non-independent sets in G is $Pr(q,T)/2^{n+m}$. Consequently, we can use $Pr(q,T)$ as an oracle to compute in polynomial time the number of non-independent sets in G. □

The previous result establishes a stark difference between the two models of uncertainty we have seen so far. In fact, while computing certain answers for CQs over incomplete databases can be solved in polynomial time in data complexity via naïve evaluation, the data complexity of CQ-ProbEvaluation is intractable.

Two lines of research have been pursued to alleviate this situation. The first one corresponds to the identification of classes of CQs for which ProbEvaluation can be solved efficiently in data complexity. The second one corresponds to the design of polynomial time approximation algorithms that return a solution to ProbEvaluation which lies within a distance from the optimal one with provable probabilistic guarantees. We present both approaches below.

A maximal tractable class of conjunctive queries

From now on we focus on the data complexity of computing answers to CQs over probabilistic databases. To simplify the presentation, when q is a fixed CQ we write $\mathsf{ProbEvaluation}(q)$ for the problem of computing $Pr(q, T, \bar{a})$ on an input given by a probabilistic database $T = (D, p)$ and a tuple $\bar{a}$ over $\mathrm{Dom}(D)$.

The literature on probabilistic databases has discovered some surprising *dichotomies* regarding the complexity of the problems $\mathsf{ProbEvaluation}(q)$, for q a CQ. These dichotomies tell us that this problem is either polynomial-time solvable or #P-hard, and one can decide given q which one of the two cases holds. Such dichotomies are much easily described and obtained for the so-called *self-join free* CQs (sjfCQs), and this is the approach we follow in this section. Formally, an sjfCQ is a CQ in which no two distinct atoms use the same relation symbol. For instance, $\exists x \exists y \exists z (R(x, y) \wedge S(y, z))$ is an sjfCQ, while $\exists x \exists y \exists z (R(x, y) \wedge R(y, z))$ is not.

Consider an sjfCQ q. For each variable $x \in \mathrm{Dom}(q)$ we denote by $\mathsf{atoms}(q, x)$ the set of atoms in q where x appears. We call q *hierarchical* if for every existentially quantified variables $x, y \in \mathrm{Dom}(q)$ one of the following statements hold:

- $\mathsf{atoms}(q, x) \subseteq \mathsf{atoms}(q, y)$,
- $\mathsf{atoms}(q, y) \subseteq \mathsf{atoms}(q, x)$, or
- $\mathsf{atoms}(q, x) \cap \mathsf{atoms}(q, y) = \emptyset$.

Notice that this is just a syntactic condition that can easily be checked on q.

As an example, it is easy to see that the sjfCQ

$$q_{\mathrm{h}} \;=\; \exists x \exists y \exists z\, (R(x, y) \wedge S(y, z))$$

is hierarchical, but the sjfCQ

$$q_{\mathrm{nh}} \;=\; \exists x \exists y\, (U(x) \wedge E(x, y) \wedge W(y))$$

used in the proof of Theorem 32.1 is not. This is because $\mathsf{atoms}(q, x) \cap \mathsf{atoms}(q, y) \neq \emptyset$, and neither $\mathsf{atoms}(q, x) \subseteq \mathsf{atoms}(q, y)$ nor $\mathsf{atoms}(q, y) \subseteq \mathsf{atoms}(q, x)$.

Recall from the proof of Theorem 32.1 that for the non-hierarchical CQ q_{nh} the problem $\mathsf{ProbEvaluation}(q_{\mathrm{nh}})$ is #P-hard. This is in fact a consequence of a more general phenomenon: the only sjfCQsq for which $\mathsf{ProbEvaluation}(q)$ can be solved in polynomial time are the hierarchical ones.

Theorem 32.2. *Let q be a* sjfCQ.

- *If q is hierarchical, then* $\mathsf{ProbEvaluation}(q)$ *can be solved in polynomial time.*

- *Otherwise,* $\mathsf{ProbEvaluation}(q)$ *is* #P*-hard.*

We only explain how to prove the first part. We develop a recursive algorithm $\textsc{HierEval}(q, T, \bar{a})$ that computes $Pr(q, T, \bar{a})$ in polynomial time, for q a sjfCQ $q(\bar{x})$ that is hierarchical. Consider an input to $\mathsf{ProbEvaluation}(q)$ given by a probabilistic database $T = (D, p)$ and a tuple $\bar{a} \in \mathrm{Dom}(D)^{|\bar{x}|}$. If q contains no existentially quantified variables, then compute the database $D_{q,\bar{a}}$ obtained from q by replacing each variable $x \in \bar{x}$ with its corresponding element $a \in \bar{a}$. It is clear then that

$$Pr(q, T, \bar{a}) \ = \prod_{R(\bar{b}) \in D_{q,\bar{a}}} p(R(\bar{b})),$$

and hence $\textsc{HierEval}(q, T, \bar{a})$ simply outputs this value (that can be computed in polynomial time).

Assume now that there are q existentially quantified variables. Since q is hierarchical, there must be at least one variable $z \in \mathrm{Dom}(q)$ such that $\mathsf{atoms}(q, z)$ is maximal (with respect to set inclusion). Suppose first that z does not appear in every atom of q. Consider the set $D[q] \setminus \mathsf{atoms}(q, z)$ of atoms in q that are not in $\mathsf{atoms}(q, z)$. Notice that $D[q] \setminus \mathsf{atoms}(q, z)$ shares no variable with $\mathsf{atoms}(q, z)$. In fact, if any such a variable x exists we would have that $\mathsf{atoms}(q, z) \cap \mathsf{atoms}(q, x) \neq \emptyset$ and $\mathsf{atoms}(q, x) \not\subseteq \mathsf{atoms}(q, z)$. But q is hierarchical and hence $\mathsf{atoms}(q, z) \subsetneq \mathsf{atoms}(q, x)$. This implies that $\mathsf{atoms}(q, z)$ is not maximal, which is a contradiction. In consequence, $\textsc{HierEval}(q, T, \bar{a})$ can be recursively solved as

$$\textsc{HierEval}(q_1, T, \bar{a}) \cdot \textsc{HierEval}(q_2, T, \bar{a}),$$

where $q_1(\bar{x})$ and $q_2(\bar{x})$ are the CQs defined by the atoms in $D[q] \setminus \mathsf{atoms}(q, z)$ and $\mathsf{atoms}(q, z)$, respectively. This works as it can easily be proved that both q_1 and q_2 are hierarchical sjfCQs and, in addition, $Pr(q_1, T, \bar{a})$ is independent from $Pr(q_2, T, \bar{a})$ since there are no self-joins in q.

Suppose now that z appears in every atom of q. Let $q'(\bar{x}, z)$ be the sjfCQ which is obtained from q by removing the existential quantification on z. Clearly, q' continues being a hierarchical sjfCQ. Then $\textsc{HierEval}(q, T, \bar{a})$ can be recursively solved as

$$1 \ - \prod_{b \in \mathrm{Dom}(D)} \big(1 - \textsc{HierEval}(q', T, (\bar{a}, b))\big).$$

This is because

$$Pr(q, T, \bar{a}) \ = \ 1 - Pr(\{D' \not\models q'(\bar{a}, b), \text{for every } b \in \mathrm{Dom}(D) \mid D' \subseteq D\}),$$

and, in addition, for any two distinct $b, b' \in \mathrm{Dom}(D)$ it is the case that $Pr(\{D' \not\models q'(\bar{a}, b) \mid D' \subseteq D\})$ is independent from $Pr(\{D' \not\models q'(\bar{a}, b') \mid D' \subseteq D\})$. The latter holds since q contains no self-joins.

It can readily be seen that the procedure HIEREVAL$(q, T, \bar{a})$ can be implemented in polynomial time. This is because each recursive step makes at most polynomially many calls in the size of Dom(D), and the depth of the recursion is bounded by the size of Dom(q), which is fixed.

An approximation algorithm

While exact evaluation of CQs over probabilistic databases is intractable even in data complexity, it is possible to develop efficient approximation algorithms for the problem with provable probabilistic guarantees. More specifically, it can be shown that for every fixed CQ $q(\bar{x})$, the problem of computing $Pr(q, T, \bar{a})$ on a given probabilistic database $T = (D, p)$ and tuple $\bar{a}$ in Dom(D) has a *fully polynomial randomized approximation scheme* (FPRAS). Intuitively, this means that there is a randomized algorithm that produces a $(1 \pm \epsilon)$-approximation of $Pr(q, T, \bar{a})$ with probability at least $1 - \delta$, for any $\epsilon > 0$ and $0 < \delta < 1$, in polynomial time in $||T||$, $1/\epsilon$, and $\ln(1/\delta)$. More formally, there is a polynomial time algorithm $\mathcal{A}_q$ that, given a probabilistic database $T = (D, p)$, a tuple $\bar{a}$ in Dom(D), and values $\epsilon > 0$ and $0 < \delta < 1$, computes in polynomial time in $||T||$, $1/\epsilon$, and $\ln(1/\delta)$ a value $\mathcal{A}_q(T, \bar{a})$ such that the probability of the following event is at least $1 - \delta$:

$$(1-\epsilon)Pr(q, T, \bar{a}) \;\leq\; \mathcal{A}_q(T, \bar{a}) \;\leq\; (1+\epsilon)Pr(q, T, \bar{a}).$$

Theorem 32.3. *Let $q(\bar{x})$ be a fixed CQ. There is an FPRAS for the problem of computing $Pr(q, T, \bar{a})$ on a given probabilistic database $T = (D, p)$ and tuple $\bar{a}$ in Dom(D).*

Proof. Let us assume that $q(\bar{x}) = \exists \bar{\varphi}(\bar{x}, \bar{y})$. We start by computing the set $\mathsf{Hom}(q, D, \bar{a})$ of all homomorphisms from q to D with $h(\bar{x}) = \bar{a}$. This can be done in polynomial time as q is fixed, and hence the number of mappings from q to D is polynomially bounded by $||D||^{O(||q||)}$. For each set of facts of the form $\varphi(\bar{a}, h(\bar{y}))$, where $h \in \mathsf{Hom}(q, D, \bar{a})$, let E_h be the event that consists of those possible worlds D' that contain all facts in $\varphi(\bar{a}, h(\bar{y}))$. Consequently, $Pr(E_h)$ is the product of the probabilities of the tuples in $\varphi(\bar{a}, h(\bar{y}))$ and this value can be computed in polynomial time in $||T||$.

Consider an enumeration $h_1, \ldots, h_t$ of the homomorphisms in $\mathsf{Hom}(q, D, \bar{a})$. Observe that $Pr(q, T, \bar{a}) = Pr(E_{h_1} \cup \cdots \cup E_{h_t})$. We develop an FPRAS for computing the expression $N = Pr(E_{h_1} \cup \cdots \cup E_{h_t})$ by tightly following the technique developed by Karp and Luby for approximately counting solutions to propositional formulas in DNF.

Let us define $M = Pr(E_{h_1}) + \cdots + Pr(E_{h_t})$. Notice that the value M satisfies $M \geq N$ and can be computed in polynomial time. Also, we define a set $S = \{(j, D') \mid j \in [1, t],\, D' \in E_{h_j}\}$ and a $G \subseteq S$ that consists of those pairs $(j, D') \in S$ such that for no $k < j$ we have that $D' \in E_{h_k}$. We refer to pairs in G as *good*. The algorithm consists in sampling s pairs from S by using the following distribution: We first choose $j \in [1, t]$ with probability $Pr(E_{h_j})/M$

and then $D' \in E_{h_j}$ with probability $Pr(D')/Pr(E_{h_j})$. Each sample can be picked in polynomial time in $||T||$. This is because all probabilities involved can be computed in polynomial time and, further, checking whether $D' \in E_{h_j}$ is in PTIME. Let g be the number of good pairs in our sample; these are the ones from G. The randomized algorithm then returns the value $\mathcal{A}_q(T, \bar{a}) = (g/s)M$ as an approximation for $Pr(q, T, \bar{a})$. Checking if a pair is good can be done in polynomial time in t and s, and thus in $||T||$ and s, and hence computing the value of $\mathcal{A}_q(T, \bar{a})$ is in PTIME. We show that there is a polynomial value s for which this algorithm is in fact an FPRAS for computing the value of N.

We establish the following properties.

- Let D' be a possible world. Then any sample of the form (j, D'), for $j \in [1, t]$, is chosen with the same probability $Pr(D')/M$.
- Let us interpret the s samples as indicator variables $X_1, \ldots, X_s$, such that for each $i \in [1, s]$ we have that

  $$X_i \;:=\; \begin{cases} 1 & \text{if the } i\text{-th sample is good,} \\ 0 & \text{otherwise.} \end{cases}$$

  Notice that the expected value $\mathbb{E}(X_i)$ of an arbitrary variable X_i, for $i \in [1, s]$, is $Pr(X_i = 1) = |G|/|S| = N/M$. Also, the output of the algorithm described above is

  $$\mathcal{A}_q(T, \bar{a}) \;=\; \frac{gM}{s} \;=\; \frac{(X_1 + \cdots + X_s)M}{s}.$$

  Therefore, we have that the the expected value $\mathbb{E}(\mathcal{A}_q(T, \bar{a}))$ of this output is N. This means that the output of the algorithm is an unbiased estimator for $Pr(q, T, \bar{a})$.
- We have that $N/M \geq 1/t$. This is because for every good pair $(j, D') \in G$ there are at most t pairs of the form (k, D') which are not good (those with $k > j$). The claim then follows since the probability of choosing any pair of the form (j, D'), for a particular $D' \subseteq D$, is uniform.
- We now apply Chernoff's bound, which in this case tells us that if

  $$s \;\in\; O(\frac{M}{\epsilon^2 N} \cdot \ln \frac{1}{\delta})$$

  we then have that the probability of the following event is at least $(1 - \delta)$:

  $$(1 - \epsilon)N \;\leq\; \mathcal{A}_q(T, \bar{a}) \;\leq\; (1 + \epsilon)N.$$

  This implies that the estimator computed by the algorithm is within the required distance from the actual value $Pr(q, T, \bar{a})$.

- The number s of samples we need to take is also $O(\frac{t}{\epsilon^2} \cdot \ln \frac{1}{\delta})$, which is

  $$O(\frac{||T||}{\epsilon^2} \cdot \ln \frac{1}{\delta}).$$

  This is because $N/M \geq 1/t$ and t is polynomial in $||T||$ for reasons explained above. It follows that s is polynomial in $||T||$, $1/\epsilon$, and $\ln(1/\delta)$, and thus that the algorithm is an FPRAS.

This finishes the proof of Theorem 32.3. □

33

Consistent Query Answering

Let D be a database and Σ a set of constraints over the same schema $\mathbf{S}$. We call D *inconsistent* with respect to Σ if $D \not\models \Sigma$, that is, there is at least one constraint in Σ such that D does not satisfy. Here we focus on the case when Σ is a set of primary keys. Recall that this means that each $R \in \mathbf{S}$ comes equipped with its own key, i.e., $key(R) = A$, where $A = \emptyset$ or $A = [1, \ldots, p]$ for some $p \in [1, ar(R)]$. Henceforth, D is inconsistent with respect to Σ if there is a symbol $R \in \mathbf{S}$ and facts $R(\bar{a}), R(\bar{b}) \in D$ such that

$$key(R) = A \quad \text{and} \quad \pi_A(\bar{a}) = \pi_A(\bar{b}).$$

In this case we call the pair $(R(\bar{a}), R(\bar{b}))$ to be *key-violating*.

When D is inconsistent with respect to a set Σ of primary keys, it is always possible to restore consistency by deleting some of the tuples from D. The idea, of course, is to do so without deleting more information than is needed. As an example, consider the relational table **SOURCES** shown in Table 33.1, and assume that **News Id** is the key for the table. Then clearly **SOURCES** is inconsistent with respect to this key. However, the consistency can be restored by deleting one tuple with key N1 and one tuple with key N4. We could also delete all tuples with key N1 and all tuples with key N4, which would yield a database that satisfies the key, but this would violate our second requirement: we would be deleting more information than is needed to restore consistency.

We can generalize this process as follows. Let us define a *block* in a database D with respect to a set Σ of primary keys to be any maximal set B of facts from D such that the facts in B are pairwise key-violating. For instance, in the previous example the table **SOURCES** contains precisely the five blocks listed below:

- $B_1 = \{(\text{N1}, \text{Internet Research Agency}, \text{300K})\}$.
- $B_2 = \{(\text{N2}, \text{Gotham Globe}, \text{27K}),\ (\text{N2}, \text{The Chippewa Buggle}, \text{950K})\}$.
- $B_3 = \{(\text{N3}, \text{New York Daily Enquirer}, \text{2M})\}$.

SOURCES		
News Id	**Media**	**Shares**
N1	Internet Research Agency	300K
N2	Gotham Globe	27K
N2	The Chippewa Bugle	950K
N3	New York Daily Inquirer	2M
N4	La Cuarta	125K
N4	Fortín Mapocho	500K
N5	Twin Peaks Gazette	350K

Table 33.1: A table **SOURCES**, which is inconsistent with respect to the key constraint that establishes that no two tuples can have the same value for **News Id**.

- $B_4 = \{(\text{N4}, \text{La Cuarta}, 125\text{K}),\ (\text{N4}, \text{Fortín Mapocho}, 500\text{K})\}$.
- $B_5 = \{(\text{N5},\ \text{Twin Peaks Gazette}, 350\text{K})\}$.

It is clear that to restore consistency from D with respect to Σ we need to delete facts from D until leaving at most one tuple in each block B of D. In addition, in order to do it in a minimal way we have to leave at least one fact in each such a block. This gives rise to the notion of a *repair* of D with respect to Σ. This is a subset D' of D that is obtained by choosing exactly one tuple from each block of D with respect to Σ. The notion of repair thus satisfies the following important property, which establish that they are precisely the maximally consistent subsets of D with respect to Σ.

Proposition 33.1. *Let D be a database and Σ a set of primary keys. The repairs of D with respect to Σ are precisely the D' with $D' \subseteq D$ such that:*

- *$D' \models \Sigma$, and*
- *there is no D'' with $D' \subsetneq D'' \subseteq D$ such that $D'' \models \Sigma$.*

Notice that repairs always exist and, if $D \models \Sigma$, then the only repair of D with respect to Σ is D itself. Moreover, the number of repairs can be exponential in the number of tuples of D (e.g., if D has n blocks, and each block has two facts, then D has 2^n repairs).

When querying a database D that is inconsistent with respect to a set Σ of primary keys, we are interested in obtaining those results that are true regardless of how consistency is restored. These are the *consistent answers*, which are those answers that are true in every possible repair of D with respect to Σ. Formally, given a query $q(\bar{x})$, a database D, and a tuple $\bar{a}$ of elements in $\text{Dom}(D)$ of the same arity as $\bar{x}$, we call $\bar{a}$ a consistent answer to q over (D, Σ) if $\bar{a} \in q(D')$, for every repair D' of D with respect to Σ. We then define

$$\text{CQA}(q, D, \Sigma) \;=\; \{\bar{a} \mid \bar{a} \text{ is a consistent answer to } q \text{ over } (D, \Sigma)\}.$$

When q is boolean, we abuse notation write $\mathrm{CQA}(q, D, \Sigma) = \mathsf{true}$ to denote that the empty tuple () belongs to $\mathrm{CQA}(q, D, \Sigma)$; i.e., $q(D') = \mathsf{true}$ for every repair D' of D with respect to Σ.

For instance, in our example we have that the set of consistent answers to the CQ $q(x) = \exists y \exists z\, \mathbf{Sources}(x, y, z)$ is the set $\{\mathrm{N1}, \mathrm{N2}, \mathrm{N3}, \mathrm{N4}, \mathrm{N5}\}$, and to the CQ $q(x, y) = \exists z\, \mathbf{Sources}(x, y, z)$ is the set

$$\{(\mathrm{N1}, \text{Internet Research Agency}), (\mathrm{N3}, \text{New York Daily Enquirer}), \\ (\mathrm{N5},\ \text{Twin Peaks Gazette})\}.$$

Complexity of consistent query answering

Given a query language $\mathcal{L}$, we are interested in the decision problem $\mathcal{L}$-$\mathsf{ConsEvaluation}$ which is defined as follows.

PROBLEM:	$\mathcal{L}$-$\mathsf{ConsEvaluation}$
INPUT:	a query q from $\mathcal{L}$, a set Σ of primary keys, a database D, a tuple $\bar{a}$ over $\mathrm{Dom}(D)$
QUESTION:	is $\bar{a} \in \mathrm{CQA}(q, D, \Sigma)$?

As for the case of probabilistic query evaluation, we can show that the problem $\mathcal{L}$-$\mathsf{ConsEvaluation}$ is computationally hard even for the class of CQs and in data complexity (where we assume both the actual query and the set of keys to be fixed). For simplicity, if q is a fixed CQ and Σ a fixed set of constraints we denote by $\mathsf{ConsEvaluation}(q, \Sigma)$ the problem of checking if $\bar{a} \in \mathrm{CQA}(q, D, \Sigma)$ on an input given by a database D and a tuple $\bar{a}$ over $\mathrm{Dom}(D)$. We can then show the following result.

Theorem 33.2. *The problem* CQ-$\mathsf{ConsEvaluation}$ *is* coNP*-hard even in data complexity. More precisely, there are a fixed Boolean CQ q and a fixed set Σ of primary keys such that* $\mathsf{ConsEvaluation}(q, \Sigma)$ *is* coNP*-hard.*

Proof. We define the boolean CQ q to be $\exists x \exists y (\mathsf{Col}(x, x') \wedge \mathsf{edge}(x, y) \wedge \mathsf{Col}(y, y) \wedge \mathsf{Bad}(x', y'))$ and Σ to consist exclusively of the constraint that establishes the first attribute of col to be a key; that is, $\Sigma = \{key(\mathsf{col}) = 1\}$. We show that there is a polynomial time reduction from $\mathsf{3\text{-}Col}$, the problem of checking if a graph is 3-colorable, to the complement of the problem of checking whether $\mathrm{CQA}(q, D, \Sigma) = \mathsf{true}$, given a database D.

Let $G = (V, E)$ be a graph. We construct a database D over the schema of q as follows. The domain of D is defined as the disjoint union of V and three fresh constants 0, 1, and 2. The relation edge^D contains all edges $(v, w) \in E$. Second, the relation col^D contains all pairs $(v, 0)$, $(v, 1)$, and $(v, 2)$, for $v \in V$. Finally, the relation Bad^D contains the pairs $(0, 0)$, $(1, 1)$, and $(2, 2)$. Since the first component of col is the key of this relation, each repair of D defines an assignment of a color in $\{0, 1, 2\}$ to each node $v \in V$. By definition, any such an assignment corresponds to a proper 3-coloring of the graph G if there are no two nodes v, w that are adjacent in G which are assigned the same color. It is easy to see then that G is 3-colorable if and only if there is a repair D' of D that does not satisfy q, i.e., it is not the case that $\mathrm{CQA}(q, D, \Sigma) = \mathsf{true}$. $\square$

Hence, consistent query answering resembles query evaluation over probabilistic databases in terms of the complexity of evaluation for CQs: Both problems are intractable even in data complexity. As we did for CQ evaluation over probabilistic databases, we present next two lines of research that have been developed to tackle this issue. The first one corresponds to identifying classes of CQs for which the problem can be solved efficiently. The second one corresponds to the design of an FPRAS for computing the "degree of certainty" with which a CQ holds over an inconsistent database. This degree of certainty refers to the number of repairs that satisfy the given CQ.

A tractable class of CQs

We focus on the class of self-join free CQs (sjfCQs), which was studied in the previous chapter, since this case is much better understood in the literature and results for it are much easier to explain than for the general case. Recall that a sjfCQ q is a CQ in which no two distinct atoms use the same relation symbol. We identify a syntactically restricted class of pairs (q, Σ), where q is a sjfCQ and Σ is a set of primary keys, for which $\mathsf{ConsEvaluation}(q, \Sigma)$ can be solved in polynomial time. This class is defined in terms of the *key-join* graphs of pairs of the form (q, Σ), a notion that we introduce below. To simplify the presentation, we consider Boolean sjfCQs only.

Let $R(\bar{x})$ be an atom over a schema $\mathbf{S}$ with $key(R) = A$. The tuple of variables in *key positions* of $R(\bar{x})$ is defined as $\emptyset$, if $A = \emptyset$, and as $(x_1, \ldots, x_p)$, if $A = [1, p]$ for $p \in [1, ar(R)]$. We often write $R(\bar{x})$ as $R(\bar{y}\,;\bar{z})$ to denote that $\bar{y}$ and $\bar{z}$ are the tuples of variables in key and non-key positions of $R(\bar{x})$, respectively. A Boolean sjfCQ $R_1(\bar{x}_1), \ldots, R_m(\bar{x}_m)$ together with a set Σ of primary keys can thus be represented in a single expression:

$$R_1(\bar{y}_1\,;\,\bar{z}_1), \ldots, R_m(\bar{y}_m\,;\,\bar{z}_m). \tag{33.1}$$

We call these expressions *constrained* sjfCQs and often denote them $\hat{q}$. We write $\mathsf{ConsEvaluation}(\hat{q})$ as a shorthand for the problem $\mathsf{ConsEvaluation}(q, \Sigma)$, where q and Σ are the CQ and set of primary keys, respectively, that are univocally associated with $\hat{q}$.

Let $\hat{q}$ be a Boolean constrained sjfCQ of the form (33.1). By definition of self-join freeness, the R_is are pairwise different. The key-join free graph of $\hat{q}$ is a directed graph whose nodes are the atoms $R_i(\bar{y}_i\,;\,\bar{z}_i)$ of $\hat{q}$ and there is a directed edge from $R_i(\bar{y}_i\,;\,\bar{z}_i)$ to $R_j(\bar{y}_j\,;\,\bar{z}_j)$, for $i \neq j$, if there is a variable in $(\bar{y}_j\,;\,\bar{z}_j)$ that appears in $\bar{z}_i$, i.e., in a non-key position of $R_i(\bar{y}_i\,;\,\bar{z}_i)$.

Example 33.3. Consider the constrained sjfCQ $\hat{q}_1 :- R(\underline{x}\,;z), S(\underline{z}\,;x)$, where we have underlined key positions for clarity. The key-join graph of $\hat{q}_1$ is shown below:

$$\begin{tikzcd} R(\underline{x}\,;z) \arrow[r, bend right, "z"'] & S(\underline{z}\,;x) \arrow[l, bend right, "x"'] \end{tikzcd}$$

The labels in the edges represent the non-key positions in the source that appear in the target.

Consider also the CQ $\hat{q}_2 :- R(\underline{x}\,;z), S(\underline{z,v}\,;w)$. The key-join graph of $\hat{q}_2$ is shown in the following figure:

$$\begin{tikzcd} R(\underline{x}\,;z) \arrow[r, "z"'] & S(\underline{z,v}\,;w) \end{tikzcd}$$

□

Following with the previous example, we have that both ConsEvaluation$(\hat{q}_1)$ and ConsEvaluation$(\hat{q}_2)$ are coNP-complete problems (this fact is left as an exercise to the reader). Notice that the key-join graph of $\hat{q}_1$ contains a directed cycle, and hence for the tractable family of CQs we aim to define in this section we can only admit constrained sjfCQs with an acyclic key-join graph. Yet, this is not sufficient. In fact, the key-join graph of $\hat{q}_2$ is acyclic. In this case, the reason why ConsEvaluation$(\hat{q}_2)$ is coNP-complete is found in the label of the edge that goes from $R(x\,;z)$ to $S(z,v\,;w)$: this does not include all variables that appear in key positions in the second atom, in particular, the variable v is not present in such a label. In technical terms, it is said that the join from the key of $S(z,v\,;w)$ to $R(x\,;z)$ is *non-full*. Henceforth, our tractable class neither can admit non-full joins in edges of key-join graphs.

As we establish below, these two restrictions suffice to guarantee tractability for the problem ConsEvaluation$(\hat{q})$. Formally, let us define $\mathcal{C}_{\text{ac-full}}$ as the set of Boolean constrained sjfCQs $\hat{q}$ for which the following statements hold:

- the key-join graph of $\hat{q}$ contains no directed cycles, and
- every edge of the key-join graph of $\hat{q}$ is *full*, that is, each such an edge of the form $R(\bar{x}\,;\bar{y}) \to S(\bar{z}\,;\bar{w})$ satisfies that every variable that appears in the tuple $\bar{z}$ of key positions in $S(\bar{z}\,;\bar{w})$ also appears in the tuple $\bar{y}$ of non-key positions in $R(\bar{x}\,;\bar{y})$.

Example 33.4. Consider the constrained sjfCQ

$$\hat{q} :- R(\underline{x,z}\,;u,x), S(\underline{u,x}\,;v,w), M(\underline{u}\,;t), L(\underline{t}\,;y).$$

The key-join graph of $\hat{q}$ is shown in the following figure:

$$\begin{tikzcd}
R(\underline{x,z}\,;u,x) \arrow[r, "\{u,x\}"] \arrow[dr, "\{u\}"'] & S(\underline{u,x}\,;v,w) & \\
 & M(\underline{u}\,;t) \arrow[r, "\{t\}"] & L(\underline{t}\,;y)
\end{tikzcd}$$

It can be observed that the key-join graph of $\hat{q}$ is acyclic and all edges are full. Hence, $\hat{q}$ in $\mathcal{C}_{\text{ac-full}}$. □

The following is an important observation regarding the class $\mathcal{C}_{\text{ac-full}}$. We leave the proof of this fact to the reader.

Lemma 33.5. *For every $\hat{q} \in \mathcal{C}_{\text{ac-full}}$ we have that the key-join graph of $\hat{q}$ must be a directed forest, i.e., every node has at most one incoming edge.*

The main result of this section is the following, which establishes the tractability of computing certain answers for the constrained CQs in $\mathcal{C}_{\text{ac-full}}$. It should be noted that this is also a meaningful result from a practical point of view, as it has been observed that constrained sjfCQs in the class $\mathcal{C}_{\text{ac-full}}$ occur often in real world scenarios.

Theorem 33.6. *Let $\hat{q}$ be a fixed Boolean constrained* sjfCQ *in $\mathcal{C}_{\text{ac-full}}$. The problem* ConsEvaluation($\hat{q}$) *can be solved in* PTIME.

Proof. We know from Lemma 33.5 that the key-join graph of $\hat{q}$ is a directed forest. For simplicity, we assume that it actually consists of a single directed tree T; the general case is slightly more involved and left as an exercise. Each node t of T is thus associated with an atom $R_t(\bar{x}_t\,;\bar{y}_t)$ of $\hat{q}$. We denote by $\hat{q}_t$ the constrained sjfCQ that is induced in $\hat{q}$ by all the atoms of the form $R_u(\bar{x}_u\,;\bar{y}_u)$, for u a (not necessarily proper) descendant of t in T. We proceed by showing that each such a constrained sjfCQ $\hat{q}_t$ admits an FO-*rewriting*. More formally there is an FO sentence ψ_t such that for every database D we have that:

$$\text{CQA}(\hat{q}_t, D) = \mathsf{true} \qquad \Longleftrightarrow \qquad D \models \psi_t.$$

Here CQA($\hat{q}_t, D$) is a shortening for CQA(q_t, D, Σ_t), where q_t and Σ_t are the sjfCQ and set of primary keys, respectively, that are univocally associated with $\hat{q}_t$. That is, the certain answer to q_t over D with respect to Σ_t can be obtained by directly evaluating the rewriting ψ_t over D. In particular, if r is the root of T then $\hat{q}_r = \hat{q}$, and hence $\hat{q}$ itself admits an FO-rewriting. Theorem 33.6 then follows since FO sentences can be evaluated in DLOGSPACE in data complexity from Theorem 7.2.

We start by proving the following claim. Here, $q'_t(\bar{x}_t)$ denotes the sjfCQ which is obtained from q_t by removing the existential quantification on $\bar{x}_t$.

Claim 33.7. *For every node t of T there is an FO formula $\psi'_t(\bar{x}_t)$ such that, for every database D and tuple $\bar{a}$ over Dom(D) with $|\bar{a}| = |\bar{x}_t|$, the following statements are equivalent:*

- *$D \models \psi'_t(\bar{a})$.*
- *For every repair D' of D with respect to Σ_t it is the case that $\bar{a} \in q'_t(D')$.*

We prove the claim by induction on the height of t. If t has height 0, then it is a leaf. We illustrate the proof with an example. Suppose that the atom $R_t(\bar{x}_t\,;\bar{y}_t)$ is $R(x, y\,; x, v)$. Then $\psi'_t(x, y)$ can be defined as:

$$\exists v\, R(x, y, x, v) \wedge \forall u \forall v\, \big(R(x, y, u, v) \rightarrow u = x\big).$$

In fact, it is easy to see that for every database D and elements $a, b \in \mathrm{Dom}(D)$ the following holds: for every repair D' of D with respect to Σ_t we have that $(a, b) \in q'_t(D')$ if and only if there is a fact of the form $R(a, b, a, d) \in D$, for some $d \in \mathrm{Dom}(D)$, and each fact in the same block than $R(a, b, a, d)$ with respect to Σ_t is of the form $R(a, b, a, d')$, for some $d' \in \mathrm{Dom}(D)$. Clearly, the latter holds if and only if $D \models \psi'_t(a, b)$. The proof of the general case, in which t is associated with an arbitrary atom of the form $R_t(\bar{x}_t\,;\bar{y}_t)$, is left to the reader.

