

Databases

Thomas Weise (汤卫思)

February 20, 2026



Abstract

This book is an introduction into databases for undergraduate and graduate students.

Contents

Contents	i
1 Introduction	1
1.1 Features that we want from a Database	4
1.2 History	8
1.3 Software	13
1.4 Further Reading	15
I Getting Started	19
2 Downloading the Example Codes	23
3 Installing PostgreSQL	25
3.1 Installing PostgreSQL under Ubuntu Linux	25
3.2 Installing PostgreSQL under Microsoft Windows	31
4 Installing LibreOffice	45
4.1 Installing LibreOffice under Ubuntu Linux	46
4.2 Installing LibreOffice under Microsoft Windows	47
5 Installing Python, PyCharm, and Psycopg	56
5.1 Installing Psycopg	56
6 Installing yEd	64
6.1 Installing yEd on Ubuntu Linux	64
6.2 Installing yEd on Microsoft Windows	68
7 Installing PgModeler	72
7.1 Installing PgModeler under Ubuntu Linux	72
7.2 Installing PgModeler under Microsoft Windows	77
7.3 Installing PgModeler under MacOS	85
II A Simple Example: The Factory Database	87
8 Creating a User and the Database	90
8.1 Creating a User	90
8.2 Creating a new Database	96
9 Creating Tables and Filling them with Data	101
9.1 The Table “product”	101
9.2 The Table “customer”	113
9.3 The Table “demand”	120
10 Join-based Select and Views	125
10.1 Joining Tables	125
10.2 Views as Virtual Tables	130
10.3 Using our View	131

11 Updating and Deleting Records	134
11.1 Updating Records	134
11.2 Deleting Records	136
11.3 A Note on Compatibility	137
12 Connecting from Python	139
12.1 Reading Data from the Database	140
12.2 Inserting Data into the Database	141
12.3 Summary	144
13 Accessing the Database from LibreOffice Base	146
13.1 Connect to the Database	146
13.2 Adding Rows to a Table and Executing Views	149
13.3 Relationship Diagrams	151
13.4 Forms	153
13.5 Reports	164
14 Cleanup After the Example	171
15 Summary	174
III Database Design and Modeling	176
16 The Database Lifecycle	179
16.1 Classical Software Engineering Design Processes	180
16.2 Databases Design Processes	182
16.3 Summary	186
17 Requirements Analysis	187
17.1 Types of Requirements	187
17.2 Requirements Gathering	188
17.3 Requirements Specification Document	188
17.4 Example: Teaching Management Platform	189
18 Conceptual Model Design	193
18.1 Entities and Attributes	193
18.2 Keys	203
18.3 Relationships	206
18.4 Weak Entities	211
18.5 The Cardinality of Relationships	212
18.6 Compact Crow's Foot Notation	219
18.7 Data Model Selection	226
18.8 Summary	227
19 Logical Model Design	228
19.1 The Relational Data Model	228
19.2 Mapping Conceptual Models to Logical Models	231
19.3 Normalization	330
Backmatter	364
Best Practices	365
Useful Tools	367
Glossary	368
SQL Commands	377

Bibliography	379
---------------------	------------

Preface

The goal of the course and book is to teach undergraduate and graduate students the topic of **databases (DBs)**. Our focus is a practice-oriented approach to the topic. This means that each concept that we introduce or discuss is always accompanied by a rich set of examples. In the course we will use many tools, ranging from

- the **PostgreSQL** database management system (DBMS) [161, 306, 339, 433], over
- **yEd**, a graph editor that can be used for conceptual modeling [384, 497],
- **LibreOffice Base**, which can be used as convenient front end for creating forms and reports for data in a DB [160, 385],
- **Python** [482], a programming language which can connect to **PostgreSQL** using the **psycopg** module [473], to the
- **PgModeler**, a tool with which we can conveniently design logical **PostgreSQL** DB schemas [8].

After completing the course, you should be able to productively work with databases, at least at a beginner level. You should be able to realize simple database-based applications. And you should be able to navigate the huge ecosystem of different database management systems, tools, and paradigms in this field in order to pick the right tool for the right problem.

This book is intended to be read on an electronic device. Please do not print it. Help preserving the environment.

This book is work in progress. It will take years to be completed and I plan to keep improving and extending it for quite some time.

This book is freely available. You can download its newest version from <https://thomasweise.github.io/databases>. This version may change since this book is, well, work in progress.

The book consists of two types of material: Materials that the author (Thomas Weise) has created by himself and such that have been created by others. The vast majority of the material is teaching material created by the author. This and only this material is released under the *Attribution-NonCommercial-ShareAlike 4.0 International license* (CC BY-NC-SA 4.0). However, the book also includes some images and figures created by others, such as logos and screenshots and more, which are marked explicitly and licensed under their authors' terms. For example, all logos and trademarks are under the copyright of their respective owners.

You can cite this book [481], e.g., by using the following BibTeX:

```
1 @book{databases ,  
2   author = {Thomas Weise} ,  
3   title = {Databases} ,  
4   year = {2025--2026} ,  
5   publisher = {School of Artificial Intelligence and Big Data ,  
6               Hefei University} ,  
7   address = {Hefei, Anhui, China} ,  
8   url = {https://thomasweise.github.io/databases}  
9 }
```

The text of the book itself is also available in the repository <https://github.com/thomasWeise/databases>. There, you can also submit **issues**, such as change requests, suggestions, errors, typos, or you can inform me that something is unclear, so that I can improve the book. Such feedback is most

welcome. The book is written using L^AT_EX and this repository contains all the scripts, styles, graphics, and source files of the book (except the source files of the example programs).

Copyright © 2025-2026

Prof. Dr. Thomas Weise (汤卫思教授)
at the School of Artificial Intelligence and Big Data (人工智能与大数据学院)
of Hefei University (合肥大学),
in Hefei, Anhui, China (中国安徽省合肥市)

Contact me via email to tweise@hfuu.edu.cn with CC to tweise@ustc.edu.cn.



[book pdf]



[course website]

<https://thomasweise.github.io/databases>

This book was built using the following software:

```
1 Alpine Linux 3.23.2
2 Linux 6.11.0-1018-azure x86_64
3
4 python: 3.12.12
5 PostgreSQL client: 16.12
6 LibreOffice: 24.2.7.2 420(Build:2)
7 psycopg: 3.3.3
8 yEd: 3.25.1
9 java: openjdk 21.0.9 2025-10-21
10 pgModeler: 1.1.0~beta1-1build2.Debian Qt 6.4.2
11
12 texgit_py: 0.9.10
13 texgit_tex: 0.9.7
14 pycommons: 0.8.88
15 pdflatex: pdfTeX 3.141592653-2.6-1.40.28 (TeX Live 2025)
16 biber: 2.21
17 makeglossaries: 4.7 (2025-05-14)
18 ghostscript: 10.06.0 (2025-09-09)
19
20 date: 2026-02-20 06:45:48 +0000
```

Chapter 1

Introduction

Data is ubiquitously present. Names, addresses, bank accounts and transactions, online purchases, train tickets, mobile phone numbers, 微信 (WeChat) chats, websites, books, manuals, program source code, map data, highschool grades, health check results and medical histories, tax data, employment histories, game scores ... everything is data. Data is maybe one of the most important resources of our digital age. And data needs to be stored, sorted, retrieved, backed-up, aggregated, summarized, updated, and managed.

There are many different kinds of data. Since these kinds differ very much in their nature, the ways in which we store and retrieve them vary as well.

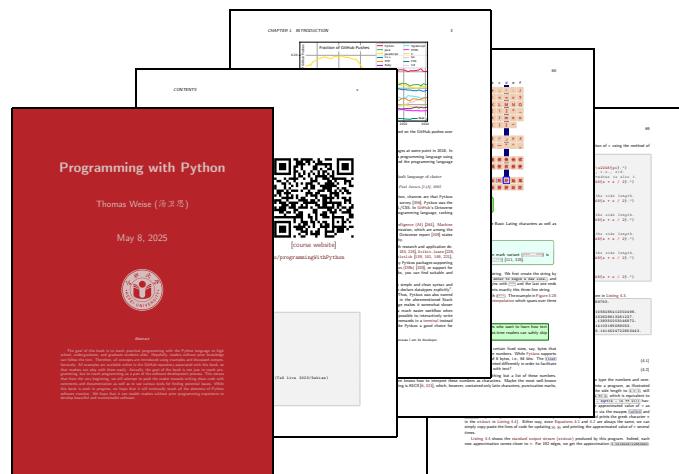


Figure 1.1: An example of an unstructured document data in form of some pages of the book *Programming with Python* [482].

A screenshot of LibreOffice Calc showing a spreadsheet with data in rows and columns. The data consists of various parameters and their values, such as 'algorithm', 'instance', 'objective', 'encoding', 'n', 'bestF_min', 'bestF_mean', 'bestF_std', and 'bestF_geom'. The table has 24 rows and 9 columns. The LibreOffice interface is visible at the top, including the menu bar and toolbars.

Figure 1.2: An example of tabular data, namely a comma-separated values (CSV) file opened and edited in LibreOffice Calc.

The figure consists of three side-by-side code editor windows. The left window shows XML code (xmlexample.xml) describing a course with a teacher and several students. The middle window shows JSON code (jsonexample.json) for the same data. The right window shows YAML code (yamlexample.yaml) for the same data.

```

1. xmlexample.xml
<?xml version='1.0' encoding='UTF-8'?>
<?xml-stylesheet type="text/xsl" href="xmlexample.xsl"?>
<course title="数据库原理与应用" year="2025">
  <teacher name="Weise" />
  <students>
    <student studentID="1234567890" name="Bibbo" score="85" />
    <student studentID="1234567891" name="Bebbo" score="73" />
    <student studentID="1234567892" name="Bibboto" score="98" />
    <student studentID="1234567893" name="Bibboha" score="97" />
  </students>
</course>

```

```

1. jsonexample.json
{
  "course": {
    "title": "数据库原理与应用",
    "year": 2025,
    "teacher": {
      "name": "Weise"
    },
    "students": [
      {
        "studentID": "1234567890",
        "name": "Bibbo",
        "score": 85
      },
      {
        "studentID": "1234567891",
        "name": "Bebbo",
        "score": 73
      },
      {
        "studentID": "1234567892",
        "name": "Bibboto",
        "score": 98
      },
      {
        "studentID": "1234567893",
        "name": "Bibboha",
        "score": 97
      }
    ]
  }
}


```

```

1. yamlexample.yaml
courses:
  - title: '数据库原理与应用'
    year: '2025'
    teacher:
      name: 'Weise'
    students:
      - studentID: '1234567890'
        name: 'Bibbo'
        score: '85'
      - studentID: '1234567891'
        name: 'Bebbo'
        score: '73'
      - studentID: '1234567892'
        name: 'Bibboto'
        score: '98'
      - studentID: '1234567893'
        name: 'Bibboha'
        score: '97'


```

(1.3.1) An example of data stored in the Extensible Markup Language (XML). (1.3.2) An example of data stored in JavaScript Object Notation (JSON). (1.3.3) An example of data stored in the YAML Ain't Markup Language™ (YAML).

Figure 1.3: Examples of the same dataset describing a course, its teachers, and the scores achieved by some of its student, presented in the dataformats XML, JSON, and YAML.

For example, there is unstructured data, like texts, graphics, or this book, as illustrated in Figure 1.1. Such unstructured pieces of data are usually stored in single document files. An organization, e.g., a company or a university, produces heaps of such documents. Every year, the students of our university write Bachelor's and Master's theses. The different schools of our university submit reports, presentations, budget plans, and so on. The university itself issues regulations and notices. There exist common structures for the different document types in our university, such as templates for theses. However, beyond such common structures, the data in the documents can vary enormously and there is no way to unify it. Therefore, we "store" such data as singular documents and maybe organize these documents in catalogs, e.g., by student ID, year, school, department, title, keyword, and so on. If we want to retrieve the data from the theses, we can maybe search them by title or keyword. Once we found the right document, we arrive at the end of what traditional technology can offer us and we have to read them. Nowadays, maybe we can generate a summary using an Artificial Intelligence (AI), but in the end, we still have unstructured information to digest.

Then, there is data that is tightly structured and self-contained. For data that can be stored in a single table, there exist simple text formats like comma-separated values (CSV) [391] and applications like Microsoft Excel [43, 187] and LibreOffice Calc [270, 385] tables are common. One example is given in Figure 1.2. If the data is more complex, maybe hierarchically structured, formats like Extensible Markup Language (XML) [53, 95, 251], JavaScript Object Notation (JSON) [52, 438], or YAML Ain't Markup Language™ (YAML) [94, 142, 251] may be more suitable. Examples of these formats are given in Figure 1.3.

All of these formats have in common that they store data in singular documents. They offer clear and strict rules how the data can be defined, structured, stored, and retrieved. They are open standards and vendor-independent. With the exception of Microsoft Excel tables, they are also text-based formats.¹ However, they are mainly suitable for data that, well, can be stored in single files efficiently. They are not suitable for manipulating huge datasets. They are also not suitable for modelling more complex relationships between different datasets.

For example, if we want to represent the personnel, schools, students, and assets of our university, as sketched in Figure 1.4, we could try to do that in a single XML document. This document would need to store which teacher belongs to which school, which student belongs to which school, which student is supervised by which teacher, and so on. This *can* be done. Maybe a single person could write such a document. The document will be *huge*. Finding something or changing something will be incredibly tedious. If an error occurs, maybe a misplaced character, maybe an unescaped quotation mark, then the whole document will no longer be valid. As soon as multiple people need to work on such a document together, everything will collapse. Clearly, another way to work with *relational* data is needed.

Welcome to the course book *Databases*. Here, we learn about exactly such a way. A way to deal with structured and relational data. A way in which multiple users can concurrently work on a large base of data while preserving data integrity. Before we begin to do that, let us clarify some simple definitions:

¹Today, Microsoft Excel tables are stored as compressed collections of XML documents.

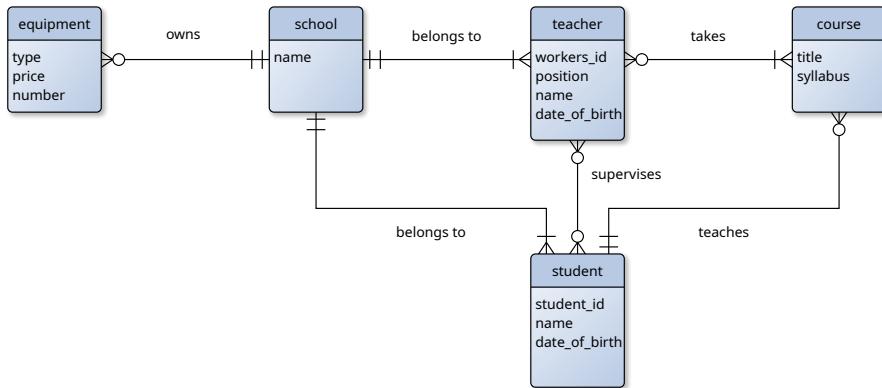


Figure 1.4: A diagram illustrating an example of the structure of data with more complex relations.

Definition 1.1: Database

A **database (DB)** is a computerized collection of interrelated stored data of potentially many types, maybe accessed by many users and applications concurrently.

There exist many types of **DBs**. There are DBs for documents, DBs for complex objects, and DBs geographical data. We, however, will mainly focus on DBs that store data in tables which may be coupled by relationships:

Definition 1.2: Relational Database

A **relational database** is a DB that organizes data into rows (tuples, records) and columns (attributes), which collectively form tables (relations) where the data points are related to each other [90, 194, 195, 408, 431, 487].

Definition 1.3: Record

A record is a group of related data items treated as a unit by an application program.

For example, a record with student information could store the student's name, **Date of Birth (DOB)**, ID number, and mobile phone number. Records are the basic units of data stored in relational databases. A second key element of relational databases are the relationships between records. These relationships guard the correctness and integrity of the data.

Before, we mentioned that a DB for our university could store “which student is supervised by which teacher.” This would mean that each student record can be linked to teacher records. Does it have to be linked to one? Maybe not, because when the student enrolls into the university, they may not yet have a supervisor. So maybe it is OK if a student record is not linked to any supervisor record. But if it is linked to a supervisor record, then it must be ensured that this record exists and is valid. Of course, it may also be permissible that a student record is linked to more than one supervisor record. Maybe the student has a primary supervisor and a secondary supervisor. As you can see, figuring out how the records can be related alone can already be a bit challenging.

So now we have arrived at the kind of data that we want to talk about. We talk about data that comprises entities of different types, say, student records, teacher records, course records, maybe even teaching room records, schedule elements, grade records, and so on. Between such records, relations exist that must not be violated. There could be arbitrarily many such records, record types, and relationships. Indeed, DBs usually grow over time. And there often are many people (or programs) who work with the DBs. It is easy to see that we will need some kind of software to manage all of that.

Definition 1.4: Database Management System

The **database management system (DBMS)** is a software system for manipulating DBs. It offers the ability to create, save, modify, and delete DBs, tables, and records, to add rows to tables, and so on. It also manages and controls the access rights to the DB.

DBMSes can be arbitrarily complex pieces of software. If you think about what things we would like to have when dealing with important data, it becomes immediately clear why.

First, there may be lots and lots of data and we want to access it. We want to find, add, remove, or change certain records. And we want to do this reasonably quickly. So first, a DBMS provides efficient access to the data. Thus, it needs to efficient and therefore complicated datastructures and access algorithms. If you have learned **Python** programming [482], then you will know that finding records in a sequential **list** is much slower than finding them in a hash table (a **dict**)..... but these data structures exist in the RAM of a computer, and DBMSes need to work with files on the disk. Therefore, the datastructures they implement become even more complicated.

Second, DBMSes also allow us to specify constraints on single data elements as well as the possible relationships between data. Maybe we only permit **DOBs** between January 1st, 1900 and December 31, 2020, for all students and university employees. Maybe we strictly want to enforce that each university employee record is assigned to one, exactly one, and only one school record. And they need enforce that the data never violates the constraints and relationships.

Third, they also need to offer a user management and access rights management. In a **DBs** for the **human resources department (HR)** in a big organization, for example, not every **HR** staff member is allowed to access (and much less to modify) all the data. The same holds for the applications that access the data and, maybe, present some of it on the website.

Fourth, a DBMSes also must take care to properly manage concurrent access, i.e., in situations where multiple users and multiple programs work on the same DB at the same time, it must preserve data integrity and consistency. It also has to offer functionality to create backups, i.e., copies of the DBs that are preserved in case of errors, hardware faults, or other situations that could destroy the original DBs. In some scenarios, DBs can become very huge, too big to be stored on a single computer. Today's DBMSes often allow us to partition the DBs and distribute them over a cluster of computers.

These are just some examples of the features that a DBMS must provide. We discuss more of them in [Section 1.1](#). From the fact that DBMSes need to take care of many complex tasks, it follows that using, configuring, and maintaining them over many years is probably challenging, too. Working with and maintaining DBs is a job that can require a lot of expertise. In many organizations, there are specific positions for DB experts:

Definition 1.5: Database Administrator

A **database administrator (DBA)** is the person or group responsible for the effective use of DB technology in an organization or enterprise.

And *many organizations* is an understatement. The HR departments for organizations like universities and companies use **relational databases** to manage the members of the organization. The financial departments keep track of salary payments, of budgets for projects, of funding, and benefits in relational databases. Asset management departments maintain lists of, well, assets, like computers, furniture, cars, tools, machines, expensive equipment, in relational databases. Banks manage accounts using relational databases. The government manages tax payments using relational databases. Relational databases store the information about the inventory and cashflow of online shops. Basically every organization with a certain size uses multiple relational databases. And therefore, many have **DBAs**.

1.1 Features that we want from a Database

So far, we developed some rough ideas about what a DB is. We developed these ideas by contrasting the requirements of organizations with the limitations of simple document files. To get maybe some further ideas of what we want to accomplish with **DBs**, let us reiterate what requirements organizations face when dealing with data. Let us work through this list of desired features by imagining that we

are building a DB for a bank. We can imagine that the DB is supposed to store customers, accounts, transactions, and bank employees. Without getting too specific, we can explore our expectations based on this idea. Our goal will be to compile some sort of wishlist of features.

1.1.1 Data Modelling and Representation

The most important capability that a **DBMS** needs to offer us is to create basic models, also called schemas, of our data. These models can be understood as analog to the datastructures or classes that imperative programming languages like **Python** offer to us. In Python, we can say: "This here is going to be a list of integer numbers." or "This is a class for students, each instance of it stores the name of a student and their national ID number." We also want something like this for **DBs**. So we need to be able to make models define what can be stored and how it can be stored. In other words, the DBMS must give us the tools into our hands to define and implement the structure of our data.

We here focus on the *relational* data model, whose basic primitives are tables, columns, the datatypes of columns, constraints, and relationships between different tables. The so-called *logical schema* for our DB (see Chapter 19) defines which tables exist, what the names and datatypes and constraints of their columns are, and how the different tables are related.

In our bank example, we would maybe want to specify that there should be a table with customer information, a table with bank employee information, a table for the accounts that exist, and a table with all the transactions. We can define which information we want to store about customers, say their name, ID number, mobile phone number, etc. This includes some basic validity constraints, e.g., names cannot be empty, ID numbers must follow the standard Chinese ID number (中国公民身份证号码) scheme [507], and so on. We must also be able to specify the relationships between these tables. For example, each bank account must be associated with exactly one customer, but one customer may have multiple accounts. Ideally, this can be done using a simple programming language.

But such a logical schema is only one of the aspects of data modelling that a DBMS must offer us. There actually exist several different levels of abstractions which are involved in this process. For example, it should be possible to separate this logical model from the way the data is actually organized on the disk, i.e., the *physical schema*. The DBMSes should *allow* the DB designers to, for instance, define indexes for tables to speed up the access. But it should *not require* the designer to deal with or to necessarily understand the layout of the actual datastructures or the files on the disk.

1.1.2 Data Independence

The fact that the logical schema and a physical schema and data layout are different things is interesting. When we think about this, we naturally arrive at the concept of *data independence*. When we design a DB-based application, there will be different levels of abstraction which should be as independent from each other as possible.

This is actually also similar to programming: If you write a program in Python, then you are using high-level instructions and concepts, like loops, functions, classes, exceptions, and so on. All of these concepts must somehow be realized in machine code. If you ever learned about assembly language or actual machine code for CPUs, you know that this looks entirely different. Also, in Python, you have datatypes like text strings which simply do not exist on the level of machine code and are instead represented as blocks of bytes in memory. A Python programmer does not need to know or understand this. But if they do, then they can probably become more efficient in their coding.

The application programs that access the data based on the logical schema should be independent from the physical organization of the data on the disk. It should not really matter to them whether records are stored or indexed in a hash, a B-tree, or as simple sequences in a file. If we create an easy-to-use application as interface for our banking DB, this interface would interact with the logical view of the data via the DBMS. How the data is actually stored does not matter to our application. This is called *physical data independence*.

Rarely there exists only one single program that accesses a large DB. In our banking example, let's say that we create two applications: We create an application for the bank employees to manage accounts and customers. We also create a web application through which the customers can log into their bank accounts and make transfers. Both applications can have a different view on the data. The first application may offer much more information and functionality only accessible to the bank employees. The view on the data available second application may naturally be much more restrictive.

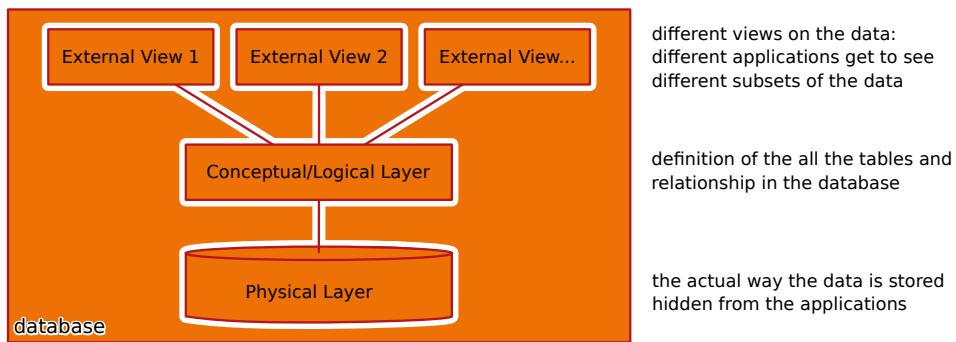


Figure 1.5: The Three-Schema Architecture of databases (DBs) [7, 61, 380, 453].

It should work independently from the information outside of this view. If the structure of these information (that it cannot see) is changed later, then this should have no impact on that program. This is called *logical data independence*.

This means that a **DBMSes** should support designing DBs in the so-called three-schema architecture [7, 61, 380, 453] illustrated in Figure 1.5: It should be possible to provide different views on the data for different users and applications. At the bottom layer of the DB is the physical model managing how data is stored. On top of that sits the conceptual (or logical) layer, where the structure of the tables and the relationships between them are defined. On the highest level, different views on the data can be provided for different applications.

In comparison, **Microsoft Excel** and **LibreOffice Calc** also offer us the ability to work with tables. However, we cannot *a priori* define a model or structure of the data that we want to store in tables. While we can create some relations between cells of different tables, enforcing relationships on new rows that we are going to add is not possible. While we can format cells and columns to emulate certain datatypes, we cannot strictly prevent that the user enters invalid data. So these applications do not offer us any modeling capability on the logical level. We have no influence on the way the data is stored and we cannot improve the performance by telling the applications to index the data in any reasonable way. So there are no physical modeling capabilities either. Finally, any person who would use a Microsoft Excel or LibreOffice Calc table as backend for different applications that have different views on data shall be stripped of their degree and credentials as a software developer. Spreadsheets therefore obviously do not have any schema design capabilities at all and most certainly do not support anything resembling a three-schema architecture. A DBMS therefore needs to offer us data modeling tools that are very far beyond spreadsheets.

1.1.3 Data Availability and Performance

The DBMS makes a collection of data available to users and applications in a meaningful format at a reasonable performance. We expect that the speed of reading, writing, changing, deleting, and adding of data to the DB be high. A simple and reasonably abstract programming language should be provided for querying and updating the DB. There should also be tools available that further simplify the modeling, editing, and sorting of data as well as the generation of reports from data.

1.1.4 Data Integrity

The correctness and validity of the data is ensured. This includes several aspects: First, the modeled relationships must be enforced. If it is stated each bank account is associated with exactly one customer, then there must never be a bank account record that is not associated with an existing customer. Also, there can never be two bank accounts with the same account ID. This property is called *Consistency*.

Let's say that for each account, we store the current balance. If a customer deposits money, then we can simply add the amount to their account's balance. This is one operation where not much can go wrong. However, if money is transferred from one account of our bank to another account, then this "looks like" two operations. But it actually is a single *transaction*. A DB should be consistent before and after each transaction.

Atomicity means that such a transaction will either complete successfully in its entirety or fail. If the transaction is completed successfully, i.e., *commits*, then all the changes become visible. If the

transaction is aborted, no change is performed. No transaction can complete partially, it is all-or-nothing. This means that either the money is correctly subtracted from one account and added to another or nothing happens. It must never happen that money is subtracted from one account but does not arrive in another due to an intermediate error.

1.1.5 Concurrency Support and Isolation

Closely related to data integrity is the support for concurrency. Multiple users and multiple processes can access and modify the DB at the same time. The DBMS preserves the integrity of the DB and of all the views that the users and process have on the data.

Different concurrent updates to the same account must take place atomically and in isolation. This means that if money is transferred from account A to account B while simultaneously money is transferred from account B to account C , the two transactions should not influence each other. Both transactions should be isolated. No money can be lost or appear from thin air.

Another aspect of concurrency is that in some cases, we want to distribute databases over cluster of computers. Maybe the databases are too big. Maybe we want to increase the performance further by dividing the data into several sets which are almost independent. Either way, an ideal DBMS should offer us the functionality for doing that.

1.1.6 Durability and Data Safety

A basic feature that any DB needs to support is *Durability*: Once a transaction is committed, i.e., successfully completed, its changes are permanent. The changes caused by the transaction are saved to the DB permanently.

The DBMS has to ensure that data can be preserved in case of unforeseen situations. Indeed, durability extends also to system crashes and failures. This includes the support of checkpoints and recovery after restarts.

Of course, there are limits to this: At some lower layers, DBs are stored as files in the filesystem. If the underlying hard disk fails, these files can be destroyed. There is no magical way a DBMS can guard against this. No system can be entirely safe from hard disk failures and crashes.

Therefore, DBMSes need to offer methods to back up the data, i.e., to create copies of the DBs and store them on other devices. Then, as long as reasonable backup strategies are employed, it must be possible to re-create the data after the original DB was lost. It is clear that for a bank, the data about its and its customers' accounts is its business. If that data is lost, the bank is lost. So ensuring that the DB is robust and that it can be recreated from backups is of paramount importance.

The features Atomicity, Consistency, /solation, and Durability together are often called *ACID* properties [183, 480].

1.1.7 Data Privacy and Security

The DBMS must enable the protection of data against access from unauthorized access from both within and outside of the organization. It must be possible to specify roles for users and processes that define which data they can access or modify. For example, the DBA of the bank's DB needs to be able to modify the structure of the DB, maybe add and modify tables. A normal bank employee should not be able to do that. But they may be allowed to enter new records into certain tables of the DB, for instance, to add customers or to create new accounts. Thinking this through even further, a bank employee may be assigned as manager to some customer's accounts. The employees should only be able to see the information in these accounts. All of this can be done by assigning roles to users and processes and access rights to roles.

The aforementioned three-schema architecture goes hand-in-hand with security measures. At the highest level, the applications, users can only see the data that they need to be able to see and nothing more. This is ensured by the different views that are provided for them.

Additionally, all access to the DB should be password protected. Communication with the DB happens over encrypted connections.

A DBMS should also support accounting and an audit trail. It must be possible to log information regarding which user changed which dataset and when. This is required to pass certain government, financial, or ISO certification audits.

1.1.8 Summary

It is quite obvious that the basic filesystem only offers very few of the properties discussed above. Using simple data formats like [CSV](#) or even [Microsoft Excel](#) tables will not be a good choice either. While one might realize concurrency by placing multiple interrelated such documents on a shared folder, there certainly would not be any *ACID* guarantees. Multiple people editing them at the same time will wreak havoc. Structured formats like [JSON](#), [XML](#), or [YAML](#) would allow us to represent complex data in single documents, but this would make the multi-user access situation even worse. And none of these solutions could provide any means to protect data integrity, let alone access control.

Instead, we require a software layer between the [DB](#) and the user processes. Honestly, we do not even really care of this software layer actually stores the data, how the data is organized internally, or how *ACID* is implemented. As long as this software, the [DBMS](#), offers us the above features, we will be happy.

1.2 History

We will now take a brief look at the history of DBs [1, 166, 278, 399, 496]. By understanding the history of DBMSes, we can better understand the current behavior and features of DBs.

People began storing and processing information a very long time ago. A driving force to collect and analyze data must have been the management of limited resources. Some of the earliest discovered writings are Sumerian accounting and tax records on clay tablets from Mesopotamia, dating back four thousand years [112, 250, 456], as illustrated in Figure 1.6. The collection and analysis of information never stopped from then onwards.

When much data is stored, three big questions emerge: *How do we store the information?*, *How can I find a specific record?*, and *How can we condense and extract representative information from all of this data?*

Examples for the data organization include the Dewey Decimal Classification system for organizing books in a library, which emerged in the 1870s [75, 229]. This system and similar systems as the one illustrated in Figure 1.7 are still in use today. Not much later, we enter the earliest stage of data processing with machines. Data storage happened by using physical tools [199]. The punched cards system by Hollerith, patented in the late 1880s [208, 209], was used in the 1890s US Census. The automatic processing of these cards allowed the census to finish under budget and ahead of time [437]. Hollerith's Tabulating Machine Company eventually merged with three other companies into International Business Machines (IBM). Punch cards similar to the latter model shown in Figure 1.8.2 made up 20% of the revenue of IBM as late as the mid-1950s [437].

In addition to punch cards, reels of punched tape emerged as data storages and later magnetic tapes. The way data can be retrieved depends on how the data is stored. Punch cards can be sorted, stacked, and otherwise cleverly be arranged to access information quickly. Tape-based storages



Figure 1.6: Several clay tablets dating back as far as 2200 before Common Era (BCE) from the collection *Cuneiform Tablets: From the Reign of Gudea of Lagash to Shalmanassar III* [112].

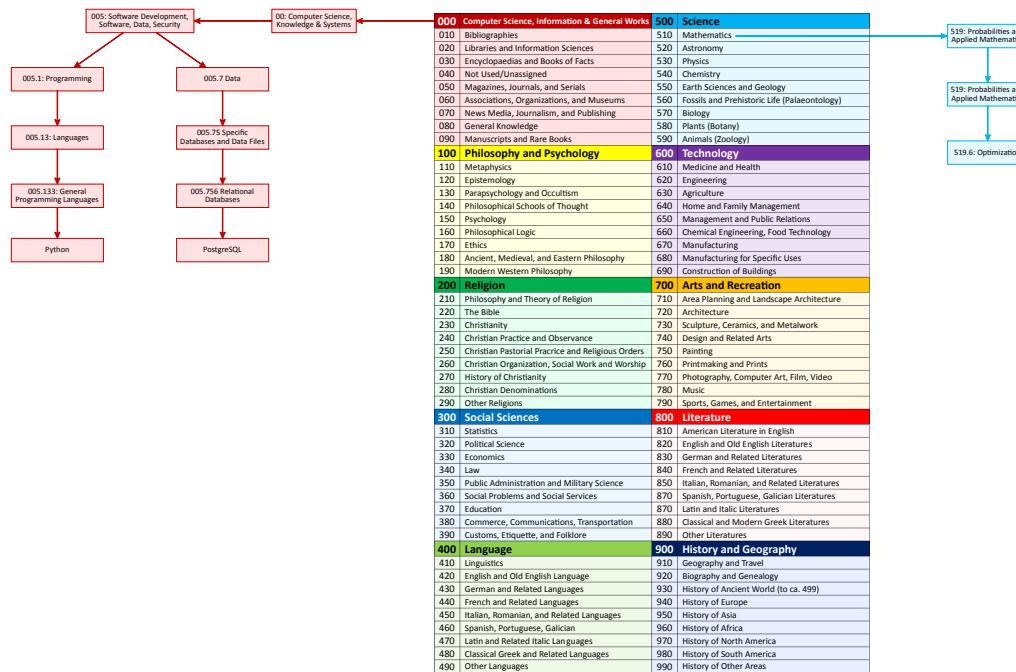
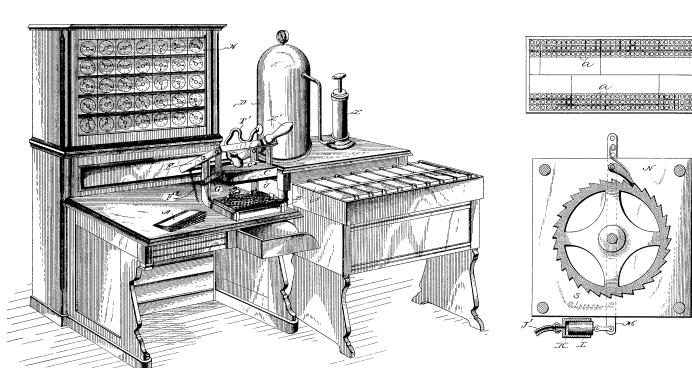
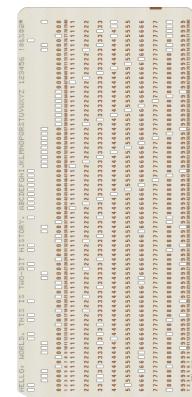


Figure 1.7: An illustration of the Melvil Decimal System [290], a free variant of the Dewey Decimal System named after its inventor Melvil Dewey, and the steps needed to locate books on Python, PostgreSQL, or optimization.



(1.8.1) Sketches of Hollerith's tabulator machine that used punched cards from his 1892 patent [209].



(1.8.2) An IBM 029 punched card. Source: [429], licensed under CC BY-SA 4.0.

Figure 1.8: A very early patented machine using punched cards and an IBM 029 punched card, which was available in the 1960s to 1980s.

requires us to sequentially spool through the tape to find the data we are looking for. Either way, efficient organization of data and media became more and more important. In 1958 the Electronic Recording Machine Accounting (ERMA) Mark 1 was developed as automated system for organizing banking records [24]. It had features of file system, although the term “files” was meant literally – they organized paper-based documents, using a structure reminiscent of the aforementioned hierarchical, number-based Dewey categorization method.

In 1956, the IBM 305 RAMAC came out: the first to use a random-access disk drive [351] as shown in Figure 1.9. It could store five to ten megabytes. Its true innovation was that from now on, it was no longer necessary to store and access data in a sequential order [72]. Instead, the data on its disks could be accessed in any order (hence the name *random-access disk drive*).

The first file systems for computers appeared in the 1960s. The Atlas system in the UK had offered a rudimentary file system functionality already in 1961 [243, 282], as sketched in Figure 1.10. The Operating System (OS) Compatible Time-Sharing System (CTSS) [102] at the MIT had a flat

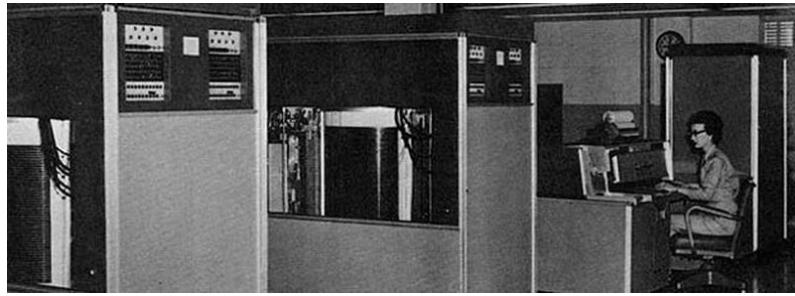


Figure 1.9: Image from 1956: An IBM 305 RAMAC (right) with two of the (at that time) very new IBM 350 hard disks (middle and left). Source: [176].

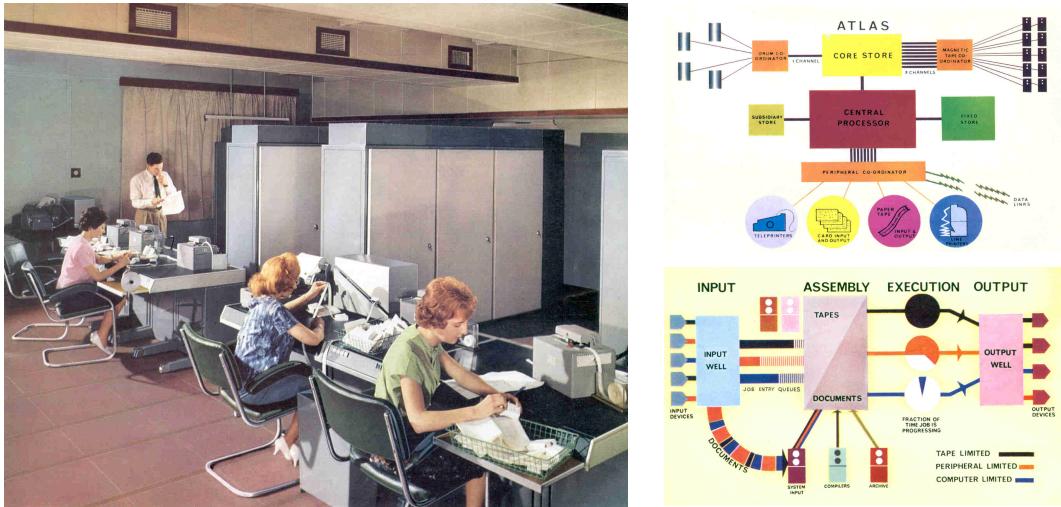


Figure 1.10: Images from the “Ferranti Computing Systems Atlas 1 Brochure: 1962” [282]. © UKRI Science and Technology Facilities Council, available from <https://www.chilton-computing.org.uk>.

```

0238 vnc
0238 as sc_admin_command: Utility.SysDemon.z: delete_old_pdds
0238 ut send_admin_command: Execution started ...
0238 ut completed.
0238 ut Records Left N VIDEOS Left N PB/PD LV Name
0238 ut 166388 98764 59 4226 34207 81 pb root
0238 ut r 0238 13.886 497
0238 ut
0238 ut
0238.1 RCP: Attached tape_00 for Utility.SysDemon.z
0238.1 RCP: Attached rdr_00 for Utility.SysDemon.z
0238.1 RCP: Attached rdr_01 for Utility.SysDemon.z
0238.1 RCP: Attached pma for Utility.SysDemon.z
0238.1 RCP: Attached pma from Utility.SysDemon.z
0238.1 RCP: Detached pma from Utility.SysDemon.z
0238.1 RCP: Detached pma from Utility.SysDemon.z
0238.1 RCP: Detached pma from Utility.SysDemon.z
V= CONSOLE: RELEASED
M= |

0238 Port (d.0000,d.0001,d.0002,d.0003,d.0004,d.0005,d.0006,d.0007,d.0008,d.0009,
d.0010,d.0011,d.0012,d.0013,d.0014,d.0015,d.0016,d.0017,d.0018,d.0019,d.0020,d.
0021,d.0022,d.0023,d.0024,d.0025,d.0026,d.0027,d.0028,d.0029,d.0030,d.0031)
Attached to line = b.0000

Multics MR12.7 Installation and location (channel d.0000)
Nodes = 1, out of 90.0 units; users = 5, 03/19/23 0241.5 put Sun
login Repair cswp

password:
New Password:
New Password again:
password changed.
You are protected from preemption.
RepairAdmin logged in 03/19/23 0241.7 put Sun from ASCII terminal "none".
RepairAdmin logged in 03/19/23 0241.7 put Sun from ASCII terminal "none".
new message in message_of_the_day:
welcome to the Multics System.

print_mode: Created /user_dir.dir/SysAdmin:Repair:Repair.value.
r 0241 3.461 32

```

```

r 06134 2.806 276
ls -a
Segments = 4, Lengths = 4.
r v 1 qedxdoc.txt
r v 1 repair.history
r v 1 Repair.histlock
r w 1 Repair.vacuum
r 00154 0.363 6

```

Figure 1.11: Some screenshots of the terminal of Multics MR12.7 taken from [70], licensed under CC BY-SA 4.0.

filesystem only one or two years later [312]. The hierarchical file system for the *Multiplexed Information and Computing Service* (Multics) OS [70, 103], published in 1965, already had surprisingly many advanced features that we know from today’s file systems: fine-grained access control for data privacy, backup ability, links, and IO queue management. Inheriting from CTSS, it itself became the ancestor of Unix which, in turn, inspired Linux. The `ls` command shown in Figure 1.11 also was a feature of Multics (adapted from CTSS) and has survived all those years [164]. File systems are very good for organizing documents and heterogeneous data. They are not very suitable to main the sort of relational data and to achieve the features that would like DBs to have.

The need for systems that supported modern DB features became apparent. At the same time, it was not really clear how that could be done. Different groups began developing concepts, ideas, and prototypes.

The first version of the **Integrated Data Store (IDS)** was developed by Bachman in 1961/62 at General Electric [14, 15]. IDS offered the first direct access DB, holding data in virtual memory. It may have been the first real DBMS and Bachman won the 1973 A.M. Turing Award for this work [193]. IDS

was based on a network model, further advancing data management by structuring complex relationships in data. The programmer acted as navigator through the data. The idea to step-by-step navigate through data, updating it if needed, is more complicated and slower compared to the relational approach of today's DBs. IDS and similar systems encode elements of the views of the data as part of the DB structure, which make IDS less flexible. The Database Task Group of the Conference on Data Systems Languages (CODASYL) [432], a standardization body for the data processing industry, took over many of Bachman's ideas for IDS in the late 1960s. CODASYL is best known for its creation of the COBOL programming language [193].

Only slightly after IDS, another DBMS that could handle the structured storage of records obeying datatype constraints appeared. IBM developed the **Information Management System (IMS)** for the Apollo space program [246]. The system was launched 1965/1967 [33]. IMS is still sold as a product and still exists today. Like IDS, it was not a DBMS for *relational databases*. Instead, it offered hierarchically structured records. This system, too, had a set of disadvantages [254]: If your data is not structured hierarchically, some records may need to be duplicated. For example, if we have a DB that assigns students to courses, then a student would appear in each of the records of the courses that she attends. The programming language offered by the IMS for the data access is also relatively low-level and, for example, the search strategy has to be implemented. Finally, changes to the logical schema will require us to perform cascading changes in the code accessing them.

Both IDS and IMS offered a strikingly new concept: The application code and the code for physical data storage and retrieval were separated. Between them, the Data Language One (DL/I) was located as interface in IMS. IDS offered the Data Description Language (DDL) to define types of logical records and the Data Manipulation Language (DML) to manipulate and navigate the data. The DBMS controls how the data is stored and loaded. Application programs can navigate through the data using the much simpler interface languages. As we briefly mentioned before: DBMSes can become arbitrarily complicated pieces of software, and one major part of this is the code to efficiently store and retrieve data.

Just think about it: All data is eventually mapped/stored to sequential files on a hard disk or other storage medium. You must be able to add new records and delete old records. You must also be able to find them. This alone is not easy to implement, but all of that has to be efficient, so (today) one would probably like to use datastructures like hashes, trees, and sorted lists ... all of which are actually located in sequential files. Before IDS and IMS, this had to be part of your application's code. Actually, it had to be part of the code of *all* applications that accessed the data, if multiple such applications exist [33]. Now this is part of IDS and IMS, and the user can ignore this complexity and just use a simple programming language where she can define which record to load, change, delete, or add.

Future users of large data banks must be protected from having to know how the data is organized in the machine (the internal representation).

— Edgar Frank “Ted” Codd [90], 1970

In 1970, the seminal paper “*A Relational Model of Data for Large Shared Data Banks*” by Codd appeared, starting with the above quote. He noticed the shortcomings of the IDS network model, namely that the programmers accessing the data still need a lot of information about how the data is actually represented and organized internally. Even Bachman himself mentioned that issues with IDS arose when users did not consider how data is internally sorted [15].

Codd wanted to protect programmers against such errors. He models data as a relational view by using tables, that he calls *relations*. Each column has a type that defines its permitted values. Each row must be distinct. One set of columns per table (usually a single column), the *primary key*, is used to uniquely identify the row. Rows in one table can reference other rows of either the same table or another table. Therefore, some column(s) of the table (called *foreign key*) then store the primary key of the row to be reference. The data in the DB is then a collection of tables. He further describes a set of operations that can work on the tables to extract information, which we will discuss later on. By the end of the 1970s, over a dozen DBMSes had been implemented based on these new concepts [244]. Codd won the A.M. Turing Award for his work.

The *semantic* division between the physical storage representation of data and the logical layout and access was a very important step in the development of DBs. This separation also became a *physical* one: Computer networks as distributed systems began to emerge in the 1960s [259]. Baran had the idea of routing messages over multiple network switches in 1960 [18–22]. In 1961, Kleinrock proposed packet switched networks, where messages are split into multiple packages that each individually travel

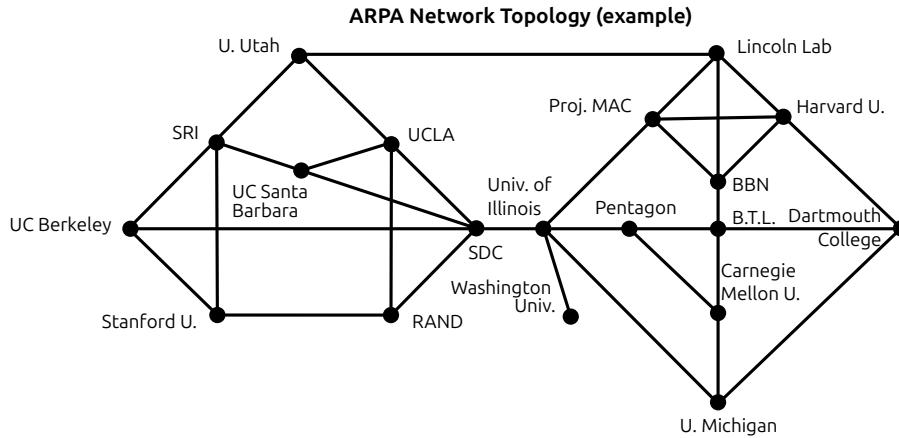


Figure 1.12: 19 node example of the ARPANET structure in the Request for Quotations shipped to vendors in 1968 [248].

```

1  SELECT name, student_id FROM table_students
2    WHERE date_of_birth >= '2000-01-01';

```

Figure 1.13: An example of an Structured Query Language (SQL) query.

their corresponding optimal paths [249]. This allows multiple users in a network to share the same data path at the same time. This idea of packet switching was also independently developed by Baran [20] and Davies [125–127].

Licklider proposed standardizing computer and networking languages to increase interoperability and to allow researchers to build upon each other's work in 1963 [271]. This could be considered as the first description of the idea of the internet. In 1965/66, the TX-2 computer at MIT Lincoln Lab in Massachusetts and the Q-32 in Santa Monica, California were connected, forming the first wide-area network [364]. In 1967, ARPANET, the precursor of the internet was conceived by the Advanced Research Projects Agency as a network connecting 35 computers at 16 sites in the USA [259, 364]. Figure 1.12 shows an 19 node example structure of the ARPANET shipped to vendors in 1968 [248]. In 1969, the first four nodes of the network became operational and in 1972, ARPANET was demonstrated publicly [259, 364].

Canaday, Harrison, Ivie, Ryder, and Wehr proposed putting a DBMS on a dedicated back-end computer in 1974 [63]. Applications and users would access the DBMS from another computer via a communication link. The first Ethernet was developed at the Xerox Palo Alto Research Center (PARC) in 1976 by Metcalfe and Boggs [99], TCP/IP became deployed 1983, and in the 1990s, the internet took off [259]. The term **clients** was coined in 1978 by Israel, Mitchell, and Sturgis [231]. From now on, the notation of the **client-server architecture** [37, 272, 313, 354, 363], in which most DBMSes are implemented today, was formalized. From then on, the development was unstoppable.

Another very important step forward for DBMSes was the SEQUEL language developed by Chamberlin and Boyce in 1974 [74] as part of the IBM System R project. The language was based on Codd's relational algebra, but wrapped it into an easy-to-understand language for data queries. Two years later, it was extended by more features, such as methods for inserting, deleting, and updating records [73]. In 1977, the SEQUEL name was shortened to **SQL**, an acronym for **Structured Query Language (SQL)** [72]. An example of an SQL query is shown in Figure 1.13. At the same time where SEQUEL was developed, the Interactive Graphics and Retrieval System (INGRES) group at the University of Berkeley in California, too, conducted research on **DBs** [421]. The open environment of publishing latest research and collaboration between the researchers contributed significantly to their success. They jointly received the ACM Software System Award in 1988 [72].

At about the same time where SQL emerged, better abstractions and tools for the design of **relational databases** began appearing. **Entity relationship diagrams (ERDs)** like Figure 1.14 are charts that allow us to model the relationship between different objects in a DB [80, 240]. They were proposed in 1975 by Chen [81] and became part of a US ANSI standard in the late 1980s [179, 321]. Chen's work extended the data structure diagrams introduced by Bachman [13] in 1969 and provided better ways

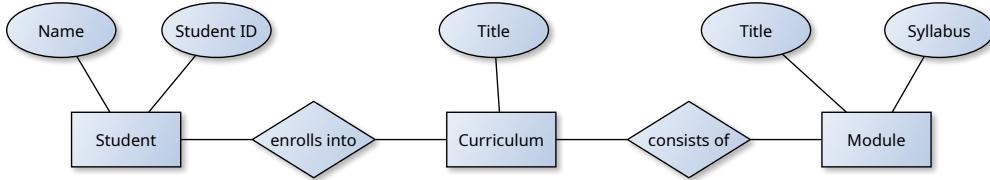


Figure 1.14: A simple example of an entity relationship diagram (ERD).

to model attributes and relationships. Chen was also inspired by his Chinese cultural heritage when developing ERDs [78, 79]:

What does the Chinese character construction principles have to do with ER modeling? The answer is: both Chinese characters and the ER model are trying to model the world – trying to use graphics to represent the entities in the real world. Therefore, there should be some similarities in their constructs.

— Peter Pin-Shan Chen [78], 1997

The **Oracle Database**, the first commercial **SQL**-based product, was released in 1979 [72] by the company Software Development Laboratories (SDL), which later renamed itself to Oracle [311]. It was an immediate success, because it was portable and could run cheaper hardware. IBM released its SQL DBMS **DB2** in 1983 [72, 84, 192]. SQL became a US ANSI standard in 1986 and an international ISO standard in 1987 [117, 131]. The standard continues to evolve, with its latest version being released in 2023 [222].

In the 1990s, several open source implementations of DBMSes became available for free [72]. We will discuss them together with some commercial **DBs** in the next section.

1.3 Software

A wide range of DBMSes for **relational databases** exist. The *DB-Engines Ranking of Relational DBMS (June 2025)* [129] lists 166 products. The “Stack Overflow 2024 Developer Survey” asked the opinions of developers on 35 systems [417]. A selection of the mentioned systems in these yearly

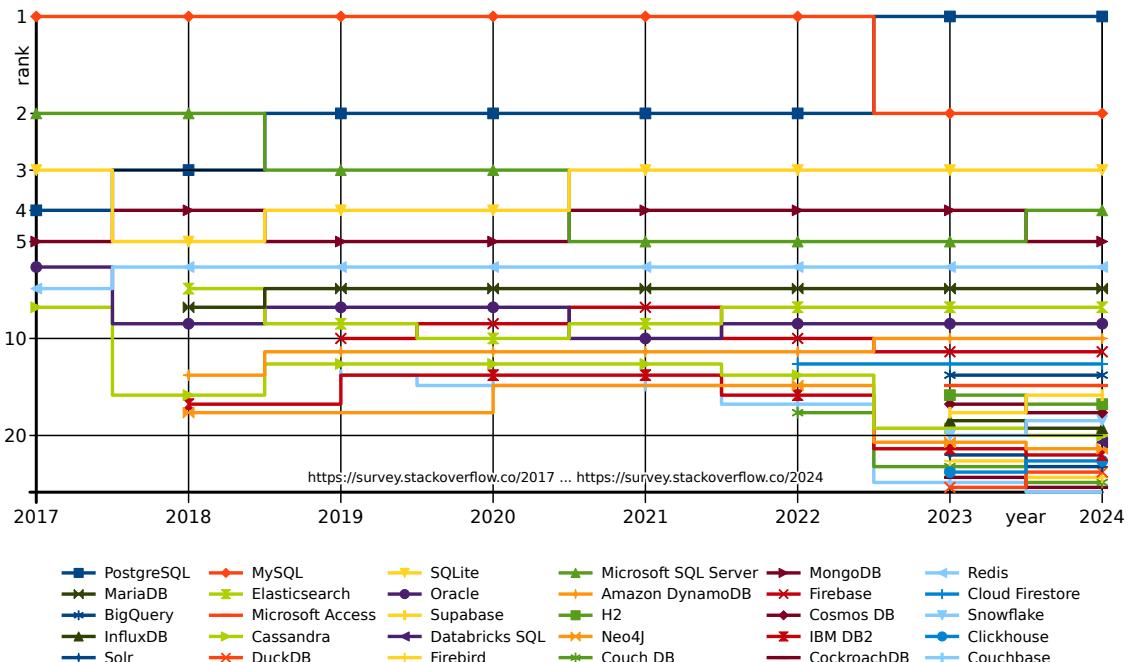


Figure 1.15: A chart of a selection of the DBMSes with which developers have worked, according to the StackOverflow Developer Surveys from 2017 to 2024 [417]. Not all of them are relational DBMSes.



Figure 1.16: The logos of several important OSSes / DBMSes.

surveys since 2017 is illustrated in [Figure 1.15](#). A few of these popular DBMSes were already mentioned in our history section. We can distinguish two types of DB products:

- **Open source software (OSS)** is software whose source code is made available for free, usually through the internet. Such software is often maintained by groups of volunteers, sometimes with financial support of enterprises or governmental grants. The software does not cost money. Open source projects are often hosted on collaborative platforms like [GitHub](#) and manage the evolution of their code via [Version Control Systemss \(VCSes\)](#) like [Git](#). According to Hoffmann, Nagle, and Zhou the value of OSS exceeds eight trillion dollars in 2024 [206].
- Proprietary / commercial software is usually closed-source and developed by an enterprise which sells it as product. These products are either sold on a per-version basis or via maintenance contracts. The enterprises often offer good service and advanced support, however it is usually not easy to change vendors and future pricing developments are hard to predict.

We here will discuss a small selection of products from either family, which, however, must remain brief and incomplete.

1.3.1 Open Source Relational Database Management Systems

A very big chunk of the [relational databases](#) is managed by open source DBMSes. These systems do not cost money and their source code is readily available in the internet. Large communities exist around them that can offer all kinds of advice. This makes them very attractive for developers and users. Several of these systems are around since the 1990s. In the early 2000s, they took off and began being used in more and more applications and systems [325]. By now, they are mature and well-tested products [72]. [Figure 1.16](#) shows some of their logos.

[MySQL](#) is such an open source DBMS for relational databases [49, 149, 358, 433, 489]. It was originally developed by Michael Widenius and David Axmark at the Swedish company MySQL AB and released in 1996 [72]. It soon gained widespread use as part of the [LAMP Stack](#), i.e., a system setup for web applications based on the [Linux OS](#), the Apache web server, the MySQL database, and the server-side scripting language PHP [64, 201]. Probably as the result of this, it ranked first in the Stack Overflow Developer Surveys in the years 2017 to 2022 [417]. In the recent survey [315] of the code of 371 Java open source projects on GitHub, MySQL was found to be the most-used relational DBMS. MySQL AB was acquired by Sun Microsystems in 2008, which in turn was acquired by Oracle in 2010 [72]. MySQL stayed open source.

After the acquisition by Oracle, some of the original developers of MySQL, including Michael Widenius, created a fork of MySQL: MariaDB [11, 12, 26, 149, 355]. They promise that MariaDB will stay open source forever. In [315], MariaDB was already the fifth most popular relational DBMS.

[PostgreSQL](#) [161, 306, 339, 433] is an object-relational DBMS, meaning that it supports concepts from [Object-Oriented Programming \(OOP\)](#), such as inheritance relationships between tables. It also supports many additional types like [JSON](#) objects and geometric types. [PostgreSQL](#) emerged from the POSTGRES project, the successor of the INGRES project at the University of Berkeley in California [72].

It may be the most fully-featured of the existing open source **SQL database systems (DBSes)**. In the “Stack Overflow 2024 Developer Survey”, it was the most popular DBMS [417] and in the open source code survey [315], it ranked second.

The **SQL DB** with the widest distribution is **SQLite** [72, 170, 203, 491]. It breaks with the common approach to offer access to the DB following a **client-server architecture**. Instead, it is directly loaded as a library in the process that uses. Today, it is installed on nearly every smartphone, computer, web browser, television, and automobile [72, 170, 491]. It was first released in 2000 and its core designer Richard Hipp received the SIGMOD Systems Award in 2017 [72].

The concept of **DBMSes** that can work on stand-alone files is also implemented in the open source office suite **LibreOffice** [171, 270, 385]: **LibreOffice Base** [160, 385]. LibreOffice Base is more than a DBMS working on single file. Instead, it offers a powerful user interface that can also connect to databases such as **MySQL**, **MariaDB** and **PostgreSQL**. The user interface allows you to conveniently design tables, views, queries, forms, and reports for the DBs it connects to. LibreOffice Base is a free alternative to the commercial product **Microsoft Access** [31, 86, 457], which offers a similar functionality.

Many of the examples used in this book will be implemented based on **PostgreSQL** and we will also play around with LibreOffice Base.

1.3.2 Commercial Relational Database Management Systems

There are many powerful commercial DBMSes. Different from the open source solutions, they do cost money and their source code is closed, i.e., the user cannot access it and only has an installer and program binaries to work with.

As already discussed, the first commercial SQL-based product was the **Oracle Database** [72, 311]. This product is still one of the most successful and widely used commercial DBs with many advanced features [32, 256]. In June 2025, it ranked first as most popular DBMS in [129] and it was the fourth most popular DBMS in [315]. How to migrate an Oracle Database to **PostgreSQL** is discussed in [258].

The third-ranking DBMS in [129] was the **Microsoft SQL Server** [6, 328, 488]. This DBMS dates its roots back to 1988, when Microsoft, Ashton-Tate, and Sybase collaborated to create a variant of Sybase SQL Server for the **OS IBM OS/2** [436]. The first version of this new DB server was released in 1989. Later versions were released for **Microsoft Windows NT**. Today, like Oracle Database, it also runs on **Linux**. With **Microsoft Access** [261], Microsoft also offers a DBMS mainly used on single computers by single users. It is an incredibly useful and convenient tool, which combines the features of a **relational database** with advanced form design and reporting features [31, 86, 288, 457]. This system is probably the role model after which LibreOffice Base was shaped. Most things that can be done with LibreOffice Base that we will explore can just as well be done with Microsoft Access, in many cases even more conveniently (but LibreOffice Base is free...).

IBM's DB2 [17, 84] is another early relational DBMS [192]. It emerged as a software for powerful mainframe computers that IBM manufactured themselves. Mainframes are very powerful central computers, often used by large organisations and businesses. While Oracle focussed on providing DBs for normal computers, IBM focussed on powerful central **servers**. By 1989, half of all mainframe customers had DB2 installed [192]. DB2 still ranked sixth in the *DB-Engines Ranking of Relational DBMS (June 2025)* [129] and 23rd in the “Stack Overflow 2024 Developer Survey” [417].

Of course, we here can only give a very brief and very incomplete overview on the different open source and commerical relational DBMSes on the market. The vast number of such systems and the fact that they are core cash cows for huge companies such as Microsoft, Oracle, and IBM underline the importance of this field. In our journey, we will use the free **PostgreSQL** as platform to experiment with DBs, which makes sense because it is the top-ranking DBMSes in [417], ranks fourth in [129], and is, well, free.

1.4 Further Reading

The author of this book is not really a DB buff. I did several DB projects, some for hobby an some even in commercial use, but neither of them had a really large scale. And most of the things I did were self-taught, mainly through technical manuals and documentation, complemented by classes I took at the university. This book is an attempt to create a resource that can help students to learn about the field of DBs on a practical level, with links to theory.

There are many resources, internet websites as well as physical books, that are of very high quality. Actually, this book cites and builds upon several of these resources. They complement, extend, and expand upon the topics that we discuss here. For every fact that we explain in this book, we try to cite proper sources at the end the book. Please always feel free to supplement the reading of this book with the following resources.

1.4.1 Lectures at Universities

Here we provide a list of lectures at different universities for which the slides are online available. When we discuss some topics, where appropriate, we will also reference the corresponding slides of these classes as easily-accessible and concise further reading.

- Yuriy Shamshin. *Databases*. ISMA University of Applied Sciences, May 2024. URL: <https://dbs.academy.lv>,
- Tim Kraska and Michael Cafarella. *6.5830/6.5831: Database Systems*. Massachusetts Institute of Technology (MIT), Aut. 2024. URL: <https://dsg.csail.mit.edu/6.5830>,
- Charles C. Palmer. *COSC 61 Winter 2025: Database Systems*. Dartmouth College, Jan.–Mar. 2025. URL: <https://www.cs.dartmouth.edu/~cs61>,
- Kevin Treu. *CSC-341: Database Management Systems*. Furman University, Spr. 2025. URL: <https://cs.furman.edu/~ktreu/csc341>,
- Junghoo “John” Cho. *CS143: Data Management Systems*. University of California – Los Angeles (UCLA), Sept. 2016–Aut. 2021. URL: <http://oak.cs.ucla.edu/classes/cs143>,
- Saty Raghavachary. *CSCI 585: Database Systems*. University of Southern California (UCS), Spr. 2024. URL: <https://bytes.usc.edu/cs585/s24-d-a-t-aaa/lectures>,
- Todd J. Green, ed. *ECS 165A Winter 2011 – Introduction to Database Systems*. University of California, Davis, Win. 2011. URL: <https://web.cs.ucdavis.edu/~green/courses/ecs165a-w11> as well as Todd J. Green, ed. *ECS 165B Spring 2011 – Database System Implementation*. University of California, Davis, Spr. 2011. URL: <https://web.cs.ucdavis.edu/~green/courses/ecs165b-s11>,
- Donnie Pinkston. *CS101b – Introduction to Relational Databases*. California Institute of Technology (Caltech), Win. 2006–Spr. 2007. URL: <http://users.cms.caltech.edu/~donnie/dbcourse/intro0607>,
- Heinz Schwerpe and Manuel Scholz. *Einführung in die Datenbanksysteme. Datenbanken für die Bioinformatik*. Freie Universität Berlin, Apr.–Oct. 2005. URL: <https://www.inf.fu-berlin.de/lehre/SS05/19517-V>, and
- Scott L. Vandenberg. *CSE 594: Database Management Systems*. University of Washington, Aut. 1999. URL: <https://courses.cs.washington.edu/courses/csep544/99au>.

1.4.2 Books on Databases in General

The following books may be useful for further reading:

- Christopher Painter-Wakefield. *A Practical Introduction to Databases*. Runestone Academy, 2022. URL: https://runestone.academy/ns/books/published/practical_db/index.html,
- Abraham “Avi” Silberschatz, Henry F. “Hank” Korth, and S. Sudarshan. *Database System Concepts*. 7th ed. McGraw-Hill, Mar. 2019. ISBN: 978-0-07-802215-9,
- Jeffrey A. Hoffer, Venkataraman Ramesh, and Heikki Topi. *Modern Database Management*. 13th ed. Pearson Education, Inc., Mar. 2021. ISBN: 978-0-13-477365-0,
- Ramez Elmasri and Shamkant Navathe. *Fundamentals of Database Systems*. 7th ed. Pearson Education, Inc., June 2015. ISBN: 978-0-13-397077-7,

- Alfons Kemper and André Eickler. *Datenbanksysteme: Eine Einführung*. Walter de Gruyter GmbH, 2015. ISBN: 978-3-11-044375-2,
- Mana Takahashi, Shoko Azuma, and Tokyo, Japan: Trend-Pro Co, Ltd. *The Manga Guide to Databases*. No Starch Press, Jan. 2009. ISBN: 978-1-59327-190-9,
- Michael J. Hernandez. *Database Design for Mere Mortals: 25th Anniversary Edition*. 4th ed. Addison-Wesley Professional, Dec. 2020. ISBN: 978-0-13-678813-3,
- Bill Karwin. *SQL Antipatterns: Avoiding the Pitfalls of Database Programming*. Pragmatic Bookshelf by The Pragmatic Programmers, L.L.C., June 2017. ISBN: 978-1-934356-55-5,
- Carlos Coronel and Steven Morris. *Database Systems: Design, Implementation, & Management*. 13th ed. Cengage Learning, Jan. 2018. ISBN: 978-1-337-62790-0,
- Christopher J. Date. *An Introduction to Database Systems*. 8th ed. Pearson Education, Inc., July 2003. ISBN: 978-0-321-19784-9,
- Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. 3rd ed. McGraw-Hill, Aug. 2002. ISBN: 978-0-07-246563-1,
- Jan L. Harrington. *Relational Database Design and Implementation*. 4th ed. Morgan Kaufmann Publishers, Apr. 2016. ISBN: 978-0-12-849902-3,
- Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database Systems: The Complete Book*. 2nd ed. Pearson Education, Inc., May 2008. ISBN: 978-0-13-187325-4,
- Terry Halpin and Tony Morgan. *Information Modeling and Relational Databases*. 3rd ed. Morgan Kaufmann Publishers, July 2024. ISBN: 978-0-443-23791-1,
- Peter Bailis, Joseph M. Hellerstein, and Michael Stonebraker, eds. *Readings in Database Systems*. 5th ed. 2015. URL: <http://www.redbook.io>,
- Jim Melton and Alan R. Simon. *SQL: 1999 – Understanding Relational Language Components*. Morgan Kaufmann Publishers, June 2001. ISBN: 978-1-55860-456-8,
- John Vincent Carlis and Joseph D. Maguire. *Mastering Data Modeling: A User Driven Approach*. Addison-Wesley Professional, Nov. 2000. ISBN: 978-0-201-70045-9,
- Allen Taylor. *Introducing SQL and Relational Databases*. Apress Media, LLC, Sept. 2018. ISBN: 978-1-4842-3841-7, and
- Ryan K. Stephens and Ronald R. Plew. *Sams Teach Yourself SQL in 21 Days*. 4th ed. Indianapolis, IN, USA: SAMS Technical Publishing and Hoboken, NJ, USA: Pearson Education, Inc., Oct. 2002. ISBN: 978-0-672-32451-2 and its German translation Ryan K. Stephens, Ronald R. Plew, Bryan Morgan, and Jeff Perkins. *SQL in 21 Tagen. Die Datenbank-Abfragesprache SQL vollständig erklärt (in 14/21 Tagen)*. 6th ed. Markt+Technik Verlag GmbH, Feb. 1998. ISBN: 978-3-8272-2020-2..

1.4.3 Books on Specific Database Technologies

The following books may be useful for further reading:

- Luca Ferrari and Enrico Pirozzi. *Learn PostgreSQL*. 2nd ed. Packt Publishing Ltd, Oct. 2023. ISBN: 978-1-83763-564-1,
- Alkin Tezuysal and Ibrar Ahmed. *Database Design and Modeling with PostgreSQL and MySQL*. Packt Publishing Ltd, July 2024. ISBN: 978-1-80323-347-5,
- Dušan Petković. *Microsoft SQL Server 2019: A Beginner's Guide*. 7th ed. McGraw-Hill, Jan. 2020. ISBN: 978-1-260-45888-6,
- Dirk Angermann. *T-SQL-Abfragen für Microsoft SQL-Server 2022*. mitp Verlags GmbH & Co. KG, June 2024. ISBN: 978-3-7475-0633-2,

- Darl Kuhn and Thomas Kyte. *Expert Oracle Database Architecture: Techniques and Solutions for High Performance and Productivity*. 4th ed. Apress Media, LLC, Nov. 2021. ISBN: 978-1-4842-7499-6,
- Ron McFadyen and Cindy Miller. *Relational Databases and Microsoft Access*. 3rd ed. Harper College, 2014–2019. URL: <https://harpercollege.pressbooks.pub/relationaldatabases>,
- Laurie A. Ulrich and Ken Cook. *Access For Dummies*. For Dummies (Wiley), Dec. 2021. ISBN: 978-1-119-82908-9,
- Christmas, FL, USA: Simon Sez IT. *Microsoft Access 2021 – Beginner to Advanced*. Packt Publishing Ltd, Aug. 2023. ISBN: 978-1-83546-911-8,
- Ben Beitler. *Hands-On Microsoft Access 2019*. Packt Publishing Ltd, Mar. 2020. ISBN: 978-1-83898-747-3,
- Raul F. Chong, Xiaomei Wang, Michael Dang, and Dwaine R. Snow. *Understanding DB2®: Learning Visually with Examples*. 2nd ed. IBM Press, Dec. 2007. ISBN: 978-0-7686-8177-2, and
- Jim Bainbridge, Hernando Bedoya, Rob Bestgen, Mike Cain, Dan Cruikshank, Jim Denton, Doug Mack, Tom Mckinley, and Simona Pacchiarini. *SQL Procedures, Triggers, and Functions on IBM DB2 for i*. IBM Redbooks, Apr. 2016. ISBN: 978-0-7384-4164-1.

1.4.4 Websites

Some websites that you might find useful are:

- *PostgreSQL Documentation*. 17.4. The PostgreSQL Global Development Group (PGDG), Feb. 2025. URL: <https://www.postgresql.org/docs/17/index.html>,
- *Database Administrators*. Stack Exchange Inc. URL: <https://dba.stackexchange.com>,
- Ben Brumm. *Database Star*. Elevated Online Services PTY Ltd., Dec. 2024. URL: <https://www.databasestar.com>,
- Adam “*djeada*” Djellouli. *Database Notes*. Feb. 2022–Mar. 2025. URL: https://adamdjellouli.com/articles/databases_notes, and
- Peter Whyte. *Microsoft SQL Server DBA Blog*. 2018–2025. URL: <https://peter-whyte.com/sql-dba-blog>.

Part I

Getting Started

The goal of this course is to introduce DBs. However, this is a practical course. So we will work on practical examples and create (more or less) “realistic” DBs on real DBMSes. This means that we will look into several interesting topics. And for most topics, we will try to get practical hands-on experience using a suitable software tool.

- We will work with an actual DBMS. So you need to install an actual DBMS. We choose PostgreSQL [161, 306, 339, 433].
- DBMSes are usually just *servers* to which you can send SQL commands via a *client terminal*. However, there exist also nicer tools offering rich *graphical user interfaces (GUIs)* that allow you to design forms and reports. Forms are structured graphical masks for data entry. Reports are documents that are automatically filled with data from a DB. We will try using such a tool as well. We choose LibreOffice Base [160, 385].
- The DBMS is usually just the backend of a landscape of tools in an enterprise or organization. Rarely will users work only with or directly on a DBMS. Instead, there may be several applications that connect to DBs through unified Application Programming Interfaces (APIs). Since we will also take a look at how that works, we need to install a programming language and corresponding “DBMS-access library.” We chose the Python programming language [10, 215, 247, 265], because we also provide a free book on it in [482]. As library to connect to the PostgreSQL DBMS, we pick psycopg [473].
- An important step of the DBs design process is to create an abstract conceptual model of the problem domain. This involves drawing graphical diagrams, so-called entity relationship diagrams (ERDs), describing the real-world entities and their relationships that should be modeled in the DB. The conceptual schema should be independent from any specific DBMS technology. As tool for drawing such diagrams, we will use yEd [384, 497].
- Conceptual models need to eventually be mapped to logical schemas that are based on certain data models and DB technologies. The logical model is how users and applications see the data. They may be implemented directly using a language like SQL. Interestingly, technology-specific conceptual models can also be designed using visual tools very similar to ERDs, which, this time, are bound to specific technologies and can be translated to commands for a DBMS. In my opinion, the best open source and free tool for drawing logical models for the PostgreSQL DBMS is PgModeler [8]. So we will use it here as well.

Best Practice 1

A computer science professional is able and always keen to learn new tools. A computer science professional should know dozens if not hundreds of different software tools for different tasks. A software engineer is a craftsperson and their knowledge of software is their tool belt.

We will use some specific DBMS, some specific visualization tools, some specific programming library to access the specific DBMS, and so on. One may wonder whether this is a good idea. If you would ever work professionally with DBs, most likely you will use entirely different software. Will the knowledge we learn be useless then?

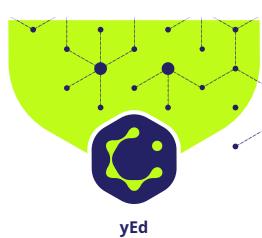
No. First of all, we will discuss the fundamentals of DBs and just use the concrete technologies as examples of how they play out in practice. So the knowledge about the fundamental concepts stays valuable, regardless of which tool you use.

Second, theoretical knowledge is not very useful in a professional setting if you cannot actually apply it. It does not help you if you have tried writing SQL code on paper but never actually executed it on a real DBMS. You do not know how to connect to a DBMS, how to input commands, how to understand error message. You would also never have searched for specific commands and documentation in the internet. You maybe would never have dealt with installing software and get it to run properly. This means that any practical application would still require you to learn lots of things and, most likely, under time pressure. However, if you *know* how to install PostgreSQL, how to connect to it using the psql client, if you actually have executed SQL commands, if you made mistakes and learned how to figure out how to fix them . . . then most likely you can relatively quickly learn how to do that for MySQL, MariaDB, or any other DBMS.



(1.17.1) The [Python](#) programming language logo is under the copyright of its owners.

(1.17.2) The logo of the [psycopg](#) module, the Python library for accessing [PostgreSQL](#). Copyright © Gabriella Albano and the Psycopg team.



(1.17.3) The logo of the [yEd graph editor](#). The yEd logo is protected by copyright. yEd is a registered trademark of [yWorks GmbH](#). Unauthorized use, reproduction, or distribution is strictly prohibited.



(1.17.4) The [PgModeler](#) logo is under the copyright of Raphael Araújo e Silva.