Assume now that t has height $h+1$, for $h \geq 0$. Let $t_1, \dots, t_k$ be the children of t in T. We then define $\psi'_t(\bar{x}_t)$ as:

$$\exists \bar{y}'_t\, R_t(\bar{x}_t, \bar{y}_t) \wedge \forall \bar{z}_t\, \Big(R_t(\bar{x}_t, \bar{z}_t) \rightarrow \chi(\bar{x}_t, \bar{z}_t) \wedge \bigwedge_{i \in [1,k]} \psi'_{t_i}(\bar{z}_{t_i})\Big),$$

where $\bar{y}'_t$ is the tuple of all variables in $\bar{y}_t$ that do not appear in $\bar{x}_t$; the tuple $\bar{z}_t$ consists exclusively of fresh variables and satisfies $|\bar{z}_t| = |\bar{y}_t|$; for each t_i with $i \in [1, k]$ the formulae ψ'_i are obtained by induction hypothesis; the tuple $\bar{z}_{t_i}$ is the one that is obtained from $\bar{z}_t$ by selecting the positions corresponding to key positions from $\bar{x}_{t_i}$ (this is well-defined as, by definition of the class $\mathcal{C}_{\text{ac-full}}$, each variable in the $\bar{x}_{t_i}$ appears in $\bar{y}_t$); and, finally, $\chi(\bar{x}_t, \bar{z}_t)$ contains each equality of the form $x = z$ such that $x \in \bar{x}_t$, $z \in \bar{z}_t$, and the variable corresponding to z in $\bar{y}_t$ is precisely x. As an example of the construction, suppose that t is associated with atom $R(x, y\,; x, v)$ and has two children t_1 and t_2 which are respectively associated with atoms $S(x\,; u)$ and $T(x, v\,; w)$. Then $\psi'_t(x, y)$ is defined as:

$$\exists v\, R(x, y, x, v) \wedge \forall u \forall v \left(R(x, y, u, v) \rightarrow \big(u = x \wedge \psi'_{t_1}(u) \wedge \psi'_{t_2}(u, v)\big)\right).$$

We start by proving the first part of Claim 33.7. Assume first that $D \models \psi'_t(\bar{a})$. Then, by definition, there is a fact of the form $R_t(\bar{a}, \bar{b}) \in D$, for some tuple $\bar{b}$ over $\mathrm{Dom}(D)$, and each fact in the same block than $R_t(\bar{a}, \bar{b})$ with respect to Σ_t is of the form $R_t(\bar{a}, \bar{b}')$, for some tuple $\bar{b}'$ over $\mathrm{Dom}(D)$ that satisfies $\chi(\bar{a}, \bar{b}')$ and, in addition, it holds that $D \models \psi'_{t_i}(\bar{b}'_i)$, where $\bar{b}'_i$ is the restriction of $\bar{b}'$ to the variables in $\bar{z}_{t_i}$. Take an arbitrary repair D' of D with respect to Σ_t. Then D' must contain some fact of the form $R_t(\bar{a}, \bar{b})$, for some tuple $\bar{b}$ over $\mathrm{Dom}(D)$. By induction hypothesis, for each $i \in [1, k]$ we have that $\bar{b}_i \in q'_{t_i}(D')$, where $\bar{b}_i$ is the restriction of $\bar{b}$ to the variables in $\bar{z}_{t_i}$. That is, there is a homomorphism h_i from q_{t_i} to D' with $h_i(\bar{x}_{t_i}) = \bar{b}_i$. Let h be a mapping defined as $h(\bar{x}_t) = \bar{a}$, $h(\bar{y}_t) = \bar{b}$, and $h(z) = h_i(z)$, for each variable z that appears in q_{t_i} but not in $(\bar{x}_t, \bar{y}_t)$ This mapping is well-defined for the following reason: Every variable that appears in q_{t_i} and q_{t_j}, for $i \neq j$, also appears in $(\bar{x}_t, \bar{y}_t)$. In fact, by definition of $\mathcal{C}_{\text{ac-full}}$ each variable z that appears in q_{t_i} but not in $(\bar{x}_t, \bar{y}_t)$ must have been introduced in some non-key position of an atom A from q_{t_i}. If z was also mentioned in an atom A' of q_{t_j}, then there would be an edge from A to A'; henceforth, T would not be a directed tree, a contradiction. We prove now that $\bar{a} \in q'_t(D')$. We do so by showing that h is a homomorphism from q_t to D'. The result then follows since $h(\bar{x}_t) = \bar{a}$. We only have to prove that h is consistent with the h_is, i.e., $h(z) = h_i(z)$ if z appears in both q_{t_i} and $(\bar{x}_t, \bar{y}_t)$. But this follows from the fact that the only variables appearing in q_{t_i} and $(\bar{x}_t, \bar{y}_t)$ are those in $\bar{x}_{t_i}$ (which we know appear also in $\bar{y}_t$); indeed, in any other case we could prove the existence of an undesired edge destroying the fact that T is a directed tree (left as an exercise). By definition we have that $h(\bar{x}_{t_i}) = \bar{b}_i = h_i(\bar{x}_{t_i})$, and hence the claim holds.

Assume now that for every repair D' of D with respect to Σ_t we have that $\bar{a} \in q'_t(D')$. Henceforth, $D \models \exists \bar{y}'_t R_t(\bar{a}, \bar{y}_t)$. Consider now an arbitrary fact $R_t(\bar{a}, \bar{b}) \in D$. First, $D \models \chi(\bar{a}, \bar{b})$. Otherwise, $\bar{a}$ does not belong to the evaluation of $\exists \bar{y} R_t(\bar{x}_t, \bar{y}_t)$ over D'', where D'' is an arbitrary repair of D that contains $R_t(\bar{a}, \bar{b})$. This contradicts the fact that $\bar{a} \in q'_t(D'')$. We show next that $D \models \psi'_{t_i}(\bar{b}_i)$, for each $i \in [1, k]$, assuming that $\bar{b}_i$ is the restriction of $\bar{b}$ to the variables in $\bar{z}_{t_i}$. By induction hypothesis, it suffices to show that for every repair D_i of D with respect to Σ_t we have that $\bar{b}_i \in q'_{t_i}(D_i)$, i.e., there is a homomorphism h from q_{t_i} to D_i with $h(\bar{x}_{t_i}) = \bar{b}_i$. Take an arbitrary such a repair D_i and let us assume that $R_t(\bar{a}, \bar{b}')$ is the only fact in the block of $R(\bar{a}, \bar{b})$ that belongs to D_i. Let D'_i be the repair defined as $(D_i - \{R_t(\bar{a}, \bar{b}')\}) \cup \{R_t(\bar{a}, \bar{b})\}$. We know that there is a homomorphism h from q_t to D'_i with $h(\bar{x}_t) = \bar{a}$. By definition, $h(\bar{y}_t) = \bar{b}$ and $h(\bar{x}_{t_i}) = \bar{b}_i$. Consider the set $h(q_{t_i})$ of facts in the image of q_{t_i} under h, for $i \in [1, k]$. None of these facts uses the relation symbol R_t because q_t is self-join free, and hence $h(q_{t_i}) \subseteq D_i$. We conclude that h is also a homomorphism from q_{t_i} to D_i with $h(\bar{x}_{t_i}) = \bar{b}_i$, and hence $\bar{b}_i \in q'_{t_i}(D_i)$. Summing up, we have that $D \models \psi'_t(\bar{a})$.

We finish by proving the following claim, which establishes that ψ_t can be defined as $\exists \bar{x}_t \psi'_t(\bar{x}_t)$ for each node t of T.

Claim 33.8. *It is the case that* $\mathrm{CQA}(\hat{q}_t, D) = \mathsf{true} \Leftrightarrow D \models \exists \bar{x}_t \psi'_t(\bar{x}_t)$*, for every database* D.

The fact that $D \models \exists \bar{x}_t \psi'_t(\bar{x}_t)$ implies $\mathrm{CQA}(\hat{q}_t, D) = \mathsf{true}$ easily follows from Claim 33.7, so we omit it here. Assume then that $\mathrm{CQA}(\hat{q}_t, D) = \mathsf{true}$. We show that there must exist a tuple $\bar{a}$ over $\mathrm{Dom}(D)$ such that, for every repair D' of D with respect to Σ_t, it is the case that $\bar{a} \in q'_t(D')$. From Claim

33.7 this implies that $D \models \psi'_t(\bar{a})$, and hence $D \models \exists \bar{x}_t \psi'_t(\bar{x}_t)$. We actually prove the following claim.

Claim 33.9. *Given two repairs D_1 and D_2 of D with respect to Σ_t, there is a repair D^* of D with respect to Σ_t such that $q'_t(D^*) \subseteq q'_t(D_1) \cap q'_t(D_2)$.*

Notice that from Claim 33.9 we obtain our desired result. In fact, by repeated application of this claim we have that there is a repair D^* of D with respect to Σ_t such that $q'_t(D^*) \subseteq q'_t(D')$, for each repair D' of D with respect to Σ_t. Since $\mathrm{CQA}(\hat{q}_t, D) = \mathsf{true}$, there is a homomorphism h from q_t to D^*. Therefore, $h(\bar{x}_t) \in q'_t(D')$ for each repair D' of D with respect to Σ_t.

The proof of Claim 33.9 is by induction on the height of t. When t has height 0, we have that $q'_t = \exists \bar{y}_t R_t(\bar{x}_t, \bar{y}_t)$. Take two repairs D_1 and D_2 of D with respect to Σ_t. We define D^* by choosing one fact for each block over R_t^D. Take an arbitrary such a block B. Both D_1 and D_2 must contain a fact in B, which we call $R_t(\bar{a}, \bar{b})$ and $R_t(\bar{a}, \bar{b}')$, respectively. If $\bar{a} \in q'_t(D_1) \cap q'_t(D_2)$, we add to D^* either $R_t(\bar{a}, \bar{b})$ or $R_t(\bar{a}, \bar{b}')$. Otherwise, if $\bar{a} \notin q'_t(D_1)$, we add $R_t(\bar{a}, \bar{b})$ to D^*; else, we add $R_t(\bar{a}, \bar{b}')$ to D^*. It is clear then that $q'_t(D^*) \subseteq q'_t(D_1) \cap q'_t(D_2)$.

Assume now that t has height $h+1$, for $h \geq 0$. Let $t_1, \dots, t_k$ be the children of t in T. Take two repairs D_1 and D_2 of D with respect to Σ_t, and let D^i, D_1^i and D_2^i be the restrictions of D, D_1 and D_2, respectively, to the relation symbols in q_{t_i}, for each $i \in [1, k]$. By definition, D_1^i and D_2^i are repairs of D^i with respect to Σ_{t_i}. Hence, by induction hypothesis, there is a repair D_*^i of D^i with respect to Σ_{t_i} such that $q'_{t_i}(D_*^i) \subseteq q'_{t_i}(D_1^i) \cap q'_{t_i}(D_2^i)$. We define a repair D^* of D with respect to Σ_t by taking the disjoint union of the D_*^is, for $i \in [1, k]$, plus one fact for each block B over R_t^D. We explain next how to choose such a fact.

Take an arbitrary block B over R_t^D. Both D_1 and D_2 must contain a fact in B, which we call $R_t(\bar{a}, \bar{b})$ and $R_t(\bar{a}, \bar{b}')$, respectively. If $\bar{a} \in q'_t(D_1) \cap q'_t(D_2)$, we add to D^* either $R_t(\bar{a}, \bar{b})$ or $R_t(\bar{a}, \bar{b}')$. Otherwise, if $\bar{a} \notin q'_t(D_1)$, we add $R_t(\bar{a}, \bar{b})$ to D^*; else, we add $R_t(\bar{a}, \bar{b}')$ to D^*. We claim that $q'_t(D^*) \subseteq q'_t(D_1) \cap q'_t(D_2)$. In fact, take a tuple $\bar{a} \in q'_t(D^*)$. Hence, there is a homomorphism h from q_t to D^* with $h(\bar{x}_t) = \bar{a}$. In particular, D^* contains a (unique) fact of the form $R_t(\bar{a}, \bar{b})$, for $\bar{b}$ a tuple over $\mathrm{Dom}(D)$. By definition, h is also a homomorphism from q_{t_i} to D_*^i, for each $i \in [1, k]$. By hypothesis, then, $\bar{b}_i \in q'_{t_i}(D_1^i) \cap q'_{t_i}(D_2^i)$, where $\bar{b}_i = h(\bar{x}_{t_i})$. Now, suppose for the sake of contradiction that $\bar{a} \notin q'_t(D_1)$; the other case, when $\bar{a} \in q'_t(D_1)$ but $\bar{a} \notin q'_t(D_2)$, is treated analogously. By the way in which D^* is constructed, it is the case that D_1 also contains the fact $R_t(\bar{a}, \bar{b})$. In addition, $\bar{b}_i \in q'_{t_i}(D_1^i)$ for each $i \in [1, k]$. It is easy to observe then that $\bar{a} \in q'_t(D_1)$, which is a contradiction. □

An approximation algorithm

Although evaluating consistent answers to CQs is an intractable problem, even in data complexity, analogously to the case of CQ evaluation over probabilistic databases we can show the existence of an FPRAS for computing the ratio between the number of repairs that satisfy the query and the total number of repairs. This can serve as a good measure of how "consistent" an answer is, especially in those cases in which exact evaluation of consistent answers is infeasible.

Formally, given a CQ $q(\bar{x})$, a database D, and a tuple $\bar{a}$ of elements in $\text{Dom}(D)$ of the same arity as $\bar{x}$, we define

$$\mathsf{ConsLevel}(q, \bar{a}, D, \Sigma) \ := \ \frac{|\{D' \mid D' \text{ is a repair of } D \text{ wrt } \Sigma \text{ with } \bar{a} \in q'(D)\}|}{|\{D' \mid D' \text{ is a repair of } D \text{ wrt } \Sigma\}|}.$$

The next result establishes that this value can be approximated with an FPRAS when q and Σ are fixed.

Theorem 33.10. *Let $q(\bar{x})$ and Σ be a fixed CQ and set Σ of primary keys, respectively. There is an FPRAS for the problem of computing the value* $\mathsf{ConsLevel}(q, \bar{a}, D, \Sigma)$ *on a given database D and tuple $\bar{a}$ in Dom(D).*

The construction is very similar to the one shown in Theorem 32.3 for evaluating CQs over probabilistic databases. We leave the proof to the reader.

34

Ontological Query Answering

Comments and Exercises for Part V

Part VI

Query Answering Paradigms

35
Bag Semantics

INITIAL DUMP

We now describe the standard operations of bag relational algebra and provide their semantics [?, ?, ?, ?]. A bag is a collection of elements with associated multiplicities (numbers of occurrences); if an element b occurs n times in a bag B, we write $\#(b, B) = n$. If $\#(b, B) = 0$, it means that b does not occur in B. Sets are just a special case when $\#(b, B) \in \{0, 1\}$.

In a database D, each k-ary relation name R of the schema is associated with a bag R^D of k-tuples; we will omit the superscript D whenever it is clear from the context. We assume that the attributes of a k-ary relation are $1, \ldots, k$, i.e., we adopt the unnamed perspective.

Syntax

The syntax of relational algebra (RA) expressions is defined as follows:

$$
\begin{array}{rclrr}
4e, e' & ::= & R & \text{(base relations)} & (35.1)\\
& \mid & \sigma_{i=j}(e) & \text{(selection)} & (35.2)\\
& \mid & \pi_\alpha(e) & \text{(projection)} & (35.3)\\
& \mid & e \times e' & \text{(Cartesian product)} & (35.4)\\
& \mid & e \uplus e' & \text{(additive union)} & (35.5)\\
& \mid & e \cup e' & \text{(max-union)} & (35.6)\\
& \mid & e \cap e' & \text{(intersection)} & (35.7)\\
& \mid & e - e' & \text{(difference)} & (35.8)\\
& \mid & \varepsilon(e) & \text{(duplicate elimination)} & (35.9)
\end{array}
$$

where i and j in $\sigma_{i=j}(e)$ are positive integers, and α in $\pi_\alpha(e)$ is a possibly empty tuple of positive integers.

The *arity* of RA expressions is defined as follows: for base relations, it is given by the schema; for $\sigma_{i=j}(e)$ and $\varepsilon(e)$, it is the arity of e; for $\pi_\alpha(e)$, it is the arity of α; for $e \times e'$, it is the sum of the arities of e and e'; for $e \star e'$ with $\star \in \{\cup, \uplus, \cap, -\}$, it is the arity of e.

We then say that an RA expression is well-formed w.r.t. a schema if: it mentions only relation names from the schema; i and j in $\sigma_{i=j}(e)$ are less than or equal to the arity of e; the elements of α in $\pi_\alpha(e)$ are less than or equal to the arity of e; the expressions e and e' in $e \star e'$, with $\star \in \{\cup, \uplus, \cap, -\}$, have the same arity.

Semantics

We give the semantics of (well-formed) RA expressions e by inductively defining the quantity $\#(\bar{a}, e, D)$, which is the number of occurrences of a tuple $\bar{a}$ (of appropriate arity) in the result of applying e to a database D. This is done as follows:

$$
\begin{aligned}
4\#(\bar{a},\, R,\, D) &= \#\big(\bar{a}, R^D\big) && (35.10)\\
\#(\bar{a},\, \sigma_{i=j}(e),\, D) &= \begin{cases} \#(\bar{a}, e, D) & \text{if } \bar{a}.i = \bar{a}.j \\ 0 & \text{otherwise} \end{cases} && (35.11)\\
\#(\bar{a},\, \pi_\alpha(e),\, D) &= \sum_{\bar{a}' \,:\, \pi_\alpha(\bar{a}') = \bar{a}} \#(\bar{a}', e, D) && (35.12)\\
\#\big(\bar{a}\bar{a}',\, e \times e',\, D\big) &= \#\big(\bar{a}, e, D\big) \cdot \#\big(\bar{a}', e', D\big) && (35.13)\\
\#(\bar{a},\, e \uplus e',\, D) &= \#(\bar{a}, e, D) + \#(\bar{a}, e', D) && (35.14)\\
\#(\bar{a},\, e \cup e',\, D) &= \max\big\{\, \#(\bar{a}, e, D),\, \#(\bar{a}, e', D) \,\big\} && (35.15)\\
\#(\bar{a},\, e \cap e',\, D) &= \min\big\{\, \#(\bar{a}, e, D),\, \#(\bar{a}, e', D) \,\big\} && (35.16)\\
\#(\bar{a},\, e - e',\, D) &= \max\big\{\, \#(\bar{a}, e, D) - \#(\bar{a}, e', D),\, 0 \,\big\} && (35.17)\\
\#(\bar{a},\, \varepsilon(e),\, D) &= \min\big\{\, \#(\bar{a}, e, D),\, 1 \,\big\} && (35.18)
\end{aligned}
$$

where $\bar{a}.i$ denotes the i-th element of $\bar{a}$, $\pi_{i_1,\ldots,i_n}(\bar{a})$ is the tuple $(\bar{a}.i_1, \ldots, \bar{a}.i_n)$, and the tuples $\bar{a}$ and $\bar{a}'$ in the rule for $e \times e'$ have the same arity as e and e', respectively.

Then, for an expression e and a database D, we define $e(D)$ as the bag of tuples $\bar{a}$ of the same arity as e so that $\#\big(\bar{a}, e(D)\big) = \#(\bar{a}, e, D)$.

36

Incremental Maintenance of Queries

Incremental maintenance (relational)
Leonid

37

Provenance Computation

semirings, propagation
Pablo

38

Top-k Algorithms

Fagin, threshold
Marcelo

39

Distributed Evaluation with One Round

Marcelo / Pablo

Leonid:
In addition to the abstract model of Frank et al don't forget the concrete hypercube

40

Enumeration and Constant Delay

Enumeration / constant delay / total polynomial time

Marcelo

Suggestion of Wim of how to present it: Present in the beginning which kind of operations you assume to go in constant time. Outputting a log(n) size number. Reading a log(n) size number. Adding them. Etc. So, perhaps try to avoid formally doing RAMs.

Comments and Exercises for Part VI

- Notably, assuming PTIME $\neq$ NP this problem cannot be solved in polynomial time:

  Proposition 40.1. *Fix $k \geq 1$. Given a CQ q, a database D, and a tuple $\bar{t}$ in D, checking if $\bar{t} \in q'(D)$ for some $\mathsf{GHW}(k)$-approximation q' of q is* NP*-complete.*

- Complexity of identifying $\mathsf{GHW}(1)$-approximations.
- Proof of Proposition 40.2.

  Number of underapproximations

  Theorem 20.3 establishes a single-exponential upper bound on the number of $\mathsf{GHW}(k)$-underapproximations that a CQ can have. As established next, this is optimal even for the case $k = 1$.

  Proposition 40.2. *There is a family $\{q_n\}_{n \geq 1}$ of boolean CQs such that each CQ q_n is of size at most $O(n)$ and has $\Omega(2^n)$ non-equivalent $\mathsf{GHW}(1)$-underapproximations.*

  Recall that Example 20.2 exhibits a boolean CQ q with two non-equivalent $\mathsf{GHW}(1)$-underapproximations. Proposition 40.2 can be proved by extending the ideas presented in such an example. In particular, q_1 corresponds to q while q_{n+1} is obtained by "appending" a copy of q to q_n in a suitable way. Details are left as an exercise to the reader (see Exercise 40).
- Lack of quantitative guarantees for approximations.
- Approximations under constraints
- Non-existence of overapproximations.
- Characterization of overapproximations.
- Decidability of $\mathsf{GHW}(1)$-overapproximations.
- Proof of Theorem **??**.

Part VII

Mappings and Views

41

Query Answering using Views

Andreas
LMSS, AD

42

Determinacy and Rewriting

Andreas

43

Mappings and Data Exchange

Andreas
chase

44

Query Answering for Data Exchange

45

Ontology-Based Data Access

Andreas

tgds, guardedness, bounded treewidth of the chase, rewriting for linear tgds?

Comments and Exercises for Part VII

Part VIII

Tree-Structured Data

Tree-structured data came to the attention of the data management community with the introduction of the Extensible Markup Language (XML) in 1998, which has since then become one of the most widespread formats for exchanging data on the Web. The other popular data interchange format, the JavaScript Data Interchange Standard (JSON), was defined in 1999 and is similar to XML in the sense that it also treats data in a tree structured manner. Tree-structured data is therefore an important part of data management and we devote this part to discuss some of its fundamental aspects.

Part VIII is organized as follows. In Chapter 46, we discuss the data models that we will use throughout this part. We then define *tree pattern queries* in Chapter 47. Tree pattern queries are a fragment of the language XPath, and can navigate through trees using the *child* and *descendant* relations. As such, they are among the most fundamental languages one can devise for querying trees. They arguably play a similar role for tree-structured data as conjunctive queries do for relational data. In Chapter 48 we treat the basic static analysis and optimization problems for tree patterns, namely *containment* and *minimization*.

Chapters 47 and 48 are presented in terms of unordered trees, that is, trees in which siblings are unordered. However, since tree pattern queries are agnostic of the sibling ordering in the data, all the results in Chapters 47 and 48 remain to hold in the setting where sibling ordering is present in the data.

We then move to more expressive queries in Chapter 49, where we introduce the navigational core of XPath and study its relationship with *first-order logic (FO)* on trees. In Chapter 50, we add even more expressiveness and treat *monadic second-order logic (MSO)* on trees. Boolean MSO formulas can define precisely the *regular tree languages* which, similarly to word languages, are also characterized by finite automata. We will see a third characterization, namely through *monadic Datalog.*

The idea of storing data as trees has led to the development of schema languages for trees, based on ideas from extended context-free grammars and regular tree languages. In Chapter 51 we define the structural core of the three most widespread schema languages for XML, namely DTD, XML Schema, and Relax NG and give some insights in their expressiveness.

We study queries and schema together in Chapter 52, where we look into query optimization *under schema information.* Queries can indeed be optimized more aggressively when schema information is taken into account, but the computational complexity for problems such as satisfiability and containment with respect to schemas also increases.

Chapter 53.

Wim:
TODO.

Things to Check and Keep in Mind

Wim:
This is just a list of notation- and other things that should be in sync with the rest of the book. Will not be a real chapter. Just for us.

Notation / Terminology

- Our signature σ_{tree} has infinitely many relations (due to Λ). However, a finite tree can of course only use finitely many of them. A formula too. We need to check if this is fine wrt the previous definitions. I added a footnote where I'm first using an infinite vocabulary. We should agree if what I say there is OK.
- Number of nodes of a tree T: $|T|$ or $||T||$?
  - There's an earlier discussion in the book about number of tuples $|D|$ of a database vs its size $||D||$. With "size", do we really mean the number of bits to represent the entire database?
  - Cardinality of a set is $|S|$?
- Do we already have from before: For a binary relation E, the relation E^{-1}. Transitive closure of a relation.
- I'm using the $\bar{v}$ notation from Chapter 12.
- I'm using $\uplus$ for disjoint union.
- In the MSO chapter, I say that variables now range over "sets of nodes" in trees. I could add "sets of elements in the universe" in the logic, for clarity. But what's our term for "universe" of a logic here? (It's called universe in Leonid's FMT book.)
- It is assumed that logics always have the equality relation. (Relevant for MSO chapter.)

- I'm denoting free variables as $\mathsf{FV}(\varphi)$ as in the logic chapter around mid-june. There was no macro.
- Size of a tree automaton is now denoted $\|A\|$. Not sure yet if we decide on $\|A\|$, $|A|$, or $||A||$. But there is a macro "size" for it, so we can change. (Same for "size of a regular expression" in RPQ evaluation chapter.)
- The MSO chapter uses "size of an NFA" in the definition of size of a tree automaton.
- We need the definition of S^*, where S is a set in the appendix on regular languages.
- I'm using the notation $Q_N = \cup_{q \in Q} \cup_{a \in \Sigma} Q_{q,a}$ (MSO chapter) because Marcelo liked it more than $Q_N = \cup_{(q,a) \in Q \times \Sigma} Q_{q,a}$.
- Notation $L(r)$ for regular expression r (schema language chapter). We also need *size of a regular expression*, currently denoted $\|r\|$.
- Do we globally use $\{\}$ for false and $\{\langle\rangle\}$ for true?
- RPQs actually use regular languages over an infinite alphabet (since we use an infinite set of labels for edges in graph dbs). We need to discuss this in the Appdx and add a reference from the body.

Other Things to Check

- We assume that the terminology *transitive closure* is known.
- We'll need to add a complexity assumption somewhere. Something like: equality tests between elements in Λ can be done in constant time. But maybe we can make these assumptions explicit in the relevant chapters.
- Do we only use logics over ordered trees in Chapters 49–52?
- Check where we first use data trees. (We give a ref in the Data Model chapter.)
- In the XPath chapter, I use Marx's Core XPath, called NavXPath in XPath Leashed. Real XPath doesn't have nextsibling, nested union, but adds absolute paths. (XPath leashed, sec 2.2) We should decide if that's what we want. (We could also call it NavXPath to be consistent with XPath Leashed.) I'm personally fine with calling it Core XPath.
- Do we have the terminology "tree language"?

To Keep in Mind

- In the vocabulary

$$\sigma_{\mathsf{tree}} = ((\mathrm{Lab}_a)_{a \in \Lambda}, R^{\mathrm{fc}}, R^{\mathrm{ns}}, R^{\mathrm{fs}}, R^{\mathrm{d}}, \mathrm{root}, \mathrm{leaf}, R^{\mathrm{ls}}) ,$$

we need the R^{ls} relation to get the equivalence between MSO and monadic datalog on unranked trees. (Note to self: do we do unranked?)

- I've put NTA emptiness in the MSO chapter, but it could go elsewhere too. It can also be used to show that schema satisfiability (EDTD non-emptiness) is in Ptime and that pattern containment wrt schema is in ExpTime.
- I do the proof of MSO to tree automata with an intermediate step through MSO_0. This idea is the most teachable I could find and comes from Wolfgang Thomas. Languages, Automata, and Logic. Bericht 9607 frim Institut für Informatik und Praktische Mathematik der Christian-Albrechts-Universität zu Kiel.
- I mention "regular tree languages" in the schema chapter. This term should be introduced in the MSO chapter somewhere.

46

Data Model

In this chapter we introduce several abstractions of tree-structured data. We focus on abstractions that are simple and elegant, yet powerful enough to prove results that help us understand the nature of tree-structured data and its query languages in practice. Our data models will be variations of trees, which have *labels* and/or *data values* associated to their nodes. We begin this chapter by defining labeled trees and show later how data values can be incorporated.

Unordered Labeled Trees

Throughout the chapters in Part VIII, let Λ be a countably infinite set of *labels*. By $\mathbb{N}_+^*$ we denote the set of words over the set $\mathbb{N}_+$, i.e., words in which every symbol is a strictly positive natural number. For instance, $1 \cdot 4$ and $12 \cdot 3$ are words over $\mathbb{N}_+$, each consisting of two symbols.[1] In order to define a tree, we first define a *tree domain*, which formalizes the set of *nodes* of the tree. A *tree domain* V is a subset of $\mathbb{N}_+^*$ such that

- for every $u \cdot i \in V$ with $u \in \mathbb{N}_+^*$ and $i \in \mathbb{N}_+$, we also have that $u \in V$; and
- for every $u \cdot i \in V$ we also have that $u \cdot j \in V$ for every $j < i$.

An *(unordered) labeled tree* is a tuple $T = (V, R^{\mathrm{c}}, \mathrm{lab})$ where

- V is a tree domain,
- $R^{\mathrm{c}} = \{(u, u \cdot i) \mid i \in \mathbb{N}_+ \text{ and } u \cdot i \in V\}$, and
- $\mathrm{lab} : V \to \Lambda$ assigns to every node its label from Λ.

[1] We denote the concatenation operator explicitly by $\cdot$ whenever omitting it can lead to confusion.

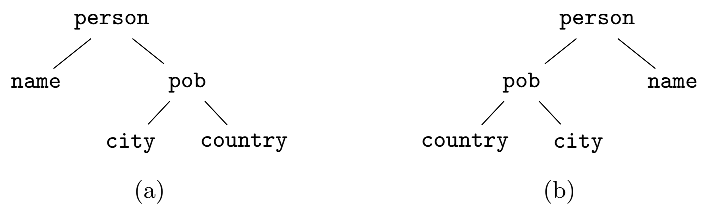


Fig. 46.1: Two trees, capturing the nested structure of names in the JSON document of Figure 46.2.

```
{"person": {
  "name": "Jimi",
  "pob": {"city": "Seattle", "country": "US"}
}}
```

Fig. 46.2: A JSON document consisting entirely of unordered content.

The empty word ε is also called the *root* of T and we sometimes denote it by $\mathrm{root}(T)$. The relation R^{c} is called the *child relation* and a node $u \cdot i$ with $i \in \mathbb{N}_+$ is called a *child* of u. Conversely, u is a *parent* of $u \cdot i$. Likewise, each node $u \cdot v$ with $v \neq \varepsilon$ is a *descendant* of u and, conversely, u is an *ancestor* of $u \cdot v$. A *leaf* is a node without child. Two nodes are *siblings* if they have the same parent. A *(directed) path* in T is a sequence $\pi = v_0 v_1 \cdots v_n$ of nodes such that v_i is a child of v_{i-1} for each $i \in [1, n]$.

We sometimes allow more liberty in the definition of unordered labeled trees in the sense that we do not require V to be a tree domain. In fact, it is convenient to also consider a tuple $(V, R^{\mathrm{c}}, \mathrm{lab})$ to be a labeled tree if V is an abstract set of nodes and $(V, R^{\mathrm{c}}, \mathrm{lab})$ is isomorphic to some labeled tree. Such a definition is convenient for working with *subtrees.* The *subtree* of T at node u, denoted $T_{|u}$ is then defined as $(V_{|u}, R^{\mathrm{c}}_{|u}, \mathrm{lab}_{|u})$, where $V_{|u}$ consists of u and all its descendants in T, the relation $R^{\mathrm{c}}_{|u}$ is $R^{\mathrm{c}} \cap (V_{|u} \times V_{|u})$, and $\mathrm{lab}_{|u}$ is the restriction of lab to $V_{|u}$.

We depict trees in the usual way as in Figure 46.1(a), with the root node on top and the leafs at the bottom. The trees we defined until now are *unordered*, which means that we consider the children of a node as an unordered set. Therefore, if we view the trees in Figures 46.1(a) and 46.1(b) as unordered trees, they are formally the same tree. We will consider unordered trees in Chapters 47 and 48.

The JSON document in Figure 46.2 describes tree-structured data with unordered content. It consists of an unordered, nested collection of so-called *name/value* (or *key/value*) pairs, which are formatted as

```
{ "<name>" : "<value>" }
```

where `<name>` is a string and `<value>` is either a string or again a nested collection of unordered name/value pairs. The unordered tree in Figure 46.1(a) therefore captures the nested structure of the *names* in the JSON document in Figure 46.2. We will see how the *values* can be added later in this chapter. **(Wim)**[2]

Ordered Labeled Trees

This Part will also consider *ordered trees*, which are tuples of the form

$$T = (V, R^{\mathrm{c}}, R^{\mathrm{ns}}, \mathrm{lab}),$$

such that V is a tree domain, R^{c} and lab are the same as for unordered labeled trees, and R^{ns} is the *next sibling* relation, that is, $R^{\mathrm{ns}} = \{(ui, u(i+1)) \mid i, i+1 \in \mathbb{N}_+ \text{ and } ui, u(i+1) \in V\}$. We call $u1$ the *first child* of u. We will sometimes denote ordered trees T as $a(T_1, \ldots, T_n)$, meaning that T consists of a root with label a and ordered sequence of children $u_1, \ldots, u_n$, where T_i is the subtree rooted at u_i for every $i \in [1, n]$. A set of ordered trees is also called a *tree language*. **(Wim)**[3]

The JSON document and the XML document in Figure 46.3 describe ordered, tree-structured data. From a theoretical perspective, they describe the same ordered tree. Figure 46.4 depicts the ordered tree that captures the nested structure of the *XML tags* in Figure 46.3(a). This tree also captures the nested structure of the *names* in the JSON document in Figure 46.3(b). We do not claim that Figures 46.3(a) and 46.3(b) are the most natural ways to model their data. For instance, `pers-id` could also have been an *XML attribute* and in JSON one may choose to model the data by omitting `"persons"` altogether.[4] We chose the examples to make the connection to trees as clear as possible.

In Chapters 49–52, we will study logics over ordered trees. Such trees can be naturally viewed as relational structures over the vocabulary

$$\sigma_{\mathsf{tree}} = ((\mathrm{Lab}_a)_{a \in \Lambda}, R^{\mathrm{fc}}, R^{\mathrm{ns}}, R^{\mathrm{fs}}, R^{\mathrm{d}}, \mathrm{root}, \mathrm{leaf}, R^{\mathrm{ls}}) \ .$$

Here, Lab_a is a unary relation containing precisely the nodes labeled a, for each $a \in \Lambda$.[5] Furthermore, R^{fc} is the first child relation, R^{ns} is the next

[2] **Wim:** In JSON without arrays, trees are deterministic in the sense that child nodes have a unique label. Mention this in bibliographic and further remarks? We could also check if this restriction has any impact on the complexity results in the next chapter.

[3] **Wim:** Not sure if we need the definitions with V isomorphic to a tree domain in the case of ordered trees.

[4] It is even more natural to model JSON documents as *edge-labeled trees*. However, most (if not all) results we present equally hold for edge-labeled trees.

[5] Notice that, in contrast to earlier in the book, this is a vocabulary with an infinite number of relations. Since an individual formula or tree only use a finite subset of these relations, we will not encounter situations where we have to treat such a vocabulary significantly differently from a finite one.

```
<?xml version="1.0"
      encoding="UTF-8"?>
<persons>
 <person>
  <pers_id> 1 </pers_id>
  <name> Jimi </name>
  <birthplace> Seattle </birthplace>
 </person>
 <person>
  <pers_id> 2 </pers_id>
  <name> Saul </name>
  <birthplace> London </birthplace>
 </person>
 <person>
  <pers_id> 3 </pers_id>
  <name> Mark </name>
  <birthplace> Glasgow </birthplace>
 </person>
</persons>
```

(a)

```
{
 "persons": [
   { "person" : [
    { "pers_id":"1" },
    { "name":"Jimi" },
    { "birthplace":"Seattle" }]},
   { "person" : [
    { "pers_id":"2" },
    { "name":"Saul" },
    { "birthplace":"London" }]},
   { "person" : [
    { "pers_id":"3" },
    { "name":"Mark" },
    { "birthplace":"Glasgow" }]}
 ]
}
```

(b)

Fig. 46.3: An XML document (a) and a JSON document (b) describing ordered content.