Figure 1.17: The logos of several of the very nice and free tools that we are using.

If you have *seen* ERDs and can read them, this does not mean that you can *efficiently* draw some if you are asked to do so. However, if you have drawn some using yEd or PgModeler, then at least you know that such tools exist and how they are used. You can probably either adapt to a new tool or find and install them if need be. But you would probably not start drawing such diagrams using a tool not suitable for that, like, e.g., a general graphics program such as Inkscape.

All the struggle of installing software, using command line arguments, connecting clients to servers, failing, and finding solutions will help you when you need to do similar things for entirely other software. Usage paradigms and even fixes for errors are often similar over different tools, so the more tools you use, the faster you become learning how to use other tools. Therefore, I strongly advocate not just learning the fundamentals of DBs, but to actually try them out, exercise them, use tools.

Finally, all the tools that we consider here are free and, ideally, open source, software. We do not use tools that cost money. Learning about DBs with this book should be free.

First Time Readers and Novices: In this part of the book, we have collected installation instructions for all the tools that we use in those books. **You do not need to install all of them right away.** Just be aware that we provide installation instructions for the tools that we need here. When we eventually need the specific software tools at some time later on, we will refer back to this part.

Either way, before we get into the necessary installation and setup steps for the software that we eventually need to really learn about DBs, we face a small problem: Today, devices with many different OS are available. For each OS, the installation steps and software availability may be different, so I cannot possibly cover them all. Personally, I strongly recommend using Linux [25, 198, 448] for programming, work, and research. If you are a student of computer science or any related field, then it is my personal opinion that you should get familiar with this operating system.

Best Practice 2

Any professional computer scientist, software developer, software architect, DBA, or system administrator should be familiar with the Linux OS.

Maybe you could start with the very easy-to-use Ubuntu Linux [87, 201]. If you are a Microsoft Windows[50] user, maybe you could install Ubuntu in a virtual machine. Either way, I strongly recommend learning and using Linux.

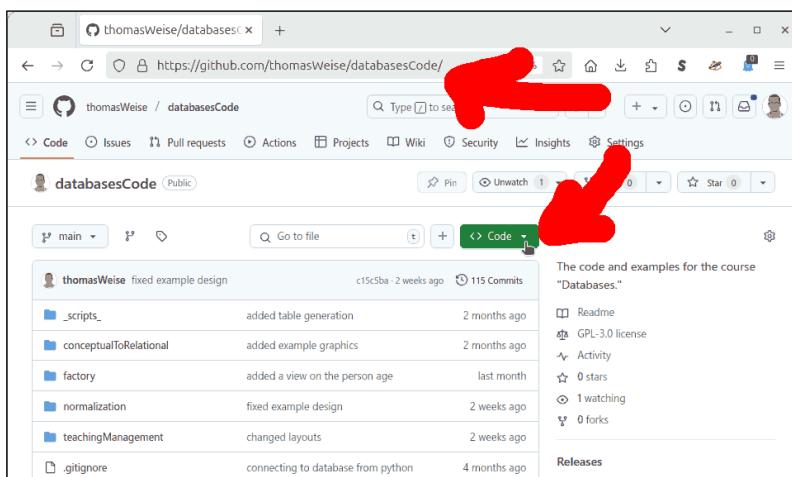
It should also be clear that the instructions provided here will eventually be outdated. I am not sure whether I will be able to keep updating them in the future. However, even a few years down the road, they should still provide some basic guidance. In the following, I will try to provide examples and instructions for both Ubuntu Linux [87, 201] and the commercial Microsoft Windows [50] OS.

Chapter 2

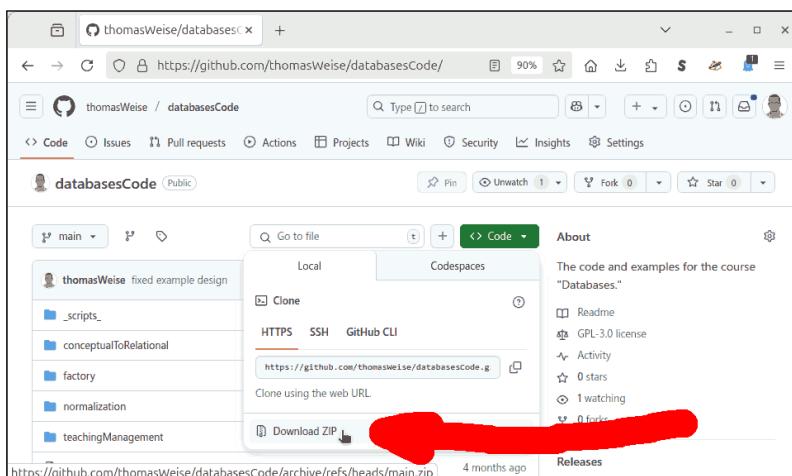
Downloading the Example Codes

In this book, we provide a lot of examples. Some of them are **SQL** scripts that can be sent to the **PostgreSQL DBMS server** via the **psql** client. Others are **ERDs** drawn with **yEd**. Yet others are domain-specific **DB** models developed with **PgModeler**. For all examples, we always provide the source files. So you do not need to type in the examples from the book. You can download them!

All these examples are provided in the repository **databasesCode** on **GitHub**. There are two basic ways to obtain all of these examples: First, you can download them using your web browser. This basically requires no prerequisites except a web browser, which is usually part of any **OS**. Second, you

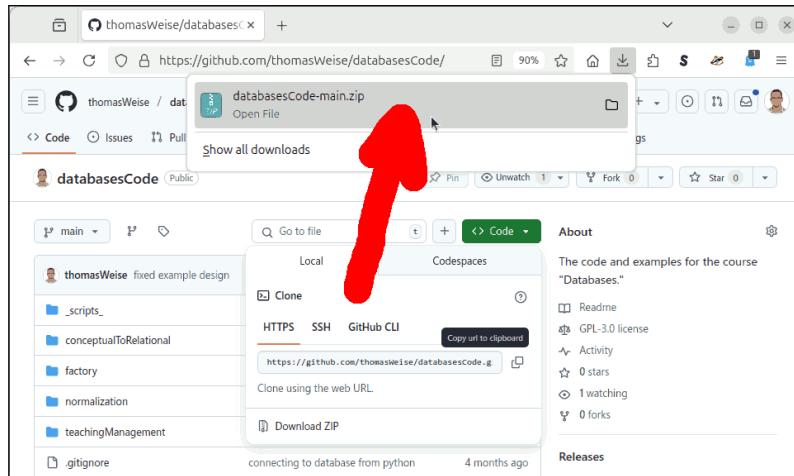


(2.1.1) We use a web browser to visit the website <https://github.com/thomasWeise/databasesCode>. On the website, we click on the green **Code** menu.



(2.1.2) A small popup-menu appears, where we click on **Download ZIP**.

Figure 2.1: Downloading the examples from this book.



(2.2.1) Once the file is downloaded, we open it.

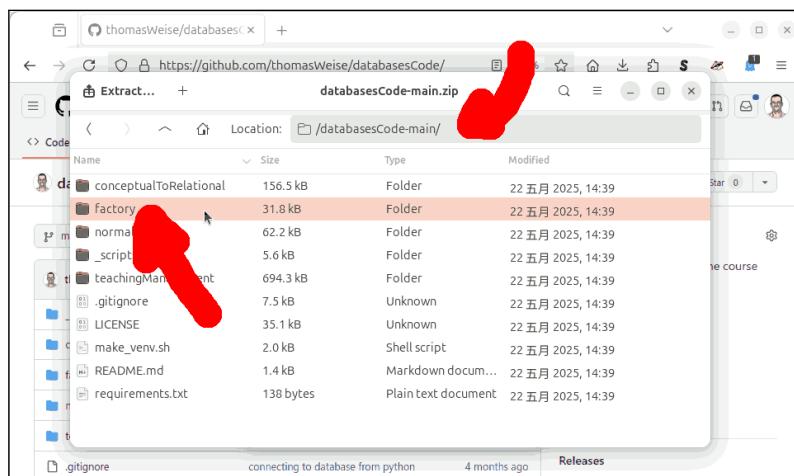
(2.2.2) In the opened archive, we can find *all* the examples of this book. In folder `factory`, we find the simple factory example.

Figure 2.2: Downloading the examples from this book. (Continued)

can use **PyCharm Integrated Development Environment (IDE)** to clone the **Git** repository and setting up **Python** dependencies in one swoop. We discuss this option in [Section 5.1.2](#). This, however, requires you to also first install Python and PyCharm, which is discussed in our freely available sister course [Programming with Python](#) [482]. If you have these tools available, then that might be the even more convenient choice.

For downloading the examples directly, you would visit this repository at <https://github.com/thomasWeise/databasesCode>. This is illustrated in [Chapter 2](#). As shown in [Figure 2.1.1](#), you would use your web browser to visit the website <https://github.com/thomasWeise/databasesCode>. Once the website has loaded, you click on the green `Code` menu. Then, in [Figure 2.1.2](#), a small popup-menu appears, where you click on `Download ZIP`. This will download a so-called ZIP-archive, i.e., a file that contains a compressed folder structure with all files in our examples repository. After the download completes, as illustrated in [Figure 2.2.1](#), you then can open the archive. In the opened archive, you can find *all* the examples of this book in a folder called `databasesCode-main`. This folder contains another folder called `factory`, where you can find all the files belonging to our present example, as shown in [Figure 2.2.2](#).

Additionally, each source code listing has a headline containing some text like “(stored in file `myfile`;” `myfile` then is a clickable link that would take you directly to the file on **GitHub**.

Chapter 3

Installing PostgreSQL

For our course, we will use the open source DBMS PostgreSQL [161, 306, 339, 433] as the actual system to experiment with DBs. Here, we briefly discuss how to install it on your machine. We will then use PostgreSQL in Part II.

PostgreSQL follows the client-server architecture. It provides the DBMS implemented as server program. It manages the DBs and stores and provides the data. Other programs and the users can connect to it to access the DBs. PostgreSQL also offers a client program, `psql`, with which human users can communicate with the DBMS server in a normal terminal. We want to install both pieces of software. You can find more information about this process at the PostgreSQL download page <https://www.postgresql.org/download>. Here, we provide step-by-step guides for Ubuntu Linux and Microsoft Windows.

3.1 Installing PostgreSQL under Ubuntu Linux

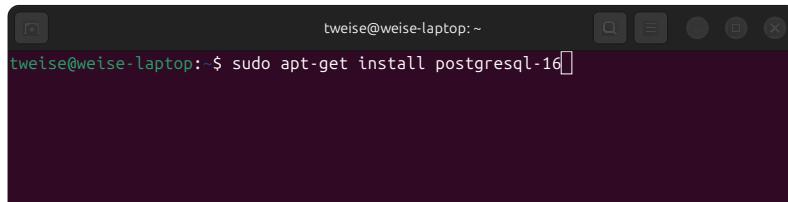
The installation PostgreSQL on Ubuntu Linux is rather simple. First, we open a Bash terminal, i.e., a window into which we can directly type commands, with `Ctrl+Alt+T`. Both the client and server program are installed via `sudo apt-get install postgresql-16`, as shown in Figure 3.1.1. Notice that we chose to install PostgreSQL version 16, denoted by the `-16` suffix, but you could probably also pick other versions or just install `postgresql` without version specification. Either way, this installation method requires super user privileges, which is why we have to start the command with `sudo`. In Figure 3.1.2, we therefore need to provide our superuser password.

When this is entered and confirmed with `↓`, the package manager checks what needs to be installed. In Figure 3.1.3 it informs us about the package itself and the dependencies that are needed, as well as how much download and space that requires. It asks us whether we are OK with that, which we confirm by typing `y + ↓` in Figure 3.1.4. Then, the download of the required packages starts. Once the download completes, the packages are installed. After this completes in Figure 3.1.5, PostgreSQL is installed and running.

We now want to double-check whether everything went well. First, we need to investigate whether the PostgreSQL DBMS server is indeed installed and running properly. We can do this by invoking `systemctl status postgresql` in Figure 3.1.6. The output on my laptop computer, given in Figure 3.1.7, indicates that the service is up and running. Notice `active` means that the DBMS is started everytime your computer boots and runs all the time (until you shutdown your computer, that is). Running this program opens some sort of paginated mode, which we can leave by hitting `q + ↓` in Figure 3.1.8.

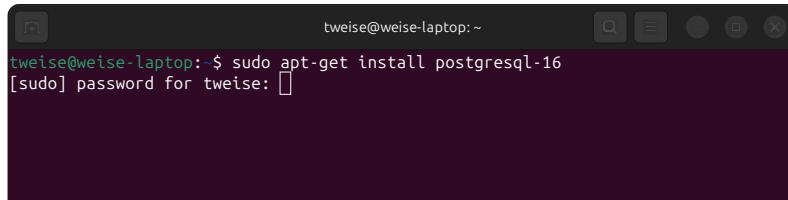
If you do not want that, then you disable the service by the command `sudo systemctl disable postgresql` in the terminal, which again requires sudo privileges. The PostgreSQL server will then no longer start automatically. If you want to access it, you then would start it manually via `sudo systemctl start postgresql`. After you are done with it, you can stop it by typing `sudo systemctl stop postgresql`. If you ever want it to start automatically again, this can be achieved by `sudo systemctl enable postgresql`.

Either way, after a successful installation, the PostgreSQL server service is activated and running. We now also want to check whether the client program `psql` was installed correctly. The client can be run in the terminal and allows us to communicate with the DBMS server. We therefore type `psql --version` in Figure 3.1.9, which will print the version of this program after we hit `↓`. The



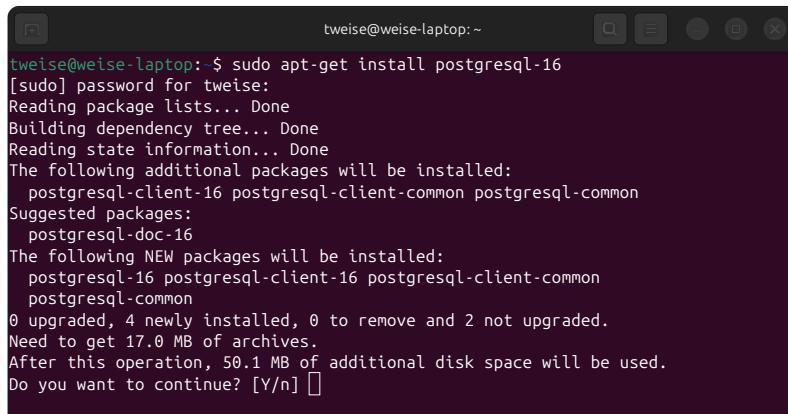
tweise@weise-laptop: \$ sudo apt-get install postgresql-16

(3.1.1) Installing PostgreSQL using the `apt-get install` command in a terminal opened with $\text{[Ctrl} + \text{Alt} + \text{T}]$.



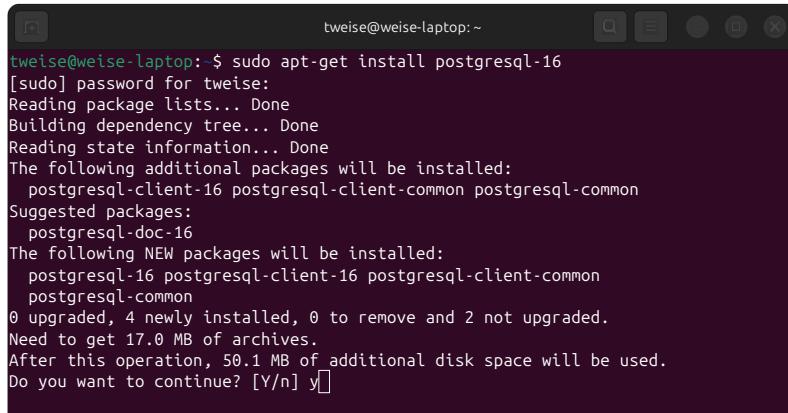
tweise@weise-laptop: \$ sudo apt-get install postgresql-16
[sudo] password for tweise:

(3.1.2) This command requires the super user password, which we type in and then press [Enter] .



```
tweise@weise-laptop: $ sudo apt-get install postgresql-16
[sudo] password for tweise:
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
The following additional packages will be installed:
  postgresql-client-16 postgresql-client-common postgresql-common
Suggested packages:
  postgresql-doc-16
The following NEW packages will be installed:
  postgresql-16 postgresql-client-16 postgresql-client-common
  postgresql-common
0 upgraded, 4 newly installed, 0 to remove and 2 not upgraded.
Need to get 17.0 MB of archives.
After this operation, 50.1 MB of additional disk space will be used.
Do you want to continue? [Y/n]
```

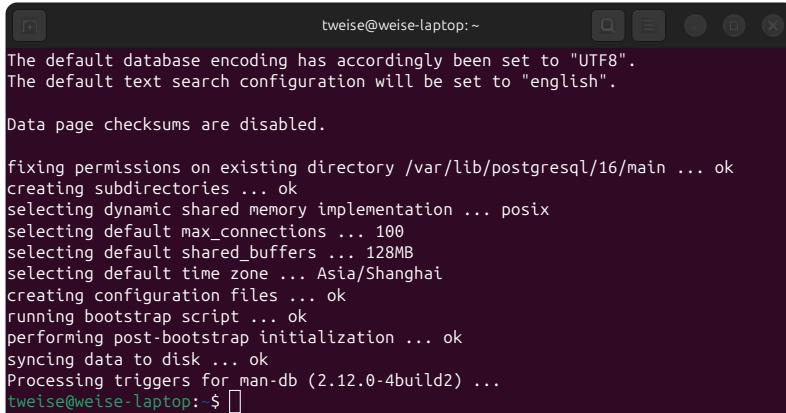
(3.1.3) We get asked whether we really want to install the required packages.



```
tweise@weise-laptop: $ sudo apt-get install postgresql-16
[sudo] password for tweise:
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
The following additional packages will be installed:
  postgresql-client-16 postgresql-client-common postgresql-common
Suggested packages:
  postgresql-doc-16
The following NEW packages will be installed:
  postgresql-16 postgresql-client-16 postgresql-client-common
  postgresql-common
0 upgraded, 4 newly installed, 0 to remove and 2 not upgraded.
Need to get 17.0 MB of archives.
After this operation, 50.1 MB of additional disk space will be used.
Do you want to continue? [Y/n] y
```

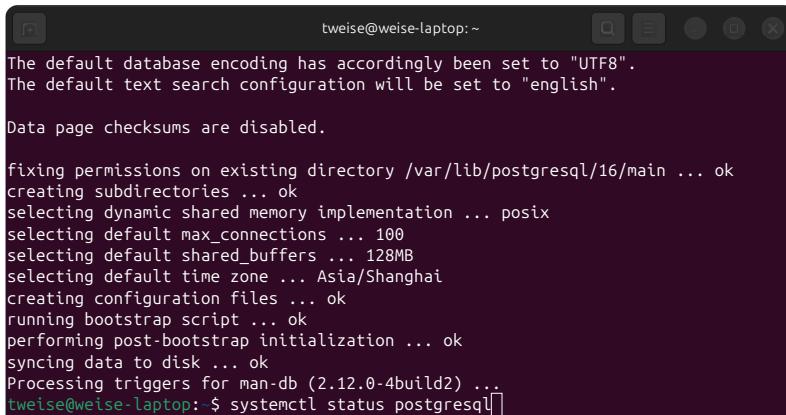
(3.1.4) We answer with $\text{[y]} + \text{[Enter]}$.

Figure 3.1: Installing PostgreSQL under Ubuntu Linux, checking its status, and setting a secure password.



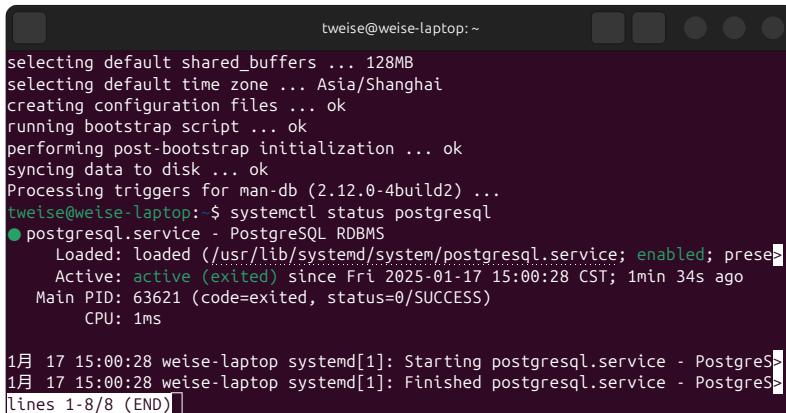
```
tweise@weise-laptop:~  
The default database encoding has accordingly been set to "UTF8".  
The default text search configuration will be set to "english".  
Data page checksums are disabled.  
fixing permissions on existing directory /var/lib/postgresql/16/main ... ok  
creating subdirectories ... ok  
selecting dynamic shared memory implementation ... posix  
selecting default max_connections ... 100  
selecting default shared_buffers ... 128MB  
selecting default time zone ... Asia/Shanghai  
creating configuration files ... ok  
running bootstrap script ... ok  
performing post-bootstrap initialization ... ok  
syncing data to disk ... ok  
Processing triggers for man-db (2.12.0-4build2) ...  
tweise@weise-laptop: $
```

(3.1.5) The installation proceeds and finishes.



```
tweise@weise-laptop:~  
The default database encoding has accordingly been set to "UTF8".  
The default text search configuration will be set to "english".  
Data page checksums are disabled.  
fixing permissions on existing directory /var/lib/postgresql/16/main ... ok  
creating subdirectories ... ok  
selecting dynamic shared memory implementation ... posix  
selecting default max_connections ... 100  
selecting default shared_buffers ... 128MB  
selecting default time zone ... Asia/Shanghai  
creating configuration files ... ok  
running bootstrap script ... ok  
performing post-bootstrap initialization ... ok  
syncing data to disk ... ok  
Processing triggers for man-db (2.12.0-4build2) ...  
tweise@weise-laptop: $ systemctl status postgresql
```

(3.1.6) We want to check the status of the fresh PostgreSQL installation. We can do this by typing in `systemctl status postgresql` and hit ↵.



```
tweise@weise-laptop:~  
selecting default shared_buffers ... 128MB  
selecting default time zone ... Asia/Shanghai  
creating configuration files ... ok  
running bootstrap script ... ok  
performing post-bootstrap initialization ... ok  
syncing data to disk ... ok  
Processing triggers for man-db (2.12.0-4build2) ...  
tweise@weise-laptop: $ systemctl status postgresql  
● postgresql.service - PostgreSQL RDBMS  
  Loaded: loaded (/usr/lib/systemd/system/postgresql.service; enabled; preser...  
  Active: active (exited) since Fri 2025-01-17 15:00:28 CST; 1min 34s ago  
    Main PID: 63621 (code=exited, status=0/SUCCESS)  
      CPU: 1ms  
  
Jan 17 15:00:28 weise-laptop systemd[1]: Starting postgresql.service - PostgreS...  
Jan 17 15:00:28 weise-laptop systemd[1]: Finished postgresql.service - Postgres...  
[lines 1-8/8 (END)]
```

(3.1.7) The output shows us that the PostgreSQL service is now running. It will always start when we boot our system.

Figure 3.1: Installing PostgreSQL under Ubuntu Linux, checking its status, and setting a secure password.

```
tweise@weise-laptop:~$ systemctl status postgresql
● postgresql.service - PostgreSQL RDBMS
    Loaded: loaded (/usr/lib/systemd/system/postgresql.service; enabled; presen...
    Active: active (exited) since Fri 2025-01-17 15:00:28 CST; 1min 34s ago
      Main PID: 63621 (code=exited, status=0/SUCCESS)
        CPU: 1ms

1月 17 15:00:28 weise-laptop systemd[1]: Starting postgresql.service - PostgreSQL RDBMS
1月 17 15:00:28 weise-laptop systemd[1]: Finished postgresql.service - PostgreSQL RDBMS
tweise@weise-laptop:~$
```

(3.1.8) We press `q` + `↵` to leave the status output page.

```
tweise@weise-laptop:~$ systemctl status postgresql
● postgresql.service - PostgreSQL RDBMS
    Loaded: loaded (/usr/lib/systemd/system/postgresql.service; enabled; presen...
    Active: active (exited) since Fri 2025-01-17 15:00:28 CST; 1min 34s ago
      Main PID: 63621 (code=exited, status=0/SUCCESS)
        CPU: 1ms

1月 17 15:00:28 weise-laptop systemd[1]: Starting postgresql.service - PostgreSQL RDBMS
1月 17 15:00:28 weise-laptop systemd[1]: Finished postgresql.service - PostgreSQL RDBMS
tweise@weise-laptop:~$
```

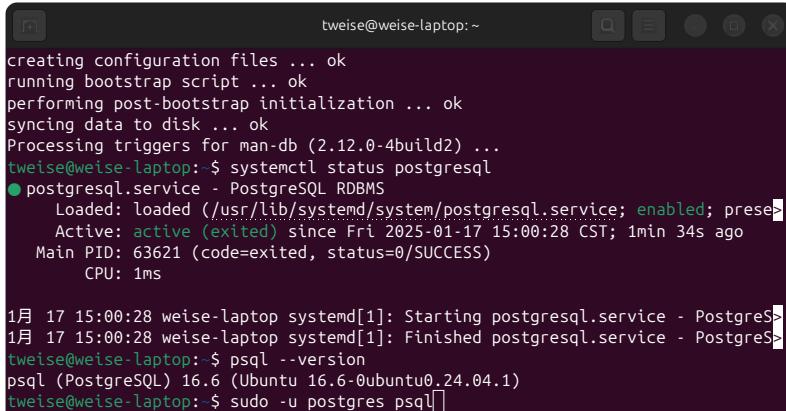
(3.1.9) `psql` is the client program that can connect to the PostgreSQL server and that gets installed as well. We can check its version by typing `psql --version` and hitting `↵`.

```
tweise@weise-laptop:~$ systemctl status postgresql
● postgresql.service - PostgreSQL RDBMS
    Loaded: loaded (/usr/lib/systemd/system/postgresql.service; enabled; presen...
    Active: active (exited) since Fri 2025-01-17 15:00:28 CST; 1min 34s ago
      Main PID: 63621 (code=exited, status=0/SUCCESS)
        CPU: 1ms

1月 17 15:00:28 weise-laptop systemd[1]: Starting postgresql.service - PostgreSQL RDBMS
1月 17 15:00:28 weise-laptop systemd[1]: Finished postgresql.service - PostgreSQL RDBMS
tweise@weise-laptop:~$ psql --version
psql (PostgreSQL) 16.6 (Ubuntu 16.6-0ubuntu0.24.04.1)
tweise@weise-laptop:~$
```

(3.1.10) At the time of this writing, I got version 16.6.

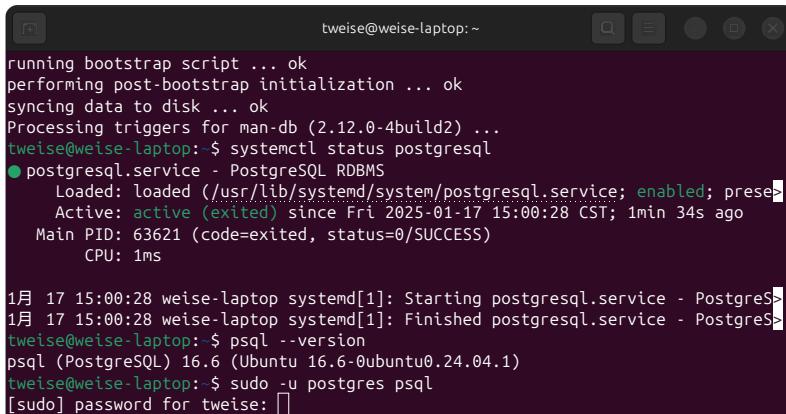
Figure 3.1: Installing PostgreSQL under Ubuntu Linux, checking its status, and setting a secure password.



```
tweise@weise-laptop:~ creating configuration files ... ok
running bootstrap script ... ok
performing post-bootstrap initialization ... ok
syncing data to disk ... ok
Processing triggers for man-db (2.12.0-4build2) ...
tweise@weise-laptop: $ systemctl status postgresql
● postgresql.service - PostgreSQL RDBMS
    Loaded: loaded (/usr/lib/systemd/system/postgresql.service; enabled; presen...
    Active: active (exited) since Fri 2025-01-17 15:00:28 CST; 1min 34s ago
      Main PID: 63621 (code=exited, status=0/SUCCESS)
        CPU: 1ms

1月 17 15:00:28 weise-laptop systemd[1]: Starting postgresql.service - Postgres...
1月 17 15:00:28 weise-laptop systemd[1]: Finished postgresql.service - Postgres...
tweise@weise-laptop: $ psql --version
psql (PostgreSQL) 16.6 (Ubuntu 16.6-0ubuntu0.24.04.1)
tweise@weise-laptop: $ sudo -u postgres psql
```

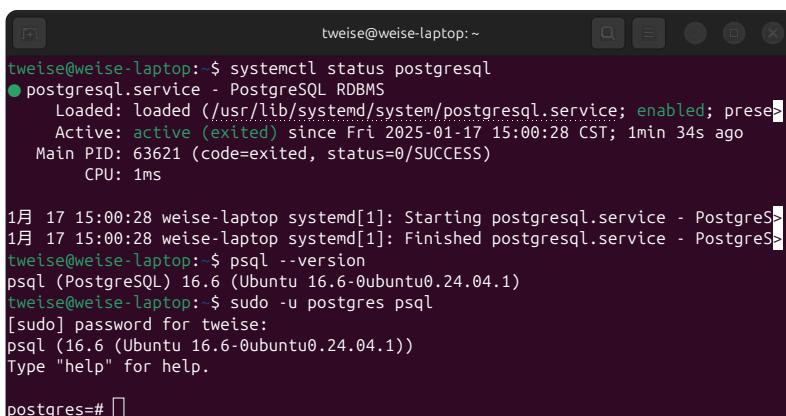
(3.1.11) In order to set a proper password for the PostgreSQL main user `postgres`, we need to log into `psql` using `sudo` privileges, but under the newly created system user `postgres`. We do this by writing `sudo -u postgres psql` and hit ↵.



```
tweise@weise-laptop:~ running bootstrap script ... ok
performing post-bootstrap initialization ... ok
syncing data to disk ... ok
Processing triggers for man-db (2.12.0-4build2) ...
tweise@weise-laptop: $ systemctl status postgresql
● postgresql.service - PostgreSQL RDBMS
    Loaded: loaded (/usr/lib/systemd/system/postgresql.service; enabled; presen...
    Active: active (exited) since Fri 2025-01-17 15:00:28 CST; 1min 34s ago
      Main PID: 63621 (code=exited, status=0/SUCCESS)
        CPU: 1ms

1月 17 15:00:28 weise-laptop systemd[1]: Starting postgresql.service - Postgres...
1月 17 15:00:28 weise-laptop systemd[1]: Finished postgresql.service - Postgres...
tweise@weise-laptop: $ psql --version
psql (PostgreSQL) 16.6 (Ubuntu 16.6-0ubuntu0.24.04.1)
tweise@weise-laptop: $ sudo -u postgres psql
[sudo] password for tweise:
```

(3.1.12) We may or may not need to enter our super user password again...



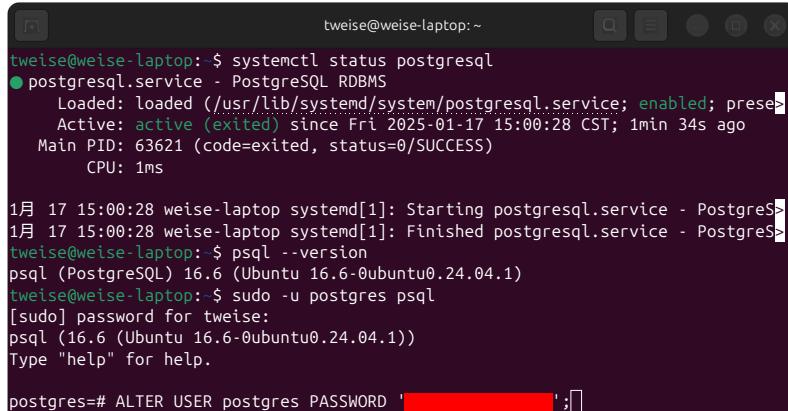
```
tweise@weise-laptop: $ systemctl status postgresql
● postgresql.service - PostgreSQL RDBMS
    Loaded: loaded (/usr/lib/systemd/system/postgresql.service; enabled; presen...
    Active: active (exited) since Fri 2025-01-17 15:00:28 CST; 1min 34s ago
      Main PID: 63621 (code=exited, status=0/SUCCESS)
        CPU: 1ms

1月 17 15:00:28 weise-laptop systemd[1]: Starting postgresql.service - Postgres...
1月 17 15:00:28 weise-laptop systemd[1]: Finished postgresql.service - Postgres...
tweise@weise-laptop: $ psql --version
psql (PostgreSQL) 16.6 (Ubuntu 16.6-0ubuntu0.24.04.1)
tweise@weise-laptop: $ sudo -u postgres psql
[sudo] password for tweise:
psql (16.6 (Ubuntu 16.6-0ubuntu0.24.04.1))
Type "help" for help.

postgres=#
```

(3.1.13) Starting `psql` like this connects us to the local PostgreSQL server.

Figure 3.1: Installing PostgreSQL under Ubuntu Linux, checking its status, and setting a secure password.

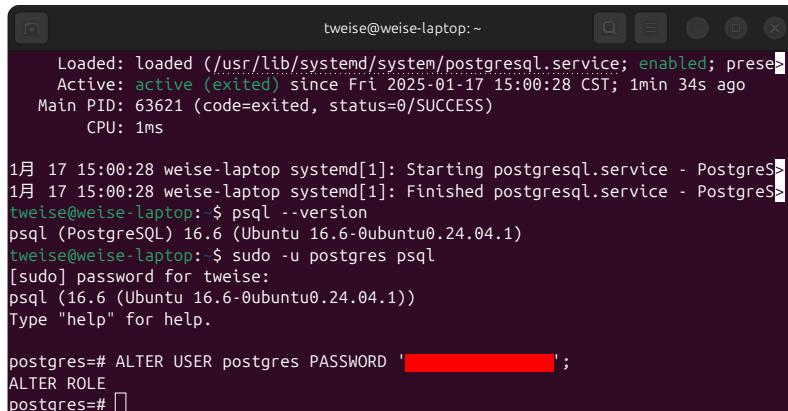


```
tweise@weise-laptop:~$ systemctl status postgresql
● postgresql.service - PostgreSQL RDBMS
  Loaded: loaded (/usr/lib/systemd/system/postgresql.service; enabled; presen...
  Active: active (exited) since Fri 2025-01-17 15:00:28 CST; 1min 34s ago
    Main PID: 63621 (code=exited, status=0/SUCCESS)
      CPU: 1ms

1月 17 15:00:28 weise-laptop systemd[1]: Starting postgresql.service - PostgreSQL...
1月 17 15:00:28 weise-laptop systemd[1]: Finished postgresql.service - PostgreSQL...
tweise@weise-laptop:~$ psql --version
psql (PostgreSQL) 16.6 (Ubuntu 16.6-0ubuntu0.24.04.0)
tweise@weise-laptop:~$ sudo -u postgres psql
[sudo] password for tweise:
psql (16.6 (Ubuntu 16.6-0ubuntu0.24.04.0))
Type "help" for help.

postgres=# ALTER USER postgres PASSWORD 'XXXXXXXXXX';
```

(3.1.14) We can now communicate with the DBMS using SQL. We use the `ALTER USER postgres PASSWORD 'XXX';` command, where `XXX` is replaced with a secure password. In the screenshot, the password is covered by a red rectangle after I replaced it with many Xs.

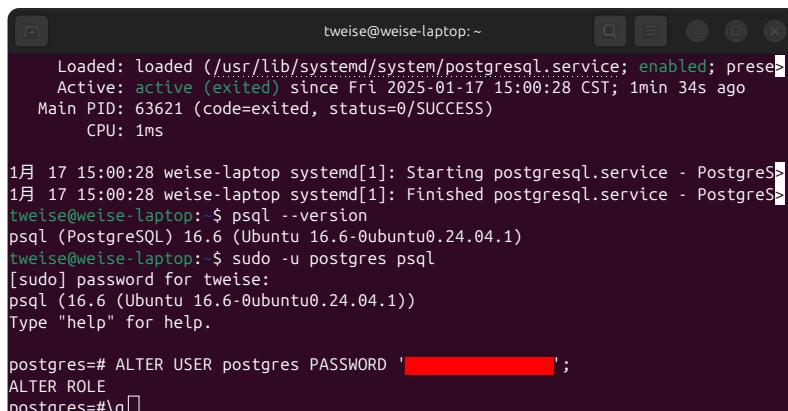


```
tweise@weise-laptop:~$ systemctl status postgresql
● postgresql.service - PostgreSQL RDBMS
  Loaded: loaded (/usr/lib/systemd/system/postgresql.service; enabled; presen...
  Active: active (exited) since Fri 2025-01-17 15:00:28 CST; 1min 34s ago
    Main PID: 63621 (code=exited, status=0/SUCCESS)
      CPU: 1ms

1月 17 15:00:28 weise-laptop systemd[1]: Starting postgresql.service - PostgreSQL...
1月 17 15:00:28 weise-laptop systemd[1]: Finished postgresql.service - PostgreSQL...
tweise@weise-laptop:~$ psql --version
psql (PostgreSQL) 16.6 (Ubuntu 16.6-0ubuntu0.24.04.0)
tweise@weise-laptop:~$ sudo -u postgres psql
[sudo] password for tweise:
psql (16.6 (Ubuntu 16.6-0ubuntu0.24.04.0))
Type "help" for help.

postgres=# ALTER USER postgres PASSWORD 'XXXXXXXXXX';
ALTER ROLE
postgres#
```

(3.1.15) After we hit `↵`, the system confirms the change by showing the command `ALTER ROLE` back to us. From now on, the server main user has a secure password.



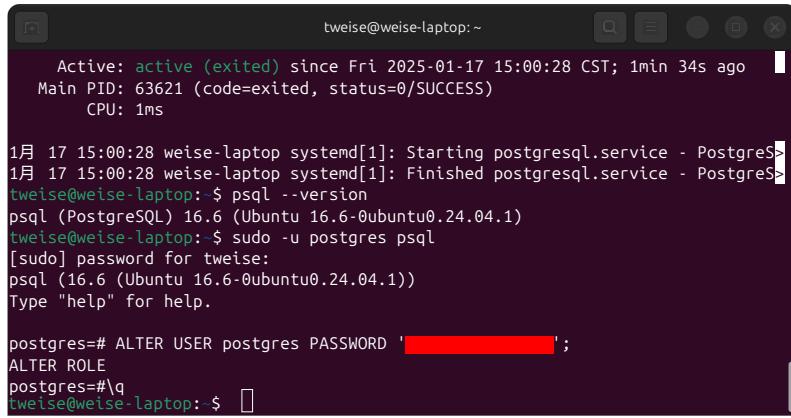
```
tweise@weise-laptop:~$ systemctl status postgresql
● postgresql.service - PostgreSQL RDBMS
  Loaded: loaded (/usr/lib/systemd/system/postgresql.service; enabled; presen...
  Active: active (exited) since Fri 2025-01-17 15:00:28 CST; 1min 34s ago
    Main PID: 63621 (code=exited, status=0/SUCCESS)
      CPU: 1ms

1月 17 15:00:28 weise-laptop systemd[1]: Starting postgresql.service - PostgreSQL...
1月 17 15:00:28 weise-laptop systemd[1]: Finished postgresql.service - PostgreSQL...
tweise@weise-laptop:~$ psql --version
psql (PostgreSQL) 16.6 (Ubuntu 16.6-0ubuntu0.24.04.0)
tweise@weise-laptop:~$ sudo -u postgres psql
[sudo] password for tweise:
psql (16.6 (Ubuntu 16.6-0ubuntu0.24.04.0))
Type "help" for help.

postgres=# ALTER USER postgres PASSWORD 'XXXXXXXXXX';
ALTER ROLE
postgres#\q
```

(3.1.16) We quit psql by typing `\q`.

Figure 3.1: Installing PostgreSQL under Ubuntu Linux, checking its status, and setting a secure password.



```
tweise@weise-laptop:~
```

```
Active: active (exited) since Fri 2025-01-17 15:00:28 CST; 1min 34s ago
Main PID: 63621 (code=exited, status=0/SUCCESS)
CPU: 1ms

1月 17 15:00:28 weise-laptop systemd[1]: Starting postgresql.service - PostgreSQL
1月 17 15:00:28 weise-laptop systemd[1]: Finished postgresql.service - PostgreSQL
tweise@weise-laptop: $ psql --version
psql (PostgreSQL) 16.6 (Ubuntu 16.6-0ubuntu0.24.04.1)
tweise@weise-laptop: $ sudo -u postgres psql
[sudo] password for tweise:
psql (16.6 (Ubuntu 16.6-0ubuntu0.24.04.1))
Type "help" for help.

postgres=# ALTER USER postgres PASSWORD '████████';
ALTER ROLE
postgres=# \q
tweise@weise-laptop: $
```

(3.1.17) We are finished.

Figure 3.1: Installing PostgreSQL under Ubuntu Linux, checking its status, and setting a secure password.

result in Figure 3.1.10 shows that the `psql` client version on my system: 16.6.

Let us now connect to the PostgreSQL server via the `client` for the first time. The goal of this exercise will be to set a proper password for the administrative user of our instance the PostgreSQL DBMS. The installation process created both a Linux system user named `postgres` as well as a user of the same name in the DBMS. The DBMS needs to have its own user and rights management implemented, because we need to be able to grant different programs and users different read and write privileges for a DB (remember Section 1.1.7). Thus, we would like the administrative account `postgres` of the DBMS to have some secure password.

We therefore connect the `psql` client to the PostgreSQL server as superuser impersonating the `postgres` system user. This sounds very strange. I am not even sure whether I explained it correctly. Either way, we type `sudo -u postgres psql` into the terminal and hit ↵ in Figure 3.1.11. We may get asked to provide the superuser password for our computer again in Figure 3.1.12, in which case we simply provide it and hit ↵.

For the first time, we are now in the `psql` client and are connected to the PostgreSQL server. We see the prompt `postgres=#` in Figure 3.1.13.

To set the password for the user (or role) `postgres`, we type in `ALTER USER postgres PASSWORD 'XXX';` in Figure 3.1.14. (This is equivalent to `ALTER ROLE postgres PASSWORD 'XXX';` in PostgreSQL.) This is actually an SQL command and we may (or may not) learn later what this exactly means (once and if I get to write such a chapter). This would set the password to `XXX`. Obviously, this is not the secure password that we are going to use. Instead, you will not type `XXX` but a secure password of your choosing. A password that you shall remember well. We hit ↵.

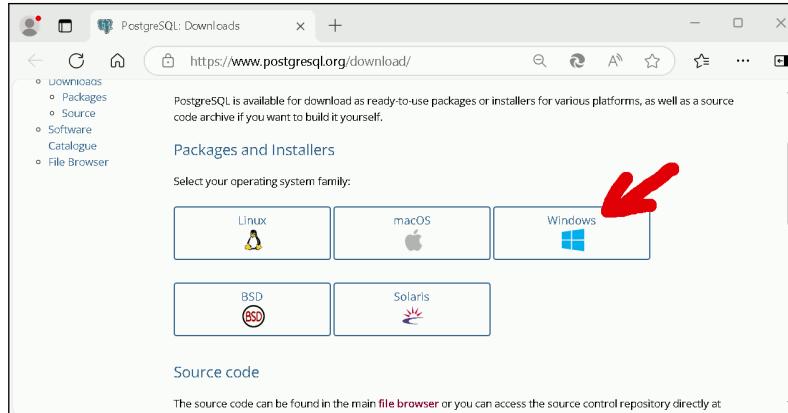
In Figure 3.1.15, we see that this prints `ALTER ROLE` back to us. It shows us the command that was performed. We did alter the role (or user) `postgres`. The new password is set. You will learn how to use that another time.

For now, we will happily exit the `psql` client by typing ↵ + q + ↵. In Figure 3.1.16, we quit the client this way. In Figure 3.1.17, we then are back in our normal Bash terminal. We have finished and validated the PostgreSQL installation. And we have set a proper password for the administrator account `postgres`.

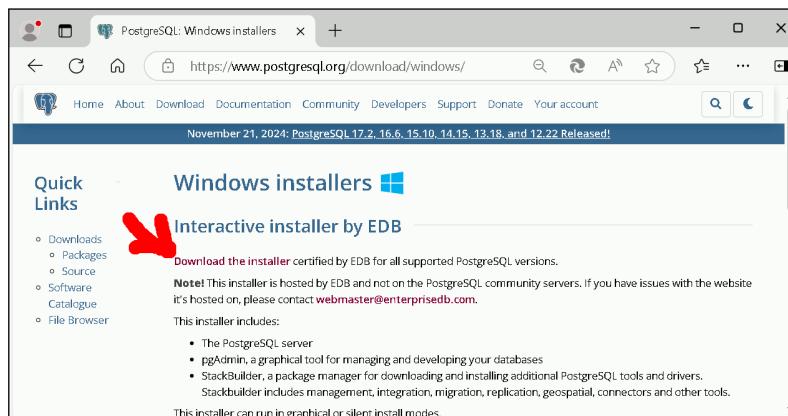
3.2 Installing PostgreSQL under Microsoft Windows

We want to install and properly configure PostgreSQL on a Microsoft Windows system. To do this, we first visit the official download page at <https://www.postgresql.org/download>, as shown in Figure 3.2.1. There, we click on the large Windows  menu. This leads to a page with more text and the link *download the installer* (Figure 3.2.2). This, in turn, takes us to <https://www.enterprisedb.com/downloads/postgres-postgresql-downloads>, as shown in Figure 3.2.3.

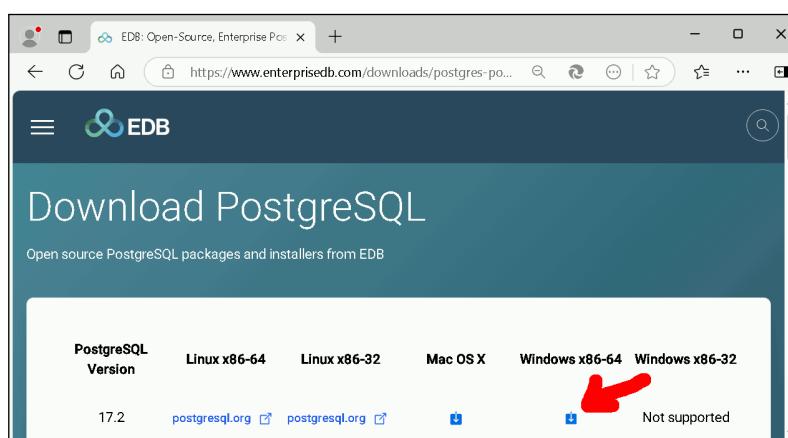
On this website, a wide variety of different versions of PostgreSQL for different OSes can be



(3.2.1) We visit the website <https://www.postgresql.org/download> and click on Windows.

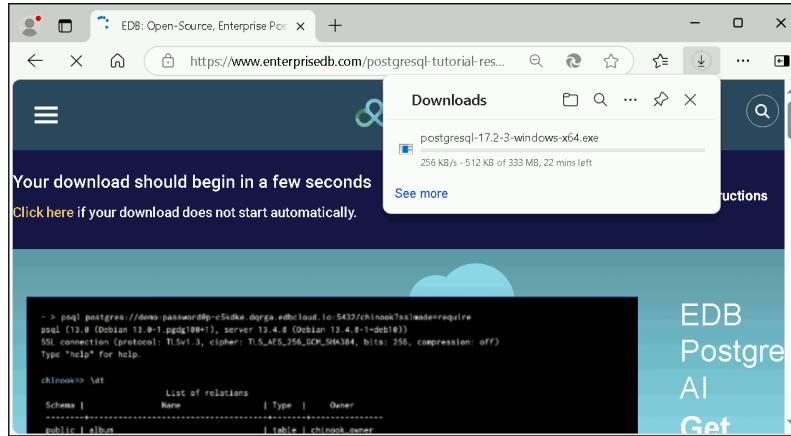


(3.2.2) We click on the link *download the installer*, which takes us to <https://www.enterprisedb.com/downloads/postgres-postgresql-downloads>.

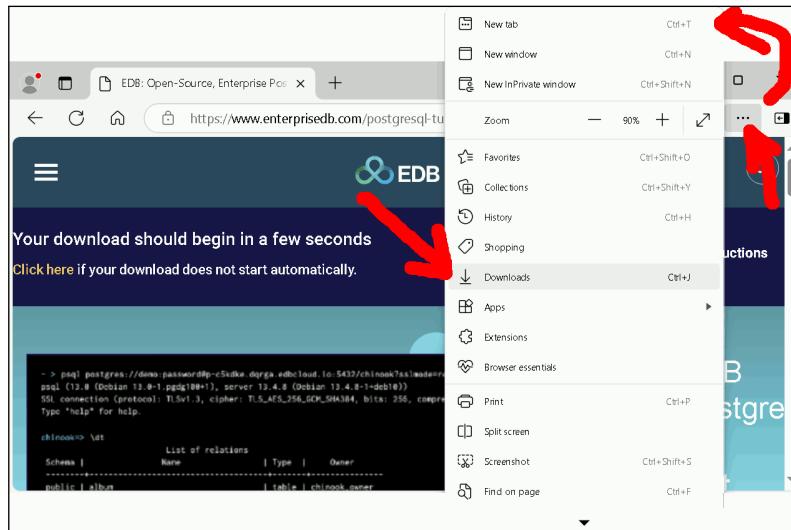


(3.2.3) A wide variety of versions for different OSes are available. We choose the newest (top-most) one for Microsoft Windows, at the time of this writing, this is version 17.2. We click the download icon.

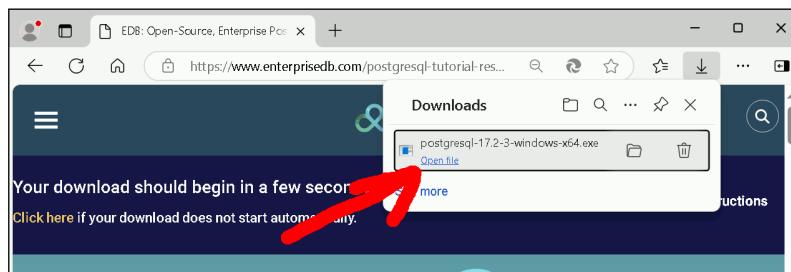
Figure 3.2: Installing and configuring PostgreSQL under Microsoft Windows.



(3.2.4) The download begins.



(3.2.5) After the download completes, we need to find it...



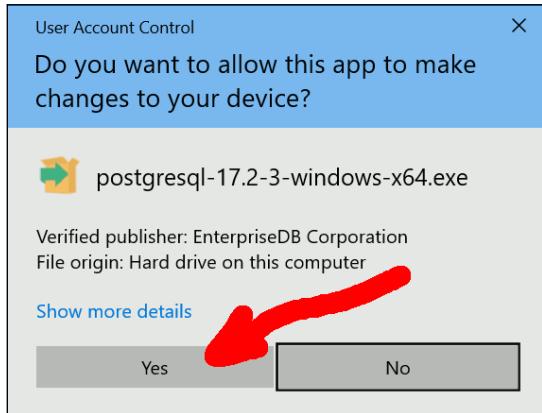
(3.2.6) ...and execute it by clicking on [Open file].

Figure 3.2: Installing and configuring PostgreSQL under Microsoft Windows (Continued).

downloaded. We choose the newest version for Microsoft Windows, which can be found in the top-most row of the list presented to us. At the time of this writing, this is version 17.2. We click the corresponding download icon.

The download begins, as shown in [Figure 3.2.4](#). Once it completes, we need to find the downloaded file, as shown in [Figure 3.2.5](#). I downloaded the software using the Microsoft Edge browser, so I need to click on the [...](#) button and then on [Downloads](#), or press [Ctrl+J](#). Regardless how you downloaded the installer file, once you found it, you need to execute it. As shown in [Figure 3.2.6](#), this can be accomplished by clicking on [Open file](#) in my case. The OS will now ask us whether we want to permit the downloaded program to make changes on our device. We certainly want that, because we want to use it to install PostgreSQL. So we click [Yes](#) in [Figure 3.2.7](#).

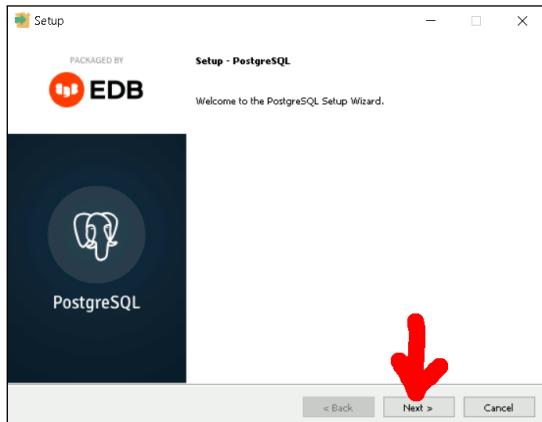
Then, the installer begins its work with a small splash screen shown in [Figure 3.2.8](#). In the following



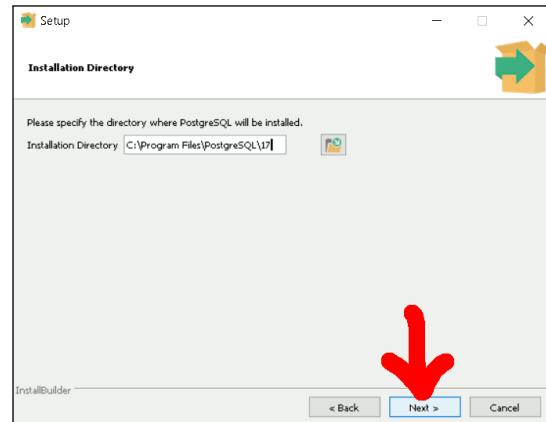
(3.2.7) When asked whether we want to allow the downloaded program to make changes on our device, we click [Yes].



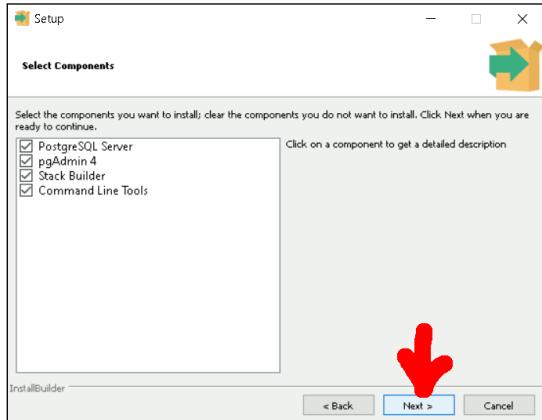
(3.2.8) Then, the installer begins its work.



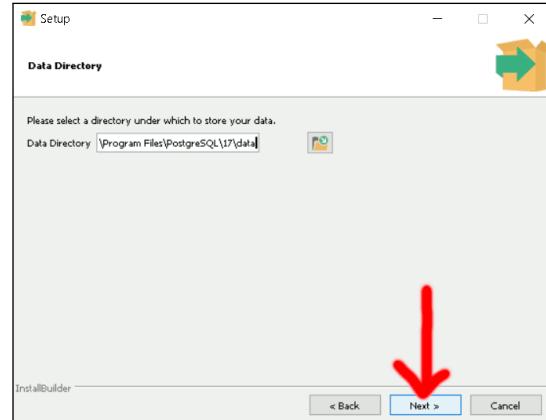
(3.2.9) In the welcome screen, we simply click [Next].



(3.2.10) We can select the directory in which PostgreSQL should be installed. Let's leave it at the default setting and click [Next].



(3.2.11) We now get to the selection of what to install. Let's leave it at the default setting and click [Next].



(3.2.12) We also leave the directory where the DBs will be stored at the default setting and click [Next].

Figure 3.2: Installing and configuring PostgreSQL under Microsoft Windows (Continued).

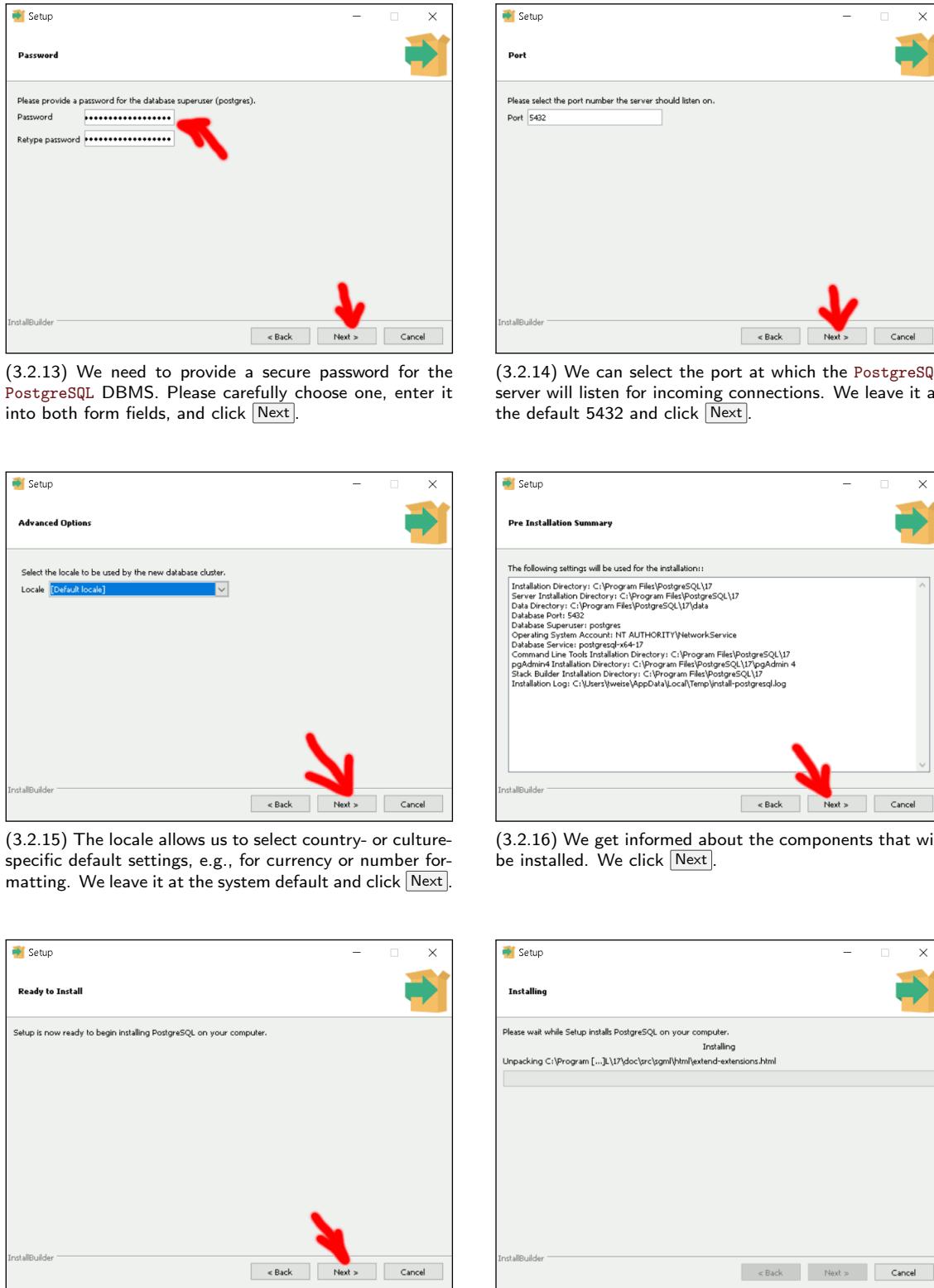
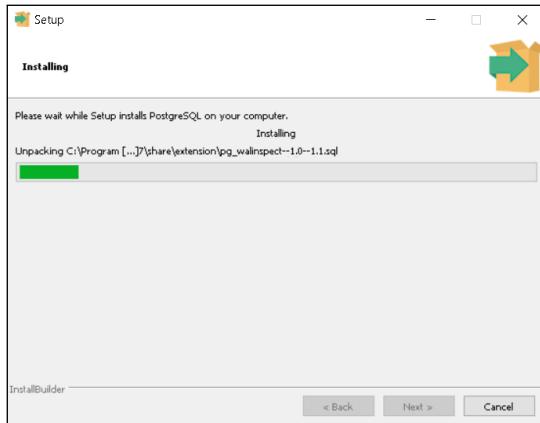
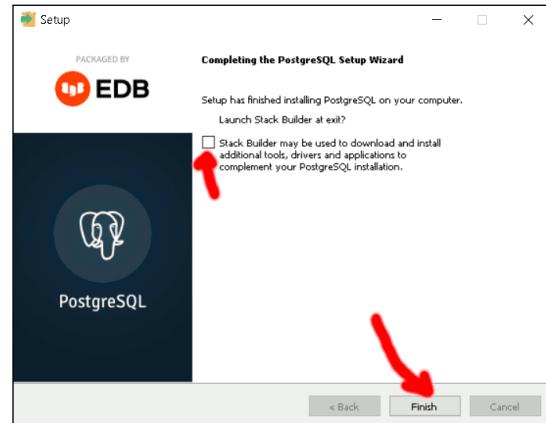


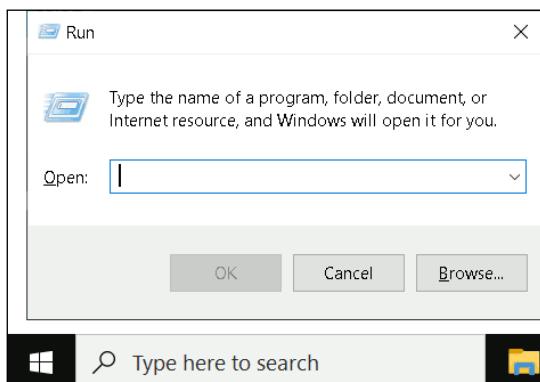
Figure 3.2: Installing and configuring PostgreSQL under Microsoft Windows (Continued).



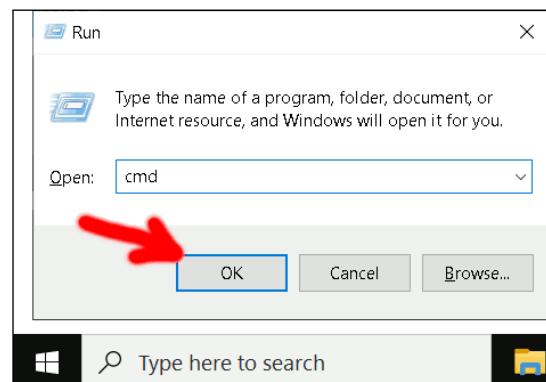
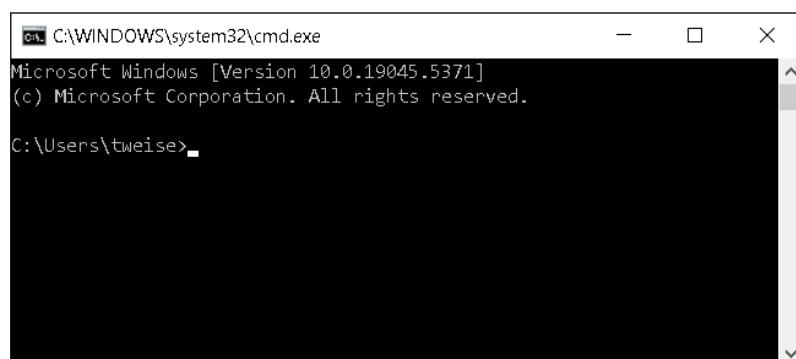
(3.2.19) The installation proceeds.



(3.2.20) Once the installation completes, we get asked whether we want to use the *Stack Builder* software to set up additional components. We do not want that, so we make sure that the checkbox is *not* checked. Then we click **Finish**. PostgreSQL is now installed.



(3.2.21) To validate the installation, we need to open a terminal. We therefore press **Windows + R**.

(3.2.22) We type in **cmd** and hit **Enter**.

(3.2.23) A new terminal window opens up.

Figure 3.2: Installing and configuring PostgreSQL under Microsoft Windows (Continued).

```
C:\WINDOWS\system32\cmd.exe
Microsoft Windows [Version 10.0.19045.5371]
(c) Microsoft Corporation. All rights reserved.

C:\Users\tweise>cd "c:\Program Files\PostgreSQL\17\bin"
```

(3.2.24) We need to enter the `bin` folder in the directory into where the PostgreSQL DBMS was installed. If we used the default settings, we can do that by typing `cd "C:\Program Files\PostgreSQL\bin"` and hitting `↵`.

```
C:\WINDOWS\system32\cmd.exe
Microsoft Windows [Version 10.0.19045.5371]
(c) Microsoft Corporation. All rights reserved.

C:\Users\tweise>cd "c:\Program Files\PostgreSQL\17\bin"

c:\Program Files\PostgreSQL\17\bin>
```

(3.2.25) We are now in that directory.

```
C:\WINDOWS\system32\cmd.exe
Microsoft Windows [Version 10.0.19045.5371]
(c) Microsoft Corporation. All rights reserved.

C:\Users\tweise>cd "c:\Program Files\PostgreSQL\17\bin"

c:\Program Files\PostgreSQL\17\bin>psql -V
```

(3.2.26) First, we want to print the version of `psql`, the client program of PostgreSQL. We can do this by typing `psql -V` and pressing `↵`.

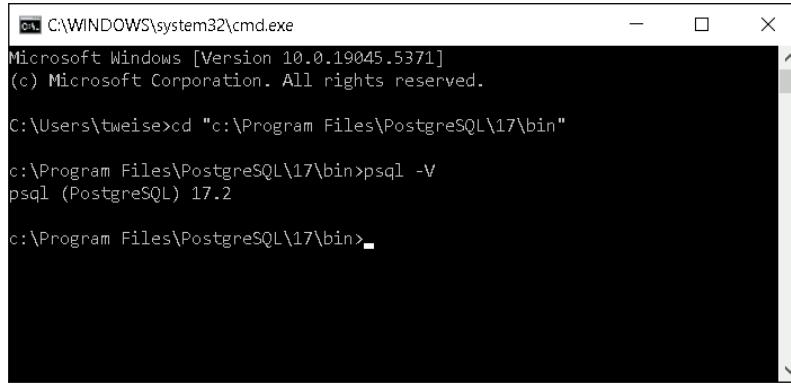
Figure 3.2: Installing and configuring PostgreSQL under Microsoft Windows (Continued).

welcome screen, we simply click `Next` (Figure 3.2.9). We now work our way through setting up the installation options step-by-step. First, we can select the directory in which PostgreSQL should be installed in Figure 3.2.10. I chose to just leave it at the default setting and click `Next`. This is what I will do for most of the rest of the installation, too.

We now get to the selection of what to install. I left this at the default setting, too, and simply click `Next` in Figure 3.2.11. Then, in Figure 3.2.12, we can also leave the directory where the DBs will be stored at the default setting and click `Next`.

In the following screen, we do need to provide a secure password for the PostgreSQL DBMS. Here we need to carefully choose a password *and remember it well*. We enter it into both form fields, and click `Next` in Figure 3.2.13.

We can now select the port at which the PostgreSQL server will listen for incoming connections. If we imagine a computer network as a transportation system, then the network protocol would be



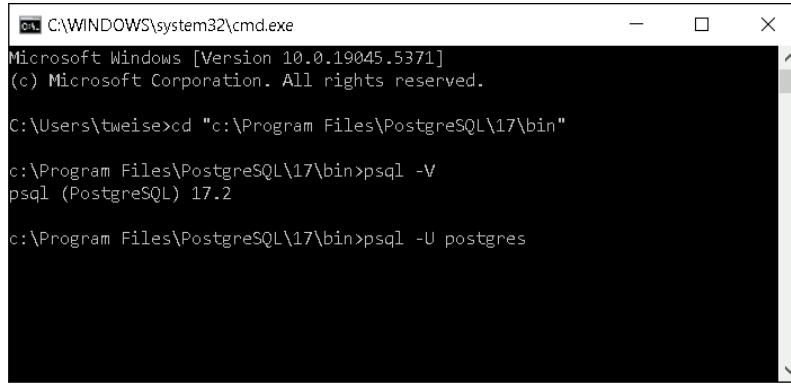
```
C:\WINDOWS\system32\cmd.exe
Microsoft Windows [Version 10.0.19045.5371]
(c) Microsoft Corporation. All rights reserved.

C:\Users\tweise>cd "c:\Program Files\PostgreSQL\17\bin"

c:\Program Files\PostgreSQL\17\bin>psql -V
psql (PostgreSQL) 17.2

c:\Program Files\PostgreSQL\17\bin>
```

(3.2.27) In my case, the output shows that version 17.2 was installed.

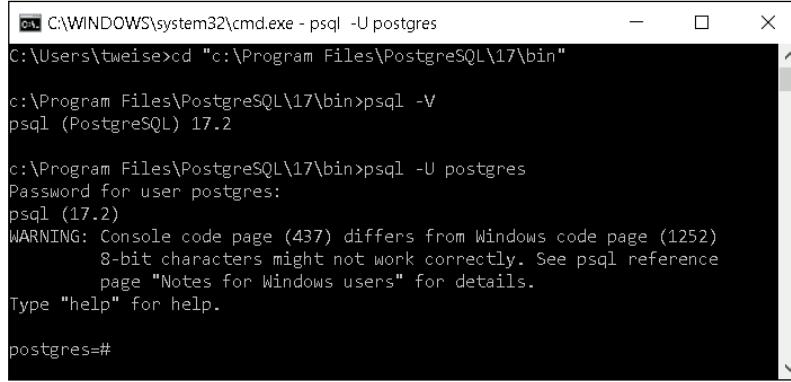


```
C:\WINDOWS\system32\cmd.exe
Microsoft Windows [Version 10.0.19045.5371]
(c) Microsoft Corporation. All rights reserved.

C:\Users\tweise>cd "c:\Program Files\PostgreSQL\17\bin"

c:\Program Files\PostgreSQL\17\bin>psql -V
psql (PostgreSQL) 17.2

c:\Program Files\PostgreSQL\17\bin>psql -U postgres
```

(3.2.28) We now want to also see which version of the PostgreSQL server was installed and, by doing so, also confirm that the server is up and running correctly. We therefore launch `psql -U postgres`, i.e., start psql using the user (or role) `postgres`.


```
C:\WINDOWS\system32\cmd.exe - psql -U postgres
C:\Users\tweise>cd "c:\Program Files\PostgreSQL\17\bin"

c:\Program Files\PostgreSQL\17\bin>psql -V
psql (PostgreSQL) 17.2

c:\Program Files\PostgreSQL\17\bin>psql -U postgres
Password for user postgres:
postgres (17.2)
WARNING: Console code page (437) differs from Windows code page (1252)
         8-bit characters might not work correctly. See psql reference
         page "Notes for Windows users" for details.
Type "help" for help.

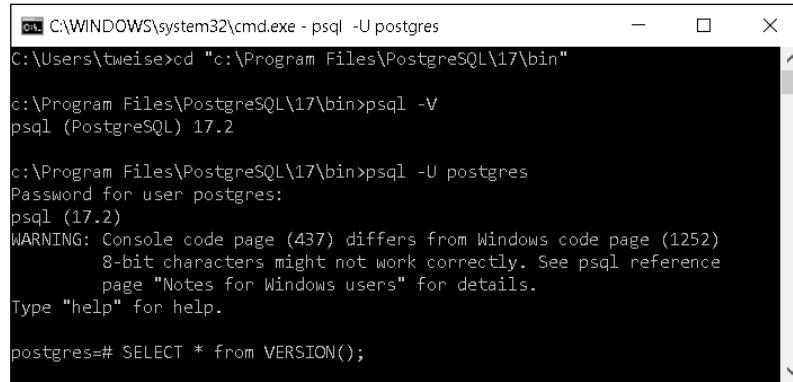
postgres=#
```

(3.2.29) When the program starts, it requires us to enter the password for the user `postgres`. This is the password that we specified during the installation in Figure 3.2.13. We enter it and press `Enter`. We are now in the psql console, and see the `postgres=#` prompt.

Figure 3.2: Installing and configuring PostgreSQL under Microsoft Windows (Continued).

the means of transportation, e.g., bus, airplane, or train. The network address would identify the station, e.g., Hefei Station or Beijing Station. Then the *port* would be something like the platform inside the station. Well, that is a very loose and very wrong analogy. Basically, the port is a number that identifies a communication partner relative to the network protocol and network address. Here, we specify the port at which the PostgreSQL server will expect incoming connections. We do not touch this parameter. We leave it at the default 5432 and click `Next` in Figure 3.2.14.

We directly encounter the next odd configuration parameter. We can choose the so-called *locale*. A locale identifies, basically, a region or country with specific cultural properties. This identifier is used to decide how numbers or currency or dates should be displayed. For instance, there is a crucial difference in British English and American English in how dates are written in numerical form: the former uses

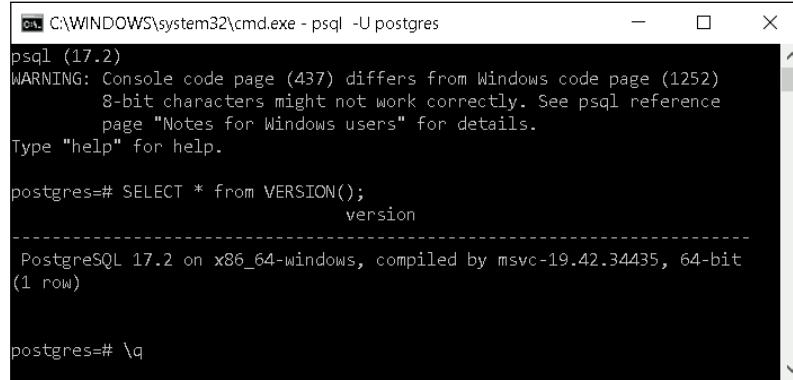


```
C:\WINDOWS\system32\cmd.exe - psql -U postgres
C:\Users\tweise>cd "c:\Program Files\PostgreSQL\17\bin"
c:\Program Files\PostgreSQL\17\bin>psql -V
psql (PostgreSQL) 17.2

c:\Program Files\PostgreSQL\17\bin>psql -U postgres
Password for user postgres:
psql (17.2)
WARNING: Console code page (437) differs from Windows code page (1252)
          8-bit characters might not work correctly. See psql reference
          page "Notes for Windows users" for details.
Type "help" for help.

postgres=# SELECT * from VERSION();
```

(3.2.30) We enter the SQL command `SELECT * FROM VERSION();` and press ↵.

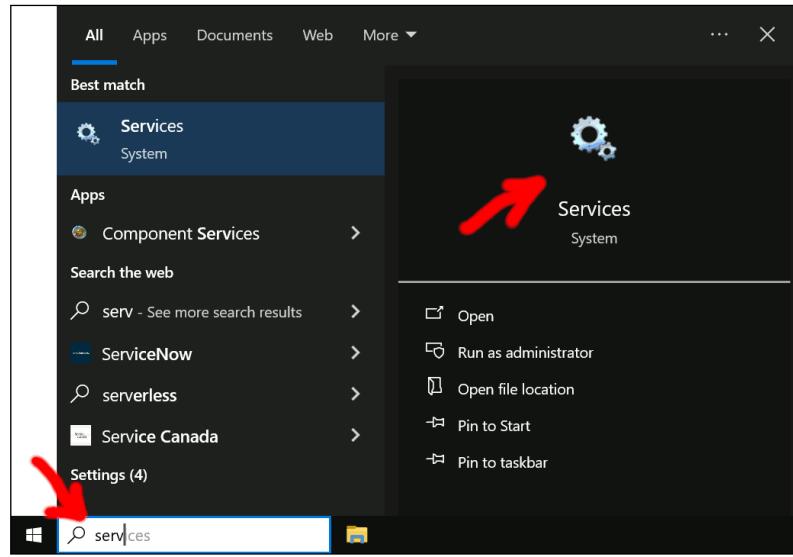


```
psql (17.2)
WARNING: Console code page (437) differs from Windows code page (1252)
          8-bit characters might not work correctly. See psql reference
          page "Notes for Windows users" for details.
Type "help" for help.

postgres=# SELECT * from VERSION();
           version
-----
 PostgreSQL 17.2 on x86_64-windows, compiled by msvc-19.42.34435, 64-bit
(1 row)

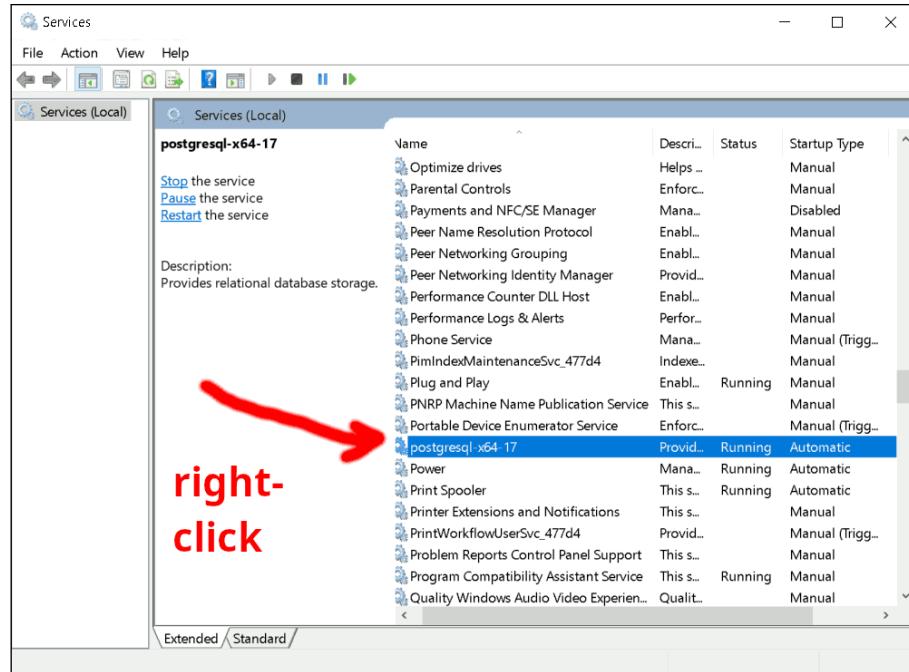
postgres=# \q
```

(3.2.31) The result gets printed. In my case, it shows that the PostgreSQL server also has version 17.2. We now type in ⌘+q+↵, which will exit psql.

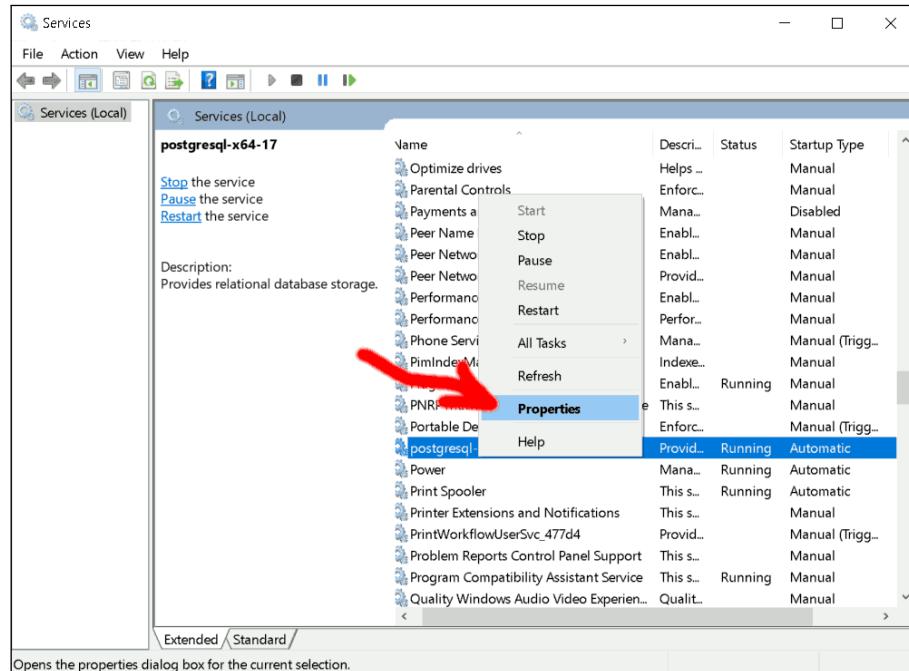


(3.2.32) Let us now explore how the PostgreSQL server is run on our Microsoft Windows machine: It is executed as a service. We press ⌘, type in `services`, and click on the "gears" symbol named `Services` that appears.

Figure 3.2: Installing and configuring PostgreSQL under Microsoft Windows (Continued).

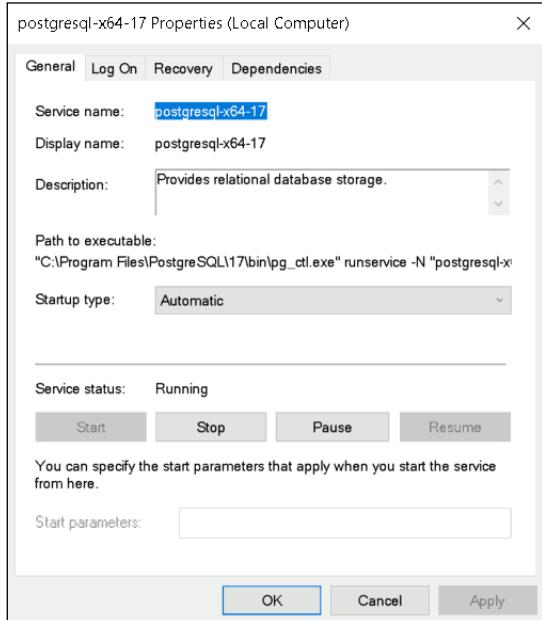


(3.2.33) The Services system setup window opens. Search for a service named something like PostgreSQL, in my case, it is `postgresql-x64-17`. Right-click on it.

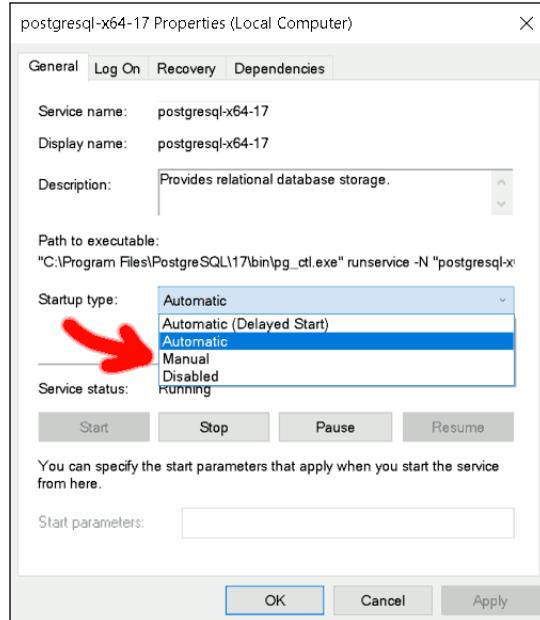


(3.2.34) In the Pop-up menu that appears, click on `Properties`.

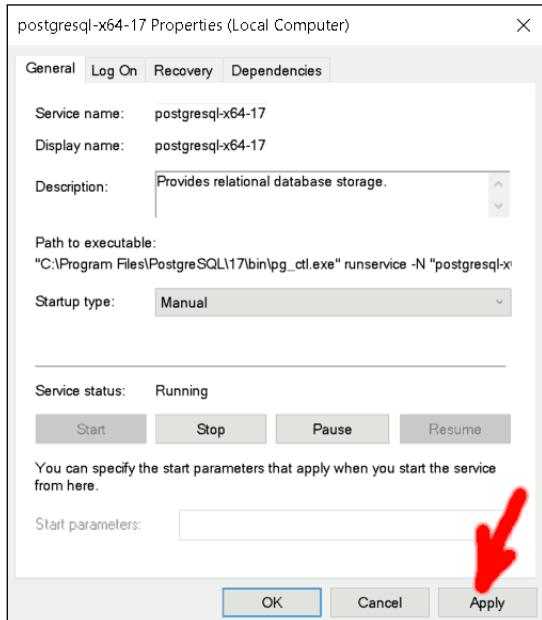
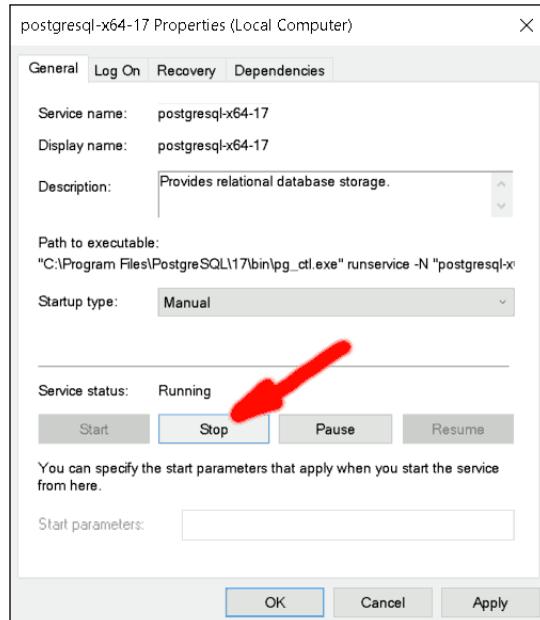
Figure 3.2: Installing and configuring PostgreSQL under Microsoft Windows (Continued).



(3.2.35) The service properties dialog appears.

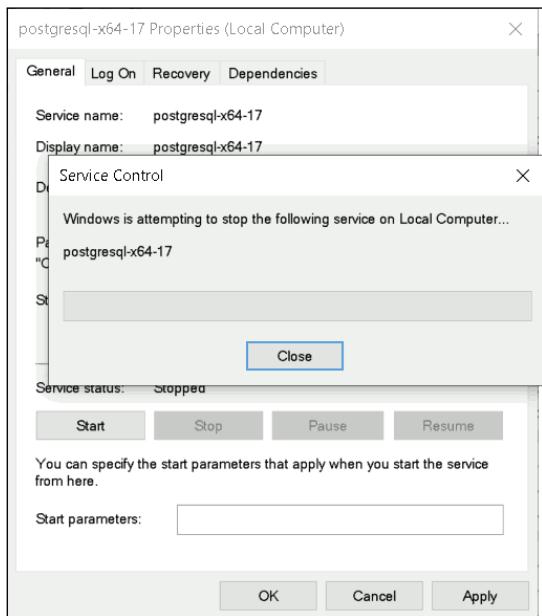


(3.2.36) Click on the drop-down box for [Startup type:]. It will be set to [Automatic], which means that the PostgreSQL DBMS server will be started automatically whenever your system starts.

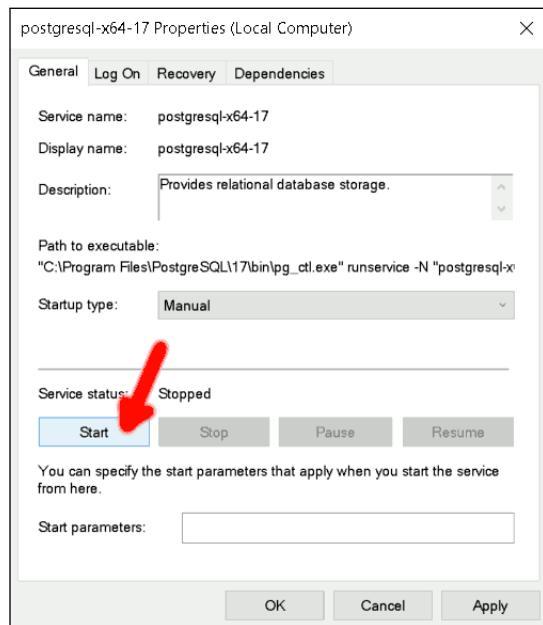
(3.2.37) The following is completely optional. You do not have to do that. If you do *not* want that, because it may make booting slower and maybe you only need the service studying for this course, you can turn off this automated startup. You would therefore select [Manual] as [Startup type:] and click the [Apply] button.

(3.2.38) The service is currently still running after that change, but it will not start automatically anymore. If you want it to start automatically again, just select [Automatic] as [Startup type:]. If you want to stop the currently-running service, click the [Stop] button.

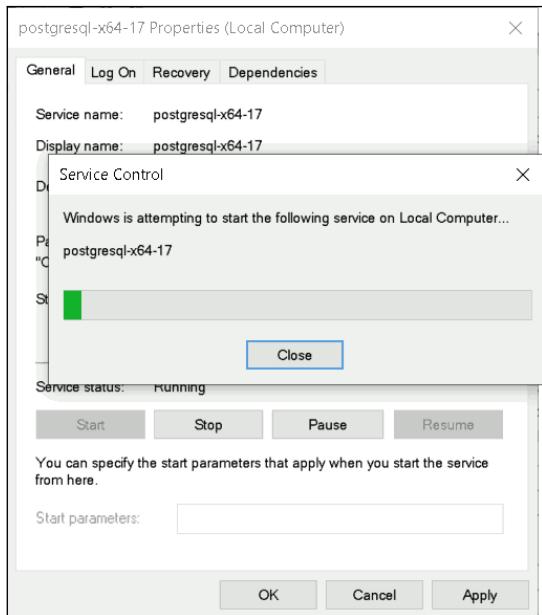
Figure 3.2: Installing and configuring PostgreSQL under Microsoft Windows (Continued).



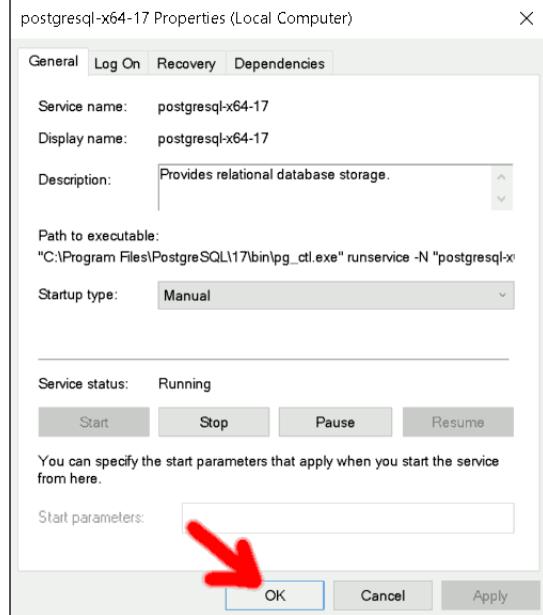
(3.2.39) Then, the service will be stopped.



(3.2.40) It is now no longer running. Let's start it again by clicking the [Start] button.



(3.2.41) The service is now starting again.



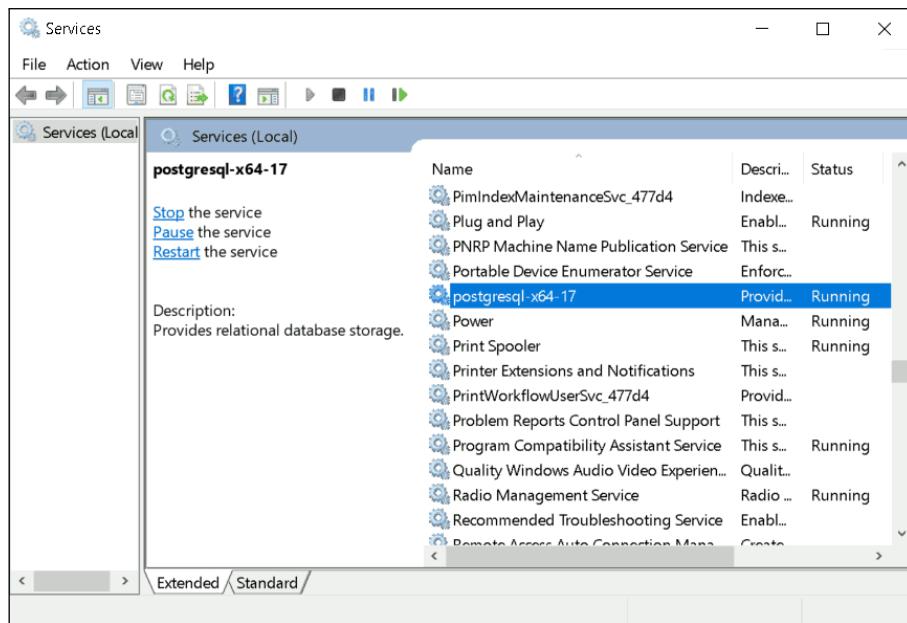
(3.2.42) We click [OK] to leave the dialog.

Figure 3.2: Installing and configuring PostgreSQL under Microsoft Windows (Continued).

the month/day/year scheme, whereas the latter uses day/month/year. If PostgreSQL is supposed to print dates in the form `.. / .. / ..`, it has to know whether it is running on an American or British English PC. Well, we leave it at the system default, which should be OK for use on our own PC, and click `Next` in Figure 3.2.15.

Now we get informed about the components that will be installed. We click `Next` in Figure 3.2.16. We get asked whether we are ready to install. We are, so we click `Next` in Figure 3.2.17. The installation begins and proceeds, as sketched in Figures 3.2.18 and 3.2.19.

Once the installation completes, we get asked whether we want to use the *Stack Builder* software to set up additional components. We do not want that, so we make sure that the checkbox is *not* checked. Then we click `Finish` in Figure 3.2.20. At this point, both the DBMS server and the client



(3.2.43) We see that the service is *Running* and in mode *Manual* (if we changed it to that mode). When you shutdown your system, the DBMS is stopped. It does not start automatically when you boot (unless you set *Automatic* as *Startup type*). So if you need it, you would again enter the *Services* system setup program and manually start it.

Figure 3.2: Installing and configuring PostgreSQL under Microsoft Windows (Continued).

`psql` of PostgreSQL are installed.

To validate the installation, we need to open a terminal. We therefore press `Windows + R`, type in `cmd`, and hit `Enter` in Figures 3.2.21 and 3.2.22. A new terminal window opens up in Figure 3.2.23. We here can write text-based commands and execute programs in this window.

To work with our new PostgreSQL installation, we first need to change into its installation folder and then into the folder where its executables are located. In other words, we need to enter the `bin` folder in the directory into where the PostgreSQL DBMS was installed. If we used the default settings, we can do that by typing `cd "C:\Program Files\PostgreSQL\bin"` and hitting `Enter` in Figure 3.2.24. We are now in that directory in Figure 3.2.25.

First, we want to print the version of `psql`, the client program of PostgreSQL. `psql` is the tool that we will use to communicate with the DBMS. We can get its version by typing `psql -V` and pressing `Enter` in Figure 3.2.26. In my case, the output shows that version 17.2 was installed, as illustrated in Figure 3.2.27.

We now want to also see which version of the PostgreSQL server was installed. By getting this information, we will also confirm that the server is up and running correctly. We therefore launch `psql -U postgres`, i.e., start `psql` using the user (or role) `postgres` in Figure 3.2.28. `postgres` is the username for the administrative account of our DBMS.

When the program starts with these parameters, it requires us to enter the password for this user (`postgres`). This is the password that we specified during the installation in Figure 3.2.13. We enter it and press `Enter`. We are now in the `psql` console, and see the `postgres=#` prompt in Figure 3.2.29.

We enter the SQL command `SELECT * FROM VERSION();` and press `Enter` in Figure 3.2.30. This is actual a command in the SQL language mentioned in the history section before. We will learn lots and lots of that later on. For now, you do not need to understand it. You only need to know that this line will print the version of the server for us. And the version gets indeed printed: In my case, it shows that the PostgreSQL server also has version 17.2. We now type in `\q` + `Enter`, which will exit `psql` in Figure 3.2.31. This will take us back to the normal Microsoft Windows terminal. We no longer have any use for it and can close it.

We now have a PostgreSQL DBMS server running on our system. What does that mean? How is it running? What if I restart my computer? Will it still be running? Will it always be running? Let us investigate these questions.

The PostgreSQL server is executed as a service in our OS. So we will switch over to the *Services* configuration window of Microsoft Windows. We therefore press , type in `services`, and click on the “gears” symbol named *Services* that appears in Figure 3.2.32. The *Services* system setup window opens. We scroll down and search for a service named something like PostgreSQL, in my case, it is `postgresql-x64-17`. We right-click on this service, i.e., we click with the mouse button on the right, in Figure 3.2.33. In the Pop-up menu that appears, click on `Properties` in Figure 3.2.34.

This opens the service properties dialog for PostgreSQL in Figure 3.2.35. We click on the drop-down box for `Startup type:` in Figure 3.2.36. Here we see the different options regarding whether and how a service is started. Right now it is set to `Automatic`, which means that the PostgreSQL DBMS server will be started automatically whenever your system starts.

The following is completely optional. You do not have to do that. If you do *not* want that the service always starts automatically, because it may make booting slower and maybe you only need the service studying for this course, you can turn off this automated startup. Maybe you want that the server is normally turned off. Whenever you need it, you want to turn it on by yourself. If that is your goal, then you would probably want to select `Manual` as `Startup type:` and click the `Apply` button, as shown in Figure 3.2.37. This means that, the next time your system boots, PostgreSQL will not be started automatically anymore.

The service is currently still running after we made that change, but it will not automatically start anymore. If you want it to start automatically start again, just select `Automatic` as `Startup type:`. If you want to stop the currently-running service, click the `Stop` button, as shown in Figure 3.2.38. Then, the service will be stopped, as you can see in Figure 3.2.39. After that, it is no longer running

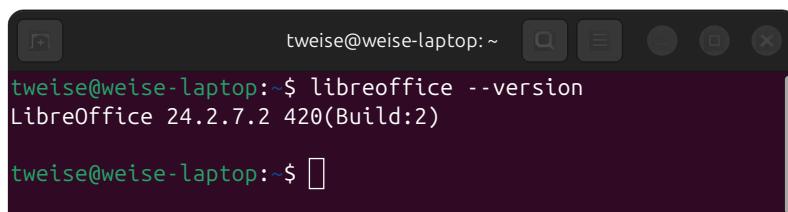
Let's start it again by clicking the `Start` button in Figure 3.2.40. The service is now starting again, as illustrated in Figure 3.2.41. We click the `OK` button to leave the dialog in Figure 3.2.42. We see that the service is *Running* and in mode *Manual* (if we changed it to that mode) in Figure 3.2.43. When you shutdown your system, the DBMS is stopped. If it is in *Manual* mode, the DBMS does not start automatically when you boot. If you set it back to *Automatic* as *Startup type*, then it will start automatically again. So if you need it, you would again enter the *Services* system setup program and manually start it.

With this we depart from the magical world of PostgreSQL server installation under Microsoft Windows. Our work here is done.

Chapter 4

Installing LibreOffice

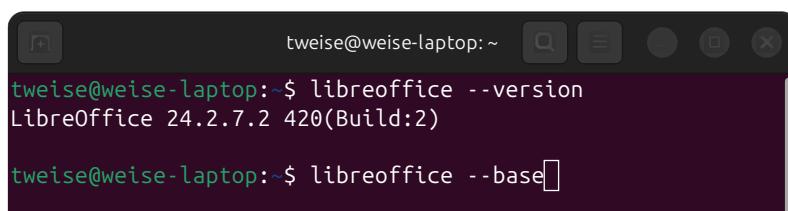
In our excursion on the available DBMS in Section 1.3, we mentioned that the open source office suite **LibreOffice** provides program named **LibreOffice Base**. LibreOffice Base can be used as either a stand-alone DBMS or as a nice user interface that can connect to other DBs [160, 385]. It has a functionality similar to **Microsoft Access** [31, 86, 457], but it is free. We will therefore use this software to connect to **PostgreSQL**, for example in Chapter 13. Let us therefore discuss how LibreOffice [171, 270, 385] can be installed.



```
tweise@weise-laptop:~$ libreoffice --version
LibreOffice 24.2.7.2 420(Build:2)

tweise@weise-laptop:~$ 
```

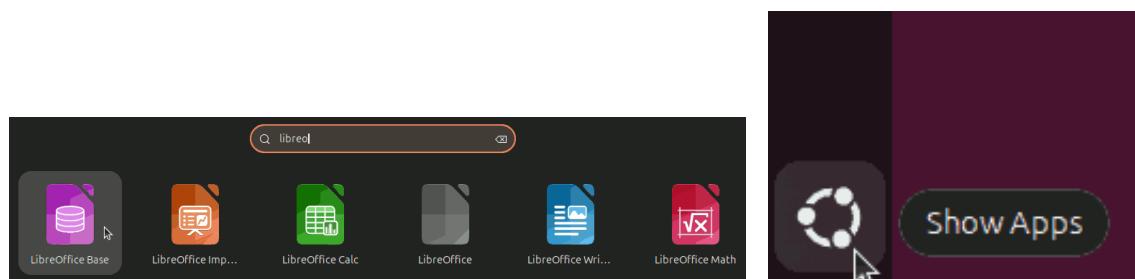
(4.1.1) Open a terminal via with **Ctrl** + **Alt** + **T**, then type **libreoffice --version** and hit **Enter**. On my system, LibreOffice 24.2.7.2 is installed.



```
tweise@weise-laptop:~$ libreoffice --version
LibreOffice 24.2.7.2 420(Build:2)

tweise@weise-laptop:~$ libreoffice --base
```

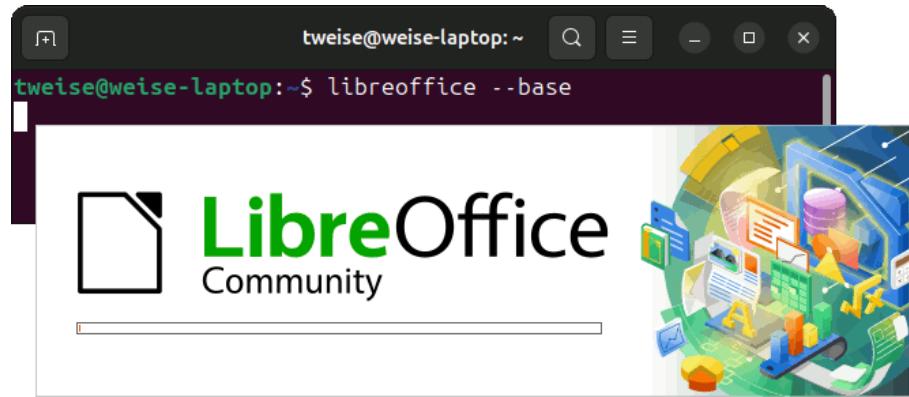
(4.1.2) We can start LibreOffice Base from the terminal by typing **libreoffice --base** and hitting **Enter**.



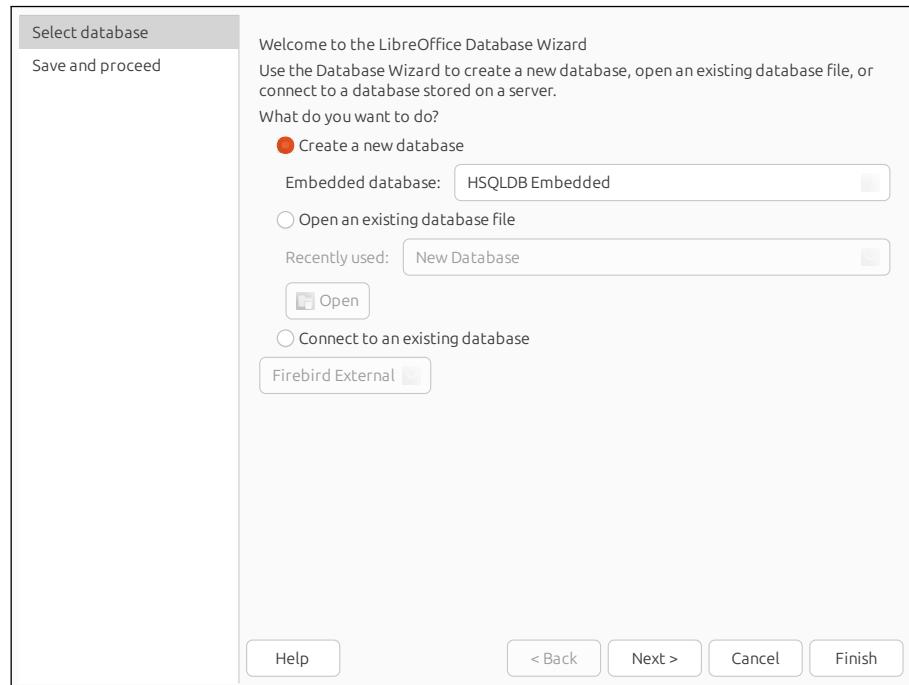
(4.1.3) Or we can open the dash by hitting **Windows**, type **libreoffice**, and then clicking on the symbol for LibreOffice Base.

(4.1.4) We can also open the dash by clicking on the little Ubuntu symbol at the bottom-left corner of the screen.

Figure 4.1: Starting LibreOffice Base under Ubuntu if it is already installed.



(4.1.5) The startup screen appears.



(4.1.6) The initial form of LibreOffice Base appears. We are done for now and close the program.

Figure 4.1: Starting LibreOffice Base under Ubuntu if it is already installed.

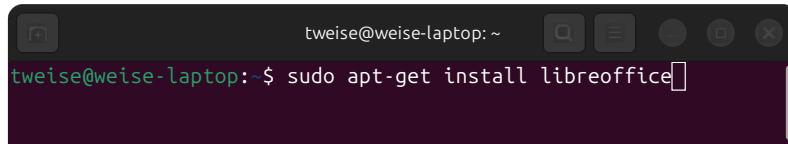
4.1 Installing LibreOffice under Ubuntu Linux

If you are a user of Ubuntu [Linux](#), then LibreOffice and, hence, LibreOffice Base, come already pre-installed on your machine. You do not actually need to do anything.

To confirm that LibreOffice is indeed installed, we open a [terminal](#) via with [**Ctrl**+**Alt**+**T**](#). We type `libreoffice --version` and hit [**Enter**](#). On my system, the result in Figure 4.1.1 shows that LibreOffice 24.2.7.2 is installed.

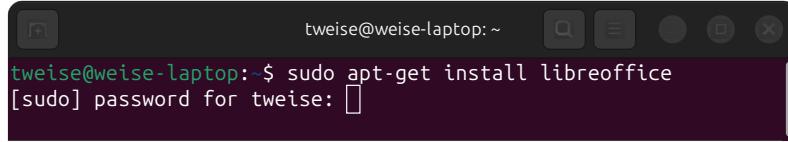
We can also start LibreOffice Base directly from the terminal by typing `libreoffice --base` and hitting [**Enter**](#), as shown in Figure 4.1.2. Alternatively, we can open the dash by hitting [**Ubuntu Dash**](#), type `libreoffice`, and then clicking on the symbol for LibreOffice Base, as illustrated in Figure 4.1.3. If you do not want to use a hotkey to open the dash, you can also click on the little [**Ubuntu**](#) symbol at the bottom-left corner of the screen (see Figure 4.1.4). Regardless of whether you opened LibreOffice Base via a terminal or the dash, the startup screen given in Figure 4.1.5 will appear. Then, the initial form of LibreOffice Base is displayed as shown in Figure 4.1.6. We are done for now and close the program.

In the unlikely case that LibreOffice was not installed on your system, the above will obviously not work. However, Figure 4.2.1 shows that we can easily install it by typing `sudo apt-get install libreoffice` into a terminal window that we opened with [**Ctrl**+**Alt**+**T**](#).



```
tweise@weise-laptop:~$ sudo apt-get install libreoffice
```

(4.2.1) In the unlikely case that LibreOffice was not installed, we can install it by typing `sudo apt-get install libreoffice` into a terminal window that we opened with $\text{Ctrl}+\text{Alt}+\text{T}$ and hit Enter .



```
tweise@weise-laptop:~$ sudo apt-get install libreoffice
[sudo] password for tweise: 
```

(4.2.2) This requires `sudo` privileges, so we need to enter the super user password.



```
tweise@weise-laptop:~$ sudo apt-get install libreoffice
[sudo] password for tweise:
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
libreoffice is already the newest version (4:24.2.7-0ubuntu0.24.04.1).
0 upgraded, 0 newly installed, 0 to remove and 4 not upgraded.
tweise@weise-laptop:~$ 
```

(4.2.3) Now LibreOffice could be installed. On my system, it is already installed. So nothing happens.

Figure 4.2: Installing LibreOffice under Ubuntu.

and hit Enter . This requires `sudo` privileges, so we need to enter the super user password in Figure 4.2.2. Now you would probably be told how much data will need to be downloaded, then asked whether you are OK with that, and then LibreOffice would be installed. Since it is already installed on my system, nothing happens. as shown in Figure 4.2.3.

The gist is that LibreOffice is usually already installed on Ubuntu Linux or if not, can easily be installed.

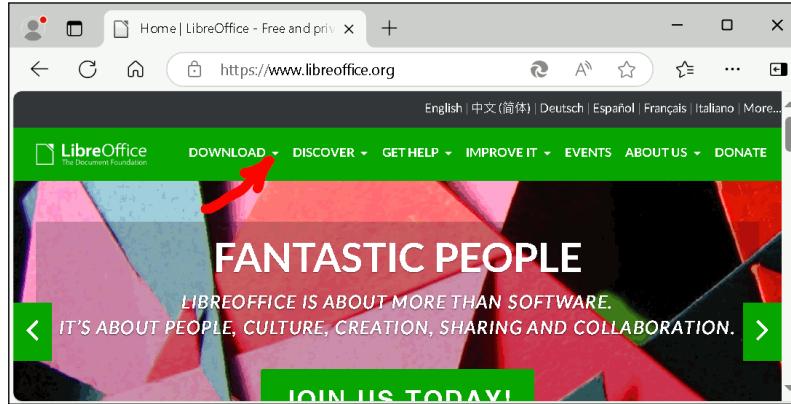
4.2 Installing LibreOffice under Microsoft Windows

Installing LibreOffice under Microsoft Windows requires us to first download the software and then to install it. We therefore open a web browser and go to <https://libreoffice.org> in Figure 4.3.1. There, we click on `DOWNLOAD`. A drop-down menu opens in Figure 4.3.2. We click on `Download LibreOffice`. This takes us to the download page, where we, again, click on `DOWNLOAD` in Figure 4.3.3. On this page, we could probably select the operating system of our choice. Unless you are using some other operating system, the default choice, `Windows (64-bit)`, is probably correct and we can leave it as is. This takes us to yet another download page in Figure 4.3.4, where we click the big button for downloading LibreOffice. Finally, the download starts in Figure 4.3.5.

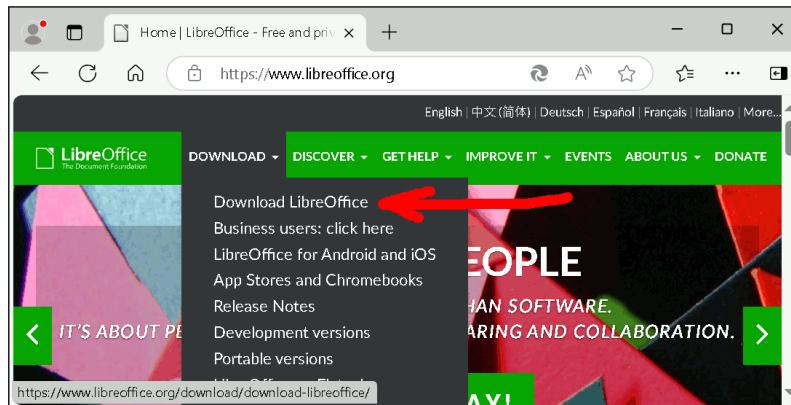
After the download completes, we open, i.e., run the downloaded file, as shown in Figure 4.3.6. A Microsoft Windows installer loading screen appears in Figure 4.3.7. You see, we downloaded a new program from the internet and not the Microsoft Store. So this worries our Microsoft Windows installation, and it asks whether we *really* want to install this program in Figure 4.3.8. In Figure 4.3.9, we click on `Install anyway`.

The installation wizard window appears and we click `Next` in Figure 4.3.10. On the next screen, we can choose whether we want to customize the installation of LibreOffice or, instead, would prefer a “typical”. We indeed prefer the “typical installation” and click `Next` in Figure 4.3.11. In the next screen in Figure 4.3.12 we confirm that we are also OK with a shortcut on our desktop and click `Install`.

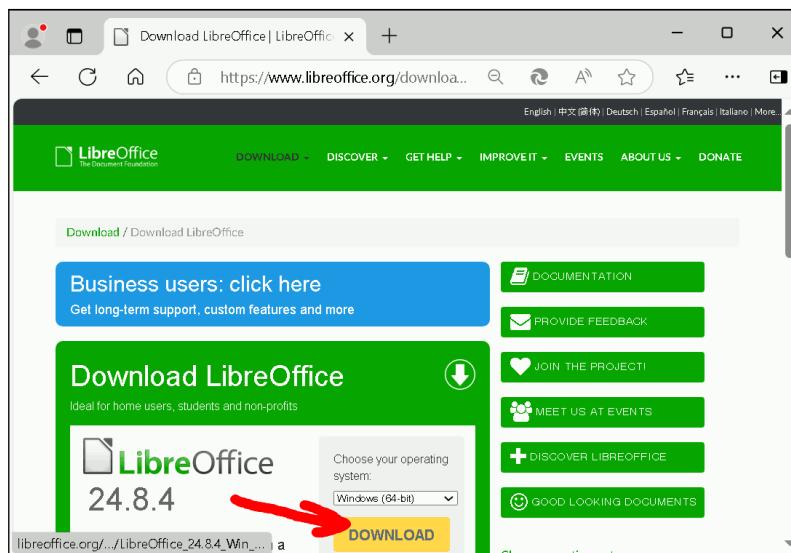
Now The installation begins (see Figure 4.3.13). Since the installer tries to modify our system, Microsoft Windows asks us whether we would like to permit the installer to make changes to our



(4.3.1) We open a web browser and go to <https://libreoffice.org>. There, we click on [DOWNLOAD].

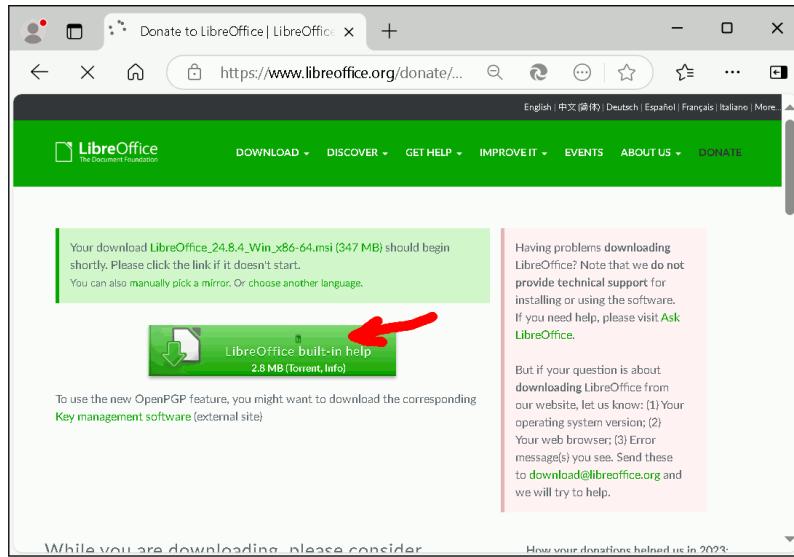


(4.3.2) In the menu that opens, we click on [Download LibreOffice].

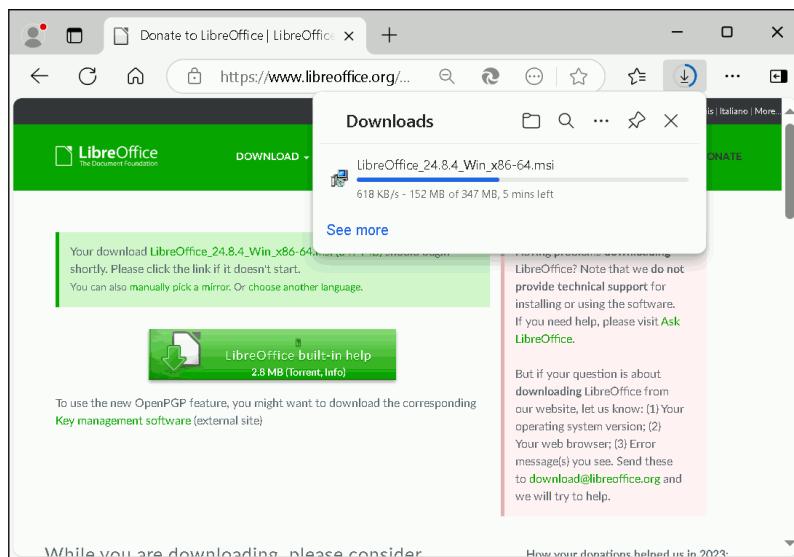


(4.3.3) This takes us to the download page, where we click [DOWNLOAD]. (We can leave the operating system at the default [Windows (64-bit)] setting.)

Figure 4.3: Installing LibreOffice under Microsoft Windows.



(4.3.4) This takes us to yet another download page, where we click the big button for downloading LibreOffice.



(4.3.5) The download starts.

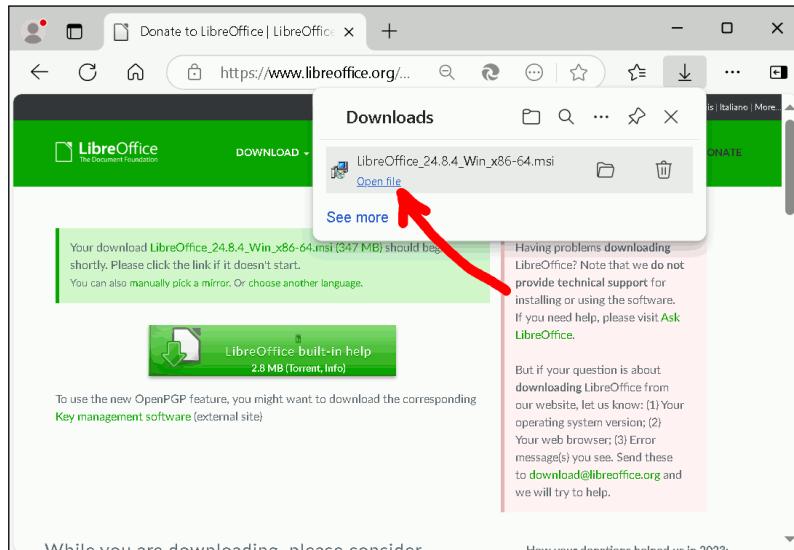
Figure 4.3: Installing LibreOffice under Microsoft Windows (Continued).

system. We indeed are OK with that, so we click **Yes** in Figure 4.3.14. The installation continues in Figure 4.3.15.

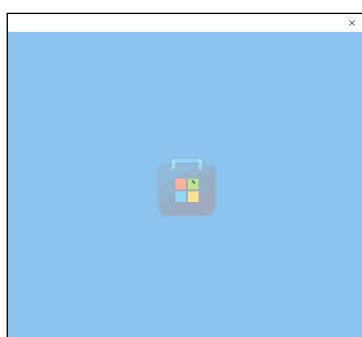
Under some circumstances, e.g., if you have the Acrobat Reader installed, it may be necessary that the installer does some complex updating. It is best to keep the option *Do not close applications*. A *reboot will be required to complete the setup*. I also tried the other option, but that leads to errors down the line. It is best to leave this at the default setting. So we just click **OK** in Figure 4.3.16. On your system, maybe this screen does not appear. Maybe you do not have Adobe Acrobat installed, maybe you have a different version or setup. Thus, maybe you will never see this screen and, as a result, do not need to reboot later. If so, good for you. If you see the screen, then just accept that you will need to reboot eventually.

Either way, the installation continues in Figure 4.3.17. And eventually, it is completed and we click **Finish** in Figure 4.3.18. At this stage, if you did see the screen from Figure 4.3.16, it becomes necessary to reboot. If so, you should close all other programs and click **Yes**, as shown in Figure 4.3.19. Then, the restart screen appears in Figure 4.3.20.

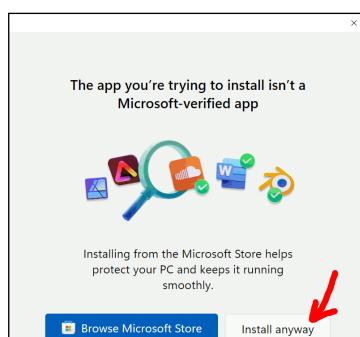
Regardless whether you needed to reboot or not, we can now find a LibreOffice icon on the desktop, as shown in Figure 4.3.21. We click on it. The LibreOffice splash screen appears in Figure 4.3.22. We



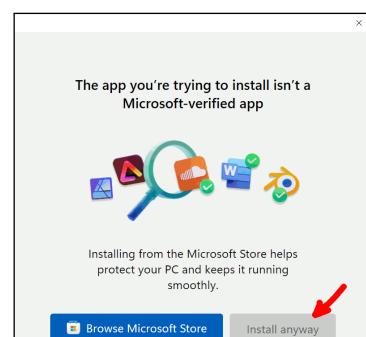
(4.3.6) After the download completes, we open and run the file.



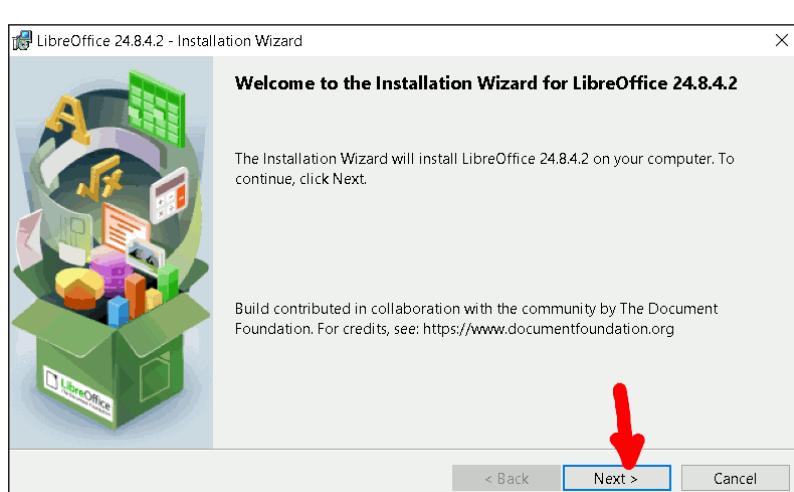
(4.3.7) A Microsoft Windows installer loading screen appears.



(4.3.8) We get asked whether we really want to install this program...

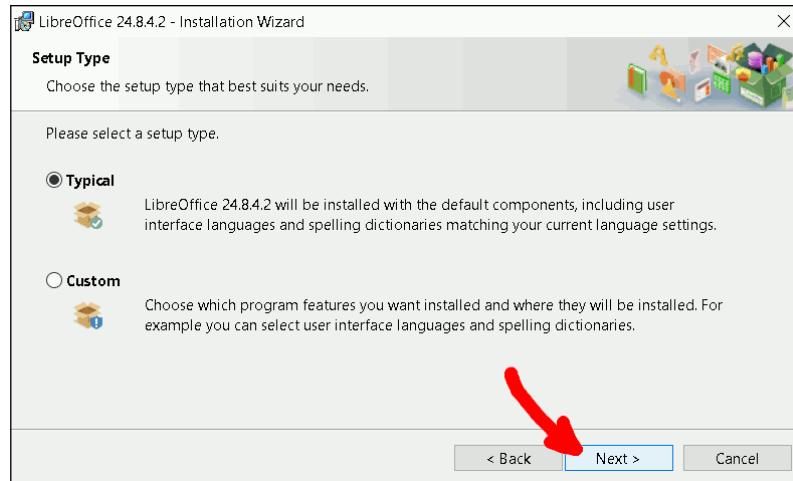


(4.3.9) ...and we click [Install anyway].

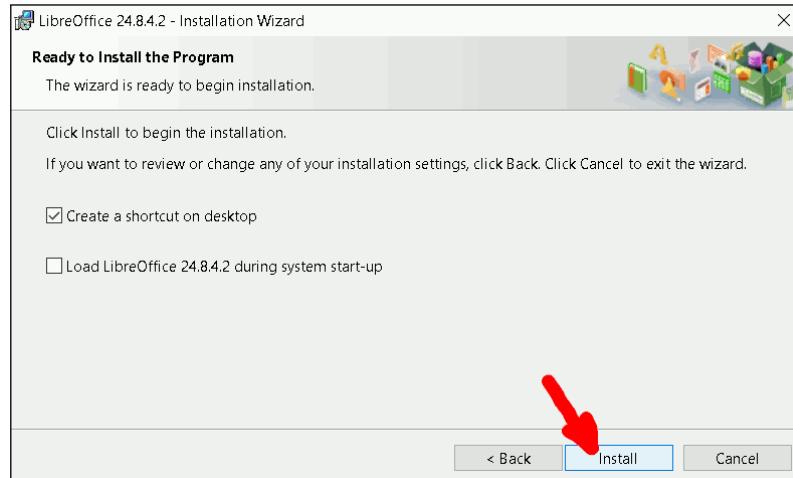


(4.3.10) The installation wizard window appears and we click [Next].

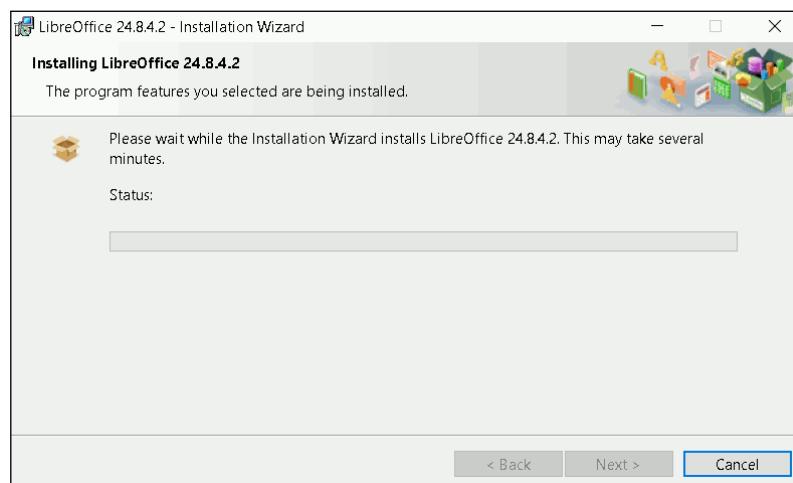
Figure 4.3: Installing LibreOffice under Microsoft Windows (Continued).



(4.3.11) We are totally fine with a “typical” installation and click [Next].

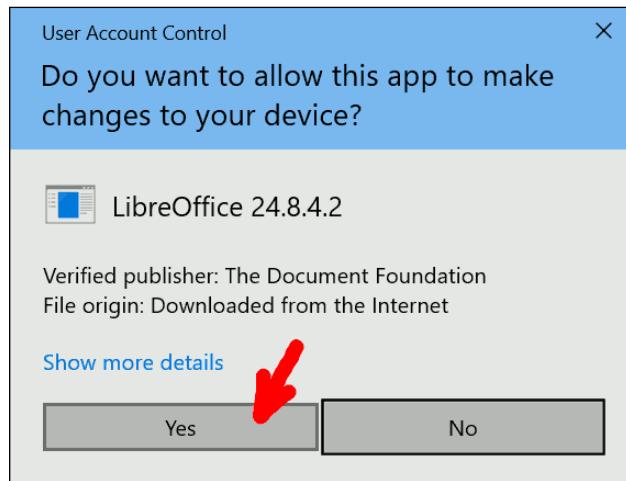


(4.3.12) We are also OK with a shortcut on our desktop and click [Install].

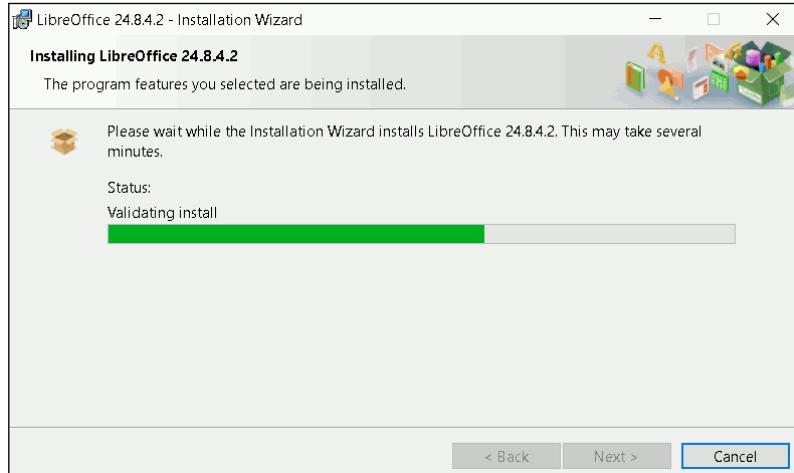


(4.3.13) The installation begins.

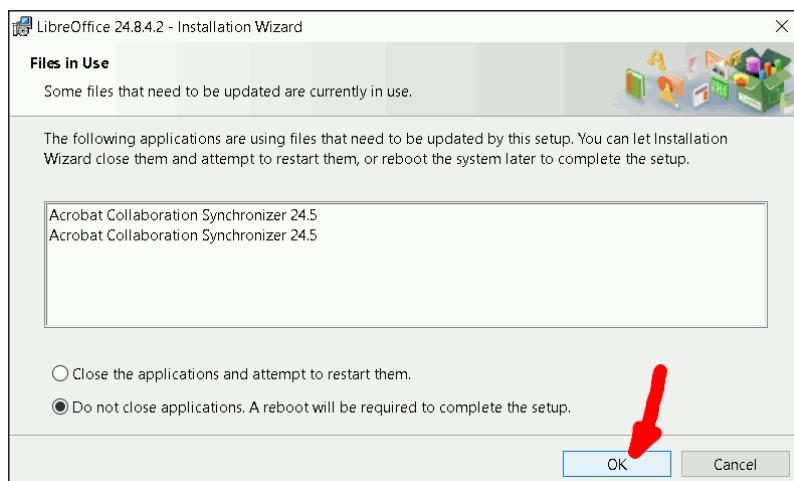
Figure 4.3: Installing LibreOffice under Microsoft Windows (Continued).



(4.3.14) Microsoft Windows asks us whether we would like to permit the installer to make changes to our system. Yes, we are, so we click **Yes**.

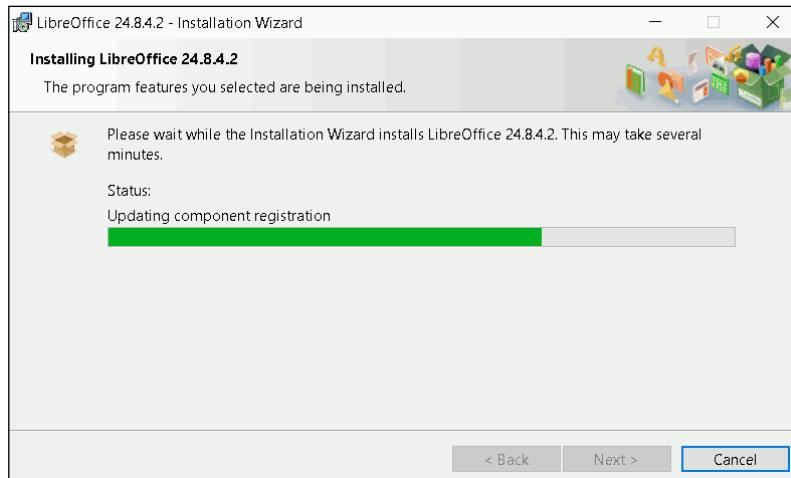


(4.3.15) The installation continues.

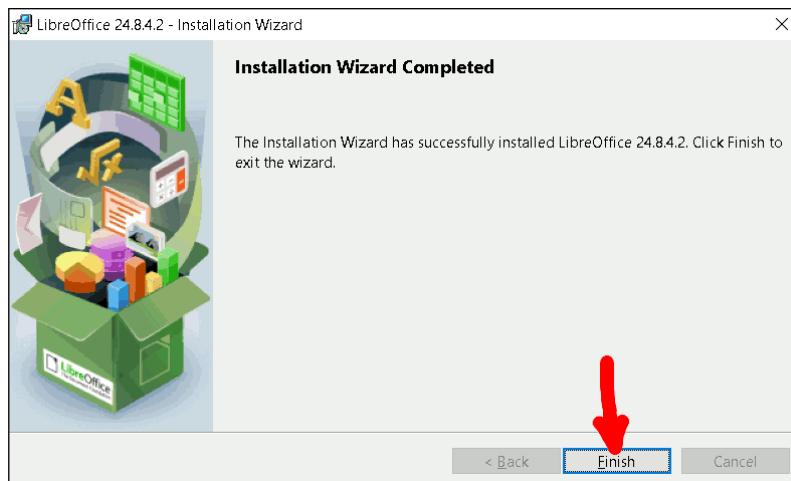


(4.3.16) Under some circumstances, e.g., if you have the Acrobat Reader installed, it may be necessary that the installer does some complex updating. It is best to keep the option *Do not close applications*. A reboot will be required to complete the setup. So we just click **OK**. On your system, maybe this screen does not appear.

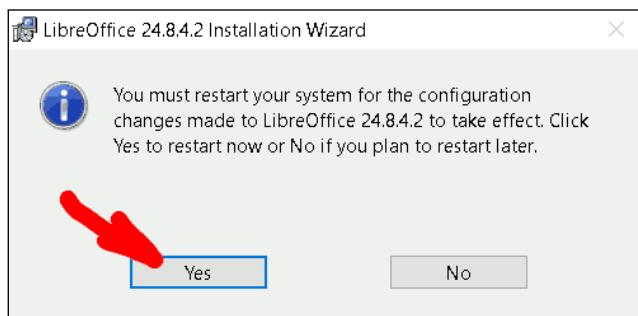
Figure 4.3: Installing LibreOffice under Microsoft Windows (Continued).



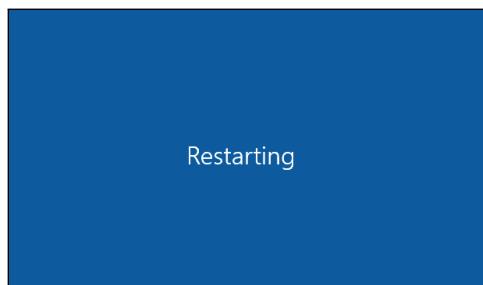
(4.3.17) The installation continues.



(4.3.18) The installation is completed. We click [Finish].

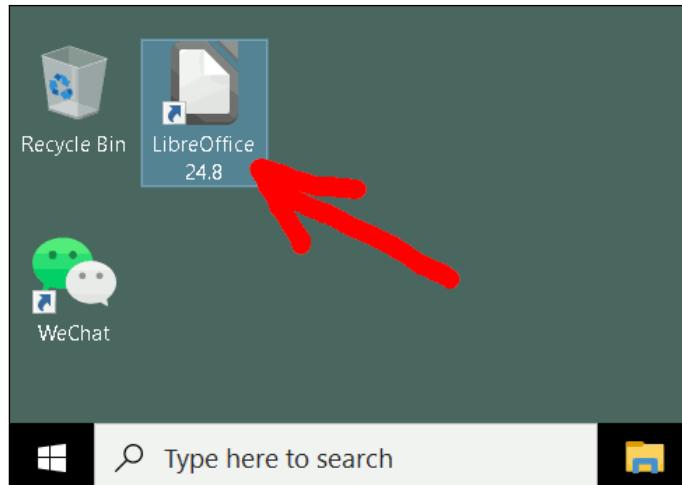


(4.3.19) It may be necessary to reboot (see Figure 4.3.16). If so, close all other programs and click [Yes].



(4.3.20) The restart screen appears

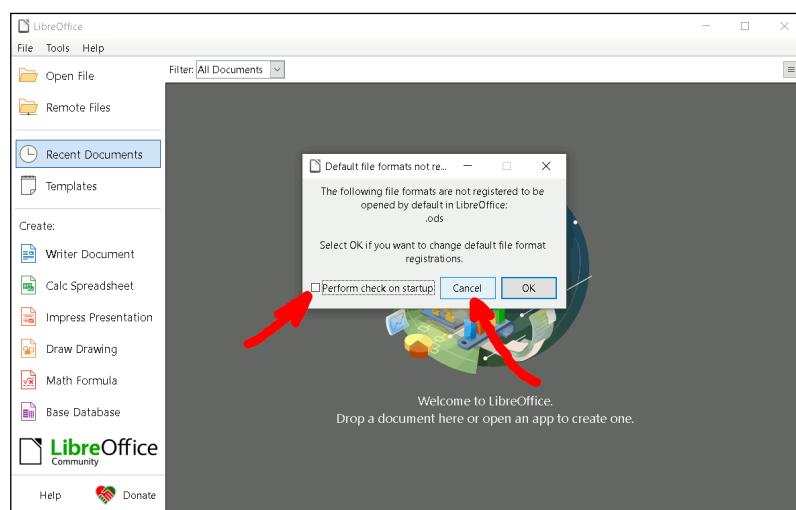
Figure 4.3: Installing LibreOffice under Microsoft Windows (Continued).



(4.3.21) We can now find a LibreOffice icon on the desktop and click on it.

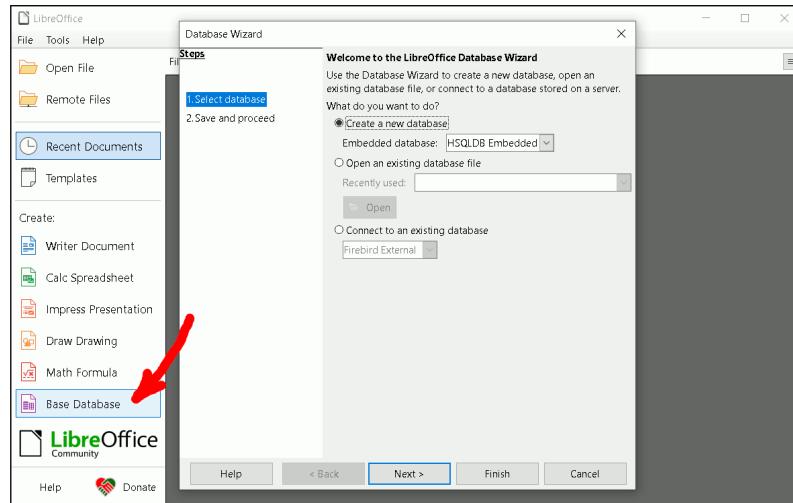


(4.3.22) The LibreOffice splash screen appears.



(4.3.23) We may get asked to set LibreOffice as default program to open some file types. In order to not mess with your existing system configuration, we un-check the *Perform check on startup* box and click [Cancel].

Figure 4.3: Installing LibreOffice under Microsoft Windows (Continued).



(4.3.24) We click on the **Base Database** icon in the menu bar on the left-hand side and arrive in the LibreOffice Base welcome screen. We are done here for now and close the program.

Figure 4.3: Installing LibreOffice under Microsoft Windows (Continued).

may get asked to set LibreOffice as default program to open some file types. In order to not mess with your existing system configuration, we un-check the *Perform check on startup* box and click on **Cancel** in [Figure 4.3.23](#).

We now click on the **Base Database** icon in the menu bar on the left-hand side and arrive in the LibreOffice Base welcome screen in [Figure 4.3.24](#). We are done here for now and close the program. We have downloaded and installed LibreOffice Base and can use it for our experiments later.

Chapter 5

Installing Python, PyCharm, and Psycopg

With the software we installed so far, we have two options to work with DBs: We can use a command line client (like `psql`) or we can use a convenient GUI (like `LibreOffice Base`). Nevertheless, the maybe most likely method to work with a DB is programmatically. Very often, high-level functionality and logic is implemented in program code on top of the SQL queries that drive the DBs. Applications can form a middle tier or top tier of an enterprise software architecture.

Therefore, in this book, we will also learn how to access and work with a DB from normal program code. If you are reading this book as part of the *Databases* course [481] at our Hefei University (合肥大学), then you may be aware that I am also teaching a course *Programming with Python* [482]. Thus it will not be a surprise that we will use `Python` [215, 265, 482] as the programming language of choice for accessing a DB. The book [482] that accompanies the *Programming with Python* course is also free and open source. It too comes with examples available in a `GitHub` repository.

If you want to install the Python programming language on your system, you can follow the instructions given in [482]. In [482], we also recommend using `PyCharm` as IDE for writing code. Of course, we also describe how to install that IDE. We will not reproduce these installation instructions here. You can look them up in [482]. In that book, we also discuss how to work with `Git` repositories and how to even download the sources of our book right here from <https://github.com/thomasWeise/databasesCode>. We even describe how to install the required packages to run this code!

Our example codes for accessing `PostgreSQL` DBs from Python do require one package: `psycopg` [473]. This library provides a bridge between the Python programming language and `PostgreSQL`. It implements the Python DB API 2.0 specification [268], which means that code written using `psycopg` for accessing `PostgreSQL` may be reusable with another library and DBMS, given a few necessary changes. Either way, if you follow the instructions in *Programming with Python* [482] in the chapter on cloning Git repositories using `PyCharm`, you will directly learn how to install this library. We will use Python and `psycopg` in Chapter 12.

5.1 Installing Psycopg

Since the installation of the programming language Python and the IDE `PyCharm` is already covered in [482], we will not reproduce this information here. However, the installation of `psycopg` is specific to this course, so we will discuss it in a bit more detail.

There are two ways to install this library: If you execute a DB application written in Python in a productive environment, you will install `psycopg` into a virtual environment via the command line in a terminal. Once this is done, you would activate the virtual environment and be able to execute your application. If you are either developing software locally or just want to explore the examples of this course, then you could install `psycopg` within a virtual environment under the `PyCharm` IDE. Both approaches to install packages have been covered in [482].

Nevertheless, since they are integral to the contents of this course, we will also discuss them here. It would best, however, if you would read [482] to a point where you know what `pip` is and what virtual environments are. We also cover the installation only under `Ubuntu Linux`, as the installation steps under `Microsoft Windows` will be analogous (see again [482]).

The most common way to use external packages in Python is to install them into virtual environments. A virtual environment is essentially something like a directory hosting a stand-alone Python

installation separate from the Python setup of the computer. This allows you to have different versions of different libraries installed (in different virtual environments).

Imagine that you have one program that requires the [Python](#) package [NumPy](#) in version 2.0 or above to run. You also need to use another application, which can only run with NumPy *below* version 2.0. This is a very common scenario. It would be impossible to realize, because you can only have one version of NumPy installed in your system. However, you can use both applications if each is installed into its own [virtual environment](#). Because each virtual environment can have its own version of NumPy. Therefore, regardless whether we want to install [psycopg](#) for productive use via the [terminal](#) or whether we instead want to use it within [PyCharm](#) to develop software, we will use virtual environments.

5.1.1 Installing Psycopg via the Terminal

[DBs](#) are storages for large amounts of structured data. They form the very backends of modern applications. They contain the data that drive business as well as the contents of huge websites. We can connect to DBs via applications and programs in order to work with the data. Often, there are middlewares like web services or application [servers](#) that implement business logic and ensure that data is manipulated or evaluated in a correct way. Such applications and programs, if developed with Python, often run in terminals. You would *never* execute them within a [IDE](#) like PyCharm.

The only proper way to run a Python application in a productive scenario is in the terminal.

— Thomas Weise (汤卫思) [482], 2024

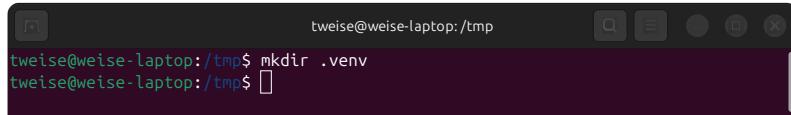
Therefore, we will first explore how to install packages for this use case. We will do so under Ubuntu Linux in the Bash terminal in [Figure 5.1](#). For [Microsoft Windows](#), you can find a similar procedure discussed in [\[482\]](#). The commands there are almost the same.

We begin by open a Bash terminal using [Ctrl](#)+[Alt](#)+[T](#). Then we navigate to the directory where our programming will take place. In my example, I chose the temporary directory [/tmp](#) because I will delete everything once I am done taking screenshots. You would choose a more sensible location. Inside this directory, we now create a new directory [.venv](#) to host a new virtual environment. In the Bash shell, we can do this by typing the command [mkdir .venv](#) and hitting [↵](#) in [Figure 5.1.1](#).

After this empty new directory is created, we can instruct Python to prepare it for use as a virtual environment in [Figure 5.1.2](#). We set up a new and empty virtual environment in this directory by writing [python3 -m venv --upgrade-deps .venv](#) and hitting [↵](#). The new virtual environment has been created. We now activate it by writing [source .venv/bin/activate](#) and hitting [↵](#) in [Figure 5.1.3](#). Once a virtual environment is active, all package installations will go into that environment. Also, if we run a Python program, it will search for installed packages in this environment. And our virtual environment [.venv](#) is now active, which we can see by the changed prompt in [Figure 5.1.4](#). Notice that the environment is only active in the current terminal. If you open another terminal using [Ctrl](#)+[Alt](#)+[T](#), our virtual environment is not active in it (but we can activate it exactly as shown in [Figure 5.1.3](#)).

In Python, we normally install packages by using the program [pip](#). [pip](#) automatically uses the current virtual environment, if one is active. We install psycopg into our new environment by writing [pip install psycopg](#) and hitting [↵](#). This invokes the [pip](#) installer, which looks the package up in [PyPI](#) and downloads it. In [Figure 5.1.5](#), you see that it downloads psycopg in version [3.2.4](#). When you do the same thing, you will probably get a newer version installed.

If we run a Python program that uses psycopg, like [Listing 12.1](#), it will find this version of the package in our virtual environment. Once we have finished programming and running programs, we can deactivate the virtual environment by writing [deactivate](#) and hitting [↵](#). The prompt changes back to normal in [Figure 5.1.6](#). Whenever we need psycopg again, we would activate the virtual environment again as shown in [Figure 5.1.3](#). Of course, we can also install more packages into this environment. And we can also create more environments if we want to, in the same way as discussed here.



```
tweise@weise-laptop:/tmp$ mkdir .venv
tweise@weise-laptop:/tmp$
```

(5.1.1) We first open a terminal using **Ctrl**+**Alt**+**T**. We navigate to the directory where our programming will take place. We now create a directory `.venv` to host a new virtual environment by typing `mkdir .venv` and hitting **↵**.



```
tweise@weise-laptop:/tmp$ mkdir .venv
tweise@weise-laptop:/tmp$ python3 -m venv --upgrade-deps .venv
tweise@weise-laptop:/tmp$
```

(5.1.2) We now set up a new and empty virtual environment in this directory by writing `python3 -m venv --upgrade-deps .venv` and hitting **↵**.



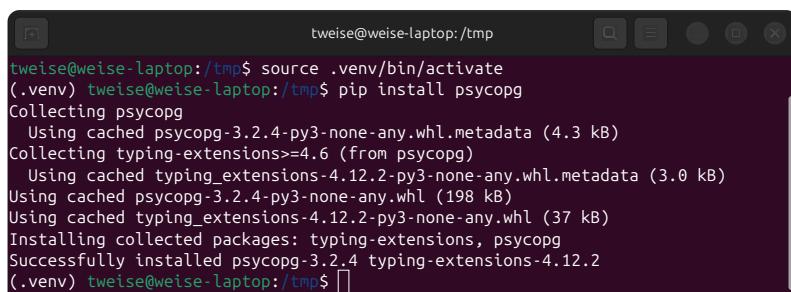
```
tweise@weise-laptop:/tmp$ mkdir .venv
tweise@weise-laptop:/tmp$ python3 -m venv --upgrade-deps .venv
tweise@weise-laptop:/tmp$ source .venv/bin/activate
tweise@weise-laptop:~$
```

(5.1.3) The virtual environment has been created. We now activate it by writing `source .venv/bin/activate` and hitting **↵**.



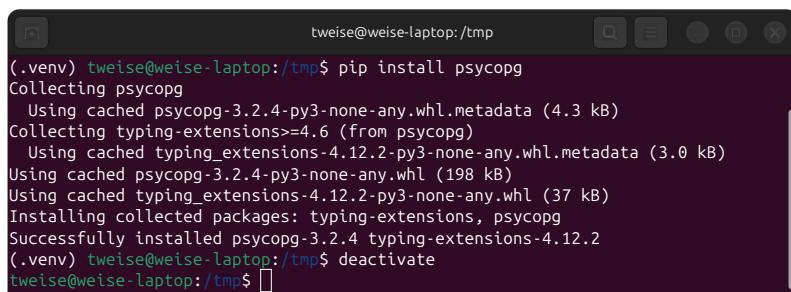
```
tweise@weise-laptop:/tmp$ mkdir .venv
tweise@weise-laptop:/tmp$ python3 -m venv --upgrade-deps .venv
tweise@weise-laptop:/tmp$ source .venv/bin/activate
(.venv) tweise@weise-laptop:/tmp$
```

(5.1.4) The virtual environment `.venv` is now active, which we can see by the changed prompt.



```
tweise@weise-laptop:/tmp$ source .venv/bin/activate
(.venv) tweise@weise-laptop:/tmp$ pip install psycopg
Collecting psycopg
  Using cached psycopg-3.2.4-py3-none-any.whl.metadata (4.3 kB)
Collecting typing-extensions>=4.6 (from psycopg)
  Using cached typing_extensions-4.12.2-py3-none-any.whl.metadata (3.0 kB)
Using cached psycopg-3.2.4-py3-none-any.whl (198 kB)
Using cached typing_extensions-4.12.2-py3-none-any.whl (37 kB)
Installing collected packages: typing-extensions, psycopg
Successfully installed psycopg-3.2.4 typing-extensions-4.12.2
(.venv) tweise@weise-laptop:/tmp$
```

(5.1.5) We install `psycopg` into this environment by writing `pip install psycopg` and hitting **↵**.



```
(.venv) tweise@weise-laptop:/tmp$ pip install psycopg
Collecting psycopg
  Using cached psycopg-3.2.4-py3-none-any.whl.metadata (4.3 kB)
Collecting typing-extensions>=4.6 (from psycopg)
  Using cached typing_extensions-4.12.2-py3-none-any.whl.metadata (3.0 kB)
Using cached psycopg-3.2.4-py3-none-any.whl (198 kB)
Using cached typing_extensions-4.12.2-py3-none-any.whl (37 kB)
Installing collected packages: typing-extensions, psycopg
Successfully installed psycopg-3.2.4 typing-extensions-4.12.2
(.venv) tweise@weise-laptop:/tmp$ deactivate
tweise@weise-laptop:/tmp$
```

(5.1.6) Once we have finished programming, we can deactivate the virtual environment by writing `deactivate` and hitting **↵**. The prompt changes back to normal.

Figure 5.1: Installing the Python package `psycopg` into a virtual environment under Ubuntu Linux using `pip` in a Bash shell terminal.

5.1.2 Cloning Repository and Installing Psycopg under PyCharm

The PyCharm IDE allows us to download all the example codes of this course and to install the Python packages they depend on in one step from GitHub. In this process, we encounter two concepts: Git and the aforementioned virtual environments. Git is a distributed VCS, meaning that it allows us to create so-called repositories, which basically are directories, whose change history is preserved and that can be collaboratively edited by teams. Every team member has a local copy of the code and can commit their local changes to the central repository as well as download the changes committed by others. GitHub is one provider that offers us to host Git repositories online. All the example files used in this book are stored in such a Git repository on GitHub: <https://github.com/thomasWeise/databasesCode>.

The process of downloading a Git repository with all its history is called *cloning*. PyCharm is an IDE for Python software development that supports Git. You can use it to clone Git repositories and it also supports using local virtual environments as well as installing packages into them based on the requirements that a repository defines. Since we want to access PostgreSQL DBs from Python using psycopg, our example repository does specify this package as a dependency.

Thus, it is possible to clone our example repository and install psycopg in one step. This can be done as follows.

First, we need to open PyCharm and then clone the repository. This can be done by clicking the `Clone Repository` button in the PyCharm welcome screen, as illustrated in Figure 5.2.1. If you don't have that welcome screen open, you can also click on the `File` menu and then on `Project from Version Control...`, as shown in Figure 5.2.2. Then, a new form pops up in Figure 5.2.3. We have to enter the Git repository URL and a location in our local file system to which the repository should be downloaded and where the new PyCharm project with its contents should be created.