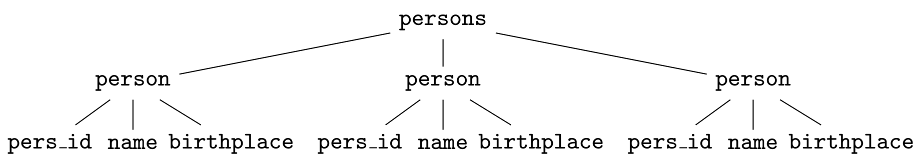


Fig. 46.4: A labeled ordered tree, capturing the "structural" part of the documents in Figure 46.3.

sibling relation, R^{fs} is the following sibling relation (i.e., the transitive closure of R^{ns}), and the R^{d} is the descendant relation (i.e., the transitive closure of $R^{\text{fc}} \bowtie_{2\doteq 1} R^{\text{fs}}$). Finally, root, leaf, and R^{ls} are unary relations that are satisfied in the root, nodes without children (i.e., the leafs), and nodes without next sibling (i.e., last siblings), respectively. In some cases, we also consider a strict subset of the vocabulary σ_{tree}, for instance, without the the transitive relations.**(Wim)**[6]

Trees with Data

Until now we have focused on the *structure* of tree-structured data. We now also take the *data values* into account. We will model data values in trees

[6] **Wim:** This makes sense for the complexity of XPath evaluation

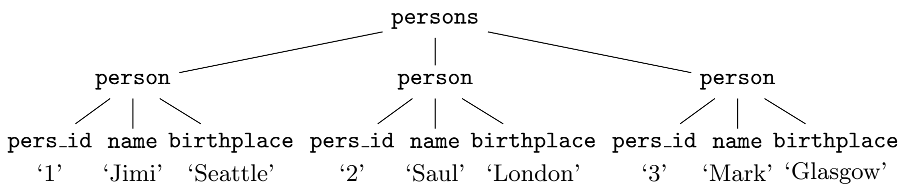


Fig. 46.5: A labeled ordered tree with data values, capturing the structure and data in the documents in Figure 46.3.

using the set Const. An *ordered data tree* or *data tree* over Λ and Const is a tuple

$$T = (V, R^{\mathrm{c}}, R^{\mathrm{ns}}, \mathrm{lab}, \mathrm{data}),$$

where $(V, R^{\mathrm{c}}, R^{\mathrm{ns}}, \mathrm{lab})$ is an ordered tree as before and data: $V \to \mathsf{Const}$ is a function associating a value to each node.

The tree in Figure 46.5 captures the structure and the data in Figure 46.3 as an ordered data tree. We depicted the values as annotations to the nodes ('Jimi', 'Seattle', ...). Some nodes do not have an associated value in Figure 46.5. Formally, these nodes can have a dummy value from Const assigned, which we omit in images to avoid overloading. **(Wim)**[7]

Trees Versus Data Trees

At first sight it may seem as if we can only learn little from trees, because the actual data is only modeled in data trees. However, this is not true. The crux is that *it depends on how trees and queries interact*. Indeed, if queries are only able to compare values with constants (e.g., "find all persons born in Seattle", "retrieve the person with `pers_id = 3`") there is no significant difference between trees and data trees from a theoretical perspective, i.e., data values and labels behave in the same way.

The difference arises when we allow to express queries that compare values in the data with each other, e.g., "find pairs of persons born in the same city" or "test whether all data values are unique". In Chapter 53, we will only allow such comparisons within Const, but not within Λ, and we will see that this can have a drastic effect on the complexity of reasoning about queries.

[7] **Wim:** Not sure if this dummy value is a good idea. I'd like to not overload images though.

47

Tree Pattern Queries

In this chapter, we introduce *tree pattern queries*, which are a fundamental and simple language for querying tree-structured data. Furthermore, they are a natural part of XPath, which is a widespread language for navigation and node selection in tree-structured data. One can argue that tree pattern queries play a similar fundamental role for tree-structured data as conjunctive queries for relational data. Due to their tree structure, however, they are always acyclic.

Definition and Semantics

A *tree pattern query (TPQ)* is a tuple

$$p = (V, R^{\mathrm{c}}, R^{\mathrm{d}}, \mathrm{lab}, \bar{v})$$

where

- $(V, R^{\mathrm{c}} \uplus R^{\mathrm{d}}, \mathrm{lab})$ is an unordered tree over $\Lambda \uplus \{*\}$ and
- $\bar{v} = (v_1, \ldots, v_k)$ is a k-tuple of *output nodes*

We refer to R^{c} and R^{d} as the *child edges* and *descendant edges* of p, respectively. If a node is labeled "$*$", we call it a *wildcard node*. When we represent TPQs graphically, we draw child edges using single lines and descendant edges using double lines, see Figure 47.1. By $|p|$ we denote $|V| + k$, i.e., the number of nodes of p plus the arity k of the tuple of output nodes. Notice that we do not require the k output nodes to be distinct. We call p a *k-ary* TPQ. If $k = 0$, we call the query *Boolean*.

Intuitively, a TPQ can be matched in a tree if there exists a homomorphism from the TPQ to the tree that satisfies all constraints imposed by the query. More precisely, let $p = (V, R^{\mathrm{c}}, R^{\mathrm{d}}, \mathrm{lab}, \bar{v})$ be a TPQ and $T = (V_T, R^{\mathrm{c}}_T, \mathrm{lab}_T)$ be a tree. We say that a label $a \in \Lambda$ *matches* a node $v \in V$ if $\mathrm{lab}(v) = a$ or $\mathrm{lab}(v) = *$. A function $m\colon V \to V_T$ is a *match* of p in T if it fulfills all the following conditions:

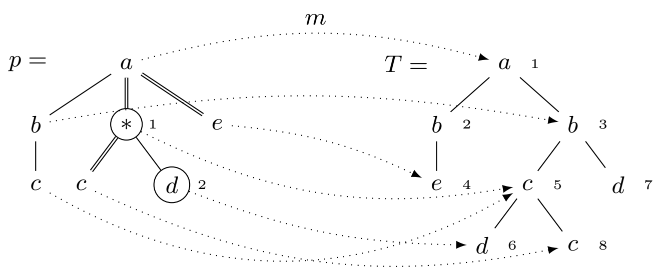


Fig. 47.1: A tree pattern query p (left), a tree T (right), and a match of p in T.

- $m(\text{root}(p)) = \text{root}(T)$;
- for every node $v \in V$, we have that $\text{lab}_T(m(v))$ matches v;
- if $(u, v) \in R^c$, i.e., it is a child edge, then $(m(u), m(v))$ is an edge in T; and
- if $(u, v) \in R^d$, i.e., it is a descendant edge, then there is a path $\pi = u_0 \cdots u_n$ in T such that $m(u) = u_0$, and $m(v) = u_n$, and $n > 0$.

Notice that we do not require a match to be injective. The *output* of p on T, denoted $p(T)$, is the set of k-tuples $(m(v_1), \ldots, m(v_k))$, where m is a match of p on T and $(v_1, \ldots, v_k)$ are the output nodes of p. We say that p *can be matched in* T if there exists a match from p on T. We denote this by $T \models p$ (pronounced "T models p"). Notice that TPQs can also be defined on ordered trees $T = (V, R^c, R^{ns}, \text{lab})$. In this case, the semantics simply ignores the sibling ordering R^{ns}.

Example 47.1. Figure 47.1 depicts a TPQ p, a tree T, and a match m of p in T. The output nodes of p are circled and annotated with 1 and 2, respectively. The match m produces the answer $(5, 6)$ in T. Another match can be obtained from m by mapping output nodes 1 and 2 to nodes 3 and 7, respectively. Furthermore, $p(T) = \{(3, 7), (5, 6)\}$. We note that there exist different matches that produce $(3, 7)$ (by mapping the c-labeled sibling of node 2 in p to node 5 or to node 8, respectively).

The semantics of TPQs can easily be extended to data trees by additionally allowing labels from Dom in leaf nodes. As such, one can define a child edge (u, v) in p with $\text{lab}_p(u) \in \Lambda$ and $\text{lab}_p(v) \in \text{Dom}$ to mean that $m(u)$ should have data value $\text{lab}_p(v)$ in a match m. By doing so, TPQs have the power to compare data values to constant values provided in the pattern.

Evaluation

We are interested in the following decision problem.

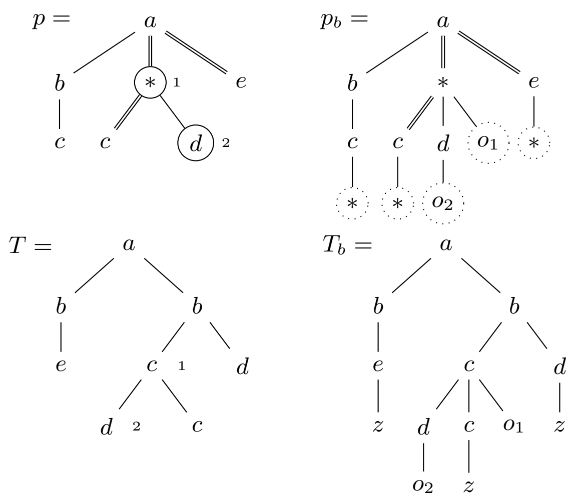


Fig. 47.2: Illustration of the reduction from TPQ-Evaluation to TPQ-Evaluation for Boolean queries

PROBLEM:	TPQ-Evaluation
INPUT:	a TPQ p, a tree $T = (V, R^{c}, \text{lab})$, a tuple $\bar{u} \in V^k$
OUTPUT:	yes if $\bar{u} \in p(T)$ and no otherwise

We will show that this problem can be solved in polynomial time. First we reduce it to a simpler, Boolean variant of the problem.

Lemma 47.2. *Let p be a k-ary TPQ, T a tree, and $\bar{u}$ a k-tuple of nodes of T. Then there exists a Boolean TPQ p_b and a tree T_b such that $\bar{u} \in p(T)$ if and only if $T_b \models p_b$. Furthermore, p_b and T_b can be computed in logarithmic space.*

Proof. Let $p = (V_p, R^{c}_{p}, R^{d}_{p}, \text{lab}_p, \bar{v})$ with $\bar{v} = (v_1, \ldots, v_k)$, let $T = (V, R^{c}, \text{lab})$, and let $\bar{u} = (u_1, \ldots, u_k)$. Let z and $o_1, \ldots, o_k$ be elements of Λ not appearing in T or p. The tree T_b is obtained from T by attaching to each node u_i a new child u'_i with label o_i and to each leaf node $u \notin \{u_1, \ldots, u_k\}$ a new child u' with label z. Similarly, pattern p_b is obtained from p by attaching to each node v_i a new child v'_i with label o_i and to each leaf node $v \notin \{v_1, \ldots, v_k\}$ a new child v' with label $*$. It is easy to show that $\bar{u} \in p(T)$ if and only if $T_b \models p_b$. $\square$

We illustrate the construction for p in Figure 47.2. One may wonder why it is necessary to attach new nodes (labeled $*$ and z, respectively) to the leaf nodes of p and T. The reason is that this is necessary for the correctness of the construction, which we show next. Consider the pattern p and tree T

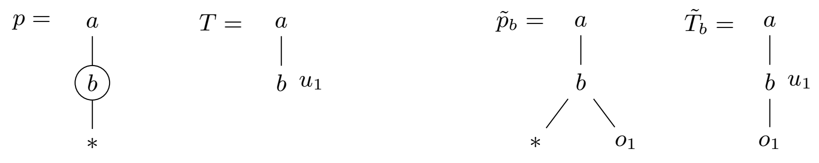


Fig. 47.3: Illustration why the reduction in Lemma 47.2 needs to attach the leaf nodes labeled z and $*$.

in Figure 47.3. Furthermore, $\tilde{p}_b$ and $\tilde{T}_b$ are obtained from p and T by the construction as in Lemma 47.2, but *without attaching the new nodes to the leaf nodes.* Assume that u_1 is the b-labeled node in T. Then $u_1 \notin p(T)$ because p requires that its output node is not a child. However, $\tilde{T}_b \models \tilde{p}_b$, which means that attaching new leaf nodes is indeed necessary.

Theorem 47.3. TPQ-Evaluation *is in* PTIME.

Proof. By Lemma 47.2, it suffices to show how to test $T \models p$ in PTIME for a Boolean pattern p. So, assume that $T = (V, R^{\text{c}}, \text{lab})$ and $p = (V_p, R_p^{\text{c}}, R_p^{\text{d}}, \text{lab}_p)$. The idea is that we compute two sets of nodes of p for each node u of T, namely

- $\mathsf{match}(u) = \{v \in V_p \mid T_{|u} \models p_{|v}\}$ and
- $\mathsf{match}'(u) = \{v \in V_p \mid \exists$ a descendant u' of u such that $T_{|u'} \models p_{|v}\}$

We compute these sets in a bottom-up fashion. First, for each leaf u of T, we define $\mathsf{match}'(u) = \emptyset$ and $\mathsf{match}(u) = \{v \in V_p \mid v$ is a leaf in p and $\text{lab}(u)$ matches $v\}$.

Now, assume that u is a node in T with children $\{u_1, \ldots, u_n\}$ such that we know all sets $\mathsf{match}(u_i)$ and $\mathsf{match}'(u_i)$. Then, $\mathsf{match}(u)$ is the set of nodes $v \in V_p$ that are matched by $\text{lab}(u)$ and such that, for every edge (v, v') of p, there exists a child u_i of u for which one of the following holds:

- If (v, v') is a child edge, then $v' \in \mathsf{match}(u_i)$.
- If (v, v') is a descendant edge, $v' \in \mathsf{match}(u_i) \cup \mathsf{match}'(u_i)$.

The set $\mathsf{match}'(u)$ is simply the union of all sets $\mathsf{match}(u_i)$ and $\mathsf{match}'(u_i)$. Finally, the algorithm accepts if $\text{root}(p) \in \mathsf{match}(\text{root}(T))$. It is easy to see that the algorithm runs in polynomial time. $\square$

The complexity in Theorem 47.3 can be improved to $O(\|p\| \cdot \|T\|)$ by a direct algorithm that does not use the reduction to Boolean patterns, see Exercise 53.1.

48

Tree Pattern Query Containment and Minimization

In this chapter we will study the main static analysis and optimization problems for tree patterns. Since tree patterns are always satisfiable, we treat equivalence, containment, and minimization.

We say that tree pattern queries p_1 and p_2 are *equivalent* (denoted $p_1 \equiv p_2$), if $p_1(T) = p_2(T)$ for every tree T. Tree pattern query p_1 is *contained* in p_2 (denoted $p_1 \subseteq p_2$), if $p_1(T) \subseteq p_2(T)$ for every tree T. We consider Containment and Equivalence problems, whose definition we recall here.

PROBLEM:	TPQ-Containment
INPUT:	TPQs p_1 and p_2
OUTPUT:	yes if $p_1 \subseteq p_2$ and no otherwise

PROBLEM:	TPQ-Equivalence
INPUT:	TPQs p_1 and p_2
OUTPUT:	yes if $p_1 \equiv p_2$ and no otherwise

We first prove that it suffices to focus on Boolean TPQs to study the complexity of containment and equivalence.

Proposition 48.1. *Let p_1 and p_2 be TPQs. Then there exist Boolean TPQs p_1^b and p_2^b such that $p_1 \subseteq p_2$ if and only if $p_1^b \subseteq p_2^b$. Furthermore, p_1^b and p_2^b can be computed in logarithmic space.*

Proof. The construction of p_i^b from p_i is the same as the construction of p_b from p in the proof of Lemma 47.2. □

Next, we prove that it suffices to focus on the contaiment problem, i.e., for Boolean TPQs, containment and equivalence are interreducible.

Remark 48.2. Assume that p_1 and p_2 are Boolean TPQs. On the one hand, $p_1 \equiv p_2$ if and only if $p_1 \subseteq p_2$ and $p_2 \subseteq p_1$. On the other hand, suppose that we want to test if $p_1 \subseteq p_2$. Let p_1' bre the TPQ obtained from p_1 by adding a new root labeled $*$ and connecting it with a child edge to the root of p_1. Likewise, let $p_1 \cap p_2$ be the TPQ obtained from p_1 and p_2 by adding a new root labeled $*$ and attaching two child edges, one to the root of p_1 one to the root of p_2, respectively. Then, we have that $p_1 \subseteq p_2$ if and only if $p_1' \equiv p_1 \cap p_2$. Therefore, and by Proposition 48.1, our focus for the containment and equivalence problems will be on TPQ-Containment for Boolean TPQs.

Containment Via Homomorphisms

Let $p_1 = (V_{p_1}, E_{p_1}, \text{lab}_{p_1})$ and $p_2 = (V_{p_2}, E_{p_2}, \text{lab}_{p_2})$ be TPQs. A mapping $h : V_{p_1} \to V_{p_2}$ is a *homomorphism from* p_1 *to* p_2, if

- $h(\text{root}(p_1)) = \text{root}(p_2)$
- $\text{lab}_{p_2}(h(v))$ matches v for every $v \in V_{p_1}$,
- if $(u, v) \in E_{p_1}$ is a child edge then $(h(u), h(v)) \in E_{p_2}$ is a child edge, and
- if $(u, v) \in E_{p_1}$ is a descendant edge then $h(u)$ is a proper ancestor of $h(v)$ in p_2.

We write $p_1 \to p_2$ if there exists a homomorphism h from p_1 to p_2. Notice that, if $p_1 \to p_2$, then $p_2 \subseteq p_1$. The reason is that, if there exists a match m of p_2 in a tree T, then $m \circ h$ is a match of p_1 in T.

Theorem 48.3. *If* p_1 *and* p_2 *are Boolean TPQs without wildcard nodes, then*

$$p_1 \subseteq p_2 \qquad \iff \qquad p_2 \to p_1$$

Proof. Assume that h is a homomorphism from p_2 to p_1. Then if m is a match of p_1 in a tree T, we have that $m \circ h$ is a match of p_2 in T.

Conversely, assume that $p_1 \subseteq p_2$. Let z be a label from Λ not appearing in p_2. Let T be the tree obtained from p_1 by replacing each descendant edge (u, v) with two child edges (u, u_{uv}) and (u_{uv}, v), where u_{uv} is a new node, labeled z. Since $T \models p_1$ and $p_1 \subseteq p_2$, there exists a match m of p_2 in T. Since p_2 does not have wildcard nodes, the image of m does not contain any new nodes of the form n_{uv} and, therefore, m is a homomorphism from p_2 to p_1. $\square$

We now turn to complexity. The following result is analogous to Theorem 47.3.

Lemma 48.4. *If* p_1 *and* p_2 *are Boolean TPQs, then it can be tested in* PTIME *whether* $p_1 \to p_2$.

Corollary 48.5. *TPQ-*Containment *and TPQ-*Equivalence *are in* PTIME *for Boolean TPQs without wildcard nodes.*

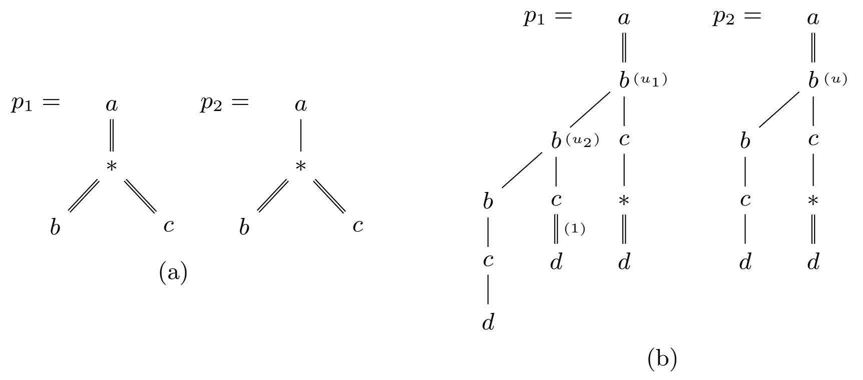


Fig. 48.1: Two examples where $p_1 \subseteq p_2$ even though $p_2 \to p_1$ is false.

If the queries use wildcard nodes, there are obvious examples that show that the existence of a homomorphism is not necessary for containment, see Figure 48.1(a). Here, the left query is contained in the right query (and is even equivalent), even though no homomorphism exists from right to left. Figure 48.1(b) illustrates a more complex example. Here one can see that the left query is contained in the right query by case distinction on the edge marked (1). Assume that $T \models p_1$ with match m. If (1) is mapped to a single edge in T, then node u of p_2 can be mapped to $m(u_1)$. Otherwise, it can be mapped to $m(u_2)$.

Star Extensions and Canonical Trees

Let p be a TPQ. A *star extension* of p is a TPQ without descendant edges, obtained from p by replacing each descendant edge (u, v) by a path $uu_1^{uv} \cdots u_k^{uv}v$ where, for every $i \in [k]$ the node u_i^{uv} is new and labeled $*$.

A tree T is a *canonical tree* of p if it can be obtained from a star extension p^* of p by labeling each wildcard node by some fixed label z. By $T_{\mathsf{z}}[p^*]$ we denote the canonical tree that was obtained in this manner. Notice that canonical trees T of p always match p, since the identity function on the nodes of p is always a match of p in T.

The importance of canonical trees is captured in the following Lemma, that shows that $p_1 \subseteq p_2$ if and only if all canonical trees of p_1 match p_2.

Lemma 48.6. *If p_1 and p_2 are Boolean TPQs. Then $p_1 \subseteq p_2$ if and only if, for every canononical tree T of p_1, we have $T \not\models p_2$.*

Proof. The direction from left to right is trivial. We prove the other direction by contraposition. Assume that $p_1 \not\subseteq p_2$ and let z be a label that does not occur in p_2. Let T be a (not necessarily canonical) tree such that $T \models p_1$ and $T \not\models p_2$, and let m_1 be a match of p_1 in T. For each descendant edge (u, v) of

p_1, let k_{uv} be the number of nodes between $m_1(u)$ and $m_1(v)$ in T. Let p_1^* be the star extension of p_1 where each descendant edge (u,v) is replaced by the path $uu_1^{uv} \cdots u_{k_{uv}}^{uv} v$. Let m^* be a match of p_1^* in T.

We claim that $T_{\mathsf{z}}[p_1^*] \not\models p_2$. Towards a contradiction, assume that $T_{\mathsf{z}}[p_1^*] \models p_2$, witnessed by the match m_2. Notice that m_2 can only map wildcard nodes of p_2 to z-labeled nodes of $T_{\mathsf{z}}[p_1^*]$, since z does not appear in p_2. But this means that $m = m^* \circ m_2$ is a match[1] of p_2 in T, which contradicts that $T \not\models p_2$. □

Worst-Case Complexity

Lemma 48.7. *If p_1 and p_2 are Boolean TPQs such that $p_1 \not\subseteq p_2$, then there exists a tree T with $|T| \leq |p_1| \cdot |p_2| + 1$ such that $T \models p_1$ and $T \not\models p_2$.*

Proof. Let z be a label that does not occur in p_1 or p_2. By Lemma 48.6 we know that there exists a canonical tree $T_{\mathsf{long}} = T_{\mathsf{z}}[p_1^*]$ such that $T_{\mathsf{long}} \models p_1$ and $T_{\mathsf{long}} \not\models p_2$.

We will now prove that long paths of z-labeled nodes in T_{long} can be shortened. Assume that T_{long} has a path $\pi = uu_1^{uv} \cdots u_k^{uv} v$ with (u,v) a descendant edge in p_1 and $k > |p_2|$, so $u_1^{uv}, \ldots, u_k^{uv}$ are new nodes in p_1^*. Let T_{short} be obtained from T_{long} by replacing π with $\pi' = uu_1^{uv} \cdots u_{|p_2|}^{uv} v$. Then clearly $T_{\mathsf{short}} \models p_1$, because it is a canonical tree. We will show that $T_{\mathsf{short}} \not\models p_2$. Assume the contrary. Then there is a match $m : p_2 \to T_{\mathsf{short}}$. Since m matches the root of p_2 to the root of T_{short}, there is at least one node v_j in $\{v_1^{uv}, \ldots, v_{|p_2|}^{uv}\}$ that is not in the image of m. But then m also matches p_2 to the tree T', obtained from T_{short} by replacing π' with $\pi'' = uv_1^{uv} \cdots v_{j-1}^{uv} v_{|p_2|+1}^{uv} \cdots v_k^{uv} v_j^{uv} \cdots v_{|p_2|}^{uv}$. However, T' is isomorphic with T_{long}, which would mean that $T_{\mathsf{long}} \models p_2$. This is a contradiction. □

Lemma 48.7 implies that TPQ Containment is in CONP, since we can guess T and check deterministically (using Theorem 47.3) that $T \models p_1$ and $T \not\models p_2$. In fact, the problem is also CONP-hard (which we do not prove). Using Remark 48.2, this leads us to the following theorem.

Theorem 48.8. *TPQ*-Containment *and TPQ*-Equivalence *are* CONP*-complete.*

Minimization

Tree pattern query minimization amounts to transforming a given TPQ into an equivalent one that is as small as possible. We introduce minimizations of TPQs in a similar way as we did for conjunctive queries.

Definition 48.9 (Minimization of TPQs). *Given a TPQ p, a TPQ p' is called a* minimization *of p if it is equivalent to p and has the smallest number of nodes among all the TPQs that are equivalent to p.*

[1] We note that m^* and m_2 can indeed be composed, since $T_{\mathsf{z}}[p_1^*]$ has the same set of nodes as p_1^*.

Algorithm 11 TrimTPQ(p)

```
Input: A tree pattern query $p$
Output: A subpattern $p^*$ of $p$ that is equivalent to $p$
1: $p^* := p$
2: while there is a leaf $u$ of $p^*$ such that $(p^* - u) \equiv p^*$ do
3:     $p^* := (p^* - u)$
   return $p^*$
```

In Chapter 15 we saw that, in the case of CQs, minimizations can be obtained by removing atoms from the query. We can now ask ourselves if something similar is true for TPQs. More precisely, we ask if TPQs can be minimized by *deleting nodes*. To this end, for a TPQ $p = (V_p, E_p, \mathrm{lab}_p, \overline{v})$ and a leaf $u \in V_p$ that does not appear in $\overline{v}$, denote by $(p - u)$ the query obtained from p by removing u from V_p and every edge of the form (u', u) from E_p. Notice that every subpattern of p that contains the root and output nodes of p can be obtained by repeatedly deleting leaves.

In Algorithm 11, we describe a procedure TrimTPQ, which is similar to the **(Wim)**[2] algorithm for CQs. It repeatedly removes leaves from the pattern as long as the pattern stays equivalent. In some important cases, it is not very difficult to show that TrimTPQ indeed computes a minimization. We leave the proof as Exercise 53.4.

Theorem 48.10. *Let p be a TPQ. Then* TrimTPQ(p) *computes a minimization of p if either*

(1) p does not contain descendant edges, or

(2) p does not have wildcard nodes.

Perhaps surprisingly, Theorem 48.10 does not hold for TPQs in general. Figure 48.2 contains on the right a tree pattern query p that has no equivalent proper subpattern, that is, there is no node v with $p \equiv (p - v)$. However, it is not minimal, because the pattern on the left is equivalent and smaller. We prove that p and q are equivalent and leave the proof that p has no equivalent proper subpattern as an exercise (Exercise 53.5).

Consider the tree pattern queries $p_1, \ldots, p_5$, depicted on the right hand sides of Figures 48.3 and 48.4. Observe that p is equivalent to $p_1 \cup p_2 \cup p_3 \cup p_4 \cup p_5$, since the only difference between these patterns and p is that they replace the lowermost descendant edge in p by paths of increasing length, where p_5 has a descendant edge to deal with paths of length at least five. Figures 48.3 and 48.4 then show how homomorphisms from q to $p_1, \ldots, p_5$ can be constructed. This shows that $p \subseteq q$. The inclusion $q \subseteq p$ is easy to see, because there exists a homomorphism from p to q. The homomorphism maps the two nodes in the dashed lines in p to the single such node in q.

[2] **Wim:** macro for ComputeCore

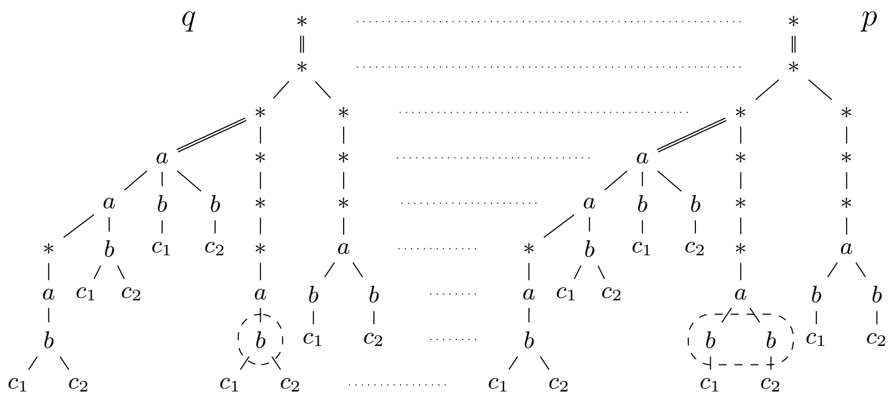
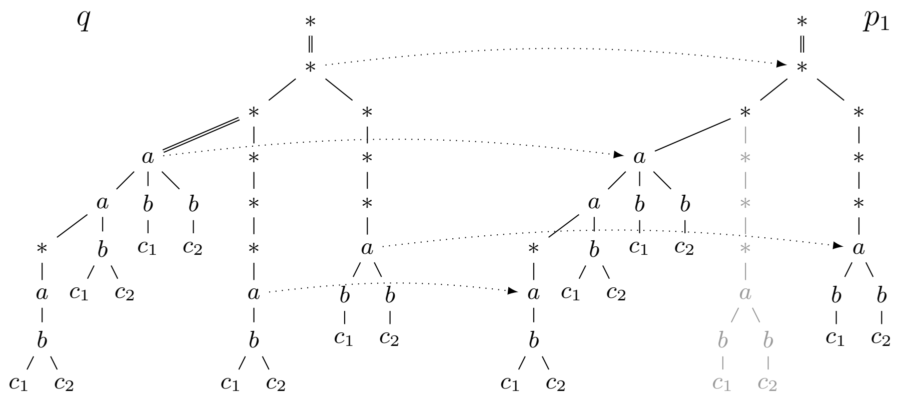
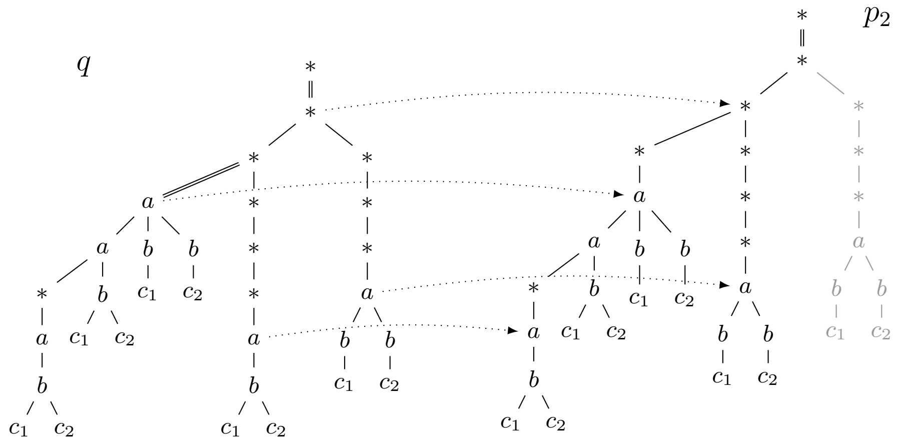


Fig. 48.2: A tree pattern query p that has no equivalent proper subpattern (right) and a tree pattern query q that is equivalent and smaller (left). The only difference between the two patterns is in the dashed lines.

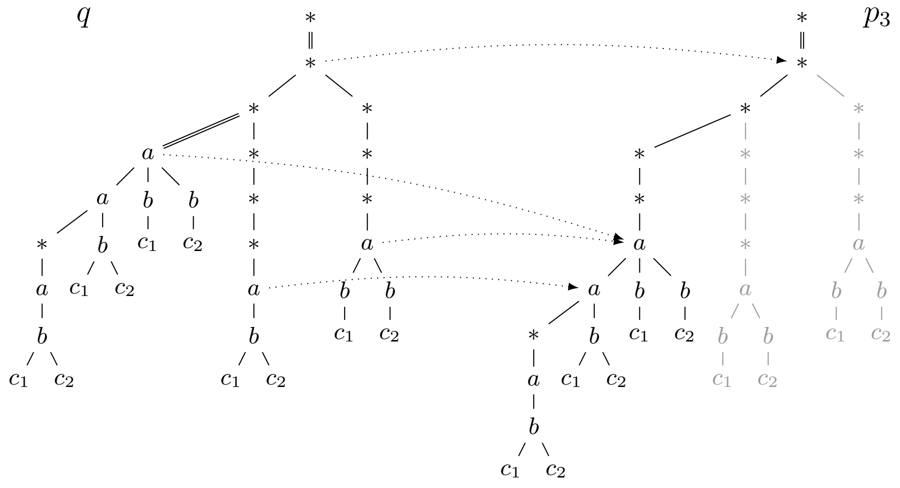
We conclude this chapter with a note on the complexity of computing minimizations of TPQs. Minimizations of a TPQs p can be computed by a naive algorithm that iterates through all TPQs that are smaller than p and tests whether they are equivalent, but such an algorithm hardly seems satisfactory. Surprisingly, we do not know an algorithm that is significantly better! From the example in Figure 48.2, we know that deleting nodes is not sufficient for minimizing TPQs since, in this case, nodes also need to be merged. But there are examples that show that deleting and merging nodes still is not sufficient: in some cases, nodes also need to be split. Part of the challenge is that it is already complex to *test* if a given TPQ is minimal. The complexity of this problem is Π_2^p-complete, which means that the "trival" algorithm that tests if every smaller TPQ is inequivalent is worst-case optimal.