As said, the URL of our repository is <https://github.com/thomasWeise/databasesCode>. As location, I chose some place in the temporary folder in Figure 5.2.4, because I am just making screenshots for this book and will delete the project afterwards. You will choose some other, more reasonable and permanent place. We click on `Clone` and the cloning process starts in Figure 5.2.5. We may get asked whether we trust this new project. After verifying that you downloaded the correct repository, you can click `Trust Project`, as shown in Figure 5.2.6.

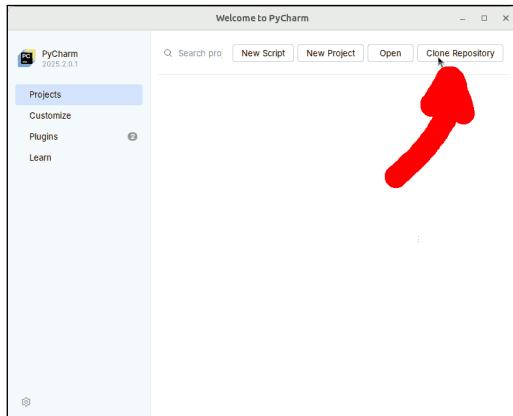
Eventually the cloning process is finished. The repository has been downloaded. Maybe PyCharm detects that this project could use Python and asks you to configure a Python interpreter as shown in Figure 5.3.1. In my case, this request disappeared too quickly, so in Figure 5.3.2, I instead clicked on the `File` menu and then on `Settings...`, or press `Ctrl + Alt + S`. Either way, we would go to the `Python` settings and choose `Interpreter`. There, we click `Add Interpreter` in Figure 5.3.3 and then `Add Local Interpreter...` (Figure 5.3.4).

In the dialog that pops open, we choose `Generate new`. As type we choose `Virtualenv`. The virtual environment is basically a directory which contains a local Python installation. This allows us to have separate Python setups for different projects. Therefore, if our course requires you to work with a version of psycopg but your other projects do not need that, then you would only install it into the local virtual environment for the examples of our course. If psycopg, in turn, requires other libraries, then these will live in this separate virtual environment, too. Therefore, nasty things like version conflicts with the setups of other projects cannot occur. We add `.venv` as location for this virtual environment to the root folder of the project. Then we click `OK` in Figure 5.3.5. We click `OK` again in Figure 5.3.6. As you can see in Figure 5.4.1, a directory named `.venv` has indeed been created. It will host all the files and packages of the environment, which resemble a local Python installation.

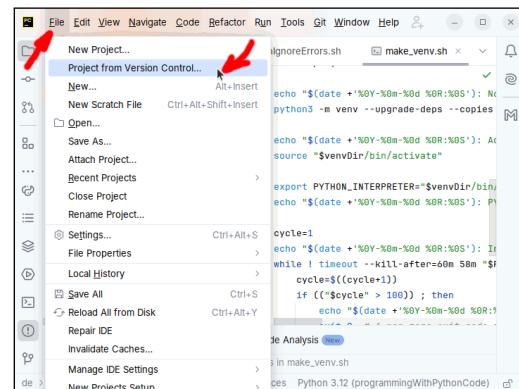
The packages that a Python project requires are usually listed in a file called `requirements.txt`. Such a file is also present in the root folder of our example repository. We click on it in PyCharm and find that it indeed lists all the packages required: We need `psycopg` at version `3.2.4` as shown in Figure 5.4.2. Of course, this is only at the time of this writing and will change in the future.

Anyway, when we explore this file, we can see some yellow hints and a yellow light bulb symbol. We click on it and it opens a dialog asking us whether we want to install psycopg. We then click on that in Figure 5.4.3. PyCharm asks us to confirm the package installation, which we do, by clicking on `Yes` in Figure 5.4.4

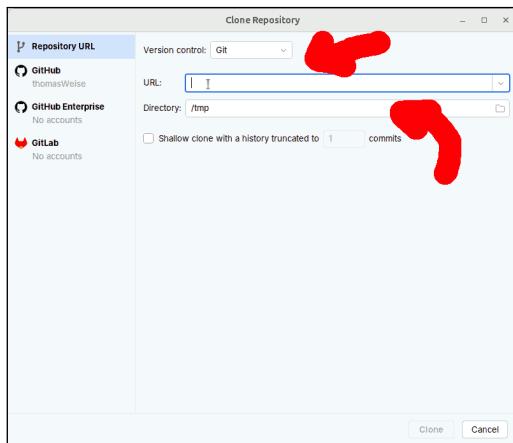
PyCharm now downloads the package for us in Figure 5.4.5 and then installs it into the virtual environment. Maybe PyCharm informs us that a newer version is available, as shown in Figure 5.4.6. Anyway, we can see that the package has been included in the `.venv` folder, as illustrated in Figure 5.5.1. It is now available for our experiments.



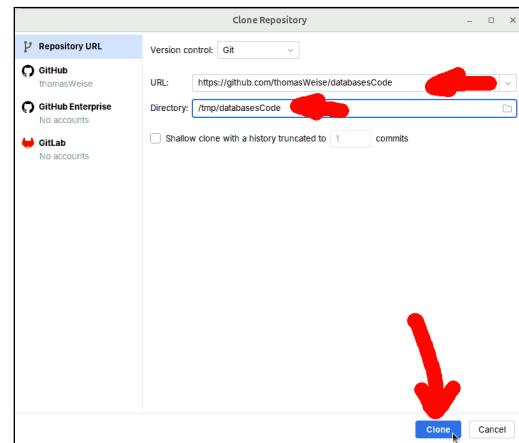
(5.2.1) To clone = download the Git repository with the examples for this class in the welcome screen of PyCharm, you need to click **Clone Repository**.



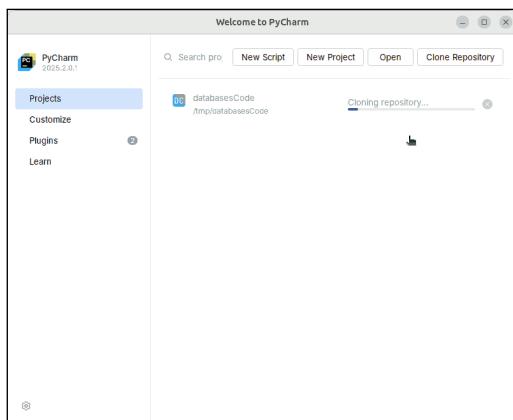
(5.2.2) If you are not in the PyCharm welcome screen, you can instead click on the **File** menu and then on **Project from Version Control...**.



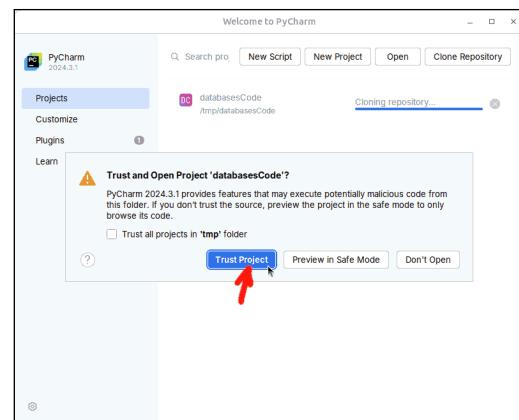
(5.2.3) A new form pops up. We have to enter the Git repository Uniform Resource Locator (URL) and a location in our local file system to which the repository should be downloaded and where the new PyCharm project with its contents should be created.



(5.2.4) The URL of our repository is <https://github.com/thomasWeise/databasesCode>. As location, I chose some place in the temporary folder – you will choose some other place. We click on **Clone**.

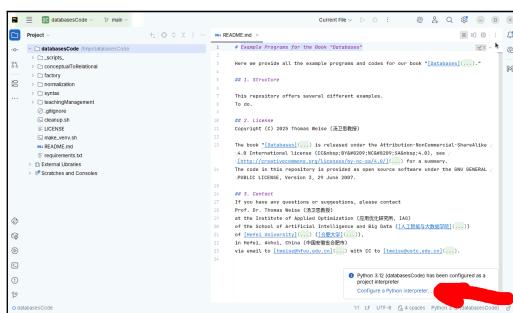


(5.2.5) The cloning process starts: The repository is downloaded.

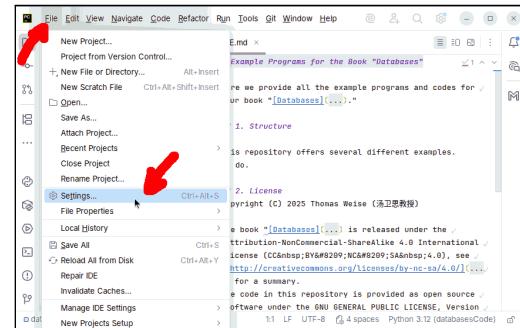


(5.2.6) We may get asked whether we trust this new project. After verifying that you downloaded the correct repository, you can click **Trust Project**.

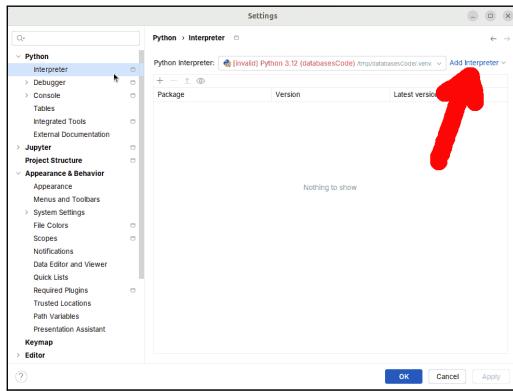
Figure 5.2: Cloning the repository with the examples in PyCharm and setting up a virtual environment inside PyCharm into which we install psycopg.



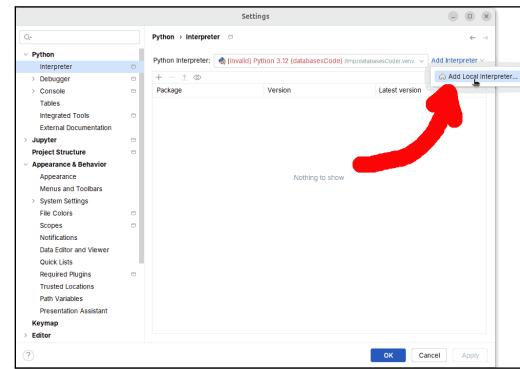
(5.3.1) The cloning process is finished. The repository has been downloaded. Maybe PyCharm detects that this project could use Python and asks you to configure a Python interpreter. If you miss this request, see Figure 5.3.2.



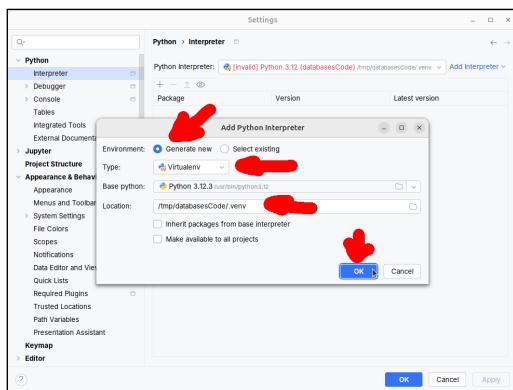
(5.3.2) We now click on the **File** menu and then on **Settings...**, or press **Ctrl+Alt+S**.



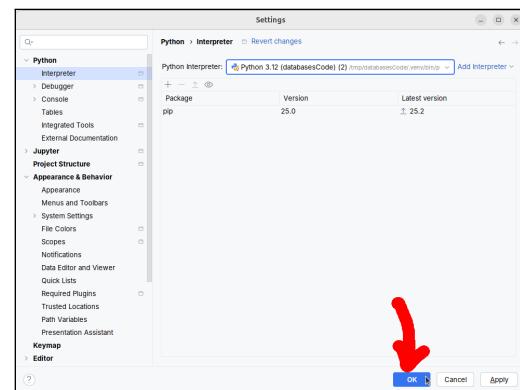
(5.3.3) We then go to the **Python** settings and choose **Interpreter**. There, we click **Add Interpreter**.



(5.3.4) ... and then **Add Local Interpreter...**.

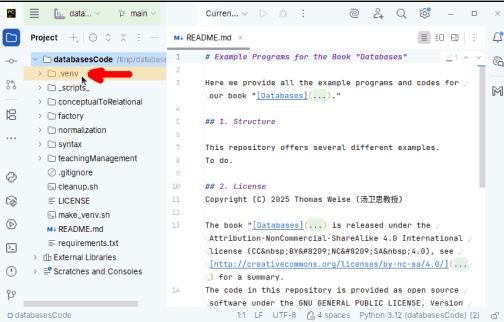


(5.3.5) In the dialog that pops open, choose **Generate new**, as type we choose **Virtualenv**, and then we simply add **.venv** as location to the root folder of the project. Then we click **OK**.

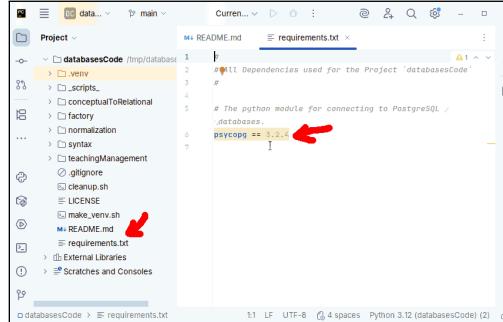


(5.3.6) We click **OK** again.

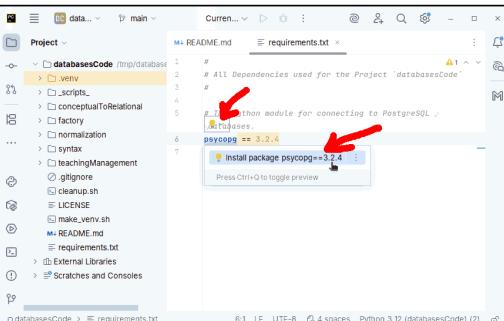
Figure 5.3: Cloning the repository with the examples in PyCharm and setting up a virtual environment inside PyCharm into which we install psycopg (Continued).



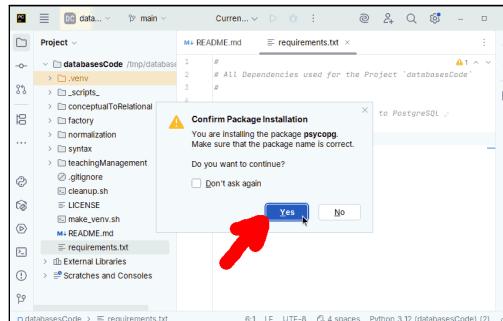
(5.4.1) As you can see, a directory named `.venv` has indeed been created. It will host all the files and packages of the environment, which resemble a local Python installation.



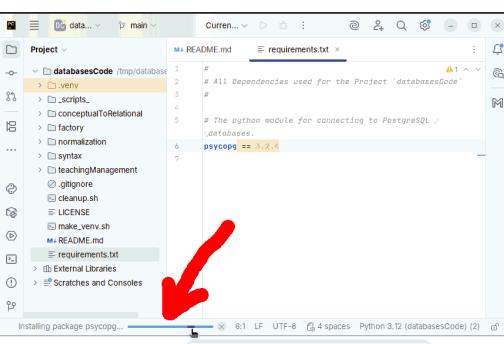
(5.4.2) We now click on the file `requirements.txt` in the root folder of the project. It lists all the packages required. In this case, we need `psycopg` at version `3.2.4`, which will change in the future.



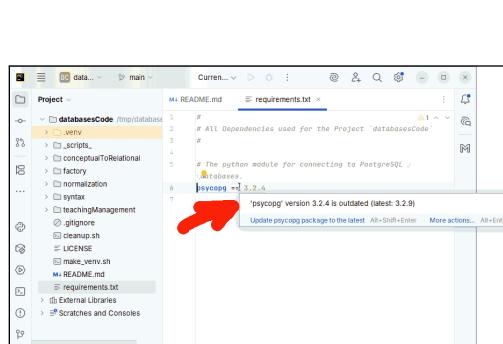
(5.4.3) We right-click on the yellow light bulb symbol, which opens a dialog asking us whether we want to install `psycopg`. We click on that.



(5.4.4) We get asked to confirm the package installation. We click on `Yes`.

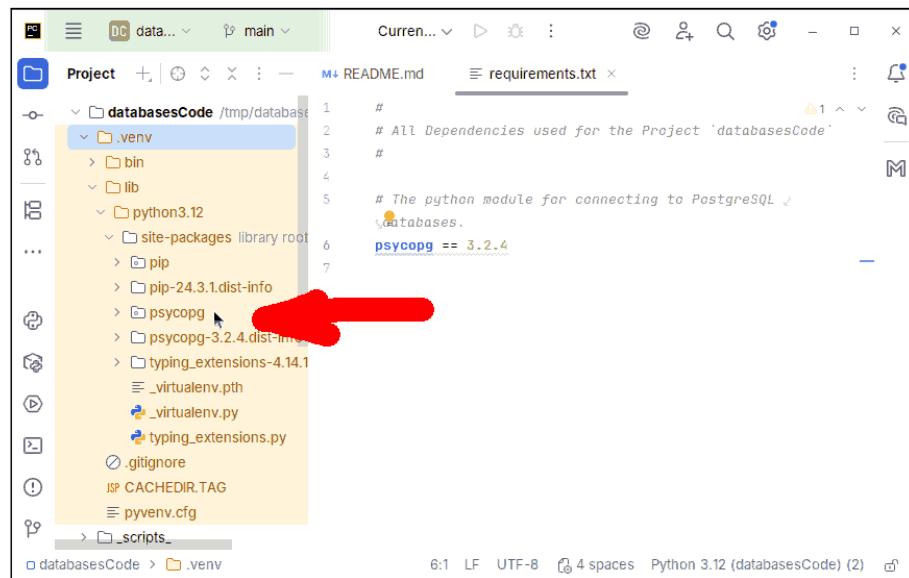


(5.4.5) The package is downloaded.



(5.4.6) The package is installed. Maybe PyCharm informs us that a newer version is available.

Figure 5.4: Cloning the repository with the examples in PyCharm and setting up a virtual environment inside PyCharm into which we install `psycopg` (Continued).



The screenshot shows the PyCharm interface with a project named "databasesCode". The left sidebar displays a file tree with a .venv folder expanded, showing subfolders like bin, lib, and python3.12. Inside lib/site-packages, there is a folder named "psycopg". A red arrow points to this folder. The right panel shows a code editor with a file named "requirements.txt" containing the following text:

```
1  #
2  # All Dependencies used for the Project 'databasesCode'
3  #
4
5  # The python module for connecting to PostgreSQL
6  psycopg == 3.2.4
7
```

(5.5.1) Indeed, we can now even find it in the `.venv` folder.

Figure 5.5: Cloning the repository with the examples in PyCharm and setting up a virtual environment inside PyCharm into which we install psycopg (Continued).

Chapter 6

Installing yEd

yEd is a free and platform-independent editor for graph data [384, 497]. It is useful when we learn about ERDs when modelling the conceptual schema of a DB, which we will discuss in Chapter 18 (and where we will also use yEd). While it is not open source software (OSS), yEd is free and works on each OS for which Java is available.

6.1 Installing yEd on Ubuntu Linux

In order to draw technology-independent ERDs, we want to install yEd on our Ubuntu Linux machine. yEd is written in Java. We will download the Java `.jar` archive with the yEd application together with the dependencies. This archive can then be executed on any system that has Java installed and it does not require any installation. We therefore need a Java installation on our machine and we then we can download the yEd application.

For installing Java, there exist many tutorials, so we will not cover this in detail. Installing Java under Ubuntu is fairly easy. You would open a console `terminal` by pressing `Ctrl+Alt+T`. First, you type in `java --version` to see if you already have Java installed. If this command is found and produces some output, then you are done here and can directly continue with downloading yEd.

If not, then you would type `sudo apt-get install openjdk-xx-jre`, where `xx` is to be replaced with the version that you wish to install. On my Ubuntu version, I can choose between 8, 11, 17, and 21. To see which options are available on your system, type `sudo apt-get install openjdk-` into your terminal and then press `Enter`. This shows you a list of available installation options. You always will want to use one that ends with `-jre`, i.e., a runtime environment (you do not need a `-jdk`, i.e., developer kit, unless you want to write programs in Java, too – which is fairly cool, so maybe you want to try this as well...). I suggest to always going with the newest `-jre` version, so 21 it is in my case. You install the newest Java `jre` version on your machine. Either way, I will assume that you have Java installed.

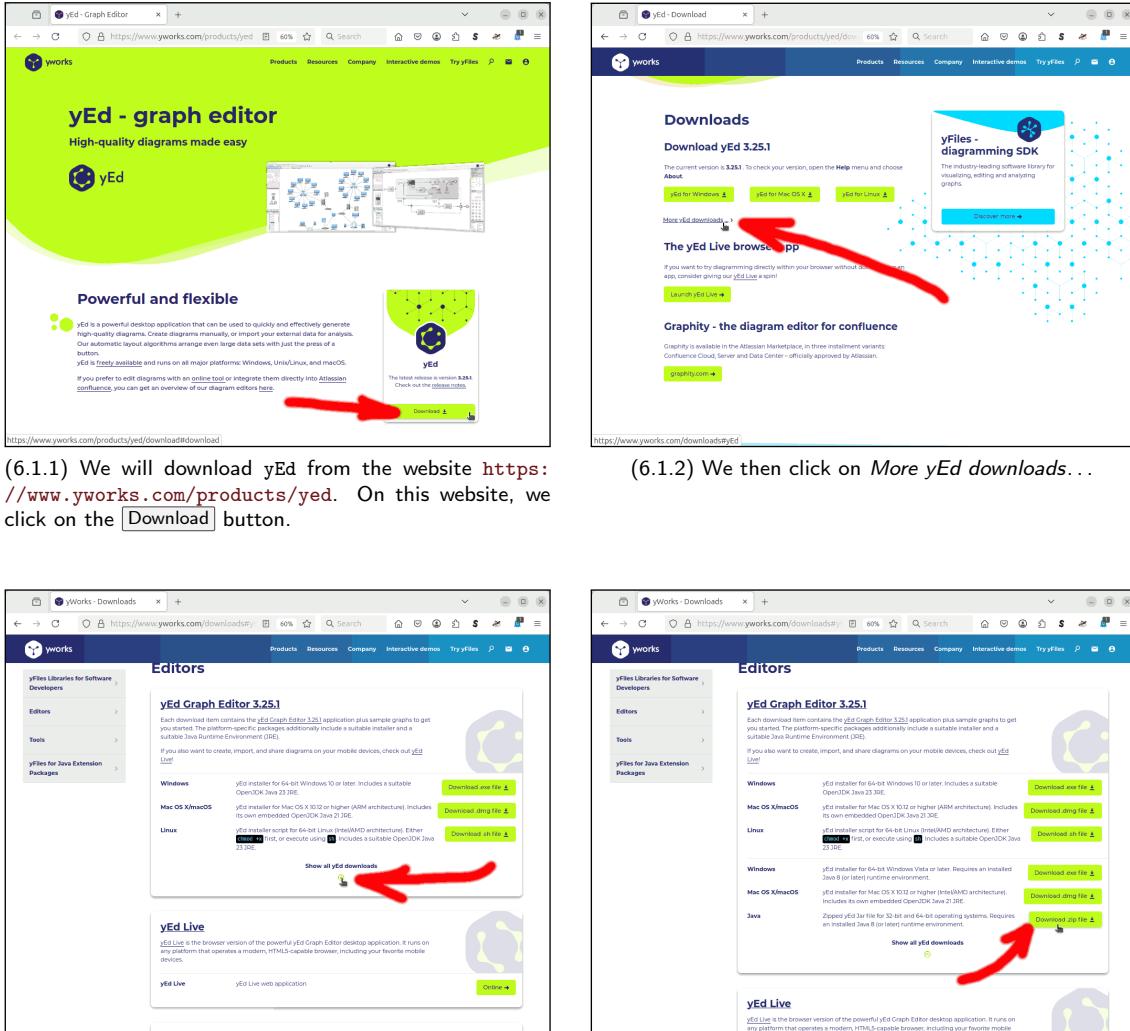
So now all we have to do is to download yEd. In Figure 6.1.1, we use our browser to access the website <https://www.yworks.com/products/yed>. On this website, we click on the `Download` button. In Figure 6.1.2, we click on `More yEd downloads`. And in Figure 6.1.3, we click on `Show all yEd downloads`.

What we want is the `.jar` archive with the yEd application. Therefore, we choose to download the `Zipped yEd .jar file...` which is suitable for all systems that have Java installed. We click `Download .zip file` in Figure 6.1.4.

Once we clicked the download button, we get taken to the license screen. We carefully read the license and if we are OK with it, select `I accept the license terms` in Figure 6.1.5. After we OKed the license, we can click on `Download` in Figure 6.1.6. The download starts in Figure 6.1.7.

Eventually, the download completed. We can click to open the folder where the file was stored in Figure 6.1.8. We find a `.zip` archive in that folder. You need to unpack this file into a proper installation location. Notice that the `.zip` archive may contain a folder inside, and inside this folder, you will find the actual application. The actual applications is the file `yed.jar`, and it is bundled with several other files that you need. So unpack the `.zip` archive and copy the files into the place where we want them in Figure 6.1.9.

Since I just make these screenshots for demonstration purposes and have yEd already installed

(6.1.3) ...and then on *Show all yEd downloads*.

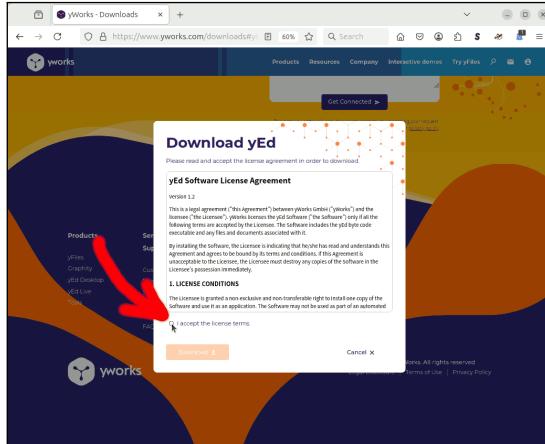
(6.1.4) We want to download the *Zipped yEd jar file*, which is suitable for all systems that have Java installed. (If Java is not installed on your machine, install it via `sudo apt-get install openjdk-xx-jre`, where `xx` can be replaced with the version, say `8` or `21`.) We click `Download zip file`.

Figure 6.1: Installing yEd under Ubuntu Linux.

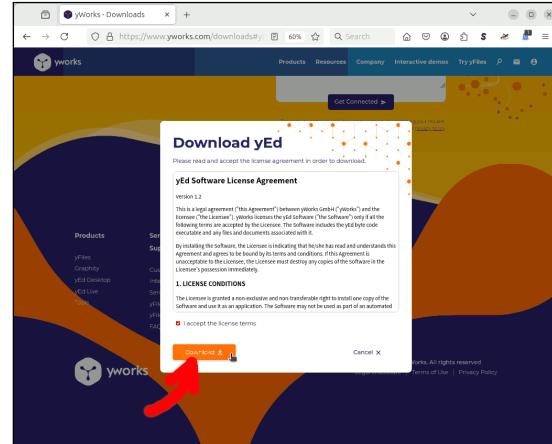
elsewhere, I here choose to place the files into `/tmp/yed`. This is *not* the correct location for your installation. You will want to choose something more permanent.

Anyway, we open a console by pressing `Ctrl+Alt+T`. We go to the folder which contains the unpacked data and the file `yed.jar`. In my case, this is `/tmp/yed` in Figure 6.1.10. We start the program by typing `java -jar yed.jar` in Figure 6.1.11.

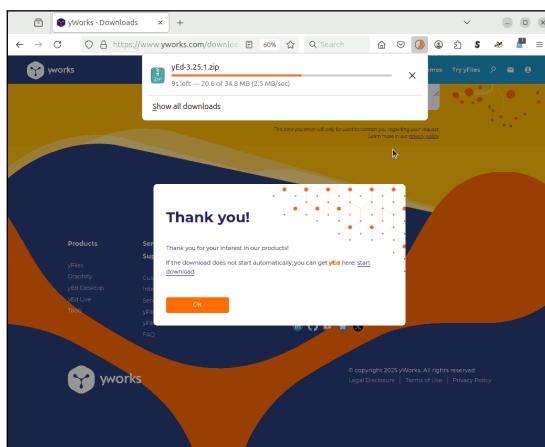
A splash screen appears when the program loads in Figure 6.1.12. Finally, Figure 6.1.13, the program has started.



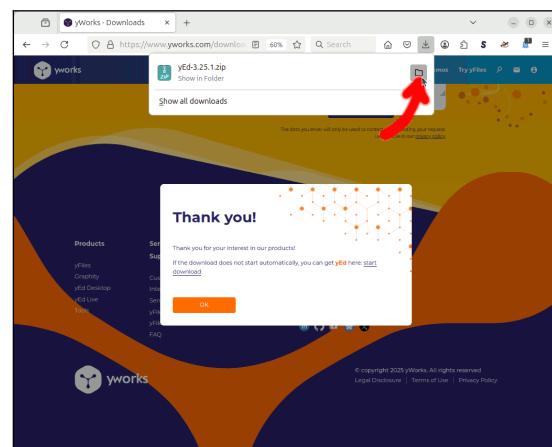
(6.1.5) Once we clicked the download button, we get taken to the license screen. We carefully read the license and if we are OK with it, select **I accept the license terms**.



(6.1.6) After we OKed the license, we can click on **Download**.

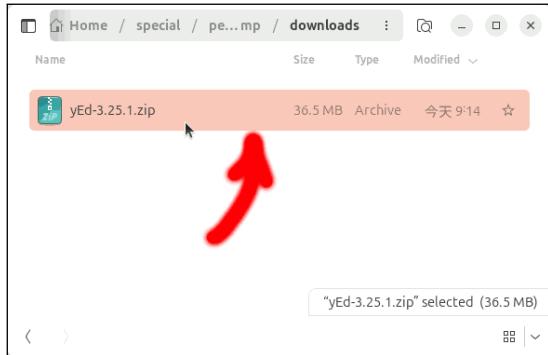


(6.1.7) The download starts.

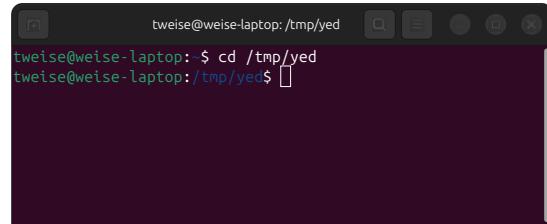


(6.1.8) Eventually, the download completed. We can click to open the folder where the file was stored.

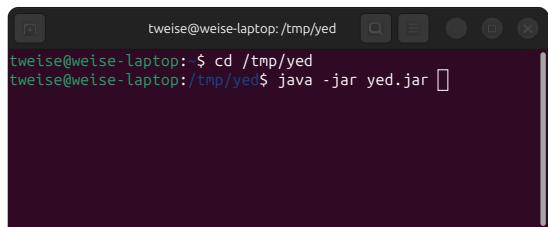
Figure 6.1: Installing yEd under Ubuntu Linux (Continued).



(6.1.9) We find a `zip` file in that folder. We unpack it to the installation location, i.e., to the place where we want to store yEd.



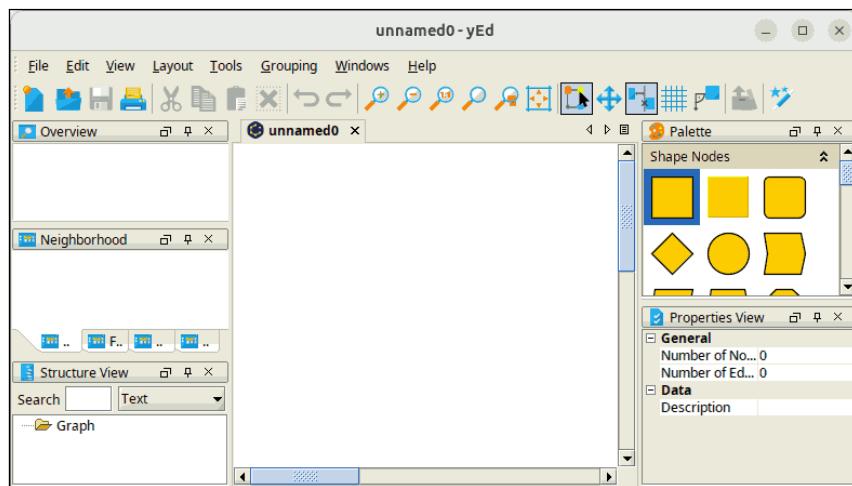
(6.1.10) We open a console by pressing `Ctrl`+`Alt`+`T`. We go to the folder which contains the unpacked data and the file `yed.jar`. In my case, this is `/tmp/yed`.



(6.1.11) We start the program by typing `java -jar yed.jar`.



(6.1.12) A splash screen appears when the program loads.



(6.1.13) The program has started.

Figure 6.1: Installing yEd under Ubuntu Linux (Continued).

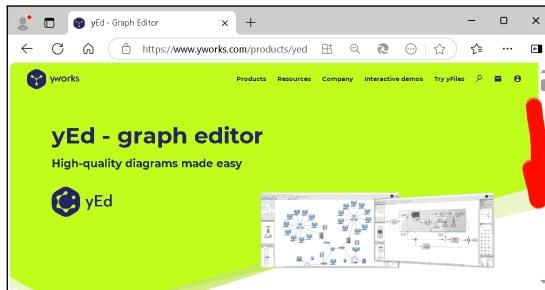
6.2 Installing yEd on Microsoft Windows

Installing yEd under Microsoft Windows is very easy. We basically just have to download and run the installer. We will download yEd from the website <https://www.yworks.com/products/yed>. We surf to this website in [Figure 6.2.1](#) and then need to scroll down. We locate the [Download](#) button on the website and click on it in [Figure 6.2.2](#).

On the next page, we get to choose for which [OS](#) we want to install yEd. We click on [yEd for Windows](#) in [Figure 6.2.3](#). In order to download the program, we have to accept the license agreement, as shown in [Figure 6.2.4](#). We carefully read the agreement. If we agree to it, we can tick the corresponding box and then we can click on [Download](#) in [Figure 6.2.5](#).

Once the download completes, we click [Open file](#) or the equivalent option in your browser in [Figure 6.2.6](#). The installation wizard begins the setup in [Figure 6.2.7](#). Shortly thereafter, Microsoft Windows asks us whether we want to permit the program to make changes on your computer, i.e., to install yEd. We click [Yes](#) in [Figure 6.2.8](#).

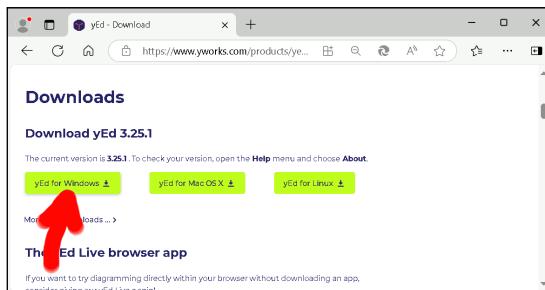
The installer's welcome screen appears. We click [Next](#) in [Figure 6.2.9](#). In order to install yEd, we must agree to the license agreement. If you are OK with it, accept the agreement, tick the corresponding



(6.2.1) We will download yEd from the website <https://www.yworks.com/products/yed>. We surf to this website and scroll down.



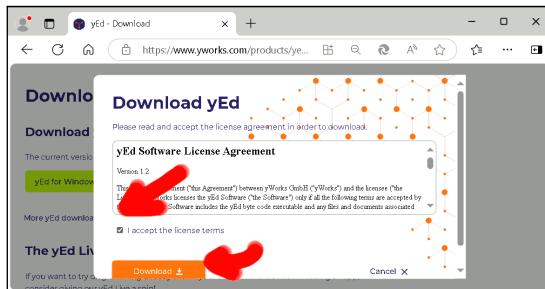
(6.2.2) We locate the [Download](#) button and click on it.



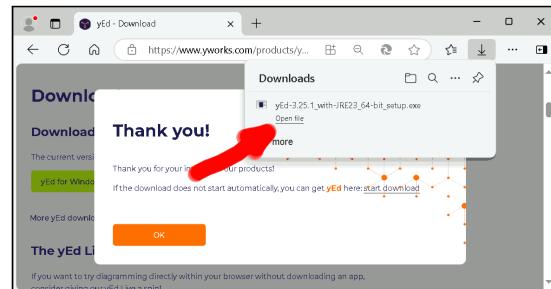
(6.2.3) We click on [yEd for Windows](#).



(6.2.4) In order to download the program, we have to first carefully check the license agreement. If we find it acceptable, we can accept it.

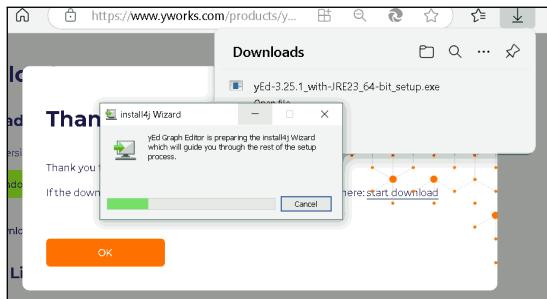


(6.2.5) Then we can click on [Download](#).

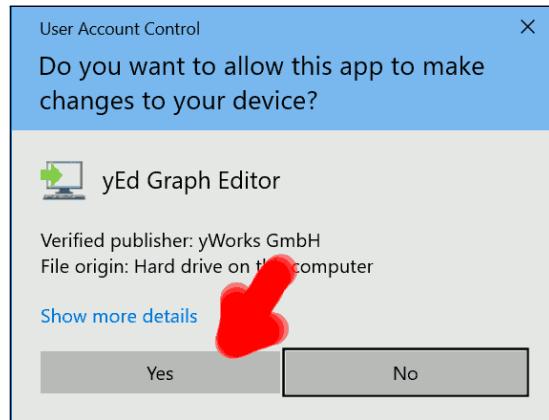


(6.2.6) Once the download completes, we click [Open file](#) or the equivalent option in your browser.

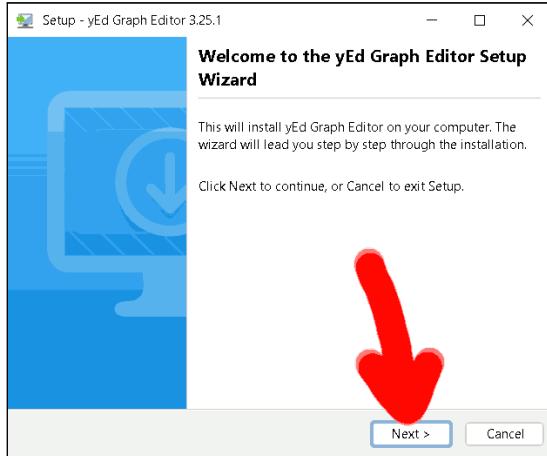
Figure 6.2: Installing yEd under Microsoft Windows.



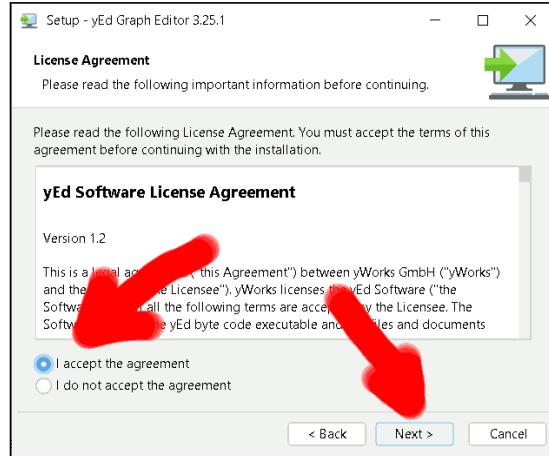
(6.2.7) The installation wizard begins the setup.



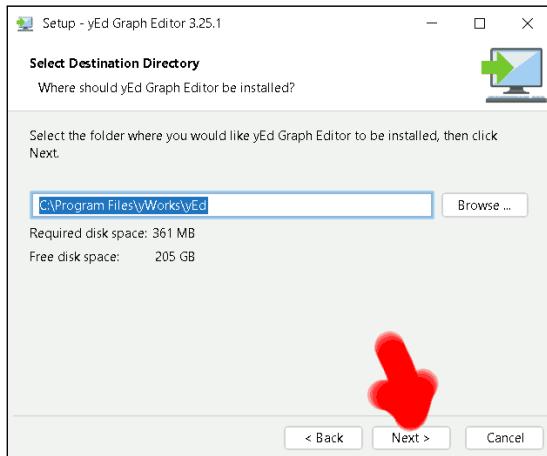
(6.2.8) Microsoft Windows asks us whether we want to permit the program to make changes on your computer (= install something). We click [Yes].



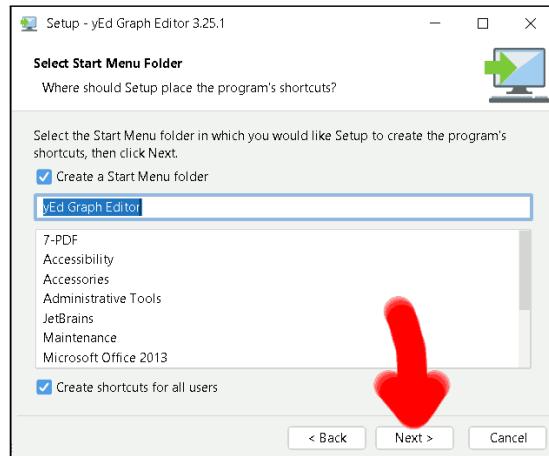
(6.2.9) We click [Next] in the installer's welcome screen.



(6.2.10) In order to install yEd, we must agree to the license agreement. If you are OK with it, accept the agreement and click [Next].

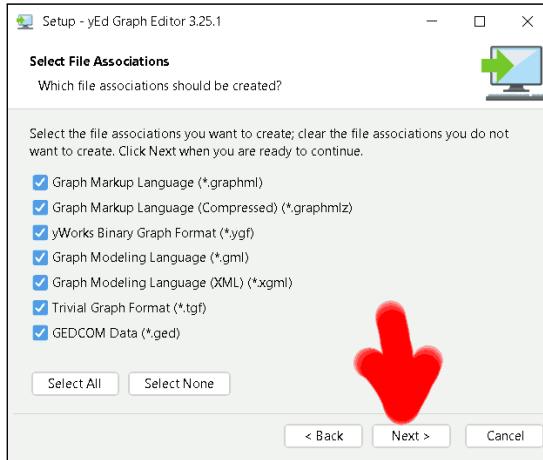


(6.2.11) We do not change the installation destination and click [Next].

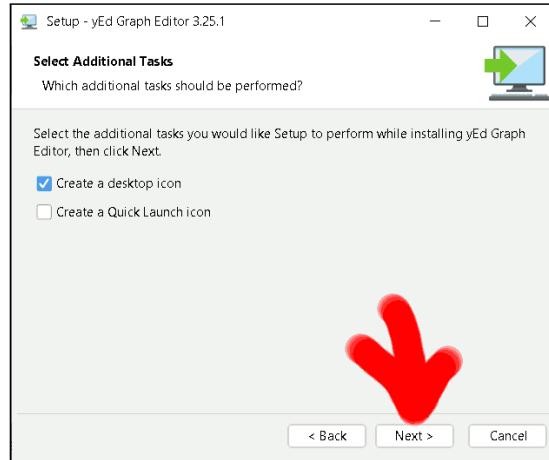


(6.2.12) We do not change the start menu settings and click [Next].

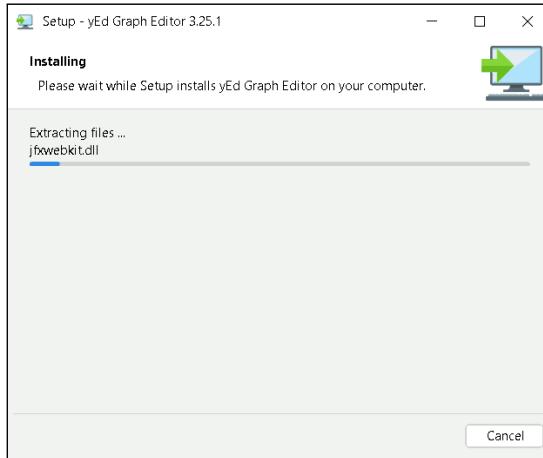
Figure 6.2: Installing yEd under Microsoft Windows (continued).



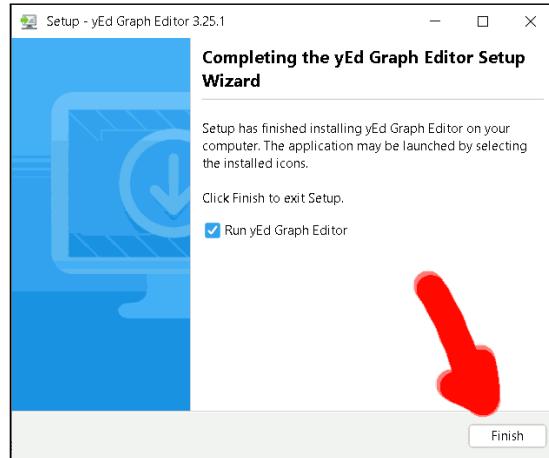
(6.2.13) We do not change the file type associations and click **Next**.



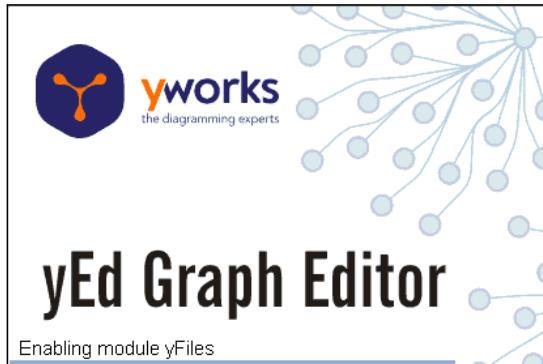
(6.2.14) We do not change the additional task settings and click **Next**.



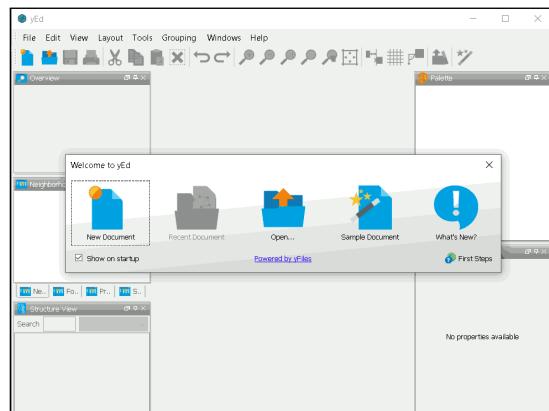
(6.2.15) The installation begins.



(6.2.16) The installation completes. We mark "Run yEd Graph Editor" and click on **Finish**.



(6.2.17) The yEd splash screen appears.



(6.2.18) yEd is installed and running.

Figure 6.2: Installing yEd under Microsoft Windows (continued).

box, and click **Next**, as shown in [Figure 6.2.10](#).

Then we can choose where to install **yEd**. We do not change the installation destination and click **Next** in [Figure 6.2.11](#). The option whether and where we want to add buttons to the start menu if **Microsoft Windows** appears. We do not change the start menu settings and click **Next** in [Figure 6.2.12](#). The installer asks which files should be associated with **yEd**. We do not change the file type associations and click **Next** in [Figure 6.2.13](#). In the screen for additional tasks, we could decide not to add a desktop icon, for instance. However, in [Figure 6.2.14](#), we do not change the additional task settings and click **Next**. The installation begins in [Figure 6.2.15](#).

Soon thereafter, the installation completes. In the final screen shown in [Figure 6.2.16](#), we make sure that “Run **yEd Graph Editor**” is marked and click on **Finish**.

Now, the **yEd** splash screen appears in [Figure 6.2.17](#). **yEd** is installed and running and we can begin to use it, as illustrated in [Figure 6.2.18](#).

Chapter 7

Installing PgModeler

The PgModeler [8] is a free open source tool with which we can create logical DB models for the DBMS PostgreSQL. PgModeler allows us to draw logical models in form of ERDs while annotating attributes with SQL-based types and constraints. It stores all files in an XML format and allows us to export our logical models to SQL, which we could then feed to the PostgreSQL DBMS via, e.g., the psql client. We will use the PgModeler in Section 19.2.1 of this book, for instance.

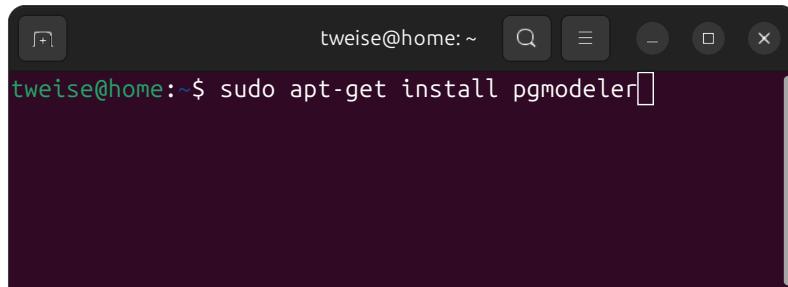
7.1 Installing PgModeler under Ubuntu Linux

Under Ubuntu Linux, we can install the PgModeler via `apt-get`. We therefore first open a Bash terminal via `Ctrl+Alt+T`. In Figure 7.1.1, we then type in `sudo apt-get install pgmodeler` and hit `Enter`. Installing software this way requires the sudo permission. In Figure 7.1.2, we thus get asked for the sudo password. We type it in, and hit `Enter`.

The system now tells us the packages that need to be installed and asks us whether we are OK with that. We are, because we need the PgModeler. So we type `Y` and hit `Enter` in Figure 7.1.3. In Figure 7.1.4, the PgModeler gets downloaded and installed.

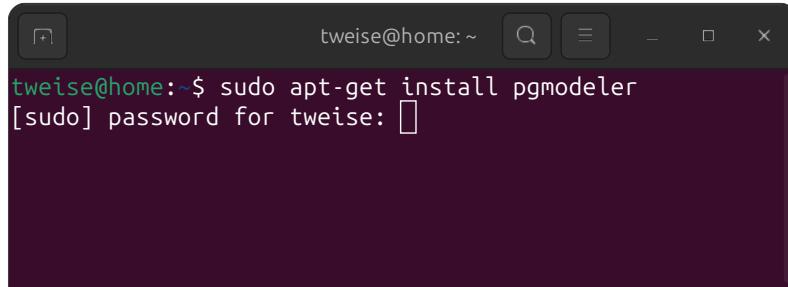
After the installation is completed, we can run the PgModeler by typing `pgmodeler` into the Bash terminal and hitting `Enter` in Figure 7.1.5. The PgModeler will start up. Usually only at the first time you start it, it may or may not display an error notification window as shown in Figure 7.1.6. If this notification pops up, we can simply ignore it and click on `OK`.

The PgModeler window opens in Figure 7.1.7. For this book, I will use the PgModeler in light mode. If you also want to use the light mode, you would click on `☰ > Edit > Settings`, as shown in Figure 7.1.8. In the `Appearance` menu, we change the theme the `Light` in Figure 7.1.9. We then click on `Apply` in Figure 7.1.10. Then, in the `Relationships` register, we make sure that the connection mode is set to “Crow’s foot notation,” as shown in Figure 7.1.11. The PgModeler is now installed and ready to use.



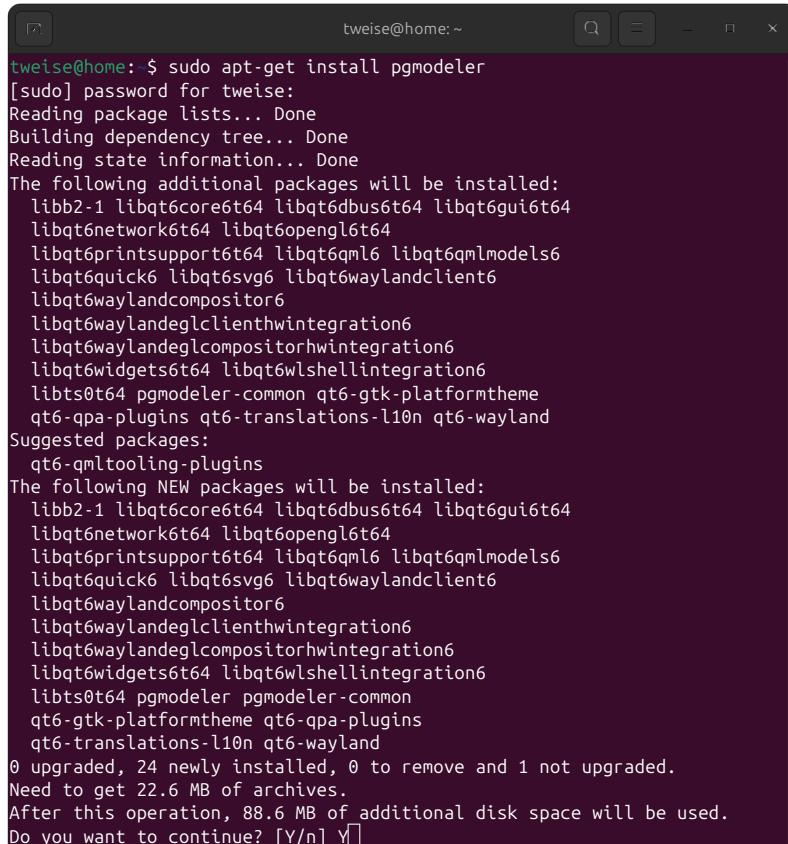
tweise@home:~\$ sudo apt-get install pgmodeler

(7.1.1) Open a Bash terminal via **Ctrl** + **Alt** + **T**. Type in `sudo apt-get install pgmodeler` and hit **Enter**.



tweise@home:~\$ sudo apt-get install pgmodeler
[sudo] password for tweise:

(7.1.2) We get asked for the sudo password, type it in, and hit **Enter**.



```
tweise@home: $ sudo apt-get install pgmodeler
[sudo] password for tweise:
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
The following additional packages will be installed:
  libbb2-1 libqt6core6t64 libqt6dbus6t64 libqt6gui6t64
  libqt6network6t64 libqt6opengl6t64
  libqt6printsupport6t64 libqt6qml6 libqt6qmlmodels6
  libqt6quick6 libqt6svg6 libqt6waylandclient6
  libqt6waylandcompositor6
  libqt6waylandeglclienthwintegration6
  libqt6waylandeglcompositorhwintegration6
  libqt6widgets6t64 libqt6wlshellintegration6
  libts0t64 pgmodeler-common qt6-gtk-platformtheme
  qt6-qpa-plugins qt6-translations-l10n qt6-wayland
Suggested packages:
  qt6-qmltooling-plugins
The following NEW packages will be installed:
  libbb2-1 libqt6core6t64 libqt6dbus6t64 libqt6gui6t64
  libqt6network6t64 libqt6opengl6t64
  libqt6printsupport6t64 libqt6qml6 libqt6qmlmodels6
  libqt6quick6 libqt6svg6 libqt6waylandclient6
  libqt6waylandcompositor6
  libqt6waylandeglclienthwintegration6
  libqt6waylandeglcompositorhwintegration6
  libqt6widgets6t64 libqt6wlshellintegration6
  libts0t64 pgmodeler pgmodeler-common
  qt6-gtk-platformtheme qt6-qpa-plugins
  qt6-translations-l10n qt6-wayland
0 upgraded, 24 newly installed, 0 to remove and 1 not upgraded.
Need to get 22.6 MB of archives.
After this operation, 88.6 MB of additional disk space will be used.
Do you want to continue? [Y/n] Y
```

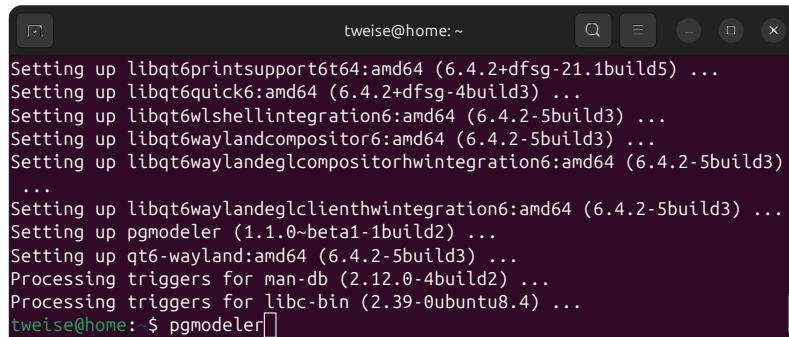
(7.1.3) The system tells us the packages that need to be installed and asks us whether we are OK with that. We type **Y** and hit **Enter**.

Figure 7.1: The installation steps for PgModeler under Ubuntu Linux.



```
tweise@home: ~
Setting up libqt6printsupport6t64:amd64 (6.4.2+dfsg-21.1build5) ...
Setting up libqt6quick6:amd64 (6.4.2+dfsg-4build3) ...
Setting up libqt6wlshellintegration6:amd64 (6.4.2-5build3) ...
Setting up libqt6waylandcompositor6:amd64 (6.4.2-5build3) ...
Setting up libqt6waylandeglcompositorhwintegration6:amd64 (6.4.2-5build3)
...
Setting up libqt6waylandeglclienthwintegration6:amd64 (6.4.2-5build3) ...
Setting up pgmodeler (1.1.0~beta1-1build2) ...
Setting up qt6-wayland:amd64 (6.4.2-5build3) ...
Processing triggers for man-db (2.12.0-4build2) ...
Processing triggers for libc-bin (2.39-0ubuntu8.4) ...
tweise@home: $
```

(7.1.4) The PgModeler gets installed.



```
tweise@home: ~
Setting up libqt6printsupport6t64:amd64 (6.4.2+dfsg-21.1build5) ...
Setting up libqt6quick6:amd64 (6.4.2+dfsg-4build3) ...
Setting up libqt6wlshellintegration6:amd64 (6.4.2-5build3) ...
Setting up libqt6waylandcompositor6:amd64 (6.4.2-5build3) ...
Setting up libqt6waylandeglcompositorhwintegration6:amd64 (6.4.2-5build3)
...
Setting up libqt6waylandeglclienthwintegration6:amd64 (6.4.2-5build3) ...
Setting up pgmodeler (1.1.0~beta1-1build2) ...
Setting up qt6-wayland:amd64 (6.4.2-5build3) ...
Processing triggers for man-db (2.12.0-4build2) ...
Processing triggers for libc-bin (2.39-0ubuntu8.4) ...
tweise@home: $ pgmodeler
```

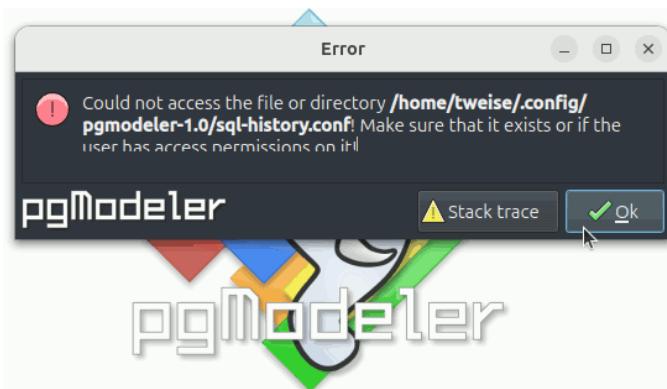
(7.1.5) We now run the PgModeler by typing `pgmodeler` into the Bash terminal and hitting `Enter`.(7.1.6) The program may or may not display an error notification window. If this notification pops up, we can simply ignore it and click on `OK`.

Figure 7.1: The installation steps for PgModeler under Ubuntu Linux (continued).



(7.1.7) The PgModeler window opens.

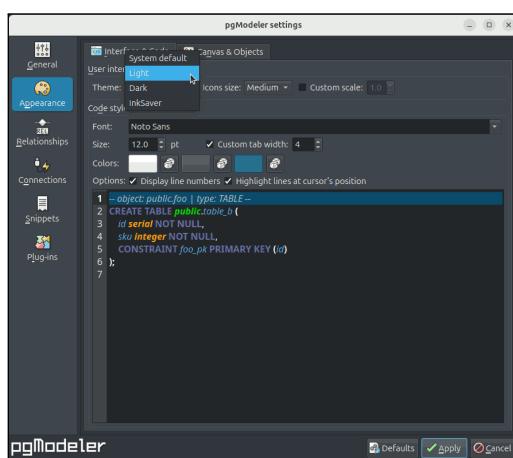
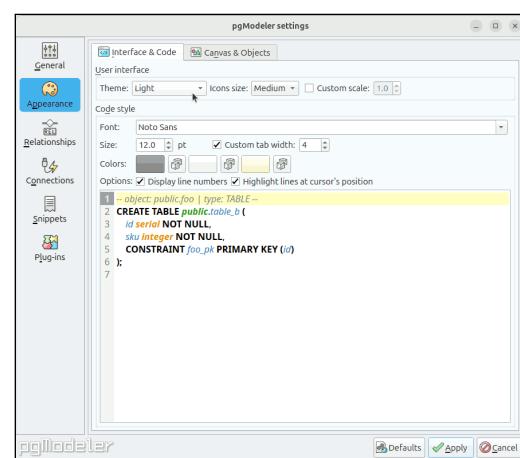
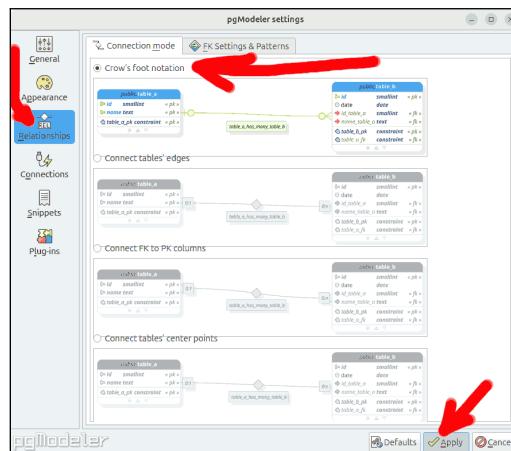
(7.1.8) For this book, I will use the PgModeler in light mode, so we click on **Edit** **Settings**.(7.1.9) In the **Appearance** menu, we change the theme to the **Light**...(7.1.10) ... and click on **Apply**.

Figure 7.1: The installation steps for PgModeler under Ubuntu Linux (continued).



(7.1.11) In the **Relationships** register, we make sure that the connection mode is set to "Crow's foot notation."

Figure 7.1: The installation steps for PgModeler under Ubuntu Linux (continued).

7.2 Installing PgModeler under Microsoft Windows

PgModeler is a software leaning heavily to the Linux side. Installing it under Microsoft Windows as-is is not possible (at least not for free). The PgModeler website recommends to first install the Minimal SYStem 2 (MSYS2). MSYS2 is a collection of tools and libraries from the Linux world providing an environment for building, installing, and running native Microsoft Windows software [455]. One can use this build environment under Microsoft Windows and then to build the program's binaries from its sources in MSYS2. This is very tedious and also error-prone, as there may be issues with package incompatibilities, issues with using the right paths, etc. Also, it requires quite some disk space. Luckily, if we install MSYS2, we can also download and install PgModeler via the package manager pacman [362, 476] used by MSYS2. This is much less work and we will therefore follow this path here.

Therefore, we first want to download MSYS2. The download is available directly at the website <https://msys2.org>, but the link provided there goes to the GitHub releases page of the project, which is not always stable. Thus, we instead visit the website <https://repo.msys2.org/distrib> with our web browser. Most Microsoft Windows systems are 64 bit computers of the x86 architecture. If this is the case for your system – which it most likely is – then you want to download `msys2-x86_64-latest.exe`. We click on the corresponding link in Figure 7.2.1.

The download starts. After it is completed, we run the installer by clicking `Open file` or whatever option your web browser offers to run a downloaded program in Figure 7.2.2.

The welcome screen of the installer opens in Figure 7.2.3. We click on `Next`. Then, we get to choose the installation location. The installer wants to install MSYS2 into the default location `C:\msys64`. We have no reason to disagree. We leave it at this default setting and click on `Next` in Figure 7.2.4. In the following screen, we can choose the start menu location. We again have no reason to change this. We leave the start menu settings as-is and click `Next` in Figure 7.2.5.

The installation process begins with unpacking the archive in Figure 7.2.6. In my case, when the installer reached about 50% as shown in Figure 7.2.7, it stalled for quite some time. It looked like nothing happens and maybe the installer hung. However, this was not the case: The installation process is just doing something time consuming here. If it seems that the installer is hanging in your system too – simply ignore it. Go and drink a cup of tea. It will be OK. Do not stop the process.

Eventually the installation is done. In the final screen, we can mark “Run MSYS2 now.” and click on `Finish`, as shown in Figure 7.2.8. From now on, we can also run the MSYS2 terminal manually. We therefore enter `MSYS2` in the start menu launcher of Microsoft Windows. The icon group for MSYS2 will appear, as shown in Figure 7.2.9, and we simply click on it.

Either way, the MSYS2 terminal is now running in Figure 7.2.10. As stated before, MSYS2 uses the pacman package manager [362, 476]. We want to use this package manager to install PgModeler. Therefore, we first search for existing PgModeler packages that we could install by typing `pacman -Ss pgmodeler` and hitting ↵ as shown in Figure 7.2.11.

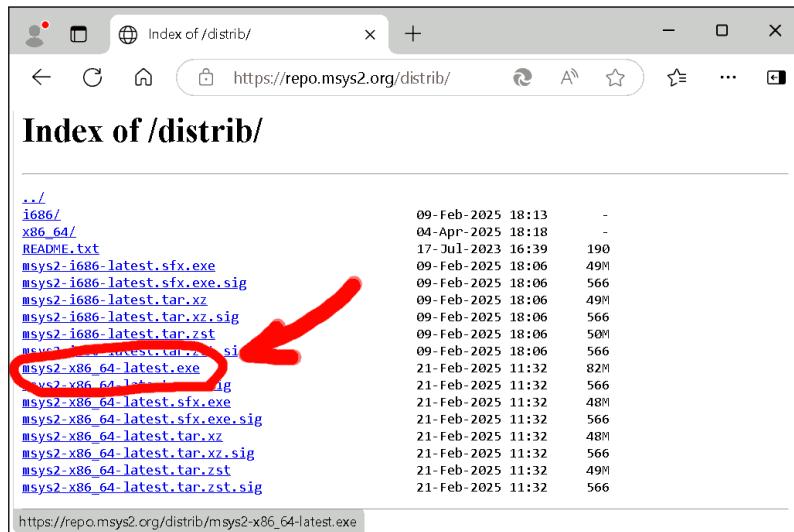
Indeed, we can find several such packages. We want the `mingw64`-one, but *not* the plugins package. The suitable package in Figure 7.2.12 therefore is `mingw64/mingw64-w64-x86_64-pgmodeler`. If you want to avoid typing this long name or if the name is different in your list, you simply select the text most similar to that. Then you can right-click into the terminal and click on `Copy`, as shown in Figure 7.2.13. Then we type `pacman -S` and right-click into the terminal and click on `Paste` in Figure 7.2.14. This means that we wrote, in total, `pacman -S mingw64/mingw64-w64-x86_64-pgmodeler`. This is the command to install PgModeler under MSYS2 in Microsoft Windows using pacman. We hit ↵ in Figure 7.2.15.

The system now shows us the package itself and the required dependencies that would be installed. It asks us whether we are OK with downloading and installing them. Unfazed, we type `Y` and hit ↵ in Figure 7.2.16. The packages are now downloaded and installed. We get back to the terminal prompt in Figure 7.2.17.

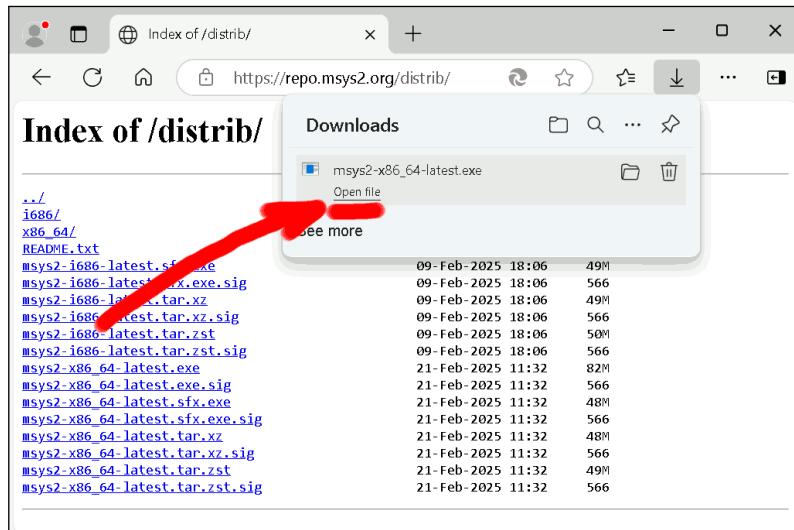
It is now important to **close and re-open** the MSYS2 terminal. After the terminal is opened again, we can type `pgmodeler` and hit ↵. The PgModeler is starting up in Figure 7.2.18.

If it does not, then some simple additional steps are required. We therefore re-open the MSYS2 terminal, as shown in Figure 7.2.20. Then we type in `nano ~/.bashrc` and hit ↵ (Figure 7.2.21). The minimalistic nano text editor opens in Figure 7.2.22. We press `Ctrl`+`End` to scroll to the end of the file and arrive there in Figure 7.2.23.

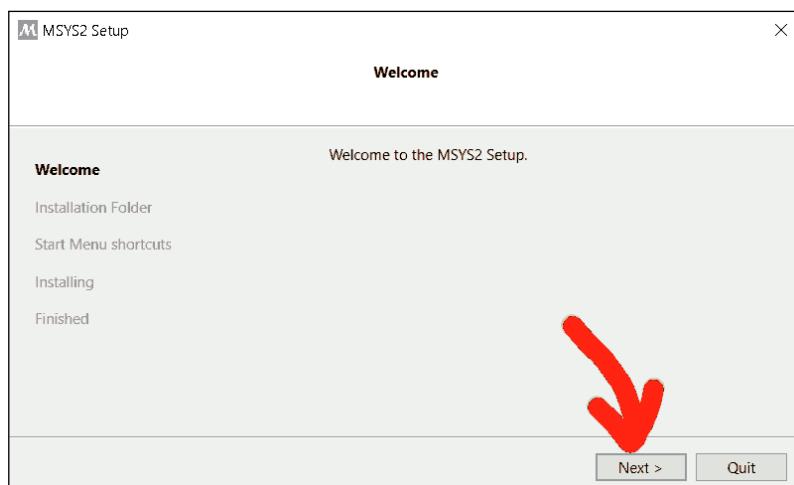
Now we type in `export PATH=$PATH:/mingw64/bin`, as shown in Figure 7.2.24. We press `Ctrl`+`O`,



(7.2.1) We first want to download Minimal SYStem 2 (MSYS2). We therefore visit the website <https://repo.msys2.org/distrib>. We click and download `msys2-x86_64-latest.exe`, if we have a 64 bit x86 system (which usually should be the case).

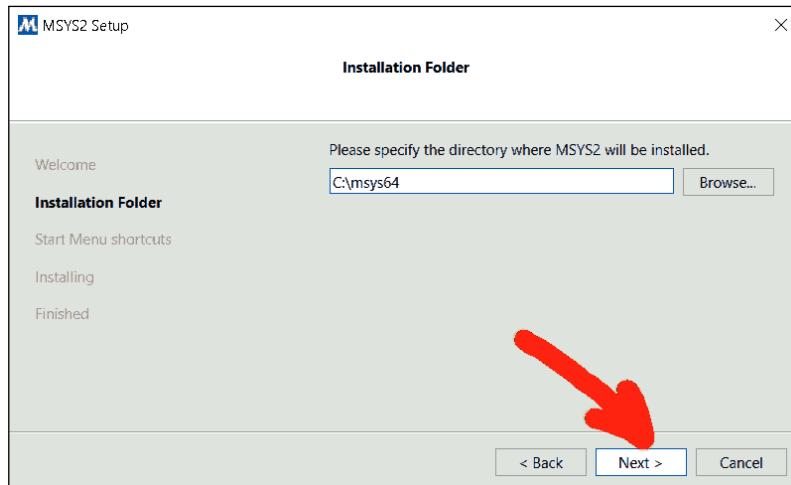


(7.2.2) After the download has completed, we run the installer by clicking `Open file` or whatever option your web browser offers to run a downloaded program.

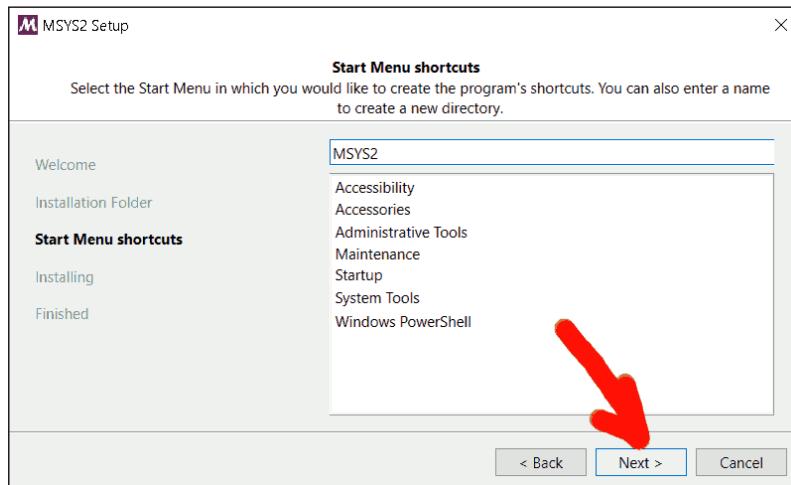


(7.2.3) We click `Next` on the installer welcome screen.

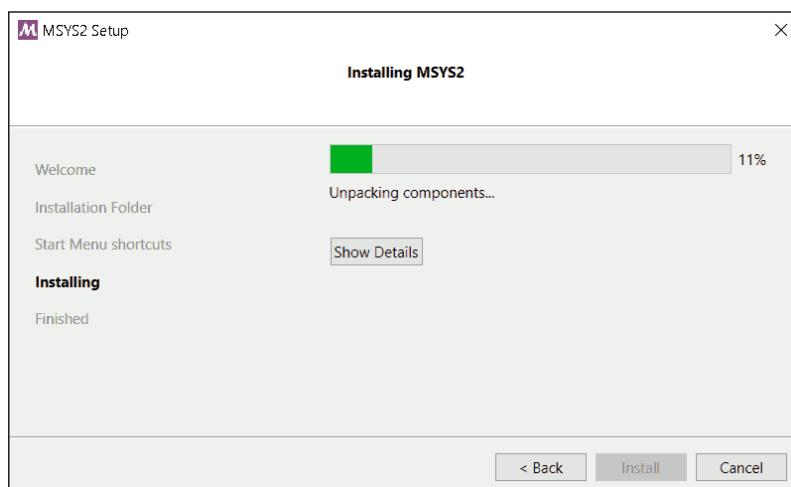
Figure 7.2: Installing PgModele under Microsoft Windows using the MSYS2 environment.



(7.2.4) The installer wants to install MSYS2 into `C:\msys64`. We agree and click `Next`.

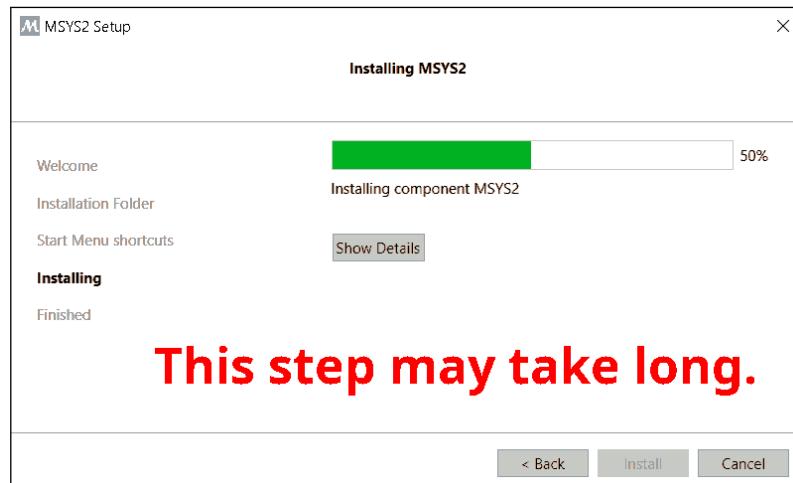


(7.2.5) We leave the start menu settings as-is and click `Next`.

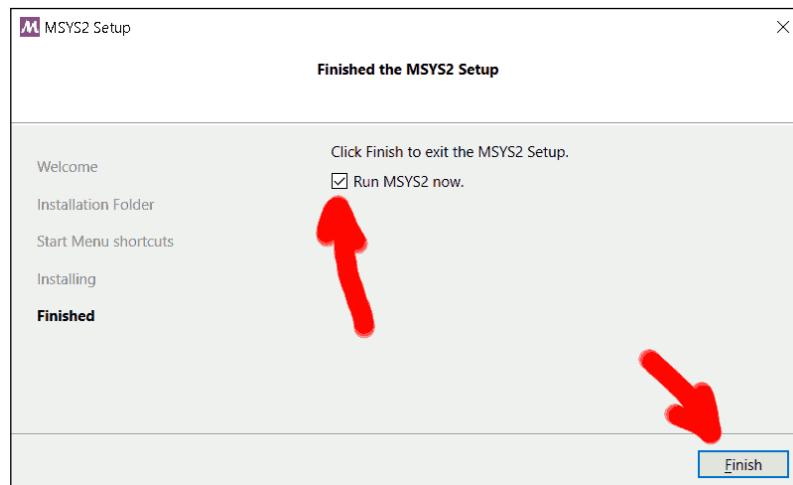


(7.2.6) The installation process begins with unpacking the archive.

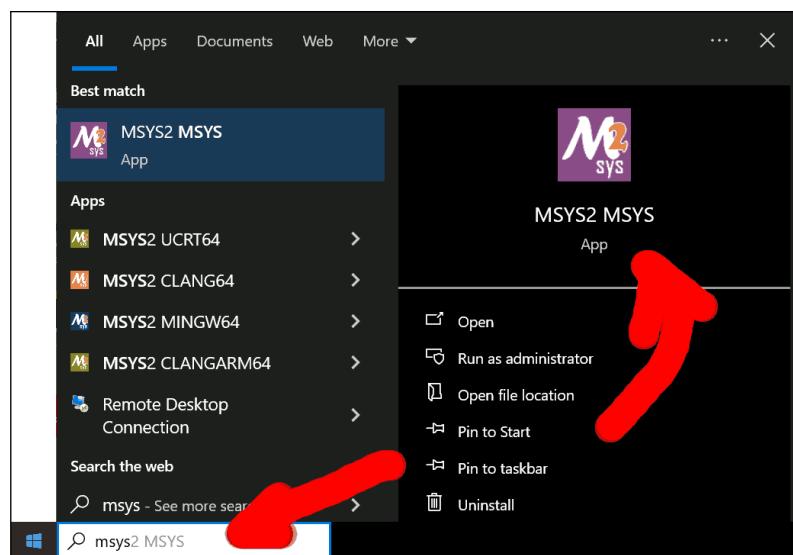
Figure 7.2: Installing PgModeler under Microsoft Windows using the MSYS2 environment (continued).



(7.2.7) When the installer reaches about 50%, it can happen that it stalls for a long time. Just wait. It will be OK. Do not stop the process.

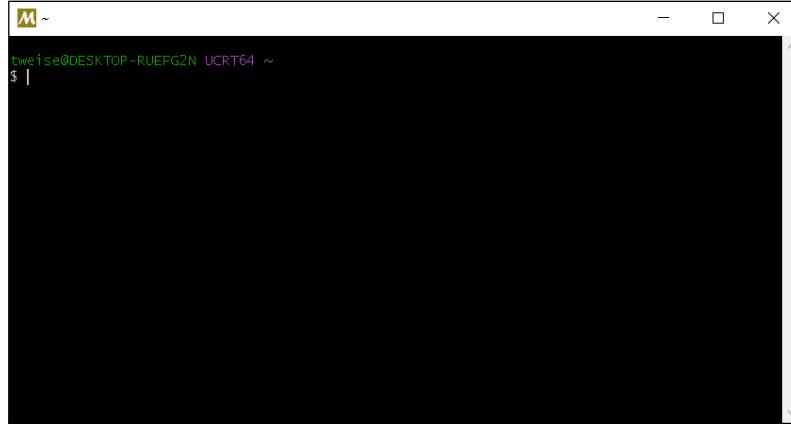


(7.2.8) Eventually the installation is done. We can mark "Run MSYS2 now." and click on **Finish**.



(7.2.9) From now on, we can also run MSYS2 by entering **MSYS2** in the start menu launcher and click on the MSYS2 icon.

Figure 7.2: Installing PgModeler under Microsoft Windows using the MSYS2 environment (continued).



(7.2.10) The MSYS2 terminal is now running.

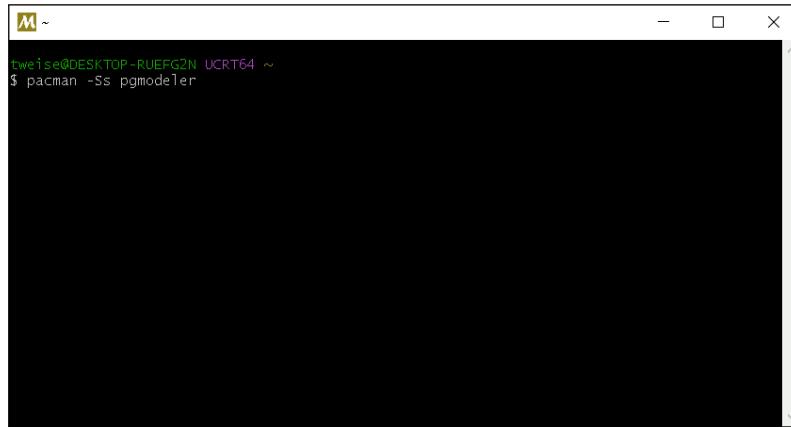
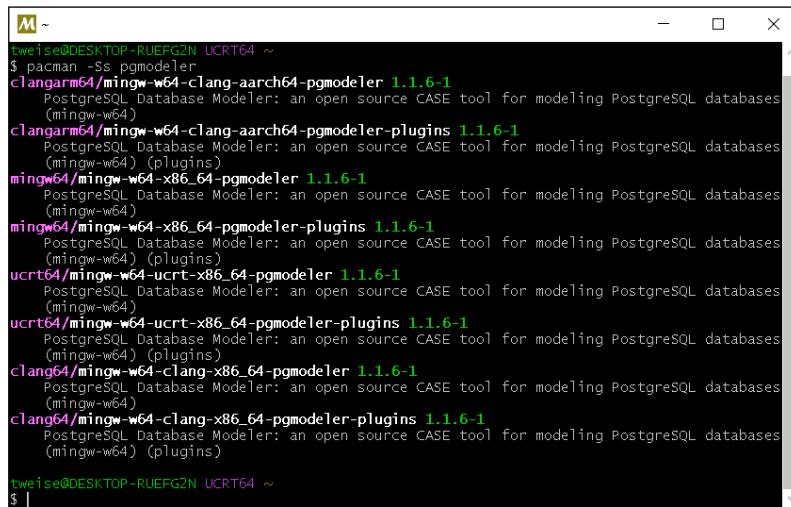
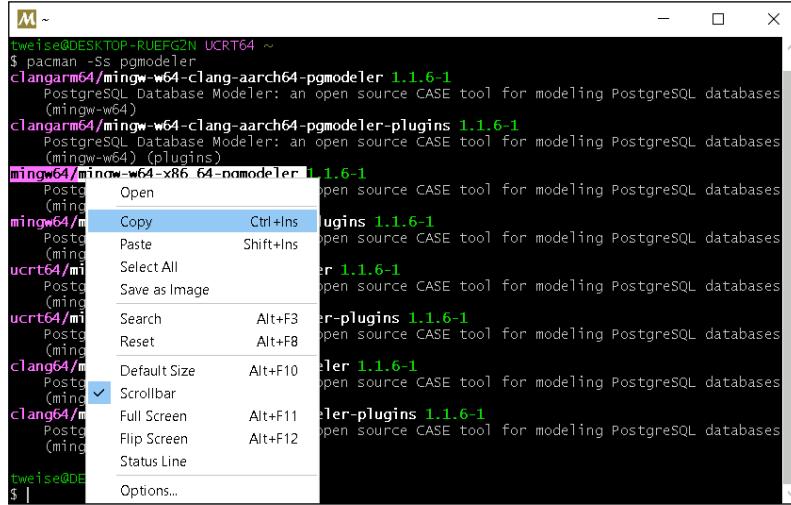
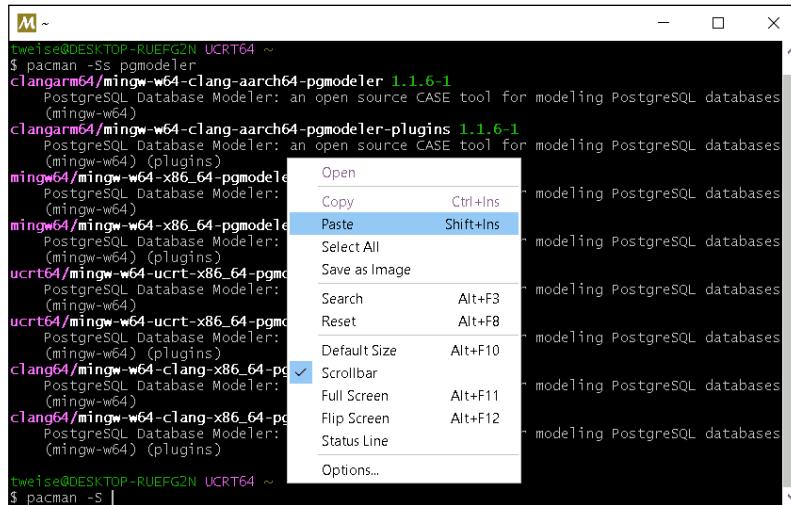
(7.2.11) MSYS2 uses the `pacman` package manager. We can search for existing PgModeler packages by typing `pacman -Ss pgmodeler` and hitting ↵.(7.2.12) Indeed, we can find several! We want the `mingw64`-one (but not the plugins). Next, we select the text `mingw64/mingw64-w64-x86_64-pgmodeler`.

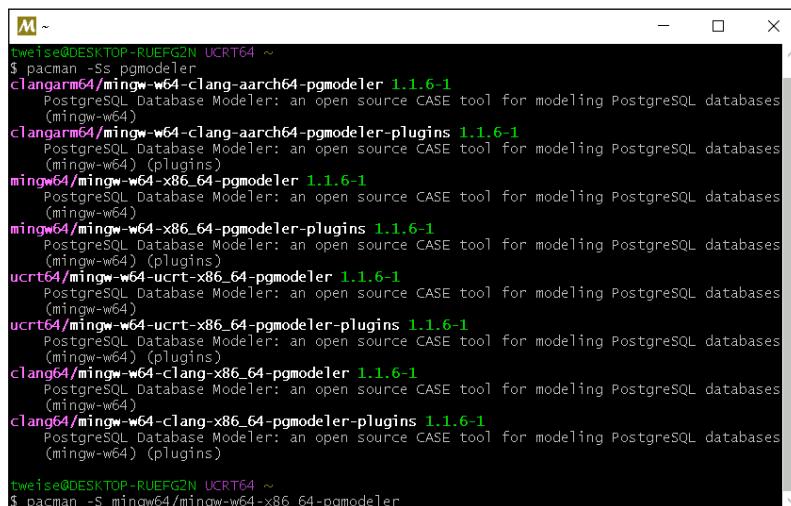
Figure 7.2: Installing PgModeler under Microsoft Windows using the MSYS2 environment (continued).



(7.2.13) So we select the text `mingw64/mingw-w64-x86_64-pgmodeler`. We right-click into the terminal and click on `Copy`.



(7.2.14) Then we type `pacman -S` and right-click into the terminal and click on `Paste`.



(7.2.15) This means that we wrote, in total, `pacman -S mingw64/mingw-w64-x86_64-pgmodeler`. We hit ↵.

Figure 7.2: Installing PgModeler under Microsoft Windows using the MSYS2 environment (continued).

```

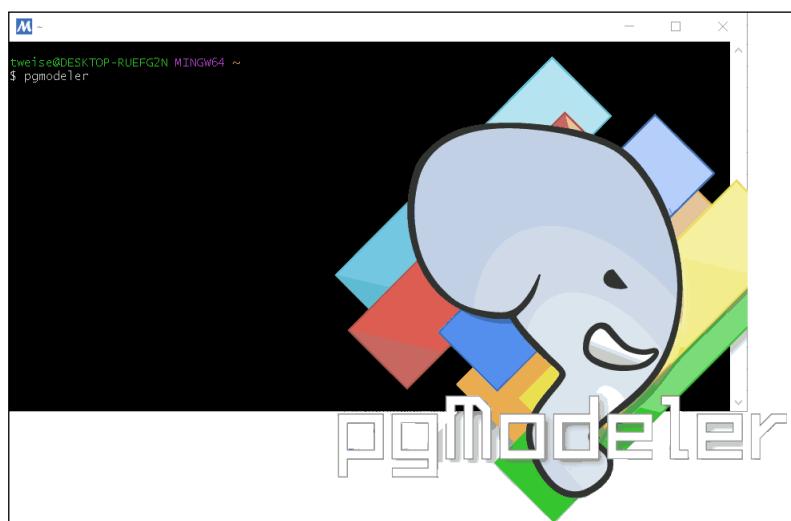
mingw-w64-x86_64-dbus-1.14.10-1 mingw-w64-x86_64-double-conversion-3.3.1-1
mingw-w64-x86_64-expat-2.6.4-1 mingw-w64-x86_64-freetype-2.13.3-1
mingw-w64-x86_64-gcc-libs-14.2.0-2
mingw-w64-x86_64-gettext-runtime-0.23.1-1 mingw-w64-x86_64-glib2-2.82.4-1
mingw-w64-x86_64-graphite2-1.3.14-3 mingw-w64-x86_64-harfbuzz-10.2.0-2
mingw-w64-x86_64-icu-76.1-1 mingw-w64-x86_64-libbb2-0.98.1-2
mingw-w64-x86_64-libiconv-1.18-1
mingw-w64-x86_64-libjpeg-turbo-3.1.0-1 mingw-w64-x86_64-libpng-1.6.46-1
mingw-w64-x86_64-libsysotre-1.0.1-6 mingw-w64-x86_64-libtre-0.9.0-1
mingw-w64-x86_64-libwinpthread-git-12.0.0.0-r509_g079e6092b-1
mingw-w64-x86_64-libxml2-2.12.10-1 mingw-w64-x86_64-lz4-1.10.0-1
mingw-w64-x86_64-m4-0.5.2-1 mingw-w64-x86_64-mpdecimal-4.0.0-1
mingw-w64-x86_64-nurses-6.5.20241228-3 mingw-w64-x86_64-openssl-3.4.1-1
mingw-w64-x86_64-pcre2-10.45-1 mingw-w64-x86_64-postgresql-17.2-2
mingw-w64-x86_64-python-3.12.9-3 mingw-w64-x86_64-python-packaging-24.2-1
mingw-w64-x86_64-qtx-base-6.8.2-1 mingw-w64-x86_64-qtx-svg-6.8.2-1
mingw-w64-x86_64-readline-8.2.013-1 mingw-w64-x86_64-sqlite3-3.47.2-1
mingw-w64-x86_64-tcl-8.6.13-1 mingw-w64-x86_64-termcap-1.3.1-7
mingw-w64-x86_64-tk-8.6.13-1 mingw-w64-x86_64-tzdata-2023a-1
mingw-w64-x86_64-vulkan-headers-1.4.304.1-1
mingw-w64-x86_64-vulkan-loader-1.4.304.1-1
mingw-w64-x86_64-wineditline-2.208-1 mingw-w64-x86_64-xz-5.6.4-1
mingw-w64-x86_64-zlib-1.3.1-1 mingw-w64-x86_64-zstd-1.5.7-1 winpty-0.4.3-3
mingw-w64-x86_64-pgmodeler-1.1.6-1

Total Download Size: 111.87 MiB
Total Installed Size: 699.26 MiB
:: Proceed with installation? [Y/n] y

```

(7.2.16) We get a list of packages that will be installed and are asked whether we are OK with installing them. Devoid of emotion, we type **[Y]** and hit **[Enter]**.

(7.2.17) The packages are downloaded and installed. We get back to the terminal prompt.



(7.2.18) We close and re-open the MSYS2 terminal. We type **pgmodeler** in the terminal and hit **[Enter]**.

Figure 7.2: Installing PgModeler under Microsoft Windows using the MSYS2 environment (continued).



(7.2.19) The PgModeler main window opens! We could now change the theme to light, as done in Figures 7.1.8 to 7.1.10.

Figure 7.2: Installing PgModeler under Microsoft Windows using the MSYS2 environment (continued).



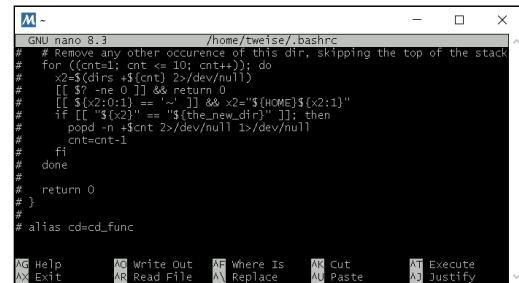
(7.2.20) We open the MSYS2 terminal.



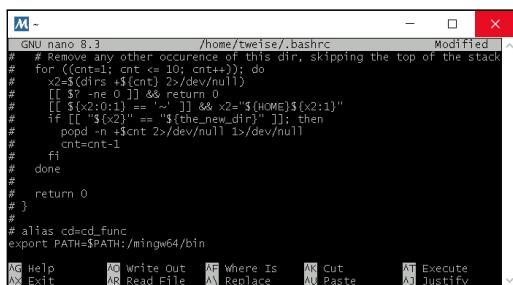
(7.2.21) We type in `nano ~/.bashrc` and hit ↵.



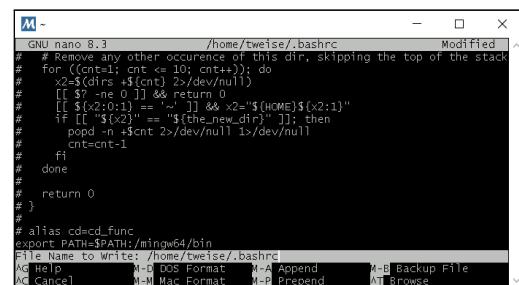
(7.2.22) The minimalistic `nano` text editor opens. We press `Ctrl+End` to scroll to the end of the text.



(7.2.23) The `nano` text editor scrolls to the end of the file.



(7.2.24) We type in `export PATH=$PATH:/mingw64/bin`.



(7.2.25) We press `Ctrl+O` to save the file. Then we press `Enter` when asked whether to store the text in the (original) file.

Figure 7.2: Installing PgModeler under Microsoft Windows using the MSYS2 environment (continued).

```

GNU nano 8.3          /home/tweise/.bashrc
# Remove any other occurrence of this dir, skipping the top of the stack
# for ((cnt=1; cnt <= 10; cnt++)); do
#   x2=${dirs[$cnt]} 2>/dev/null
#   [[ $2 == ~ ]] && return 0
#   [[ ${x2:0:1} == '~' ]] && x2="${HOME}${x2:1}"
#   if [[ ${x2:2} == "${the_new_dir}" ]]; then
#     popd "+$cnt" 2>/dev/null
#   fi
#   done
#   return 0
# }
# alias cdcd_func
export PATH=$PATH:/mingw64/bin

```

(7.2.26) After pressing **[Enter]**, the text is written to the file. We now press **Ctrl+X** to quit the editor.

(7.2.27) We are now back in the terminal, which we close. The next time we open it, the **pgmodeler** command should work.

Figure 7.2: Installing PgModeler under Microsoft Windows using the MSYS2 environment (continued).

which causes the changed file to be saved. Then we press **[Enter]** when asked whether to store the text in the (original) file (Figure 7.2.25). After pressing **[Enter]**, the text is written to the file. We now press **Ctrl+X** to quit the editor in Figure 7.2.26.

We are now back in the **MSYS2 terminal**, which we close in Figure 7.2.27. The next time we open it, the **pgmodeler** command should work.

We thus enter **pgmodeler** command in a newly opened MSYS2 terminal. The **PgModeler** main window has opened in Figure 7.2.19. If you want, you can change its color theme to light, as done in Figures 7.1.8 to 7.1.10. At this stage, the program is running and usable.

7.3 Installing PgModeler under MacOS

Installing the **PgModeler** under **macOS** is similar to the installation under **Microsoft Windows**. Since I do not have an Apple computer, this section needs to survive without screenshots. The basic steps that you will have to take are as follows:

1. Go to <https://www.macports.org> and then to the register “*Installing MacPorts*” – or directly visit <https://www.macports.org/install.php>.
2. Download the MacPorts version suitable for your version of macOS. For example, if you have “macOS Ventura”, then you would download the installer for “macOS Ventura”
3. Install the downloaded version of MacPorts.
4. Update the package list of the MacPorts system.
 - a) Open a *new* terminal window.
 - b) Write `sudo port -v selfupdate` and press **[Enter]**. It will ask you for your (**sudo**) password, i.e., the (administrator) password for your computer. Enter it and hit **[Enter]**.
 - c) After the command completes, close the terminal.
5. We also need **Xcode** as a dependency. So if you do not yet have it installed...
 - a) Open a *new* terminal window.
 - b) Type in `xcode-select --install` and press **[Enter]**. If you need to provide your password again, do so.
 - c) After the installation completes, close the terminal.
6. Finally, we should be able to install the **PgModeler** package.
 - a) Open a *new* terminal window.
 - b) Enter `sudo port install pgmodeler` and press **[Enter]**. It will ask you for your (**sudo**) password, i.e., the (administrator) password for your computer. Enter it and hit **[Enter]**.
 - c) After the installation completes, close the terminal.

7. Open a *new* terminal window.
8. You should now be able to execute PgModeler using the command `pgmodeler` + `Enter`.

The above follows a similar pattern compared as in the Microsoft Windows situation. We first need some abstraction layer that makes Linux software available on this OS. What MSYS2 offers for Microsoft Windows, MacPorts offers for macOS [366]. This time, we also need an additional dependency, namely Xcode. If it is not yet installed, we install it. After that, PgModeler can be installed. And after that, it can be used.

Part II

A Simple Example: The Factory Database

Before we step-by-step learn about the features and intricacies of DBs, let us look into a simple self-contained example as a teaser. So far, you have learned about the history of DBs. Many courses that I know then, from here on, focus on several quite useful things: First, an outline of the DB design process is given. Second, you will learn about modeling data, how to draw ERDs. Third, you will learn about normalization of data. Fourth, you will learn the so-called σ -algebra, which is an mathematical notation abstracting from the technological aspects of relational databases. Fifth, they will tell you how to select, insert, and remove data using SQL. Some will give really practical examples and let you explore how to work with a DB via homeworks. Sixth and finally, you are taught how DBs work internally, what datastructures they use, and how they achieve efficient storage and high query performance.

I agree that it is a good didactic method to approach DBs from several different angles as an abstract subject. However, when I learned something as a student, I never really learned it this way. I learned it by doing it. I am a believer that practical ability is nine tenth of mastering a field. And practical ability comes easiest by playing around with the basic tools.

Also, as far as I can see, quite a few courses seem to treat DBs as something “single,” something that exists “separately” from other things. But this is not necessarily true. Many of the subjects you learn elsewhere may be connected to DBs. For example, you often have programming classes, where you learn a programming language like Python [482]. What is the connection here? You will learn that, yeah, DBs are often accessed from program code, maybe by so-called *application servers*, that implement the business logic of an enterprise. How does that work? Often, you will not learn that.

Then again, DBs are also often accessed from GUIs. They maybe allow us to enter information via convenient forms. Because users like office workers would feel puzzled if asked to enter and retrieve information using SQL queries. Maybe we can also print reports with information extracted from the data in the DB.

So far, for you, a DB is just a nondescript thing to store data. But you may have no idea about all the cool things that you could do with a DB. That you could use a DB for your own personal purposes, ranging from keeping track of your finances over managing a bibliography of papers to storing information about your music collection.

Long story short: We will now explore a small and self-contained example not to teach you how exactly to do things, but to show you what is possible. And that many things are possible without in-depth knowledge and lots of work. To make you curious. To invite you to explore things out of your own interest and to circle back to this book to combine practical experience with background information when you like. It is important that this is a *learning by doing* example. When reading the text, please reproduce the example step-by-step on your own computer.

Our example is a DBs for a small company that produces shoes and handbags. Imagine that you were hired to build an information technology (IT) department for the company. On your first day, your boss enters and tells you *Make an application for storing all our product variants and customer orders*.

We use a concrete technological environment for our work. In particular, we rely on the PostgreSQL ecosystem. We chose it because it is a very mature open source DBMS. It is widely adopted and was the most popular DBMS in the “Stack Overflow 2024 Developer Survey” [417] and in the open source code survey [315], it ranked second.

Useful Tool 1

PostgreSQL [161, 306, 339, 433] is an advanced relational DBMS. It is free and open source and the basis for all hands-on examples in our course.

In this example, you will learn several things. You will learn about the general structure and primitives of relational databases. You will learn that a DB consists of tables, that tables have columns and rows. The rows represent the single records that are stored. All the records in a table have the same attributes, each of which corresponds to a column. You will learn how to create such tables, how to store records in them, and how to query the records from the tables. This is the core of relational databases.

Talking about creating tables, storing records, and querying them immediately leads to the question “How?” We said that we will use PostgreSQL. Will the “How?” that we learn be limited to this system and useless for other DBs? Or will it also work for MariaDB, Microsoft SQL Server, Oracle Database, MySQL, SQLite, and so on? Yes it will. Because we will interact with the DB using the language SQL,

which basically is a programming language for relational databases [72, 117, 123, 131, 222, 291, 412, 419, 420, 431]. Of course, we cannot exhaustively discuss SQL in the framework for a simple example. But you will see some, well, examples of its use.

Finally, we will briefly touch the other topics mentioned earlier, i.e., forms, reports, and access from a programming language. So, without further ado, here we go.

Chapter 8

Creating a User and the Database

Our boss has asked us to create an application for managing products, customers, and demands. Since these are different types of entities with different properties, managing with an [Microsoft Excel](#) or [LibreOffice Calc](#) spreadsheet makes little sense. We will need a [DB](#).

The first step to fulfill this request would thus be to create a new and empty DB. We already installed [PostgreSQL](#) (see [Chapter 3](#)). It is running on a dedicated [server](#) computer in our small [IT](#) department / office. This DB server will host all the DBs of the company, probably ranging from payroll data to fancy business analytics.

However, when we discuss the idea for a new DB application with our boss, they state that they want to have full access to the new DB. Of course, they are not a trained [database administrators \(DBAs\)](#). Many things could go wrong if we would design a nice DB and then unleash untrained personnel onto it. We would be even more reluctant to give them administrative access to the complete [DBMS](#) server. This server could house many different DBs for different purposes. We want to keep it under our control and, at least, limit the “full access” of our boss to only this one single new DB.

In a first step, we would therefore create a new role or user account on our DBMS. This account should only be able to access the new factory DB. If they make a mistake, this mistake will only affect this single DB. If some outside attacker can obtain their password, then the impact will only be limited to this DB and not affect, e.g., payroll data or other confidential data in other DBs. After such a user account is created, we can then create the actual DB and have the new user be its owner.

8.1 Creating a User

For the sake of simplicity, assume that we are locally logged in the DB server computer and that the password to the administrator user `postgres` is set to `XXX`. As illustrated in [Figure 8.1.1](#), we first open a terminal (console). This can be done via [`Ctrl+Alt+T`](#) under Ubuntu Linux, while under Microsoft Windows, you need to press [`Windows + R`](#), type in `cmd`, and hit [`Enter`](#). We want to start the client program `psql` with the proper connection [Uniform Resource Identifier \(URI\)](#) [98] to access our [PostgreSQL](#) server.

Useful Tool 2

`psql` is a text-based console program that can be used to connect to a [PostgreSQL](#) server. From the `psql` console, we can send SQL commands to the [PostgreSQL](#) server and receive its answers.

We can connect to the [PostgreSQL](#) server on our local machine by providing the connection URI `postgres://postgres:XXX@localhost`. This URI is constructed as follows [98]:

- The `postgres://` tells `psql` that this is, indeed, a connection URI.
- The second occurrence of `postgres` is actually the user name. As you may remember from the installation, the administrator user for the whole [PostgreSQL](#) server has the name `postgres`.
- The colon “`:`” separates the user name from the password `XXX`. Of course, you should not actually use a password like `XXX`. Please replace it with a password you deem reasonable. Let

```
tweise@weise-laptop: $ psql postgres://postgres:XXX@localhost
```

(8.1.1) We open a console via **[Ctrl]+[Alt]+[T]** under Ubuntu Linux or by press **[Win]+[R]**, type in **cmd**, and hit **[Enter]** under Microsoft Windows. We type in the command to connect the `psql` client to the `PostgreSQL` server listening at the default port on our current computer (`localhost`) and tell it to log in as user `postgresql` with the password `XXX` (which you need to replace with whatever password you are using) and hit **[Enter]**.

```
tweise@weise-laptop: $ psql postgres://postgres:XXX@localhost
psql (16.9 (Ubuntu 16.9-0ubuntu0.24.04.1))
SSL connection (protocol: TLSv1.3, cipher: TLS_AES_256_GCM_SHA384, compression: off)
Type "help" for help.

postgres=#
```

(8.1.2) `psql` is now running and we can enter commands.

```
tweise@weise-laptop: $ psql postgres://postgres:XXX@localhost
psql (16.9 (Ubuntu 16.9-0ubuntu0.24.04.1))
SSL connection (protocol: TLSv1.3, cipher: TLS_AES_256_GCM_SHA384, compression: off)
Type "help" for help.

postgres=# CREATE USER boss WITH ENCRYPTED PASSWORD 'superboss123';
```

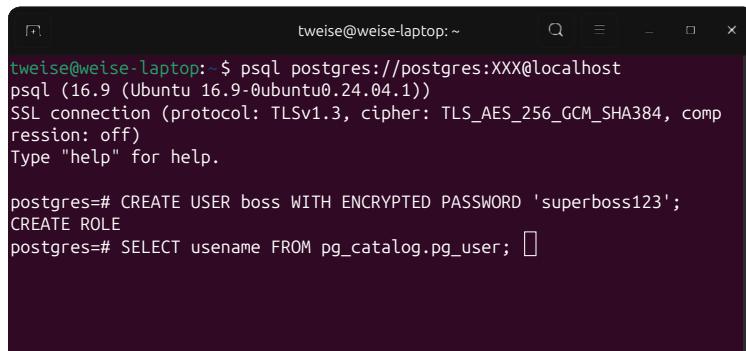
(8.1.3) We type in the command `CREATE_USER` to create the new user `boss` with the password `superboss123`. We hit **[Enter]**.

```
tweise@weise-laptop: $ psql postgres://postgres:XXX@localhost
psql (16.9 (Ubuntu 16.9-0ubuntu0.24.04.1))
SSL connection (protocol: TLSv1.3, cipher: TLS_AES_256_GCM_SHA384, compression: off)
Type "help" for help.

postgres=# CREATE USER boss WITH ENCRYPTED PASSWORD 'superboss123';
CREATE ROLE
postgres#
```

(8.1.4) The command is executed successfully and prints its output `CREATE_ROLE`.

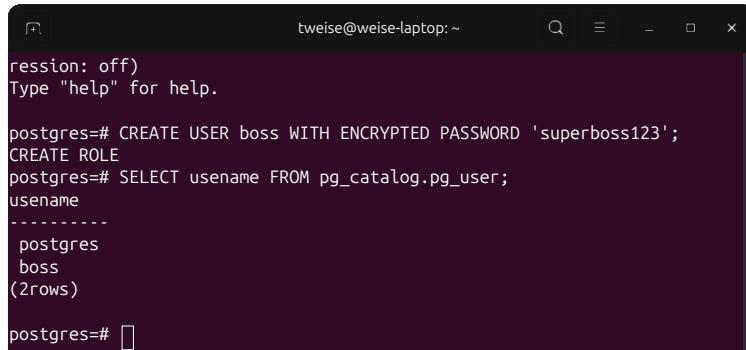
Figure 8.1: Creating the new user/role `boss` with password `superboss123` on the `PostgreSQL` server via the `psql` client.



```
tweise@weise-laptop:~$ psql postgres://postgres:XXX@localhost
psql (16.9 (Ubuntu 16.9-0ubuntu0.24.04.1))
SSL connection (protocol: TLSv1.3, cipher: TLS_AES_256_GCM_SHA384, compres-
session: off)
Type "help" for help.

postgres=# CREATE USER boss WITH ENCRYPTED PASSWORD 'superboss123';
CREATE ROLE
postgres=# SELECT username FROM pg_catalog.pg_user; ┌─┐
```

(8.1.5) To confirm whether the command has succeeded, we now list all users on the PostgreSQL server. We therefore select all user names (`uname`) from the system table `pg_catalog.pg_user`.



```
session: off)
Type "help" for help.

postgres=# CREATE USER boss WITH ENCRYPTED PASSWORD 'superboss123';
CREATE ROLE
postgres=# SELECT username FROM pg_catalog.pg_user;
username
-----
postgres
boss
(2rows)

postgres=# ┌─┐
```

(8.1.6) Besides the DBMS administrator user `postgres`, there now also exists a new user named `boss`.

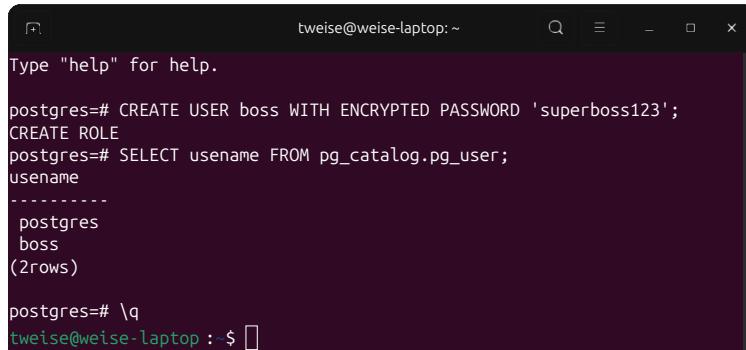


```
session: off)
Type "help" for help.

postgres=# CREATE USER boss WITH ENCRYPTED PASSWORD 'superboss123';
CREATE ROLE
postgres=# SELECT username FROM pg_catalog.pg_user;
username
-----
postgres
boss
(2rows)

postgres=# \q
pq: (standard input) is at end of file
tweise@weise-laptop:~ ┌─┐
```

(8.1.7) We quit this session by typing in the command `\q` and hit `Enter`.



```
Type "help" for help.

postgres=# CREATE USER boss WITH ENCRYPTED PASSWORD 'superboss123';
CREATE ROLE
postgres=# SELECT username FROM pg_catalog.pg_user;
username
-----
postgres
boss
(2rows)

postgres=# \q
tweise@weise-laptop :~ ┌─┐
```

(8.1.8) The psql session has ended and we are back in the terminal.

Figure 8.1: Creating the new user/role `boss` with password `superboss123` on the PostgreSQL server via the `psql` client. (Continued)

Listing 8.1: Using SQL to create a the user `boss` with password `superboss123`. (stored in file `create_user.sql`; output in Listing 8.2)

```

1  /* In this example, we create a new user for our database. */
2
3  -- On PostgreSQL, there is a table `pg_catalog.pg_user` with all users.
4  -- We print the column `username` with the user names.
5  SELECT username FROM pg_catalog.pg_user;
6
7  -- Create the user 'boss'.
8  -- He will be the owner of the database that we will create.
9  CREATE USER boss WITH ENCRYPTED PASSWORD 'superboss123';
10
11 -- Now there is a new user: 'boss'.
12 SELECT username FROM pg_catalog.pg_user;

```

Listing 8.2: The standard output stream (`stdout`) of the program `create_user.sql` given in Listing 8.1.

```

1  $ psql "postgres://postgres:XXX@localhost" -v ON_ERROR_STOP=1 -e bf
   ↪ create_user.sql
2  username
3  -----
4  postgres
5  (1 row)
6
7  CREATE ROLE
8  username
9  -----
10 postgres
11 boss
12 (2 rows)
13
14 # psql 16.12 succeeded with exit code 0.

```

me stress again: Never use something like `XXX` as a password. I am also not doing that. For the examples in the book, I just replaced the actual password with `XXX`.

- After an “@” comes the network address or host name where the **PostgreSQL server** is running. We write `localhost`, which stands for the current machine itself. It corresponds to the IP address `127.0.0.1`. Basically, it means that we want to connect to the **PostgreSQL** server running directly on the very machine in front of which we are sitting.
- After this, we could add a **port**. Since we left the port `5432` at the standard setting during the installation, we do not need to provide it. We could write `:5432` directly after `localhost`, or replace `5432` with whatever port at which the **PostgreSQL** server is listening.
- If we would like to connect to a specific **DB**, say with the name `dbname` then we would then write `/dbname`. But we do not want to do this, because we did not create any DB yet. So we do not specify any DB in the connection **URI**.

Hence, as illustrated in Figure 8.1.2, our connection **URI** is `postgres://postgres:XXX@localhost`. The complete command to launch the **psql client** is thus `psql postgres://postgres:XXX@localhost`. Once the **psql terminal** is open, we can begin typing commands in the **SQL** language.

First Time Readers and Novices: Yes, we now use some SQL commands. You did not yet learn anything about SQL. Do not let that bother you too much. Most of these commands are relatively close to the natural English language. We will explain the commands and datatypes that we use while we are using them. Remember when we said *learning by doing* at the beginning of Part II? We really mean it. You can read more about SQL, for example, in [117, 123, 131, 222, 291, 412, 419, 420, 431].

We want to create a new user for the `psql` server. As username, we pick "boss". The password shall be "superboss123". Obviously, this is a very unsafe password. The boss will have to change it as soon as possible.

As illustrated in Figure 8.1.3, in order to create the new user with that password, we write `CREATE USER boss WITH ENCRYPTED PASSWORD 'superboss123';`. The first part of the command, `CREATE USER xyz`, tells the server to create a new user account under the name `xyz`. The second part, `WITH ENCRYPTED PASSWORD 'abc'`, tells the server that the password `abc` should be used for this user. So we learned our first SQL-command:

```

1 -- Create a New User.
2 --
3 -- userName: the name of the user that we want to create.
4 -- password: the password that we want to assign to this user.
5 CREATE USER userName WITH ENCRYPTED PASSWORD 'password';

```

Passwords are always stored in an encrypted way anyway in PostgreSQL, but it never hurts to specify this clearly. Maybe we want to run the same commands later on another DBMS where it is necessary to explicitly say that passwords shall be encrypted. You can read more about SQL commands in the PostgreSQL reference [412].

Best Practice 3

Regardless which programming language you are using (and we can count SQL as a programming language for DBs), it is important to write code and scripts in a consistent style, to use a consistent naming scheme for all things that can be named, and to follow the generally established best practices and norms for that language.

For many programming languages, there exist comprehensive and clear style guides. Since we usually work collaboratively on larger projects, writing code in a consistent style is very important. Ideally, all collaborators can open a source code file and easily read and understand our code. If everybody writes code in different styles, maybe using different indentations or different naming conventions, reading code can become harder and even confusing. Therefore, style guides often tell us how to name things and how to structure code consistently. For SQL, there does not exist one generally accepted best practice standard style guide. We will still try to define some general rules.

Best Practice 4

Keywords in SQL should always be written completely in uppercase [124].

Well, technically, SQL keywords are not case-sensitive, so `WHERE` and `where` work the same. It is most important to be consistent in your casing, regardless whether you prefer upper- or lowercase [59]. Nevertheless, I prefer uppercase and the PostgreSQL documentation does so too [338]. So we will use uppercase throughout this book and I consider this a best practice.

Anyway, we type in the command and hit `↵`. The command is executed successfully. The system signals this to us with the output `CREATE ROLE` in Figure 8.1.4. `psql` always signals success by typing the command back to us, and under PostgreSQL, `CREATE ROLE` and `CREATE USER` are (almost) the same.

But how can we confirm the user "boss" has really been created successfully? How do we know that it exists now? We can simply list all users. You see, a fully-fledged relational DBMS stores all information in tables, not just the actual data, but also the names of databases, users, and tables themselves (as we will learn later in Section 19.1.3). The result of this is that we can access information about users in the same way as normal data – via SQL queries. OK, as said, you do not yet really know how that works, but for now, just bare with us.

In PostgreSQL, all users are stored in the table named `pg_catalog.pg_user`.¹ So we can just query this table. The SQL command `SELECT username FROM pg_catalog.pg_user;` will list the value of the

¹On other DBMSes, the users may be stored differently.

column `username` for all rows in the table `pg_catalog.pg_user`. We type it in Figure 8.1.5 and hit `\d`. If we had run this command *before* creating the user, on a fresh PostgreSQL installation, it would only list the single user `postgres`, i.e., the administrator of the whole DBMS. But if we run it again now, *after* creating the new user, it will also list `boss`. You can see this in Figure 8.1.6.

With this, we just learned the second SQL-command, which will be a very important asset in our following experiments. In relational DBs, *everything* is stored in tables (even the list of existing users and tables!). While we do not yet know how to store things, we now know how to read – with `SELECT...FROM`. This type of query is structured as follows:

```

1  -- Obtain the values in specific column(s) of a table.
2  --
3  -- This command returns the values of column 'columnName' in table
4  -- 'tableName' for all the rows.
5  -- The result is basically a (temporary) table, with the only the
6  -- selected column(s) from the table 'tableName'.
7  --
8  -- columnName: the name of the column that we want to read.
9  -- tableName: the name of the table to which the column belongs.
10 SELECT columnName FROM tableName;
11
12 -- You can also select multiple columns at once.
13 -- This command here selects N columns from the table.
14 -- The names of the columns are separated by commas (" , ").
15 SELECT columnName1 , columnName2 , ... , columnNameN FROM tableName;
16
17 -- Select all the values from all the columns of a given table.
18 SELECT * FROM tableName;
```

We can now close this `psql` session by typing `\q` and hitting `\d`. As shown in Figure 8.1.7, this takes us back to our normal OS terminal shown in Figure 8.1.8.

Actually, there are two ways to use `psql`: We can either open an interactive session. In such a session, we type in the SQL commands, execute them by pressing `\d`, and then read their output. After that, we can type in the next command, execute it, read its output, and so on. Eventually, we quit by executing `\q`.

The other way to use `psql` is to simply tell it to open a session, execute all the commands in a file (a so-called *script*), and then to close the session. This second way is illustrated in Listings 8.1 and 8.2. Listing 8.1 is the SQL script with the commands to be executed. As you can see, in this script, we first list all the existing users on the DBMS. The only user existing right after the PostgreSQL installation should be `postgres`. So the output of that first command should be only this single value. Then we create the new user via `CREATE USER`. The output of this command, if executed successfully, should be `CREATE USER` printed back to us. Then we query the existing users again, which now should return a table with two values, `postgres` and `boss`. The second listing, Listing 8.2, is the captured output (the so-called *stdout*) of this non-interactive, script-executing `psql` session. It contains exactly the text that we expect. However, its first and last line are different.

The first line in output listings like Listing 8.2 is always marked with *dark red color* and begins with `$`. It contains the actual `psql` execution parameters, which are explained in detail in Table 8.1 and [345]. The last line in the output listings is always marked with *dark blue color* and begins with `#`. It signifies the version of the software and the *exit code*. Notice that you can download the whole example and run exactly the commands (the *red text* after the `$`) from our repository <https://github.com/thomasWeise/databasesCode>.

Table 8.1: The parameters of `psql`, as defined in [345] and used in executions such as Listing 8.2.

```
psql "postgres://user:password@host:port/database"-v ON_ERROR_STOP=1 -ebf script.sql
```

<code>psql</code>	The SQL binary, i.e., the program that is executed.
Connection URI	The connection URI, should best be written inside quotation marks ("..."). <ul style="list-style-type: none">• <code>postgres://</code> indicates that this is, in fact, a connection URI.• <code>user:password</code> are the user name and password.• <code>host</code> is the network address or host name of the computer where the PostgreSQL server is waiting for incoming connections. We usually use <code>localhost</code>, which is the current computer on which <code>psql</code> is executed.• <code>port</code> is the <code>port</code> at which the server is listening, which can be omitted if it is equal to 5432, which is our default setting.• <code>database</code> is the name of the DB to access, which can be omitted if we work on the system itself.
<code>-v ON_ERROR_STOP=1</code>	Tells the program to stop and exit immediately if an error happens. In some examples, we intentionally cause errors to demonstrate problems. <ul style="list-style-type: none">• <code>-e</code> Print all (successful) queries back to the <code>stdout</code>.• <code>-b</code> Print failed queries to the standard error stream (<code>stderr</code>).
<code>-f filename</code>	Read all commands from the file <code>filename</code> .
<code>-ebf filename</code>	Equivalent to <code>-e -b -f filename</code> .

8.2 Creating a new Database

Having created the new user “boss”, we can now create the DB “factory” to be owned and worked on by that user `boss`. For this, we first open a new `psql` session in Figures 8.2.1 and 8.2.2. Notice that we still need to execute this command under the DBA role `postgres`. Also, we can put the connection URI inside of quotes ("..."), which is useful if, for example, the password contains strange characters. This user has the right to create DBs for other users which can then work with them. We then need to execute a single SQL-command, namely:

```
1 -- Create a New Database.
2 --
3 -- 'databaseName': the name of the database that we want to create.
4 -- 'userName': the name of the user that will be the owner of the
5 -- new database.
6 CREATE DATABASE databaseName OWNER userName;
```

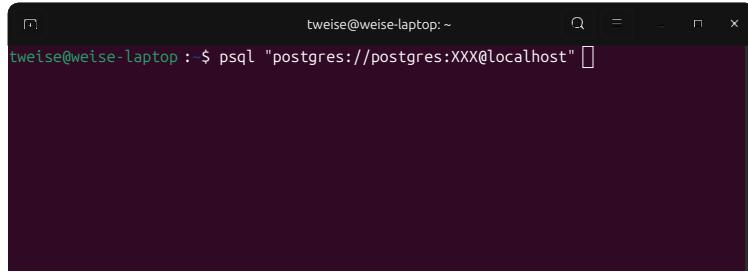
Thus, all we have to do is to type the SQL command `CREATE DATABASE factory OWNER boss;` in Figure 8.2.3. This command pretty much explains itself. It will create a new DB with the name `factory`. The user `boss` will be owner of this DB, i.e., they will have full access to add and manipulate its data. The command completes successful. No error message appears and the command is printed back to us in Figure 8.2.4.

Documentation is a very important task in the whole field of software engineering. It is always a good idea to store lots of comments that explain each DB, table, column, constraint, role, user, view, stored procedure, and whatever other object could exist on a DBMS. Usually, to keep the examples small enough to fit on single pages, we will not have enough space for that. However, here, at our very first DB, let’s do it right: In Figure 8.2.5, we use the `COMMENT ON` command to store a comment that describes the purpose of our DB `factory`. This command also succeeds and is printed back to us in Figure 8.2.6.

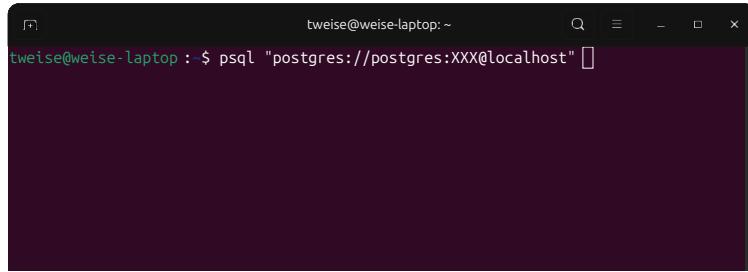
We can also get a list of all the DBs in our system. For this purpose, we write `SELECT datname FROM pg_database;` in Figure 8.2.7. See, all the names of all DBs inside the PostgreSQL DBMS are stored in the column `datname` of the table `pg_database`.² If we run this command before creating the new DB on a fresh PostgreSQL installation, we find that it will list some standard DBs, which we will ignore here.

On the installation where I executed it, there were some more DBs, so the command found seven DBs. Sometimes, if a lot of data is returned by an SQL command, `psql` will change into a paginated

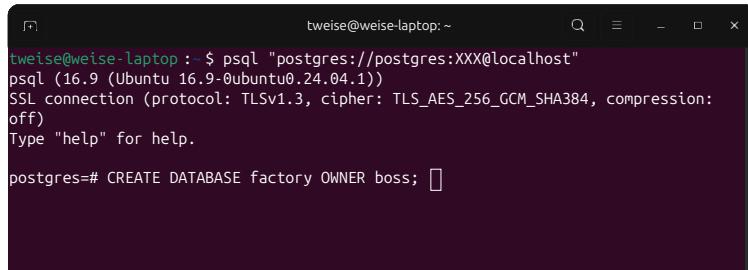
²On other DBMSes, the DBs may be stored differently.



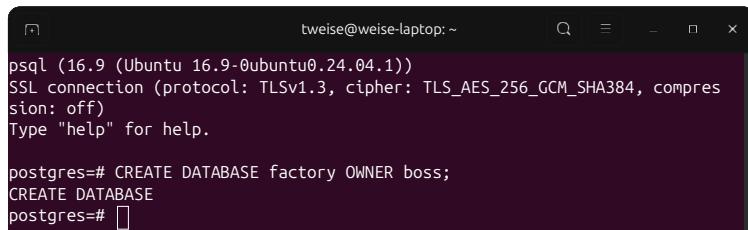
(8.2.1) As in [Figure 8.1.1](#), we open a console via [\[Ctrl\]+\[Alt\]+\[T\]](#) under Ubuntu Linux or by press [\[Windows\]+\[R\]](#), type in [cmd](#), and hit [\[Enter\]](#) under Microsoft Windows. We connect the psql client to the PostgreSQL server listening at the default port on our current computer (localhost) and tell it to log in as user [postgres](#) with the password [XXX](#) and hit [\[Enter\]](#). Notice: We can also put the URI in quotes, which is good if the password contains strange characters.



(8.2.2) As in [Figure 8.1.2](#), the psql session is now open.



(8.2.3) We type in the command `CREATE DATABASE factory OWNER boss` which will create the DB [factory](#). The parameter `OWNER boss` sets the new user [boss](#) to be the owner of this DB.



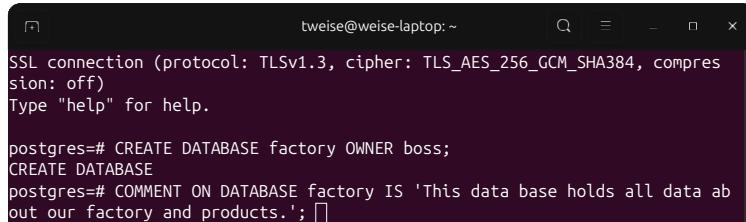
(8.2.4) To indicate success, psql prints the command back to us.

[Figure 8.2: Creating a new DB, adding a comment to it, and checking whether it really was created, all via psql.](#)

mode, as shown in [Figure 8.2.8](#). There, we can see the DBs. We can then exit this mode simply by pressing [\[q\]](#) in [Figure 8.2.9](#).

This takes us back into our [psql](#) terminal session, which we now will leave by typing in [\[\q\]](#) in [Figure 8.2.10](#). This ejects us back into the normal terminal in [Figure 8.2.11](#).

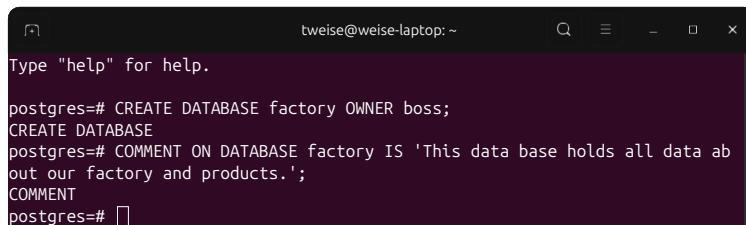
All of the above commands are combined into a single script in [Listing 8.3](#). [Listing 8.4](#) shows their output when being executed on a clean and fresh PostgreSQL installation (but after the user [boss](#) was created, obviously).



```
tweise@weise-laptop: ~
SSL connection (protocol: TLSv1.3, cipher: TLS_AES_256_GCM_SHA384, compression: off)
Type "help" for help.

postgres=# CREATE DATABASE factory OWNER boss;
CREATE DATABASE
postgres=# COMMENT ON DATABASE factory IS 'This data base holds all data about our factory and products.';
```

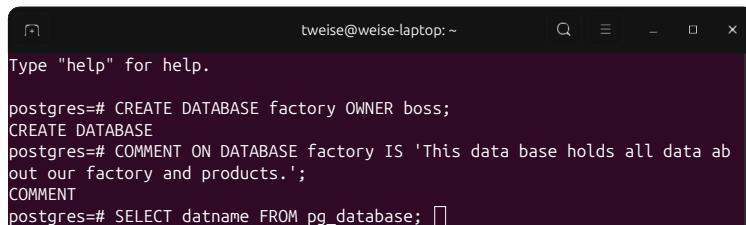
(8.2.5) We can add comments to many of the elements that we create. Comments are good for documenting the meaning and reasons of the DB elements. Therefore, by using the `COMMENT ON ... IS` command, we add some documentation to our new DB.



```
tweise@weise-laptop: ~
Type "help" for help.

postgres=# CREATE DATABASE factory OWNER boss;
CREATE DATABASE
postgres=# COMMENT ON DATABASE factory IS 'This data base holds all data about our factory and products.';
COMMENT
postgres=#
```

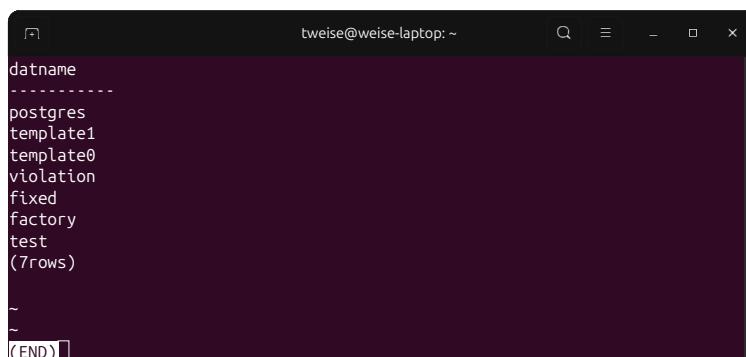
(8.2.6) To indicate success, psql prints the command back to us.



```
tweise@weise-laptop: ~
Type "help" for help.

postgres=# CREATE DATABASE factory OWNER boss;
CREATE DATABASE
postgres=# COMMENT ON DATABASE factory IS 'This data base holds all data about our factory and products.';
COMMENT
postgres=# SELECT datname FROM pg_database;
```

(8.2.7) We now want to see the list of all DBs in our DBMS. We therefore type the command `SELECT datname FROM pg_database;`. `pg_database` is a system table holding all DBs, and its column `datname` contains their names.

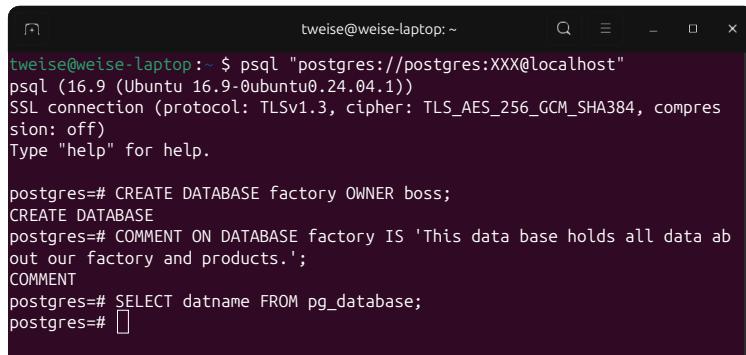


```
tweise@weise-laptop: ~
datname
-----
postgres
template1
template0
violation
fixed
factory
test
(7rows)

~
~
(END)
```

(8.2.8) The command will print all DBs on the current system. On this installation, there are 7 DBs, on your fresh installation, there will be fewer. The command may enter a paginated view, as is the case here.

Figure 8.2: Creating a new DB, adding a comment to it, and checking whether it really was created, all via psql. (Continued)

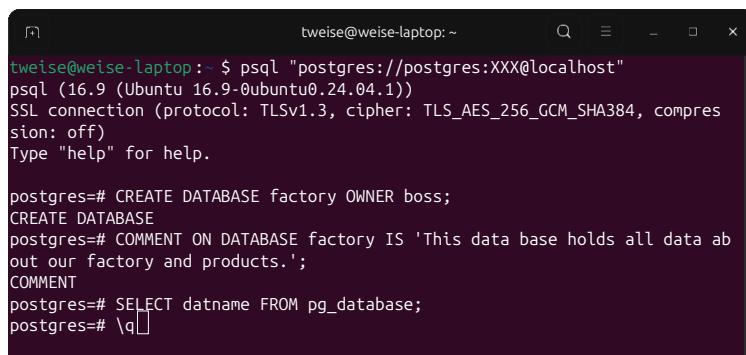


```
tweise@weise-laptop:~ $ psql "postgres://postgres:XXX@localhost"
psql (16.9 (Ubuntu 16.9-0ubuntu0.24.04.1))
SSL connection (protocol: TLSv1.3, cipher: TLS_AES_256_GCM_SHA384, compression: off)
Type "help" for help.

postgres=# CREATE DATABASE factory OWNER boss;
CREATE DATABASE
postgres=# COMMENT ON DATABASE factory IS 'This data base holds all data about our factory and products.';
COMMENT
postgres=# SELECT datname FROM pg_database;
postgres=#

```

(8.2.9) If the paginated view was entered, you can leave it by pressing `\q`. If it was not entered, the command will directly return.

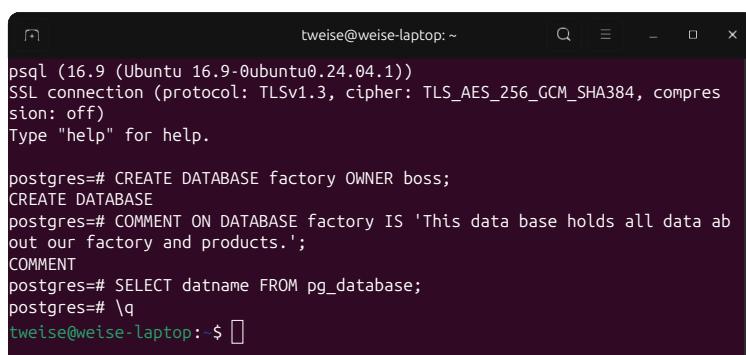


```
tweise@weise-laptop:~ $ psql "postgres://postgres:XXX@localhost"
psql (16.9 (Ubuntu 16.9-0ubuntu0.24.04.1))
SSL connection (protocol: TLSv1.3, cipher: TLS_AES_256_GCM_SHA384, compression: off)
Type "help" for help.

postgres=# CREATE DATABASE factory OWNER boss;
CREATE DATABASE
postgres=# COMMENT ON DATABASE factory IS 'This data base holds all data about our factory and products.';
COMMENT
postgres=# SELECT datname FROM pg_database;
postgres=#
\q

```

(8.2.10) We left the paginated result view and now want to leave this psql session, by typing `\q` and hitting `↵`.



```
tweise@weise-laptop:~ $ psql (16.9 (Ubuntu 16.9-0ubuntu0.24.04.1))
SSL connection (protocol: TLSv1.3, cipher: TLS_AES_256_GCM_SHA384, compression: off)
Type "help" for help.

postgres=# CREATE DATABASE factory OWNER boss;
CREATE DATABASE
postgres=# COMMENT ON DATABASE factory IS 'This data base holds all data about our factory and products.';
COMMENT
postgres=# SELECT datname FROM pg_database;
postgres=#
\q
tweise@weise-laptop:~ $

```

(8.2.11) We are back in the normal terminal.

Figure 8.2: Creating a new DB, adding a comment to it, and checking whether it really was created, all via psql. (Continued)

Listing 8.3: Using SQL to create a database for user `boss`. (stored in file `create_database.sql`; output in Listing 8.4)

```

1  /* In this example, we create a new database named 'factory'. */
2
3  -- On PostgreSQL, there is a table `pg_databases` listing all databases.
4  -- We print the column `datname` with the names of the databases.
5  SELECT datname FROM pg_database;
6
7  -- Create the database 'factory', owned by user 'boss'.
8  CREATE DATABASE factory OWNER boss;
9
10 -- Store a comment about the purpose of our database.
11 COMMENT ON DATABASE factory is
12   'This database holds all data about our factory and products.';
13
14 -- Now there is a new database in the list, namely 'factory'.
15 SELECT datname FROM pg_database;

```

Listing 8.4: The stdout of the program `create_database.sql` given in Listing 8.3.

```

1 $ psql "postgres://postgres:XXX@localhost" -v ON_ERROR_STOP=1 -e bf
2   ↪ create_database.sql
3
4   datname
5   -----
6
7   postgres
8   template1
9   template0
10  (3 rows)
11
12  CREATE DATABASE
13  COMMENT
14    datname
15
16  -----
17  postgres
18  factory
19  template1
20  template0
21  (4 rows)
22
23  # psql 16.12 succeeded with exit code 0.

```

Chapter 9

Creating Tables and Filling them with Data

Let us now design the actual DB. Normally, you would do this in a fancy process where you would draw ERDs and deeply think about the structure of the data, the performance requirements, and so on (see Chapter 16 for details). Be that as it may, we are here operating on a learning-by-doing level. We will just go ahead and build something that looks reasonable, without worrying too much about design principles.

In a relational database, all the data is stored in *tables*. You are maybe familiar with spreadsheet software such as Microsoft Excel [6, 328, 488] or LibreOffice Calc [171, 270, 385]. There, data is organized in tables, too. In a relational database, however, the columns are strongly typed, i.e., you cannot “write” a text into a field for numbers. Also, there can be multiple tables, where a record (row) in one table can be linked to one or multiple records in other tables. This format allows us to nicely divide into our data according to different semantic aspects:

We will create a table for the products that our factory produces. We will create a table for the customers that order these products. And we will create a table for the orders that these customers issue.

9.1 The Table “product”

We begin with storing the information about the products that our company produces and sells. We want to store all information that may be relevant to customers and the delivery department. We will give our first new table the name `product`.

Best Practice 5

Table names should be singular nouns written in lowercase without any prefix (i.e., no “tbl_” in front) [59].

Product data is comprised of different datatypes, ranging from text to numerical values. Our first table will thus help us to get a glimpse of some of the datatypes supported by SQL. More importantly, we will learn how to create tables, how to insert data into them, and how to read the data back from the DB.

9.1.1 Creating the Table

So let us create the table `product`. We start by thinking about what columns does this table need? Well, first of all, each product has a *name*. The name is text, so there should be one column that can host text. Each unit of a product also has a price. So there needs to be a column that can hold the price, i.e., a currency value. In order to deliver products, we need to put them into boxes that have a certain width, height, and depth as well as a weight. These are clearly numbers, too.

This fairly straightforward structure is illustrated in Figure 9.1. All we need to do now is to translate this design to SQL commands, which includes choosing names and SQL-datatypes for the columns. Fittingly, the command for creating tables is called `CREATE TABLE`. As illustrated below, the command

id	name	price	weight	width	height	depth
1	Shoe, Size 36	150.99	1300	350	250	130
2	Shoe, Size 37	152.99	1325	350	250	130
3	Shoe, Size 38	154.99	1350	350	250	130
4	Shoe, Size 39	156.99	1375	350	250	130
5	Shoe, Size 40	158.99	1400	350	250	130
6	Shoe, Size 41	160.99	1425	350	250	130
7	Shoe, Size 42	162.99	1450	350	250	130
8	Shoe, Size 43	164.99	1475	350	250	130
9	Small Purse	100	500	350	250	130
10	Medium Purse	120	750	400	300	200
11	Large Purse	150	1500	600	300	250
...

Figure 9.1: How our table `product` could look like if we designed it in a spreadsheet software like Microsoft Excel or LibreOffice Calc.

is followed by the table name and then, in parentheses, the list of attributes, i.e., columns, which are separated by commas. Each column has a datatype and may be annotated with some validity constraints [107]. The structure of the command looks like this:

```

1  -- Create a New Table.
2  --
3  -- tableName: the name of the table to create
4  -- columnX: the name of the X-th column.
5  -- typeOfColumnX: the datatype to be used for the X-th column.
6  -- constraintsOnColumnX: constraints imposed on the X-th column, can be
7  -- omitted or a combination of UNIQUE, NOT NULL,
8  -- PRIMARY KEY, etc.
9  CREATE TABLE tableName(
10    column1 typeOfColumn1 constraintsOnColumn1,
11    column2 typeOfColumn2 constraintsOnColumn2,
12    column3 typeOfColumn3 constraintsOnColumn3,
13    ...
14 );

```

Here, we will first write down the whole command for creating our table `product` and execute it. Then we discuss all of its parameters and their meanings. We start an interactive `psql` session and type in the commands in Figure 9.2.1. Importantly, we do this as user `boss` with password `SQLsuperboss123`. This user is the owner of our DB `factory` and has the right to create tables and insert data.

We type the full `CREATE TABLE` command in Figure 9.2.3. It is also not very easy to understand, because we are not yet familiar with SQL. When we highlight the components of the commands in Figure 9.2.4, things get a little bit clearer: The command indeed contains the column names and the column datatypes as well as constraints that ensure that the data entered is valid. We will now build the command step-by-step and discuss the meaning of each of its elements.

We begin with the column for the product name, which we will call `name`. As already said, each product must have a name. The names could be long or short, but 100 characters per product should suffice. SQL provides the datatype `VARCHAR`, which is used for variable-length text strings, whose maximum length is specified in parentheses [76]. We could thus choose `VARCHAR(100)` for the column `name` for our table `product`. We thus write `name VARCHAR(100)` when defining this column.

When creating a column, we can also add certain *constraints* for their values right away. For example, product names must be unique. We can never have two different products with the same name. So we add the keyword `UNIQUE` [100]. If, at some point in time, someone tries to enter a record into our table whose `name` value already exists in another record, then this will fail with an error. The DBMS will simply not allow that two rows in table `product` have the same value in column `name`. Nice.

```
tweise@weise-laptop:~$ psql "postgres://boss:superboss123@localhost/factory"
```

(9.2.1) We open a console via **Ctrl**+**Alt**+**T** under Ubuntu Linux or by press **Windows**+**R**, type in **cmd**, and hit **Enter** under Microsoft Windows. We type in the command to connect the `psql` client to the **PostgreSQL** server listening at the default port on our current computer (`localhost`) and tell it to log in as user `boss` with the password `superboss123` and hit **Enter**.

```
tweise@weise-laptop:~$ psql "postgres://boss:superboss123@localhost/factory"
psql (16.9 (Ubuntu 16.9-0ubuntu0.24.04.1))
SSL connection (protocol: TLSv1.3, cipher: TLS_AES_256_GCM_SHA384, compression: off)
Type "help" for help.

factory=>
```

(9.2.2) The session has started.

```
tweise@weise-laptop:~$ psql "postgres://boss:superboss123@localhost/factory"
psql (16.9 (Ubuntu 16.9-0ubuntu0.24.04.1))
SSL connection (protocol: TLSv1.3, cipher: TLS_AES_256_GCM_SHA384, compression: off)
Type "help" for help.

factory=> CREATE TABLE product (
    id      INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
    name    VARCHAR(100) NOT NULL UNIQUE,
    price   DECIMAL(10,2) NOT NULL,
    weight  INT          NOT NULL,
    width   INT          NOT NULL,
    height  INT          NOT NULL,
    depth   INT          NOT NULL);
```

(9.2.3) We enter the `CREATE TABLE` command for the new table `product` with all the column specifications and press **Enter**.

Datatypes

Column Names

Constraints

```
tweise@weise-laptop:~$ psql "postgres://boss:superboss123@localhost/factory"
psql (16.9 (Ubuntu 16.9-0ubuntu0.24.04.1))
SSL connection (protocol: TLSv1.3, cipher: TLS_AES_256_GCM_SHA384, compression: off)
Type "help" for help.

factory=> CREATE TABLE product (
    id      INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
    name    VARCHAR(100) NOT NULL UNIQUE,
    price   DECIMAL(10,2) NOT NULL,
    weight  INT          NOT NULL,
    width   INT          NOT NULL,
    height  INT          NOT NULL,
    depth   INT          NOT NULL);
```

(9.2.4) The components of the command are highlighted for illustration purposes. When creating a table, we provide its name and then all the columns that it should have in parentheses, separated by commas. For each column, we must provide the name and datatype and my provide constraints.

```
tweise@weise-laptop:~$ psql (16.9 (Ubuntu 16.9-0ubuntu0.24.04.1))
SSL connection (protocol: TLSv1.3, cipher: TLS_AES_256_GCM_SHA384, compression: off)
Type "help" for help.

factory=> CREATE TABLE product (
    id      INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
    name    VARCHAR(100) NOT NULL UNIQUE,
    price   DECIMAL(10,2) NOT NULL,
    weight  INT          NOT NULL,
    width   INT          NOT NULL,
    height  INT          NOT NULL,
    depth   INT          NOT NULL);
CREATE TABLE
factory=>
```

(9.2.5) The command succeeds and is printed back to us.

Figure 9.2: Creating a new table in our DB.

```
tweise@weise-laptop:~
```

```
psql (16.9 (Ubuntu 16.9-0ubuntu0.24.04.1))
SSL connection (protocol: TLSv1.3, cipher: TLS_AES_256_GCM_SHA384, compression: off)
Type "help" for help.

factory=> CREATE TABLE product (
    id      INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
    name   VARCHAR(100) NOT NULL UNIQUE,
    price  DECIMAL(10,2) NOT NULL,
    weight INT          NOT NULL,
    width  INT          NOT NULL,
    height INT          NOT NULL,
    depth  INT          NOT NULL);
CREATE TABLE
factory=> SELECT tablename FROM pg_catalog.pg_tables WHERE tableowner = 'boss';
```

(9.2.6) We want to get a list of all tables belonging to user `boss` with this `SELECT` command.

```
tweise@weise-laptop:~
```

```
name   VARCHAR(100) NOT NULL UNIQUE,
price  DECIMAL(10,2) NOT NULL,
weight INT          NOT NULL,
width  INT          NOT NULL,
height INT          NOT NULL,
depth  INT          NOT NULL);

CREATE TABLE
factory=> SELECT tablename FROM pg_catalog.pg_tables WHERE tableowner = 'boss';
tablename
-----
product
(1row)

factory=>
```

(9.2.7) There is now exactly one such table, namely `product`.

```
tweise@weise-laptop:~
```

```
name   VARCHAR(100) NOT NULL UNIQUE,
price  DECIMAL(10,2) NOT NULL,
weight INT          NOT NULL,
width  INT          NOT NULL,
height INT          NOT NULL,
depth  INT          NOT NULL);

CREATE TABLE
factory=> SELECT tablename FROM pg_catalog.pg_tables WHERE tableowner = 'boss';
tablename
-----
product
(1row)

factory=> \q
```

(9.2.8) We end this session by typing in `\q` and hitting `Enter`.

```
tweise@weise-laptop:~
```

```
product
(1row)

factory=> \q
tweise@weise-laptop : $
```

(9.2.9) The session is terminated and we are back in the normal terminal.

Figure 9.2: Creating a new table in our DB. (Continued)

We also want to enforce that every single product record indeed has a `name` value set. There cannot be a product without a name. If this was an [Microsoft Excel](#) or [LibreOffice Calc](#) table, we would not have any means from stopping a user from leaving a cell empty. However, this is a table in a [relational database](#), and we *can* specify that there must never be any row without a proper value in column `name`. To this end, we add the `NOT NULL` constraint. All in all, we write `name VARCHAR(100) UNIQUE NOT NULL` [100]. Very nice. We just defined our very first column.

Next we should create a column for price at which we sell the product. We will call it `price`. The price is clearly a number. Sadly, this early in our first example, we already encounter a bit of a tricky situation. But well, tricky situations are not unusual and it never hurts if you know that, well, sometimes solutions are a bit complicated.

Computers usually provide two fundamental types of numbers on the hardware level, integer numbers and floating point numbers (a subset of \mathbb{R}). These types also exist in many programming languages. [Python](#) [482], for example, offers us the two very basic datatypes `int` and `float` corresponding to integers and floating point numbers, respectively. Which one should we use? Prices naturally are

values with fractions, something like \$99.99 or 17.75元 or 3.79€, so at first glance, the SQL equivalent of Python's `float` seems to be a reasonable choice here. It is actually not, though, because in our companion book *Programming with Python* [482], we can find the following information:

Best Practice 6

Always assume that any `float` value is imprecise. Never expect it to be exact [28, 341, 482].

Taking this sentence at face value already means that floating point numbers are not a suitable choice. Money values must be exact. Always. Bookkeeping and billing are two of the things where you must never ever mess up. Send a wrong shoe to a customer, ship an order to a wrong destination – that may be annoyances that you can recover from. But making mistakes in tax payment to the government because of inconsistencies in your financial records ... that can become really nasty. That being said, let's take a closer look at the issue:

Using double precision IEEE Standard 754 floating point numbers [207, 220], the expression `0.1 + 0.1 + 0.1 - 0.3` may yield `5.551115123125783e-17` as result. Let say that we would store 0.3元 as a price in our DB by using floating point numbers and that a customer transferred three times 一毛钱, i.e., three single dimes, to our account. Then the bank may encounter problems if our system would automatically try to transfer $5.551115123125783 \cdot 10^{-17}$ 元 back as change... We therefore learn that:

Best Practice 7

Never represent monetary data with floating point numbers [376, 485].

So floating point numbers are out. How about integer numbers? We could use integer numbers representing the number of cents instead, but then we would always need to use some arithmetics to properly display prices (divide by 100). This also creates a potential for errors [485].

Actually neither integers nor floating point numbers are the right choice! Instead, we will use the datatype `DECIMAL`, which can represent a fractional number with a pre-defined number of digits exactly. Writing `price DECIMAL(10, 2)` allows us to store values with 10 digits, 2 of these 10 digits are after the comma [301]. This means that we can store values between -99 999 999.99 and 99 999 999.99. This should be enough for prices of products in our shoe and handbag factory. Of course, each product must have a price, so we add again the `NOT NULL` keyword.

Best Practice 8

Store monetary data using the `DECIMAL` datatype [65, 485].

With this, we can move on to the next set of product attributes. We already explained that we will sell each unit of a product in a box. Therefore, we also want to store the width, height, and depth as well as the weight of the packaged box. So we create four columns, called `width`, `height`, `depth`, and `weight`, respectively. For the height, width, and depth, we will use millimeter as unit and for the weight we use grams. We can store values using the datatype `INT`, which is a shorthand for `INTEGER`. It can hold values from -2 147 483 648 to +2 147 483 647. This is more than enough for us, as it allows the dimensions of our boxes to range from 1mm to over 2000km and the weights to be up to 2000t. Each product must have values specified for all four dimensions, so we again mark each of them as `NOT NULL`.

The keen reader has noticed that we did not mention the oddest-looking part of the command in Figure 9.2.3. “`id INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY`”. What the heck is that?

Tables in a DB do not exist in an isolated manner. Instead, they will reference each other. For example, later we will want to store which customer bought which product. For this, we will need at least three tables: A table of customers (which we do not yet have), the table of products (that we are creating right now), and a table for the customer orders. Each record in that last table will reference one row in the customers table and one row in the products table.

For allowing the latter, we need a unique way to identify each row in our table. It must be possible to know exactly which product was ordered. There must not be any ambiguity.

On the plus side, our table already has a column with unique values, namely `name`. While these values are unique, they may not necessarily be stable. Maybe we now have a “Shoe, Size 36” but at some time, the marketing department wants to rebrand our products and change the name to “Super-Speed Sneaker, Size 36.” Then we maybe have many other records referencing the product under its old name which no longer exists. This could wreak havoc to our DB. Additionally, there also is the issue that the column `name` is type `VARCHAR`. The product names could be long strings. This would mean that each record in the orders table would also need to store the long `name` string to reference the product. This would probably be a waste of space [211].