(a) Homomorphism from q to p_1

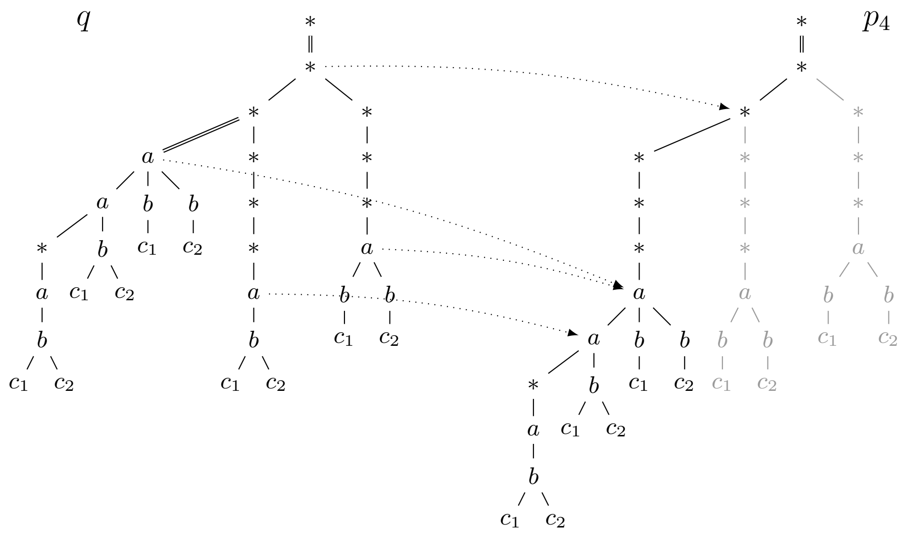


(b) Homomorphism from q to p_2



(c) Homomorphism from q to p_3

Fig. 48.3: Showing that patterns p and q in Figure 48.2 are equivalent


(a) Homomorphism from q to p_4

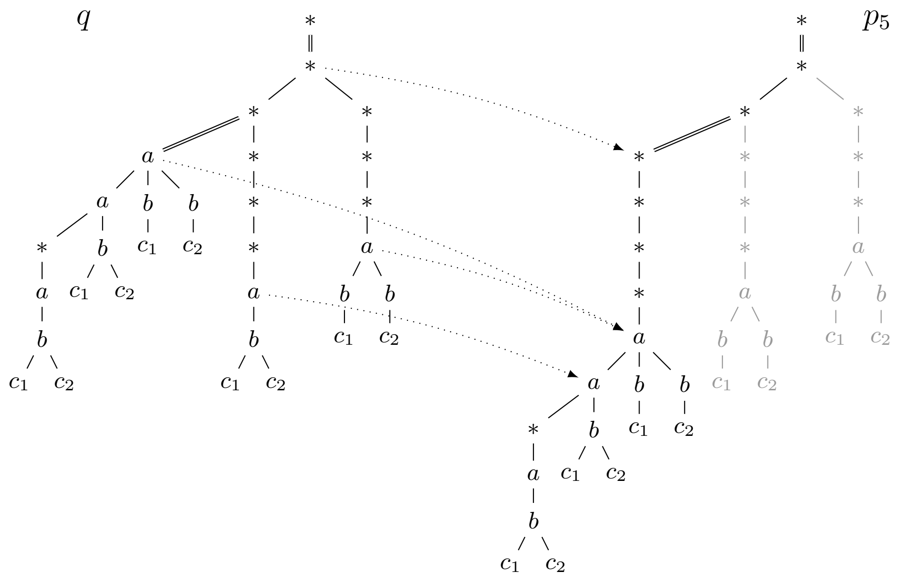

(b) Homomorphism from q to p_5

Fig. 48.4: Showing that patterns p and q in Figure 48.2 are equivalent

49

XPath

XPath is a powerful language, designed for navigation and node-selection in trees. In this chapter, we focus on a few clean languages that are heavily inspired[1] by XPath and that navigate in *trees without data*, that is, structures over the vocabulary $((\mathrm{Lab}_a)_{a \in \Lambda}, R^{\mathrm{fc}}, R^{\mathrm{ns}})$. **(Wim)**[2]

XPath uses so-called *axes* as primitive operations for navigating in trees. We use the axes self, child, parent, descendant, descendant-or-self, ancestor, ancestor-or-self, next-sibling, following-sibling, previous-sibling, preceding-sibling, following, and preceding.[3] We explain the meaning of each axis x by associating it to a binary relation R_x. To this end, consider a structure $T = (V, E_\downarrow, E_\rightarrow)$ over the vocabulary $((\mathrm{Lab}_a)_{a \in \Lambda}, R^{\mathrm{fc}}, R^{\mathrm{ns}})$. Then, $R_{\mathsf{self}} = \{(u,u) \mid u \in V\}$, $R_{\mathsf{child}} = E_\downarrow$, and $R_{\mathsf{next\text{-}sibling}} = E_\rightarrow$. The relations $R_{\mathsf{descendant}}$ and $R_{\mathsf{following\text{-}sibling}}$ are the transitive closures of R_{child} and $R_{\mathsf{next\text{-}sibling}}$, respectively, and $R_{\mathsf{descendant\text{-}or\text{-}self}}$ is the reflexive and transitive closure of R_{child}. By *inverse* of a relation R, we mean the relation $\{(v,u) \mid (u,v) \in R\}$. Then, the relations R_{parent}, $R_{\mathsf{previous\text{-}sibling}}$, R_{ancestor}, $R_{\mathsf{preceding\text{-}sibling}}$, and $R_{\mathsf{ancestor\text{-}or\text{-}self}}$ are the inverses of R_{child}, $R_{\mathsf{next\text{-}sibling}}$, $R_{\mathsf{descendant}}$, $R_{\mathsf{following\text{-}sibling}}$, and $R_{\mathsf{descendant\text{-}or\text{-}self}}$, respectively. Finally, $R_{\mathsf{following}}$ is defined as $R_{\mathsf{ancestor\text{-}or\text{-}self}} \circ R_{\mathsf{following\text{-}sibling}} \circ R_{\mathsf{descendant\text{-}or\text{-}self}}$ and $R_{\mathsf{preceding}}$ is the inverse of $R_{\mathsf{following}}$. **(Wim)**[4] **(Wim)**[5]

Core XPath and Conditional XPath

Core XPath (CoreXPath) *path expressions* p are built up from *steps* and *node expressions* q using the following grammar

[1] We explain the differences with XPath in Section 49.
[2] **Wim:** Or $\sigma_{\mathsf{tree}} = ((\mathrm{Lab}_a)_{a \in \Lambda}, R^{\mathrm{fc}}, R^{\mathrm{ns}}, R^{\mathrm{fs}}, R^{\mathrm{d}}, \mathrm{root}, \mathrm{leaf}, R^{\mathrm{ls}})$?
[3] XPath does not have the axes next-sibling and previous-sibling.
[4] **Wim:** Using $\circ$ for composition here. Could also be $\bowtie$.
[5] **Wim:** Mention how XPath extends tree patterns.

$$\begin{aligned} p &::= \mathsf{step} \mid p/p \mid p \cup p \\ \mathsf{step} &::= \mathsf{axis} \mid \mathsf{step}[q] \\ q &::= ?p \mid \mathsf{lab} = a \mid q \wedge q \mid q \vee q \mid \neg q \end{aligned}$$

Here, axis stands for one of the aforementioned axes and a is a label from Λ.

Conditional XPath (CondXPath) is obtained from CoreXPath by replacing the rule for step with

$$\mathsf{step} ::= \mathsf{axis} \mid \mathsf{step}[q] \mid (\mathsf{step}[q])^*$$

Let T be a tree, i.e., a structure over the vocabulary $((\mathrm{Lab}_a)_{a\in\Lambda}, R^{\mathrm{fc}}, R^{\mathrm{ns}})$. We define the semantics of CoreXPath and CondXPath expressions on T using the binary and unary relations $[\![p]\!]_{\mathsf{p}} \subseteq \mathrm{Dom}(T)^2$ and $[\![q]\!]_{\mathsf{n}} \subseteq \mathrm{Dom}(T)$, respectively. More precisely, we have

$$\begin{aligned} [\![\mathsf{axis}]\!]_{\mathsf{p}} &:= R_{\mathsf{axis}} \\ [\![\mathsf{step}[q]]\!]_{\mathsf{p}} &:= \{(u,v) \in [\![\mathsf{step}]\!]_{\mathsf{p}} \mid v \in [\![q]\!]_{\mathsf{bool}}\} \\ [\![p_1/p_2]\!]_{\mathsf{p}} &:= [\![p_1]\!]_{\mathsf{p}} \circ [\![p_2]\!]_{\mathsf{p}} \\ [\![p_1 \cup p_2]\!]_{\mathsf{p}} &:= [\![p_1]\!]_{\mathsf{p}} \cup [\![p_2]\!]_{\mathsf{p}} \end{aligned}$$

and

$$\begin{aligned} [\![\mathsf{lab} = a]\!]_{\mathsf{n}} &:= \mathrm{Lab}_a \\ [\![?p]\!]_{\mathsf{n}} &:= \{u \mid \exists v \text{ with } (u,v) \in [\![p]\!]_{\mathsf{p}}\} \\ [\![q_1 \wedge q_2]\!]_{\mathsf{n}} &:= [\![q_1]\!]_{\mathsf{n}} \cap [\![q_2]\!]_{\mathsf{n}} \\ [\![q_1 \vee q_2]\!]_{\mathsf{n}} &:= [\![q_1]\!]_{\mathsf{n}} \cup [\![q_2]\!]_{\mathsf{n}} \\ [\![\neg q]\!]_{\mathsf{n}} &:= \mathrm{Dom}(T) - [\![q]\!]_{\mathsf{n}} \end{aligned}$$

Differences with W3C XPath Standards

Wim:
XPath leashed has a paragraph about differences with XPath 1.0, which does not have the axes next-sibling or previous-sibling. It also does not allow nested union. I have no idea yet about other versions of XPath.

Complexity of Evaluation

We show that CoreXPath and CondXPath expressions can be evaluated in linear time combined complexity.

Theorem 49.1. *Let T be a tree and q be a CoreXPath or CondXPath node expression. Then we can compute $[\![q]\!]_{\mathsf{n}}$ in time $O(\|q\|\|T\|)$.*

Proof. Let p be a CoreXPath path expression, q be a CoreXPath node expression and T be a tree. We prove inductively that

- $[\![q]\!]_{\mathsf{n}}$ can be computed in time $O(\|q\|\|T\|)$ and

- given a set $V \subseteq \mathrm{Dom}(T)$, the set $[\![p]\!]_{\mathsf{p}}(V) := \{u \mid \exists v \in V \text{ with } (u,v) \in [\![p]\!]_{\mathsf{p}}\}$ can be computed in time $O(\|q\|\|T\|)$.**(Wim)**[6]

We first consider node expressions q. There is only one base step, i.e., $q = (\mathrm{lab} = a)$, which is obvious. **(Wim)**[7] Moving to the induction, if q is one of $q_1 \wedge q_2$, $q_1 \vee q_2$, then computing $[\![q]\!]_{\mathsf{n}}$ in in time $O(((1 + \|q_1\| + \|q_2\|))\|D\|)$ is immediate from the induction hypothesis, since, for every node in D, we can immediately infer membership in $[\![q]\!]_{\mathsf{n}}$ based on its membership in $[\![q_1]\!]_{\mathsf{n}}$ and $[\![q_2]\!]_{\mathsf{n}}$. The proof where $q = \neg q_1$ is analogous. The final case $q = ?p$ is immediate from the induction hypothesis, taking $V = \mathrm{Dom}(T)$.

We now consider path expressions. Let $V \subseteq \mathrm{Dom}(T)$ and p a path expression. If $p = \mathsf{axis}$, then $[\![p]\!]_{\mathsf{p}}(V)$ can be computed for each possibility of axis in time $O(\|D\|)$. E.g., when $\mathsf{axis} = \mathsf{descendant}$, then $[\![p]\!]_{\mathsf{p}}(V)$ is the set of nodes $\{u \mid \exists v \in V \text{ with } (u,v) \in R_{\mathsf{descendant}}\}$, which are the ancestors of nodes in V. If $p = p_1/p_2$, then we first compute $[\![p_2]\!]_{\mathsf{p}}(V)$ and then $[\![p_1]\!]_{\mathsf{p}}([\![p_2]\!]_{\mathsf{p}}(V))$. The case $p = \mathsf{step}[q]$ is immediate from the induction hypotheses. The final case $p = p_1 \cup p_2$ is immediate as well. This concludes the proof for $\mathsf{CoreXPath}$.

In the case of $\mathsf{CondXPath}$, **(Wim)**[8] the only extra case we need to deal with is $p = (\mathsf{step}[q])^*$. According to the definition of $\mathsf{CondXPath}$, p can either be of the form $(\mathsf{axis}[q])^*$, $(\mathsf{step}'[q'][q])^*$, or $((\mathsf{step}'[q'])^*[q])^*$. In all cases, $[\![q]\!]_{\mathsf{n}}$ can be precomputed in time $O(\|q\|\|D\|)$.

In the first case, if axis is transitive, then p is equivalent to $\mathsf{self} \cup \mathsf{axis}[q]$, which we have already solved. If axis is not transitive, e.g., $\mathsf{axis} = \mathsf{child}$, then we need to compute all ancestors u of nodes in V such that all nodes on the path from u to V are in $[\![q]\!]_{\mathsf{n}}$, which can obviously be done in the required time. The other cases where axis is not transitive are analogous.

In the second case, p is equivalent to $(\mathsf{step}'[q'][q])^*$ which is equivalent to $(\mathsf{step}'[q' \wedge q])^*$ and therefore reduces to the first or third case.

In the third case, p is equivalent to $\mathsf{self} \cup (\mathsf{step}'[q'])^*[q]$, which also reduces to cases that we already dealt with (union and $\mathsf{step}[q]$). This concludes the proof. □

Expressiveness

Here we see trees as structures over $\sigma_{\mathsf{tree}} = ((\mathrm{Lab}_a)_{a \in \Lambda}, R^{\mathrm{fc}}, R^{\mathrm{ns}}, R^{\mathrm{fs}}, R^{\mathrm{d}}, \mathrm{root}, \mathrm{leaf}, R^{\mathrm{ls}})$ or subsignatures thereof.

[6] **Wim:** I do it in this direction, because that's the convenient one for the case $q = ?p$.

[7] **Wim:** Needed: RAM-type computation. We can follow a pointer and test label in constant time.

[8] **Wim:** I made this proof up, so it may be wrong ;-) If we just apply induction we arrive at quadratic time instead of linear. I thought that having linear time combined complexity (with a slightly longer proof) would be better.

Theorem 49.2. *Over the signature $((Lab_a)_{a\in\Lambda}, R^{ns}, R^{fs})$, CoreXPath node expressions are equally expressive as FO^2.* ***(Wim)***[9]

Proof. **(Wim)**[10] Since we only consider navigation in one direction in this proof, we denote the XPath axes with next, previous, following, and preceding. For every CoreXPath node expression p, we define an FO^2-formula $\varphi_p(x)$ as follows:

- When $p = (\text{lab} = a)$, then $\varphi_p(x) = \text{Lab}_a(x)$.
- When p is of the form $\neg p_1$ or $p_1 \wedge p_2$, then $\varphi_p(x) = \neg\varphi_{p_1}(x)$ or $\varphi_p(x) = \varphi_{p_1}(x) \wedge \varphi_{p_2}(x)$, respectively.
- When p is of the form next$[q]$ or previous$[q]$, then $\varphi_p(x) = \exists y(R^{\text{ns}}(x,y)\wedge\varphi_q)$ or $\varphi_p(x) = \exists y(R^{\text{ns}}(y,x) \wedge \varphi_q)$, respectively.
- When p is of the form following$[q]$ or preceding$[q]$, then $\varphi_p(x) = \exists y(R^{\text{fs}}(x,y)\wedge \varphi_q)$ or $\varphi_p(x) = \exists y(R^{\text{fs}}(y,x) \wedge \varphi_q)$, respectively.

Conversely, let $\varphi(x)$ be a unary FO^2 formula. We show a recursive translation procedure to turn $\varphi(x)$ into a CoreXPath formula q_φ. We can assume w.l.o.g. that $\varphi(x)$ only uses the Boolean connectives $\vee$ and $\neg$. When $\varphi(x)$ is atomic, i.e., of the form $\text{Lab}_a(x)$, then $q_\varphi = (\text{lab} = a)$. When $\varphi(x)$ is of the form $\psi_i \vee \psi_2$ or $\neg\psi$, then we recursively compute $q_{\psi_1} \wedge q_{\psi_2}$ or $\neg q_\psi$, respectively. Now, there are only two remaining cases, i.e., $\varphi(x)$ is of the form $\exists x\varphi'(x)$ or $\exists y\varphi'(x,y)$. In the first case, $\varphi(x)$ is equivalent to $\exists y\varphi'(y)$, which is equivalent to $\exists y\varphi'(x,y)$ when we view x as a dummy free variable in $\varphi'(y)$.

So the only remaining case is $\varphi(x) = \exists y\varphi'(x,y)$. In this case, we can write

$$\varphi'(x,y) = \beta\big(\chi_0(x,y),\ldots,\chi_{r-1}(x,y),\xi_0(x),\ldots,\xi_{s-1}(x),\zeta_0(y),\ldots,\zeta_{t-1}(y)\big) ,$$

where

- β is a propositional formula (and we substitute its variables with $\chi_0(x,y),\ldots,\zeta_{t-1}(y)$),
- each χ_i is an atomic formula, and
- each ξ_i and ζ_i is an atomic or existential FO^2 formula.

In order to be able to recurse on subformulas of $\varphi(x)$, we have to separate the ξ_i's from the ζ_i's. We first introduce a case distinction on which of the subformulas ξ_i's hold or not and obtain $\varphi(x) \equiv$

$$\bigvee_{\bar\gamma\in\{\texttt{true},\texttt{false}\}^s} \left(\bigwedge_{i<s} (\xi_i \leftrightarrow \gamma_i) \wedge \exists y \beta(\chi_0,\ldots,\chi_{r-1},\gamma_0,\ldots,\gamma_{s-1},\zeta_0,\ldots,\zeta_{t-1})\right) .$$

[9] **Wim:** Or do it over a domain $[1,n]$ and use child and descendant?
[10] **Wim:** Proof is from Theorem 1 in [7]. Their proof assumes that nodes can have multiple labels.

We proceed with a case distinction on which order relation holds between x and y. These are five mutually exclusive cases, determined by the following formulas, which we call *order types*: $x = y$, $R^{\mathrm{ns}}(x,y)$, $R^{\mathrm{ns}}(y,x)$, $R^{\mathrm{fs}}(x,y) \wedge \neg R^{\mathrm{ns}}(x,y)$, and $R^{\mathrm{fs}}(y,x) \wedge \neg R^{\mathrm{ns}}(y,x)$. When we assume that one of these order types τ is true, each atomic order formula evaluates to either `true` or `false`. In particular, each of the χ_i's evaluates to either `true` or `false`; we will denote this truth value by χ_i^τ. Taking Υ as the set containing the five order types, we can now obtain $\varphi(x) \equiv$

$$\bigvee_{\bar\gamma \in \{\mathtt{true},\mathtt{false}\}^s} \Big(\bigwedge_{i<s} (\xi_i \leftrightarrow \gamma_i) \wedge \bigvee_{\tau \in \Upsilon} \exists y \big(\tau \wedge \beta(\chi_0^\tau, \ldots, \chi_{r-1}^\tau, \bar\gamma, \bar\zeta)\big) \Big) .$$

If τ is an order type, $\psi(x)$ an FO^2 formula, and q_ψ and equivalent CoreXPath formula, there is an obvious way to obtain a CoreXPath formula $q\langle\tau,\psi\rangle$, as shown in the following table.

τ	$x = y$	$R^{\mathrm{ns}}(x,y)$	$R^{\mathrm{ns}}(y,x)$	$R^{\mathrm{fs}}(x,y) \wedge \neg R^{\mathrm{ns}}(x,y)$	$R^{\mathrm{fs}}(y,x) \wedge \neg R^{\mathrm{ns}}(y,x)$
$q\langle\tau,\psi\rangle$	q_ψ	next$[q_\psi]$	previous$[q_\psi]$	next/following$[q_\psi]$	previous/preceding$[q_\psi]$

Our recursive procedure will therefore recursively compute q_{ξ_i} for $i < s$ and $q_{\eta_i(x)}$ for $i < t$ and output

$$\bigvee_{\bar\gamma \in \{\mathtt{true},\mathtt{false}\}^s} \Big(\bigwedge_{i<s} (q_{\xi_i} \leftrightarrow \gamma_i) \wedge \bigvee_{\tau \in \Upsilon} q\langle\tau, \beta\big(\chi_0^\tau, \ldots, \chi_{r-1}^\tau, \bar\gamma, q_{\zeta_0(x)}, \ldots, q_{\zeta_{t-1}(x)}\rangle\big) \Big) .$$

Here, $\zeta_i(x)$ denotes $\zeta_i(y)$ in which we substituted the free occurrences of y with x. This concludes the translation. □

Wim:
The story should now say that a^*b cannot be expressed in CoreXPath, however.

Definition 49.3 (Def 3.1 from Marx). *Let L be a language for which $[\![R]\!]_{\mathit{tree}}$ is defined for every $R \in L$. We call L* expressively complete for first-order definable paths *if for every $\varphi(x,y) \in FO^{\mathit{tree}}$, there exists a formula $R \in L$ such that $[\![\varphi(x,y)]\!]_{\mathit{tree}} = [\![R]\!]_{\mathit{tree}}$.*

We state the following result without proof.

Theorem 49.4. *CondXPath is expressively complete for first-order definable paths. In particular,*

- *every CondXPath path formula is equivalent to an FO^{tree} formula $\varphi(x,y)$ in three variables and*
- *for every FO^{tree} formula $\varphi(x,y)$ there exists an equivalent CondXPath expression.*

Complexity of Satisfiability

Theorem 49.5. *The satisfiability problems for* CoreXPath *and* CondXPath *are* ExpTime*-complete.*

Wim:
Upper bound is basically because emptiness of 2-way alternating tree walking automata is in ExpTime. Too technical. Lower bound was shown by Neven / Schwentick. Didn't check their proof yet but it's probably by reduction from 2-player corridor tiling, as much of the similar proofs are. These techniques are nice but may blow up the chapter too much. I think that Chapter 48 will demonstrate these techniques also, but they will be easier there.

Wim:
Maybe a proof for a fragment. Downward. Immediately the exptime algorithm. (Without mentioning tree automata explicitly.)

Wim:
I'm not sure yet about what to do here. I asked Diego Figueira if he has an idea (Sept 6, 2019).

Wim:
Try: Give the result. Give definition of alternating tree walk automata in exercise section and give as an exercise the translation into the automata. Say that people can use emptiness of 2ATWA as a black box.

50

MSO, Tree Automata, and Monadic Datalog

In this chapter, we go to even more expressive queries by extending first-order logic with quantification over sets. The logic we obtain in this manner is called *monadic second-order logic* or simply *MSO*. We will prove that the expressive power of MSO corresponds precisely to that of *finite automata over trees* and *monadic Datalog*. As such, all these formalisms capture precisely the so-called *regular tree languages*.

Monadic Second Order Logic over Trees

We assume a countably infinite set of *set variables*. Set variables will be denoted by X, Y, Z, ... Intuitively, such variables will range over sets of nodes in our trees. We inductively define formulae of *monadic second-order logic (MSO)* over the vocabulary

$$\sigma_{\mathsf{tree}} = ((\mathrm{Lab}_a)_{a \in \Lambda}, R^{\mathrm{fc}}, R^{\mathrm{ns}}, R^{\mathrm{d}}, R^{\mathrm{fs}}, \mathrm{root}, \mathrm{leaf}, \mathrm{LastSib}) ,$$

introduced in Chapter 46, as follows:

- Every first-order formula over vocabulary σ_{tree} is an MSO formula over σ_{tree}.
- If X is a set variable and y a first-order variable, then $X(y)$ is an MSO formula.
- If φ_1 and φ_2 are MSO formulae, then $(\varphi_1 \wedge \varphi_2)$, $(\varphi_1 \vee \varphi_2)$, and $(\neg\varphi_1)$ are MSO formulae.
- If φ is a formula and X is a set variable, then $(\exists X\, \varphi)$ and $(\forall X\, \varphi)$ are MSO formulae.

The set of *free* variables of a formula φ, denoted $\mathsf{FV}(\varphi)$, is defined as follows:

- $\mathsf{FV}(x = y) = \mathsf{FV}(R(x, y)) = \{x, y\}$ for every $R \in \{R^{\mathrm{fc}}, R^{\mathrm{ns}}, R^{\mathrm{d}}, R^{\mathrm{fs}}\}$ and $\mathsf{FV}(S(x)) = \{x\}$ for every $S \in \{\mathrm{root}, \mathrm{leaf}, \mathrm{LastSib}\}$.
- $\mathsf{FV}(X(y)) = \{X, y\}$.
- $\mathsf{FV}(\neg\varphi) = \mathsf{FV}(\varphi)$.
- $\mathsf{FV}(\varphi_1 \vee \varphi_2) = \mathsf{FV}(\varphi_1 \wedge \varphi_2) = \mathsf{FV}(\varphi_1) \cup \mathsf{FV}(\varphi_2)$.
- $\mathsf{FV}(\exists x\ \varphi) = \mathsf{FV}(\forall\ \varphi) = \mathsf{FV}(\varphi) - \{x\}$.
- $\mathsf{FV}(\exists X\ \varphi) = \mathsf{FV}(\forall X\ \varphi) = \mathsf{FV}(\varphi) = \{X\}$.

We now define the semantics of an MSO formula φ on a tree T. To this end, we view T as a database over schema σ_{tree}. We will inductively define the semantics with respect to an *assignment* η, which associates an element of $\mathrm{Dom}(T)$ with each free first-order variable of φ and a subset of $\mathrm{Dom}(T)$ with each free set variable of φ.

We only highlight the parts of the defnition that extend the definition of the semantics of first-order logic.

- If $\varphi = X(y)$ then $(T, \eta) \models \varphi$ if and only if $\eta(x) \in \eta(X)$.
- If $\varphi = (\exists X \psi)$, then $(T, \eta) \models \varphi$ if and only if $(T, \eta[S/X])$ for some $S \subseteq \mathrm{Dom}(T)$.
- If $\varphi = (\forall X \psi)$, then $(T, \eta) \models \varphi$ if and only if $(T, \eta[S/X])$ for every $S \subseteq \mathrm{Dom}(T)$.

Remark 50.1. The relations R^{d}, R^{fs}, root, leaf, and LastSib are expressible in MSO over the vocabulary

$$\sigma_{\mathsf{fcns}} = ((\mathrm{Lab}_a)_{a \in \Lambda}, R^{\mathrm{fc}}, R^{\mathrm{ns}}) \,.$$

We will therefore sometimes only focus on MSO over this vocabulary.

Tree Automata

A *nondeterministic unranked tree automaton (NTA)* is a quadruple

$$A = (Q, \Sigma, \delta, F) \,,$$

where Q is a finite set of *states*, $\Sigma \subseteq \Lambda$ is a finite, non-empty set of labels, $F \subseteq Q$ is a set of *accepting states*, and $\delta : Q \times \Sigma \to 2^{Q^*}$ is its *transition function*, which is such that $\delta(q, a)$ is a regular word language over Q for every $a \in \Sigma$ and $q \in Q$. Unless we say otherwise, we always assume that $\delta(q, a)$ is represented by a nondeterministic finite automaton (NFA) over words. The *size* of A, denoted $\|A\|$ is defined as $|Q| + \sum_{(q,a) \in Q \times \Sigma} \|N_{q,a}\|$, where $\|N_{q,a}\|$ is the size of the NFA representing $\delta(q, a)$.

A *run* of A on a tree $T = (V, E_{\downarrow}, E_{\rightarrow}, \mathrm{lab})$ is a mapping $\lambda : V \to Q$ such that, for every $u \in V$ with ordered sequence of children $u_1, \ldots, u_n$, the word

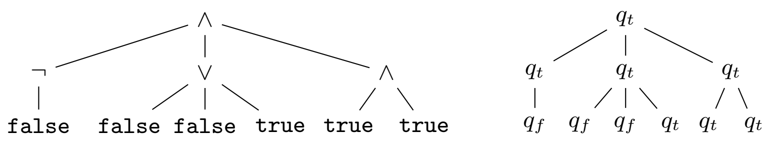


Fig. 50.1: A tree (left) and a run of the tree automaton of Example 50.2 over the tree.

$\lambda(u_1)\cdots\lambda(u_n)$ is in $\delta(\lambda(u), \text{lab}(u))$. Notice that, when u has no children, this condition reduces to $\varepsilon \in \delta(\lambda(u), \text{lab}(u))$. A run is *accepting* if $\lambda(\text{root}(T)) \in F$, that is, it maps the root to an accepting state. A tree T is *accepted by A* if there exists an accepting run of A on T. The set of all accepted trees is denoted by $L(A)$ and is called the *language of A*.

We extend the definition of δ to trees by defining a function δ^* as follows:

- $\delta^*(a) := \{q \mid \varepsilon \in \delta(q, a)\}$; and
- $\delta^*(a(T_1, \ldots, T_n)) := \{q \mid \exists q_1 \in \delta^*(T_1), \ldots, \exists q_n \in \delta^*(T_n) \text{ with } q_1 \cdots q_n \in \delta(q, a)\}$.

Notice that a tree T is accepted by A if and only if $\delta^*(T) \cap F \neq \emptyset$.

Example 50.2. We give an example of a tree automaton A that recognizes tree representations of Boolean formulas that evaluate to `true` (as in Figure 50.1, left). The automaton has state set $Q = \{q_t, q_f\}$ and accepting states $F = \{q_t\}$. The transition function is defined as follows (we use regular expressions to denote the regular languages):

$$\begin{array}{llll} \delta(q_t, \wedge) = q_t^* & \delta(q_t, \vee) = q_f^* q_t (q_t + q_f)^* & \delta(q_t, \neg) = q_f & \delta(q_t, \texttt{true}) = \varepsilon \\ \delta(q_f, \vee) = q_f^* & \delta(q_f, \wedge) = q_t^* q_f (q_t + q_f)^* & \delta(q_f, \neg) = q_t & \delta(q_f, \texttt{false}) = \varepsilon \end{array}$$

A run of A on the tree in Figure 50.1(left) is depicted in Figure 50.1(right).

Algorithms on Tree Automata

We review some standard algorithms on tree automata. First, we show that unions and intersections of tree automata can be constructed in combined linear time.

Theorem 50.3. *Let A_1 and A_2 be NTAs. Then NTAs for $L(A_1) \cap L(A_2)$ and $L(A_1) \cup L(A_2)$ can be constructed in time $O(\|A_1\|\|A_2\|)$.*

Proof. The result follows from a product construction for NTAs. Let $A_1 = (Q_1, \Sigma, \delta_1, F_1)$ and $A_2 = (Q_2, \Sigma, \delta_2, F_2)$. We first explain how to construct an automaton A for $L(A_1) \cap L(A_2)$. Here, $A = (Q_1 \times Q_2, \Sigma, \delta, F_1 \times F_2)$, where $\delta((q, q'), a) = \{(q_1, q_1') \cdots (q_n, q_n') \mid q_1 \cdots q_n \in \delta(q, a) \text{ and } q_1' \cdots q_n' \in \delta(q', a)\}$. If the NFAs for $\delta(q, a)$ and $\delta(q', a)$ use k_1 and k_2 states respectively, then an

Algorithm 12 Computing the set R of reachable states of an NTA

```
1: $\mathsf{R}_1 := \{q \in Q \mid \exists a \in \Sigma$ such that $\varepsilon \in \delta(q,a)\}$;
2: **for** $i := 2$ to $|Q|$ **do**
3:     $\mathsf{R}_i := \{q \in Q \mid$ there exists an $a \in \Sigma$ such that $\delta(q,a) \cap \mathsf{R}^*_{i-1} \neq \emptyset\}$;
4: $\mathsf{R} := \mathsf{R}_{|Q|}$;
```

NFA for $\delta((q,q'),a)$ with k_1k_2 states can be constructed. The construction for $L(A_1) \cup L(A_2)$ is analogous but uses $(F_1 \times Q_2) \cup (Q_1 \times F_2)$ as accepting states. □

We now discuss the problem of testing whether a tree automaton accepts at least one tree. This problem is an automata-theoretic version of *satisfiability* in logic and is typically called *non-emptiness*, since it asks if the language of the automaton is non-empty.

PROBLEM:	NTA-Non-emptiness
INPUT:	a non-deterministic tree automaton A
OUTPUT:	yes if $L(A) \neq \emptyset$ and no otherwise

Theorem 50.4. *NTA-Non-emptiness is in* PTIME.

Proof. Solving NTA-Non-emptiness amounts to deciding if there exists a tree T such that $\delta^*(T)$ contains an accepting state. Let $A = (Q, \Sigma, \delta, F)$ be an NTA. Algorithm 12 computes the set of states $\mathsf{R} := \{q \mid \exists$ tree T such that $q \in \delta^*(T)\}$ in a bottom-up manner. Clearly, $L(A) \neq \emptyset$ if and only if $\mathsf{R} \cap F \neq \emptyset$. Note that $\mathsf{R}_i \subseteq \mathsf{R}_{i+1}$ and $\mathsf{R}_1 = \{\delta^*(a) \mid a \in \Sigma\}$. We argue that the algorithm is in PTIME. Clearly, R_1 can be computed in linear time. Further, the for-loop makes a linear number of iterations. Every iteration is a linear number of non-emptiness tests of the intersection of an NFA with R^*_{i-1} where $\mathsf{R}_{i-1} \subseteq Q$. As emptiness for NFAs is in linear time, the latter is also in linear time. □

Equivalence of MSO and Tree Automata

We will prove that MSO and tree automata are equally expressive *over binary trees*, that is, trees in which every node has zero or two children. The result also holds for general trees, but the proof for binary trees is less technical. The difference between the two proofs mainly lies in the following lemma, which shows that NTAs can be complemented. We denote the set of binary trees with labels in Σ by $\mathcal{T}^b_\Sigma$.