It is better to use an automatically generated unique small value as key that will never change even if we change the name of a product [59].

We therefore add another column that we are going to call `id`, which should hold a unique integer value. We define it as `id INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY`. The `GENERATED BY DEFAULT AS IDENTITY` [175, 218] means that we do not need to specify values for this column when storing data. The system will automatically choose the next integer value that was not yet used.

We will use this column as the so called `PRIMARY KEY`, i.e., as the value that should be used by other tables to reference rows in our `product` table. The annotation with `PRIMARY KEY` also automatically enforces that only unique values can be stored.

Best Practice 9

Prefer using surrogate primary keys based on automatically incremented integers [59]. See also Definition 18.16.

Best Practice 10

Whenever using an automatically incremented integer as primary key for a table, name it `id`. While there is some controversy about this topic [213], anybody accessing your DB will immediately understand the meaning of the `id` columns and this practice is used in many sources [59, 338].

This completes the `SQL` command for creating our very first table. If we had typed it into the `psql` console, then it would succeed and print `CREATE TABLE` back to us, as shown in Figure 9.2.5.

If instead we had written the command into a script, then we would also fire up `psql` again. Since the DB belongs to the user `boss`, we now log in as `boss` using their password `superboss123`. We also want to work on the DB `factory`. The `PostgreSQL` connection `URI` for the DB `server` running on our current computer (identified by `localhost`) and therefore is `postgres://boss:superboss123@localhost/factory`. Assume that the script with the command was stored in a file called `create_table_product.sql`, then we would write `-e bf create_table_product.sql` after the connection `URI` parameter of `psql`. Listing 9.2 shows what happens if we execute the script Listing 9.1 this way.

Well, actually the script contains more commands: We also want to check whether the commands worked correctly. Before creating the new table, we thus print the list of tables owned by user `boss`. All of them are stored in table `pg_catalog.pg_tables` in the `PostgreSQL` server. We only print the table names, which are in column `tablename`. We only want to see those owned by `boss`, so we add the statement `WHERE tableowner = 'boss'`. The name of the user owning each table is stored in column `tableowner`, and only if it equals `boss`, we print the table name.

This results in the query `SELECT tablename FROM pg_catalog.pg_tables` with the clause `WHERE tableowner = 'boss'`; [386]. As you can see, before executing the `CREATE TABLE` command, this query returns nothing. Afterwards, one new table exists, namely `product`. This output is also illustrated in Figures 9.2.6 and 9.2.7.

Listing 9.1: Creating the table `product` to store the products we produce and sell. (stored in file `create_table_product.sql`; output in Listing 9.2)

```

1  /* We create the new table 'product' in our factory database. */
2
3  -- List all tables of the user 'boss' in database 'factory'
4  -- There are no tables yet.
5  SELECT tablename FROM pg_catalog.pg_tables
6    WHERE tableowner='boss';
7
8  -- The table 'product' stores all the produces that we can produce.
9  -- Each row of this table identifies one such product.
10 CREATE TABLE product (
11   id INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
12   name VARCHAR(100) NOT NULL UNIQUE, -- must exist, must be unique
13   price DECIMAL(10, 2) NOT NULL, -- price (RMB): 10 digits, 2 after .
14   weight INT NOT NULL, -- the weight of the product, in grams
15   width INT NOT NULL, -- the width of the product, in mm
16   height INT NOT NULL, -- the height of the product, in mm
17   depth INT NOT NULL -- the depth of the product, in mm
18 );
19
20 -- List all tables of the user 'boss' in database 'factory'
21 -- Now we see the table 'product'.
22 SELECT tablename FROM pg_catalog.pg_tables
23   WHERE tableowner='boss';

```

Listing 9.2: The stdout of the program `create_table_product.sql` given in Listing 9.1.

```

1 $ psql "postgres://boss:superboss123@localhost/factory" -v ON_ERROR_STOP=1
2   ↪ -ebs create_table_product.sql
3
4 tablename
5 -----
6 (0 rows)
7
8 CREATE TABLE
9   tablename
10 -----
11 product
12 (1 row)
13
14 # psql 16.12 succeeded with exit code 0.

```

9.1.2 Inserting some Data

Now the table `product` exists, but it is empty. Let us fill it with data. Our factory has two products: "Shoe" and "Purse." The shoes come in sizes 36 to 43. Their prices start at 150.99元 for size 36 and increase by 2元 per size. They all fit into the same box. The smallest shoes weight 1300g and the weight increases by 25g per size. Purses come in sizes *small*, *medium*, and *large*, at prices of 100元, 120元, and 150元, respectively. They weight 500g, 750g, and 1500g, respectively. The smallest purse fits into a shoebox, but the bigger ones require bigger boxes. In other words, we want to enter exactly the data presented in Figure 9.1 at the beginning of this section.

We store this data into the table `product` by an `INSERT INTO` statement. Here, we first need to provide the table name (`product`) and the attributes that we want to store in parentheses, i.e., "`(...)`". We will store values for the fields `name`, `price`, `weight`, `width`, `height`, and `depth`. We do not need to store values for `id`, because they will be automatically generated for us. After saying what we want to store, we specify the `VALUES` to store. Each row is written in parentheses, values and rows are separated by commas. The command follows the syntax given below.

```

1 -- The SQL syntax for inserting / appending data into a table.
2 --
3 -- tableName: the name of the table
4 -- columnI: the name of column I, as used when the
5 -- table was created
6 -- value for rowJ columnK: the value for column K of the the J-th
7 -- inserted row
8 --
9 -- Notice: Columns whose values are automatically generated as well as
10 -- columns that are *not* marked as 'NOT NULL' do not need to
11 -- be provided. They then are omitted from both the column name
12 -- list and the values lists for the inserted the rows.
13 INSERT INTO tableName (column1, column2, ...)
14 VALUES (value for row1column1, value for row1column2, ...),
15     (value for row2column1, value for row2column2, ...),
16     (value for row3column1, value for row3column2, ...),
17     ...
18     (value for rowNcolumn1, value for rowNcolumn2, ...);

```

The complete command for storing all the data is shown in Listing 9.3. There, we first print all the data currently in the table by typing `SELECT * FROM product;` [386]. This prints nothing, because the table is empty. Then we insert the eleven products via one single `INSERT INTO` command. Afterwards, we try `SELECT * FROM product;` again – and now it prints 11 rows.

We can invoke `INSERT INTO`-commands arbitrarily often. For example, instead of inserting all eleven rows at once, as we did above, we could have issued eleven separate `INSERT INTO` statements inserting one row each. And of course we can also call `INSERT INTO` later on to append more rows. Since this is easy and works in an obvious way, we will not explicitly try this out here.

Before we continue, though, let us briefly check what the `UNIQUE` constraint that we have defined on column `name` does. We wrote `name VARCHAR(100) NOT NULL UNIQUE`, meaning that column `name` should be a variable-length text string of no more than 100 characters, that it must always be specified, and that it is `UNIQUE`. Basically, the `UNIQUE` here means that there cannot be two records in our table with the same value of `name`. Therefore, if we would try to insert another product with name `'Shoe, Size 36'` into the table, this should fail. Because we already have a row with this value. We test this in Listing 9.5. Indeed, Listing 9.6 shows that this fails with an error and the data remains unchanged. There would have no good way to prevent such errors in Microsoft Excel or LibreOffice Calc, but in a DBMS, we can protect the integrity of our data.

Listing 9.3: Storing some products in the table `product`. (stored in file `insert_into_table_product.sql`; output in Listing 9.4)

```

1 /* Store some data into the table 'product'. */
2
3 -- Print all the contents from table 'product': Nothing.
4 SELECT * FROM product;
5
6 -- Insert 11 products into our table.
7 INSERT INTO product (name, price, weight, width, height, depth)
8 VALUES ('Shoe, Size 36', 150.99, 1300, 350, 250, 130),
9       ('Shoe, Size 37', 152.99, 1325, 350, 250, 130),
10      ('Shoe, Size 38', 154.99, 1350, 350, 250, 130),
11      ('Shoe, Size 39', 156.99, 1375, 350, 250, 130),
12      ('Shoe, Size 40', 158.99, 1400, 350, 250, 130),
13      ('Shoe, Size 41', 160.99, 1425, 350, 250, 130),
14      ('Shoe, Size 42', 162.99, 1450, 350, 250, 130),
15      ('Shoe, Size 43', 164.99, 1475, 350, 250, 130),
16      ('Small Purse', 100, 500, 350, 250, 130),
17      ('Medium Purse', 120, 750, 400, 300, 200),
18      ('Large Purse', 150, 1500, 600, 300, 250);
19
20 -- Now there are 11 rows.
21 SELECT * FROM product;

```

Listing 9.4: The stdout of the program `insert_into_table_product.sql` given in Listing 9.3.

```

1 $ psql "postgres://boss:superboss123@localhost/factory" -v ON_ERROR_STOP=1
2   ↵ -ebs insert_into_table_product.sql
3 id | name | price | weight | width | height | depth
4 -----+-----+-----+-----+-----+-----+
5 (0 rows)
6
7 INSERT 0 11
8 id | name | price | weight | width | height | depth
9 -----+-----+-----+-----+-----+-----+
10 1 | Shoe, Size 36 | 150.99 | 1300 | 350 | 250 | 130
11 2 | Shoe, Size 37 | 152.99 | 1325 | 350 | 250 | 130
12 3 | Shoe, Size 38 | 154.99 | 1350 | 350 | 250 | 130
13 4 | Shoe, Size 39 | 156.99 | 1375 | 350 | 250 | 130
14 5 | Shoe, Size 40 | 158.99 | 1400 | 350 | 250 | 130
15 6 | Shoe, Size 41 | 160.99 | 1425 | 350 | 250 | 130
16 7 | Shoe, Size 42 | 162.99 | 1450 | 350 | 250 | 130
17 8 | Shoe, Size 43 | 164.99 | 1475 | 350 | 250 | 130
18 9 | Small Purse | 100.00 | 500 | 350 | 250 | 130
19 10 | Medium Purse | 120.00 | 750 | 400 | 300 | 200
20 11 | Large Purse | 150.00 | 1500 | 600 | 300 | 250
21 (11 rows)
22 # psql 16.12 succeeded with exit code 0.

```

Listing 9.5: Showing how the `UNIQUE` constraint on column `name` prevents us from inserting a product with the same name as an already existing one into table `product`. (stored in file `insert_into_table_product_error.sql`; output in Listing 9.6)

```
1 /* Show how the UNIQUE constraint protects our table 'product' */
2
3 INSERT INTO product (name, price, weight, width, height, depth)
4 VALUES ('Shoe, Size 36', 151.99, 1300, 350, 250, 130);
```

Listing 9.6: The stdout of the program `insert_into_table_product_error.sql` given in Listing 9.5.

```
1 $ psql "postgres://boss:superboss123@localhost/factory" -v ON_ERROR_STOP=1
2   ↪ -eef insert_into_table_product_error.sql
3 psql:factory/insert_into_table_product_error.sql:4: ERROR:  duplicate key
4   ↪ value violates unique constraint "product_name_key"
5 DETAIL:  Key (name)=(Shoe, Size 36) already exists.
6 psql:factory/insert_into_table_product_error.sql:4: STATEMENT:  /* Show how
7   ↪ the UNIQUE constraint protects our table 'product' */
8 INSERT INTO product (name, price, weight, width, height, depth)
9 VALUES ('Shoe, Size 36', 151.99, 1300, 350, 250, 130);
10 # psql 16.12 failed with exit code 3.
```

9.1.3 Selecting Data

Now we have stored data in the table `product`. But how can we get it out again? Well, you already learned a good part of how to do that: `SELECT * FROM product;`. This query lists basically all of the data in the table. You have seen its output at the bottom of Listing 9.4.

Yet, most often, we do not want to retrieve *all* of the data in a table. Usually, we only want some part of the data. Maybe we only want to see the rows (records) that match certain criteria. Maybe we only want to see a subset of the columns. Maybe we even want to compute some statistics. How can we do that? A large part of the answer is “With the `SELECT` statement.”

This is a seduce-to-use example, something to play around with. So we will play around with the data for a bit in Listing 9.7.

First, let's say that we want a list of the names and prices of all types of purses that we sell. Let us amend the original query `SELECT * FROM product;` for this purpose. The `*` here means that all columns should be printed. Naturally, we would replace it with the columns that we want, namely, `name` and `price`. We write `SELECT name, price FROM product;`. This gives us the names and prices of *all* products in our table. We need to narrow this down to purses. We can add a `WHERE` clause at the end of the query where we can supply a condition. Only the records that match the condition will be shown. What condition can we use? SQL offers us some *pattern matching methods* [323]. The pattern `LIKE '%Purse%'` will match any string that contains the text “Purse”. The condition `name LIKE '%Purse%'` therefore requires that the product name contains the text “Purse”. Our first real query thus becomes `SELECT name, price FROM product WHERE name LIKE '%Purse%';`. As you can see in Listing 9.8, it will return three rows of purse-related data.

Assume now that you are a lady who wants to purchase some fashion accessory to accentuate your beauty. Naturally, you would want to buy the product that gives you the best deal in terms of product weight per monetary unit, i.e., g per 元. Therefore, for each product, we would like to divide the weight by the price and give this new value the name `g_per_yuan`. Luckily, SQL supports mathematical expressions [284], so it is possible to write `weight / price`. The query `SELECT name, weight / price AS g_per_yuan FROM product;` would return the product name and the weight-cost ration. `weight / price AS g_per_yuan` means that the ratio of weight and price will be computed and given the name `g_per_yuan`.

Suppose that our table `product` contains hundreds of entries. It would be very hard to spot good deals in that heap of data. Luckily, SQL also allows to sort data. We would like to see our list sorted based on `g_per_yuan` from large to small values. This way, the best deals will come first. We can do that by simply adding `ORDER BY g_per_yuan DESC`. `ORDER BY` sorts the rows of the query result by the fields listed afterwards. If just write `ORDER BY` or, optionally, add `ASC`, then it sorts the data in ascending order. This means that small values coming first. We want a descending order, so we also write `DESC`.

Finally, we may realize that there are still way to many entries returned. We only care about the best five or so deals, the rest does not matter anyway. All we have to do to limit the number of rows returned to five is to, well, add `LIMIT 5` to our query. With this, the query is completed. It is the second one in Listing 9.7.

Its result in Listing 9.8 shows us that the large purse is definitely the best deal here. For every single 元, we can get 10g of product! Indeed, the purse weights 1.5kg and costs 150元, so the result is correct. The second-best deal would be the largest shoe in stock. At size 43, we can 8.94g of product per 元.

What else can we find out about the data in this table? What if we wanted to know whether shoes or purses costed more on average? First, let's figure out how to compute arithmetic means in SQL. It is rather easy. We could write `SELECT AVG(price) FROM product;` to get the arithmetic mean over all values in the column `price` in the table `product` [4]. This query would return a single row with a single value named `avg`. That value would be `148.53818181818182` if you want to try it by yourself.

Let's give the value a better name. Let's try `SELECT AVG(price) AS mean_price FROM product;` The result would still be pretty much the same, but now the returned column is named `mean_price` (and there still only a single row).

As the next step, let's compute the average price for purses. We can reuse the condition `WHERE name LIKE '%Purse%'` from before and write `SELECT AVG(price) AS mean_price FROM product WHERE name LIKE '%Purse%';` This returns a single

Listing 9.7: Selecting information from the table `product`. (stored in file `select_from_table_product.sql`; output in Listing 9.8)

```

1  /* Extract information from the table product. */
2
3  -- List the names and prices of all purses.
4  SELECT name, price FROM product WHERE name LIKE '%Purse%';
5
6  -- Get the top-five products in terms of grams per yuan.
7  SELECT name, weight / price AS g_per_yuan FROM product
8    ORDER BY g_per_yuan DESC LIMIT 5;
9
10 -- Get the average price per product type.
11 SELECT 'Shoe' AS kind, AVG(price) AS mean_price
12   FROM product WHERE name LIKE '%Shoe%'
13 UNION ALL SELECT 'Purse' AS kind, AVG(price) AS mean_price
14   FROM product WHERE name LIKE '%Purse%';

```

Listing 9.8: The stdout of the program `select_from_table_product.sql` given in Listing 9.7.

```

1  $ psql "postgres://boss:superboss123@localhost/factory" -v ON_ERROR_STOP=1
2    ↪ -efb select_from_table_product.sql
3
4      name      | price
5  -----
6  Small Purse | 100.00
7  Medium Purse | 120.00
8  Large Purse | 150.00
9  (3 rows)
10
11     name      | g_per_yuan
12  -----
13  Large Purse | 10.000000000000000000000000000000
14  Shoe, Size 43 | 8.9399357536820413
15  Shoe, Size 42 | 8.8962513037609669
16  Shoe, Size 41 | 8.8514814584756817
17  Shoe, Size 40 | 8.8055852569343984
18  (5 rows)
19
20     kind      | mean_price
21  -----
22  Shoe | 157.990000000000000000000000000000
23  Purse | 123.3333333333333333333333333333
24  (2 rows)
25
26 # psql 16.12 succeeded with exit code 0.

```

column named `mean_price` and a single row. The value in that row is now `123.33333333333333`, which indeed is the arithmetic mean of 100, 120, and 150. To make clear that this is the purse price, we can simply add an artificial column named `kind` with value `'Purse'`. `SELECT 'Purse' AS kind, AVG(price) AS mean_price FROM product WHERE name LIKE '%Purse%'`. Now we got two columns, `kind` and `mean_price`, and one row with the values `Purse` and `123.33333333333333`.

We can, of course, do the same for shoes. All we have to do is to replace `'%Purse%'` with `'%Shoe%'` and change the `kind` column accordingly. `SELECT 'Shoe' AS kind, AVG(price) AS mean_price FROM product WHERE name LIKE '%Shoe%'` returns also a single row with values `Shoe` and `157.990000000000000000000000000000`. Indeed, the average price of all of our shoes is 157.99元.

So we have two queries that each return two values with the same names. At this point, we already know that purses are cheaper than shoes in average. But having two queries is somehow unsatisfying.

We want to package both results together, so that we get the two rows as the result of a single **SQL** command.

Nothing easier than that! We just have to remove the trailing `;` from the first query and write a `UNION ALL` directly in front of the second query [96]! The `UNION ALL` statement effectively appends the results of the second query to the results of the first query. The combined statement is shown at the bottom of Listing 9.7 and its result is given in Listing 9.8.

At this point, please notice the beauty of queries: We can continue to add data to our table. We can change values or delete values. But the queries will still work all the same and always give us the up-to-date results.

9.2 The Table “customer”

Next, we design a table for managing customer data. The kind of customer data that we want to play with here is mostly textual. This gives us a chance to play with some more advanced text-related features of **DBs**. At the same time, we will learn more methods to constrain the valid ranges for the data in columns.

9.2.1 Creating the Table

We will fittingly call the table for customers `customer`. The structure of the table could look a bit like Figure 9.3 (if it was a Microsoft Excel or LibreOffice Calc spreadsheet): For the minimalistic example here, it will be sufficient to store the name, mobile phone number, and address of the customers.

We create the table `customer` using the `CREATE TABLE` command in Listing 9.9. The first column again should be the primary key `id`, which we again define as `INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY`.

Customers have names, so we again need a `name` column. Names can have different length, but 100 characters seems to be a reasonable limit. Therefore, we will again use `VARCHAR(100)`. For each customer, a name must be specified, so we again also write `NOT NULL`. Do names need to be unique? No they don't. There can easily be two different customers with the same name. Therefore, we will *not* require this column to be `UNIQUE`.

However, when thinking about names, we realize that `NOT NULL` is not really a good bottom line for valid names. If we enforce that names must always be entered ... is that enough to ensure that names are *correct*? Actually, if we go back to our previous table `product`, we find that we could easily enter a product with the name '`bla bla`', '', or even an empty name ''. We just demanded that the name be set, not that it cannot be set to a text string of length 0.

What does *correct* actually mean? If we enter “Bebbo” as customer name, clearly a **DBMS** cannot ensure that the real name of the person we mean is actually “Bebbo.” Maybe that “Bebbo” person lied to our customer representative and intentionally gave a wrong name. Maybe there was a misunderstanding and their name is actually “Beb-Bó.” No DBMS can guard against this. But there are other sources of errors: Maybe the person who types the name into our DBMS made an error! Maybe they typed “Bebbo”, i.e., accidentally hit space (the key) when beginning to write. Such a tiny error could be invisible but could cause all sorts of issues down the line. Or maybe they hit `⌫` when entering the name, leaving an empty string in the form field. These are the kind of errors that we want to prevent. And we try to do this by specifying validity constraints at all levels of our application.

So what would be a good bottom line for valid names? Well, it should probably start with a “word character”, say, “A”, “b”, or even “张” and also end with one. Inbetween, we would allow arbitrary

<code>id</code>	<code>name</code>	<code>phone</code>	<code>address</code>
1	Bibbo	999999999999	Hefei, Jinxiu Dadao 99, China
2	Bebbo	555555555555	Rathaus, Chemnitz, Germany
3	Bebba	333333333333	Times Square, NY, USA
4	Bobbo	444444444444	Eiffel Tower, Paris, France
...

Figure 9.3: How our table `customer` could look like if we designed it in a spreadsheet software like Microsoft Excel or LibreOffice Calc.

Listing 9.9: Creating the table `customer` to store the information about our factory's customers. (stored in file `create_table_customer.sql`; output in Listing 9.10)

```

1  /* We create the new table 'customer' in our factory database. */
2
3  -- The table 'customer' stores all the customers that we have.
4  -- Each row of this table identifies one such a customer.
5  CREATE TABLE customer (
6      id      INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
7      name    VARCHAR(100) NOT NULL,           -- must exist
8      phone   VARCHAR(11)  NOT NULL UNIQUE,   -- the phone number
9      address VARCHAR(255) NOT NULL,          -- the address
10     CONSTRAINT customer_name_ok CHECK (name ~ '^\w.*\w$'),
11     CONSTRAINT customer_phone_ok CHECK (phone ~ '^\d{10,11}$')
12 );
13
14 -- List all tables of the user 'boss' in database 'customer'
15 -- Now we see the table 'customer'.
16 SELECT tablename FROM pg_catalog.pg_tables
17   WHERE tableowner='boss';

```

Listing 9.10: The stdout of the program `create_table_customer.sql` given in Listing 9.9.

```

1  $ psql "postgres://boss:superboss123@localhost/factory" -v ON_ERROR_STOP=1
2  ↪ -ebsf create_table_customer.sql
3  CREATE TABLE
4  tablename
5  -----
6  product
7  customer
8  (2 rows)
9  # psql 16.12 succeeded with exit code 0.

```

characters. This does not prevent anybody from entering “sgjw9345 s熊貓fki345Q” as name, but at least we would prevent the user from accidentally entering leading or trailing space characters or entering an empty name. How can we accomplish that?

SQL offers the mechanism of *constraints* [100]. Actually, `NOT NULL` and `UNIQUE` are short-hands for two constraints. But we can also define more fancy ones. Constraints can be written directly after declaring the columns in the `CREATE TABLE` command. The syntax for this is `CONSTRAINT constraint_name CHECK (expression)`. In other words, we give the constraint a name and provide an expression that should be checked whenever data is entered or changed in the table. Only if the expression is `TRUE` the insertion or change is permitted.

```

1  -- Create a New Table with additional Constraints.
2  --
3  -- tableName: the name of the table to create
4  -- columnName: the name of the X-th column.
5  -- typeOfColumnX: the datatype to be used for the X-th column.
6  -- constraintsOnColumnX: simple constraints imposed on the X-th column,
7  -- can be omitted or a combination of UNIQUE,
8  -- NOT NULL, PRIMARY KEY, etc.
9  -- constraintNameY: the name of the Y-th (check) constraint
10 -- expressionY: the expression of the Y-th (check) constraint
11 --
12 -- Everytime a row is entered into the table or changed, the expressions
13 -- of all constraints are checked. If one of them is not TRUE, then the
14 -- operation is cancelled, not performed, and an error is reported.
15 CREATE TABLE tableName(
16   column1 typeOfColumn1 constraintsOnColumn1,
17   column2 typeOfColumn2 constraintsOnColumn2,
18   ...
19   CONSTRAINT constraintName1 CHECK expression1,
20   CONSTRAINT constraintName2 CHECK expression2,
21   ...
22 );

```

This means that we have to figure out how we can define “`name` must start and end with a ‘word character’ and can have arbitrary characters inbetween.” as such an expression. This is a bit beyond what we can do with `LIKE`. Luckily, **regular expressions (regexes)** come to the rescue. Regexes are text patterns that can be matched against text strings. They are supported by many tools and programming languages (such as [Python \[482\]](#)), and also by [SQL](#) and, hence, [PostgreSQL \[336\]](#). Regexes are a whole different kind of subject that you can read about in [336]. Here we will just briefly introduce and directly use them.

We will define the constraint that the values in the column `name` must match to the regex `'^\w.*\w$'`. The `^` matches the beginning of a text. `\w` stands for a single “word character”, `\w` thus means that the text string must start with a word character. `*` passt auf jede beliebige Zeichen. `*` means that the expression item directly before the `*` can occur any number of times, from zero to infinity. Hence, the `.*` following the `^\w` means that, after the word character at the beginning of the text, an arbitrary number of arbitrary characters may follow. Finally, `$` matches to the end of the text. Therefore, the `\w$` at the end of the regex means that the text that we compare the regex with must have a word character at its end for the regex to match.

In summary, by writing `'^\w.*\w$'`, we say: The beginning of the text, i.e., the very first character, must be a “word character”, e.g., “A”, “b”, or “李”. Then, there can be an arbitrary number of other characters, including spaces, numbers, signs, whatever. At the end of the text, we again expect a word character. This means that we demand that names consist of at least two word characters. We could refine this to also allow single-character names, to prevent characters such as “@” from occurring, etc., but let’s keep it at this for now.

Having a reasonable limitation for the name, we now define the rule `customer_name_ok` as `CONSTRAINT customer_name_ok CHECK (name ~ '^\w.*\w$')`. The `CONSTRAINT` marks the beginning of a constraint, `customer_name_ok` is the name, and `CHECK` tell us that we will next define an expression (as opposed to simply `NOT NULL`). `(name ~ '^\w.*\w$')` says that the field `name` must match the regex `'^\w.*\w$'`. Here, `~` stands for regex-based pattern matching. With this, the names “S” and “schwipschawp” are prohibited, but “Thomas Weise” and “熊猫先生” are OK.

Each customer also needs to have a phone number, so we add a column `phone`. While phone numbers may appear to be integer numbers, they could have leading zeros. Therefore, we will store them as text strings. In China, landlines have 10 digits and mobile numbers have 11 digits [508]. Therefore, we choose `VARCHAR(11)` as the datatype. The phone numbers must be `NOT NULL` and this time, we also insist on them to be `UNIQUE`. While there might be two different customers with the same name, we do not permit two different customers having the same phone number. Indeed, if our sales department wants to enter a customer’s information into the DB and there is already one record with the same phone number, then most likely this would be the same customer and they are in the process of creating a duplicate entry by accident.

Now we want to limit the phone number text to represent valid phone numbers. 'ABC', for example, is not a valid phone number and neither is 1. We want to only permit numbers consisting of ten to eleven digits. We can do this in exactly the same way in which we guarded our `name` field: by defining a constraint. We write `CONSTRAINT customer_phone_ok CHECK (phone ~ '^\\d{10,11}$')`. The name of the constraint will be `customer_phone_ok`. We again match a regex via `~`, but this time we match the value of `phone`.

The regex `'^\\d{10,11}$'` reads as follows: At the start of the text (denoted by `^`), there is a sequence of 10 to 11 digits, and then comes the end of the text (denoted by `$`). Here, `\\d` stands for a single character that is a digit, i.e., 0...9. The `{10,11}` specifies between 10 and 11 repetitions of this pattern. Therefore, our constraint requires that any value of `phone` to be stored consists of ten to eleven digits (and only digits). This rules out phone numbers containing other characters, as well as customers registering under number 110.

Of course, this still does not guarantee that the phone numbers that are entered are valid. On the one hand, we could have a much more complex expression that checks whether the ten-digit numbers have proper area codes as their first digits and that the eleven-digit numbers start with proper mobile phone provider prefixes. On the other hand, there still would be no way to verify, from within SQL at least, that the phone number actually exists. Then again, even if we could check that, we still would not know whether the phone number is registered under the customer's name. These things are far beyond the scope here. To keep the DB clean, the simple check defined here shall suffice.

Finally, each customer should have a shipping address. Here we will just settle for `VARCHAR(255)`. A length of at most 255 characters seems reasonable, as 255 is also used in many other systems as the limit. We require the `address` to be `NOT NULL`, but it does *not* have to be `UNIQUE`. For laziness sake I will not specify a `CONSTRAINT` that sanity-checks addresses ... maybe you could do this as a small exercise when reproducing the example?

Either way, the full SQL command for creating our table `customer` is given in Listing 9.9. Notice that we, again, use the `boss` user with password `superboss123` to execute these commands. We create the table and, afterwards, check whether a new table appeared in the `pg_catalog.pg_tables`. It does: the user `boss` now owns two tables, `product` and `customer`, as you can see in Listing 9.10.

9.2.2 Inserting some Data

We now enter the data of the four (imaginary) customers of our company. We can do this again with the `INSERT INTO` command. We first need to specify the table, which is `customer`, and then the columns, namely `name`, `phone`, and `address`. We do not need to provide values for the `id` column, because it will automatically be set. The customer names are Mr. Bibbo, Mr. Beppo, Mrs. Beppa, and Mr. Bobbo. Bibbo lives in the south campus of our Hefei University (合肥大学), Beppo lives in the town hall of Chemnitz city in Germany, Beppa lives on Times Square in New York, and Bobbo resides on top of the Eiffel Tower in Paris, France. Their phone numbers are, by sheer chance, all composed of eleven repetitions of a single digit. Either way, we can insert these values by specifying them row-for-row, using commas to separate rows. Each row is given in parentheses and the values are listed in the same sequence as we specified the columns and separated by commas as well.

Listing 9.11 and the corresponding `psql` output in Listing 9.12 show that, first, the table is empty. `SELECT * from customer;` yields 0 rows. Then we execute the `INSERT INTO` command. Afterwards `SELECT * from customer;` prints the four expected rows.

Of course, we also need to test whether our constraints work or not. In Listing 9.13, we attempt to store a faulty customer record. Can you spot the error before looking at the `psql` output in Listing 9.14?

Listing 9.11: Storing some customer records in the table `customer`. (stored in file `insert_into_table_customer.sql`; output in Listing 9.12)

```

1  /* Store some data into the table 'customer'. */
2
3  -- Print all the contents from table 'customer': Nothing.
4  SELECT * FROM customer;
5
6  -- Insert 4 customers into our table.
7  INSERT INTO customer (name, phone, address)
8  VALUES ('Bibbo', '99999999999', 'Hefei, Jinxiu Dadao 99, China'),
9    ('Bebbo', '55555555555', 'Rathaus, Chemnitz, Germany'),
10   ('Bebba', '33333333333', 'Times Square, NY, USA'),
11   ('Bobbo', '44444444444', 'Eiffel Tower, Paris, France');
12
13 -- Now there are 4 rows.
14 SELECT * FROM customer;

```

Listing 9.12: The stdout of the program `insert_into_table_customer.sql` given in Listing 9.11.

```

1  $ psql "postgres://boss:superboss123@localhost/factory" -v ON_ERROR_STOP=1
2      ↵ -ebs insert_into_table_customer.sql
3  id | name | phone | address
4  -----+-----+-----+
5  (0 rows)
6
7  INSERT 0 4
8  id | name | phone | address
9  -----+-----+-----+
10  1 | Bibbo | 99999999999 | Hefei, Jinxiu Dadao 99, China
11  2 | Bebbo | 55555555555 | Rathaus, Chemnitz, Germany
12  3 | Bebba | 33333333333 | Times Square, NY, USA
13  4 | Bobbo | 44444444444 | Eiffel Tower, Paris, France
14
15 # psql 16.12 succeeded with exit code 0.

```

Listing 9.13: Trying to store a faulty customer record. Can you spot the error? (stored in file `insert_into_table_customer_error.sql`; output in Listing 9.14)

```
1 /* Store some faulty data into the table 'customer'. */
2
3 -- Try to insert a faulty customer record. Can you spot the error?
4 INSERT INTO customer (name, phone, address)
5 VALUES ('Eugen', '88888B88888', 'Hochschule Osnabrück, Germany');
```

Listing 9.14: The stdout of the program `insert_into_table_customer_error.sql` given in Listing 9.13.

```
1 $ psql "postgres://boss:superboss123@localhost/factory" -v ON_ERROR_STOP=1
2   ↪ -eef insert_into_table_customer_error.sql
3 psql:factory/insert_into_table_customer_error.sql:5: ERROR:  new row for
4   ↪ relation "customer" violates check constraint "customer_phone_ok"
3 DETAIL:  Failing row contains (5, Eugen, 88888B88888, Hochschule Osnabrück,
4   ↪ Germany).
4 psql:factory/insert_into_table_customer_error.sql:5: STATEMENT:  /* Store
5   ↪ some faulty data into the table 'customer'. */
5 -- Try to insert a faulty customer record. Can you spot the error?
6 INSERT INTO customer (name, phone, address)
7 VALUES ('Eugen', '88888B88888', 'Hochschule Osnabrück, Germany');
8 # psql 16.12 failed with exit code 3.
```

9.2.3 Selecting Data

Let us complete this exercise by extracting some data from the new table `customer` in Listing 9.15. First, we want to have a list of all customers from France. This can be done with the query `SELECT name, address FROM customer WHERE address LIKE '%France%'`; [323]. The query prints the `name` and `address` of all customers whose `address` field contains, somewhere, the text `France`.

Next, we want to know how many domestic Chinese customers we have and how many customers purchase our products from abroad, i.e., not from China. We first need to decide whether a customer is based in China or not. We can do this by `address ILIKE '%china%'` as `domestic` [323]. Notice that we this time wrote `ILIKE` instead of `LIKE`. `ILIKE` works basically the same as `LIKE`, with the exception that it compares text case-insensitive. For `LIKE`, `China` and `china` are two different strings. For `ILIKE`, they are the same. By writing `SELECT address ILIKE '%china%'` as `domestic` `from customer`, we would get a one-column result which contains the value `TRUE` for every customer from China and `FALSE` for every customer from abroad. In a first step, we select the names of all customers together with the value for our synthetic attribute `domestic`. We find that the only domestic customer is Mr. Bibbo. Only he has `t`, which means `TRUE`, as value of `domestic`. All other customers have `f`, which means `FALSE`, as value of `domestic`.

We just want to know the numbers of domestic and non-domestic, i.e., foreign, customers. To get these numbers, we can divide the customers into groups based on this col-

Listing 9.15: Obtaining information from our new table `customer`. (stored in file `select_from_table_customer.sql`; output in Listing 9.16)

```

1  /* Extract information from the table customer. */
2
3  -- Try to find customers who may be living in France.
4  SELECT name, address FROM customer WHERE address LIKE '%France%';
5
6  -- Which customers are domestic, i.e., live in China?
7  SELECT name, address ILIKE '%china%' as domestic FROM customer;
8
9  -- Count how many domestic and foreign customers we have.
10 SELECT COUNT(*), address ILIKE '%china%' as domestic FROM customer
11   GROUP BY domestic;
```

Listing 9.16: The stdout of the program `select_from_table_customer.sql` given in Listing 9.15.

```

1  $ psql "postgres://boss:superboss123@localhost/factory" -v ON_ERROR_STOP=1
2    ↪ -ebsf select_from_table_customer.sql
3
4  name | address
5  -----+
6  Bobbo | Eiffel Tower, Paris, France
7  (1 row)
8
9  name | domestic
10 -----+
11 Bibbo | t
12 Beppo | f
13 Beppa | f
14 Bobbo | f
15 (4 rows)
16
17 count | domestic
18 -----+
19     3 | f
20     1 | t
21 (2 rows)
22
23 # psql 16.12 succeeded with exit code 0.
```

umn `domestic` by adding `GROUP BY domestic` to the query. We count the number of customers in each group by also selecting the new value `COUNT(*)`. `COUNT` is another aggregate statistic function, just like `AVG`, which we already used before. `COUNT(*)` counts the rows in the current group and returns the result as integer number. The complete query is `SELECT COUNT(*), address ILIKE '%china%' as domestic FROM customer GROUP BY domestic;`.

It produces two rows. The first row has `domestic` as `f`, which, again, means `FALSE`. In its `count` column, we see the value 3. The second row has `domestic` as `t`, which stands for `TRUE`. In its `count` column, we see the value 1. Indeed, there are three foreign customers and only one domestic one in our DB.

If you looked at this example carefully, then you noticed that the method of deciding whether a customer is from China or not, as well as the method of detecting french customers, are not very precise. For example, if a customer would have specified their country as 中国, i.e., China written in Chinese, we would have considered them a foreigner. Then again, if the director of the imaginary *Donut Factory Vive la France* in Shanghai, China would order shoes from us, we would consider his address to be french and domestic at the same time.¹ Maybe we should have had another column `country`? This would at least solve the problem with the donut factory. We would still have the problem that people could use different spellings for the same country, say China, china, 中国, People's Republic of China, PRC, P.R.C., 中华人民共和国, République populaire de Chine, Chine, and so on, though. To solve this problem, we would most likely need a table with countries and link that table to our `customer` table... Next, we will explore how tables can be “linked” together.

9.3 The Table “demand”

We now have two tables. In the first table, we have the products that we can sell. In the second table, we have a list of customers. Now we want to store the actual orders, the sales of our company. We here only consider a very simplistic approach to order management: With each issued order, a customer can buy a certain amount of exactly one product. From the real world, you know that usually you can order multiple products in each purchase in an online shop, decide for a payment option, and maybe specify a shipping address different from the address associated with your account. However, we here keep it plain and simple.

9.3.1 Creating the Table

Naturally, we would like to call this table, which stores orders, something like `order` (as we always use singular table names as stated in [Best Practice 5](#)). Unfortunately, we already learned that `ORDER` is a

¹This situation arises because we here violate the [first normal form \(1NF\)](#), which we will discuss much much later, in [Section 19.3.1](#). Indeed, dividing the `address` column into multiple columns would be part of the solution.

<code>id</code>	<code>customer</code>	<code>product</code>	<code>amount</code>	<code>ordered</code>
1	1 (Bibbo)	7 (Shoe, Size 42)	12	2024-11-21
2	2 (Bebbo)	3 (Shoe, Size 38)	2	2024-12-09
3	3 (Bebba)	2 (Shoe, Size 37)	7	2024-12-16
4	2 (Bebbo)	5 (Shoe, Size 40)	7	2024-12-30
5	1 (Bibbo)	5 (Shoe, Size 40)	3	2025-01-05
6	2 (Bebbo)	6 (Shoe, Size 41)	4	2025-01-12
7	3 (Bebba)	11 (Large Purse)	10	2025-01-16
8	2 (Bebbo)	3 (Shoe, Size 38)	6	2025-02-05
...

Figure 9.4: How our table `demand` could look like if we designed it in a spreadsheet software like Microsoft Excel or LibreOffice Calc. Since this spreadsheet would need to link its data to two other tables (for customers and products), actually implementing our factory management with a spreadsheet would become quite hard at this point. (Not impossible, but very error prone and requiring serious Excel Fu.)

Listing 9.17: Creating the table `demand` to store the orders of our customers. (stored in file `create_table_demand.sql`; output in Listing 9.18)

```

1  /* We create the new table 'demand' in our factory database. */
2
3  -- The table 'demand' stores all the customer orders.
4  CREATE TABLE demand (
5      id      INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
6      customer  INT NOT NULL REFERENCES customer(id),
7      product   INT NOT NULL REFERENCES product(id),
8      amount    INT NOT NULL,
9      ordered   DATE NOT NULL,
10     CONSTRAINT ordered_amount_ok CHECK (
11         (amount > 0) AND (amount < 1000000)),
12     CONSTRAINT ordered_date_ok CHECK (
13         (ordered > '2024-10-01') AND (ordered < '3000-01-01'))
14 );
15
16 -- List all tables of the user 'boss' in database 'factory'
17 -- Now we see the table 'demand'.
18 SELECT tablename FROM pg_catalog.pg_tables
19   WHERE tableowner='boss';

```

Listing 9.18: The stdout of the program `create_table_demand.sql` given in Listing 9.17.

```

1  $ psql "postgres://boss:superboss123@localhost/factory" -v ON_ERROR_STOP=1
2  ↪ -e bf create_table_demand.sql
3  CREATE TABLE
4  tablename
5  -----
6  product
7  customer
8  demand
9  (3 rows)
10 # psql 16.12 succeeded with exit code 0.

```

reserved keyword in SQL.

Best Practice 11

Never use **SQL** keywords or reserved words as names, e.g., for columns or tables [59].

OK, fine, so we go with a synonym and call the table `demand` in Listing 9.17. The structure of the table is a bit different from what we had before. The structural sketch in Figure 9.4 indeed looks a bit odd.

But let us start slowly. Like in the other two tables, each demand record must have a unique primary key `id`. This should again be an integer number which is automatically generated by the DB for us. Therefore, we define the column `id INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY`, in the same way we already did before.

Now, however, comes something really cool: Only customers can make orders. Therefore, every record in the table `demand` must be linked to exactly one record in the table `customer`. How can we do that? Via so-called *foreign keys* [167]. You see, in both of our existing tables, we have defined a `PRIMARY KEY`. We used auto-generated integers, as we also do in the new table. We can now define a column which also is of type `INT` and that references the primary key of another table.²

We can write `customer INT NOT NULL REFERENCES customer(id)` to define such a reference. The name of this new column in our table `demand` will be `customer`. We declare it as an `INT` which must be

²Much later, we will formalize the concept of foreign keys in Definition 19.5.

`NOT NULL`. Now we write `REFERENCES customer(id)`, which basically is a constraint. This constraint is called *foreign key*. For any value of `customer` in our new table, makes sure that there *always* is a row in table `customer` whose `id` value is the same. You cannot add a row to our table `demand` if the `customer` value does not match to an `id` of one row in table `customer`. You also cannot delete a record from `customer` if its `id` is used somewhere in table `demand`. This means every single record in our table `demand` will definitely be linked to one record in table `customer`. Of course, a record in table `customer` can be linked to many records in table `demand`.

Now, the customer makes a purchase. So we want to link this purchase also to a product. For the sake of simplicity, we only allow the customer to purchase one product at a time. Otherwise we need yet another table ... and this example will get too exhausting. So after linking the demand records to customer records, we also need to link them to product records. We therefore add another column that we will call `product` by writing `product INT NOT NULL REFERENCES product(id)`. As you can see, we again mark this column as foreign key by specifying a `REFERENCES` constraint. This time, it references the column `id` of table `product`. In other words, every single record that we will put into the table `demand` will be linked to exactly one record in table `customer` and to exactly one record in table `product`.

So the customer has ordered one product. Next we want to specify the amount of the product that the customer orders. We create the column `amount INT NOT NULL`. For each demand, we must specify an amount (`NOT NULL`) and that amount must be an integer, hence the `INT`. Having learned about constraints a while ago, we want to protect our data a bit better. For example, we want to make sure that `amount` is always positive, i.e., greater than zero. Also, orders for over one million units of any product are unrealistic. If we are about to insert a record into our `demand` table where someone orders a million shoes, chances are that something went wrong. So we want to define a constraint enforcing `amount` to stay in 1..999 999.

We can write `CONSTRAINT ordered_amount_ok CHECK (amount > 0)`. This will create the constraint `ordered_amount_ok` which will check that `amount` is greater than zero. We can also write `CONSTRAINT ordered_amount_ok CHECK (amount < 1000000)`. This would instead make sure that the ordered amount is less than one million. We can combine both conditions into one and simply write `CONSTRAINT ordered_amount_ok CHECK ((amount > 0) AND (amount < 1000000))`. Indeed, SQL supports logical operators such as `AND`, `OR`, and `NOT`. With this, we prevent any order for less than one or more than 999 999 items.

Of course, we also want to store *when* a customer issued the demand. For storing dates, SQL offers the `DATE` type. It allows us to specify dates in our (Gregorian) calendar [204, 335]. As notation, the ISO standard format “`YYYY-MM-DD`” [119] is used, where `YYYY` is the four-digits of the year, `MM` stands for the number month in two digits, and `DD` is the day specified with two digits as well. We then can compare dates and do all sorts of arithmetics with them. Like the type `DECIMAL` being the canonical datatype to be used for monetary things, `DATE` is the right type for dates. It is also a reserved word, so we cannot call our new column `DATE` and we also cannot call it `WHEN`, as this is also reserved [414]. Lets use `ordered` as name for the column storing date when the customer ordered our product. We write `ordered DATE NOT NULL` as column definition, because we want to enforce that it is `NOT NULL`, i.e., the order date must always be specified.

Let us also insert a sanity check constraint to make sure that dates make sense. Assume that we built our database in October 2024, then no order with a date before `2024-10-01` can exist. Furthermore, it would be unlikely that our software was still running in a thousand years, so let's also not except any date greater than or equal to `3000-01-01`. We write `CONSTRAINT ordered_date_ok CHECK ((ordered > '2024-10-01') AND (ordered < '3000-01-01'))` to combine both constraints. As you see, arithmetic comparisons have been implemented for the `DATE` datatype.

The table is created using the completed command in Listing 9.17. In Listing 9.18 we see the result – there is now a new table `demand` in our DB.

9.3.2 Inserting and Selecting some Data

Inserting data into our new table `demand`, however, is a bit annoying. We need to refer to the customers and the products by their `id`. In Listing 9.19, we use an `INSERT INTO` command to insert eight orders into the table. We need to specify the value of `customer`, `product`, `amount`, and the `ordered` date for each of them. The first row is `(1, 7, 12, '2024-11-21')`. It means that customer 1, namely

Listing 9.19: Storing some order records in the table `demand`. (stored in file `insert_into_table_demand.sql`; output in Listing 9.20)

```

1  /* Store some data into the table 'demand'. */
2
3  -- Print all the contents from table 'demand': Nothing.
4  SELECT * FROM demand;
5
6  -- Insert 8 orders into our table.
7  INSERT INTO demand (customer, product, amount, ordered)
8  VALUES (1, 7, 12, '2024-11-21'), (2, 3, 2, '2024-12-09'),
9      (3, 2, 7, '2024-12-16'), (2, 5, 7, '2024-12-30'),
10     (1, 5, 3, '2025-01-05'), (2, 6, 4, '2025-01-12'),
11     (3, 11, 10, '2025-01-16'), (2, 3, 6, '2025-02-05');
12
13  -- Now there are 8 rows.
14  SELECT * FROM demand;
```

Listing 9.20: The stdout of the program `insert_into_table_demand.sql` given in Listing 9.19.

```

1  $ psql "postgres://boss:superboss123@localhost/factory" -v ON_ERROR_STOP=1
2      ↪ -efb insert_into_table_demand.sql
3  id | customer | product | amount | ordered
4  ----+-----+-----+-----+
5  (0 rows)
6
6  INSERT 0 8
7  id | customer | product | amount | ordered
8  ----+-----+-----+-----+
9  1 | 1 | 7 | 12 | 2024-11-21
10 2 | 2 | 3 | 2 | 2024-12-09
11 3 | 3 | 2 | 7 | 2024-12-16
12 4 | 2 | 5 | 7 | 2024-12-30
13 5 | 1 | 5 | 3 | 2025-01-05
14 6 | 2 | 6 | 4 | 2025-01-12
15 7 | 3 | 11 | 10 | 2025-01-16
16 8 | 2 | 3 | 6 | 2025-02-05
17 (8 rows)
18
19 # psql 16.12 succeeded with exit code 0.
```

Mr. Bibbo orders twelve units of product 7, i.e., “Shoe, Size 42.” He did this on November 21, 2024. The second row is `(2, 3, 2, '2024-12-09')`, meaning that customer 2, i.e., Mr. Beppo, ordered 2 units of product 3, namely “Shoe, Size 38.” This happened on December 9, 2024. We insert several rows like this. For instance, row `(3, 11, 10, '2025-01-16')` identifies Mrs. Bebba (customer 3) as the purchaser of ten large purses (product 11) on January 16, 2025.

[Listing 9.19](#) shows the effect of this command. Querying `SELECT * FROM demand;` will show us all the rows of this table. Making sense of this data is, however, not straightforward. Luckily, **SQL** does not just offer us the ability to link records between tables in order to maintain data integrity: It also provides us means to connect the information from different tables together.

Before we explore pulling data from different tables together, let us briefly check whether our **DBMS** really protects the integrity of our data. In [Listing 9.21](#), we attempt to insert an order record with the invalid product `id` 77 into the `demand` Table. Maybe the data entry clerk wanted to enter an order for product 7, but accidentally hit the `7` key twice. The SQL query therefore fails with an error in [Listing 9.24](#).

A similar mistake happens in [Listing 9.23](#): Here, the order data was accidentally typed out as `'1024-11-21'` instead of `'2024-11-21'`. This violates our constraint `ordered_date_ok` and, hence, fails in [Listing 9.24](#).

Listing 9.21: The attempt to store an order with an invalid product `id` fails. (stored in file `insert_into_table_demand_error_1.sql`; output in Listing 9.22)

```

1 /* Store some incorrect data into the table 'demand'. */
2
3 -- Trying to insert an order with an invalid product ID.
4 INSERT INTO demand (customer, product, amount, ordered)
5 VALUES (1, 77, 12, '2024-11-21');
```

Listing 9.22: The stdout of the program `insert_into_table_demand_error_1.sql` given in Listing 9.21.

```

1 $ psql "postgres://boss:superboss123@localhost/factory" -v ON_ERROR_STOP=1
   ↪ -eef insert_into_table_demand_error_1.sql
2 psql:factory/insert_into_table_demand_error_1.sql:5: ERROR:  insert or
   ↪ update on table "demand" violates foreign key constraint "
   ↪ demand_product_fkey"
3 DETAIL:  Key (product)=(77) is not present in table "product".
4 psql:factory/insert_into_table_demand_error_1.sql:5: STATEMENT:  /* Store
   ↪ some incorrect data into the table 'demand'. */
5 -- Trying to insert an order with an invalid product ID.
6 INSERT INTO demand (customer, product, amount, ordered)
7 VALUES (1, 77, 12, '2024-11-21');
8 # psql 16.12 failed with exit code 3.
```

Listing 9.23: The attempt to store an order with typo in the order date fails as well. (stored in file `insert_into_table_demand_error_2.sql`; output in Listing 9.24)

```

1 /* Store some incorrect data into the table 'demand'. */
2
3 -- Trying to insert an order with an invalid date.
4 INSERT INTO demand (customer, product, amount, ordered)
5 VALUES (1, 7, 12, '1024-11-21');
```

Listing 9.24: The stdout of the program `insert_into_table_demand_error_2.sql` given in Listing 9.23.

```

1 $ psql "postgres://boss:superboss123@localhost/factory" -v ON_ERROR_STOP=1
   ↪ -eef insert_into_table_demand_error_2.sql
2 psql:factory/insert_into_table_demand_error_2.sql:5: ERROR:  new row for
   ↪ relation "demand" violates check constraint "ordered_date_ok"
3 DETAIL:  Failing row contains (10, 1, 7, 12, 1024-11-21).
4 psql:factory/insert_into_table_demand_error_2.sql:5: STATEMENT:  /* Store
   ↪ some incorrect data into the table 'demand'. */
5 -- Trying to insert an order with an invalid date.
6 INSERT INTO demand (customer, product, amount, ordered)
7 VALUES (1, 7, 12, '1024-11-21');
8 # psql 16.12 failed with exit code 3.
```

Chapter 10

Join-based Select and Views

So far, things are going well. On the positive side, we have learned that we can divide the data of factory into separate aspects and then store it into different tables. This is nice. We also learned that we can ensure the integrity of the data inside a single table in various ways. For example, we can use datatypes that strictly enforce their domains. `DECIMAL`, for example, makes sure that numbers are represented exactly without loss of precision to a fixed number of decimals [301]. `DATE` ensures that valid dates based on the Gregorian calendar are entered [204, 335]. We can also define constraints, ranging from ensuring that all columns are entered (`NOT NULL`) over preventing name clashes (`UNIQUE`) to sanity checks (via `CHECK` constraints) [100]. We also learned that we can even ensure the consistency of the data in our DB across different tables via the `REFERENCES` constraint [167]. So we cannot just have tables, they can have strongly-typed columns and the data integrity can be enforced throughout the complete DB. This puts **relational databases** well ahead the spread sheets produced by the likes of Microsoft Excel or LibreOffice Calc.

However, on the negative side, we have to admit that using DBs is more complicated. And we are certainly losing in terms of readability of our data. Indeed, the meaning of the `demand` data shown in Listing 9.20 is not obvious without knowing the contents of the other tables. Now we will see that we can also *use* these contents of other tables directly in our queries, to produce clear and human-readable output.

10.1 Joining Tables

We want to have a list of all of our customers, and for each customer we want to see the `id` values of all of the demands (orders) they have issued. We build the query used in Listing 10.1 step-by-step. The result of our query should have two columns. The first column should hold the customer names. Let's call it `name`. The second column should hold the `id` of the order the customer made. Let's call this column `demand_id`. Now if a customer issued multiple orders, then they should appear multiple times in this table, once for each demand. If a customer did not issue any orders, then they should still appear, but only once and with a `NULL` value in the `demand_id` column.

This means that first, we need all the customer names. A `SELECT name FROM customer;` would do this for us. Now we need to cross-reference the customer `id` values in table `customer` with the column `customer` in table `demand`. We also want to list the customers who did not issue any order.

10.1.1 LEFT JOIN

What we want to do is to apply a so-called `LEFT JOIN`, also called , of the table `customer` to the table `demand` [233]. The syntax of a `LEFT JOIN` is as follows:

```

1 -- The Syntax of a LEFT JOIN.
2 -- We combine the information of the two tables Table_1 and Table_2.
3
4 -- Usually (but not necessarily), Table_1 has primary key column 'a'
5 -- which is referenced as foreign key by column 'b' in Table_2.
6 -- In this example, we return the values of the columns 'x' of Table_1
7 -- and 'y' of Table_2 after joining the two tables.
8 -- For rows in Table_1 without match in Table_2, result rows are created
9 -- but they have NULL for the attributes that would come from Table_2.
10 SELECT Table_1.x, Table_2.y FROM Table_1
11 LEFT JOIN Table_2 ON (Table_1.a = Table_2.b);
12
13 -- Table_1          Table_2          Query Result
14 -- a | x           b | y           x | y
15 -- -+-----+-----+-----+-----+
16 -- 1 | Hello       1 | A           Hello | A
17 -- 2 | World       2 | B           World | B
18 -- 3 | from        2 | C           World | C
19 -- 4 | HFUU        4 | D           World | E
20 --          2 | E           from | <NULL>
21 --          HFUU      D

```

In this query, usually `Table_1` has primary key `a`, which is referenced as foreign key from column `b` of `Table_2`. In our case, `Table_1` would be the table `customer`. Instead of `a`, the primary key column is `id`. `Table_2` corresponds to table `demand`. Column `customer` in table `demand` then corresponds to `Table_2.b`. Notice that we can prefix column names with table names to add clarity: `Table_1.a` is the same as column `a` of table `Table_1`.

The important part of the `LEFT JOIN` is the `ON` condition. It can be an arbitrarily complex condition, involving `AND` and `OR` and whatnot. However, in the simplest form, it just picks a column from `Table_1`, here `Table_1.a` and says that its value must be the same as a column in `Table_2`, here `Table_2.b`. This condition then applies to the rows that are returned, with the only exception that, if there is no entry in `Table_2` for some value of `Table_1.a`, then `NULL` will be returned for all fields used from `Table_2` in the corresponding result row.

We apply this to our situation. As said, `Table_1` is obviously `customer` and `Table_2` is `demand`. The thing that we want to select is `customer.name` and the corresponding values of `demand.id`. The fields that need to match are `customer.id` and `demand.customer`.

In Listing 10.1, we `SELECT customer.name, demand.id` and we do so `FROM customer LEFT JOIN demand ON (customer.id = demand.customer)`. When executed, this query goes through the table `customer`. For each record, it will take the field `id`, i.e., `customer.id`. It will then search for any row in the table `demand` whose field `customer`, i.e., `demand.customer` has the same value. For any such row, it will write a row to the output with the `name` of the customer and the `id` of the row in `demand`. If and only if no such row exists for a customer, i.e., if the customer did not yet make any purchase, it writes a row to the output with the `name` of the customer and `NULL` as the `demand` id. This means that we get the demands associated with each customer name. We also see which customer did not make any purchase.

To clean up the output, we rename `customer.name AS name` and `demand.id AS demand_id` to make it clearer which name and ID values will be displayed. We also add an `ORDER BY name` to sort the output by customer name.

The result of this query, shown in Listing 10.2, has nine rows. We have eight demands, which are associated with customers Bebba, Bebbo, and Bibbo. Bobbo did not yet make any purchase, so he appears as a single row with `NULL` associated as `demand_id`. The `psql` client just leaves `demand_id` blank in the output.

We now have cross-referenced two tables! Let us take this a step farther and ask “How many orders did each customer make so far?” We again answer this question in Listing 10.1. Doing this is rather easy: First, we replace `demand.id AS demand_id` in the query with `COUNT(*) AS demands`. We already learned that `COUNT` just counts rows over groups. But in order to let it count meaningfully, we need to create these groups. All we have to do for this is to add a `GROUP BY name` to the query. All rows that share a `name` go into the same group. `COUNT` now counts the rows in these groups. The output in

Listing 10.1: Get the per-customer demands. (stored in file `select_customer_demand.sql`; output in Listing 10.2)

```

1  /* Get the number of demands per customer. */
2
3  -- List all demand IDs for each customer.
4  -- LEFT JOIN: 'Bobbo' also appears once, with a NULL demand.
5  SELECT customer.name AS name, demand.id AS demand_id FROM customer
6      LEFT JOIN demand ON (customer.id = demand.customer)
7      ORDER BY name;
8
9  -- Now we count the demands.
10 SELECT customer.name AS name, COUNT(*) AS demands FROM customer
11     LEFT JOIN demand ON (customer.id = demand.customer)
12     GROUP BY name ORDER BY name;

```

Listing 10.2: The `stdout` of the program `select_customer_demand.sql` given in Listing 10.1.

```

1  $ psql "postgres://boss:superboss123@localhost/factory" -v ON_ERROR_STOP=1
2      ↪ -ef select_customer_demand.sql
3
4  name | demand_id
5  -----+-----
6  Bebba |      7
7  Bebba |      3
8  Bebbo |      8
9  Bebbo |      4
10 Bebbo |      2
11 Bebbo |      6
12 Bibbo |      1
13 Bibbo |      5
14 Bobbo |
15 (9 rows)
16
17 name | demands
18 -----+-----
19 Bebba |      2
20 Bebbo |      4
21 Bibbo |      2
22 Bobbo |      1
23 (4 rows)
24
25 # psql 16.12 succeeded with exit code 0.

```

Listing 10.2 shows us that Bebba made two orders, Bebbo four, and Bibbo also two. Bobbo so far has zero orders in his name.

With this, we now know how to cross-reference tables. We have essentially solved the basic annoyance mentioned before: Yes, our data is distributed over different tables and without knowing what product a product id refers to and which customer corresponds to which customer id, it is hard to understand anything. However, we can just join tables pull all the information together in a query.

10.1.2 INNER JOIN

We are now able to cross-reference different tables and to pull data together. Let us create a query where, for each demand record, we see the customer name, the product name, the product price, the ordered amount, and the order date. The result of this query would then be a clear and human readable overview of the business transactions of our company. Based on the previous example, where we joined the table `customer` with the table `demand`, we anticipate that we will need two “joins” now. We will need to combine the `demand` table with the `customer` table and with the `product` table. The query will therefore probably be more complicated.

Let us construct the new query. We first define what we want in a way more close to SQL: “For each row in table `demand`, we need the corresponding row in table `customer` and the corresponding row in table `product`. ” We know that there always exists exactly one `customer` row, because we defined the foreign key `REFERENCES` constraint that enforces this. We also know that there always exists exactly one `product` row, because we defined the foreign key `REFERENCES` constraint that enforces this. So it cannot be that we have a row `demand` but no corresponding row in `customer` or `product`. We already know that it is totally possible the other way around, i.e., that there is a row in `customer` without any related row in `demand` – but this is not important here.

Given this information, we could use the `LEFT JOIN` from before. It would generate rows also for rows of `demand` that do not match to any `customer` or `product`. But such rows do not exist anyway. However, what we actually would like to use here is an `INNER JOIN` [233]. The `INNER JOIN` only creates output rows when the rows of all involved tables match each other. While this is not an issue here, it is an important aspect in many other situations.

```

1 -- The Syntax of an INNER JOIN.
2 -- We combine the information of the two tables Table_1 and Table_2.
3
4 -- Usually (but not necessarily), Table_1 has primary key column 'a'
5 -- which is referenced as foreign key by column 'b' in Table_2.
6 -- In this example, we return the values of the columns 'x' of Table_1
7 -- and 'y' of Table_2 after joining the two tables.
8 -- For rows in Table_1 without match in Table_2, no result row is
9 -- created.
10 SELECT Table_1.x, Table_2.y FROM Table_1
11   INNER JOIN Table_2 ON (Table_1.a = Table_2.b);
12
13 -- Table_1          Table_2          Query Result
14 -- a | x           b | y           x | y
15 -- ---+---+-----+---+-----+---+---+
16 -- 1 | Hello       1 | A           Hello | A
17 -- 2 | World      2 | B           World | B
18 -- 3 | from       2 | C           World | C
19 -- 4 | HFUU       4 | D           World | E
20 --               2 | E           HFUU | D

```

But how do we go about joining *three* tables? We can chain arbitrary join expressions. Thus, to merge data from three tables, we simply chaining two `INNER JOIN` expressions:

```

1 -- The Syntax of an Two INNER JOINS.
2 -- We combine the information of tables Table_1, Table_2, Table_3.
3
4 -- Output rows are only created if rows in all three tables match.
5 SELECT Table_1.x, Table_2.y, Table_3.z FROM Table_1
6   INNER JOIN Table_2 ON (Table_1.a = Table_2.b)
7   INNER JOIN Table_3 ON (Table_2.y = Table_3.c);
8
9 -- Table_1          Table_2          Table_3          Query Result
10 -- a | x           b | y           c | z           x | y | z
11 -- ---+---+-----+---+-----+---+---+-----+---+---+
12 -- 1 | Hello      1 | A           A | 10          Hello | A | 10
13 -- 2 | World      2 | B           B | 20          World | B | 20
14 -- 3 | from       2 | C           B | 30          World | B | 30
15 -- 4 | HFUU       4 | D           C | 40          World | C | 40
16 --               2 | E           D | 50          HFUU | D | 50
17 --                   F |             E | 60

```

Equipped with the understanding of this concept, we begin by doing basically the same thing as before: We `SELECT <something> FROM demand` and then `INNER JOIN customer ON (customer.id = demand.customer)`. For each row in `demand`, this gives us the corresponding row in `customer`. It also would drop each row in `demand` for which no row in `customer` exists from the output, but that cannot happen anyway.

Listing 10.3: Selecting the sales data by joining the three tables `demand`, `customer`, and `product`. (stored in file `select_sale_data.sql`; output in Listing 10.4)

```

1  /* Query the sales information. */
2
3  -- We combine the three tables customer, product, and demand.
4  -- Basically, for each row in 'demand', we find the corresponding rows
5  -- in the 'customer' and 'product' tables (via the INNER JOIN).
6  -- This gives us one long row with all the information for one 'demand'.
7  -- We now choose only some elements of this long row and rename them.
8  -- We extract the name of the customer and refer to it "customer_name".
9  -- We extract the name of the product and refer to it as "product_name".
10 -- We also print the "amount" from each customer demand and the price.
11 -- We also print when the purchase was made.
12 SELECT customer.name AS customer_name, product.name AS product_name,
13      product.price AS price,           demand.amount AS amount,
14      demand.ordered AS ordered
15 FROM demand INNER JOIN customer ON (customer.id = demand.customer)
16      INNER JOIN product ON (product.id = demand.product)
17 ORDER BY customer_name, ordered, product_name, price, amount;

```

Listing 10.4: The stdout of the program `select_sale_data.sql` given in Listing 10.3.

```

1  $ psql "postgres://boss:superboss123@localhost/factory" -v ON_ERROR_STOP=1
   ↪ -ebs select_sale_data.sql
2  customer_name | product_name | price | amount | ordered
3  -----+-----+-----+-----+-----+
4  Bebba    | Shoe, Size 37 | 152.99 |    7 | 2024-12-16
5  Bebba    | Large Purse  | 150.00 |   10 | 2025-01-16
6  Bebbo    | Shoe, Size 38 | 154.99 |    2 | 2024-12-09
7  Bebbo    | Shoe, Size 40 | 158.99 |    7 | 2024-12-30
8  Bebbo    | Shoe, Size 41 | 160.99 |    4 | 2025-01-12
9  Bebbo    | Shoe, Size 38 | 154.99 |    6 | 2025-02-05
10 Bibbo   | Shoe, Size 42 | 162.99 |   12 | 2024-11-21
11 Bibbo   | Shoe, Size 40 | 158.99 |    3 | 2025-01-05
12 (8 rows)
13
14 # psql 16.12 succeeded with exit code 0.

```

Now we simply *chain* the next `INNER JOIN` by writing it directly after that! We write `INNER JOIN product ON (product.id = demand.product)`. This will also select the corresponding row from table `product`. If such a row does not exist, then the whole current combined row from `demand` and `product` would be dropped from the output. Thanks to our `REFERENCES` constraints in table `demand`, this, again, can never happen anyway.

We have nicely pulled the customer and product data for each demand order. Having all the data at hand, we can choose the columns that we want to output and replace the `<something>` we first nonchalantly wrote when beginning to design the query. We want to get the customer name and product name. Notice how both the table `customer` and the table `product` have a column called `name`. To be able to tell them apart, we will not write `name` for the customer name like in the previous big query that we designed. Instead, we will select the customer name as `customer.name AS customer_name`. For the product name, we write `product.name AS product_name`. Thus, there can never be any confusion here, because we refer to these columns as `customer.name` and `product.name`, respectively. Next we want the list product price and amount associated with an order, i.e., `product.price AS price` and `demand.amount AS amount`. Finally, we also return the order date and therefore add column `demand.ordered AS ordered`.

For good measures, we add an `ORDER BY` clause and sort the output by `customer_name`. If two entries have the same `customer_name`, then we sort the one with the earlier order date (`ordered`) first. If two entries have the same `customer_name` and `ordered` date, then we sort the one with the lexicographically smaller `product_name` first. We try to resolve further draws by `price` and `amount`.

The full query is shown in Listing 10.3. The output in Listing 10.4 shows the eight orders issued. However, unlike the raw data from table `demand`, the output is now clear and easy to read.

10.2 Views as Virtual Tables

The very clear sales information that we just saw is very useful for our factory. Indeed, it is likely that we will need this query structure several times. For example, we probably want to be able to query all the orders that were made in a specific year. This would basically extend the same query with a `WHERE` clause attached that limits the demand dates. Or maybe we want to sum up the cash flow per customer, to see who our most lucrative customers are and to maybe make them some special offers. Or maybe we want to summarize the total sales per product. This would tell us which of our products sell well and which don't.

Every time we would re-write the query, just a bit differently. Of course, this is not a good approach. On the one hand, every time we write a query, there is a chance that we introduce a mistake. On the

Listing 10.5: Creating a view, i.e., a stored SQL query that can be treated like a table, to list the demands in human-readable form. (stored in file `create_view_sale.sql`; output in Listing 10.6)

```

1  /* Create a view showing the sales information. */
2
3  -- We combine the three tables customer, product, and demand.
4  -- Basically, for each row in 'demand', we find the corresponding rows
5  -- in the 'customer' and 'product' tables (via the INNER JOIN).
6  -- This gives us one long row with all the information for one 'demand'.
7  -- We now choose only some elements of this long row and rename them.
8  -- We extract the name of the customer combined with their phone number
9  -- and refer to it "customer_name".
10 -- We extract the name of the product and refer to it as "product_name".
11 -- We also print the "amount" from each customer demand and the price.
12 -- We also print when the purchase as made.
13 CREATE VIEW sale AS
14     SELECT customer.name || ', ' || customer.phone AS customer_name,
15             product.name AS product_name, product.price AS price,
16             demand.amount AS amount, demand.ordered AS ordered
17     FROM demand INNER JOIN customer ON (customer.id = demand.customer)
18             INNER JOIN product ON (product.id = demand.product)
19     ORDER BY customer_name, ordered, product_name, price, amount;
20
21 -- We can use the view as if it was a table!
22 SELECT * FROM sale;
```

Listing 10.6: The stdout of the program `create_view_sale.sql` given in Listing 10.5.

```

1  $ psql "postgres://boss:superboss123@localhost/factory" -v ON_ERROR_STOP=1
2  ↪ -ebs create_view_sale.sql
3
4  CREATE VIEW
5      customer_name | product_name | price | amount | ordered
6  -----+-----+-----+-----+-----+
7  Bebba, 33333333333 | Shoe, Size 37 | 152.99 | 7 | 2024-12-16
8  Bebba, 33333333333 | Large Purse | 150.00 | 10 | 2025-01-16
9  Bebbo, 55555555555 | Shoe, Size 38 | 154.99 | 2 | 2024-12-09
10 Bebbo, 55555555555 | Shoe, Size 40 | 158.99 | 7 | 2024-12-30
11 Bebbo, 55555555555 | Shoe, Size 41 | 160.99 | 4 | 2025-01-12
12 Bebbo, 55555555555 | Shoe, Size 38 | 154.99 | 6 | 2025-02-05
13 Bibbo, 99999999999 | Shoe, Size 42 | 162.99 | 12 | 2024-11-21
14 Bibbo, 99999999999 | Shoe, Size 40 | 158.99 | 3 | 2025-01-05
15 (8 rows)
16
17 # psql 16.12 succeeded with exit code 0.
```

other hand, if we ever decided to change the structure of DB, maybe by allowing multiple items per demand, then we are stuck with a heap of queries that all need to be changed.

Luckily, the SQL language and relational databases offer us a tool for this situation: We can store the query as so-called *view* [108]. A view is basically a stored query that we can work with exactly as if it was a table. We can apply other `SELECT` queries on top of it. Thus, all we have to do is to store this query as view so that we can re-use it whenever it pleases us.

However, suddenly we realize that there is a potential problem with our query: As we discussed before, customer names are not necessarily `UNIQUE` in our DB design. Product names are `UNIQUE`, but eventually, we may have two customers with the same name. This would be a flaw of this query, as it does not give us any means to distinguish two different customers with the same name. In real systems, you have customer numbers, unique usernames, or phone numbers to avoid this issue. Actually, we *do* have phone numbers here, too, and they are `UNIQUE`. Then again, just displaying the phone number is probably confusing to whoever reads the result of the view.

So what are we going to do? Shall we display the customer names, which are not unique? Then, if we add up sales per customer *name*, people with the same name will be grouped together and the result may be wrong. Or should we display phone numbers? This would render the output close to unreadable for human operators.

Let's split the difference: We simply concatenate the customer name and their phone number. We not just select `customer.name AS customer_name`. Instead, we write `customer.name || ', ' || customer.phone AS customer_name`. The `a || b` operator concatenates two strings `a` and `b` [424]. Thus `'Hello' || 'Word!'` will result in the string `'Hello World!'`. Our third customer, Mrs. Bebba, would be displayed as `'Bebba, 3333333333'`. Since the phone numbers are `UNIQUE` in our DB, customer names plus phone numbers must also necessarily be unique. As the result, the data is readable, because the human operator can see the names. And if ever another Mrs. Bebba would become our customer, maybe she has phone number 7777777777, then it would be impossible to mix up the two Bebbas. One would show up as `'Bebba, 3333333333'` and the other one as `'Bebba, 7777777777'`.

We now can confidently save this improved query into the DB as a *view*. This way, we can re-use it whenever we want. Storing it as a view is very very simple. We first need to pick a name for it. Let's call it `sale`. Then, we just need to write `CREATE VIEW sale AS` right before our new query [108]. If we do this, the query is not actually executed. It is stored under the name `sale` as a view. We do this in Listing 10.5.

Then how do we actually execute the query? Simple: We can treat `sale` as if it was a table. We write `SELECT * FROM sale;`. We do this as well in Listing 10.5.

The result is shown in Listing 10.6. Eight beautiful human readable rows of data.

10.3 Using our View

Above, we said that the view `sale` can be used like a table, with the exception that you cannot add or change data. `SELECT * FROM sale;` looks very much like that. But we can do a lot more. We now want to answer the questions "Which customer bought products for the most money for us?" and "Which product sold for the most money?"

The data we need to answer these questions is already available in the view `sale`. For each demand, we have the `price` of one unit of the ordered product as well as the `amount`, the number of product units purchased, in our table. If we multiply them with each other, i.e., compute `amount * price` we know how much the customer paid. All we need to do is to sum up this quantity per customer and we know how much money each customer sent to us. We can also sum `amount * price` for each product and we get how much money came in on a per-product basis.

In the former case, we would `GROUP BY customer_name` and then `SELECT customer_name, SUM(amount * price) AS SUM from sale`. We could give the sum the name `customer_sale`. To directly see who purchased most from us, we could also add a `ORDER BY customer_sale DESC`. The `DESC` enforces sorting in descending order, i.e., larger values come first.

We can do pretty much the same for products, in which case we would `GROUP BY product_name`. We could compute `SUM(amount * price) AS product_sale` and we would know for how much money each product sold [4]. On a per-product basis, it makes also sense to look at the total units sold, so

Listing 10.7: Compute the per-customer and per-product sales based on the view `sale`. (stored in file `select_from_view_sale_1.sql`; output in Listing 10.8)

```

1  /* Extract some information from our database using the view 'sale'. */
2
3  -- Get the total sales per customer.
4  SELECT customer_name, SUM(amount * price) AS customer_sale FROM sale
5    GROUP BY customer_name ORDER BY customer_sale DESC;
6
7  -- Get the total sales per product.
8  SELECT product_name, SUM(amount) AS total_amount,
9    SUM(amount * price) AS product_sale FROM sale
10   GROUP BY product_name ORDER BY product_sale DESC;
```

Listing 10.8: The stdout of the program `select_from_view_sale_1.sql` given in Listing 10.7.

```

1  $ psql "postgres://boss:superboss123@localhost/factory" -v ON_ERROR_STOP=1
2    ↪ -efb select_from_view_sale_1.sql
3  customer_name | customer_sale
4  -----+-----
5  Bebbo, 55555555555 | 2996.81
6  Bebba, 33333333333 | 2570.93
7  Bibbo, 99999999999 | 2432.85
8  (3 rows)
9
10 product_name | total_amount | product_sale
11 -----+-----+-----
12 Shoe, Size 42 | 12 | 1955.88
13 Shoe, Size 40 | 10 | 1589.90
14 Large Purse | 10 | 1500.00
15 Shoe, Size 38 | 8 | 1239.92
16 Shoe, Size 37 | 7 | 1070.93
17 Shoe, Size 41 | 4 | 643.96
18 (6 rows)
19 # psql 16.12 succeeded with exit code 0.
```

we could additionally compute `SUM(amount)AS total_amount`. Of course, we could sort the data by the product sale in descending order, i.e., write `ORDER BY product_sale DESC`.

The complete queries can be seen in Listing 10.7. Their result is given in Listing 10.8. We find that Bebbo is our most valuable customer. He purchased products for over 2996元 from us. Bebba purchased for over 2570元 and Bibbo sent about 2433元 to us.

The top-selling product, by far, is our famous “Shoe, Size 42.” We sold 12 pairs of it for about 1956元. Figuratively speaking, “Shoe, Size 42” is the runner up (hehe) with 1590元 for ten pairs sold. Many of our customers also liked to buy “Large Purse.” No wonder that customers love it, as it offers 10g of product per 元 (see Listing 9.8).