Lemma 50.5. *Let A be an NTA such that $L(A) \subseteq \mathcal{T}^b_\Sigma$. Then an NTA for $\mathcal{T}^b_\Sigma - L(A)$ can be constructed in time $O(2^{\|A\|})$.*

Proof. Let $A = (Q_A, \Sigma, \delta_A, F_A)$. The main idea of the proof is to compute a powerset automaton for A and to complement it. More precisely, we first construct an NTA $B = (Q_B, \Sigma, \delta_B, F_B)$ for $L(A)$ as follows. Define $Q_B = 2^{Q_A}$ and $F_B = \{S \mid S \cap F_A \neq \emptyset\}$. The transition function δ_B is defined such that, for each tree T, $\delta_B^*(T) = \{S\}$, where $S = \delta_A^*(T)$. That is, $\delta_B(S, a) = \{\varepsilon \mid \varepsilon \in \delta(q, a)$ for every $q \in S\} \cup \{S_1 S_2 \mid \forall q \in S, \exists q_1 \in S_1, \exists q_2 \in S_2$ such that $q_1 q_2 \in \delta_A(q, a)\}$. Finally, the NTA for $\mathcal{T}_\Sigma^b - L(A)$ is $\overline{B} = (Q_B, \Sigma, \delta_B, Q_B - F_B)$. □

Theorem 50.6. *Let L be a set of binary trees using labels from a nonempty finite set Σ. Then L is MSO-definable if and only if it is recognizable by an NTA.*

Proof. Let L be recognizable by an NTA $A = (Q, \Sigma, \delta, F)$. Assume that $Q = [1, n]$. We construct an MSO formula φ such that $L = \{T \mid T \models \varphi\}$. Intuitively, φ expresses the existence of an accepting run. For every $i \in [1, n]$, it uses a set variable X_i for the set of nodes visited in state i. Consider the formulae

$$\begin{aligned}
\varphi_1 &= \big(\forall x\, (\vee_{i=1}^n X_i(x))\big) \wedge \big(\textstyle\bigvee_{i \neq j} \forall x\, (X_i(x) \to \neg X_j(x))\big),\\
\varphi_2 &= \forall x\, \big(\neg(\exists y\, R^{\text{fc}}(y, x)) \to \textstyle\bigvee_{i \in F} X_i(x)\big),\\
\varphi_3 &= \textstyle\bigwedge_{a \in \Sigma} \forall x\, \big(\text{Lab}_a(x) \wedge \neg(\exists y\, R^{\text{fc}}(x, y))\big) \to \textstyle\bigvee_{\varepsilon \in \delta(i,a)} X_i(x)\big), \text{ and}\\
\varphi_4 &= \textstyle\bigwedge_{a \in \Sigma} \forall x \forall x_1 \forall x_2\, \big((\text{Lab}_a(x) \wedge R^{\text{fc}}(x, x_1) \wedge R^{\text{ns}}(x_1, x_2))\\
&\qquad\qquad \to (\textstyle\bigvee_{(k,\ell) \in \delta(i,a)} (X_i(x) \wedge X_k(x_1) \wedge X_\ell(x_2)))\big)\,.
\end{aligned}$$

Formula φ_1 expresses that every node is assigned exactly one state, φ_2 expresses that the root is assigned an accepting state, φ_3 expresses that the leafs are labeled in accordance with δ, and φ_4 expresses the internal nodes are labeled in accordance with δ. It is easy to show that A accepts a tree T if and only if there exist sets $X_1, \ldots, X_n$ that satisfy φ_1, φ_2, φ_3, and φ_4. We conclude this direction by defining

$$\varphi = \exists X_1 \cdots \exists X_n\, (\varphi_1 \wedge \varphi_2 \wedge \varphi_3 \wedge \varphi_4)\,.$$

For the other direction, let φ be an MSO-formula over the signature $\sigma_{\mathsf{fcns}} = ((\text{Lab}_a)_{a \in \Lambda}, R^{\text{fc}}, R^{\text{ns}})$. We can assume w.l.o.g. that φ is generated by the grammar

$$\varphi := \text{Lab}_a(x) \mid R^{\text{fc}}(x, y) \mid R^{\text{ns}}(x, y) \mid x = y \mid \varphi \wedge \varphi \mid \neg\varphi \mid \exists x\, \varphi \mid \exists X\, \varphi\,,$$

where $a \in \Sigma$.[1] We will further simplify the syntax of MSO-formulae by eliminating the first-order variables. That is, we introduce another logic with the same expressive power than MSO over σ_{tree} that we call MSO$_0$. The idea is that MSO$_0$ introduces some syntactic sugar, but constrains formulae elsewhere. MSO$_0$-formulae φ over σ_{fcns} are generated over the grammar

[1] For instance, subformulae $\text{Lab}_b(x)$ with $b \notin \Sigma$ can be replaced by $\text{Lab}_a(x) \wedge \neg\text{Lab}_a(x)$ for some $a \in \Sigma$.

$$\varphi := \mathsf{sing}(X) \mid \mathrm{Lab}_a(X) \mid X \subseteq Y \mid R^{\mathrm{fc}}(X,Y) \mid R^{\mathrm{ns}}(X,Y) \mid \varphi \wedge \varphi \mid \neg\varphi \mid \exists X\, \varphi\,,$$

meaning respectively that X is a singleton; all variables in X are labeled a; X is a subset of Y; X and Y are singletons $\{x\}$, $\{y\}$ with $R^{\mathrm{fc}}(x,y)$; X and Y are singletons $\{x\}$, $\{y\}$ with $R^{\mathrm{ns}}(x,y)$; and $\exists X\varphi$. The semantics of these formulae can be defined in MSO over σ_{fcns}. For instance, $\mathsf{sing}(X)$ can be written as $\forall x\forall y\,(X(x) \wedge X(y)) \rightarrow (x = y)$ and $\mathrm{Lab}_a(X)$ as $\forall x(X(x) \rightarrow \mathrm{Lab}_a(x))$. Conversely, it is easy to show by structural induction that each MSO-formula can be tranlated into an equivalent MSO_o-formula.

We can assume w.l.o.g. that φ is an MSO_0 formula, that it does not re-use variables, and that its variables are $X_1, \ldots, X_n$. For a subformula ψ of φ with free variables S, we will construct an NTA that accepts trees for which the labels are pairs (a, f), where $a \in \Sigma$ and $f \in \{0,1\}^S$, i.e., f is a function from S to $\{0,1\}$. Intuitively, such a tree T represents a tree T' over Σ, together with an assigment of the free variables, and the task of the NTA is to accept precisely those that satisfy ψ.

More precisely, let T be a tree over $\Sigma \times \{0,1\}^S$. We denote by T_Σ the tree obtained from T by replacing each label (a, f) by a. We define η_T to be the assigment obtained from T by mapping each free variable X_i to $\eta(X_i) = \{u \mid \mathrm{lab}(u) = (a, f) \text{ and } f(X_i) = 1\}$.

We now inductively define NTAs A_ψ that accept precisely the trees T such that $(T_\Sigma, \eta_T) \models \psi$. An NTA for $\mathrm{Lab}_a(X_i)$ tests that the tree has no node labeled (b, f) with $f(X_i) = 1$ and $b \neq a$. An NTA for $X \subseteq Y$ is similar. An NTA for $R^{\mathrm{fc}}(X_i, X_j)$ tests that there is exactly one node u_1 labeled (a, f_1) with $f_1(X_i) = 1$, exactly one node u_2 labeled (b, f_2) with $f_2(X_j) = 1$, and that u_2 is a child of u_1. The NTAs for $\mathsf{sing}(X)$ and $R^{\mathrm{ns}}(x_i, x_j)$ are similar.

For the inductive step, in the cases $\psi = \neg\psi_1$ and $\psi = \psi_1 \wedge \psi_2$, we can use Theorem 50.3 and Lemma 50.5. In the case $\psi = \exists X_i\, \psi'$, assume that we have an NTA for ψ'. The NTA for ψ reads labels (a, f) for which $f(X_i)$ is undefined. It guesses labels of the form (a, f') for which f and f' agree on every free variable from ψ, and then runs as if it would be the NTA for ψ'. $\square$

Equivalence of MSO and Monadic Datalog

We now establish another characterization of MSO-definable queries, namely in terms of Datalog programs (see Chapter 27). We call a Datalog program Π *monadic* if all intensional relations are unary. We will prove that the Boolean monadic Datalog queries are precisely the Boolean MSO-definable queries. The equivalence between MSO-definable queries and monadic Datalog queries is known to hold for unary queries as well, but the result presented here is easier to prove. **(Wim)**[2]

Theorem 50.7. *A set L of binary trees is MSO-definable if and only if it is recognizable by a Boolean monadic Datalog program.*

[2] **Wim:** Sync terminology ("recognizable") with Datalog chapter.

Proof. Let (Π, R_1) be a monadic Datalog query, where $\mathsf{edb}(\Pi) = \sigma_{\mathsf{tree}}$ and $\mathsf{idb}(\Pi) = \{R_1, \ldots, R_n\}$. Then, the MSO query can be defined as

$$\varphi := \forall R_1 \cdots \forall R_n \bigwedge_{\rho \in \Pi} \varphi_\rho \,,$$

where φ_ρ is the first-order sentence associated to Datalog rule ρ, that was used for defining the model-theoretic semantics of Datalog in Chapter 27. **(Wim)**[3]

Conversely, let $A = (Q, \Sigma, \delta, F)$ be an NTA. We define a Boolean monadic Datalog program (Π, R) that tests if there exists an accepting run of A on a given tree T. To this end, (Π, R) will have a unary predicate q for each $q \in Q$ that will be satisfied in node u of T if and only if $q \in \delta^*(T_{|u})$. **(Wim)**[4]

For every state $q \in Q$ and $a \in \Sigma$, let $N_{q,a} = (Q_{q,a}, Q, \delta_{q,a}, I_{q,a}, F_{q,a})$ be an NFA for the language $\delta(q,a)$. We can assume w.l.o.g. that Q and all state sets $Q_{q,a}$ are pairwise disjoint. In the following, we will always use q, q' to denote states from Q and p, p' to denote states from the sets $Q_{q,a}$.

Notice that we can view all $N_{q,a}$ as one automaton $(Q_N, Q, \delta_N, I_N, F_N)$, where $Q_N = \cup_{q \in Q} \cup_{a \in \Sigma} Q_{q,a}$, $I_N = \cup_{q,a} I_{q,a}$, $F_N = \cup_{q,a} F_{q,a}$, and δ_N is defined as follows. For a given $p \in \cup_{q,a} Q_{q,a}$, we define $\delta_N(p,q)$ to be the set $\delta_{q',a}(p,q)$ for the unique q' and a such that $p \in Q_{q',a}$. We define I_q to be the set of states reachable in N by reading q from an initial state, that is, $I_q := \{p \mid \exists p' \in I_N$ such that $p \in \delta_N(p',q)\}$.

For each transition such that $\varepsilon \in \delta(q,a)$, the program Π has the rule

$$q(x) \mathrel{:-} \mathrm{leaf}(x) \wedge \mathrm{Lab}_a(x).$$

Next, we define rules that capture the computation of the automata $N_{q,a}$. Therefore, the rules

$$\begin{array}{lll}
p(x) & \mathrel{:-} R^{\mathrm{fc}}(y,x), \mathrm{leaf}(x), q(x). & \text{for every } p \in I_q \\
p(x) & \mathrel{:-} R^{\mathrm{fc}}(y,x), R^{\mathrm{fc}}(x,y'), q(x). & \text{for every } p \in I_q \\
p(x) & \mathrel{:-} R^{\mathrm{ns}}(y,x), p'(y), \mathrm{leaf}(x), q(x). & \text{for every } p \in \delta'(p',q) \\
p(x) & \mathrel{:-} R^{\mathrm{ns}}(y,x), p'(y), R^{\mathrm{fc}}(x,y'), q(x). & \text{for every } p \in \delta'(p',q)
\end{array}$$

are added to Π. The rules

$$q(x) \mathrel{:-} \mathrm{Lab}_a(x), R^{\mathrm{fc}}(x,y), \mathrm{LastSib}(y,y'), p(y') \qquad \text{for every } p \in F_{i,a}$$

test if the children of x can be visited in a sequence of states that is accepted by $N_{q,a}$. Finally, the rules $R() \mathrel{:-} \mathrm{root}(x), q(x)$ for every $q \in F$ tests if the root can be labeled with an accepting state. $\square$

[3] **Wim:** Conversion should be correct because the minimal model is the intersection of all models of Π, and an interpretation of $R_1, \ldots, R_n$ is a model of Π iff $\bigwedge_{\rho \in \Pi} \varphi_\rho$ is true.

[4] **Wim:** Not sure if this is good terminology.

51

Schemas for XML

Database schemas for tree-structured data follow a rather different approach from those for relational data. In fact, their underlying ideas come from *context-free grammars* and the tree automata we introduced in Chapter 50. A very simple formalism that was used to define schemas for XML data are so-called *Document Type Definitions (DTDs)*. On an abstract level, a DTD is an *extended context-free grammar*, as the following definition shows.

Definition 51.1. *A* Document Type Definition (DTD) *over alphabet Σ is a triple $d = (\Sigma, \rho, S)$ where $\Sigma \subseteq \Delta$ is a finite set of* labels*, ρ is a function from Σ to the set of regular expression over Σ, and $S \subseteq \Sigma$ is a set of* start labels*.*

In the context of XML schema languages, the labels Σ are also called *element names*. An ordered tree $T = (V, E_{\downarrow}, E_{\rightarrow}, \text{lab})$ *satisfies* d if $\text{lab}(\text{root}(T)) \in S$ and, for every $u \in V$ with ordered sequence of children $u_1, \ldots, u_n$, the word $\text{lab}(u_1) \cdots \text{lab}(u_n)$ is in $L(r)$, where $r = \rho(\text{lab}(u))$. By $L(d)$ we denote the set of trees satisfying d. In real-world DTDs, ρ is usually written as a set of rules rather than a function. For this reason, we sometimes also treat ρ as a set of rules, where we write $a \rightarrow r$ if $\rho(a) = r$. The *size* of a DTD is $\sum_{a \in \Sigma} \|\rho(a)\|$.

Example 51.2. We will use a running example as depicted in Figure 51.1. The data is an XML document describing the inventory of a store, which sells items under different categories. Each item has a maker, model, price, and optionally a description and a discount. Figure 51.2 depicts the tree structure induced by the nesting of the tags in the XML document. It is at this level of abstraction that DTDs operate.

A DTD describing such data could be defined with $\Sigma = \{$inventory, category, item, maker, model, description, price, discount$\}$ and $S = \{$inventory$\}$. The function ρ is defined as follows:

$$\begin{array}{lll}
\mathsf{inventory} & \rightarrow & \mathsf{category}^* \\
\mathsf{category} & \rightarrow & \mathsf{item}^* \\
\mathsf{item} & \rightarrow & \mathsf{maker\ model\ price\ description?\ discount?}
\end{array}$$

```
<?xml version="1.0" encoding="UTF-8"?>
<inventory>
  <category name="concert guitars">
    <item>
      <maker> Tandler </maker>
      <model> Advanced Student </model>
      <description> Spruce top,
                    Indian rosewood back and sides </description>
      <price> Please ask </price>
    </item>
    <item>
      <maker> Hanika </maker>
      <model> Grand Concert </model>
      <description> Spruce top,
                    Palo Escrito back and sides </description>
      <price> 4299 </price>
      <discount> 10 </discount>
    </item>
    ...
  </category>
  ...
</inventory>
```

Fig. 51.1: Fragment of an XML document representing the inventory of a store

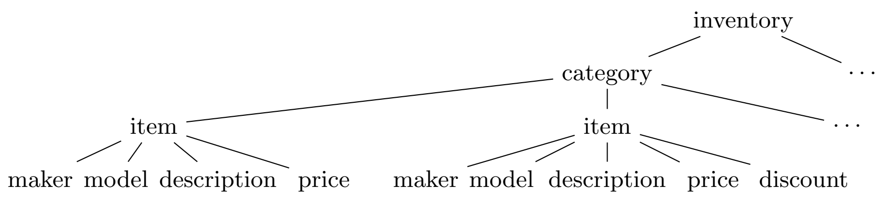


Fig. 51.2: The tag structure of the data in Figure 51.1 as a sibling-ordered tree

It describes that inventory should have zero or more category children which, in turn, have zero or more item children. Each item has children labeled maker, model, price and, optionally description and discount, from left to right. The tree in Figure 51.2 satisfies this DTD.

Now assume that we want to express in our running example that each category contains at least one item with a discount child. DTDs cannot express this, since they can associate a single regular expression to item.[1] Other

[1] In Theorem 51.10 we will learn a method to formally prove that DTDs cannot express this property.

schema languages such as XML Schema and Relax NG solve this limitation by extending DTDs with *types*.

Definition 51.3. *An* extended DTD (EDTD) *is a tuple* $D = (\Sigma, \Gamma, \rho, S, \mu)$*, where* $\Sigma \subseteq \Delta$ *is a finite set of* labels*,* $\Gamma \subseteq \Delta$ *is a finite set of* types*,* (Γ, ρ, S) *is a DTD over alphabet* Γ*, and* $\mu : \Gamma \to \Sigma$*.*

Intuitively, a tree satisfies an EDTD if there exists an assignment of types to all nodes such that the typed tree is a derivation tree of the underlying DTD (Γ, ρ, S). Formally, a sibling-ordered tree T *satisfies* D if there exists a tree $T^\Gamma \in L((\Gamma, \rho, S))$ such that $T = \mu(T^\Gamma)$. Here, we denote by $\mu(T^\Gamma)$ the tree obtained from T by replacing each label a by $\mu(a)$. We call T^Γ a *witness* for T. Again, we denote the set of trees satisfying D by $L(D)$. The *size* of an EDTD D is the size of the underlying DTD (Γ, ρ, S). In examples, we denote EDTDs as sets of rules, just as we did before with DTDs. We say that an EDTD D is *reduced* if d is a reduced DTD.

Example 51.4. Continuing Example 51.2, we will define an EDTD that describes inventories in which each category has at least one item with a discount child. We take Σ as before and use $\Gamma = \{$inventory, category, normal, discounted, maker, model, description, price, discount$\}$. For defining ρ, we will use a simplified notation, much like how schemas are written in XML Schema or Relax NG: we use $a[b]$ to denote $a \in \Sigma$ and $b \in \Gamma$ with $\mu(b) = a$, and we abbreviate $a[a]$ by a. The mapping ρ can now be defined as follows.

$$\begin{array}{ll}
\mathsf{inventory} & \to \mathsf{category}^* \\
\mathsf{category} & \to \mathsf{item[normal]}^*\ \mathsf{item[discounted]} \\
 & \qquad\qquad\qquad (\mathsf{item[normal]} + \mathsf{item[discounted]})^* \\
\mathsf{item[normal]} & \to \mathsf{maker\ model\ price\ description?} \\
\mathsf{item[discounted]} & \to \mathsf{maker\ model\ price\ description?\ discount}
\end{array}$$

The rule for category requires that at least one child is assigned the type discounted. As we can see in the rule for item[discounted], this means that the item must have a child with label discount.

Wim:
I believe that it could help here to add a figure containing the typed version of the tree in Figure 51.2. But this would cost some extra space.

Figure 51.3a shows how the rule for category in Example 51.4 can be defined in Relax NG (with a partial definition of the rule for tye type "normal"). In Relax NG, `element` blocks are used to define Σ elements (and their content), whereas `define` blocks are used to define types. Since we don't need a special type for category, we can define it immediately using an `element` block.

In Figure 51.3b, we try to mimic this behavior using XML Schema. However, the schema is not syntactically correct because it violates the *Element Declarations Consistent (EDC)* constraint, which forbids the occurrence of different types associated to the same element name in a regular expression

```
<element name="category">
  <zeroOrMore><ref name="normal"/></zeroOrMore>
  <ref name="discounted"/>
  <zeroOrMore>
    <choice><ref name="normal"/><ref name="discounted"/></choice>
  </zeroOrMore>
</element>

<define name="normal">
  <element name="item"> ... </element>
</define>
```

(a) Fragment of a Relax NG schema corresponding to the EDTD rule for category

```
<element name="category">
  <complexType>
    <sequence>
      <element name="item" type="normal"
               minOccurs="0" maxOccurs="unbounded"/>
      <element name="item" type="discounted"/>
      <choice minOccurs="0" maxOccurs="unbounded">
        <element name="item" type="normal"/>
        <element name="item" type="discounted"/>
      </choice>
    </sequence>
  </complexType>
</element>

<complexType name="normal">
  <element name="item"> ... </element>
</complexType>
```

(b) Fragment of an XSD (violating EDC) corresponding to the EDTD of Example 51.4.

$\rho(t)$. So, the occurrence of both types normal and discount associated to the same element name item is a violation of this constraint. We formalize this constraint next.

Definition 51.5. *Let $D = (\Sigma, \Gamma, \rho, S, \mu)$ be an EDTD. A regular expression r over Γ is* single-type *if it does not contain distinct symbols t_1 and t_2 with $\mu(t_1) = \mu(t_2)$. We say that D is a* single-type EDTD (stEDTD) *when S does not contain distinct types t_1 and t_2 with $\mu(t_1) = \mu(t_2)$ and the regular expression $\rho(t)$ is single-type for every $t \in \Gamma$.*

Even though single-type EDTDs are easy to define, they capture a large and important part of XML Schema, namely the part that specifies the structure and labels of the tags in XML trees.

EDTDs are Equivalent to Tree Automata

Theorem 51.6. *Each EDTD D can be translated in time $O(\|D\|)$ to an equivalent NTA. Conversely, each NTA A over alphabet Σ can be translated in $O(\|A\|\|\Sigma\|^2)$ time to an equivalent EDTD.*

Proof. Let $D = (\Sigma, \Gamma, \rho, S, \mu)$ be an EDTD. An equivalent NTA $A = (Q, \Sigma, \delta, F)$ is obtained by taking $Q = \Gamma$, $F = S$, and $\delta(t, \mu(t)) = \rho(t)$ for every type $t \in \Gamma$.

Conversely, let $A = (Q, \Sigma, \delta, F)$ be an NTA. An equivalent EDTD $D = (\Sigma, \Gamma, \rho, S, \mu)$ is obtained by taking $\Gamma = Q \times \Sigma$, $S = F \times \Sigma$, and $\mu((q, a)) = a$ for each type (q, a). Let $L_{(q,a)}$ denote the language obtained from $\delta(q, a)$ by replacing, in each word, every symbol $p \in Q$ by the disjunction $\sum_{b \in \Sigma}(p, b)$. Since $\delta(q, a)$ is a regular language over Q, the language $L_{(q,a)}$ is also regular. For every $(q, a) \in \Gamma$, we define $\rho((q, a))$ to be a regular expression for $L_{(q,a)}$.
□

Complexity of Validation

We now consider the problem of testing if a tree satisfies a given schema. Since we have three types of schema languages (DTD, EDTD, and stEDTD), we define the problem in a general form.

PROBLEM:	$\mathcal{L}$-Validation
INPUT:	a tree T and a schema D from the class $\mathcal{L}$
OUTPUT:	yes if $t \in L(D)$ and no otherwise

Theorem 51.7. *EDTD-Validation is in* PTIME.***(Wim)***[2]

Proof. Let T be the tree and $D = (\Sigma, \Gamma, \rho, S, \mu)$ the EDTD. We assume w.l.o.g. that T only uses labels in Σ, since otherwise it can never be in $L(D)$. We compute in a bottom-up manner the set of types that can be assigned to each node u of T. If u is a leaf with label $a \in \Sigma$, then $\mathsf{Types}(u) = \{t \in \Gamma \mid \mu(t) = a \text{ and } \varepsilon \in L(\rho(t))\}$. If u is an internal node with label a and ordered list of children $u_1, \ldots, u_n$, then $\mathsf{Types}(u) = \{t \in \Gamma \mid \mu(t) = a$ and $\exists t_1 \cdots t_n \in L(\rho(t))$ such that $t_i \in \mathsf{Types}(u_i)$ for every $i \in [1, n]\}$. We accept if $\mathsf{Types}(\text{root}(T))$ contains a type from S. □

We have the following corollary since DTDs and stEDTDs are special cases of EDTDs.

Corollary 51.8. *DTD-Validation and stEDTD-Validation are in* PTIME.

[2] **Wim:** If we would discuss tree automata evaluation before, then this is an immediate consequence of Thm 51.6.

Expressiveness

The following is immediate from Theorem 51.6.

Theorem 51.9. *EDTDs recognize precisely the regular tree languages.*

We will now also characterize the languages that are definable by DTDs and stEDTDs. To this end, we need the notion of *subtree exchange*. Let $T = (V, E_\downarrow, E_\rightarrow, \text{lab})$ and $T' = (V', E'_\downarrow, E'_\rightarrow, \text{lab}')$ be two trees, and let $u \in V$. We assume w.l.o.g. that V and V' are disjoint (otherwise, the nodes in V' can be renamed). We denote by $T[u \leftarrow T']$ the tree obtained by replacing the subtree $T|_u$ of T rooted at u by the tree T'. (We omit a formal definition, but $T[u \leftarrow T']$ has none of the nodes in T_u, all the nodes of T' and has the edge $(v, \text{root}(T'))$ instead of the edge (v, u).)

We say that a tree language L is *closed under node-guarded subtree exchange*, if the following holds. Whenever two trees $T_1, T_2 \in L$ have nodes u_1 and u_2 respectively, with $\text{lab}(u_1) = \text{lab}(u_2)$, then $T_1[u_1 \leftarrow T_2|_{u_2}] \in L$.

Theorem 51.10. *DTDs recognize precisely the regular tree languages that are closed under node-guarded subtree exchange.*

Proof. Observe that, by definition, every language recognized by a DTD is closed under node-guarded subtree exchange. Conversely, let $D = (\Sigma, \Gamma, \rho, S, \mu)$ be an EDTD such that $L(D)$ is closed under node-guarded subtree exchange. We construct a DTD $d = (\Sigma, \rho_d, S_d)$ as follows. Define $S_d = \{\mu(t) \mid t \in S\}$. For a regular expression r over Γ, denote by $\mu(r)$ the expression obtained from r by replacing each symbol t with $\mu(t)$. Then we define $\rho_d(a)$ as the disjunction of all $\mu(\rho(t))$ over $\{t \mid \mu(t) = a\}$. It is easy to show that $L(d) = L(D)$. □

We now provide a similar characterization for stEDTDs. For a tree $T = (V, E_\downarrow, E_\rightarrow, \text{lab})$ and node $u \in V$, denote by $\mathsf{ancestorlab}^T(u)$ the word $\text{lab}(u_1) \cdots \text{lab}(u_n)$, where $u_1 \cdots u_n$ is the path from $\text{root}(T)$ to u in T. A tree language L is *closed under ancestor-guarded subtree exchange*, if the following holds. Whenever two trees $T_1, T_2 \in L$ have nodes u_1 and u_2 respectively, with $\mathsf{ancestorlab}^{T_1}(u_1) = \mathsf{ancestorlab}^{T_2}(u_2)$, then $T_1[u_1 \leftarrow T_2|_{u_2}] \in L$. **(Wim)**[3]

Theorem 51.11. *stEDTDs recognize precisely the regular tree languages that are closed under ancestor-guarded subtree exchange.*

Proof. It is not difficult to prove that every language definable by an stEDTD is a regular language closed under ancestor-guarded subtree exchange, see Exercise 53.8.

Conversely, let $D = (\Sigma, \Gamma, \rho, S, \mu)$ be an EDTD such that $L(D)$ is closed under ancestor-guarded subtree exchange. We will construct a single-type EDTD E such that $L(E) = L(D)$. We will assume w.l.o.g. that D only

[3] **Wim:** This can be nicely illustrated with a picture, but this would cost space.

uses *useful* types, that is, for every type $t \in \Gamma$, there exists a fixed tree $T_t \in L((\Gamma, \rho, \{t\}))$. We will make use of the following general property, which immediately follows from the definition of EDTDs:

(†) If T_1, T_2 are trees in $L(D)$ with witnesses T_1', T_2', respectively, such that u_1 in T_1 and u_2 in T_2 have the same type in T_1' and T_2', respectively, then $T_1[u_1 \leftarrow T_2|_{u_2}] \in L(D)$.

For a word $w \in \Sigma^*$ and $a \in \Sigma$, let $\mathsf{types}(wa)$ be the set of all types t_a for which there is a tree $T \in L(D)$ with witness $T' \in L((\Gamma, \rho, S))$, and a node v such that $\mathsf{ancestorlab}^T(v) = wa$ and $\mathrm{lab}^{T'}(v) = t_a$. For each $a \in \Sigma$, let $\Gamma(D, a)$ be the set of all nonempty sets $\mathsf{types}(wa)$, with $w \in \Sigma^*$. Clearly, each $\Gamma(D, a)$ is finite.

We next define $E = (\Sigma, \Gamma_E, \rho_E, S_E, \mu_E)$. Its set of types Γ_E is $\bigcup_{a \in \Sigma} \Gamma(D, a)$ and its set of start types S_E is $\{\mathsf{types}(a) \mid a \in \Sigma\}$. For every type $\tau \in \Gamma(D, a)$, set $\mu_E(\tau) = a$. The mapping ρ_E maps each type $\mathsf{types}(wa)$ to the disjunction of all $\rho(t_a)$ for $t_a \in \mathsf{types}(wa)$, with each t_b in $\rho(t_a)$ with $\mu(t_b) = b$ replaced by $\mathsf{types}(wab)$. Notice that, by (†), the definition of the rules of ρ_E does not depend on the actual choice of wa.

Clearly, E is single-type and $L(D) \subseteq L(E)$. We show that $L(E) \subseteq L(D)$. To this end, let $T \in L(E)$ and let T' be a witness. We call a set N of nodes of T *well-formed*, if (1) for each node $v \in N$, all its ancestors are in N and (2) if a child u of a node v is in N then all children of v are in N. The singleton set N_0 containing the root is well-formed. We say that a tree T_1 *agrees* with T on a well-formed set N_1 of nodes of T_1, if N_1 can be mapped to a well-formed set of nodes N of T by a mapping m which respects the child-relationship, the order of siblings, and the labels of nodes. By definition, $\mathsf{ancestorlab}(v) = \mathsf{ancestorlab}(m(v))$ for every $v \in N$.

As the trees in $L(D)$ and $L(E)$ have the same possible root labels, there exists a tree $T_1 \in L(D)$ which agrees with T on N_0. To complete the proof, it is sufficient to prove the following.

Claim 51.12. *If there exists a tree $T_1 \in L(D)$ which agrees with $T = (V, E_\downarrow, E_\rightarrow, lab)$ on a well-formed set N with $m(N) \subsetneq V$ then there exists $T_2 \in L(D)$ which agrees with T on a well-formed, strict superset of N.*

For the proof of this claim, let T_1 be as stated and let T_1' be its witness. Let $v \in N$ be such that the children of $m(v)$ are not in $m(N)$ and let $wa = \mathsf{ancestorlab}^{T_1}(v) = \mathsf{ancestorlab}^T(m(v))$.

By construction of E, the regular expression $\rho_E(\mathsf{types}(wa))$ is a disjunction $\sum_{t \in \mathsf{types(wa)}} \rho'(t)$, where ρ' is defined as ρ but, for every $b \in \Sigma$, has the type $\mathsf{types}(wab) \in \Gamma_E$ instead of $t_b \in \Gamma$ if $\mu(t_b) = b$. Let t_a be such that the children of $m(v)$ are typed in T' according to a disjunct $\rho'(t_a)$, with $t_a \in \mathsf{types}(wa)$. Thus, there is a tree $T_3 \in L(D)$ with a node u such that $\mathsf{ancestorlab}^{T_3}(u) = wa$ and the type of u is t_a in the witness T_3' for T_3.

Let $v_1, \ldots, v_n$ be the ordered sequence of children of $m(v)$ in T and choose, for each $i \in [1, n]$, a type $f(v_i)$ such that $f(v_1) \cdots f(v_n)$ matches $\rho(t_a)$. Let T_4 be obtained from T_3 by (1) removing everything below u, (2) adding $v_1, \ldots, v_n$ below u, and (3) adding for each child v_i the subtree $T_{f(v_i)}$ which exists because $f(v_i)$ is a useful type. Clearly, $T_4 \in L(D)$ by (†). Furthermore, by the ancestor-closed subtree exchange property, the tree T_2 resulting from T_1 by replacing the subtree rooted at v by the subtree of T_4 rooted at u is in $L(D)$, too. □

52

Static Analysis Under Schema Constraints

In this chapter, we study a number of optimization problems that involve both query- and schema information. The reason why this is relevant is because, while a query may be optimal when considered over the set of all trees, it may not be optimal on the subset of trees described by our schema. A very basic question in this respect is the following.

PROBLEM:	TPQ-Satisfiability with EDTD Information
INPUT:	a Boolean TPQ p and an EDTD D
OUTPUT:	yes if there exists a $T \in L(D)$ such that $T \models p$ and no otherwise

We note that we only consider Boolean queries in this problem, but it can be shown that the general version can be efficiently reduced to this one, see Exercise 53.9.

A DTD $d = (\Sigma, \rho, S)$ is *reduced* if, for every $a \in \Sigma$, there exists a tree $T \in L(d)$ and a node u of T such that $\text{lab}(u) = a$. Hence, for example, the DTD $(\{a\}, \{a \to a\}, a)$ is *not* reduced. An EDTD $D = (\Sigma, \Gamma, \rho, S, \mu)$ is *reduced* if its DTD (Γ, ρ, S) is reduced.

Lemma 52.1. *Given an EDTD, a reduced equivalent EDTD can be computed in* PTIME.

Proof. **(Wim)**[1] The procedure is analogous to reducing context-free grammars. Let $D = (\Sigma, \Gamma, \rho, S, \mu)$ be an EDTD. By *removing a type t from D*, we refer to the following operation. First, remove the rule $t \to r$ from ρ and, for every remaining rule $t' \to r'$, replace every occurrence of t in r' by $\emptyset$. Then, simplify the remaining expression according to the usual equivalence rules of regular expressions such that the expression becomes $\emptyset$ or the symbol $\emptyset$ does not occur anymore. Finally, delete t from Γ, from S, and from the domain of μ. The algorithm for reducing D consists of two steps.

[1] **Wim:** This could become an exercise if the chapter becomes too long.

1. Compute the set Γ_1 of types t such that $(\Sigma, \Gamma, \rho, \{t\}, \mu)$ defines a non-empty set of trees. Then, $D_1 = (\Sigma, \Gamma_1, \rho_1, S_1, \mu_1)$ is obtained from D by removing every type in $\Gamma - \Gamma_1$.
2. Consider the directed graph G_{Γ_1} with nodes Γ_1 and edges $\{(t_1, t_2) \mid t_1 \to r_1 \in \rho$ and t_2 is a symbol in $r_1\}$. Let Γ_2 be the types reachable from some $t \in S_1$ in G_{Γ_1}. Then, D_2 is obtained from D_1 by removing every type in $\Gamma_1 - \Gamma_2$

The resulting EDTD D_2 is reduced and equivalent to D. **(Wim)**[2] □

We now turn to the complexity of the satisfiability of TPQs with respect to EDTD information.

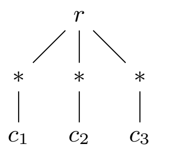
Theorem 52.2. *TPQ-*Satisfiability *with EDTD-Information is* NP*-complete.*

Proof. Let p be a TPQ and let $D = (\Sigma, \Gamma, \rho, S, \mu)$ be an EDTD. By Lemma 52.1, we can assume that D is reduced. We first show that, if p is satisfiable with respect to D, then there is a tree $T \in L(D)$ such that $T \models p$ and T has depth less than $M = (2 \cdot \|p\| + 1)(\|D\| + 1)$. So, assume that p is satisfiable with respect to D and let $T \in L(D)$ be such that $T \models p$ and T has a minimal number of nodes. Towards a contradiction, assume that $\mathsf{depth}(T) \geq M$. Let T_Γ be a witness of T and let m be a match of p in T. We call the nodes in the image of m and their nearest common ancestors the *m-critical nodes*, so there are at most $2 \cdot \|p\|$ m-critical nodes in T. Since $\mathsf{depth}(T) \geq M$, there are two critical nodes nodes u, v in T such that u is ancestor of v, no critical node is on the path from u to v, and $\mathsf{depth}(v) - \mathsf{depth}(u) > |D|$. By the Pigeonhole Principle, there are nodes $u' \neq v'$ on the path from u to v such that u' and v' have the same label in T_Γ. Assume w.l.o.g. that v' is a descendant of u'. By definition of DTDs, $T'_\Gamma = T_\Gamma[u' \leftarrow T|_{v'}]$ is a derivation tree of (Γ, ρ, S) and therefore $T' = T[u' \leftarrow T_{v'}]$ is also in $L(D)$. Furthermore, since all critical nodes of T are still present in T', the mapping m is still a match of p in T'. But this means that T did not have a minimal number of nodes among the trees in $L(D)$ that satisfy p, contradiction. Therefore, if p is consistent with D, then there is a tree $T \in L(D)$ such that $T \models p$ and T depth less than M.

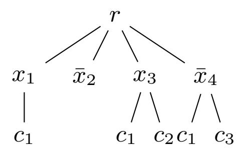
Let $T \in L(D)$ be such a tree of depth less than M such that $T \models p$, and let m be a match of p in T. Let S_m be the set of nodes that are either m-critical, or lie on a path between two m-critical nodes. We refer to the tree induced by S_m as a *witness skeleton*. That is, a witness skeleton is a tree of polynomial size that can be extended to a tree $T \in L(D)$ such that $T \models p$. It follows from the previous discussion that, if p is satisfiable w.r.t. D, then there exists a polynomial-size witness skeleton.

The NP algorithm for testing of p is satisfiable w.r.t. D guesses a polynomial-size tree T_w and tests if it is a witness skeleton. To this end, the algorithm tests

[2] **Wim:** Correctness proof omitted. But it's analogous to the one for context-free grammars.



(a) Tree pattern for for formula (52.1)



(b) A tree generated by the DTD corresponding to formula (52.1)

Fig. 52.1: Gadgets for the NP hardness part of Theorem 52.2.

1. if there exists a match m of p in T_w and
2. if T_w can be extended to a tree in $L(D)$.

For the second test, it guesses for each node u a type t_u. Furthermore, if u has children $u_1, \ldots, u_k$, it tests if there exists a word w in $L(\rho(t_u))$ that has $t_{u_1} \cdots t_{u_k}$ as a subsequence, i.e., $w = w_0 t_{u_1} w_1 \cdots w_{k-1} t_{u_k} w_k$. To test if such a w exists, it suffices to guess a polynomial-length path through the NFA for $L(\rho(t_u))$. Therefore, the problem is in NP.

We show that NP-hardness already holds for DTDs. We reduce from 3SAT. Let $\varphi = c_1 \wedge \cdots \wedge c_m$ be a propositional logic formula in CNF using variables $\{x_1, \ldots, x_n\}$. Each clause c_i is a disjunction of literals, which can be positive or negative. For each variable x_j, denote by $\text{pos}(x_j)$ the set of clauses in which x_j is a positive literal and by $\text{neg}(x_j)$ the set of clauses in which x_j is a negative literal.

The DTD d accepts trees that correspond to truth assigments for $\{x_1, \ldots, x_n\}$, together with the clauses in φ that they satisfy. More precisely, d consists of the following rules:

$$\begin{array}{lll} r & \to (x_1 + \bar{x}_1)(x_2 + \bar{x}_2) \cdots (x_n + \bar{x}_n) & \\ x_i & \to \prod_{c_j \in \text{pos}(x_i)} c_j & \text{for every } i \in \{1, \ldots, n\} \\ \bar{x}_i & \to \prod_{c_j \in \text{neg}(x_i)} c_j & \text{for every } i \in \{1, \ldots, n\} \end{array}$$

The pattern p tests if every c_i appears in the tree. For instance, if

$$\varphi = (x_1 \vee x_3 \vee \neg x_4) \wedge (x_2 \vee x_3 \vee x_4) \wedge (\neg x_1 \vee \neg x_3 \vee \neg x_4) \tag{52.1}$$

then d has nine rules, namely

$$\begin{array}{llll} r \to (x_1 + \bar{x}_1)(x_2 + \bar{x}_2)(x_2 + \bar{x}_3) & & & \\ x_1 \to c_1 & \bar{x}_1 \to c_3 & x_2 \to c_2 & \bar{x}_2 \to \varepsilon \\ x_3 \to c_1 c_2 & \bar{x}_3 \to c_3 & x_4 \to c_2 & \bar{x}_4 \to c_1 c_3 \end{array}$$

The tree pattern p for φ is depicted in Figure 52.1a. Figure 52.1b contains a tree accepted by d. It is easy to prove that φ is satisfiable if and only if p is consistent w.r.t. D. □

We now turn to a different problem, namely that of containment of TPQs with schema information.

PROBLEM: TPQ-Containment with EDTD Information
INPUT: a Boolean TPQs p_1, p_2 and an EDTD D
OUTPUT: yes if $T \models p_1$ implies $T \models p_2$ for every tree $T \in L(D)$ and no otherwise

We will show that this problem is EXPTIME-complete and that the computational difficulty already lies in a special case, namely testing if a Boolean TPQ can be matched in *every* tree of a given EDTD.

PROBLEM: TPQ-Validity with EDTD Information
INPUT: a Boolean TPQ p and an EDTD D
OUTPUT: yes if $T \models p$ for every tree $T \in L(D)$ and no otherwise

Lemma 52.3. *TPQ-Validity with EDTD Information is* EXPTIME-*hard.*

Proof. The proof is by a reduction from two-player corridor tiling, which is EXPTIME-complete, see Appendix C. Let $\mathcal{T} = (T_C, T_S, u^{\text{first}}, u^{\text{last}}, V, H, n)$ be a two-player tiling system. We construct a TPQ p and EDTD $D = (\Sigma, \Gamma, \rho, S, \mu)$ such that p is valid w.r.t. D if and only if Constructor does not have a winning strategy. Intuitively, the pattern p will be valid w.r.t. D if there does not exist a winning strategy tree for Constructor. **(Wim)**[3] Intuitively, the EDTD will accept (encodings of) strategy trees that are well-formed and satisfy the horizontal constraints and the pattern will match if these trees violate a vertical constraint.

Assume that $|T_C| + |T_S| = k$ and fix an ordering $t_1, \ldots, t_k$ on $T_C \cup T_S$. We encode each tile t_i by a word $\text{enc}(t_i)$ of length $2k$, using the symbols $\{0, 1, 2\}$. Here, $\text{enc}(t_i)$ is of the form $b_1 \cdots b_k c_1 \cdots c_k$, where $b_i = 1$ and $b_j = 0$ for every $j \in [1, k] - \{i\}$. Furthermore, $c_j = 2$ if $(t_i, t_j) \notin V$ and $c_j = 0$ otherwise. As such, if we have a concatenation $\text{enc}(t)\ \text{enc}(s)$ of two tile encodings, then $(t, s) \notin V$ if and only if the symbol 1 occurs precisely k positions after the symbol 2.

Wim:
Explain assumptions on strategy tree somewhere. It must be proved in the appendix that we can do these. Assumptions: (1) Each time the spoiler has exactly two options. (2) Last tile is at top right.

Consider the EDTD D with the following rules.

- $S = \{u_1^{\text{first}}\}$;
- for every tile type t and $i \in [1, 2k-1]$, we have the rule $t_i \to t_{i+1}$;
- for every tile type $t \in T_C$, we have the rule $t_{2k} \to \sum_{(t,s)\in H} \sum_{(t,s')\in H} s_1 s'_1$;

[3] **Wim:** Introduce two-player system in Appendix.

- for every tile type $t \in T_S$, we have the rule $t \to \sum_{(t,s)\in H} s$; and
- $u_{2k}^{\text{last}} \to \varepsilon$.

The tree pattern p simply tests if the tree has a node with label 2 which is $2k(n-1)+k$ levels above some node with label 1. It is depicted in Figure **??**. This concludes the reduction.

Wim:
Give correctness sketch?

□

Lemma 52.4. *Given a tree pattern p, there exist exponential-size NTAs N_p and $N_{\neg p}$ such that*

1. $L(N_p) = \{T \mid T \models p\}$ *and*
2. $L(N_{\neg p}) = \{T \mid T \not\models p\}$.

Proof. We first prove (a). Let $p = (V, \ldots)$ It is easiest to think of N_p as an automaton that reads trees in a bottom-up fashion. The intuition behind N_p is closely related to the evaluation algorithm for TPQs in Theorem 47.3. The automaton uses the state set $2^V \times 2^V$ and the intention is that it visits each node u in the state $(\mathsf{match}(u), \mathsf{match}'(u))$ with the same meaning as in Theorem 47.3's proof. **(Wim)**[4] Finally, the accepting states of N_p are $\{(X,Y) \mid \text{root}(p) \in X\}$.

Concerning (b), the automaton $N_{\neg p}$ is identical to N_p, except that it uses $\{(X,Y) \mid \text{root}(p) \notin X\}$ as its set of accepting states. □

Wim:
Say something about correctness.

Lemma 52.5. *Given tree patterns p_1 and p_2 and EDTD D, it can be checked in* ExpTime *if $T \models p_1$ implies $T \models p_2$ for every $T \in L(D)$.*

Proof. Notice that $T \models p_1$ implies $T \models p_2$ for every $T \in L(D)$ if and only if $L(N_{p_1}) \cap L(N_{\neg p_2}) \cap L(D)$ is not empty. By Theorems 50.3 and 51.6, we can construct an NTA N for $L(N_{p_1}) \cap L(N_{\neg p_2}) \cap L(D)$ in time $2^{O(n^c)}$ for some constant c. By Theorem 50.4, we can test in time polynomial in $\|N\|$ if $L(N)$ is non-empty. Altogether, this is an ExpTime algorithm.

The following Theorem is immediate from Lemmas 52.3 and 52.5.

Theorem 52.6. *TPQ-*Containment *with EDTD* Information *and TPQ-*Validity *with EDTD* Information *are* ExpTime*-complete.*

[4] **Wim:** Give example of a transition?

53

Static Analysis on Data Trees

Leonid

evaluation

use automata + keys + foreign keys

coding with linear constraints

TA and constraints, examples: keys/foreign keys, add undecidability, FO2, undecidability FO3?

Comments and Exercises for Part VIII

Exercise 53.1. We note that the complexity in Theorem 47.3 can be improved to $O(|p||T|)$. *Hint:* Let the input to TPQ Evaluation be p, T, and $(u_1, \ldots, u_k)$. Let $(v_1, \ldots, v_k)$ be the output nodes of p. Then the overall idea of the direct algorithm is the same as in Theorem 47.3, but it treats the output nodes of p differently. More precisely, it only allows an output node v_i to be in a set $\mathsf{match}(u)$ if $u = u_i$.

Exercise 53.2. Consider *conjunctive queries over trees*, i.e., conjunctive queries evaluated over tree structures, where the built-in relations are *child* and *descendant*. What is the complexity of their evaluation problem?

Exercise 53.3. Show that the output of ComputeCore(p) (Algorithm 11 on page 317) is not unique up to isomorphism.

Exercise 53.4. Prove Theorem 48.10.

Wim:
I think that this is by structural induction on the patterns. By Kimelfeld/Sagiv, this is essentially the result that says that minimality equals nonredundancy for stable patterns. (Holds both for Boolean and non-Boolean.) Leave the proof as an exercise.

Exercise 53.5. Show that the pattern p in Figure 48.2 has no equivalent proper subpattern.

Exercise 53.6. What is the precise complexity of evaluation for downward XPath? (Open problem.)

Exercise 53.7. The cores of XML Schema and Relax NG, seen as grammars, are a bit more involved than what we described in Chapter 51, even if we allow arbitrary regular expressions. For XML Schema, it can be more accurately defined as tuples (Σ, τ, R, S) where $S \subseteq \Sigma \cup (\Sigma \times \tau)$ and R maps $\Sigma \cup \tau$ to regular expressions over $\Sigma \cup (\Sigma \times \tau)$. For Relax NG, R maps $\Sigma \cup \tau$ to regular expressions over $\Sigma \cup \tau \cup (\Sigma \times \tau)$. **(Wim)**[1] Prove that this is equally expressive.

[1] **Wim:** Make this more precise.

Exercise 53.8. Prove that every language definable by an stEDTD is a regular language closed under ancestor-guarded subtree exchange.

Hint: Use a partial function $f : \Sigma^+ \to \Gamma$ that maps each "ancestor-string" to the relevant type, i.e.,

- $f(a) = t$, where $t \in S$ is unique such that $\mu(t) = a$ and
- $f(wa) = t$, where $f(w) = t'$ and t is the unique type occurring in $\rho(t')$ with $\mu(t) = a$.

Some ideas for exercises:

- Can we postpone canonical models to here?
- Prove that minimal patterns are not unique in a strong sense?
- Prove that you cannot define "trees for which the deepest node is on even depth" with a tree automaton.
- Prove that you cannot define "the tree has a unique b-labeled leaf" with XSD (single-type EDTD).
- Turn a regular expression into a deterministic one?

Exercise 53.9. Prove that, given a TPQ p and EDTD D, you can construct in polynomial time a Boolean TPQ p_b and EDTD D_b such that the following are equivalent:

(1) There exists a tree $T \in L(D)$ such that $p(T)$ is not empty.
(2) There exists a tree $T \in L(D_b)$ such that $T \models p_b$.

Hint: add children with unique labels to the output nodes of p.

Exercise 53.10. When reducing a DTD, it is important that steps 1 and 2 in the proof of Lemma 52.1 are performed in that order. Show that, if the order is reversed, the result is not always a reduced DTD.

Exercise 53.11. Prove that the NP-hardness of Theorem 52.2 also holds for DTDs and tree patterns without descendant edges or wildcards. *Hint: Try to "flatten" the trees defined by the DTD.*

BLA

The TATA book.

Wim:
In the bib part: Thatcher JCSS shows EDTD (extended CFG) equals tree automata

The specifications for DTD and XML Schema require content models to be *deterministic*, see [**?**, Appendix E] and [**?**, Appendix J]. Formally, this constraint can be abstracted as follows. Let r be a regular expression. Let $\bar{r}$ stand for the regular expression obtained from r by replacing, for every integer i and alphabet symbol a, the i-th occurrence of a in r by a_i (counting occurrences from left to right). For example, for $r = b^*a(b^*a)^*$ we have $\bar{r} = b_1^*a_1(b_2^*a_2)^*$.

Definition 53.1. *A regular expression r is* deterministic *if there are no words wa_iv and wa_jv' in $L(\bar{r})$ such that $a \in \Sigma$ and $i \neq j$.*

Notice that the expression $(a+b)^*a$ is not deterministic since both words a_2 and a_1a_2 are in $L((a_1+b_1)^*a_2)$. The equivalent expression $b^*a(b^*a)^*$ is deterministic. Brüggemann-Klein and Wood showed that not every regular expression is equivalent to a deterministic one and, therefore, that the set of deterministic regular expressions are strictly less expressive than the regular expressions. The canonical iexample for a language that is not DRE-definable is $(a+b)^*b(a+b)$ [**?**]. Czerwinski et al. showed that it is PSpace-complete in general to decide if the language of a given regular expression is definable by a deterministic one.

- The TATA book
-

Part IX

Graph-Structured Data

Things to Check and Keep in Mind

Wim:
This is just a list of notation- and other things that should be in sync with the rest of the book. Will not be a real chapter. Just for us.

Notation / Terminology

- Graphs (for RPQc etc.) versus data graphs (for data value equality tests).

  Graphs. Edge-labeled directed graphs. Finite set of edge labels. I'd like to explain that they have a natural correspondence to ordinary DBs, where the edge labels correspond to relation names and the node IDs to the data values. This makes clear in Fig 52.1 why the labels are as they are.

  Data graphs. Here we introduce an extra layer. Notes get labels from an infinite alphabet. We need to explain why b/c in the previous correspondence to ordinary DBs, the node IDs were actually the data values. But in data graphs we want to be more flexible and "data value equality" should not always correspond to node identity.

  If I can do it like this, then I think all the results in the graph chapter make sense. I think that these are the simplest definitions with which we can present everything we want (up to Chap 54).

- We kind of need to define $q(G)$ for an RPQ q and graph G, since this is how we define things in the book. On the other hand, it's also good to have this notation $[\![L]\!]_G$, which is the same if $q = L(x, y)$. It could also be a good idea to give $[\![L]\!]_G$ a name. (Perhaps a bit more descriptive than "semantics of q on G"; because we will define the *semantics of queries* anyway. Currently I'm using *connectedness relation* – but it's easy to change.

- When we define a mapping η, which associates an element of V to each free variable of φ, we also immediately use the notation $\eta(x, y, z)$ and such. (This also appears in the def of semantics of FO.) We should mention somewhere globally in the book that we mean $(\eta(x), \eta(y), \eta(z))$ by this.
- Property graphs. Should we talk about *values* or *data values*? (In the relational part, it may be clearer that values refers to the actual data. Here we have this mix between data and structure. Moreover, if you take the natural correspondence between graph DBs and relational DBs, then the node IDs actually become the relational data values.)
- Property graphs. I'm using *data path* to refer to a path in a data graph. Their labels are called *data words*. I know that this is terminology overloading, but it's really essentially the same thing. Other terminology was causing me trouble.

Other Things to Check

- Note to self. In the data model chapter, discuss the difference with Chapter 46. Here we can have multiple edges between the same nodes; with different labels. We need this if we want to compute completions of graphs.
- For defining FO-RPQs, we can't just say "Take FO over some schema" because schemas are always a finite set of relations. Here I need countably many (one for every regular word language).
- The same remark "We don't use constants?" as in the beginning of the book applies to FO-RPQ queries.

To Keep in Mind

-

54

Data Model and Queries

Consider the graph in Figure 54.1, which contains some information about two persons (Aretha and Saul), who live in North America. We will use this example to illustrate some basic concepts about graph databases.

Data Model

Throughout the chapters in Part IX, we use a finite set Σ of *labels*. Arguably, the simplest data model for a graph database is an edge-labeled directed graph. As such, a graph database can be seen as a tuple $G = (V, E)$, where V is a finite, nonempty set of *nodes*, $E \subseteq V \times \Sigma \times V$ is a set

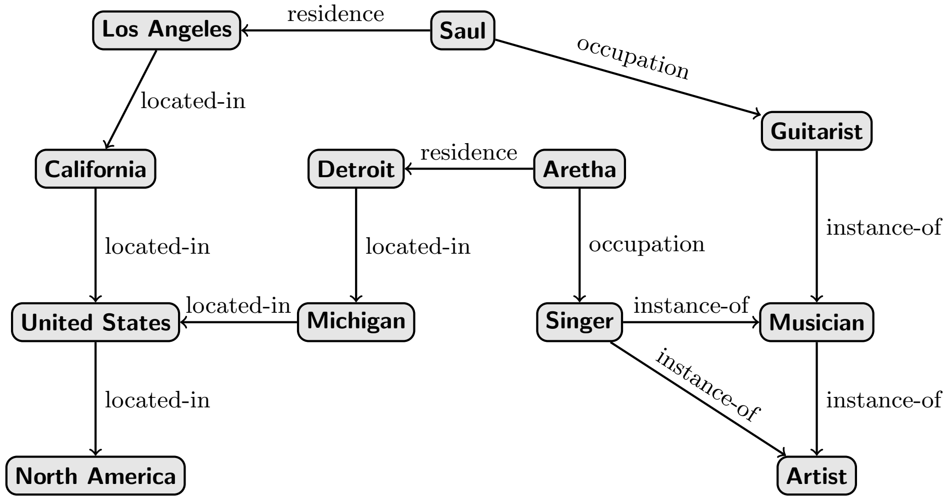


Fig. 54.1: A tiny graph database about artists in North America

of directed and labeled *edges*. A *(directed) path* is a sequence of edges $\pi = (v_0, a_1, v_1)(v_1, a_2, v_2) \ldots (v_{n-1}, a_n, v_n)$.**(Wim)**[1] We say that π is *from* v_0 *to* v_n and has length n. The *label* of path π is $\text{lab}(\pi) = a_1 \cdots a_n$.

Example 54.1. Our definition of graph database can capture the node- and edge structure of the graph in Figure 54.1, and the edge labels. For instance, we could take $V = \{\text{Aretha, Saul, } \ldots\}$ and $E = \{(\text{Aretha, occupation, Singer}), \ldots\}$.

When we consider Figure 54.1, we see that there is a natural correspondence between the edge labels in the graph and the relation names we have used earlier in the book. In a relational database, we would probably model the information with relations such as Residence(Person,City). Indeed, graph databases and relational databases are closely related in the sense that a graph database $G = (V, E)$ with $\Sigma' = \{a \mid (u, a, v) \in E\}$ is isomorphic to a relational database $D = (V, R_{a_1}, \cdots, R_{a_k})$, where $R_{a_i} = \{(u, v) \mid (u, a_i, v) \in E\}$ for every $i \in [1, k]$. Notice that V plays a similar role as the *active domain* in relational databases. Due to this correspondence between graph and relational databases, a wide number of query languages that we have considered before (and their complexity results) carry over to graph databases.

One may believe that this correspondence is also the reason why we choose Σ to be a finite set. However, whether Σ is finite or infinite is inconsequential for Chapters 55 and 56. We just choose the simplest definition we need to prove the results in these chapters. We will introduce infinite sets of labels in Chapter 57.

Regular Path Queries

An important difference between graph- and relational databases is that there is a larger focus on *navigation* and *connections* in the data when querying graph databases. *Regular path queries* can express complex and long-distance connections between nodes, using regular word languages to specify constraints on paths. More precisely, a *regular path query (RPQ)* is an expression of the form

$$L(x, y)$$

where L is a regular word language over the set Σ and x, y are variables. Unless otherwise mentioned, we will use regular expressions to represent L. This means that, on a Turing machine, we encode the query $q = L(x, y)$ as a triple (r, x, y) where $L = L(r)$. As a notational convention, we will from now on use r_q to denote the regular expression defining the regular language in the RPQ q.

For an RPQ $L(x, y)$ and a graph $G = (V, E)$, we can define a *reachability relation* of $L(x, y)$ on G, which is defined as

[1] **Wim:** We do it like this b/c of trails.

$$\llbracket L \rrbracket_G = \{(u,v) \in V \times V \mid \text{ there exists a directed path } \pi \text{ from } u \text{ to } v \text{ in } G \text{ with } \mathrm{lab}(\pi) \in L\} \ .$$

The *output* of an RPQ $q = L(x,y)$ on G is simply the reachability relation of q on G, that is,

$$q(G) = \llbracket L \rrbracket_G \ .$$

Since the reachability relations of RPQs are binary relations, it is natural to treat RPQs similarly to binary relation symbols in queries. Doing so leads to the idea of *conjunctive regular path queries* and *first-order regular path queries*, which we define next.

Conjunctive Regular Path Queries

Let $\bar{x} = (x_1, \ldots, x_n)$ and $\bar{y} = (y_1, \ldots, y_m)$ be tuples of node variables such that $\{x_1, \ldots, x_n\}$ and $\{y_1, \ldots, y_m\}$ are disjoint. A *conjunctive regular path query (CRPQ)* is an expression of the form

$$\varphi(\bar{x}) \coloneq L_1(z_1, z_1'), \ldots, L_k(z_k, z_k')$$

where

(a) $L_i(z_i, z_i')$ is a regular path query for each $i \in [1,n]$ and
(b) $\{z_i, z_1', \ldots, z_n, z_n'\} = \{x_1, \ldots, x_n\} \cup \{y_1, \ldots, y_m\}$.

Notice that the variables $z_1, z_1', \ldots, z_n, z_n'$ are not necessarily distinct. We refer to the subexpressions $L_i(z_i, z_i')$ as the *atoms* of φ. **(Wim)**[2]

A *homomorphism* from the CRPQ $\varphi(\bar{x})$ to a graph database G is a mapping $h : \bar{x} \cup \bar{y} \to V$ such that $(h(z_i), h(z_i')) \in \llbracket L_i \rrbracket_G$ for every $i = 1, \ldots, k$. The *output* of a CRPQ $q = \varphi(\bar{x})$ on G is defined as

$$q(G) = \{h(\bar{x}) \mid h \text{ is a homomorphism from } \varphi(\bar{x}) \text{ on } G\} \ .$$

Example 54.2. Consider $L_{\mathsf{residence}} = L(\text{residence}(\text{located-in})^*)$ and $L_{\mathsf{occupation}} = L(\text{occupation}(\text{instance-of})^*)$. The CRPQ

$$\varphi(x) = L_{\mathsf{occupation}}(x, \text{Artist}), L_{\mathsf{residence}}(x, \text{United States})$$

returns arists who live in the United States on the graph database of Figure 54.1. **(Wim)**[3]

[2] **Wim:** I later use atoms of the form $x = y$. This is actually just an alternative notation for $\varepsilon(x,y)$.

[3] **Wim:** We actually don't have constants at the moment. This makes it not easy to express "artists who live in the US". We should decide if we want to incorporate constants in CRPQs or not. If we want to give convincing examples, we probably should.

First-Order Regular Path Queries

For defining *first-order regular path queries (FO-RPQs)*, we first inductively define FO-RPQ expressions.

- If x and y are variables, then $x = y$ is an atomic FO-RPQ expression.
- If $L(x, y)$ is an RPQ, then it is an atomic FO-RPQ expression.
- If φ_1 and φ_2 are FO-RPQ expressions, then $(\varphi_1 \wedge \varphi_2)$, $(\varphi_1 \vee \varphi_2)$, and $(\neg\varphi_1)$ are FO-RPQ expressions.
- If φ is an FO-RPQ expression and x is a variable, then $(\exists x\, \varphi)$ and $(\forall x\, \varphi)$ are FO-RPQ expressions.

Similar to FO, we use the shorthand $\varphi \to \psi$ for $\neg\varphi \vee \psi$ and $\varphi \leftrightarrow \psi$ for $(\varphi \to \psi) \wedge (\psi \to \varphi)$. Also, to reduce notational clutter, we often write $\exists\bar{y}\, \varphi$ for $\exists y_1 \exists y_2 \ldots \exists y_m\, \varphi$, where $\bar{y} = (y_1, \ldots, y_m)$, and likewise for universal quantifiers $\forall\bar{y}$. Free variables for FO-RPQ expressions are definite analogously to those of FO formulas.

The semantics of an FO-RPQ expression φ is defined similarly to that of an FO formula. Formally, we use an *assignment* η, which associates an element of V to each free variable of φ. If the formula φ is satisfied in G under assignment η, we write $(G, \eta) \models \varphi$. This is now defined inductively:

- If φ is $x = y$, then $(G, \eta) \models \varphi$ iff $\eta(x) = \eta(y)$.
- If φ is $L(x, y)$, then $(G, \eta) \models \varphi$ iff $\eta(x, y) \in [\![L]\!]_G$.
- $(G, \eta) \models \neg\varphi$ iff $(G, \eta) \models \varphi$ does not hold.
- If $\varphi = \varphi_1 \wedge \varphi_2$, then $(G, \eta) \models \varphi$ iff $(G, \eta) \models \varphi_1$ and $(G, \eta) \models \varphi_2$.
- If $\varphi = \varphi_1 \vee \varphi_2$, then $(G, \eta) \models \varphi$ iff $(G, \eta) \models \varphi_1$ or $(G, \eta) \models \varphi_2$.
- If φ is $\exists x\, \psi$, then $(G, \eta) \models \varphi$ iff $(G, \eta[v/x]) \models \psi$ for some $v \in V = \mathrm{Dom}(G)$; here $\eta[v/x]$ is the assignment that modifies η by setting $\eta(x) = v$.
- If φ is $\forall x\, \psi$, then $(G, \eta) \models \varphi$ iff $(G, \eta[v/x]) \models \psi$ for all $v \in V = \mathrm{Dom}(G)$.

An FO-RPQ expression φ without free variables is called a *sentence*. For such a sentence φ it is either the case that $(G, \eta) \models \varphi$ for *every* assignment η, or $(G, \eta) \models \varphi$ for *no* assignment η. If the former is the case, we simply write $G \models \varphi$.

An *FO-RPQ query* is an expression of the form $\varphi(\bar{x})$, where φ is an FO-RPQ expression and $\bar{x}$ is *some* tuple that mentions all free variables in φ (perhaps with repetitions). Given a tuple $\bar{v}$ of nodes from a database G, we use notation $G \models \varphi(\bar{v})$ to indicate that there exists an assignment η such that $\eta(\bar{x}) = \bar{v}$ and $(G, \eta) \models \varphi$. Moreover, assuming that the length of $\bar{v}$ is k, the *output* of φ on G is defined as

$$\varphi(G) \;=\; \{\bar{v} \mid \bar{v} \in V^k \text{ and } D \models \varphi(\bar{v})\}\,.$$

Notice that $\varphi(\bar{x})$ defines a k-ary query. To emphasize that these expressions define database queries, we shall often use notation $q(\bar{x})$ for them.

Example 54.3.

Wim:
If we can compare with constants, we could take Artists who don't live in LA.

Two-Way Navigation

RPQs are often extended with *two-way navigation*. The idea is that, if an edge (u, v) is labeled with a, then one can use the symbol a^- to navigate from v to u. Formally, we use a set $\Sigma^- = \{a^- \mid a \in \Sigma\}$, which we assume to be disjoint from Σ, and we define $\Sigma^\pm = \Sigma \uplus \Sigma^-$.

A *two-way regular path query (2RPQ)* is an expression of the form

$$L(x, y)$$

where L is a regular word language over the set $\Sigma^\pm$ and x, y are variables.

In order to define the semantics of 2RPQs, we use *bidirectional paths*, **(Wim)**[4] which are sequences $\pi = (v_0, \ell_1, v_1)(v_1, \ell_2, v_2) \cdots (v_{n-1}, \ell_n, v_n)$ such that, for each $i \in [1, n]$ either

- $(v_{i-1}, a, v_i) \in E$ and $\ell_i = a$ or
- $(v_i, a, v_{i-1}) \in E$ and $\ell_i = a^-$.

Notice that every directed path is a bidirectional path. We say that π is *from* v_0 *to* v_n and has length n. The *label* of path π is $\text{lab}(\pi) = \ell_1 \cdots \ell_n$.

If $L(x, y)$ is a 2RPQ, *reachability relation of* $L(x, y)$ *on* G, denoted $[\![L]\!]_G$ is the set of pairs $(u, v) \in V \times V$ such that there exists a bidirectional path π from u to v with $\text{lab}(\pi) \in L$.

Conjunctive two-way regular path queries (C2RPQs) are obtained from 2RPQs analogously as CRPQs from RPQs, that is, conjunctive two-way regular path query is an expression of the form

$$\varphi(\bar{x}) :\!-\ L_1(z_1, z_1'), \ldots, L_k(z_k, z_k')$$

where

(a) $L_i(z_i, z_i')$ is a two-way regular path query for each $i \in [1, n]$ and
(b) $\{z_i, z_1', \ldots, z_n, z_n'\} = \{x_1, \ldots, x_n\} \cup \{y_1, \ldots, y_m\}$.

[4] **Wim:** Doing it like this to avoid trouble with simple paths / trails, which doesn't work if I use graph completions here.

The *output* of a 2RPQ $\varphi(\bar{x})$ on G, is defines analogously as for CRPQs. A *homomorphism* from $\varphi(\bar{x})$ to G is a mapping $h : \bar{x} \cup \bar{y} \to V$ such that $(h(z_i), h(z'_i)) \in [\![L_i]\!]_G$ for every $i = 1, \ldots, k$. The *output* of $q = \varphi(\bar{x})$ on G is defined as

$$q(G) = \{h(\bar{x}) \mid h \text{ is a homomorphism from } \varphi(\bar{x}) \text{ on } G\} .$$

Likewise, the definition of *first-order two-way regular path queries (FO-2RPQs)* is completely analogous to that of FO-RPQs. The only difference is that FO-2RPQs can use 2RPQs instead of just RPQs as atomic expressions.

Example 54.4.

Wim:
If we have comparison with constants we could take occupations of people who live in California. Or countries where no artists are living.

Simple Paths and Trails

A (bidirectional) path $\pi = (v_0, a_1, v_1)(v_1, a_2, v_2) \cdots (v_{n-1}, a_n, v_n)$ is *simple* if all nodes $v_0 v_1 \cdots v_n$ are different. **(Wim)**[5] It is a *trail* if all edges used in π are different. Formally, the *base edge* of a tuple $e = (u, \ell, v)$ with $\ell \in \Sigma^{\pm}$ is denoted by base(e) and defined as

$$\text{base}(e) = \begin{cases} (u, a, v) & \text{if } \ell = a \in \Sigma \\ (v, a, u) & \text{if } \ell = a^- \in \Sigma^- \end{cases}$$

Finally, a bidirectional path $\pi = e_1 \cdots e_n$ is a trail if all edges base$(e_1), \ldots,$ base(e_n) are different.

The *reachability relation of 2RPQ $L(x, y)$ on G under simple path semantics*, denoted $[\![L]\!]^{\mathsf{s}}_G$, is the set of pairs $(u, v) \in V \times V$ such that there exists a bidirectional simple path π from u to v with lab$(\pi) \in L$. Analogously, the *reachability relation of $L(x, y)$ on G under trail semantics*, denoted $[\![L]\!]^{\mathsf{t}}_G$, is the set of pairs $(u, v) \in V \times V$ such that there exists a bidirectional trail π from u to v with lab$(\pi) \in L$. The *answer of $q = L(x, y)$ on G under simple path (resp., trail) semantics* is its reachability relation under simple path (resp., trail) semantics.

It is easy to define simple path or trail semantics for C2RPQs and FO-2RPQs. Indeed, the definition simply exchanges the ordinary reachability relations $[\![L]\!]_G$ with $[\![L]\!]^{\mathsf{s}}_G$ or $[\![L]\!]^{\mathsf{t}}_G$, respectively.

[5] **Wim:** This means that a simple cycle is not a simple path, which is OK as far as I'm concerned.

55

Graph Query Evaluation

In this chapter we study fundamental query evaluation problems on graph databases. To be compatible with our definition of query evaluation ($\mathcal{L}$-Evaluation) in Chapter 2, we will take the graph $G = (V, E)$ as the database D and take $\mathrm{Dom}(D) = V$. We will stick to the standard notation in the graph database literature though, and denote graph databases as $G = (V, E)$.

The most basic problem in this chapter is RPQ-Evaluation. Given a graph database G, an RPQ q, and a pair $\bar{a} = (u, v) \in V \times V$, it asks if $(u, v) \in q(G)$. Recall from Chapter 54 that an RPQ is always given in the input as a regular expression r, which is why we measure the complexity of RPQ-evaluation in terms of $\|G\|$ and $\|r\|$.

Theorem 55.1. *The problems RPQ-*Evaluation *and 2RPQ-*Evaluation *are in time $O(\|r\|\|G\|)$.*

Proof. We first explain the complexity for RPQs. Let N be an ε-NFA for $L(r)$, which can be obtained in time $O(\|r\|)$ by Appendix D. **(Wim)**[1] Given a pair $\bar{a} = (u, v) \in V \times V$, we need to decide if there exists a directed path π from u to v with $\mathrm{lab}(\pi) \in L(N)$, which can be done using a construction on D and N analogous to the *product construction* of finite automata. More precisely, let $N = (Q, \Sigma, \delta, I, F)$. Consider the graph $P = (V \times Q, E_P)$, where

$$E_P = \{((v_1, q_1), (v_2, q_2)) \mid (v_1, a, v_2) \in E \text{ and } q_2 \in \delta(q_1, a)\} \\ \uplus \{((v, q_1), (v, q_2)) \mid q_2 \in \delta(q_1, \varepsilon)\} \ .$$

Then, $(u, v) \in [\![L(r)]\!]_G$ if and only if (v, f) is reachable from (u, i) in P for some $f \in F$ and $i \in I$. The latter can be tested in time $O(\|r\|\|G\|)$ using

[1] **Wim:** Precise reference. Is this also possible if we don't allow epsilons? (ε-elimination may take quadratic time. Take the expression $\Sigma\Sigma$. Glushkov automaton for this expression is quadratic.) Hmm not sure if $a_1?a_2?a_3? \cdots a_n?$ even allows a linear-size NFA.

standard algorithms. **(Wim)**[2] Notice that we only need to test reachability and that P does not need to be explicitly constructed.