It is interesting to briefly think about how the above works. On the top level, we have an SQL query that selects from data from the view `sale`. Fine, you already have a feeling about how such queries work. But remember: `sale` is *not* a table. It itself is just a query! You can imagine that what happens here is that the query stored as view `sale` is executed. Its results are taken and then on top of these results, the new query that sums up the customer purchases (or the product purchases) is executed. The results of `sale` are not stored. `sale` is not a table. Instead, whenever we access it, the query it represents is executed. And we can work with its results as if it was a table. Isn’t that cool?

Assume that your DB is now used for several years. Queries like the above one would take into consideration the whole history of data. Maybe we are only interested in the present year, which, at the time of this writing, is 2025. So what we would like to do is to limit our query from Listing 10.7 to only consider data from 2025.

For this, we add the clause `WHERE (ordered >= '2025-01-01') AND (ordered < '2026-01-01')` to

Listing 10.9: Compute the per-customer and per-product sales based on the view `sale`. (stored in file `select_from_view_sale_2.sql`; output in Listing 10.10)

```

1  /* Extract information from 'sale' but limit the date range. */
2
3  -- Get the total sales per customer in 2025.
4  SELECT customer_name, SUM(amount * price) AS customer_sale FROM sale
5    WHERE (ordered >= '2025-01-01') AND (ordered < '2026-01-01')
6    GROUP BY customer_name ORDER BY customer_sale DESC;
7
8  -- Get the total sales per product in 2025.
9  SELECT product_name, SUM(amount) AS total_amount,
10     SUM(amount * price) AS product_sale FROM sale
11    WHERE (ordered >= '2025-01-01') AND (ordered < '2026-01-01')
12    GROUP BY product_name ORDER BY product_sale DESC;
```

Listing 10.10: The `stdout` of the program `select_from_view_sale_2.sql` given in Listing 10.9.

```

1  $ psql "postgres://boss:superboss123@localhost/factory" -v ON_ERROR_STOP=1
2    ↪ -efb select_from_view_sale_2.sql
3
4  customer_name      | customer_sale
5  -----+-----
6  Bebbo, 55555555555 |      1573.90
7  Bebba, 33333333333 |      1500.00
8  Bibbo, 99999999999 |      476.97
9  (3 rows)
10
11 product_name   | total_amount | product_sale
12 -----+-----+-----
13 Large Purse   |          10 |      1500.00
14 Shoe, Size 38 |          6 |      929.94
15 Shoe, Size 41 |          4 |      643.96
16 Shoe, Size 40 |          3 |      476.97
17 (4 rows)
18
19 # psql 16.12 succeeded with exit code 0.
```

our queries. `ordered` is the date when the customer made the order. We only want to consider records for which the date is greater than or equal to January 1st, 2025. Also, the date must be less than January 1st, 2026. So we combine `(ordered >= '2025-01-01')` and `(ordered < '2026-01-01')` with `AND` and put them into a `WHERE` clause. The updated queries can be found in Listing 10.9.

Their results in Listing 10.10 show us that Mr. Bebbo is still our most valuable customer in 2025, and that Mrs. Bibba still bought products for more money than Mr. Bibbo. However, in 2025, “Large Purse” is our best-selling product with a margin of about 600元 over the runner up. “Shoe, Size 42” did not even make the list in 2025.

At this stage, we have again become more powerful. We now have effectively solved the problem that our data is hard to understand. We now have tools, namely the `JOIN` queries, that allow us to pull together the data from different tables. We originally separated the data to enforce cleanliness without redundancy. We stored the information about Mr. Bebbo only once, even though he made four purchases. The downside was that, in the purchase records for Mr. Bebbo, the name “Bebbo” did not appear. Instead, it was referenced by pointing to another table via its id 2. This made the data hard to read. But now we know how we can put the records back together.

Admittedly, the sales query that pulls all the data together (given in Listing 10.5) is not very easy to read if you are new to SQL. On one hand, once you are more familiar with SQL, such queries become a joy to write. On the other hand, we also can simply store the query in the DB as a so-called *view*. We can re-use this view whenever we want. We can also plug other queries on top of it. So we basically have to do the hard work of designing a good query to pull the data together only once.

Chapter 11

Updating and Deleting Records

We have seen how new data can be inserted into a DB row-by-row in Chapter 9. Often, we also want to modify or delete data. This can be done similarly easily.

11.1 Updating Records

Assume that our factory found another producer for shoe boxes. The new boxes are 5mm less high and therefore cheaper. In Listing 11.1 we want to change the package size for all products in shoe boxes accordingly. We do this with an `UPDATE <table> SET <fields>` statement [461].

```
1 -- Syntax of an UPDATE Command .
2
3 -- We want to change the values of certain columns of all rows in table
4 -- 'tableName' that meet a certain condition .
5 -- We specify the tableName right after UPDATE .
6
7 -- In the 'SET' part , we provide an expression with a new value to be
8 -- assigned to each column that we want to change . The expression can be
9 -- a simple constant value like '1' , but it could also be a mathematical
10 -- expression based on the old values of the affected row , like
11 -- 'column_1 = column_1 * 7' .
12
13 -- If we do not want to change all of the rows , we can provide a 'WHERE'
14 -- clause . The change is then only applied to the rows for which the
15 -- 'WHERE' condition evaluates to True .
16
17 -- Some systems , like PostgreSQL , support a 'RETURNING' statement , which
18 -- is similar to doing a 'SELECT' after the update that only has the
19 -- affected rows as input .
20 UPDATE tableName
21   SET column_1 = expression_1 , column_2 = expression_2 , ...
22 WHERE <condition to select rows that should be changed>
23 RETURNING * ; -- optional / PostgreSQL : get data from affected rows
```

The table is clearly `product`. There is only a single field that we want to change, namely `height`.

The shoe box that we used so far were $350\text{mm} \times 250\text{mm} \times 130\text{mm}$ in size. For each product that ships in such a box, we want to change the height as `height = height - 5`, i.e., we set the “new” height to be the “old” height minus 5mm. We could write `height = 245` just as well, but in the example I wanted to show that we can use arbitrary expressions when updating column.

However, we do not want to change all the rows in the table. Only those that used the shoe boxes of the aforementioned dimensions. So we use an `WHERE` condition the selects only the rows for which `width = 350`, `height = 250`, and `depth = 130` hold.

Finally, we want to see the result of our update query. The `UPDATE` statement, as implemented by PostgreSQL, allows us to supply, basically, a `SELECT` query that chooses data only from the `changed`

Listing 11.1: Modifying some records in the table `product`. (stored in file `update_table_product.sql`; output in Listing 11.2)

```

1  /* We change entries in the table 'product' in our factory database. */
2
3  -- We want to reduce the height of all shoe boxes by 5mm.
4  -- We know what product comes in a shoe box by the box dimensions of
5  -- 350mm * 250mm * 130mm. If it has this size, it's a shoe box.
6  UPDATE product SET height = height - 5 -- new height = old height - 5
7      WHERE (width = 350) AND (height = 250) AND (depth = 130)
8      RETURNING *; -- Same as SELECT * FROM product WHERE ...;
9  -- Now the shoe boxes are 245mm high. Before they were 250mm high.

```

Listing 11.2: The stdout of the program `update_table_product.sql` given in Listing 11.1.

```

1  $ psql "postgres://boss:superboss123@localhost/factory" -v ON_ERROR_STOP=1
2      ↪ -e bf update_table_product.sql
3
4  id |     name      | price | weight | width | height | depth
5  ---+-----+-----+-----+-----+-----+-----+
6  1 | Shoe, Size 36 | 150.99 | 1300  | 350   | 245   | 130
7  2 | Shoe, Size 37 | 152.99 | 1325  | 350   | 245   | 130
8  3 | Shoe, Size 38 | 154.99 | 1350  | 350   | 245   | 130
9  4 | Shoe, Size 39 | 156.99 | 1375  | 350   | 245   | 130
10 5 | Shoe, Size 40 | 158.99 | 1400  | 350   | 245   | 130
11 6 | Shoe, Size 41 | 160.99 | 1425  | 350   | 245   | 130
12 7 | Shoe, Size 42 | 162.99 | 1450  | 350   | 245   | 130
13 8 | Shoe, Size 43 | 164.99 | 1475  | 350   | 245   | 130
14 9 | Small Purse   | 100.00 | 500   | 350   | 245   | 130
15 (9 rows)
16
17 UPDATE 9
18
19 # psql 16.12 succeeded with exit code 0.

```

rows¹. This query is written at the end of the statement as `RETURNING`. We return all the data from the changed rows.

As you can see Listing 11.2, nine rows are affected. Their `height` indeed changed to 245.

When we designed the table `demand`, we create two foreign key columns: `customer` and `product`. The latter one references the rows in table `product` by their corresponding `id` values. We did learn that it is impossible to insert a row into the table `demand` whose value in column `product` does not occur somewhere in column `id` of table `product`. In other words, we cannot have a `demand` record that stored the sale of a product that does not exist. If we create a `demand` record, then there must be corresponding and existing row in `product` associated with it. Otherwise, the record creation failes, as we demonstrated in Listing 9.22.

What, however, happens if we change the value of the `id` of an existing product? In the very first row in table `demand`, customer Bibbo orders 12 units of product "Shoe, Size 42". "Shoe, Size 42" has `id = 7`. Therefore, that `demand` record stores value 7 in its `product` attribute.

What will happen if we change the value of `id` of the record with `id = 7` in table `product` to, say, `id = 20`? If we do this, then all records in the table `demand` which hold `product = 7` would become invalid. Because there no longer would be any row in table `product` with `id = 7`. But this is another table, right? If we change table `product`, will this be affected by the constraints that we defined upon a *completely different* table (namely table `demand`)? In Listing 11.3 we try to do exactly that. And we cannot! Listing 11.4 clearly shows that, no, we cannot violate the referential integrity that way. We cannot change the `id` value of a record in table `product` if that record is referenced elsewhere via a foreign key (`REFERENCES`) constraint.

¹This is a PostgreSQL addition to SQL [463] and does not seem to be part of the SQL standard, but it is also supported by SQLite [361].

Listing 11.3: The attempt to update the table `product` in a way that would violate the referential integrity: We cannot change the primary key `id` value of a record that is referenced by a foreign key from table `demand`. (stored in file `update_table_product_error.sql`; output in Listing 11.4)

```
1 /* Try to change the id of a product referenced elsewhere. */
2 -- Product id = 7 is used in one demand record.
3 -- It therefore cannot be changed to 20.
4 UPDATE product SET id = 20 WHERE id = 7 RETURNING *;
```

Listing 11.4: The stdout of the program `update_table_product_error.sql` given in Listing 11.3.

```
1 $ psql "postgres://boss:superboss123@localhost/factory" -v ON_ERROR_STOP=1
2   ↪ -ebs update_table_product_error.sql
3 psql:factory/update_table_product_error.sql:4: ERROR:  update or delete on
4   ↪ table "product" violates foreign key constraint "demand_product_fkey"
4   ↪ on table "demand"
5 DETAIL:  Key (id)=(7) is still referenced from table "demand".
6 psql:factory/update_table_product_error.sql:4: STATEMENT:  /* Try to change
6   ↪ the id of a product referenced elsewhere. */
7 -- Product id = 7 is used in one demand record.
8 -- It therefore cannot be changed to 20.
9 UPDATE product SET id = 20 WHERE id = 7 RETURNING *;
10 # psql 16.12 failed with exit code 3.
```

11.2 Deleting Records

After running our company for some time, we realized that our stock of “Shoe, Size 36” is never declining. Indeed, nobody ever purchased these smallest-size shoes. Brokenheartedly, we decide to discontinue this product. This means that we somehow need to remove it from the table `product`, as it will no longer be sold.

In Listing 11.5, we do this with the command `DELETE FROM product WHERE id = 1;` [135]. This command is pretty self-explanatory:

```
1 -- Delete all records from table 'tableName' that meet the 'WHERE'
2 -- condition.
3 -- Some DBMSes (like PostgreSQL) allow to use a RETURNING clause, which
4 -- acts like a SELECT applied *only* to the deleted rows (before they
5 -- are purged).
6 DELETE FROM tableName WHERE <condition>
7 RETURNING column1, column2, ...;
```

In Listing 11.5, we use it to delete all the rows from table `product` where the field `id` has value `1`. Only one such row exists, namely the one with “Shoe, Size 36”. It is deleted. Under PostgreSQL, we can directly access the deleted rows via the `RETURNING` statement². To see whether this really worked, we print `SELECT COUNT(*) as number_of_products from product;` before and after the `DELETE FROM` query. Indeed, Listing 11.6 shows that we originally had 11 products in our palette. After deleting the smallest-sized shoes, only ten products are left.

We know that changing records via `UPDATE` is not permitted if it would lead to a violation of referential integrity. How about deletion of records? Let’s say we want to delete the record of customer Bebbo, which has `id = 2`. This customer issued several orders in our system, so there are several rows in table `demand` that reference this record via their foreign key `customer`. In Listing 11.7, we attempt to delete Mr. Bebbo from our DB. And it fails, as it should, because then we would have orphaned records in our table `demand`. We could, however, first delete all of these records from table `demand` (while leaving unrelated records intact, of course). Then we could delete Mr. Bebbo’s `customer` record afterwards.

²This is a PostgreSQL addition to SQL [463] and does not seem to be part of the SQL standard, but it is also supported by MariaDB [136] and SQLite [361].

Listing 11.5: Deleting a row from the table `product`. (stored in file `delete_from_table_product.sql`; output in Listing 11.6)

```

1  /* We delete an entry from the table 'product' in our factory database. */
2
3  -- We got 11 products.
4  SELECT COUNT(*) AS number_of_products FROM product;
5
6  -- Delete the 'Shoe, Size 36' ... nobody ever bought it.
7  DELETE FROM product WHERE id = 1 -- The id of 'Shoe, Size 36' is 1.
8  RETURNING id, name; -- get the id and name of the deleted row.
9
10 -- We now we got only 10 products.
11 SELECT COUNT(*) AS number_of_products FROM product;

```

Listing 11.6: The stdout of the program `delete_from_table_product.sql` given in Listing 11.5.

```

1 $ psql "postgres://boss:superboss123@localhost/factory" -v ON_ERROR_STOP=1
2   ↪ -ebsf delete_from_table_product.sql
3   number_of_products
4   -----
5   11
6   (1 row)
7
7   id |      name
8   ---+-----
9   1  | Shoe, Size 36
10  (1 row)
11
12 DELETE 1
13   number_of_products
14   -----
15   10
16  (1 row)
17
18 # psql 16.12 succeeded with exit code 0.

```

11.3 A Note on Compatibility

In this section, we learned a lot of useful things. We learned how we can update and delete data from tables. This basically completes our understanding of the most important operations in SQL.

However, we also touched another important subject: While most relational databases support the **SQL** language, many of them still differ in the features that they implement and may even add own statements and extensions. PostgreSQL, for example, provides the `RETURNING` clause in its `INSERT`, `UPDATE`, and `DELETE` statements [463]. This is a very very useful clause, because we often want to know, e.g., which data has been changed, or which automatically generated `id`-value has been assigned to a row upon insertion. This clause is only supported by some other DBMSes and even then maybe only partially [136, 226, 361]. Many DBMSes offer the same basic functionality, but may differ in features and command syntax. So even if SQL-based relational databases seem to be compatible at first glance, most often they are not. Choosing a DBMS for a project therefore is often a decision that cannot easily be changed later and, thus, has to be deliberated carefully.

Listing 11.7: We try to delete a record in table `customer` that is used in records in table `demand`. This fails, because otherwise, the referential integrity of our data would be violated. (stored in file `delete_from_table_customer_error.sql`; output in Listing 11.8)

```
1 /* Try to delete customer Bebbo. */
2 -- Customer Bebbo has id = 2. However, he as a demand record referencing
3 -- him. So he cannot be deleted.
4 DELETE FROM customer WHERE id = 2 RETURNING id, name;
```

Listing 11.8: The stdout of the program `delete_from_table_customer_error.sql` given in Listing 11.7.

```
1 $ psql "postgres://boss:superboss123@localhost/factory" -v ON_ERROR_STOP=1
2   ↪ -ebe delete_from_table_customer_error.sql
3 psql:factory/delete_from_table_customer_error.sql:4: ERROR:  update or
4   ↪ delete on table "customer" violates foreign key constraint "
5   ↪ demand_customer_fkey" on table "demand"
3 DETAIL: Key (id)=(2) is still referenced from table "demand".
4 psql:factory/delete_from_table_customer_error.sql:4: STATEMENT: /* Try to
5   ↪ delete customer Bebbo. */
6 -- Customer Bebbo has id = 2. However, he as a demand record referencing
6 -- him. So he cannot be deleted.
7 DELETE FROM customer WHERE id = 2 RETURNING id, name;
8 # psql 16.12 failed with exit code 3.
```

Chapter 12

Connecting from Python

When we look at what we have achieved so far, we find that it is all pretty nice. However, there is one general problem we did not really consider yet: The data is entirely inside the DB. At first glance, this is where it belongs. Giving this a second thought, a realization strikes us: Nobody except us (the DBAs and developers) can really work with this. Yes, we created the user account `boss` for our boss so that they can log in and work with the data. But are we really going to explain to them that they will have to use SQL for this? Will a sales manager really insert customer orders into our DB by firing up the SQL client and then typing `INSERT INTO demand (...)`? Probably not.

The data is in the DB, where it belongs. The DBMS can protect it by enforcing our constraints and via its user and rights management. But only cool people like us can really work with that. Unsophisticated personnel will gaze at `psql` puzzled.

We need a way to access and work with the data from the outside. For this, several possible choices exist.

1. We can write a our own client program, which offers the user comfortable methods to enter and visualize data. The program then communicates with the DB. The user is never bothered with SQL and that alike.
2. We can also write the front end in form of a web application, maybe based on the `Flask server`. The user then can access our front end via the web browser. Our program again does the heavy lifting in terms of SQL and DB interaction.
3. We use a general interface such as `LibreOffice Base` and connect with it to our DB. In such a tool, we can conveniently design forms for entering the data and reports for visualizing it. Users can use this front end and still have full access to SQL and the entrails of our DB.

Here, we will look at the first and third choice. The second choice involves maybe a bit too much background knowledge for this stage of “playing with a DB” example.

So let us begin by writing a program that accesses a PostgreSQL DB. We will use the Python programming language [482]. I strongly recommend to read our course book *Programming with Python* [482] on this subject either before or in parallel. Things like how to install or work with packages are described there, as well as `for` loops and such and such.

For this part of the example, we need the Python programming language and the `psycopg` library installed. It would probably also be useful to have the `PyCharm IDE` ready. How these pieces of software can be obtained is discussed in Chapter 5 and in [482].

Python is a programming language. It allows us to write arbitrary programs using datatypes such as `int`, `float`, and `str`. We can use control flow statements like `if...then...else`, `for` loops, and `while` loops. We can define functions using `def`. It supports OOP and we can create classes using the `class` keyword. The language does not have any built-in connection with PostgreSQL.

However, Python can use packages, which are libraries that offer additional functionality. Some well-known packages are NumPy [137, 196, 232, 302], Pandas [29, 269, 320], Scikit-learn [326, 353], SciPy [232, 477], TensorFlow [2, 262], or PyTorch [322, 353]. Such packages are offered in the central PyPI repository and can be installed using `pip`. Usually, we will install them into a virtual environment and then use them with our application. More on this can, again, be found in [482].

Listing 12.1: A Python program connecting to our factory DB and executing a `SELECT` statement. (stored in file `connect_and_select.py`; output in Listing 12.2)

```

1  """Connect to our factory database and select some records."""
2
3  from psycopg import connect
4  from psycopg.rows import dict_row
5
6  # Connect to the database and create a cursor to interact with the db:
7  with (connect("postgres://boss:superboss123@localhost/factory") as conn,
8        conn.cursor(row_factory=dict_row) as cur): # SELECT returns dicts
9
10     print("Now performing a SELECT request for customer Bebbo (id 2).")
11     cur.execute("SELECT * FROM demand WHERE customer=2")
12     for record in cur: # Iterate over the records in the cursor.
13         print(record) # Print them as-is.
14
15 print("All done.")

```

↓ `python3 connect_and_select.py` ↓

Listing 12.2: The stdout of the program `connect_and_select.py` given in Listing 12.1.

```

1 Now performing a SELECT request for customer Bebbo (id 2).
2 {'id': 2, 'customer': 2, 'product': 3, 'amount': 2, 'ordered': datetime.
   ↪ date(2024, 12, 9)}
3 {'id': 4, 'customer': 2, 'product': 5, 'amount': 7, 'ordered': datetime.
   ↪ date(2024, 12, 30)}
4 {'id': 6, 'customer': 2, 'product': 6, 'amount': 4, 'ordered': datetime.
   ↪ date(2025, 1, 12)}
5 {'id': 8, 'customer': 2, 'product': 3, 'amount': 6, 'ordered': datetime.
   ↪ date(2025, 2, 5)}
6 All done.

```

The important thing is that there is also a Python package for connecting with the PostgreSQL DBMS. This package is called `psycopg` [473] and it implements the standardized *Python Database API Specification v2.0* [268]. We briefly outline how to install this package in Section 5.1. `psycopg` allows us to construct SQL queries inside our Python programs, transmit them to the PostgreSQL server, and to receive the results back in the Python program. This kind of low-level access is perfect for us, since we already learned some basic SQL queries. Thus, the only new thing we need to understand is how to use `psycopg` to send them to the DBMS and how to process the results. Then, we can use Python to interact with our DBs. This gives us the full power of the programming language to process data, which is of course far beyond the capabilities of SQL.

12.1 Reading Data from the Database

From here on, let us assume that `psycopg` is installed and you have opened Listing 12.1 in the PyCharm IDE. At its begin, this Python program imports two functions from the package `psycopg`, which we will use later on, namely `connect` and `dict_row`. We will need them to connect to PostgreSQL and to define how we want to receive the results from `SELECT` statements, respectively.

We now get to the meat of the example. The first and simplest thing that we can do from Python is to read some data from our DB. For this, we would send a `SELECT` request to PostgreSQL. However, we will not start the `psql` client and type the command in. Instead, we will send it from our Python program via `psycopg`.

To do this, we first need to establish a connection to the DB. We therefore enter a `with` statement, which you also find discussed in our sister course *Programming with Python* [482]. The sessions and cursors to the DB are implemented as context managers [493], which means that they will automatically be closed at the end of the `with` block. At the beginning of the block, a connection `conn` is opened to the PostgreSQL DBMS [471] by using the `connect` function. For this purpose, the same connection

URI that we already used with `psql` needs to be specified. It defines the where the **PostgreSQL** server can be found, which user and password to use for logging in, and which DB we want to work on. Once the connection is open, the second line in the `with` block header opens a cursor [268, 472] in the connection.

Cursors are the objects for sending commands to the **PostgreSQL** server via a connection and to receive the server's responses. They are created with the `cursor` method of the connection object. If a cursor is used return results from queries, one may optionally specify a `row_factory` parameter. We here pass in the `dict_row` function, which returns each record (row) resulting from a query as a dictionary of key-value pairs, i.e., a `dict`. The cursor is stored in a variable named `cur`. Like the connection object, it will be closed at the end of the `with` block.

The cursor object `cur` has a method `execute` which we can use to, well, execute an **SQL** command [268]. The first parameter of the function is the SQL statement that should be executed. The optional second parameter can be a sequence of query parameters, but for this first example, we do not need it. All we want to do is to read all the orders issued by customer Mr. Bebbo, i.e., by the customer with ID 2. We thus use the cursor `cur` to issue the command `"SELECT * FROM demand WHERE customer=2"` via the `execute` method.

Once the query is issued, the cursor can be used as `Iterator` [498] in a `for` loop. We can write `for record in cur:` and this will return the query result row by row. And since we specified `dict_row` as `row_factory` when creating the cursor, each row will be a `dict`. This `dict` will have the column names keys and the values as, well, values.

After all these `dicts` are printed to the `stdout`, the `for` loop terminates. Then, the `with` block is over, too, which means that first the cursor is closed and then the connection to the **DB**. You can find the output of our program in Listing 12.2. We remember that Mr. Bebbo has four orders in his name and we would expect these four orders to be returned. And they are returned. As you can see, we have a convenient way to extract data from our DB and to use this data in **Python**.

You can also see that the data has appropriate types: Data which is integer by nature, such as IDs, is printed as integer numbers. Date or time data is returned as instances of `datetime.date`, i.e., the Python datatype to handle such data.

12.2 Inserting Data into the Database

We can do more. Let us now insert some data into the DB with program Listing 12.3. At the top of the listing, we import the type `LiteralString` from the standard `typing` package [415]. `LiteralString` is a type annotation for string *literals*.

What we want to do is to insert some new demand records into our DB. Therefore, we will use the `INSERT INTO` statement. Now the full `INSERT INTO` statement is rather long and does not fit on a single line in Listing 12.3. To make using it twice (as we will) a bit less cumbersome, we store it in a variable `statement`. This variable is annotated with the type hint `LiteralString`, denoting it as a string that was typed in as-is, that is not the result of `(string) interpolation`, concatenation, or any other operation. Either way, we store our statement in this variable. A bit later we will discuss the meaning and structure of this statement. For now, just accept it as is.

We then again enter basically the same `with` statement as in the previous example. At the beginning of the block, again a connection `conn` is opened to the **PostgreSQL** DBMS [471]. Then again a cursor `cur` [472] in the connection is created.

12.2.1 Inserting One Row

We now use the method `execute` of the cursor. The first parameter is, as said, the SQL statement that should be executed. The second parameter is a sequence or mapping with parameters of the statement. This time we need it: Let us circle back to our statement string constant. We wrote `"INSERT INTO demand (customer, product, amount, ordered)VALUES (%s,%s,%s,%s)"`.

The first part of this statement is pretty clear: We will insert a new row into the table `demand` by using the `INSERT INTO` command. We specify the name of the table (`demand`) and the names of the columns that we will set, i.e., `customer`, `product`, `demand`, and `sqlilordered`. So far, the command is the same as used in Listing 9.19.

Listing 12.3: A Python program connecting to our factory DB and then using `INSERT INTO` to insert one record. (stored in file `connect_and_insert_1.py`; output in Listing 12.4)

```

1  """Connect to our factory database and insert one record."""
2
3  from typing import LiteralString
4
5  from psycopg import connect
6
7  # The insert statement has to be a literal string.
8  statement: LiteralString = "INSERT INTO demand (customer, product, amount,
9   → ordered) VALUES (%s,%s,%s,%s)"
10
11 # Connect to the database and create a cursor to interact with the db:
12 with (connect("postgres://boss:superboss123@localhost/factory") as conn,
13       conn.cursor() as cur):
14     print("Executing a single INSERT statement.")
15     cur.execute(statement, # Insert one new demand record.
16                 (3, 4, 5, "2025-03-05"))
17
18 print("All done.")

```

↓ `python3 connect_and_insert_1.py` ↓

Listing 12.4: The stdout of the program `connect_and_insert_1.py` given in Listing 12.3.

```

1 Executing a single INSERT statement.
2 All done.

```

The difference is in the second part. After specifying the fields to be set, the original SQL command in Listing 9.19 gave the record to be stored in parentheses with field values separated by commas after the `VALUES` keyword. Here, we instead write `"VALUES (%s,%s,%s,%s)"`. These `%s` are replaced, one by one, by the parameter values supplied in the second argument. These values are safely converted to the proper SQL representation from which ever datatype we use.

This means that `cur.execute(statement, (3, 4, 5, "2025-03-05"))` inserts one new row into our table `demand`, where `3` is used as value for `customer`, `4` as value for `product`, `5` is used as `amount`, and `"2025-03-05"` is used as value for the attribute `ordered`. Hence, the customer ID for this record is 3, the product ID is 4, the product amount is 5, and the ordered date is March 5th, 2025.

12.2.2 Brief Excursion: Security Concerns

Databases are amongst the most valuable assets of the IT ecosystem in an organization. We need to protect them from both outside and inside attackers. It is important to always focus on their security. Special care must be taken to prevent the theft, manipulation, or destruction of data. When programs access our DBs, then we need to make sure that these programs cannot be used to attack our system.

Programs often receive their input data either from user input, from files, via the internet, or from other programs. If this input is directly channeled to a DB without taking precautions, the result can be mayhem. Trying to manipulate the input data of programs that access DBs is a very old and yet very common attack vector [375].

And this is why the process of parameter substitution that we had to go through when inserting data into our DB looks a bit odd: In Python we have `f-strings`, which are the go-to solution for constructing strings containing parameter values. One may ask: Why would we construct a query in this odd fashion instead of using a perfectly suitable tool offered by the language? Also, before, we used this strange type `LiteralString`. What do we need that for? We did not give a reasonable explanation. The explanation is: systems security.

`LiteralString` is basically a special string type, a subclass of `str`, for strings `literals`, i.e., strings that were written as text in Python. If a value or variable is of type `LiteralString`, then it is still a string. However, this string is explicitly declared to not be the result of any string operation, not be the result of concatenation, and no f-string either. So we do not just not use f-strings, we even explicitly

state that we do not use them intentionally. Why would we do that?

SQL injection attacks (SQL) [109, 257, 324, 375, 481] are attacks on SQL servers that are made possible if the SQL queries are constructed via string concatenation (+) or (string) interpolation. For example, assume that we would construct our query like this: `"INSERT INTO demand (customer, product, amount, ordered)VALUES ("+ s + ")"`. Then someone could set the string `s` to `s = "1, 2, 3, '2023-02-02'); DROP TABLE demand;"`. The `);` in `s` would terminate the insertion query. The rest of the string would then be a new query: `DROP TABLE...` would delete the entire table `demand`¹. And this is just one example. Other typical examples are bypassing user authentication to obtain access to sensitive data. An attacker could do all kinds of evil things to our DB if we would not guard against this kind of behavior.

`LiteralString` is provided by Python for security reasons [415], with the goal to prevent such SQLi attacks. The idea is that, in the future, static type checking tools could check whether a string is constructed or a `literal` and raises errors in security critical situations [415, 475]. It is forbidden to use any form of string construction during the assembly of an SQL command in `psycopg`. It is enforced that each command be a string literal, i.e., of type `LiteralString`. Therefore, attacks exploiting string operations as vector are prevented (as long as we follow these rules).

But then the question arises: OK, no `f-strings` ... then how do we get parameter values into the queries? If all queries would need to be static, fixed, hard-coded string literals, then we couldn't do many useful things, could we? For this the `"%s"` exist as placeholders for query parameter. The idea is that the system can apply proper escaping, meaning that all characters that could lead to problems are replaced with proper `escape sequences` and hence turned into harmless text. Therefore, regardless of what parameter value is supplied, no malicious string for an attack can be constructed. The parameter values will be substituted into the query string in a safe fashion that prevents SQLi attacks.

Best Practice 12

Regardless which programming language or tool you use to access a DB, you must **never** construct SQL commands using string operations such as concatenation or (string) interpolation. Otherwise, you open the door to SQLi attacks. Always use the proper tools, such as query parameters, for dynamic queries.

Also: We should never hard-code usernames and passwords in our programs. We did this here anyway ... to keep the examples simple. But seriously: Don't do that.

12.2.3 Inserting Multiple Rows

After inserting one record via the method `execute`, we now use the `executemany` method to insert multiple records in Listing 12.5. The difference is that this method lets the cursor perform several invocations of a SQL command. The second parameter is a *sequence of* parameter sequences or mappings. We use this command to insert three rows into the `demand` table. If you read the code in Listing 12.5, you find that it is very self-explanatory. The command looks pretty similar to our normal SQL commands, except that we issue it from Python code instead of the `psql` client. After the command is executed, the cursor and the connection to the DB are closed. The output of the program in Listing 12.6 looks successful.

¹We will learn that later, in Chapter 14.

Listing 12.5: A Python program connecting to our factory DB and then using `INSERT INTO` to insert multiple records. (stored in file `connect_and_insert_2.py`; output in Listing 12.6)

```

1  """Connect to our factory database and insert some records."""
2
3  from typing import LiteralString
4
5  from psycopg import connect
6
7  # The insert statement has to be a literal string.
8  statement: LiteralString = "INSERT INTO demand (customer, product, amount,
9   → ordered) VALUES (%s,%s,%s,%s)"
10
11 # Connect to the database and create a cursor to interact with the db:
12 with (connect("postgres://boss:superboss123@localhost/factory") as conn,
13       conn.cursor() as cur):
14     print("Executing three INSERT statements at once.")
15     cur.executemany(statement, ( # Insert three new demand records.
16         (3, 5, 2, "2025-03-16"), (2, 7, 1, "2025-03-29"),
17         (1, 10, 5, "2025-04-05")))
18
19 print("All done.")

```

↓ `python3 connect_and_insert_2.py` ↓

Listing 12.6: The stdout of the program `connect_and_insert_2.py` given in Listing 12.5.

```

1 Executing three INSERT statements at once.
2 All done.

```

12.2.4 Checking the Result

We want to check if it really worked by executing Listing 12.1 for a second time. Listing 12.2 now lists five orders under customer Bebbo's ID. Indeed, we just inserted a new order and it correctly appears.

Let us now verify manually that the changes really reached and are stored in the DB. For this purpose, we fire up `psql` and repeat the query from Listing 10.7. The result in Listing 12.9 the expected increase in sales for customers and products.

12.3 Summary

At first glance, what we did here is not very spectacular. We have basically written SQL commands, like before. The difference is that we issued them from Python instead of the `psql` client shipping with PostgreSQL. This difference, however, is very significant. With `psql`, we can execute SQL commands,

Listing 12.7: The output produced by Listing 12.1 if we execute it a second time after running Listing 12.5.

```

1 Now performing a SELECT request for customer Bebbo (id 2).
2 {'id': 2, 'customer': 2, 'product': 3, 'amount': 2, 'ordered': datetime.
3  → date(2024, 12, 9)}
4 {'id': 4, 'customer': 2, 'product': 5, 'amount': 7, 'ordered': datetime.
5  → date(2024, 12, 30)}
6 {'id': 6, 'customer': 2, 'product': 6, 'amount': 4, 'ordered': datetime.
7  → date(2025, 1, 12)}
8 {'id': 8, 'customer': 2, 'product': 3, 'amount': 6, 'ordered': datetime.
9  → date(2025, 2, 5)}
10 {'id': 13, 'customer': 2, 'product': 7, 'amount': 1, 'ordered': datetime.
11  → date(2025, 3, 29)}
12
13 All done.

```

Listing 12.8: Repeating the query from Listing 10.7 after inserting the records using our Python program from Listing 12.5. (stored in file `select_from_view_sale_1.sql`; output in Listing 12.9)

```

1  /* Extract some information from our database using the view 'sale'. */
2
3  -- Get the total sales per customer.
4  SELECT customer_name, SUM(amount * price) AS customer_sale FROM sale
5    GROUP BY customer_name ORDER BY customer_sale DESC;
6
7  -- Get the total sales per product.
8  SELECT product_name, SUM(amount) AS total_amount,
9    SUM(amount * price) AS product_sale FROM sale
10   GROUP BY product_name ORDER BY product_sale DESC;
```

Listing 12.9: The stdout of the program `select_from_view_sale_1.sql` given in Listing 12.8.

```

1  $ psql "postgres://boss:superboss123@localhost/factory" -v ON_ERROR_STOP=1
2    ↪ -ebs select_from_view_sale_1.sql
3
4  customer_name | customer_sale
5  -----+-----
6  Bebba, 33333333333 |      3673.86
7  Bebbo, 55555555555 |      3159.80
8  Bibbo, 99999999999 |      3032.85
9  (3 rows)
10
11
12
13
14
15
16
17
18
19
20
21 # psql 16.12 succeeded with exit code 0.
```

but that's basically it. With Python, we can write arbitrarily complex programs.

We could have a user interface with windows and buttons for entering data. Or we could read data from [CSV](#) or [XML](#) files and send them to the DBMS. In the other direction, we can also do lots of sophisticated stuff with data that we pull from the DB. We could statistically evaluate, or use it to train a [Machine Learning \(ML\)](#) or [AI](#) model that predicts which customer will order which product and when. We thus now have a new and infinitely powerful tool in our hand to both generate and analyze data.

And Python is not the only programming language. Connectors similar to [psycopg](#) exist for Java [340] and C [337] as well, and probably for many other programming languages, too. The power to efficiently store and retrieve data and to maintain the data integrity of a DBMS can therefore be used from arbitrary programs.

Useful Tool 3

[psycopg](#) [473] is a library that allows us to connect to the PostgreSQL DBMS from Python code. This way, we can design complex applications in Python that interact with a PostgreSQL DB.

Chapter 13

Accessing the Database from LibreOffice Base

In the previous example, we have discussed how we can connect with [Python](#) to our DBMS. In Python, we can develop an application with a nice user interface. Or we could make a Python program that draws data from different sources, like web sites, web services, other [DBs](#), sensors in production machines, etc., and insert the data into our DB. Or maybe we write a Python program that analyzes the data from or DB using [AI](#) and [ML](#) to predict future sales. Or maybe our Python code runs inside a [web server](#) and offers us a web form to enter data manually. All of these are rather “big” applications. They are designed for special purposes by software engineers, implemented and maintained by programmers, and thus cost serious money. Rarely will they happen in small office situations or small companies.

However, even in less affluent environment, it often makes sense to use DBs. There are many situations where smaller businesses or even individuals have the need to store and process data. In such cases, they may look for a commercial solution but such solution may not be available for their specific needs or may be too expensive. Or maybe we are at an early stage of rolling out a DB in a bigger organization. We may plan big investments in the future. But we first need to convince the organization that there is merit in our project. We want to have quick and very cheap prototype that allows us to at least enter and access data in our DB.

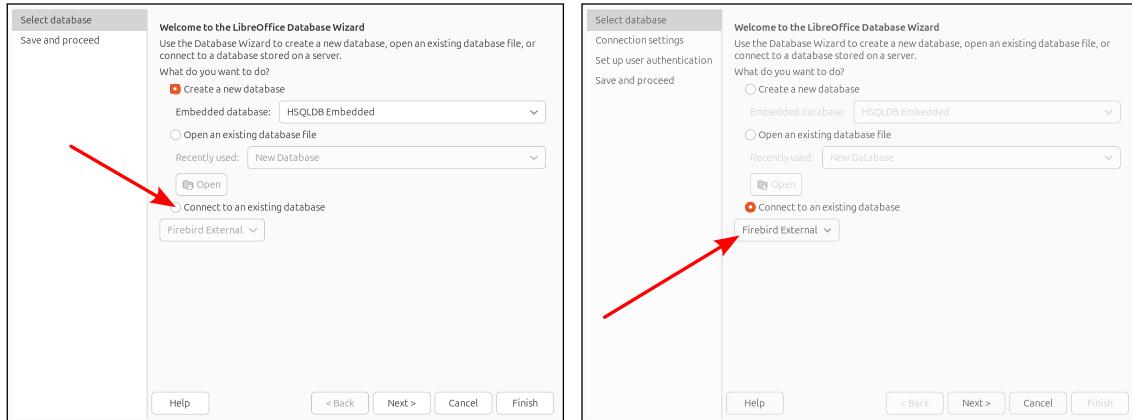
We already learned how to access DBs via [SQL](#) and how to access them from a programming language. A third and completely different way to work with DBs on a professional DBMS is to connect to them from [GUIs](#). Typical examples for such a GUI are the commercial [Microsoft Access](#)[31, 86, 457] and the free and open source [LibreOffice Base](#)[160, 385]. Both allow you to create DB tables in single files and work on them on your local computer. They realize such DBs as local files for the current user to work on. As you can infer from the things we have already seen and done, a “real DBMS” offers us the ability to store the DB on one computer and to connect to this computer from other computers using multiple [clients](#) to work on the centrally managed data. So neither Microsoft Access nor LibreOffice Base can be recommended as DBMS for a complex application going beyond simple hobbyist tasks or small-office scenarios.

However, they both offer some pretty cool tools, such as reports and forms. And instead using them as DBMS, they can be used as GUIs to connect to a DB inside another DBMS. And then we can use these cool tools to work with the DB maintained by a professional DBMS. And that is what we are going to do right now, using the free LibreOffice Base. Please refer to [Chapter 4](#) if you have not yet installed it on your machine.

13.1 Connect to the Database

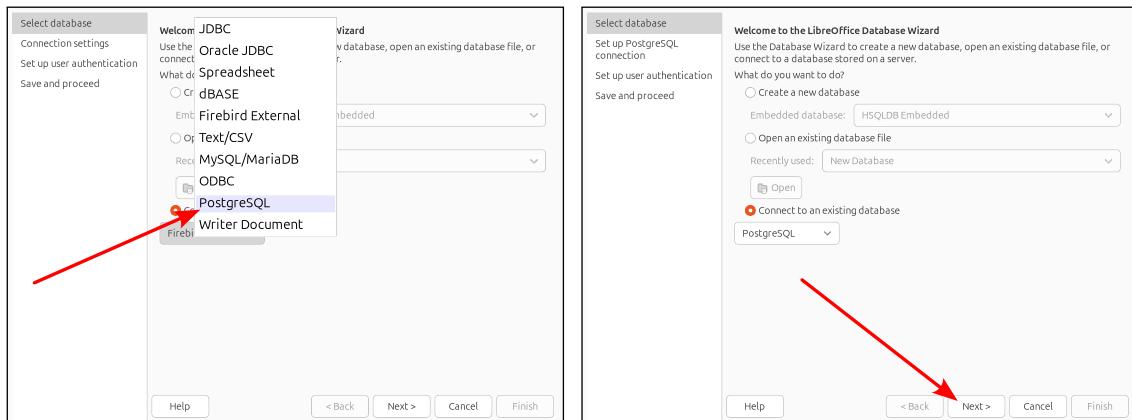
We first need to connect from LibreOffice Base to the [PostgreSQL](#) server and our [factory](#) DB. We therefore open LibreOffice Base as discussed back in [Chapter 4](#). The [PostgreSQL](#) server must be running, too. In the LibreOffice Base opening screen, we choose “Connect to existing database” as shown in [Figure 13.1.1](#), because that is what we want to do. We need to select the [PostgreSQL](#) connector. So we click on the driver drop-down box in [Figure 13.1.2](#). We select [PostgreSQL](#) as DB connection driver in [Figure 13.1.3](#). Now that [PostgreSQL](#) is selected, we can click [Next](#) as shown in [Figure 13.1.4](#).

In the next screen, we need to enter the information about the DBMS and DB we want to connect



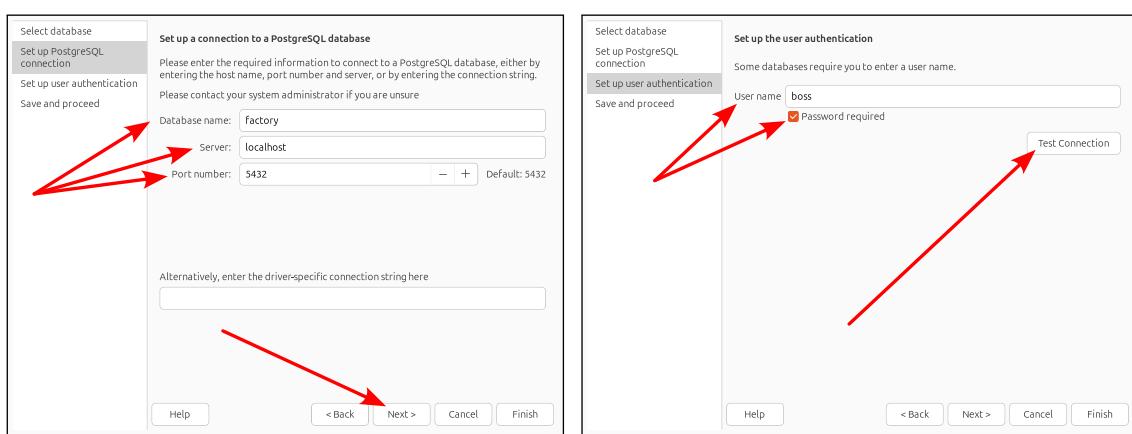
(13.1.1) In the LibreOffice Base opening screen, we choose "Connect to existing database."

(13.1.2) We then click on the DB driver drop-down box.



(13.1.3) We select PostgreSQL as DB driver.

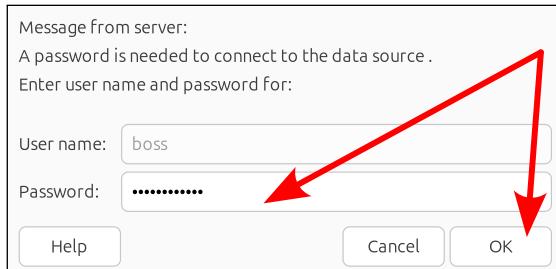
(13.1.4) Once PostgreSQL is selected, we can click Next.



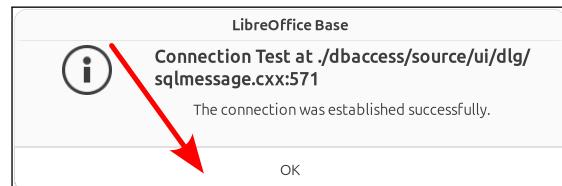
(13.1.5) We enter factory as DB name, localhost as server, 5321 as port, and then click Next.

(13.1.6) As user name we enter boss and select Password required. Then we click Test Connection.

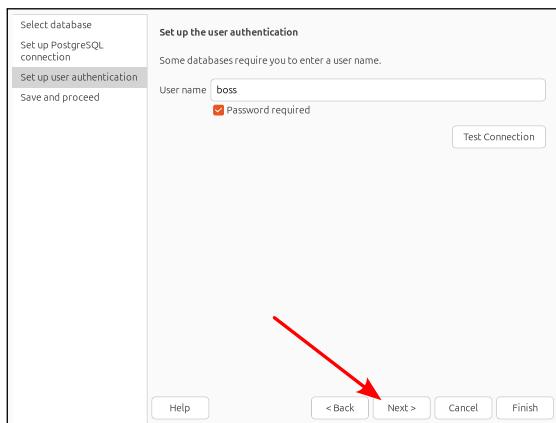
Figure 13.1: Connecting to our example *factory* DB using LibreOffice Base.



(13.1.7) In the authentication window, we enter the password **superboss123** and click **OK**.



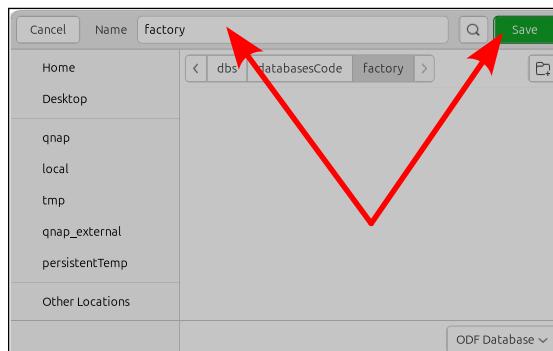
(13.1.8) We get notified that the connection succeeded. We click **OK**.



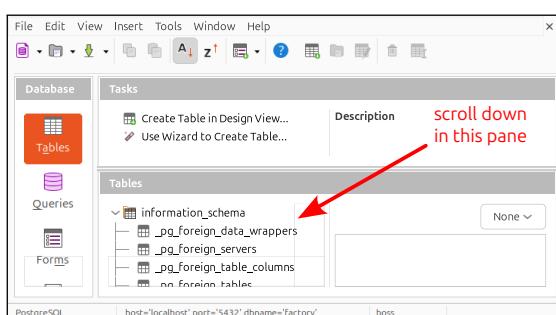
(13.1.9) We now can click **Next** in the authentication menu.



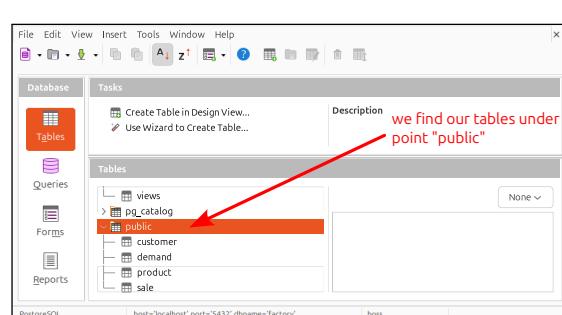
(13.1.10) We do not want to register the DB, we want to open it for editing, and click **Finish**.



(13.1.11) We now save the LibreOffice Base document to a suitable file.



(13.1.12) In the newly opened screen, we go into the **Tables** pane and scroll down.



(13.1.13) If we scroll down to the **public** node, we can find our three tables and the view **sale**.

Figure 13.1: Connecting to our example *factory* DB using LibreOffice Base.

to. The name of the DB is `factory`. As server, we select localhost, because the DBMS is running on our local computer. Of course, if the DBMS was running on another computer, we could enter its IP address here. As port, we select the standard `PostgreSQL` port 5321. Then we click `Next` in Figure 13.1.5.

In the following screen, we enter the authentication information. This is how `LibreOffice Base` will log into the `DBMS`. We can enter the user name, which is `boss`. We had set the password `superboss123` for this user, but we cannot enter it here. It has to be entered explicitly everytime we open this connection. This is probably in order to avoid storing sensible data, like a password, in the LibreOffice Base document. This way, credentials cannot get lost or accidentally published when sharing the document with other users. Either way, we need to select `Password required`, because, yes, we have to authenticate the user `boss` via password. In Figure 13.1.6, we then click `Test Connection`.

In the authentication window that pops up in Figure 13.1.7, we enter the password `superboss123` and click `OK`. As you can see in Figure 13.1.8, the connection succeeded. We close this dialog by clicking `OK`.

Back in the authentication screen in Figure 13.1.9, we can now click `Next` in the authentication menu. That takes us to the final screen of the `DB` document creation dialog. We choose that we do not want to register the DB. We choose that we want to open it for editing. Finally, we click `Finish` in Figure 13.1.10.

We now have to save the LibreOffice Base document to a suitable file. The file type is `odb`, which is basically a zip-compressed collection of `XML` documents. Be that as it be, we choose `factory.odb` as file name in Figure 13.1.11. Then we click `Save`.

Finally, the DB `GUI` opens up. We can see lots of stuff. First, let us look for the DB objects we already have created. In the newly opened screen, we go into the `Tables` pane and scroll down in Figure 13.1.12. If we scroll down to the `public` node, we can find our three tables and the view `sale`, as shown in Figure 13.1.13. LibreOffice Base can now see the objects in our factory DB.

We successfully have connected to our example DB and can now work with it from the LibreOffice Base GUI. Of course, this would work in a very similar way with other DBMSes. Also, we could have done something similar using `Microsoft Access` instead of LibreOffice Base. The important point is that there exist third-party tools like Microsoft Access and LibreOffice Base with which we can connect to DBs that are managed by `servers` like `PostgreSQL`, `MySQL`, `Oracle Database`, or `Microsoft SQL Server`. But what can we do with such tools? Soon you will find out.

13.2 Adding Rows to a Table and Executing Views

The first use case of LibreOffice Base for our factory DB is entering data. So far, the only method that we know to enter data is via `SQL`. This is not suitable for the vast majority of people in an organization. People are used stuff like `Microsoft Excel` tables or `Microsoft Word` documents. They are most certainly not be thrilled if they have to learn a programming language for data. Therefore, we now use LibreOffice Base as a simple GUI to enter data into our DB.

We want to use LibreOffice Base to interact with a table inside our `PostgreSQL` DB. We therefore first open our dokument `factory.odb` using LibreOffice Base. To connect with the DB, we will have to enter the password `superboss123`.

Now we can select the table `demand` in the “Tables” pane and double-click on it in Figure 13.2.1. A new window opens. In this window, we see the contents of the table. The column names are column titles. Each record is visualized as a row in the table. This is already good. As said, many people are used to dealing with the likes of Microsoft Excel tables. This view looks a bit like that. They can intuitively understand the concept of columns and rows. This is much clearer and nicer than the output we could get with `psql...`

We can edit the data right in this view. We can also add new records. Therefore, we place the cursor into the second field of the empty row at the bottom Figure 13.2.2. (We skip the `id` column, because its value can automatically be set by the DBMS.) As data, we choose Mr. Bobbo and thus enter customer id 4 in Figure 13.2.3. On April 12, 2025, he ordered 11 units of the product with id 7, i.e., shoes of size 37. After entering this data, when our cursor is in the last cell of the row, we press `Enter`.

The new record is sent to the DBMS in Figure 13.2.4. Notice that, at this stage, the window displays the `id` field of the new row as 0. The reason is that we did not enter any value here. The system

The screenshot shows the LibreOffice Base interface. On the left, the 'Tables' pane is open, displaying a tree view of database objects. A red arrow points from the 'Tables' icon in the sidebar to the 'demand' table node in the tree. Another red arrow labeled 'double click' points to the 'demand' table in the tree. On the right, a separate window displays a table with columns: id, customer, product, amount, and ordered. The table contains 12 rows of data. A red arrow labeled 'click here' points to the second cell in the empty row at the bottom.

	customer	product	amount	ordered
1	1	7	12	11/21/24
2	2	3	2	12/09/24
3	3	2	7	12/16/24
4	2	5	7	12/30/24
5	1	5	3	01/05/25
6	2	6	4	01/12/25
7	3	11	10	01/12/25
8	2	3	6	02/05/25
9	3	4	5	03/05/25
10	3	3	2	03/16/25
11	2	7	1	03/29/25
12	1	10	5	04/05/25

(13.2.1) We double-click on the table `demand` in the “Tables” pane.

(13.2.2) The table `demand` opens in a separate window, displaying the table content. Click into the second cell in the empty row at the bottom.

The screenshot shows two windows. The left window is the 'Tables' pane with a red arrow pointing to the 'demand' table. The right window is a table editor for the 'demand' table. A red arrow labeled 'we enter a new row, then press Tab after entering the 'ordered' value' points to the last row, which is empty except for the 'ordered' column. A red arrow labeled 'value' points to the 'ordered' cell in the last row. The right window also shows a view named 'sale' with a red arrow pointing to its name.

	customer	product	amount	ordered
1	1	7	12	11/21/24
2	2	3	2	12/09/24
3	3	2	7	12/16/24
4	2	5	7	12/30/24
5	1	5	3	01/05/25
6	2	6	4	01/12/25
7	3	11	10	01/16/25
8	2	3	6	02/05/25
9	3	4	5	03/05/25
10	3	5	2	03/16/25
11	2	7	1	03/29/25
12	1	10	5	04/05/25
13	4	7	11	04/12/25

(13.2.3) We enter a new row of data, leaving the `id` column empty. When reaching the end of row, i.e., after entering all the data, we press .

(13.2.4) The row has now been sent to the DBMS. It has not been loaded back from the DBMS, so the `id` is still 0. We click on the “reload” option.

Figure 13.2: Adding a row to the table `demand` and executing the view `sale` from LibreOffice Base.

has sent the new record to the DBMS. The DBMS then sets the `id` field automatically. However, the LibreOffice Base GUI does not know this. In order to see the actual value of the field `id`, we have to reload the data.

We therefore click on the refresh button in Figure 13.2.5. Indeed, now the `id` field of our new record has the new and correct value 13. We close the table window and go back to the main window.

A DB view provides an application with a perspective on the data. A view looks a bit like a table but is actually something like a solidified SQL query. We can also access views from LibreOffice Base and there, too, they look like tables.

Back to the “Tables” pane we now double-click on the view `sale` in Figure 13.2.6. This again opens a new window in Figure 13.2.7. We now see the results of the query in a nice tabular form. It may be that the `customer_name` column is displayed a bit odd on your computer. In this case, just resize it and make it a bit wider by dragging its right border to the left with the mouse. Then all the data will appear correctly.

Remember that above, we just added a new demand into our table? Back in Listing 12.9, there was not a single order for Mr. Bobbo in our system. But now a new one appears in the view window,

	id	customer	product	amount	ordered
▶	1	1	7	12	11/21/24
	2	2	3	2	12/09/24
	3	3	2	7	12/16/24
	4	2	5	7	12/30/24
	5	1	5	3	01/05/25
	6	2	6	4	01/12/25
	7	3	11	10	01/16/25
	8	2	3	6	02/05/25
	9	3	4	5	03/05/25
	10	3	5	2	03/16/25
	11	2	7	1	03/29/25
	12	1	10	5	04/05/25
	13	4	7	11	04/12/25

(13.2.5) After refreshing the data by pressing the `id` field is now 13 as it should be.

customer_name	product_name	price	amount	ordered
Bebba, 333333333333	Shoe, Size 37	152.99	7	12/16/24
Bebba, 333333333333	Large Purse	150.00	10	01/16/25
Bebba, 333333333333	Shoe, Size 39	156.99	5	03/05/25
Bebba, 333333333333	Shoe, Size 40	158.99	2	03/16/25
Beppo, 555555555555	Shoe, Size 38	154.99	2	12/09/24
Beppo, 555555555555	Shoe, Size 40	158.99	7	12/30/24
Beppo, 555555555555	Shoe, Size 41	160.99	4	01/12/25
Beppo, 555555555555	Shoe, Size 38	154.99	6	02/05/25
Beppo, 555555555555	Shoe, Size 42	162.99	1	03/29/25
Bibbo, 999999999999	Shoe, Size 42	162.99	12	11/21/24
Bibbo, 999999999999	Shoe, Size 40	158.99	3	01/05/25
Bibbo, 999999999999	Medium Purse	120.00	5	04/05/25
Bibbo, 999999999999	Shoe, Size 39	156.99	7	05/07/25
Bobbo, 444444444444	Shoe, Size 42	162.99	11	04/12/25

(13.2.6) We now double-click on the view `sale` in the “Tables” pane.

(13.2.7) The view is executed as expected. The newly entered demand also showed up: There now is a sale for Mr. Bobbo.

Figure 13.2: Adding a row to the table `demand` and executing the view `sale` from LibreOffice Base.

at the bottom row. In other words, the view has been executed and led to the expected results. It also confirms that our changes to the DB were indeed persistently stored.

At this stage, we can already see an important benefit of using a **GUI** like **LibreOffice Base**. We can now enter and view the data in our tables much more conveniently. Before, we had to imagine the abstract tabular form of our tables. But now, we can really see and edit them as actual tables. This provides much better haptics to our **DB**. With an interface like that, a normal user could already work. That's quite nice.

13.3 Relationship Diagrams

An important tool in DB development are entity relationship diagrams (ERDs). ERDs are normally used when we design a abstract (conceptual) model of a DB. They are visual representations of the objects and the relationships between them. It makes a lot of sense to first create a model of the real-world objects or information that we want to store in the DB before implementing the DB using **SQL**. The modeled objects can then be mapped to tables and the relationships could become foreign key relationships.

In Figure 13.3, we present a small ERD that sketches the objects that make up our factory DB. As you can see, all three object types and their attributes are illustrated. The lines linking them present their relationships. Here we see that one customer (and product) entity can be linked to an arbitrary number of demand entities. Each demand entity, on the other hand, is linked to exactly one

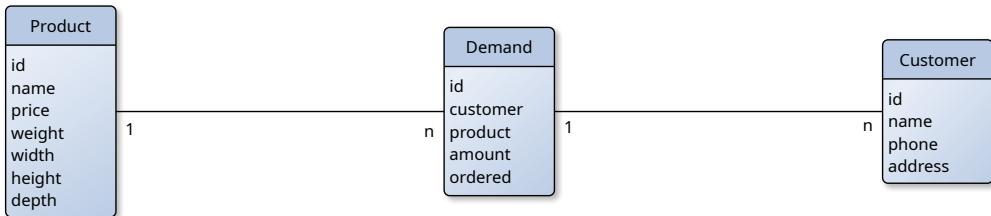
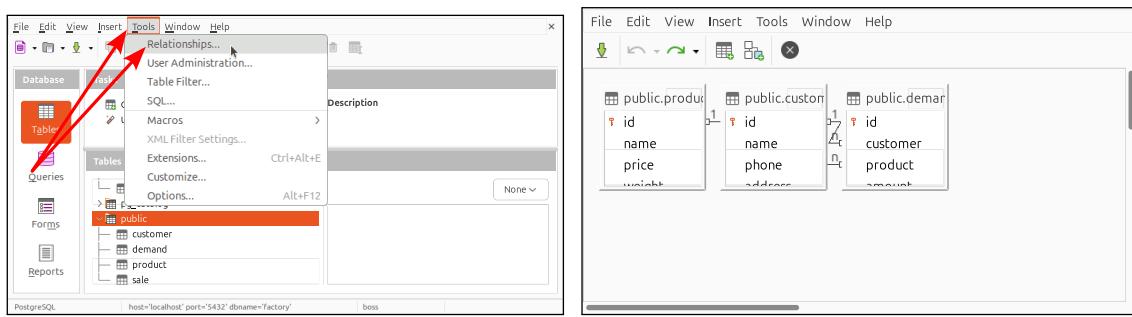
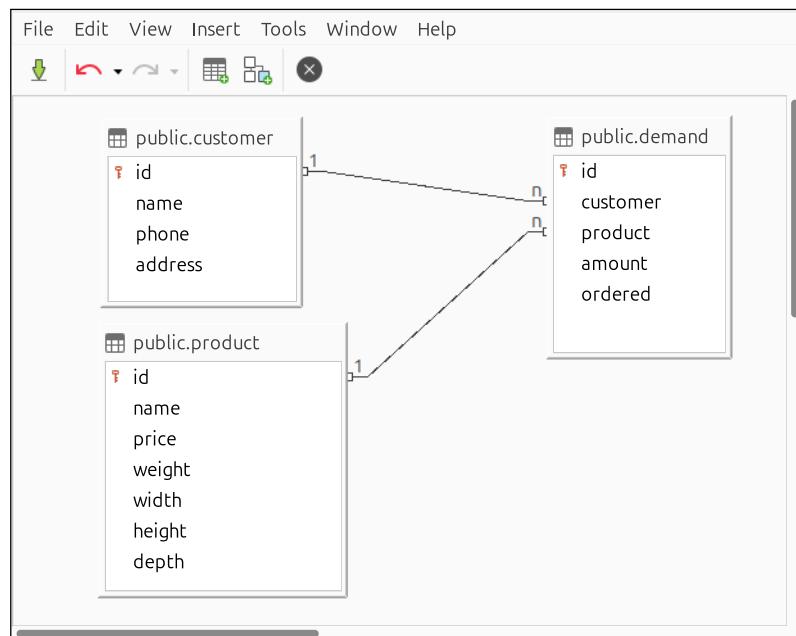


Figure 13.3: An ERD for our factory DB, hand-drawn with yEd. You will learn more about that in Chapter 18.



(13.4.1) To get to the ERD view, we click on **Tools** and then **Relationships...**.

(13.4.2) An ERD view of the tables in our DB appears. It is a bit cluttered, so we drag the tables around and resize them a bit.



(13.4.3) The ERD now looks very clean and illustrates the relationships between the tables in our DB.

Figure 13.4: Viewing an ERD of the tables and their relations in our DB in LibreOffice Base.

customer (and product) entity. We can construct the DB structure based on this design. In Chapter 18, we will in-depth discuss this approach to conceptual DB modeling.

In our example DB, however, we just directly started with the table design in SQL. We wanted to see action as quickly as possible and disregarded any concern about efficient design. This example is not about fancy stuff, it is about exploring the world of DBs.

Let's say that we did indeed design a DB based on entities modeled in a ERD. The DB is created via SQL commands. The tables and their foreign key relationships then correctly represent the model painted as ERD. Then, the information in the ERD is also present in the DB. It is reflected the structure of the tables and the foreign keys. If this is true, then we should be able to reconstruct the ERD at least partially from a DB. Of course, we cannot reconstruct the semantics, i.e., the meaning behind the relationships and objects. But we can well reconstruct the objects and relationships on a purely syntactical level. Actually, we can do that so quickly that we can always work with an ERD-based representation of our DB.

LibreOffice Base can do that. We therefore first open our dokument `factory.odb` using LibreOffice Base. To connect with the DB, we will have to enter the password `superboss123`. We open the menu `Tools` and then click on `Relationships`, as illustrated in Figure 13.4.1. This opens a very cluttered diagram view. This view includes all three tables that we designed in our DB, but in Figure 13.4.2 they are not neatly arranged.

We can click on them, though, and drag them around. We can also drag the bottom edges of the tables and expand them. After some re-arranging, we get indeed a very nice overview on our DB in Figure 13.4.2.

We can see that the `id` columns are the primary keys of the tables, because they are marked with key icons . Furthermore, we see that each `id` value in table `customer` can be used in n records as `customer` value in table `demand`. The same holds for each `id` value in table `product`, which can be stored in n records as `product` value in table `demand`. n here stands for "arbitrarily often."

This ERD is a really overview on the structure of our DB. And it can automatically be generated for us by the LibreOffice Base GUI. If our DB was more complex, with more tables and relationships, this illustration could be quite helpful. Imagine that we join a department of an organization as the new DBA. Imagine you get to work on an existing DB. Sadly, there is no documentation available. All we can do is to access the DBMS and try to figure out how the DB works via SQL. Well, if we would connect with LibreOffice Base to the DBMS, we could at least get a quick and comprehensive overview on the structure of the DB, what tables exist, and how they are related.

13.4 Forms

At some point in this example, we realized that entering data into a DB via a SQL client like `psql` is maybe not a very convenient way. In Section 13.2, we found that we can also enter data into tables in a much more convenient way by typing it in a table-based GUI. This is a big step forward, because it does not require any understanding of SQL. We, as the DBAs, can create and manage a DB using SQL. Then we can connect a frontend like LibreOffice Base to it and give this as client to a secretary. They can then enter the data in a way that is more natural to them. Our DB will stay consistent and the data integrity is preserved by its constraints. This is much better than an Microsoft Excel sheet or that alike. Multiple people can work with DB concurrently using the clients on multiple computers. Different types of related data can be entered simultaneously. Also, it is much harder to create invalid data.

However, entering records with foreign keys is still a total buzzkill. This comes to light when dealing with table `demand`. The user needs to keep the tables `product` and `customer` open, too. They need to look up customer ids and product ids and use them manually. And there is no protection against entering the id of the wrong customer or product. While we can pull the data together nicely in the view `sale` once it is entered. . . . we do not have a way to enter it comfortably. Not yet.

Because now we learn an easy way to build an input method for datasets with foreign keys. This method is called *forms*. Both Microsoft Access and LibreOffice Base allow us to develop so-called forms. Forms are entry masks that can be designed in different ways. We can place different controls onto forms. The controls may take their data from different tables in the DB. This is what we are after. So we will now design a form for entering customer demands into our DB from LibreOffice Base.

We therefore first open our dokument `factory.odb` using LibreOffice Base. To connect with the

(13.5.1) In order to create a new form, we click on "Create Form in Design View..."

(13.5.2) A new and empty form opens in the design view. We want to insert a tabular structure. The corresponding option is most likely hidden behind double-angle button $\langle\langle$ near the bottom on the pane on the left side.

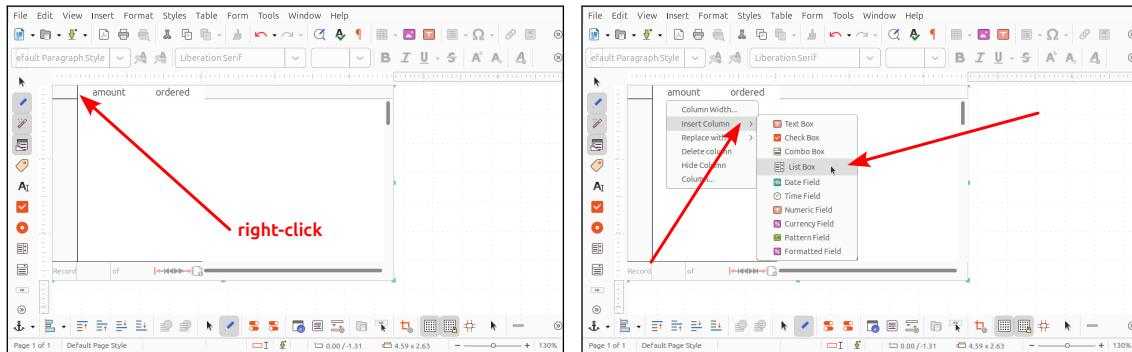
(13.5.3) Click on the **Table Control** button .

(13.5.4) We click into the empty form body and drag the mouse to draw the table area. Then we release the mouse.

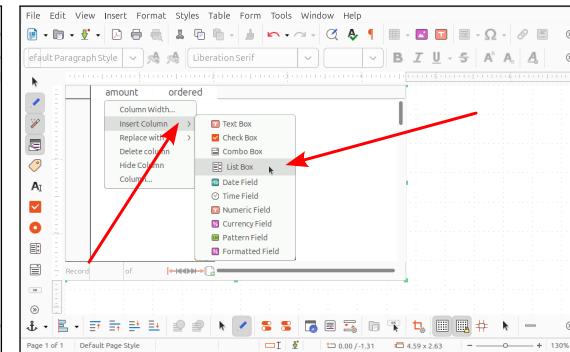
(13.5.5) A dialog opens. It asks about the data we want to use in the table. We scroll down all the way in the view on the right side and select "public.demand." We click **Next**.

(13.5.6) In the next dialog we can select columns to insert. We choose **amount** and **ordered**. We click **Finish**.

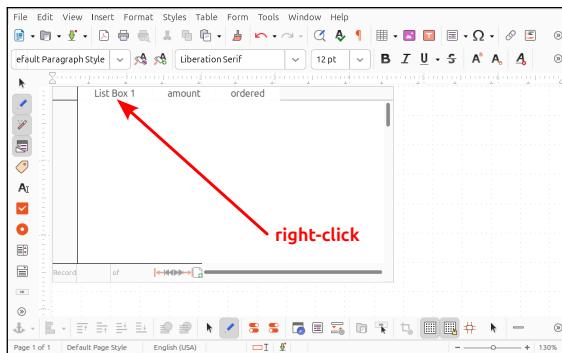
Figure 13.5: Creating a form for entering demand orders into our DB in LibreOffice Base.



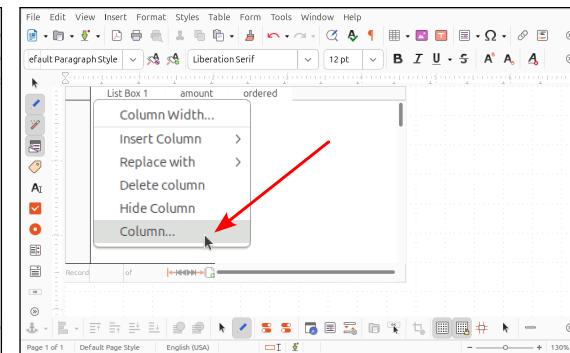
(13.5.7) The table is now inserted in draft form. We want to add more columns. Right-click at the left corner of the **amount** column.



(13.5.8) In the menu that opens, click **Insert Column** and then **List Box**.



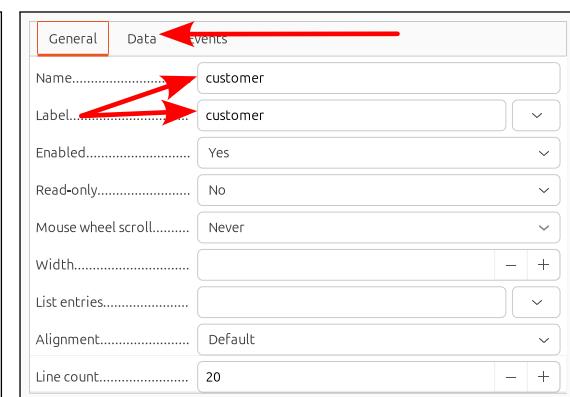
(13.5.9) A new column "List Box 1" has been added. We right-click on it.



(13.5.10) In the menu that opens, click **Column...**.

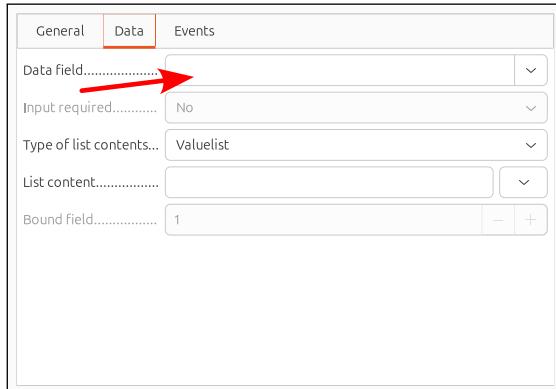


(13.5.11) First, we select the **General** tab in the dialog that opens. We want to change the name and label of the new column.

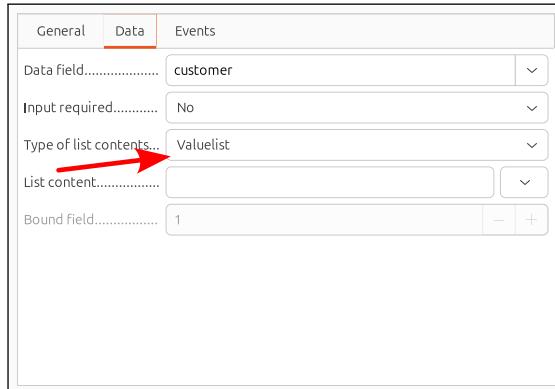


(13.5.12) We change the label and the name of the column to "customer". Then we click on the **Data** pane.

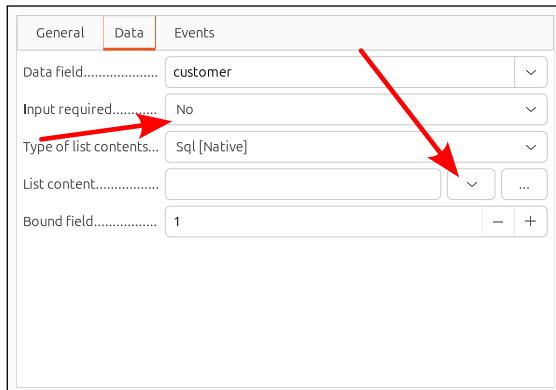
Figure 13.5: Creating a form for entering demand orders into our DB in LibreOffice Base (Continued).



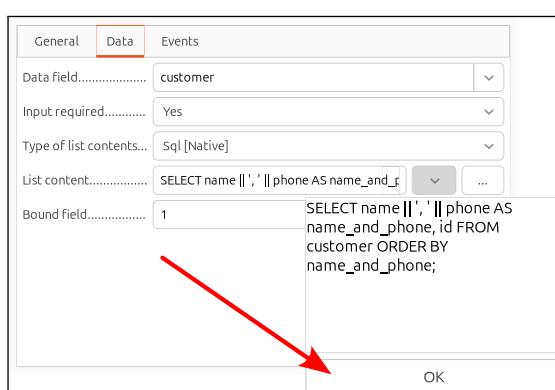
(13.5.13) We first need to choose the column of our `demand` table that should be set via this form field. We type in `customer`.



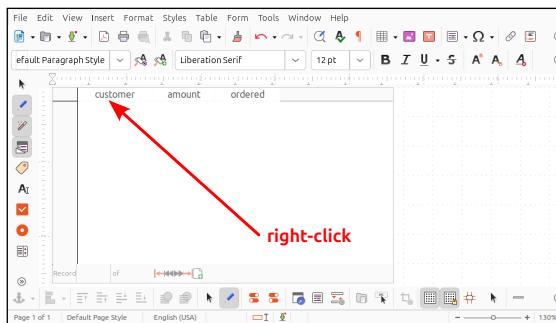
(13.5.14) Next we want to change the *Type of list contents* and *Input Required...*



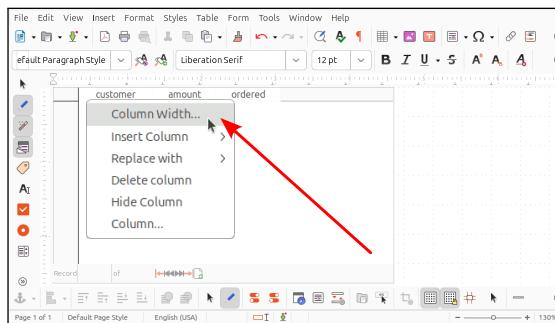
(13.5.15) We change *Input Required* to `Yes`. We set *Type of list contents* to `Sql [Native]`. We click on the small wedge next to *List content*.



(13.5.16) We need to enter a SQL `SELECT` query that returns two values. We select both from table `customer`, as follows: (1) the one that should be displayed. We choose `name || ',' || phone`, i.e., a concatenation of the name and phone number string, separated by a comma. (2) the one to be stored in the `customer` field of the `demand` table. We choose the customer `id`.



(13.5.17) After we clicked `OK` and closed the dialog, the name of the new column has changed to `customer`. It appears to be a bit small for the information that it will contain, so we right-click on it...