For 2RPQs the approach is the same, but we need to define P differently, taking bidirectional paths into account. More precisely, we define

$$\begin{aligned} E_P = \{((v_1,q_1),(v_2,q_2)) \mid (v_1,a,v_2) \in E \text{ and } q_2 \in \delta(q_1,a)\} \\ \uplus \{((v_1,q_1),(v_2,q_2)) \mid (v_2,a,v_1) \in E \text{ and } q_2 \in \delta(q_1,a^-)\} \\ \uplus \{((v,q_1),(v,q_2)) \mid q_2 \in \delta(q_1,\varepsilon)\} \;. \end{aligned}$$

The rest of the proof is analogous. □

Theorem 55.2. *The problems* CRPQ-Evaluation *and* C2RPQ-*evaluation are* NP-*complete.*

Proof. The upper bound proofs follow from Theorem 55.1 and Theorem 14.1 and are analogous for CRPQs and C2RPQs. Indeed, if $\varphi(\bar{x}) \coloneq L_1(z_1,z_1'),\ldots,L_k(z_k,z_k')$ is the query, we can compute binary relations R_i containing precisely $[\![L_i]\!]_G$ for every $i \in [1,k]$ in time polynomial in $\|r_i\|$, where r_i is the regular expression defining L_i in the input. Then, a tuple $\bar{a}$ is an answer to $\varphi(\bar{x})$ on G if and only if it is an answer to the CQ $q(\bar{x}) \coloneq R_1(z_1,z_1'),\ldots,R_k(z_k,z_k')$. **(Wim)**[3] The lower bound is immediate from Theorem 14.1 since CRPQs generalize CQs. □

Theorem 55.3. *The problems* FO-RPQ-Evaluation *and* FO-2RPQ-Evaluation *are* PSpace-*complete.*

Proof. The upper bounds are proved similarly as in Theorem 55.2. The relations R_i can be computed in polynomial time, which reduces both FO-RPQ-Evaluation and FO-2RPQ-Evaluation to evaluation of FO queries. According to Theorem 7.1, these FO queries can be evaluated in PSpace, which concludes the upper bound proofs. The PSpace lower bounds are immediate from Theorem 7.1. □

Simple Paths and Trails

The following problem plays a central role in this section.

Problem:	Two Disjoint Paths
Input:	directed graph G and node s_1, t_1, s_2, t_2
Output:	yes if there exist paths π_1 from s_1 to t_1 and π_2 from s_2 to t_2 in G that have no nodes in common and no otherwise

[2] **Wim:** Make more precise?
[3] **Wim:** We don't have a macro for CQ?

Observe that it does not make a difference for the Two Disjoint Paths problem to ask whether π_1 and π_2 are simple paths or not. We will therefore assume from now on that Two Disjoint Paths asks for simple paths. We will use the following closely related problem to deal with trail semantics of RPQs.

Problem:	Two Disjoint Trails
Input:	directed graph G and node s_1, t_1, s_2, t_2
Output:	yes if there exist trails π_1 from s_1 to t_1 and π_2 from s_2 to t_2 in G that have no edges in common and no otherwise

The first problem is well known to be NP-complete. The same complexity can be easily obtained for the Two Disjoint Trails by using the so-called *split graph* construction (Exercise 58.1).

Theorem 55.4. *The problems RPQ-*Evaluation*, 2RPQ-*Evaluation*, CRPQ-*Evaluation*, and C2RPQ-*Evaluation *under simple path semantics are* NP*-complete.*

Proof. The NP upper bounds are immediate, since an algorithm can guess witnessing paths and homomorphisms in polynomial time for each of the problems. For the lower bound, we prove that it is NP-hard to test if a given pair of nodes (u, v) is in $[\![a^*ba^*]\!]^s_G$. This result implies that all the problems are indeed NP-hard. We reduce from the NP-complete Two Disjoint Paths problem, which is defined as follows. **(Wim)**[4] Given a directed graph $G_{tdp} = (V_{tdp}, E_{tdp})$ and nodes s_1, t_1, s_2, and t_2, are there simple paths from s_1 to t_1 and from s_2 to t_2 that do not have a node in common?

The reduction is almost trivial: if we consider the graph database $G = (V, E)$ with $V = V_{tdp}$ and $E = \{(u, a, v) \mid (u, v) \in E_{tdp}\} \cup \{(t_1, b, s_2)\}$, then G has a simple path from s_1 to t_2 labeled with a word from $L(a^*ba^*)$ if and only if there are simple, node disjoint paths from s_1 to t_1 and from s_2 to t_2 in G_{tdp}. $\square$

Proposition 55.5. *The problem RPQ-*Evaluation *under simple path semantics is already* NP*-hard in data complexity, namely for the RPQs with regular expressions (a^*ba^*) and $(aa)^*$.*

Proof. We already showed NP-hardness for (a^*ba^*) in the proof of Theorem 55.4. Hardness for $(aa)^*$ can also be obtained using a reduction from the Two Disjoint Paths problem. To this end, let $G_{tdp} = (V_{tdp}, E_{tdp})$ and s_1, t_1, s_2, and t_2 be an input for Two Disjoint Paths. We construct the graph database $G = (V, E)$ with $V = V_{tdp} \uplus V_E \uplus \{t\}$, where $V' = \{u_e \mid e \in E_{tdp}\}$ is a set of new nodes for every edge in E_{tdp} and t is yet another new node. The edges of G are defined as

[4] **Wim:** Add in bibliographic remarks: [9].

$$E = \{(u, a, u_e), (u_e, a, v) \mid u, v \in V \text{ and } e = (u, v) \in E_{tdp}\} \\ \uplus \{(t_1, a, s_2), (t_2, a, t)\}$$

Now there is a simple path from s_1 to t that matches $(aa)^*$ if and only if there are simple, node disjoint paths from s_1 to t_1 and from s_2 to t_2 in G_{tdp}. $\square$

Theorem 55.6. *The problems* FO-RPQ-Evaluation *and* FO-2RPQ-Evaluation *under simple path semantics are* PSPACE*-complete.*

Proof. The algorithm starts by guessing relations $R_L = [\![L]\!]^{\mathsf{s}}_G$ for every RPQ $L(x, y)$ occurring in the input. Notice that these relations are at most quadratically large and that, for every pair (u, v), it can be checked if it is in $[\![L]\!]^{\mathsf{s}}_G$ or not in PSPACE. After this phase, the algorithm has a polynomial size relational database at its disposal, on which an FO query needs to be evaluated. This is possible in PSPACE according to Theorem 7.1. $\square$

Theorem 55.7. *The problems* RPQ-Evaluation*,* 2RPQ-Evaluation*,* CRPQ-Evaluation*, and* C2RPQ-Evaluation *under trail semantics are* NP*-complete.*

Proof. The NP upper bounds are immediate, since witnessing paths and homomorphisms can be guessed in polynomial time for each of the problems. For the lower bound, we prove that it is NP-hard to test if a given pair of nodes (u, v) is in $[\![a^*ba^*]\!]^{t}_G$, this time by reducing from the NP-complete TWO DISJOINT TRAILS problem. In fact, the reduction is analogous than the one from TWO DISJOINT PATHS in Theorem 55.4. $\square$

Proposition 55.8. *The problem* RPQ*-evaluation under trail semantics is already* NP*-hard in data complexity, namely for the* RPQs *with regular expressions* (a^*ba^*) *and* $(aa)^*$*.*

Proof. We already showed NP-hardness for (a^*ba^*) in the proof of Theorem 55.7. Hardness for $(aa)^*$ can also be obtained using a reduction from the TWO DISJOINT TRAILS problem, and the reduction is analogous to the one in the proof of Proposition 55.5. $\square$

Theorem 55.9. *The problems* FO-RPQ-Evaluation *and* FO-2RPQ-Evaluation *under trail semantics are* PSPACE*-complete.*

Proof. This proof is analogous to the proof of Theorem 55.6.

56

Containment

We study containment, i.e., for two given queries $\varphi_1(\bar{x})$ and $\varphi_2(\bar{y})$ whether every answer returned by φ_1 is also returned by φ_2. Since this can only happen if φ_1 and φ_2 have the same arity, and since the semantics of queries does not change by renaming variables, we assume throughout the chapter that φ_1 and φ_2 have the same tuple of output variables, i.e., $\bar{x} = \bar{y}$.

Regular Path Queries

Theorem 56.1. *RPQ-Containment and 2RPQ-Containment are* PSPACE-*complete.*

Proof. Given two regular expressions r_1, r_2 that define RPQs q_1 and q_2, it is easy to see that $q_1 \subseteq q_2$ if and only if $L(r_1) \subseteq L(r_2)$. As the latter question is PSPACE-complete, the result for RPQs follows. Concerning 2RPQs, the argumentation is the same, but reduces the problem to containment of *two-way NFAs*. **(Wim)**[1] $\square$

Conjunctive Regular Path Queries

Definition 56.2 (Expansions and Canonical Graph Databases). *Let $\varphi(\bar{x})$:- $L_1(z_1, z'_1), \ldots, L_k(z_k, z'_k)$ be a CRPQ. A* CQ *$\mathcal{E}$ is an* expansion *of φ if there exist words $w_i \in L_i$ such that $\mathcal{E}$ can be obtained from φ by replacing each atom $L_i(z_i, z'_i)$*

- *with $(z_i = z'_i)$ if $w_i = \varepsilon$ and **(Wim)**[2]*

[1] **Wim:** How deeply do we want to explain this? Talk about this problem in the Appendix?

[2] **Wim:** Hmmm this requires equality in CQs. Do we have that? The definition will become messier if we don't.

- *with a conjunction*

$$R_{a_1}(z_i, \#_i^1), R_{a_2}(\#_i^1, \#_i^2), \ldots, R_{a_{k_i}}(\#_i^{k_i}, z_i')$$

if $w_i = a_1 \cdots a_{k_i} \neq \varepsilon$.

Here, we assume that all $\#_i^j$ *are new, pairwise distinct variables.*

We also associate a canonical graph database *to an expansion* $\mathcal{E}$*. To this end, we define two variables* x, y *of* $\mathcal{E}$ *to be equivalent, if* $x = y$ *is a conjunct in* $\mathcal{E}$*. By* $[x]$ *we denote the equivalence class of* x*. The canonical graph database of* $\mathcal{E}$ *is defined as* $G_{\mathcal{E}} = (V_{\mathcal{E}}, E_{\mathcal{E}})$*, where* $V_{\mathcal{E}} = \{[x] \mid x \text{ is a variable of } \mathcal{E}\}$ *and* $E_{\mathcal{E}} = \{([x], a, [y]) \mid R_a(x, y) \text{ is an atom of } \mathcal{E}\}$*. Finally, the canonical graph databases of* φ *are the canonical graph databases of the expansions of* φ*.*

Lemma 56.3. *The following are equivalent for CRPQs* $\varphi_1(\bar{x})$ *and* $\varphi_2(\bar{x})$*.*

(1) $\varphi_1 \subseteq \varphi_2$
(2) For every expansion $\mathcal{E}_1$ *of* φ_1 *there exists an expansion* $\mathcal{E}_2$ *of* φ_2 *such that* $(\mathcal{E}_2, \bar{x}) \to (\mathcal{E}_1, \bar{x})$*.* ***(Wim)***[3]
(3) For every expansion $\mathcal{E}_1$ *of* φ_1 *there exists a homomorphism* $h : (\varphi_2, \bar{x}) \to (G_{\mathcal{E}_1}, x)$*.* ***(Wim)***[4]

Proof. Proof as exercise?

Theorem 56.4. *CRPQ-*Containment *is* ExpSpace*-complete.*

Proof. We first prove the upper bound. To this end, let $\varphi_1(\bar{x})$ and $\varphi_2(\bar{x})$ be CRPQs. As mentioned in Lemma 56.3, we have that $\varphi_1 \subseteq \varphi_2$ if and only if for every expansion $\mathcal{E}_1$ of φ_1 there exists an expansion $\mathcal{E}_2$ of φ_2 such that $(\mathcal{E}_2, \bar{x}) \to (\mathcal{E}_1, \bar{x})$. We will reduce this condition to the language containment problem of NFAs, that is, we will define two NFAs A_1 and A_2 such that $L(A_1) \subseteq L(A_2)$ if and only if $\varphi_1 \subseteq \varphi_2$. Intuitively, A_1 will recognize (encodings of) expansions $\mathcal{E}_1$ of φ_1 and A_2 (encodings of) expansions $\mathcal{E}_1$ of φ_1 for which there exists an expansion $\mathcal{E}_2$ of φ_2 with $(\mathcal{E}_2, \bar{x}) \to (\mathcal{E}_1, \bar{x})$.

Warm-up. Assume that $\varphi_1(\bar{x}) :- L_1(z_1, z_1'), \ldots, L_n(z_n, z_n')$ and that $\mathcal{E}$ is an expansion of φ_1 for which we chose $w_i \neq \varepsilon$ for every $i \in [1, n]$.[5] Then $\mathcal{E}$ can be encoded as a word over the alphabet $\mathcal{A} := \Lambda \cup \mathcal{V}_1 \cup \{\#, \$\}$, where $\mathcal{V}_1$ is the set of variables of φ_1, as follows:

$$\$ z_1 w_1' z_1' \$ z_2 w_2' z_2' \$ \; \cdots \; \$ z_n w_n' z_n' \$ \;\in\; \mathcal{A}^* \tag{56.1}$$

where $w_i' = a_1 \# a_2 \# \cdots \# a_n$ if $w_i = a_1 \cdots a_n$.

[3] **Wim:** Notation $(\mathcal{E}_2, \bar{x}) \to (\mathcal{E}_1, \bar{x})$ from CQ chapter.
[4] **Wim:** Notation not yet defined.
[5] In the next step, we show how to get rid of this assumption.

One can construct in polynomial time an NFA A over the alphabet $\mathcal{A}$ recognizing the language of all such encodings of expansions of φ_1 (Exercise 58.2). Notice that each occurrence of a symbol z_i, z_i', or # corresponds to an occurrence of a variable in $\mathcal{E}_1$ and that this corresponcence is bijective.

Automaton 1. Every expansion of φ_1 can be encoded as a word over the alphabet $\mathcal{A}_1 = \Lambda \cup 2^{\mathcal{V}_1} \cup \{\#, \$\}$ as follows. If $\mathcal{E}_1$ is the expansion of φ_1 obtained by choosing the word w_i for each $i \in [1,n]$, then we can encode $\mathcal{E}_1$ as a word

$$\$Z_1 w_1' Z_1' \$ Z_2 w_2' Z_2' \$ \cdots \$ Z_n w_n' Z_n' \$ \ \in \ \mathcal{A}_1^* \tag{56.2}$$

where $Z_i, Z_i' \subseteq \mathcal{V}_1$ for every $i \in [1,n]$ and

(1) $w_i' = a_1 \# a_2 \# \cdots \# a_n$ if $w_i = a_1 \cdots a_n$,
(2) $z_i \in Z_i$ and $z_i' \in Z_i'$ for all $i \in [1,n]$,
(3) the set $\{Z_i, Z_i' \mid i \in [1,n]\}$ is a partition[6] of $\mathcal{V}_1$, and
(4) $Z_i = Z_i'$ if and only if $w_i = \varepsilon$.

We refer to such words as φ_1-*words*. One can construct in exponential time an NFA A_1 recognizing all φ_1-words (Exercise 58.2). Notice that we use the sets Z_i and Z_i' to deal with cases where variables of φ_1 must be mapped to the same node in the graph database, i.e., when $w_i = \varepsilon$.

Example 56.5. TODO. Show a query and a few encodings of expansions.

Automaton 2. To define automaton A_2, we first introduce yet another way of representing expansions $\mathcal{E}_1$ of φ_1, namely through *annotated* φ_1-*words*. The idea is that we additionally encode how a homomorphism could map φ_2 onto the canonical graph of $\mathcal{E}_1$. An *annotated* φ_1-*word* is a word of the form

$$W = \$(\ell_1, \gamma_1) \cdots (\ell_m, \gamma_m)\$$$

where $\$\ell_1 \cdots \ell_m\$$ is a φ_1-word and $\gamma_i \subseteq \mathcal{V}_2$ for all $i \in [1,m]$. Intuitively, W encodes an expansion $\mathcal{E}_1$ of φ_1 and the homomorphism $h : (\varphi_2, \bar{x}) \to (G_{\mathcal{E}_1}, \bar{x})$ **(Wim)**[7] that maps each variable in γ_i to the node ℓ_i in $G_{\mathcal{E}_1}$, for all $i \in [1,n]$. (If ℓ_i is not a node, then $\gamma_i = \emptyset$.)

An automaton A_2' can test if an annotated φ_1-word encodes $\mathcal{E}_1$ and h using the following three tests:

(1) For every set $\ell_i \subseteq \mathcal{V}_1$ containing an output variable x and every $\ell_j = \ell_i$, check that γ_j also contains x.
(2) If a variable $y \in \mathcal{V}_2$ is in γ_i, then either
    (2i) $\ell_i = \#$ and y only appears in γ_i, or

[6] Notice that this does not forbid that some sets among the Z_i, Z_i' are identical.
[7] **Wim:** Define notation.

(2ii) $\ell_i \subseteq \mathcal{V}_1$ and y appears precisely in the γ_j for which $\ell_i = \ell_j$.

(3) For every atom $L'_i(x'_i, y'_i)$ of φ_2, there exists a path from $h(x'_i)$ to $h(y'_i)$ in $G_{\mathcal{E}_1}$ that is labeled with some word in L'_i.

Condition (2) ensures that the encoded mapping h is well-defined. Conditions (1) and (2) can be tested with NFAs that can be constructed in exponential time. **(Wim)**[8] Condition (3) is the most challenging to check and requires that (2) already holds. We explain how to do it with a *two-way* NFA.

For checking if there exists a path from $h(x'_i)$ to $h(y'_i)$ in $G_{\mathcal{E}_1}$ that is labeled with some word in L'_i, a two-way NFA B_i can first find an encoding of the node $h(x'_i)$ in W. Due to condition (2), this is a symbol of the form (ℓ', γ') where $x'_i \in \gamma'$. When having reached the node $h(x'_i)$, the two-way NFA continues reading W to the right, while simulating an NFA N_i for L_i, until either

(a) N_i accepts at node $h(y'_i)$ or

(b) we meet a symbol of the form (ℓ, γ) with $\ell \subseteq \mathcal{V}_1$.

In case (a), the two-way NFA B_i accepts. In case (b), it pauses the simulation of N_i and traverses the string (backward or forward) to search for a pair (ℓ, δ) (which encodes the same node of $G_{\mathcal{E}_1}$ where we paused simulation), immediately to the right of a symbol \$. By (2ii), it can do this by guessing a variable $y \in \gamma$ and simply searching for a pair (ℓ'', γ'') where $y \in \gamma''$. It then continues simulation as before, until (a) or (b) happens. This simulation continues until N_i accepts at node $h(y'_i)$. This can be done with a polynomial-size two-way NFA B_i. A polynomial-size two-way NFA B for checking condition (3) can be obtained by executing the automata $B_1, \ldots, B_{n'}$ one after the other (where n' is the number of atoms of φ_2). This two-way NFA can be converted in exponential time into an equivalent NFA A'_2 (Appendix **(Wim)**[9]).

It remains to explain the automaton A_2, which accepts a φ_1-word $\$\ell_1 \cdots \ell_m\$$ if and only if there exists an annotated word $\$(\ell_1, \gamma_1) \cdots (\ell_m, \gamma_m)\$$ that is accepted by A'_2. Notice that A_2 can be obtained from A'_2 by simply replacing every symbol (ℓ, γ) with ℓ on each transition. Therefore, A_2 is equally large as A'_2, which is exponential size.

As such, we have shown that there exist exponential size NFAs A_1 and A_2 such that $A_1 \subseteq A_2$ if and only if $\varphi_1 \subseteq \varphi_2$. The ExpSpace decision algorithm for CRPQ-Containment is now obtained by applying the PSpace algorithm for containment on the exponential-size automata A_1 and A_2.

For the lower bound, we reduce from exponential corridor tiling. To this end, let $\mathcal{T} = (T, H, V, t_s, t_f, n)$ be a tiling system. Our plan is to define queries φ_1 and φ_2 such that $\varphi_1 \not\subseteq \varphi_2$ if and only if $\mathcal{T}$ does not have a valid tiling,

[8] **Wim:** TODO exercise. For (2) it makes sense to do a linear product of exponential-size NFAs.

[9] **Wim:** There's this standard Vardi construction that may be interesting to include in the Appendix if we want.

i.e., every tiling has some error. We encode tilings as words in the language $((0+1)^n T)^*$, that is, as sequences of pairs (N, t), where N is an n-bit binary number and t is a tile.

The first query is $\varphi_1(x_1, x_2) \coloneq L(x_1, x_2)$, with L defined by the regular expression $0^n t_s((0+1)^n T)^* 1^n t_f$, that is, it says that tilings are correctly encoded and that the first and final tile are correct.**(Wim)**[10] The second query is of the form

$$\varphi_2(x_1, x_2) \coloneq L_{\text{pre}}(x_1, y_1) \wedge (\bigwedge_{i=0}^{n} L_i(y_1, y_2)) \wedge L_{\text{suff}}(y_2, x_2) \ ,$$

where the regular expressions defining L_{pre} and L_{suff} are both $((0+1)^n T)^*$, and we explain the expressions r_i defining L_i next. The idea of L_{pre} and L_{suff} is that they match prefixes and suffixes of correctly encoded tilings.

Concerning the middle part, each expression r_i is of the form $r_i = r_H + r_{V_i} + r_c$, where r_H detects horizontal errors, the conjunction of the r_{V_i} detects vertical errors, and r_c detects errors concerning encodings of binary numbers. We define

$$r_H = \sum_{(t_1,t_2)\notin H} ((0+1^n)T)^*(0+1)^n t_1 (0+1)^n t_2 ((0+1)^n T)^* \ .$$

There are two kinds of expressions detecting vertical errors. First of all, we have

$$r_{V_0} = \sum_{(t_1,t_2)\notin V} (0+1)^n t_1 ((0+1)^n T)^* (0+1)^n t_2 \ ,$$

which serves for detecting vertical errors, but the tiles t_1 and t_2 can be arbitrarily far apart. In order to make sure that they are preceded by the same binary number, we use the expressions r_{V_i} for $i \in [1, n]$, which we define as

$$r_{V_i} = r_{V_i}^0 + r_{V_i}^1$$

where, taking $b \in \{0, 1\}$,

$$\begin{aligned} r_{V_i}^b = {} & (0+1)^{i-1} b (0+1)^{n-i} T \\ & ((0+1)^* b (0+1)^* T)^* \\ & \bar{b}^n T \\ & ((0+1)^* b (0+1)^* T)^* \\ & (0+1)^{i-1} b (0+1)^{n-i} T \end{aligned}$$

Here, we denote $1 - b$ with $\bar{b}$. The intuition behind these expressions is that $r_{V_i}^b$ first matches a binary number where the bit b occurs at position i. It then continues reading binary numbers until it finds b again at position i, but until then it can read the number $\bar{b}^n$ only once. As such, all r_{V_i} together match

[10] **Wim:** Assumes t_f in the top right corner.

words of the form Nt_1wNt_2, where N is an n-bit binary number and the subword 0^n or 1^n occurs exactly once in w. Therefore, t_1 and t_2 are tiles in column N in successive rows.

Finally, the expression r_c matches words of the form

$$((0+1)^nT)^*N_1t_1N_2t_2((0+1)^nT)^* ,$$

where N_1 and N_2 are non-consecutive binary numbers (modulo 2^n). We leave the construction of r_c as an exercise. **(Wim)**[11] □

First-Order Regular Path Queries

Theorem 56.6. *FO-RPQ*-Containment *and* *FO-2RPQ*-Containment *are undecidable.*

Proof. Immediate from the undecidability of FO on relational database, Theorem 8.1. □

[11] **Wim:** Or do we put it in?

57

Querying Property Graphs

Property graphs are a data model for graph databases that associates (sets of) data values to nodes and edges in the graph. These data values are assumed to come from an infinite domain, just like the set Const we use througout this book. Here, we assume a slightly simplified model in which edges are labeled by *labels* from a finite alphabet Σ and nodes can contain *(data) values.* **(Wim)**[1]

Definition 57.1 (Data graphs). *A* data graph *(over Σ and Const) is a triple $G = (V, E, \rho)$, where:*

- *V is a finite set of nodes;*
- *$E \subseteq V \times \Sigma \times V$ is a set of labeled edges; and*
- *$\rho : V \to \mathsf{Const}$ is a function that assigns a value to each node in V.*

Notice that we assume a single data value per node. This is just for notational simplicity and is not a restriction at all, since a node u with $k > 1$ attributes can be modeled by a node u with k outgoing edges, leading to k new nodes containing the data values.

Paths in data graphs are defined analogously to those in graph databases, but we refer to them as *data paths* to emphasise that the paths carry data values. Just as in graph databases, we will associate words with data paths, but here we also take data values into account. To this end, a *data word* is a sequence

$$d_0 a_1 d_1 a_2 \cdots a_n d_n \ ,$$

where $d_i \in \mathsf{Const}$ and $a_j \in \Sigma$ for all $i \in [0, n]$ and $j \in [1, n]$. We allow $n = 0$, in which case the data word consists of a single data value. By DW we denote the set of all data words. A *(directed) data path* in a data graph G (from v_0 to v_n) is a sequence of edges $\pi = (v_0, a_1, v_1)(v_1, a_2, v_2) \ldots (v_{n-1}, a_n, v_n)$. The *data word associated to* π is

[1] **Wim:** Check if this should be Const, Const, or Dom.

$$\mathsf{dlab}(\pi) = \rho(v_1)a_1\rho(v_2)a_2\rho(v_3)\ldots\rho(v_{n-1})a_{n-1}\rho(v_n)\ .$$

We will look into RPQ-like languages for data paths.

Definition 57.2 (Data Path Queries). *Let $L \subseteq \mathsf{DW}$, i.e., L is a language of data words. A* data path querys *q is an expression of the form $L(x, y)$. When evaluated on a data graph $G = (V, E, \rho)$, it returns the output*

$$q(G) = \{(u, v) \in V \times V \mid \text{ there exists a directed data path } \pi \text{ from } u \text{ to } v \text{ in } G \text{ with } \mathsf{dlab}(\pi) \in L\}\ .$$

A Note on the Complexity of Data Path Queries

It turns out that evaluation of data path queries can become complex quickly, which we illustrate next. Let L_{eq} be the language of data path labels **(Wim)**[2] that contain two equal data values. We will denote its complement, i.e., the set of data path labels containing pairwise different data values, by $\overline{L_{eq}}$.

Theorem 57.3. *The data complexity of evaluating the data path query $q = \overline{L_{eq}}(x, y)$ over data graphs is* NP*-complete.*

Proof. To see that the problem is in NP, observe that, given a data graph G and two nodes (s, t), we can test if $(s, t) \in q(G)$ by guessing a linear length data path from s to t and checking if all data values on the path are pairwise different.

The proof of the NP lower bound is by reducing from Two Disjoint Paths (see Chapter 55). Let (G, s_1, t_1, s_2, t_2) be an instance of the Two Disjoint Paths problem. First, we argue that we can assume that s_1, t_1, s_2, and t_2 are distinct. This is because we can always add two new nodes for each repeated node and connect them with all the nodes the repeated node was connected to, thus modifying our problem to have all source and target nodes different.

Assume that $G = (V, E)$ is a digraph and s_1, t_1, s_2, t_2 are four distinct nodes in G. Recall that our query is $q = \overline{L_{eq}}(x, y)$. Since q disregards edge labels we can take $\Sigma = \{a\}$. We will construct a data graph G' and two nodes $s, t \in G'$ such that $(s, t) \in q(G')$ if and only if there are two disjoint paths in G from s_1 to t_1 and from s_2 to t_2.

Let $V = \{v_1, \ldots, v_n\}$. The graph G' will contain two disjoint isomorphic copies of G (extended with data values and labels) connected by a single edge. We define the two isomorphic copies $G_1 = (V_1, E_1, \rho_1)$ and $G_2 = (V_2, E_2, \rho_2)$ by:

- $V_1 = \{v'_1, \ldots, v'_n\}$,

[2] **Wim:** I'm very tempted to use the term *data words* here. These things are essentially the same thing too... So why not call them the same?

- $V_2 = \{v_1'', \dots, v_n''\}$,
- $E_1 = \{(v_i', a, v_j') \mid (v_i, v_j) \in E\}$,
- $E_2 = \{(v_i'', a, v_j'') \mid (v_i, v_j) \in E\}$ and
- $\rho_1(v_i') = \rho_2(v_i'') = i$, for $i = 1 \dots n$,

and then let $G' = (V', E', \rho')$, where

- $V' = V_1 \cup V_2$,
- $E' = E_1 \cup E_2 \cup \{(t_1', a, s_2'')\}$ and
- $\rho' = \rho_1 \cup \rho_2$.

Note that ρ' is well defined since V_1 and V_2 are disjoint. Finally, we define $s = s_1'$ and $t = t_2''$. It is easy to show that $(s, t) \in q(G')$ if and only if there are two disjoint paths in G from s_1 to t_1 and from s_2 to t_2 (Exercise **??**). □

Corollary 57.4. *Assume that we have a formalism for data paths that can define L_{eq} and that is closed under complement. Then the data complexity of evaluating data path queries is* NP*-hard.* ***(Wim)***[3]

Extensions of Regular Path Queries for Data Graphs

We will define a query language for data paths that can do Boolean combinations of atomic $=, \neq$ comparisons of data values. To define such conditions formally, assume that, for each $k > 0$, we have variables $x_1, \dots, x_k$. Then conditions in $\mathcal{C}_k$ are given by the grammar:

$$c \quad := \quad x_i^= \mid x_i^{\neq} \mid z^= \mid z^{\neq} \mid c \wedge c \mid c \vee c \mid \neg c, \qquad \text{for } 1 \leq i \leq k,$$

where z is a data value from Const, also referred to as the *constant*. Let $\mathsf{Const}_\perp = \mathsf{Const} \uplus \{\perp\}$, where $\perp$ is a special symbol that we assign to a variable x_i if it has not been assigned a value from Const yet. The satisfaction of a condition is defined with respect to a data value $d \in \mathsf{Const}$ and a tuple $\tau = (d_1, \dots, d_k) \in \mathsf{Const}_\perp^k$ as follows:

- $d, \tau \models x_i^=$ iff $d = d_i$;
- $d, \tau \models x_i^{\neq}$ iff $d \neq d_i$;
- $d, \tau \models z^=$ iff $d = z$;
- $d, \tau \models z^{\neq}$ iff $d \neq z$;
- $d, \tau \models c_1 \wedge c_2$ iff $d, \tau \models c_1$ and $d, \tau \models c_2$ (and likewise for $c_1 \vee c_2$);
- $d, \tau \models \neg c$ iff $d, \tau \not\models c$.

[3] **Wim:** Too informal.

In what follows, ε is a shorthand for a condition that is true for any valuation and data value (e.g. $c \vee \neg c$). **(Wim)**[4]

Definition 57.5 (Regular expressions with memory). *Let Σ be a finite alphabet and $x_1, \ldots, x_k$ a set of variables. Then* regular expressions with memory (REM) *are defined by the grammar:*

$$e \quad := \quad \varepsilon \ \mid \ a \ \mid \ e + e \ \mid \ e \cdot e \ \mid \ e^+ \ \mid \ e[c] \ \mid \ \downarrow\overline{x}.e \tag{57.1}$$

where a ranges over alphabet letters, c over conditions in $\mathcal{C}_k$, and $\overline{x}$ over tuples of variables from $x_1, \ldots, x_k$.

A regular expression with memory e is *well-formed* if it satisfies two conditions:

- Subexpressions e^+, $e[c]$, and $\downarrow\overline{x}.e$ are not allowed if e reduces to ε. Formally, e reduces to ε if it is ε, or it is $e_1 + e_2$ or $e_1 \cdot e_2$ or e_1^+ or $e_1[c]$ or $\downarrow\overline{x}.e_1$ where e_1 and e_2 reduce to ε.
- No variable appears in a condition before it appears in $\downarrow\overline{x}$.

The extra condition of being well-formed is to rule out pathological cases like $\varepsilon[c]$ for checking conditions over empty subexpressions, or $a[x^=]$ for checking equality with a variable that has not been defined. From now on we assume that all REMs are well-formed.

Example 57.6. We discuss a few examples of REMs. The language L_{eq} of data path labels in which two data values are the same is given by the expression $\Sigma^* \cdot \downarrow x.\Sigma^+[x^=] \cdot \Sigma^*$, where Σ is the shorthand for $a_1 + \ldots + a_\ell$, whenever $\Sigma = \{a_1, \ldots, a_\ell\}$ and Σ^* is the shorthand for $\Sigma^+ + \varepsilon$. It says: at some point, bind x, and then check that after one or more edges, we have the same data value.

The language where the finite alphabet part is a sequence of as, and where each data value differs from the first one, while the second value also differs from 5, is given by $\downarrow x \cdot a[5^{\neq} \wedge x^{\neq}] \cdot (a[x^{\neq}])^*$. It starts by binding x to the first data value; then it proceeds checking that the letter is a and condition $5^{\neq} \wedge x^{\neq}$ is satisfied, which is expressed by $a[5^{\neq} \wedge x^{\neq}]$. After that it proceeds to the end by reading letters a and data values different from the first, expressed by $(a[x^{\neq}])^*$.

To define the semantics of REMs, we first define the *concatenation* of two data words $w = d_1 a_1 \ldots a_{n-1} d_n$ and $w' = d_n a_n \ldots a_{m-1} d_m$ as $w \cdot w' = d_1 a_1 \ldots a_{n-1} d_n a_n \ldots a_{m-1} d_m$. Notice that it is only defined if the last data value of w equals the first data value of w'. The definition naturally extends to concatenation of more data words. If $w = w_1 \cdots w_\ell$, we refer to $w_1 \cdots w_\ell$ as a *splitting* of a data word w (into $w_1, \ldots, w_\ell$).

[4] **Wim:** Is this used?

The semantics of REMs is defined by means of a relation $(e, w, \sigma) \vdash \sigma'$, where e is a REM, w is a data word, and both σ and σ' are k-tuples over $\mathsf{Const} \cup \{\perp\}$ (the symbol $\perp$ means that a register has not been assigned yet). The intuition is as follows: one can start with a memory configuration σ (i.e., values of $x_1, \ldots, x_k$) and parse w according to e in such a way that at the end the memory configuration is σ'. The language of e is then defined as

$$L(e) = \{w \mid (e, w, \overline{\perp}) \vdash \sigma \text{ for some } \sigma\},$$

where $\overline{\perp}$ is the tuple of k values $\perp$.

The relation $\vdash$ is defined inductively on the structure of expressions. Recall that a data word can consist of a single data value d. We use the notation $\sigma_{\overline{x}=d}$ for the valuation obtained from σ by setting all the variables in $\overline{x}$ to d.

- $(\varepsilon, w, \sigma) \vdash \sigma'$ iff $w = d$ for some $d \in \text{Dom}$ and $\sigma' = \sigma$.
- $(a, w, \sigma) \vdash \sigma'$ iff $w = d_1 a d_2$ and $\sigma' = \sigma$.
- $(e_1 \cdot e_2, w, \sigma) \vdash \sigma'$ iff there is a splitting $w = w_1 \cdot w_2$ of w and a valuation σ'' such that $(e_1, w_1, \sigma) \vdash \sigma''$ and $(e_2, w_2, \sigma'') \vdash \sigma'$.
- $(e_1 + e_2, w, \sigma) \vdash \sigma'$ iff $(e_1, w, \sigma) \vdash \sigma'$ or $(e_2, w, \sigma) \vdash \sigma'$.
- $(e^+, w, \sigma) \vdash \sigma'$ iff there are a splitting $w = w_1 \cdots w_m$ of w and valuations $\sigma_0, \sigma_1, \ldots, \sigma_m$ with $\sigma_0 = \sigma$, $\sigma_m = \sigma'$, and $(e, w_i, \sigma_{i-1}) \vdash \sigma_i$ for all $i \in [m]$.
- $(\downarrow\overline{x}.e, w, \sigma) \vdash \sigma'$ iff $(e, w, \sigma_{\overline{x}=d}) \vdash \sigma'$, where d is the first data value of w.
- $(e[c], w, \sigma) \vdash \sigma'$ iff $(e, w, \sigma) \vdash \sigma'$ and $\sigma', d \models c$, where d is the last data value of w.

Notice that in the last item we require that σ', and not σ, satisfies c. The reason for this is that our initial assignment might change before reaching the end of the expression and we want this change to be reflected when we check that condition c holds.

Definition 57.7 (Regular Queries with Memory). *The* regular queries with memory (RQM) *are the data path queries for which the language L of data words is given by a regular expression with memory.*

Wim:
The following is just a code dump at the moment.

Theorem 57.8. *The data complexity of RQMs over data graphs is* NLogSpace-*complete and the combined complexity of RQMs over data graphs is* PSpace-*complete.*

Proof. We first consider data complexity. The NLogSpace lower bound is immediate from reachability in graphs. The NLogSpace upper bound immediately follows from Corollary **??** and Theorem **??**. Indeed, Corollary **??** states that each REM can be translated into some (constant-size) register data word automaton, for which data complexity is in NLogSpace according to Theorem **??**. Combined omitted for now.

Definition 57.9 (Expressions with equality). *Let Σ be a finite alphabet. Then* regular expressions with equality (REE) *are defined by the grammar:*

$$e \quad := \quad \varepsilon \;\mid\; a \;\mid\; e+e \;\mid\; e\cdot e \;\mid\; e^{+} \;\mid\; e[c] \;\mid\; e_{=} \;\mid\; e_{\neq} \tag{57.2}$$

where a ranges over alphabet letters and c is a simplified condition.

The language $L(e)$ of data words denoted by a regular expression with equality e is defined as follows.

- $L(\varepsilon) = \{d \mid d \in \mathrm{Dom}\}$.
- $L(a) = \{dad' \mid d, d' \in \mathrm{Dom}\}$.
- $L(e \cdot e') = L(e) \cdot L(e')$.
- $L(e + e') = L(e) \cup L(e')$.
- $L(e^{+}) = \{w_1 \cdots w_k \mid k \geq 1 \text{ and each } w_i \in L(e)\}$.
- $L(e[c]) = \{d_1 a_1 \ldots a_{n-1} d_n \in L(e) \mid d_n \models c\}$.
- $L(e_{=}) = \{d_1 a_1 \ldots a_{n-1} d_n \in L(e) \mid d_1 = d_n\}$.
- $L(e_{\neq}) = \{d_1 a_1 \ldots a_{n-1} d_n \in L(e) \mid d_1 \neq d_n\}$.

Definition 57.10 (Regular Queries with Data Tests). *The* regular queries with data tests (RQD) *are the data path queries for which the language L of data words is given by a regular expression with equality.*

Example 57.11. Coming back to the database from Figure **??**, we can now ask the following queries.

- The query asking for people with a finite Bacon number is again the same as in Example **??**.
- The query that checks if there is a movie in the database with at least two different actors is defined by $Q = \mathsf{RPQ}xye$, with $e := (\mathsf{stars\_in} \cdot \mathsf{actor})_{\neq}$. Note that a nonempty answer to this query merely signifies that such a movie exists. To actually retrieve the movie we would need to use conjunctive queries with *rqd*s as atoms (Section **??**).

Theorem 57.12. *The data complexity of RQDs over data graphs is* NLogSpace*-complete and the combined complexity of RQDs over data graphs is in* Ptime*.*

Proof. For data complexity, the proof is analogous to the one in Theorem 57.8. The bound for combined complexity follows from Theorem **??** which is shown for a fragment of the GXPath language which is strictly stronger.

Wim:
What else do we want to put in the chapter? Say that XPath-like query languages seem to behave well?

58

RDF and SPARQL

Marcelo

RDF model and abstraction of SPARQL via posRA + outerjoin

Comments and Exercises for Part IX

Exercise 58.1. Reduce the Two Disjoint Paths problem to the Two Disjoint Trails problem. *Hint:* Replace every node v with an edge $(v_{\text{in}}, v_{\text{out}})$.

Exercise 58.2. (1) Show that, in the proof of Theorem 56.4, one can construct in polynomial time an NFA B over the alphabet $\mathcal{A}$ that recognizes the language of all encodings of expansions of φ_1 as explained in *Warm-up*.

(2) Show that, in the proof of Theorem 56.4, one can construct in polynomial time an NFA B_1 over the alphabet $\mathcal{A}_1$ that recognizes the language of all encodings of expansions of φ_1 as explained in *Step 1*.
Hint. Construct B_1 as a product of five automata, each of which has at most exponential size. For each $i \in [1, 4]$, take $A_{1,i}$ to be the NFA that checks condition (i) on words $w_{\mathcal{E}_1}$. The fifth automaton is an NFA A_{wf} that tests well-formedness of the input word.

Bibliographic Remarks

CRPQ containment procedure is from [4].

In Chapter 57, we assume a single data value per node for simplicity and note that nodes u with $k > 1$ attributes can be modeled by a node u with k outgoing edges, leading to k new nodes containing the data values. For a discussion on how this can be implemented for languages considered in this paper we refer the reader to [**?**]. The same reference also illustrates how the approach when data values reside in the edges, or in both nodes and edges, can be reduced to the approach taken in Chapter 57.

References

1. S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.
2. S. Arora and B. Barak. *Computational Complexity - A Modern Approach*. Cambridge University Press, 2009.
3. D. A. M. Barrington, N. Immerman, and H. Straubing. On uniformity within NC^1. *J. Comput. Syst. Sci.*, 41(3):274–306, 1990.
4. D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Y. Vardi. Containment of conjunctive regular path queries with inverse. In *KR 2000, Principles of Knowledge Representation and Reasoning Proceedings of the Seventh International Conference, Breckenridge, Colorado, USA, April 11-15, 2000.*, pages 176–185, 2000.
5. C. J. Date and H. Darwen. *A Guide to the SQL Standard*. Addison-Wesley, 1996.
6. H. B. Enderton. *A mathematical introduction to logic*. Academic Press, 1972.
7. K. Etessami, M. Y. Vardi, and T. Wilke. First-order logic with two variables and unary temporal logic. *Inf. Comput.*, 179(2):279–295, 2002.
8. R. Fagin. Probabilities on finite models. *J. Symb. Log.*, 41(1):50–58, 1976.
9. S. Fortune, J. E. Hopcroft, and J. Wyllie. The directed subgraph homeomorphism problem. *Theor. Comput. Sci.*, 10:111–121, 1980.
10. H. Gaifman. On local and non-local properties. In *Proceedings Herbrand Symposium Logic Colloquium, North Holland, 1982*, pages 105–135, 1982.
11. H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database Systems - the Complete Book*. Pearson Education, 2009.
12. E. Grädel, P. Kolaitis, L. Libkin, M. Marx, J. Spencer, M. Vardi, and S. Weinstein. *Finite Model Theory and its Applications*. Springer, 2008.
13. P. Guagliardo and L. Libkin. A formal semantics of SQL queries, its validation, and applications. *PVLDB*, 11(1):27–39, 2017.
14. P. R. Halmos. *Measure Theory*. Springer, 1974.
15. W. P. Hanf. Model-theoretic methods in the study of elementary logic. In *The Theory of Models; Addison, Henkin, and Tarski, eds., North Holland 1965*, pages 132–145, 1965.

16. L. Hella, L. Libkin, J. Nurmonen, and L. Wong. Logics with aggregate operators. *J. ACM*, 48(4):880–907, 2001.
17. J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 2003.
18. N. Immerman. *Descriptive Complexity*. Springer, 1999.
19. A. C. Klug. Equivalence of relational algebra and relational calculus query languages having aggregate functions. *J. ACM*, 29(3):699–717, 1982.
20. A. C. Klug. On conjunctive queries containing inequalities. *Journal of the ACM*, 35(1):146–160, 1988.
21. D. Kozen. *Automata and Computability*. Springer, 1997.
22. L. Libkin. Logics with counting and local properties. *ACM Trans. Comput. Log.*, 1(1):33–59, 2000.
23. L. Libkin. Expressive power of SQL. *Theor. Comput. Sci.*, 296(3):379–404, 2003.
24. L. Libkin. *Elements of Finite Model Theory*. Springer, 2004.
25. G. Özsoyoglu, Z. M. Özsoyoglu, and V. Matos. Extending relational algebra and relational calculus with set-valued attributes and aggregate functions. *ACM Trans. Database Syst.*, 12(4):566–592, 1987.
26. C. H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.
27. R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw-Hill, 2003.
28. B. Rossman. Homomorphism preservation theorems. *J. ACM*, 55(3):15:1–15:53, 2008.
29. Y. Sagiv and M. Yannakakis. Equivalences among relational expressions with the union and difference operators. *J. ACM*, 27(4):633–655, 1980.
30. A. Silberschatz, H. F. Korth, and S. Sudarshan. *Database System Concepts*. McGraw-Hill Book Company, 2005.
31. M. Sipser. *Introduction to the Theory of Computation*. PWS Publishing Company, 1997.
32. J. Spencer. *The Strange Logic of Random Graphs*. Springer, 2010.
33. R. van der Meyden. The complexity of querying indefinite data about linearly ordered domains. *J. Comput. Syst. Sci.*, 54(1):113–135, 1997.
34. H. Vollmer. *Introduction to Circuit Complexity - A Uniform Approach*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 1999.
35. I. Wegener. *Complexity Theory*. Springer, 2005.

Part X

Appendix: Theory of Computation

A

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

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

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

where Q is the finite set of states, Σ is the finite set of input symbols (also called the *alphabet*), δ is the transition function, and $s \in Q$ is the start state. We assume that Q contains the *accepting state* "yes", and the *rejecting state* "no". Accepting and rejecting states are needed for decision problems: they determine whether the input belongs to the language or not. The transition function δ is of the form

$$\delta : \ (Q - \{\text{“yes”}, \text{“no”}\}) \times \Sigma \quad \to \quad Q \times \Sigma \times \{\rightarrow, \leftarrow, -\}.$$

Note that the accepting and rejecting states do not have outgoing transitions. In the context of TMs, $\sqcup$ and $\rhd$ are special symbols, called the *blank symbol* and the *left marker*. We assume $\sqcup$ and $\rhd$ always to be in Σ.

A *configuration* of a TM is a tuple

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

where $q \in Q$ and u, v are words in Σ^*, where u is 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 $\rhd$. Moreover, the transition function δ is restricted in such a way that $\rhd$ 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 $\delta(q, a) = (q', b, \mathrm{dir})$, where $\mathrm{dir} \in \{\rightarrow, \leftarrow, -\}$. 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 $\rhd$ symbol (the transition function δ is restricted in such a way that this cannot happen: if $\delta(q, \rhd) = (q', a, \mathrm{dir})$, then $a = \rhd$ and $\mathrm{dir} \neq \leftarrow$). For example, if $c = (q, \rhd 01, 100)$ and $\delta(q, 1) = (q', 0, \leftarrow)$, then $c' = (q', \rhd 0, 0100)$. In this case, we write $c \rightarrow_M c'$, and we also write $c \rightarrow_M^m c'$ if c' can be reached from c in m steps, and $c \rightarrow_M^* c'$ if $c \rightarrow_M^m c'$ for some $m \geq 0$ (we assume that $c \rightarrow_M^0 c$). Finally, if $v = \varepsilon$ and $\mathrm{dir} =\rightarrow$, 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 $a_i \in \Sigma - \{\sqcup, \rhd\}$ for each $i \in [n]$. The *start configuration of M on input w* is $sc(w) = (s, \rhd, w)$. We call a configuration c *accepting* if its state is “yes”, and *rejecting* if its state is “no”. The TM M *accepts* (respectively, *rejects*) input w if $sc(w) \rightarrow_M^* c$ for some accepting (respectively, rejecting) configuration c.

Nondeterministic Turing Machines as Acceptors

We also use nondeterministic Turing Machines as acceptors. A *nondeterministic Turing Machine (NTM)* is defined as a tuple $M = (Q, \Sigma, \delta, s)$ in the same way as (deterministic) TMs, with the key difference that δ is of the form

$$\delta : \ (Q - \{\text{“yes”}, \text{“no”}\}) \times \Sigma \quad \to \quad \mathcal{P}(Q \times \Sigma \times \{\rightarrow, \leftarrow, -\}).$$

Therefore, 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 TMs that, apart from a read-write working tape, they also have a read-only input tape. A 2-*tape (deterministic) Turing Machine (*2-*TM)* is defined as a tuple $M = (Q, \Sigma, \delta, s)$ in the same way as (deterministic) TMs, with the crucial difference that δ is of the form

$$\delta : \ (Q - \{\text{“yes”}, \text{“no”}\}) \times \Sigma \times \Sigma \quad \to \quad Q \times \{\to, \leftarrow, -\} \times \Sigma \times \{\to, \leftarrow, -\}.$$

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_1v_1 and the head of this tape is reading the last symbol of u_1, while the working tape has content u_2v_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_iv_i, and always as the first symbol of u_i.

Assume that M is in a configuration $c = (q, u_1a_1, v_1, u_2a_2, v_2)$, where $q \in Q - \{\text{“yes”}, \text{“no”}\}$, $a_1, a_2 \in \Sigma$ and $u_1, v_1, u_2, v_2 \in \Sigma^*$, and assume that $\delta(q, a_1, a_2) = (q', \mathrm{dir}_1, b, \mathrm{dir}_2)$, where dir_i is a direction, i.e., one of $\{\to, \leftarrow, -\}$. 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_1a_1, v_1 by moving the head one step in the direction dir_1, and u_2', v_2' is obtained from u_2a_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 $\delta(q, 1, \triangleright) = (q', \leftarrow, \triangleright, \to)$, 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. Formally, a 2-*tape nondeterministic Turing Machine (2-NTM)* is defined as a tuple $M = (Q, \Sigma, \delta, s)$ as for 2-TMs, with the difference that δ is of the form

$$\delta : \ (Q - \{\text{"yes"}, \text{"no"}\}) \times \Sigma \times \Sigma \quad \rightarrow \quad \mathcal{P}(Q \times \{\rightarrow, \leftarrow, -\} \times \Sigma \times \{\rightarrow, \leftarrow, -\}).$$

Therefore, for a given 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 a given input w. The transition function δ of a *Turing Machine with output (TMO)* is

$$\delta : \ (Q - \{\text{"halt"}\}) \times \Sigma \times \Sigma \quad \rightarrow \quad Q \times \{\rightarrow, \leftarrow, -\} \times \Sigma \times \{\rightarrow, \leftarrow, -\} \times \Sigma.$$

If $\delta(q, a_1, a_2) = (q', \mathrm{dir}_1, b, \mathrm{dir}_2, c)$, then $q', \mathrm{dir}_1, b, \mathrm{dir}_2$ are used exactly as in the case of a 2-TM accepting a language. Moreover, if $c \neq \sqcup$, then c is written on the output tape and the head of this tape is moved one position to the right; otherwise, no changes are made on this tape. The start configuration of a TMO M on input w is $sc(w) = (s, \triangleright, w, \triangleright, \varepsilon, \triangleright, \varepsilon)$. The output of M on input w is the word u such that $sc(w) \rightarrow_M^* (\text{"halt"}, u_1, v_1, u_2, v_2, \triangleright u, \varepsilon)$.

Complexity Classes

We proceed to introduce some central complexity classes that are used in this book. From now on, assume 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

$$\begin{aligned}\mathrm{TIME}(f(n)) \;=\; \{L \mid \; & \text{there exists a TM that decides } L \\ & \qquad\qquad \text{and runs in time } f(n)\}.\end{aligned}$$

and

$$\begin{aligned}\mathrm{NTIME}(f(n)) \;=\; \{L \mid \; & \text{there exists a NTM that decides } L \\ & \qquad\qquad \text{and runs in time } f(n)\}.\end{aligned}$$

The following time complexity classes are used in this book:

$$\begin{array}{ll}\textsc{PTime} = \bigcup_{k \in \mathbb{N}} \mathrm{TIME}(n^k) & \mathrm{NP} = \bigcup_{k \in \mathbb{N}} \mathrm{NTIME}(n^k) \\ \textsc{ExpTime} = \bigcup_{k \in \mathbb{N}} \mathrm{TIME}(2^{n^k}) & \textsc{NExpTime} = \bigcup_{k \in \mathbb{N}} \mathrm{NTIME}(2^{n^k}) \\ 2\textsc{ExpTime} = \bigcup_{k \in \mathbb{N}} \mathrm{TIME}(2^{2^{n^k}}) & \end{array}$$

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 $|u_2 v_2| \leq f(|w|)$.[2] We say that M *decides* a language L if M accepts every word in L and rejects every word not in L. We define the classes of decision problems

$$\begin{aligned}\mathrm{SPACE}(f(n)) \;=\; \{L \mid \; & \text{there exists a 2-TM that decides } L \\ & \qquad\qquad \text{and runs in space } f(n)\}.\end{aligned}$$

and

$$\begin{aligned}\mathrm{NSPACE}(f(n)) \;=\; \{L \mid \; & \text{there exists a 2-NTM that decides } L \\ & \qquad\qquad \text{and runs in space } f(n)\}.\end{aligned}$$

The following space complexity classes are used in this book:

[1] The running time of a TMO is defined in the same way.

[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 $|u_2 v_2| \leq f(|w|)$.

$$\begin{array}{ll} \textsc{DLogSpace} = \mathrm{SPACE}(\log n) & \textsc{NLogSpace} = \mathrm{NSPACE}(\log n) \\ \textsc{PSpace} = \bigcup_{k\in\mathbb{N}} \mathrm{SPACE}(n^k) & \textsc{NPSpace} = \bigcup_{k\in\mathbb{N}} \mathrm{NSPACE}(n^k) \\ \textsc{ExpSpace} = \bigcup_{k\in\mathbb{N}} \mathrm{SPACE}(2^{n^k}) & \textsc{NExpSpace} = \bigcup_{k\in\mathbb{N}} \mathrm{NSPACE}(2^{n^k}) \end{array}$$

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 co$\mathcal{C}$ is defined as the set of complements of the problems in $\mathcal{C}$, that is, co$\mathcal{C} = \{\Sigma^* - L \mid L \in \mathcal{C}\}$. It is known that

$$\begin{gathered} \textsc{DLogSpace} \subseteq \textsc{NLogSpace} \subseteq \textsc{Ptime} \subseteq \mathrm{NP} \subseteq \textsc{PSpace} = \textsc{NPSpace} \\ \subseteq \textsc{ExpTime} \subseteq \textsc{NExpTime} \subseteq \textsc{ExpSpace} = \textsc{NExpSpace} \subseteq 2\textsc{ExpTime} \end{gathered}$$

$$\textsc{Ptime} \subsetneq \textsc{ExpTime} \subsetneq 2\textsc{ExpTime}$$

$$\mathrm{NP} \subsetneq \textsc{NExpTime}$$

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]

[3] We could define hardness for the complexity class 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.

We say that a decision problem is *tractable* if it is in PTIME. As such, problems 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).

Perhaps the most fundamental problem that is presumably intractable is SAT: Given a Boolean formula φ, is it satisfiable? It is well-known that SAT is NP-complete. The SAT problem is a special case of the *quantified satisfiability* (QSAT), also known under the name *quantified Boolean formula* or QBF. We define it next. For a Boolean formula φ and a tuple $\bar{x}$ of Boolean variables, we denote by $\varphi(\bar{x})$ the fact that φ uses precisely the variables in $\bar{x}$. A *quantified Boolean formula* is an expression of the form

$$\exists\bar{x}_1\forall\bar{x}_2\exists\bar{x}_3\cdots Q_n\bar{x}_n\,\varphi(\bar{x}_1,\ldots,\bar{x}_n)\,,$$

where Q_n is $\forall$ if n is even and $\exists$ if n is odd. Then QSAT is defined as follows:

PROBLEM:	QSAT
INPUT:	A quantified Boolean formula ψ
OUTPUT:	yes if ψ is true and no otherwise

Notice that SAT is the special case of QSAT where $\psi = \exists\bar{x}\,\varphi(\bar{x})$. Two special cases of QSAT will be particularly important for this book, namely the ones with exactly one quantifier alternation:

PROBLEM:	$\forall\exists$QSAT
INPUT:	A quantified Boolean formula $\forall\bar{x}_1\exists\bar{x}_2\,\varphi(\bar{x}_1,\bar{x}_2)$
OUTPUT:	yes if $\forall\bar{x}_1\exists\bar{x}_2\varphi$ is true and no otherwise

PROBLEM:	$\exists\forall$QSAT
INPUT:	A quantified Boolean formula $\exists\bar{x}_1\forall\bar{x}_2\,\varphi(\bar{x}_1,\bar{x}_2)$
OUTPUT:	yes if $\exists\bar{x}_1\forall\bar{x}_2\varphi$ is true and no otherwise

We define Σ_2^p as the class of decision problems reducible to $\exists\forall$QBF. Similarly, Π_2^p is the class of decision problems reducible to $\forall\exists$QBF. The problem QSAT is PSPACE-complete. As such, we know that

$$\begin{tikzcd}
 & \mathrm{NP} \arrow[dr, phantom, "\subseteq" sloped] & & \Sigma_2^p \arrow[dr, phantom, "\subseteq" sloped] & \\
\mathrm{PTIME} \arrow[ur, phantom, "\subseteq" sloped] \arrow[dr, phantom, "\subseteq" sloped] & & \mathrm{NP} \cup \mathrm{coNP} \arrow[ur, phantom, "\subseteq" sloped] \arrow[dr, phantom, "\subseteq" sloped] & & \mathrm{PSPACE} \\
 & \mathrm{coNP} \arrow[ur, phantom, "\subseteq" sloped] & & \Pi_2^p \arrow[ur, phantom, "\subseteq" sloped] &
\end{tikzcd}$$

We finally remark that the smallest complexity class we consider here is DLogSpace. In database theory, and especially in its logical counterpart – 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 here. 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.

Encoding Inputs and Outputs

To reason about the computational complexity of problems such as query evaluation, we need to represent their inputs (that is, queries and databases) as inputs to Turing Machines, i.e., as words over some finite alphabet.

Queries will most commonly be coming from a query language defined by a formal syntax, and we thus associate a query with its parse tree, which of course can easily be encoded as a word over a finite alphabet.

We then assume that there is some way of encoding the values from Const that assigns a word $enc(c)$ to every $c \in \mathsf{Const}$. For example, if $\mathsf{Const} = \{c_0, c_1, \ldots\}$, then we may simply take $enc(c_i)$ to be the number i in binary. To extend this to databases of a schema $\mathbf{S}$ with relations $R_1, \ldots, R_n$, we add new separator symbols $\triangle, \#, \$, \square$ to the alphabet, and encode a database instance D with $\mathrm{Dom}(D) = \{a_1, \ldots, a_k\}$ as

$$enc(D) = \triangle enc(a_1)\triangle \ldots \triangle enc(a_k)\triangle\# enc(R_1^D)\# \ldots \# enc(R_n^D)\#$$

where

- for each relation R_i^D with tuples $\{\bar{t}_1, \ldots, \bar{t}_m\}$,

$$enc(R_i^D) = \$ enc(\bar{t}_1)\$ \ldots \$ enc(\bar{t}_m)\$\,,$$

- and for each tuple $\bar{t} = (a_1, \ldots, a_l)$,

$$enc(\bar{t}) = \square enc(a_1)\square \ldots \square enc(a_l)\square\,.$$

This, in fact, is not the only possibility. The key property of this encoding is that, for a database instance D and a tuple $\bar{t}$ given as their encodings $enc(D)$ and $enc(\bar{t})$, one can check, in DLogSpace, whether $\bar{t} \in R_i^D$ for any relation R_i in the schema. Any other encoding with this property can be used, and it will not change any of the complexity results in this book.

B

Big-O Notation

We use $\mathbb{R}_0^+$ to denote the set of non-negative real numbers. We write $\mathbb{R}^+$ 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*.

Definition B.1. *Let $f, g : \mathbb{N} \to \mathbb{R}_0^+$. We say that $f(x)$* is in *$O(g(x))$ if there exist $k \in \mathbb{R}^+$ and $n_0 \in \mathbb{N}$ such that, for every $x \geq n_0$, we have $f(x) \leq k \cdot g(x)$.*

The big-O notation is typically defined for single variable functions. This is why the domain of f and g is $\mathbb{N}$ in Definition B.1. It is possible to generalize the above definition to multiple variable functions, which is particularly useful in the database setting, where the input to key problems usually consists of several different components. 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.

Definition B.2. *Let $f, g : \mathbb{N}^\ell \to \mathbb{R}_0^+$, where $\ell \geq 1$. We say that $f(x_1, \ldots, x_\ell)$* 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 $i \in [\ell]$, $f(x_1, \ldots, x_\ell) \leq k \cdot g(x_1, \ldots, x_\ell)$.*

C

Tiling Problems

Wim:
Plan: Tiling problems for PSpace, ExpTime, 2ExpTime. Give master reduction so that students understand the connection to TM computations. If we decide to use two-player corridor tiling in the book, introduce it here.

A Turing Machine computation can be seen as a grid, labeled with alphabet symbols and state-symbol pairs. This is also called a *computation table* in the literature.

Wim:
This is just a dump from my lecture notes; so notation may clash with what we have. It's all macros, so easy to change.

We first define the domino tiling problem. A *domino tiling instance* is a tuple $\mathcal{T} = (T, u^{\text{first}}, u^{\text{last}}, V, H)$, where

- T is a finite set of elements which we call *tile types*;
- u^{first} is the *first tile type*;
- u^{last} is the *last tile type*;
- $H \subseteq T \times T$ is a set of *horizontal constraints*; and
- $V \subseteq T \times T$ is a set of *vertical constraints*.

Given $\mathcal{T} = (T, u^{\text{first}}, u^{\text{last}}, V, H)$, a mapping

$$\lambda : \{0, \dots, n\} \times \{0, \dots, m\} \to T$$

is called a *tiling of* $\mathcal{T}$. We sometimes also call λ an $n \times m$ *tiling of* $\mathcal{T}$ if we want to emphasize the dimensions of λ. Here, we interpret $(0, 0)$ as the bottom left coordinate and (n, m) as the top right coordinate of a grid. A tiling is *valid*, if

- the first tile type is u^{first} and the last tile type is u^{last}, i.e. $\lambda(0,0) = u^{\text{first}}$ and $\lambda(n,0) = u^{\text{last}}$; and
- the horizontal and vertical constraints are satisfied, i.e., $(\lambda(i,j), \lambda(i+1,j)) \in H$ for all $0 \le i < n, 0 \le j \le m$ and $(\lambda(i,j), \lambda(i,j+1) \in V$ for all $0 \le i \le n, 0 \le j < m$.

Finally, the tiling problem is defined as follows.

PROBLEM:	TILING
INPUT:	A tiling instance $\mathcal{T}$
OUTPUT:	yes if $\mathcal{T}$ has a valid tiling and no otherwise

Theorem C.1. *TILING is undecidable.*

Proof.

Wim:
Master reduction here?

Wim:
Then infer the following as corollaries?

PROBLEM:	CorridorTiling
INPUT:	A tiling instance $\mathcal{T}$ and $k \in \mathbb{N}$ in unary
OUTPUT:	yes if $\mathcal{T}$ has a valid $n \times k$ tiling for some $n \in \mathbb{N}$ and no otherwise

PROBLEM:	EXPTiling
INPUT:	A tiling instance $\mathcal{T}$ and $k \in \mathbb{N}$ in unary
OUTPUT:	yes if $\mathcal{T}$ has a valid $2^k \times 2^k$ tiling and no otherwise

PROBLEM:	2EXPTiling
INPUT:	A tiling instance $\mathcal{T}$ and $k \in \mathbb{N}$ in unary
OUTPUT:	yes if $\mathcal{T}$ has a valid $2^{2^n} \times 2^{2^n}$ tiling and no otherwise

Theorem C.2.

(1) CorridorTiling is PSPACE-*complete.*

(2) EXPTiling is EXPTIME-*complete.*

(3) 2EXPTiling is 2EXPTIME-*complete.*

Wim:
We need two-player corridor tiling too.

D

Regular Languages

Wim:
Define regular expressions and finite automata. Say that there's an efficient conversion from expressions to automata; and that the conversion in the other way is exponential.

Wim:
Boolean closures. Emptiness.

Regular Expressions and Finite Automata

Conversions

Complexity Results

Theorem D.1. *(1) Containment for NFAs is* PSPACE*-complete.*

(2) Containment for regular expressions is PSPACE*-complete.*

Proof.

Wim:
This is just a dump from one of my papers (SICOMP 2009 on chain regular expressions). Proof is for a special class of regular expressions so can probably be simplified.

We first show that containment is PSPACE-hard for $\mathrm{RE}(a, (+a)^+)$ and we consider the case of $\mathrm{RE}(a, (+a)^*)$ later. In both cases, we use a reduction from the CORRIDOR TILING problem, which is known to be PSPACE-complete.

To this end, let $D = (T, H, V, \bar{b}, \bar{t}, n)$ be a tiling system. Without loss of generality, we assume that $n \geq 2$. We construct two regular expressions R_1 and R_2 such that

$$L(R_1) \subseteq L(R_2) \text{ if and only if there exists no correct corridor tiling for } D.$$

We will use symbols from $\Sigma = T \times \{1, \ldots, n\}$, where, for each i, $\Sigma_i = \{[t, i] \mid t \in T\}$ will be used to tile the i-th column. For ease of exposition, we denote $\Sigma \cup \{\$\}$ by $\Sigma_\$$ and $\Sigma \cup \{\#, \$\}$ by $\Sigma_{\#,\$}$. We encode candidates for a correct tiling by strings in which the rows are separated by the symbol \$, that is, by strings of the form

$$\$\bar{b}\$\Sigma^+\$\Sigma^+\$\cdots\$\Sigma^+\$\bar{t}\$. \qquad (\dagger)$$

The following regular expressions detect strings of this form which do not encode a correct tiling:

- $\Sigma_\$^+[t, i][t', j]\Sigma_\$^+$, for every $t, t' \in T$, where $i = 1, \ldots, n-1$ and $j \neq i+1$. These expressions detect consecutive symbols that are not from consecutive column sets;
- $\Sigma_\$^+\$[t, i]\Sigma_\$^+$ for every $i \neq 1$ and $[t, i] \in \Sigma_i$, and $\Sigma_\$^+[t, i]\$\Sigma_\$^+$ for every $i \neq n$ and $[t, i] \in \Sigma_i$. These expressions detect rows that do not start or end with a correct symbol. Together with the previous expressions, these expressions detect all candidates with at least one row not in $\Sigma_1 \cdots \Sigma_n$.
- $\Sigma_\$^+[t, i][t', i+1]'\Sigma_\$^+$, for every $(t, t') \notin H$, and $i = 1, \ldots, n-1$. These expressions detect all violations of horizontal constraints.
- $\Sigma_\$^+[t, i]\Sigma^+\$\Sigma^+[t', i]\Sigma_\$^+$, for every $(t, t') \notin \Sigma$ and for every $i = 1, \ldots, n$. These expressions detect all violations of vertical constraints.

Let $e_1, \ldots, e_k$ be an enumeration of the above expressions. Notice that $k = \mathcal{O}(|D|^2)$. It is straightforward that a string w in ($\dagger$) does not match $\bigcup_{i=1}^k e_i$ if and only if w encodes a correct tiling.

Let $e = e_1 \cdots e_k$. Because of leading and trailing $\Sigma_\$^+$ expressions, $L(e) \subseteq L(e_i)$ for every $i = 1, \ldots, k$. We are now ready to define R_1 and R_2:

$$\begin{aligned} R_1 &= \overbrace{\#e\#e\#\cdots\#e\#}^{k \text{ times } e}\$\bar{b}\$\Sigma_\$^+\$\bar{t}\$\overbrace{\#e\#e\#\cdots\#e\#}^{k \text{ times } e} \quad \text{and} \\ R_2 &= \qquad\qquad \Sigma_{\#,\$}^+\#e_1\#e_2\#\cdots\#e_k\#\Sigma_{\#,\$}^+. \end{aligned}$$

Notice that both R_1 and R_2 are in $\mathrm{RE}(a, (+a)^+)$ and can be constructed using only logarithmic space. It remains to show that $L(R_1) \subseteq L(R_2)$ if and only if there is no correct tiling for D.

We first show the implication from left to right. Thereto, assume $L(R_1) \subseteq L(R_2)$. Let uwu' be an arbitrary string in $L(R_1)$ such that $w \in \$\bar{b}\$\Sigma_\$^+\$\bar{t}\$$ and $u, u' \in L(\#e\#e\#\cdots\#e\#)$. Hence, $uwu' \in L(R_2)$. Let m be a match such that $uwu' \models_m R_2$. Notice that uwu' contains $2k+2$ times the symbol "#". However, as uwu' starts and ends with the symbol "#", m matches the first and the last "#" of uwu' onto the $\Sigma_{\#,\$}^+$ sub-expressions of R_2. Thus $vwv' \models_m \#e_1\#e_2\#\cdots\#e_k\#$, for some suffix v of u and prefix v' of u'. In particular, $w \models_m e_i$, for some i. So, w does not encode a correct tiling. As the $\$\bar{b}\$\Sigma_\$^+\$\bar{t}\$$ defines all candidate tilings, the system D has no solution.

To show the implication from right to left, assume that there is a string $uwu' \in L(R_1)$ that is not in R_2, where $u, u' \in L(\#e\#e\# \cdots \#e\#)$. Then $w \notin \bigcup_{i=1}^{k} L(e_i)$ and, hence, w encodes a correct tiling.

The PSpace-hardness proof for $\mathrm{RE}(a, (+a)^*)$ is completely analogous, except that every "Σ^+" has to be replaced by a "Σ^*" and that

$$R_2 = \#\Sigma^*_{\#,\$}\#e_1\#e_2\# \cdots \#e_k\#\Sigma^*_{\#,\$}\#.$$

$\square$

Wim:
In the graph DB containment chapter, I use two-way NFAs. The proof actually needs that we can get rid of 2-way navigation and projection, using a single exponential blow-up.