(13.5.18) ...and select `Column width...` in the menu that pops up.

Figure 13.5: Creating a form for entering demand orders into our DB in LibreOffice Base (Continued).

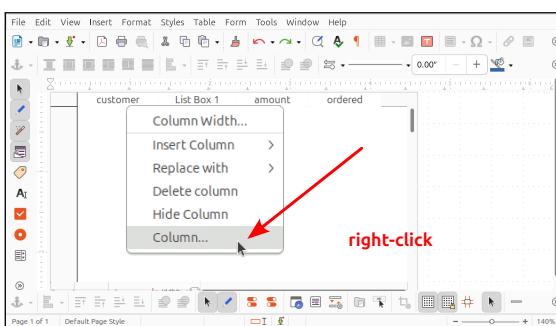
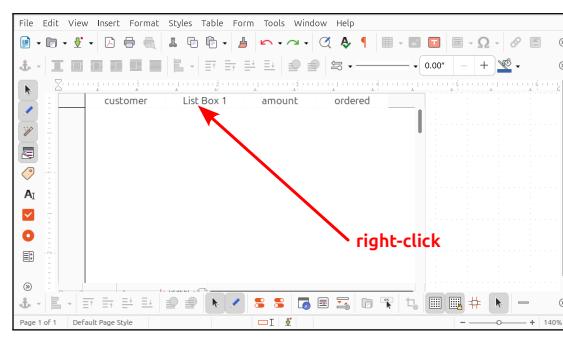
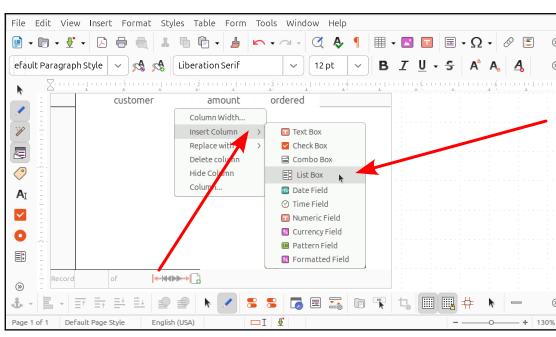
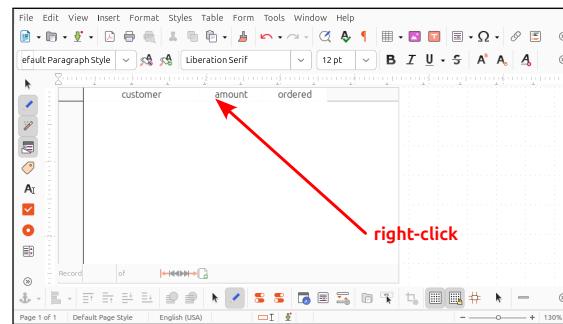
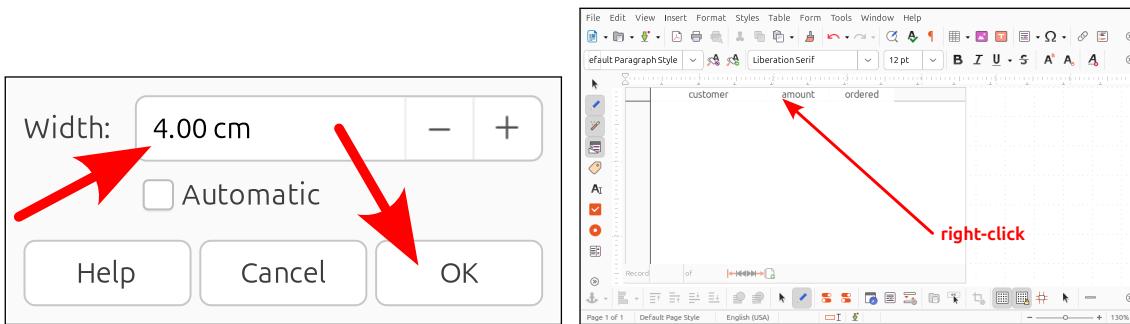


Figure 13.5: Creating a form for entering demand orders into our DB in LibreOffice Base (Continued).

DB, we will have to enter the password `superboss123`. In the **Database** pane on the right-hand side of the LibreOffice Base window, we select **Forms**. Then, under **Tasks**, we click on “Create Form in Design View...” in Figure 13.5.1. Then, in Figure 13.5.2, a new and empty form opens in the design view. Now there are several different possible structures in which we can create a form. Forms can look like the classical dialogs that we used so far in LibreOffice. Or they can look more like the tables-based view that we used to enter demand orders manually in Section 13.2. We want to design a form following this structure, but we want to make it more easy to use, of course. Therefore, we will insert a table control, which can be reached by the button . Depending on your screen size, this button may be directly located on the control palette on the left-hand side of the screen. Or, as is the case on my screen, may be hidden behind double-angle button near the bottom of the control palette. In Figure 13.5.3, we click on that button, a small window with additional controls opens, and then we click on on the table control button .

We now click into the empty form body and drag the mouse to span a reasonably sized area before

General Data Events

Data field..... product

Input required..... Yes

Type of list contents... Sql [Native]

List content..... SELECT name, id FROM product ORDER BY name;

Bound field..... 1

OK

(13.5.25) In the **Data** pane, we again enter a SQL query. This time, we select from table **product**. The **name** column should be displayed to the user, while the **id** column should be stored in the form.

General Data Events

Data field..... product

Input required..... Yes

Type of list contents... Sql [Native]

List content..... SELECT name, id FROM product ORDER BY name;

Bound field..... 1

(13.5.26) The **id** from the selected **product** should be stored in the **product** field of the **demand** record. Therefore, we select **product** as **Data field**...

File Edit View Insert Format Styles Table Form Tools Window Help

Default Paragraph Style Liberation Serif

customer product amount ordered

Record 1 of 1 Default Page Style

(13.5.27) We close the dialog. The form design is now finished. We unselect the pencil symbol in the bottom bar.

File Edit View Insert Format Styles Table Form Tools Window Help

Default Paragraph Style Liberation Serif

customer product amount ordered

Bibbo, 9999999999 Shoe, Size 42 12.00 11, Bebba, 5555555555 Shoe, Size 38 2.00 12, Bebba, 3333333333 Shoe, Size 37 7.00 12, Bebba, 5555555555 Shoe, Size 40 7.00 12, Bibbo, 9999999999 Shoe, Size 40 3.00 01, Bebba, 5555555555 Shoe, Size 41 4.00 01, Bebba, 3333333333 Large Purse 10.00 01, Bebba, 5555555555 Shoe, Size 38 6.00 02, Bebba, 3333333333 Shoe, Size 39 5.00 03, Bebba, 3333333333 Shoe, Size 40 2.00 02,

Record 1 of 14 Default Page Style English (USA)

(13.5.28) This opens our form. It now displays the data to us in a form similar to the **sale** view.

File Edit View Insert Format Styles Table Form Tools Window Help

Default Paragraph Style Liberation Serif

customer product amount ordered

Bebba, 5555555555 Shoe, Size 41 4.00 01, Bebba, 3333333333 Large Purse 10.00 01, Bebba, 5555555555 Shoe, Size 38 6.00 02, Bebba, 3333333333 Shoe, Size 39 5.00 03, Bebba, 3333333333 Shoe, Size 40 2.00 03, Bebba, 5555555555 Shoe, Size 42 1.00 03, Bibbo, 9999999999 Medium Purse 5.00 04, Bobbo, 4444444444 Shoe, Size 42 11.00 04,

Record 14 of 14 Default Page Style English (USA)

(13.5.29) However, it is a **form**, not a **view**. We can edit the data and insert new data. We scroll down to the empty row at the very bottom. We click on the small wedge symbol in the **customer** column.

File Edit View Insert Format Styles Table Form Tools Window Help

Default Paragraph Style Liberation Serif

customer product amount ordered

Bebba, 5555555555 Shoe, Size 41 4.00 01, Bebba, 3333333333 Large Purse 10.00 01, Bebba, 5555555555 Shoe, Size 38 6.00 02, Bebba, 3333333333 Shoe, Size 39 5.00 03, Bebba, 3333333333 Shoe, Size 40 2.00 03, Bebba, 5555555555 Shoe, Size 42 1.00 03, Bibbo, 9999999999 Medium Purse 5.00 04, Bobbo, 4444444444 Shoe, Size 42 11.00 04,

Bebba, 3333333333
Bibbo, 9999999999
Bobbo, 4444444444

Record 14 of 14 Default Page Style English (USA)

(13.5.30) A drop-down list opens from which we can select the customer in *clearly readable form*. We choose Mr. Bibbo.

Figure 13.5: Creating a form for entering demand orders into our DB in LibreOffice Base (Continued).

The screenshot shows a LibreOffice Base form for a 'demand' table. The table has columns: customer, product, amount, and ordered. A red arrow points to the small wedge icon in the 'product' column of the data grid.

customer	product	amount	ordered
Bebbo, 5555555555	Shoe, Size 41	4.00	01,
Bebbo, 3333333333	Large Purse	10.00	01,
Bebbo, 5555555555	Shoe, Size 38	6.00	02,
Bebbo, 3333333333	Shoe, Size 39	5.00	03,
Bebbo, 5555555555	Shoe, Size 40	2.00	03,
Bibbo, 9999999999	Medium Purse	5.00	04,
Bobbo, 4444444444	Shoe, Size 42	11.00	04,
Bibbo, 9999999999			

(13.5.31) We now click on the small wedge in the **product** column.

The screenshot shows a dropdown menu in LibreOffice Base. The list contains various products and sizes. A red arrow points to the 'Shoe, Size 39' option in the list.

- Large Purse
- Medium Purse
- Shoe, Size 37
- Shoe, Size 38
- Shoe, Size 40
- Shoe, Size 41
- Shoe, Size 42
- Shoe, Size 43
- Small Purse

(13.5.32) In the drop-down list that opens, we select "Shoe, Size 39".

The screenshot shows the LibreOffice Base form after selecting 'Shoe, Size 39'. The 'product' field now contains 'Shoe, Size 39'.

customer	product	amount	ordered
Bebbo, 5555555555	Shoe, Size 41	4.00	01,
Bebbo, 3333333333	Large Purse	10.00	01,
Bebbo, 5555555555	Shoe, Size 38	6.00	02,
Bebbo, 3333333333	Shoe, Size 39	5.00	03,
Bebbo, 3333333333	Shoe, Size 40	2.00	03,
Bebbo, 5555555555	Shoe, Size 42	1.00	03,
Bibbo, 9999999999	Medium Purse	5.00	04,
Bobbo, 4444444444	Shoe, Size 42	11.00	04,
Bibbo, 9999999999	Shoe, Size 39		

(13.5.33) We can now enter the amount.

The screenshot shows the LibreOffice Base form with the 'amount' field highlighted and containing the value '7.00'. A red arrow points to the 'ordered' date field.

customer	product	amount	ordered
Bebbo, 5555555555	Shoe, Size 41	4.00	01,
Bebbo, 3333333333	Large Purse	10.00	01,
Bebbo, 5555555555	Shoe, Size 38	6.00	02,
Bebbo, 3333333333	Shoe, Size 39	5.00	03,
Bebbo, 3333333333	Shoe, Size 40	2.00	03,
Bebbo, 5555555555	Shoe, Size 42	1.00	03,
Bibbo, 9999999999	Medium Purse	5.00	04,
Bobbo, 4444444444	Shoe, Size 42	11.00	04,
Bibbo, 9999999999	Shoe, Size 39	7.00	

(13.5.34) We enter the number 7 as amount. We move on to the **ordered** date field.

The screenshot shows the LibreOffice Base form with the 'ordered' date field highlighted and containing the value '05/07/25'. A red arrow points to the next row in the table.

product	amount	ordered
Shoe, Size 41	4.00	01/12/25
Large Purse	10.00	01/16/25
Shoe, Size 38	6.00	02/05/25
Shoe, Size 39	5.00	03/05/25
Shoe, Size 40	2.00	03/16/25
Shoe, Size 42	1.00	03/29/25
Medium Purse	5.00	04/05/25
Shoe, Size 42	11.00	04/10/25
Shoe, Size 39	7.00	05/07/25

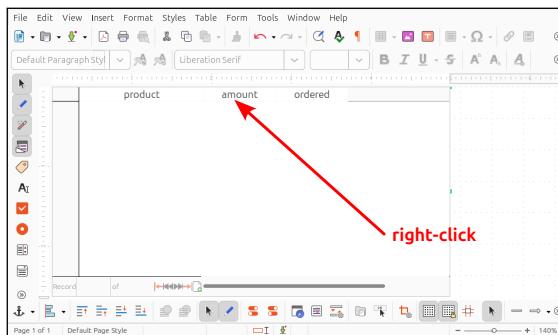
(13.5.35) As order date, we type **05/07/25**, corresponding to May 7th, 2025. We press **Enter**. The cursor jumps to a new row as our record is inserted into the **demand** table.

The screenshot shows the LibreOffice Base form with a new row added to the table. The new row contains 'Shoe, Size 39' in the 'product' field and '05/07/25' in the 'ordered' field. A red arrow points to the pencil symbol in the bottom bar.

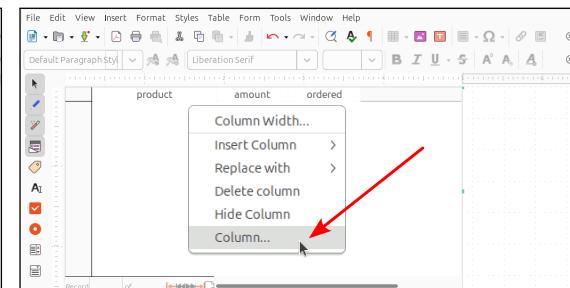
product	amount	ordered
Large Purse	10.00	01/16/25
Shoe, Size 38	6.00	02/05/25
Shoe, Size 39	5.00	03/05/25
Shoe, Size 40	2.00	03/16/25
Shoe, Size 42	1.00	03/29/25
Medium Purse	5.00	04/05/25
Shoe, Size 42	11.00	04/12/25
Shoe, Size 39	7.00	05/07/25

(13.5.36) We are unhappy with the way amounts and dates are displayed. We want to edit the form and thus click on the pencil symbol in the bottom bar.

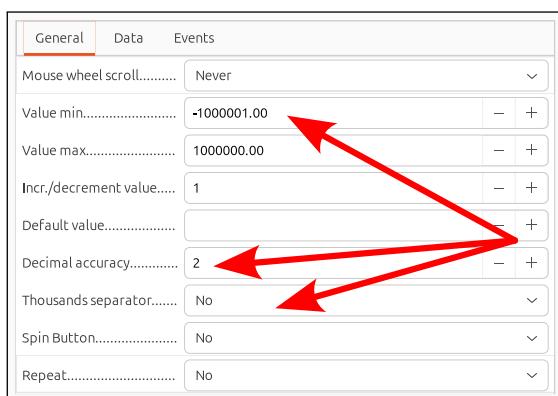
Figure 13.5: Creating a form for entering demand orders into our DB in LibreOffice Base (Continued).



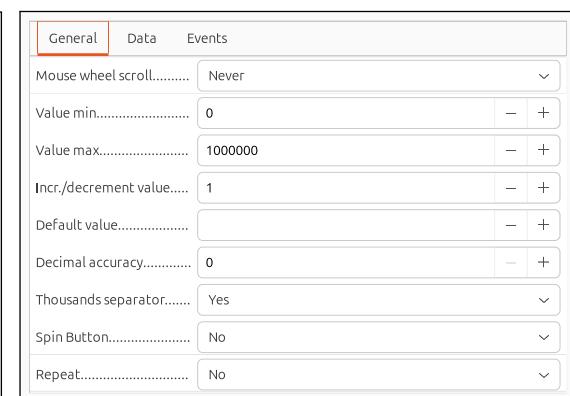
(13.5.37) The design view of the form opens again. We right-click on the `amount` column.



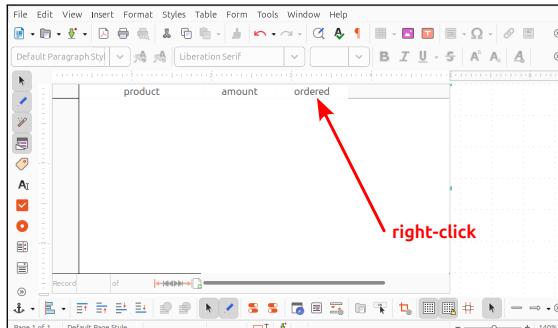
(13.5.38) We click `Column...` in the drop-down menu that opens.



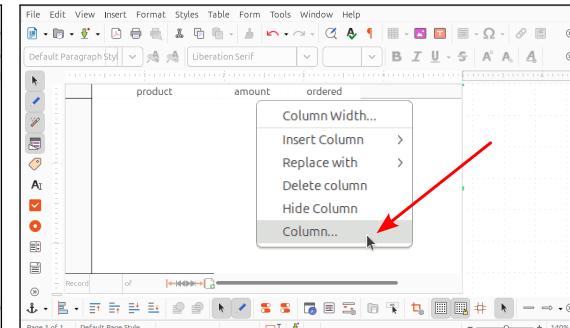
(13.5.39) In the `General` tab, we want to change the minimal value, the decimal accuracy, and the thousand separator setting.



(13.5.40) The minimum gets set to 0. We do not allow fractions, so the decimal accuracy becomes 0. For the case that someone orders thousands of shoes, we would display a thousand separator. We close this dialog.



(13.5.41) We now right-click on the `ordered` column.



(13.5.42) In the drop-down menu that opens, we click `Column...`.

Figure 13.5: Creating a form for entering demand orders into our DB in LibreOffice Base (Continued).

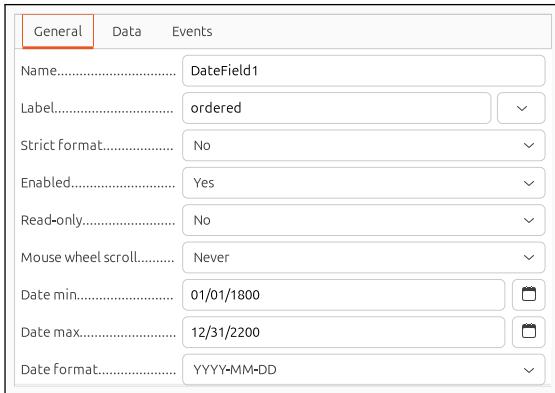
releasing the button in Figure 13.5.4. A dialog opens in Figure 13.5.5. It asks about the data we want to use in the table control. In other words, we now need to link our table *control* to a table in our *DB*. There are lots of tables we can select from. Most of them are storing some meta-data about the DB and are not for us to meddle with. All of "our" tables are available under the prefix `public`. We scroll down all the way in the view on the right side and select `public.demand`. Then we click `Next`.

In the next dialog depicted in Figure 13.5.6 we can select columns to insert. We only choose `amount` and `ordered`. As said, we want to be able to enter the `customer` and `product` field not as numbers. Therefore, we will not include them here. We will design specialized controls for them later. We click `Finish`.

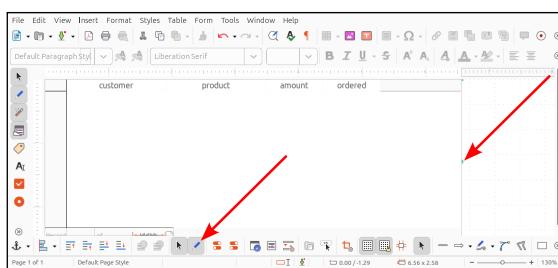
The table control is now inserted in draft form in Figure 13.5.7. It has the two columns `amount`



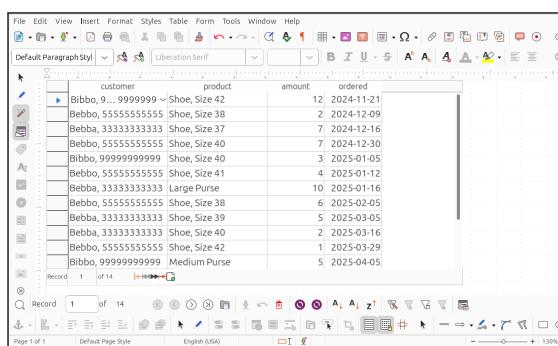
(13.5.43) In the **General** pane, we want to change the **Date format** and not use this awful short US format.



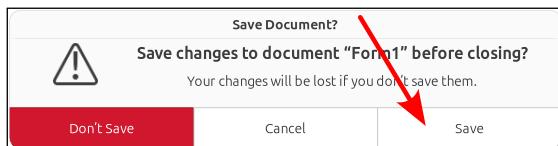
(13.5.44) We set the date format to "YYYY-MM-DD", which is the same format used in our SQL queries. We close the dialog.



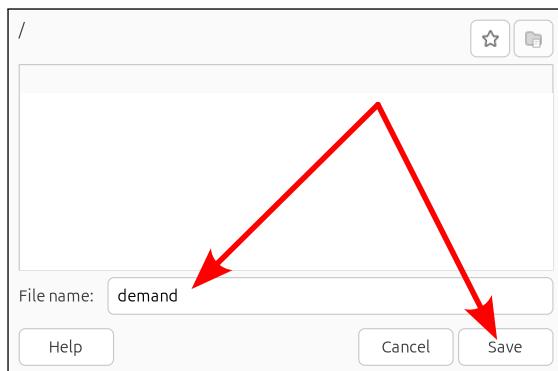
(13.5.45) We also want to make the form a bit wider. We click on the green handle in the middle of the right side of the table area and drag it outwards. After releasing it, we again click on the pencil symbol to leave the design view.



(13.5.46) The form is again displayed "in action." It looks very nice now. We are done with our work.

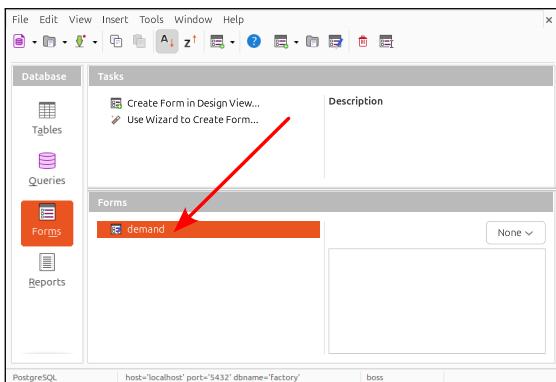


(13.5.47) We close the form. We get asked whether want to save it. We of course click on **Save**.



(13.5.48) We choose the name **demand** for our form and type this into the **File name:** box. We click **Save**.

Figure 13.5: Creating a form for entering demand orders into our DB in LibreOffice Base (Continued).



(13.5.49) We are back in the main window of LibreOffice Base. Let's click again on our new `demand` form to see whether it is still there and whether it still works.

	customer	product	amount	ordered
Bibbo, 9...	Shoe, Size 42		12	2024-11-21
Bibbo, 555555555555	Shoe, Size 38		2	2024-12-09
Bibbo, 333333333333	Shoe, Size 37		7	2024-12-16
Bibbo, 555555555555	Shoe, Size 40		7	2024-12-30
Bibbo, 999999999999	Shoe, Size 40		3	2025-01-05
Bibbo, 555555555555	Shoe, Size 41		4	2025-01-12
Bibbo, 333333333333	Large Purse		10	2025-01-16
Bibbo, 555555555555	Shoe, Size 38		6	2025-02-05
Bibbo, 333333333333	Shoe, Size 39		5	2025-03-05
Bibbo, 333333333333	Shoe, Size 40		2	2025-03-16
Bibbo, 555555555555	Shoe, Size 42		1	2025-03-29
Bibbo, 999999999999	Medium Purse		5	2025-04-05
Record	1	of 14		

(13.5.50) The form opens in all of its beauty.

Figure 13.5: Creating a form for entering demand orders into our DB in LibreOffice Base (Continued).

and `ordered` as prescribed. Now we want to add columns for `customer` and `product`. Therefore, we right-click at the left corner of the `amount` column.

In the menu that opens, click `Insert Column` and then `List Box` in Figure 13.5.8. Indeed, back to the draft of our form in Figure 13.5.9, we can see that a new column named “List Box 1” has been added. We right-click on it. In the menu that opens in Figure 13.5.10, we click on `Column...`. A new dialog opens up in which we can configure the new column. As shown in Figure 13.5.11, we first select the `General` tab in this dialog. We change both the label and the name of the column to “customer” in Figure 13.5.12. Then we click on the `Data` pane. Here we will make our column clever.

We first need to choose the data field of our `demand` table that should be set via this form field in Figure 13.5.13. We type in `customer`. Next we want to change the `Type of list contents` and `Input Required...` in Figure 13.5.14. For `Type of list contents`, we choose `[Sql [Native]]`. We change `Input Required` to `Yes` in Figure 13.5.15.

We now need to enter a `SQL` query that should reflect the content for this control by clicking on the small downward facing wedge symbol at the right of the “List content” field. This query must return two columns: The first column should contain the text that the user sees. The second column should contain the value that will be stored in the `customer` field. We could simply query `SELECT name, id FROM customer`. This means that if a user later works with our column, all they would see are the customer names. However, the values that our form would actually store would be the corresponding customer ids. We can make the whole thing a bit nicer by adding an `ORDER BY name` clause. Then, if a user would click on the customer field of our view, they could select the customer from an alphabetically ordered list.

At this point, we again remember that customer names are *not* `UNIQUE` in the table `customer`. The `phone` fields are. Thus, a user could easily confuse two customers who happen to have the same name. In our `sale` view, we solved this by concatenating the customer name with the customer phone numbers. We can simply do the same thing again. We design the query to return data in the form “name, phone”. Then, the user can comfortably work with names and sees the phone numbers, too, making each row unique. So we choose `name || ', ' || phone`, i.e., a concatenation of the name and phone number string, separated by a comma, as the first column of our query. We call this column `name_and_phone` and we modify the `ORDER BY` clause accordingly in Figure 13.5.16. We click on `OK` and close the dialog.

The name of the new column has changed to `customer` in Figure 13.5.17. We see that this column appears to be a bit small, i.e., not wide enough, for the contents that it will contain. As a small exercise, we want to make it wider. So we right-click on it again. In the menu that pops up in Figure 13.5.18, we this time click on `Column width...`. A small dialog appears. We write `4cm` as `Width` and click `OK` in Figure 13.5.19.

As the dialog disappears, we see that the column is now indeed wider in Figure 13.5.20. We now want to add the `product` column in the same way. We right-click between it and the `amount` column.

The same menu we have seen already a few times opens in Figure 13.5.21. We again click `Insert Column` and then `List Box`. A new “List Box 1” column appears (again) in Figure 13.5.22. We

right-click on it. In the menu that opens, we click on `Column...` in Figure 13.5.23.

The dialog for configuring the new column opens again. We first go to the `General` properties pane. We set both the name and label to `product`. We also set the width to 1.57 inches in Figure 13.5.24. Then we move over to the `Data` pane. We again enter a `SQL` query and therefore click on the small downward facing wedge symbol on the right-hand side of the "List content" field. This time, we select our data from table `product`. The `name` column should be displayed to the user, while the `id` column should be stored in the form. Therefore, in Figure 13.5.25, we enter the query `SELECT name, id FROM product ORDER BY name;`. After clicking on `OK`, we also need to bind the form column to an actual column of the table `demand`. The `id` from the selected `product` should be stored in the `product` field of the `demand` record. Therefore, we select `product` as *Data field* in Figure 13.5.26. We close the dialog.

The form design is now finished in Figure 13.5.27. We unselect the pencil symbol in the bottom bar. As long as this symbol is selected, we are in the "Design Mode" and can edit the form. By unselecting it, we change into the actual usage mode of the form.

This opens our form in Figure 13.5.28. It now displays the data to us in a form similar to the `sale` view. Different from this view, however, it makes sure that we get to see unique customer values.

While it looks like the `sale` view, it is a *form*, not a *view*. We can edit the data and insert new data! We scroll down to the empty row at the very bottom. We click on the small wedge symbol in the `customer` column in Figure 13.5.29.

A drop-down list opens from which we can select the customer in *clearly readable form*. In Figure 13.5.30, we choose Mr. Bibbo. We now click on the small wedge in the `product` column in Figure 13.5.31. In the drop-down list that opens, we select "Shoe, Size 39" in Figure 13.5.32. Next, in Figure 13.5.33, we can enter the amount. We enter the number 7 as amount. We move on to the `ordered` date field in Figure 13.5.34. As order date, we type `05/07/25`, corresponding to May 7th, 2025. We press . The cursor jumps to a new row as our record is inserted into the `demand` table in Figure 13.5.35.

From now on, the user can enter demand records in a much more convenient way. They do not have to look up customer ids or product ids anymore. They only need to work with human-readable names. It becomes much faster to enter data. And mixing up customers or products becomes much less likely.

However, we are a bit unhappy with the way amounts and dates are displayed. Amounts are displayed as fractional numbers, our `7` has become a `7.00`. This is not OK for us, as amounts are integers and should also be displayed as integers. Also, we find the date format confusing. The `MM/DD/YY` format is odd. We want dates to be displayed in our beloved ISO format `YYYY-MM-DD` [119].

Therefore, we want to go back to editing the form. We thus click on the pencil symbol in the bottom bar in Figure 13.5.36.

The design view of the form opens again. We right-click on the `amount` column in Figure 13.5.37. We click `Column...` in the drop-down menu that opens in Figure 13.5.38. In the `General` tab shown in Figure 13.5.39, we want to change the minimal value, the decimal accuracy, and the thousand separator setting. The minimum gets set to `0`.¹ We do not allow fractions, so the decimal accuracy becomes `0`. For the case that someone orders thousands of shoes, we would display a thousand separator. We close this dialog in Figure 13.5.40.

Next, we right-click on the `ordered` column in Figure 13.5.41. In the drop-down menu that opens, we again click `Column...` in Figure 13.5.42. In the dialog that opens, we again select the `General` pane. We want to change the *Date format* and not use this awful short US format in Figure 13.5.43. We set the date format to "`YYYY-MM-DD`", which is the same format used in our SQL queries. We close the dialog in Figure 13.5.44.

We also want to make the form a bit wider. Therefore, we click on the green handle in the middle of the right side of the table area and drag it outwards. After releasing it, we again click on the pencil symbol to leave the design mode in Figure 13.5.45.

In Figure 13.5.46, the form is again displayed "in action." It looks very nice now. We are done with our work. We close the form.

We get asked whether want to save it. We of course click on `Save` in Figure 13.5.47. We choose

¹We actually should set it to 1, but well, I entered 0 and I will not take another screenshot.

the name `demand` for our form and type into into the *File name:* box. We click `Save` in Figure 13.5.48.

We are taken back to the main window of LibreOffice Base. To see whether our new `demand` form was properly saved and still works, we double-click on it in Figure 13.5.49. The form opens in all of its beauty in Figure 13.5.50. We close it again. We are done for now.

What we achieved here is something quite nice. The strength of **relational databases** is that we can create multiple tables of interrelated data. One table can reference existing records in other tables via foreign keys. The **DBMS** ensures that the referential integrity is always maintained. However, inserting records into such tables was cumbersome. Even after we transitioned from **SQL** to the simple table-based views in tools like **Microsoft Access** and LibreOffice Base, we would still need to know the IDs of customers and products to correctly enter records into the table `demand`. This is not just complicated, but also very error-prone.

Now we can design structured forms for entering data. We only scratched the surface of this topic. We only designed one form that looked like a table. With this form, we were able to link data together much similar to our view `sale`. It became unnecessary to know customer and product IDs. Instead, we could directly work with customer and product names. We can now enter data from tables that are interlinked in a complex way very conveniently. The “We” here includes also people who do not really understand what relational databases are. With such a form, a normal person can already work.

There are many more ways to design forms. They do not have to look like tables. They may have much cooler controls, drop-down boxes, radio buttons, check boxes, etc. There are many more cool things to learn. However, the important point is this: We know that forms exist and what we can use them for. And with this, we can depart and move on to the next topic.

13.5 Reports

The last common facette in small-scale DB applications that we will consider are *reports*. A report is basically a nicely-styled document created from the data in a DB. Forms are a convenient way to enter data into a DB. Reports are a convenient way to display data from a DB. If the secretary managing the customer orders wants to compose a overview on the business transactions, then they could look at the output of our view `sale`. While its output is indeed nicely readable, it is not exactly something that we would print on paper and put on the table of manager. Now they could copy the data into a **Microsoft Word** document and then print it. However, this would mean that, everytime such overview is needed, the same copy-paste-format-print workflow needs to be applied. At the bottom line, reports automate this process. They offer automated workflows for pre-formatted documents presenting data from a DB.

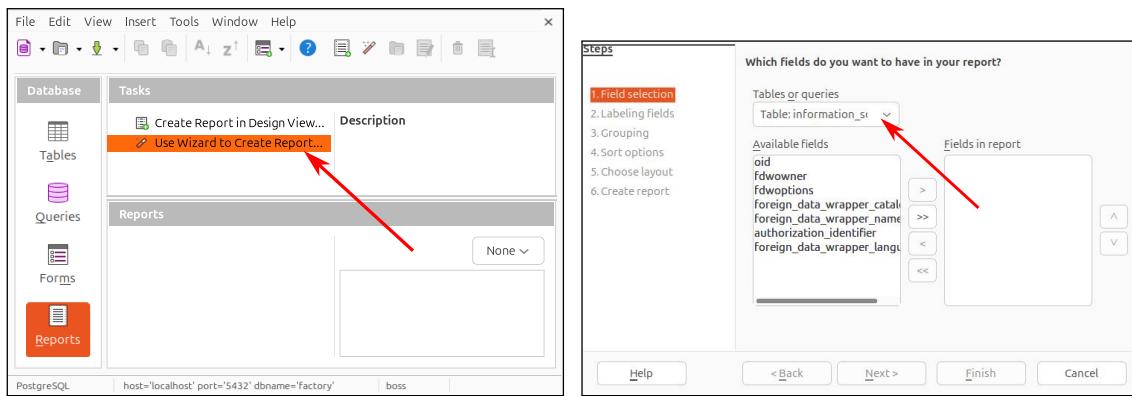
LibreOffice Base also offers the functionality to construct some basic reports. To explore feature this at least a little bit, first open our dokument `factory.odb` using LibreOffice Base. To connect with the DB, we will have to enter the password `superboss123`. Then we select `Reports` in the `Database` pane and click on “Use Wizard to Create Report...” in the LibreOffice Basemain window as shown in Figure 13.6.1.

In the dialog that opens up, we need to choose the data source for the new report. We click on “Tables or queries” in Figure 13.6.2. In the table list that pops up, we scroll down all the way and select “Table: public.sale.” We will use the data from our view `sale`, as shown in Figure 13.6.3.

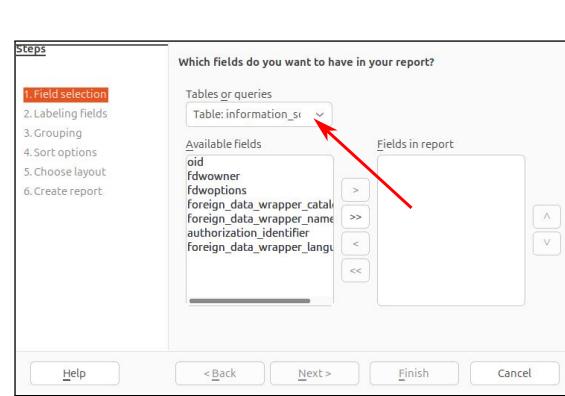
In the next step, the dialog shows us all the columns of this view in Figure 13.6.4. We select them all and click on the double greater-than button `>>` to add them to the “Fields in report” pane. Now that the columns are set up in Figure 13.6.5, we click `Next`.

Now the dialog asks us how we want to label the fields in Figure 13.6.6. The column titles of our view are pre-entered for us and while their meaning is clear, text like `customer_name` does not look nice in a printed document. We will enter some more appropriate names. After choosing some more appropriate labels and click `Next` in Figure 13.6.7.

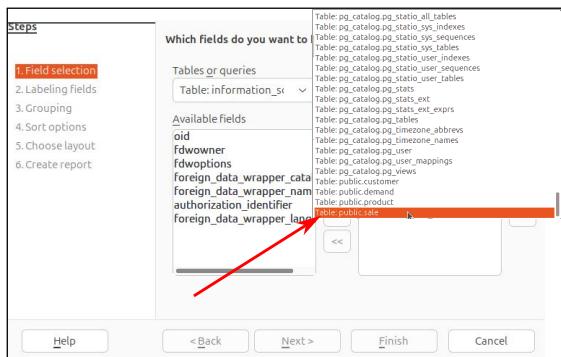
The basic building block of a report is a nicely formatted list of all the records in an SQL query. Now the rows can just be printed one by one, or we can structure the report by grouping the data based on the values in certain columns. In the next step, we can define such a division of the data into groups. We want to divide the data into one section per customer. In the `Fields` pane, we select `customer_name` in Figure 13.6.8. Then, we click the greater-than button `>` to move it to the `Groupings` pane. The column now appears in the `Groupings` pane. We click `Next` in Figure 13.6.9.



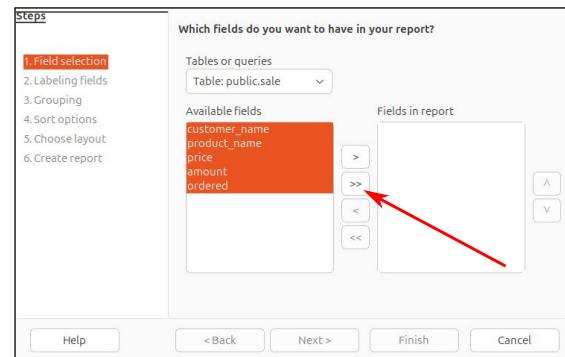
(13.6.1) We select [Reports] in the [Database] pane and click on "Use Wizard to Create Report..."



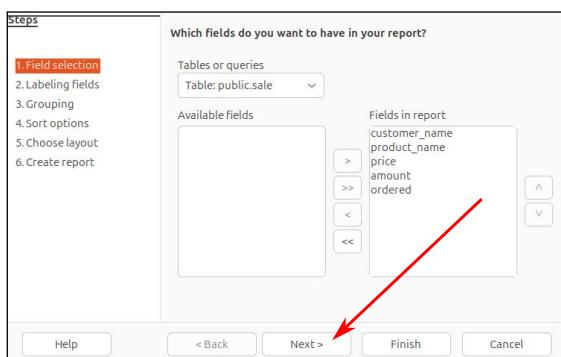
(13.6.2) We need to choose the data source for the new report. We click on "Tables or queries."



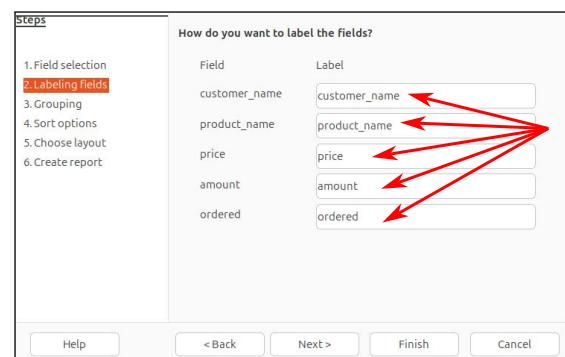
(13.6.3) In the table list that pops up, we scroll down all the way and select "Table: public.sale."



(13.6.4) We get shown all the available columns and select them all. Then we click the double greater-than button \gg to add them to the "Fields in report" pane.



(13.6.5) Now that the columns are set up, we click [Next].



(13.6.6) We get asked how we want to label the fields. We will enter some more appropriate names.

Figure 13.6: Creating and executing DB reports in LibreOffice Base.

The second aspect concerning the linear representation of the data is the ordering. Since we group the data based on `customer_name`, it will automatically be sorted by the customer names. But we can additionally sort the data based on more columns in Figure 13.6.10. So our data will be sorted by customer. However, inside each customer section, it is unsorted. This does not suit our taste. We want to order the sales demands by `ordered` date. We want the newest orders to come first, so we will choose descending order. If multiple demands are issued by a customer on the same date, then we want to break the sorting ties by the `product` name (ordered in ascending fashion). Any remaining ties should be broken by order amount. We enter this information in Figure 13.6.11 and click `Next`.

We can now choose the layout of the report. We are OK with the standard tabular look, but want the report be in portrait format. When printing documents *portrait* format means that the format is

Steps

1. Field selection
2. **Labeling fields**
3. Grouping
4. Sort options
5. Choose layout
6. Create report

How do you want to label the fields?

Field	Label
customer_name	Customer
product_name	Product
price	Price
amount	Amount
ordered	Date

Help < Back Next > Finish Cancel

(13.6.7) We have chosen more appropriate labels and click **Next**.

Steps

1. Field selection
2. Labeling fields
3. **Grouping**
4. Sort options
5. Choose layout
6. Create report

Do you want to add grouping levels?

Fields	Groupings
customer_name	
product_name	
price	
amount	
ordered	

> < ^ v Help < Back Next > Finish Cancel

(13.6.8) We now can divide the data into groups. We want to divide it into one section per customer in **Fields**. So we select **customer_name** and click the greater-than button **>** to move it to **Groupings**.

Steps

1. Field selection
2. Labeling fields
3. **Grouping**
4. Sort options
5. Choose layout
6. Create report

Do you want to add grouping levels?

Fields	Groupings
product_name	customer_name
price	
amount	
ordered	

> < ^ v Help < Back Next > Finish Cancel

(13.6.9) It appears in **Groupings**. We click **Next**.

Steps

1. Field selection
2. Labeling fields
3. Grouping
4. **Sort options**
5. Choose layout
6. Create report

According to which fields do you want to sort the data?

Sort by	Order
customer_name	Ascending
Then by	Descending
ordered	Descending
Then by	Ascending
product_name	Ascending
Then by	Descending
amount	Descending

Help < Back Next > Finish Cancel

(13.6.10) Now we can sort the data records. They will already be sorted and grouped by customer. However, but inside the customer sections, we want to order the sales demands by date, product (in case of tied dates), and amount.

Steps

1. Field selection
2. Labeling fields
3. Grouping
4. **Sort options**
5. Choose layout
6. Create report

According to which fields do you want to sort the data?

Sort by	Order
customer_name	Ascending
Then by	Descending
ordered	Descending
Then by	Ascending
product_name	Ascending
Then by	Descending
amount	Descending

Help < Back Next > Finish Cancel

(13.6.11) We have added the fields. Ordering in descending fashion means bigger values first. Ordering in ascending fashion means smaller values first. We click **Next**.

Steps

1. Field selection
2. Labeling fields
3. Grouping
4. Sort options
5. **Choose layout**
6. Create report

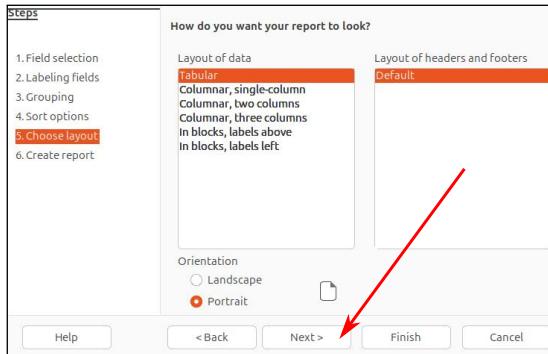
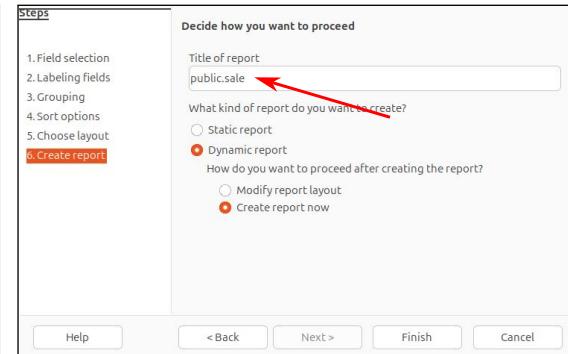
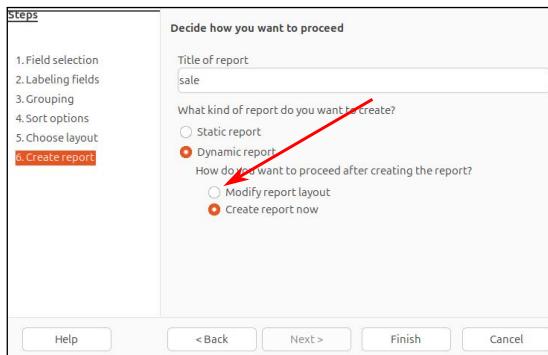
How do you want your report to look?

Layout of data	Layout of headers and footers
Tabular	Default
Columnar, single-column	
Columnar, two columns	
Columnar, three columns	
In blocks, labels above	
In blocks, labels left	

Orientation
Landscape
Portrait
Help < Back Next > Finish Cancel

(13.6.12) We can now choose the layout. We are OK with the standard tabular look, but want the report be in portrait format.

Figure 13.6: Creating and executing DB reports in LibreOffice Base (continued).

(13.6.13) After changing the layout, we click **Next**.(13.6.14) As report name, we think **sale** will be better than **public.sale**.

(13.6.15) We also do not just want to print the report right away, but we want to "Modify report layout."

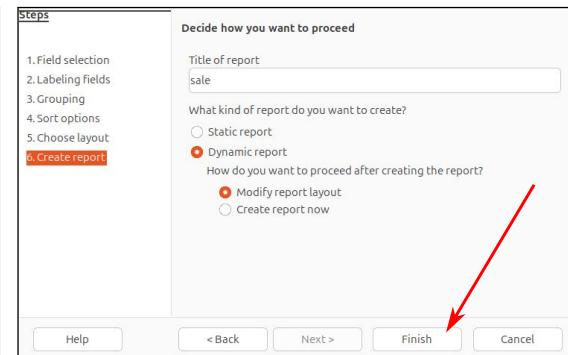
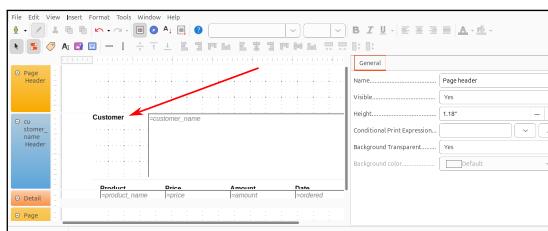
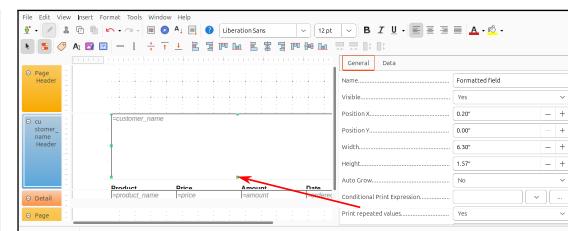
(13.6.16) We can now click **Next**.(13.6.17) The report is created and opened in design view. It looks a bit clunky. For example, the customer field takes way too much space. And it does not need a label. So we click the label and press **Del**.(13.6.18) We also dragged the left corner of the **customer** field to the left border. We now want to drag its bottom edge up, because it does not need so much space.

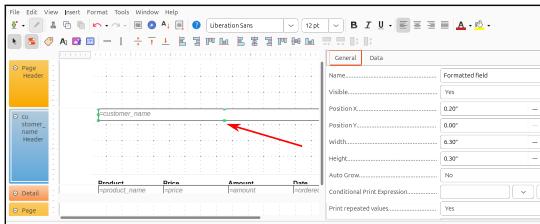
Figure 13.6: Creating and executing DB reports in LibreOffice Base (continued).

higher than wide, like the pages in this book. The *landscape* format pre-selected in Figure 13.6.12 format means wider-than-high, i.e., similar how you would paint or take a photo of a landscape. After changing the layout, we click **Next** in Figure 13.6.13.

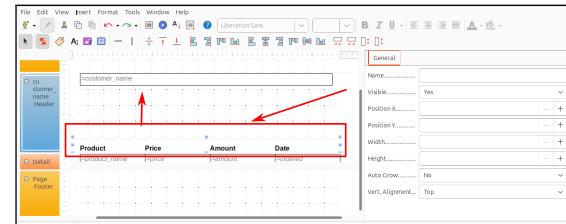
As report name, we think **sale** will be better than **public.sale** in Figure 13.6.14. We also do not just want to print the report right away, but we want to "Modify report layout" so we make the corresponding selection in Figure 13.6.15. We can now click **Next** in Figure 13.6.16.

The report is created and opened in design view. We can see all the controls and data fields placed, but it is not yet clear how the thing will look like when actually printed. Before we open it, let us make a few small changes to beautify it (you will thank me later).

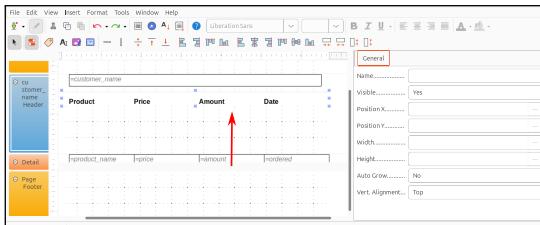
Anyway, currently, the report looks a bit clunky. For example, the customer field takes way too much space. And it does not need a label. So we click the label and press **Del**. in Figure 13.6.17. The label disappears. This creates more space on the left side of the **customer** field. So we now also dragged the left corner of the **customer** field to the left border of the report. We now want to drag its



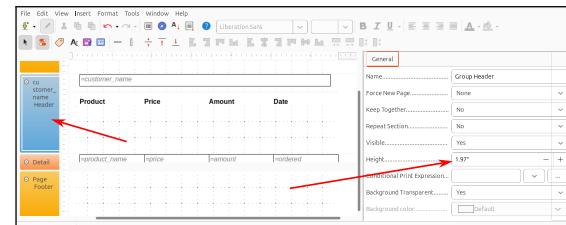
(13.6.19) Now it has an appropriate size. Next we want to move all the table headers upward.



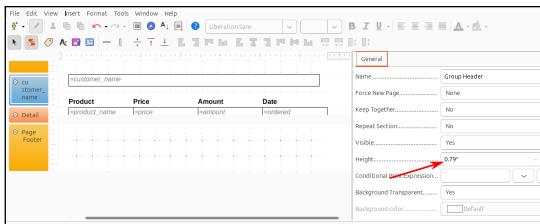
(13.6.20) We right-click and drag a selection box around the table header and the horizontal line. Then we press the **↑** key a few times to move the header up.



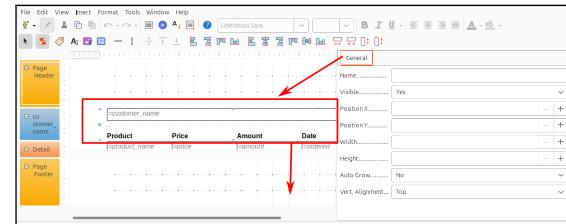
(13.6.21) The header is now moved up.



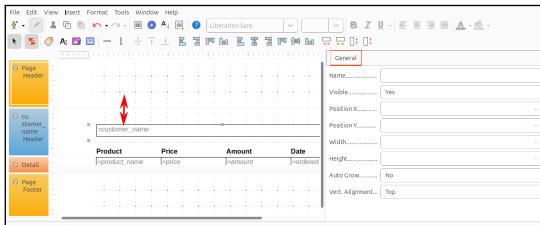
(13.6.22) We now want to make the group header field of the report a bit smaller. We click on the “customer_name Header” pane and then into the **Height** property.



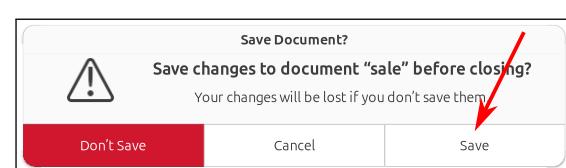
(13.6.23) We make it nice and small. Finally, to create some space between customer groups, we want to move all the header controls down a bit again.



(13.6.24) To move them all down, we select them first. We right-click into the report and drag a selection box over them.



(13.6.25) Then we press the **↓** key a few times. We are done and close the report.



(13.6.26) Upon closing the report, we get asked whether want to save it. We click on **Save**. We save it under the name **sale**.

Figure 13.6: Creating and executing DB reports in LibreOffice Base (continued).

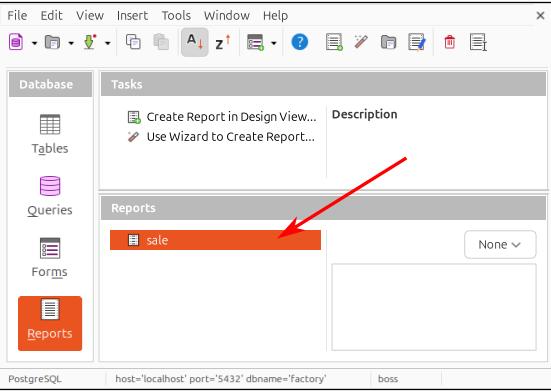
bottom edge up, because it does not need so much space in Figure 13.6.18. Now it has an appropriate size, but lots of useless space is reated below it.

Next we want to move all the table headers upward in Figure 13.6.19. We therefore right-click and drag a selection box around the table header and the horizontal line. Then we press the **↑** key a few times to move the header up in Figure 13.6.20. The header is now moved up in Figure 13.6.21.

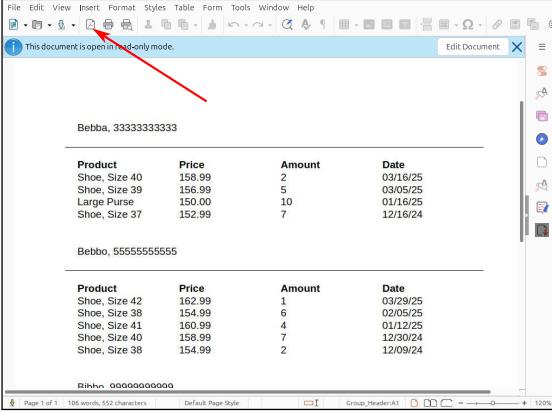
We now want to make the group header field of the report a bit smaller. We therefore click on the “customer_name Header” pane on the left-hand side. We then click into the **Height** property in the form on the right in Figure 13.6.22. We make it value nice and small in Figure 13.6.23.

This, however, will make the report look cluttered because now all the data and groups will stick together. Finally, to create some space between customer groups, we want to move all the header

(13.6.27) And it appears under this name in the Reports pane. We double-click on it.



(13.6.28) A new document opens in LibreOffice Writer. It contains all the data in a very nicely formatted fashion. We can even export it to Portable Document Format (PDF) by clicking the PDF symbol .



Product	Price	Amount	Date
Shoe, Size 40	158.99	2	03/16/25
Shoe, Size 39	156.99	5	03/05/25
Large Purse	150.00	10	01/16/25
Shoe, Size 37	152.99	7	12/16/24

Product	Price	Amount	Date
Shoe, Size 42	162.99	1	03/29/25
Shoe, Size 38	154.99	6	02/05/25
Shoe, Size 41	160.99	4	01/12/25
Shoe, Size 40	158.99	7	12/30/24
Shoe, Size 38	154.99	2	12/09/24

Bibbo, 999999999999

Product	Price	Amount	Date
Shoe, Size 39	156.99	7	05/07/25
Medium Purse	120.00	5	04/05/25
Shoe, Size 40	158.99	3	01/05/25
Shoe, Size 42	162.99	12	11/21/24

Bobbo, 444444444444

Product	Price	Amount	Date
Shoe, Size 42	162.99	11	04/12/25

(13.6.29) The exported PDF document.

Figure 13.6: Creating and executing DB reports in LibreOffice Base (continued).

controls down a bit again. To move them all down, we select them first in Figure 13.6.24. We right-click into the report and drag a selection box over them. Then we press the  key a few times. This creates the needed space in Figure 13.6.25. We are done and close the design view of the report.

Upon closing the report, we get asked whether want to save it. Of course we want to. We click on  in Figure 13.6.26.

The report now appears under this name in the  pane. We double-click on it in Figure 13.6.27 to finally see how it looks like in action.

A new document opens in LibreOffice Writer. It contains all the data in a very nicely formatted fashion. We can even export it to PDF by clicking the PDF symbol  in Figure 13.6.28. The exported PDF document is shown in Figure 13.6.29.

This looks quite nice. Of course, we just quickly clicked this report together. There is much more that can be done.

Reports often support the ability to compute and present statistics. We could present the total sales income per customer, for example. We could just as well as the display overall total income of our company in the report. It is also often possible to define filters for the data. We could probably have limited that data to with a start date and end data that the user should enter.

Reports often also allow us to include diagrams. It is totally possible to include a chart per customer showing when they made purchases and for how much money. LibreOffice Base has this functionality ... but my version of LibreOffice Base on computer crashes when I try to use it... Well, you can play around a bit and see if you get it to work on your machine.

Either way, the important point is that you now also got to take a glimpse on what reports are in the field of DBs. Reports provide us the ability to automatically generate nicely formatted and pritable views on the data in our DB. Their output can be understood by people who do not even know what a DB is. Therefore, they are a very common tool in many small and mid-scale applications. They are supported by tools such as LibreOffice Base or Microsoft Access. There exist whole software libraries that can generate reports. For example, there probably are Python libraries that can produce beautiful reports, maybe in conjunction with Matplotlib and all the other functionality that the Python ecosystem can offer us. And since you already learned how to access a PostgreSQL DB from Python, you can also roughly guess how you would get such a report library to work with our DB.

Useful Tool 4

LibreOffice Base [160, 385] offers us a simple GUI that can connect to a DBMS and provides capabilities such as executing SQL queries as well as designing and executing forms and reports.

Chapter 14

Cleanup After the Example

We are now approaching the end of this brief journey crisscrossing the domain of relational databases. We obtained a rough impression of what they are and how they can be used. To finish it, i.e., to conclude our factory example, let us delete all the things we created. Of course, we have to do that in the opposite order in which we created them.

```
1 /* The syntax for deleting elements: Using the 'DROP' Command.
2 /*
3 * We can only delete elements that are not referenced by other
4 * elements. Users can only be deleted if they do not own anything
5 * anymore. Tables can only be deleted if they are not part of any
6 * foreign key constraint, not used in any view, etc. And so on.
7 */
8
9 -- Delete table "table_name": Fails if the table does not exist.
10 DROP TABLE table_name;
11 -- Delete table "table_name": It is OK if the table does not exist.
12 DROP TABLE IF EXISTS table_name;
13
14 -- Delete view "view_name": Fails if the view does not exist.
15 DROP VIEW view_name;
16 -- Delete view "view_name": It is OK if the view does not exist.
17 DROP VIEW IF EXISTS view_name;
18
19 -- Delete database "database_name": Fails if the DB does not exist.
20 DROP DATABASE database_name;
21 -- Delete database "database_name": It is OK if the DB does not exist.
22 DROP DATABASE IF EXISTS database_name;
23
24 -- Delete user "user_name": Fails if the user does not exist.
25 DROP USER user_name;
26 -- Delete user "user_name": It is OK if user does not exist.
27 DROP USER IF EXISTS user_name;
```

The SQL syntax for deleting various DB objects is rather easy [143–146]. You write `DROP` followed by the object type followed by the object name. The object type could, e.g., be `TABLE`, `VIEW`, `USER`, or `DATABASE`. These commands will always fail if the object is still in use, i.e., if it is referenced by any other object. You cannot delete a table that is part of a foreign key constraint, for instance. The command will also fail if the object does not exist. This can be circumvented by inserting `IF EXISTS` between the object type and the object name.

In our case, we do know that the elements that we want to delete do actually exist. So why would we explicitly mention this `IF EXISTS` option? Because it has some interesting use cases. If we wanted to create a backup copy of our entire DB, then one thing we could do is to export the whole DB as a single large SQL script [413]. This script could, basically, be the concatenation of all of our listings, with the `CREATE ...` and `INSERT INTO...` commands and all. Running this script would re-create the DB in exactly its present state.

Listing 14.1: User `boss` deletes all tables and views inside the database, in the inverse order of their creation. (stored in file `cleanup_inside_database.sql`; output in [Listing 14.2](#))

```

1  /* Cleanup after the example: Delete all Tables and Views. */
2
3  -- This must be run inside the database, by user 'boss'.
4
5  -- Delete the views.
6  DROP VIEW IF EXISTS sale;
7
8  -- Delete the tables, in the inverse order of creation.
9  DROP TABLE IF EXISTS demand;
10 DROP TABLE IF EXISTS product;
11 DROP TABLE IF EXISTS customer;
```

Listing 14.2: The stdout of the program `cleanup_inside_database.sql` given in [Listing 14.1](#).

```

1  $ psql "postgres://boss:superboss123@localhost/factory" -v ON_ERROR_STOP=1
   ↪ cleanup_inside_database.sql
2  DROP VIEW
3  DROP TABLE
4  DROP TABLE
5  DROP TABLE
6  # psql 16.12 succeeded with exit code 0.
```

Listing 14.3: Delete the database `factory` and the user `boss`. This must be executed by the administrator account `postgres`. (stored in file `cleanup_database_and_user.sql`; output in [Listing 14.4](#))

```

1  /* Cleanup after the example: Delete the Database and User */
2
3  -- This must be run by the administrative user 'postgres'.
4
5  -- If the database 'factory' exists, we delete it.
6  DROP DATABASE IF EXISTS factory;
7
8  -- If the user 'boss' already exists, we delete it.
9  -- We can only delete the user after all objects associated with it,
10 -- e.g., the databases, have been deleted.
11 DROP USER IF EXISTS boss;
```

Listing 14.4: The stdout of the program `cleanup_database_and_user.sql` given in [Listing 14.3](#).

```

1  $ psql "postgres://postgres:XXX@localhost" -v ON_ERROR_STOP=1 -e
   ↪ cleanup_database_and_user.sql
2  DROP DATABASE
3  DROP ROLE
4  # psql 16.12 succeeded with exit code 0.
```

Except that sometimes, it won't. If the DB already exists, it will fail. Because we cannot create an object that already exists. Or maybe if we had a crash or a user issued some faulty SQL commands that wreaked havoc. Then the DB maybe now exists only partially. Maybe some tables or views have disappeared, maybe some rows in some tables are missing. One method to make the “backup SQL script” robust to deal with such issues is to add `DROP ... IF EXISTS` clauses *before* the commands for re-creating each table or view. Then we can restore the DB or parts of the DB even if the DB still exists or partially exists.

Anyway, let us now use these commands to clean up the DBMS at the end of our example. We want to remove all the objects that we have created. In [Listing 14.3](#), we first delete the view `sale`. We can do this by the command `DROP VIEW sale;` [146]. The `DROP` is the SQL command for deleting

things. The `VIEW` is specified to make clear what type of object we want to delete. This prevents us from accidentally deleting something else. Then we give the name of the view to delete, in our case, that is `sale`.

We make this command a bit more handy: We insert an `IF EXISTS` inbetween the object type and the view's name. This condition is self-explanatory: If a view of the provided name exists, then it is deleted. If not, then nothing happens. Without the `IF EXISTS`, this case would cause an error. With it, we simply don't need to care. We can issue the command twice and it will be OK.

Anyway, we use the same method to delete the three tables. We issue the commands `DROP TABLE IF EXISTS demand;`, `DROP TABLE IF EXISTS product;`, and `DROP TABLE IF EXISTS customer;` [144]. Notice that we delete table `demand` before we delete the tables `customer` and `product`. This is because of the `REFERENCES` constraints. We cannot delete an object that is still referenced. As long as these constraints exist, tables `customer` and `product` are in use and cannot be deleted. By deleting table `demand` first, the constraints also disappear. Tables `customer` and `product` are no longer referenced by other objects and can be disposed of.

All of these deletion steps are done by using the user `boss` and their password `superboss123`. Let us finally also get rid of the entire `DB` and of that user as well in Listing 14.3. First, we delete the DB by executing `DROP DATABASE IF EXISTS factory;` [143]. Then we remove the user via `DROP USER IF EXISTS boss;` [145]. Notice that we log in as the `DBA` user `postgres` to do that. With this, all remains of our example are gone from the DBMS server.

All the objects that we can create in a DBMS can be deleted as well. Deleting DB objects is something that we do not do very often. But in several situations, like the restoration of data via backup scripts, it comes in handy.

Chapter 15

Summary

With this, we have reached the end of our simple introductory example.

What did we learn? First of all, we got some hands-on experience using one of the world's leading DBMSes, PostgreSQL. We connected to the PostgreSQL server using the psql client software. We issued commands in the SQL language to the DBMS.

What kind of commands did we issue? Well, we created a new user (or role) and a DB. We created tables inside the DB. We issued queries to insert data into a table and to read data back from a table. We used queries to join data from different tables. We created a view and we built queries on top of that view. We learned how to modify the data in tables. Finally, we deleted everything again. This means that we have seen several of the most important SQL commands. Surely, we only have played with them. We are not even close to really understand their full behavior, special cases, performance issues, nor do we have a clear picture of what happens behind the scenes.

But one thing is clear: If somebody would come and ask us to create a DB for some certain application, we could probably do it. We could stitch together something that does the trick. Would it be an efficient or elegant DB? Definitely not. We did not yet learn anything about how to design nice DBs. But we now already have a bit of experience. We do have some raw and unrefined knowledge and ability in this field.

Even more so, we also understand how to access a DB from outside, from a programming language like Python. Based on what we learned, we have a rough feeling of what a DBMS can and probably cannot do with SQL. This allows us to, in principle, develop ideas for even more complex applications. The things that the DBMS can do should go into the DB. The things that it cannot, like displaying forms, processing data from some source like sensors or files, or training ML models on the data would go into the Python program code. The two parts of our application would communicate via a library like psycopg2.

Then again, maybe we are a small-scale business where only three or four people have to actually work with the DB. In such a situation, we probably still want the power of PostgreSQL. But making a whole separate program to work with the data may just not be worth the effort. Plugging a GUI like LibreOffice Base in front of the DB may then be fully sufficient. At least entering, viewing, and changing the data will be much much easier compared to using the psql client. It also does no longer require any understanding of SQL to work with the data. While we, the DBAs, can still use the full features of PostgreSQL, the secretary in the sales office can now enter new customer data and orders in a way much more natural to them.

For entering data into tables that have complex relationships with other tables, we learned about *forms*. Forms allow us to deal with the problem of fields that reference rows in other tables via their primary keys. A customer, for example, is represented only by their `id` value in our `demand` table. Such value is basically meaningless to a salesperson and they would need to look it up in the `customer` table to enter the right value. We can, however, make forms where this looking-up is automated: The secretary entering the data sees the real customer name but what is stored is the customer `id`.

The second abstract tool we learned about are *reports*. While it is nice to read and process data in either the table view of LibreOffice Base (or Microsoft Access for that matter), this is not the kind of format we would use when printing the data. A form is also not suitable for this purpose, because forms are designed for entering data. Reports are the tool exactly designed for this use case: They allow us to create nicely formatted documents that are automatically filled with data whenever we execute them.

After reading this part of our book, you now have seen several of standard tools and abstractions when working with DBs in action. You do not have any in-depth or theoretical foundation ... but probably a good hunch of what is what and what tool may be suitable for which kind of question. At this stage, I hope that your curiosity is tickled. Maybe you even have some problems or application ideas for which you might want to design your own DB. Maybe to catalog your books or music collection, maybe to construct your ancestry tree, maybe to store your bibliographic references. Nothing will help you more in learning about DBs than doing your own little pet projects. The second-best thing you can do is to read the rest of this book (or maybe any other book).

Part III

Database Design and Modeling

In the introductory factory example in the previous book part, we had some idea about how a DB should look like. Then we just implemented it. Of course, this is not how that works. In the real world, DBs are much more complicated. There are not just three tables. The interactions between objects, processes, and people are much more complex. We now do have some rough idea about what kind of technological tools we have available to work with DBs. We have a good handle on the things that can be done and how they could be done. But we have very little understanding of the practical process of DB design. We do not yet know how we would realize a real DB project in a professional manner.

Imagine that we were people who want to build cars. We know what wheels are and how they work. We know what the battery does and how we can use it to make the motor run. We know that normal people want to sit on seats inside the car and how to install seats. We know what a chassis and a car body are, how they can be constructed, and how to put in the other parts. Yet, we cannot really construct roadworthy vehicles yet. Because this is not a puzzle game where we try to step-by-step plug parts into places where they fit until all the parts are included. This involves a properly documented design process. We need to first make a clear plan of what goes where and when to do what. Only after the planning, the construction can begin.

Similarly, DBs are designed in a methodical and systematic manner. We start with an analysis of the requirements and step over increasingly precise models of our planned application. This is what we want to learn in this part of the book. At its core, the process of designing a DB will involve the creation of three models (also called schemas) [157]: the conceptual model, the logical model, and the physical model.

Definition 15.1: Conceptual Model

The *conceptual model* (or *conceptual schema*) of a DB is a model of the real-world entities about which the system stores data as well as the relationships between them.

The conceptual model is the high-level view on our application scenario. The purpose of this model is to provide a clear concept of the application that is understandable by project stakeholders. Such a model can be seen as a formalization of the the data-related part of the requirements. It is also a vital part of the project documentation. It is independent from any concrete data model or DB technology. It represents entities, the attributes of entities, as well as the relationships among them.

Definition 15.2: Logical Model

The *logical model* (or *logical schema*) of a DB is a model of the data grounded to a specific type of technology (e.g., relational databases, hierarchical DB, NoSQL DB, ...). It represents the entities, their attributes, and relationships as well as constraints using concrete datatypes and a structured, formal format.

The logical model is the level on which the users and applications interact with the DB. Here, the conceptual model is translated to a technical specification. This model is often specified in a formal DB language like **SQL**. It may be grounded to a specific **DBMS** or type of DBMS.

Definition 15.3: Physical Model

The *physical model* (or *physical schema* or *internal schema*) is the specification of the concrete technological realization of the logical model. It is bound to a concrete DBMS and specifies exactly how the data is stored and accessed. The physical model impacts performance and system requirements, but not the way users and applications access the DB (because that is defined by the logical model).

In many smaller DB applications, the logical model can be used as physical model. The logical model may be specified in a language like SQL. In this case we have a clear definition how tables are created and accessed via queries and views. From a **CREATE TABLE**-command, for example, most normal DBMSes can already infer proper default settings for storing the data.

However, to achieve high performance, the physical model can add further specifications: How should large records be stored? Should the data be sorted in any particular way? Or should there be indexes, i.e., additional data structure for speeding up certain queries?

These specifications are invisible to the applications and users. They access the DB by using, for example, **SQL**. Their queries do not change in any way if the physical model changes. However, the physical model has an impact on the query performance and the storage requirements of the data. For larger DBs, this can make a huge difference.

To a good share, the DB design process is concerned with getting these models correct. There are different challenges at each step, from discussing with the future users to fine-tuning a **DBMS**. We will explore these steps on a second example. We will look a bit more closely at the process and the steps involved in creating a reasonably elaborate DB application. As overarching example, we want to design a DB for managing students, teachers, and courses in a university.

Chapter 16

The Database Lifecycle

The design of DBs is not a straightforward process of “idea → design → finished.” Instead, designing good DBs involves several steps. These steps are follow a DB lifecycle.

We need to clearly understand the requirements for the DB. We should create a rough sketch of what things will be included. We need to decide what tables and relations should exist. This conceptual model (information about the types of data and their relationships) must ultimately be mapped to a physical model (concrete implementation instructions for a selected DBMS). There should be some sort of prototype, where we and the users can enter a subset of the data to test whether everything works as predicted. At this stage, we may uncover some problems and may need to improve our design. The actual application must then be tested. Then, once we are happy with everything, the application and DB enter the productive environment. Then the DB needs to be maintained and backed up regularly.

Due to the longevity of DBs, eventually, new features and changes may become necessary, at which point we may need to revise our design. Interestingly, differently from normal software projects, DB applications do not seem to grow in complexity during their evolution [405], but they do indeed evolve over time after their initial development. There exist several general approaches on how to manage or conduct the DB design process, how to bring this process into some structure [191].

DBs are software artifacts, so we could use one of the many available **Software Development Life Cycles (SDLCs)** [225, 298]. Yet, DBs are also special. They are objects with a comparatively long lifetime [380]. The hardware in your computer stays the same maybe for three to five years. Software programs often change every five to ten years. Every two years, a new **long-term support (LTS)** version of the **Ubuntu operating system** comes out [444]. **LibreOffice** releases every six months [434]. Major versions of **Microsoft Windows** come out every two to six years [173]. Once new major versions of such important software appear, the old versions usually fade out of support within five years and need to be replaced.

DBs, however, may stay in use for several decades. Indeed, a long time ago, I designed a DB that was used by a branch office of a midsized company for well over a decade. DBs and the applications built on top of them are important assets of an organization. They contain valuable information, both historical data as well as the data required by the current operation.

DBs are not just used to store data, the data they mirror real-world entities and processes. In our small factory example, the data mirrors the products, the customers, and the interactions of the customers with the factory. Any of them may well change over time. For example, maybe the company eventually changes from selling products with a fixed name and configuration to configurable products. Maybe one day a customer can decide the color and size of the shoes they want to order together with the cloth to be used as well as the material of the sole. This could yield so many possible combinations that the current way to store products is no longer feasible. Maybe the scope of the DB is eventually expanded to also keep track of the product stock in the warehouse. DBs may exist over many years and may evolve during their lifetime.

All of the above together lead to several demands on the DB development lifecycle. This lifecycle is very similar to the SDLC. It often even is either the backend or the foundation of software development. However, there are two aspects that make the DB development special: First, the longevity of DBs. Second, there is the “low-levelness” of DBs. The DB is the very foundation upon which other applications like reports, forms, and websites are developed. Therefore, they are the first point of contact between project stakeholders and developers. The DB development process is where the basic understanding of the data and processes in an organization is built. This is where the situation is most

uncertain, where most of the misunderstandings will happen.

The foremost and obvious requirement to the DB design process is that it guides us to translate the information from the stakeholders to a fully functional and running DB application. It should allow us to plan how to do the project, i.e., to create a timeline defining when we will reach which milestone, how many work hours may be needed, and how much things are going to cost. But these things are all affected by uncertainty.

More often than not, the stakeholders are initially not entirely clear about their processes and data. During the first few discussions about what the DB should store and do, several important details may be missed. For example, "Each product has a name." seems to be a reasonable statement when designing a DB for a factory. The DB designer thus may just assume that this is true and not double-check or discuss this aspect in-depth. However, sometimes we may be in for a surprise, maybe the following could happen: "Ah, indeed, we use different product names for different customers. This customer, for example, is a big company that uses our screws to build cars. When we offer them our 'Screw 3B' we call it 'double-inch screw,' because this is how they refer to it in their production processes." Obviously, such a situation could not be captured with a single table per product anymore.

Also, many real-world processes are not entirely formally specified. Processes that modify the data may have a very clear core, e.g., "We send the product to the customer's address once they paid." But there might be fuzzy edges, like "Very few of our customers have been with us for many years or buy lots of products, they can pay after receiving the product." If we design a DB for managing students, it may be that the school says: "A student can repeat an exam at most twice after failing it." In reality, there might be special circumstances under which some students may be granted a third trial, maybe a student arrived late to an exam due to traffic and was marked as failure, but the exam commission decides to give them another chance. Therefore, problems may be discovered early in the DB design process but could just as well be encountered a year after the DB application has been deployed.

Sometimes, the customer may also just assume that some things are common knowledge. "Of course, for customers with a delivery address outside of the European Union, we add an export tax. All businesses do that. Everybody knows that." The DB designer may not know that, though.

Therefore, a DB design process should also allow us to progressively enhance our design in response to uncovered issues [191, 483]. At the same time, it should avoid scope creep, i.e., a situation where progressive enhancements keep modifying the project structure, adding more and more features, and drifting away from the originally planned project [191, 483].

16.1 Classical Software Engineering Design Processes

Several different design methods, i.e., SDLCs, have been proposed in the field of software engineering. The simplest one is probably the waterfall model [191, 225, 298, 342, 369] published in 1970 by Royce. An adoption of this model to DB development could look like Figure 16.1. The waterfall model is a sequential process of project planning, requirements definition, design, development, testing, and installation and project acceptance. Each step produces deliverables as output which become the input for the next phase. The waterfall model specifies which activities should be carried out in each phase as well as the deliverables that should be produced. During the requirements definition phase, for example, the project scope is restricted based on the discussions with the stakeholders. The waterfall model is simple and easy to implement. It allows the developers to work their way along a well-known

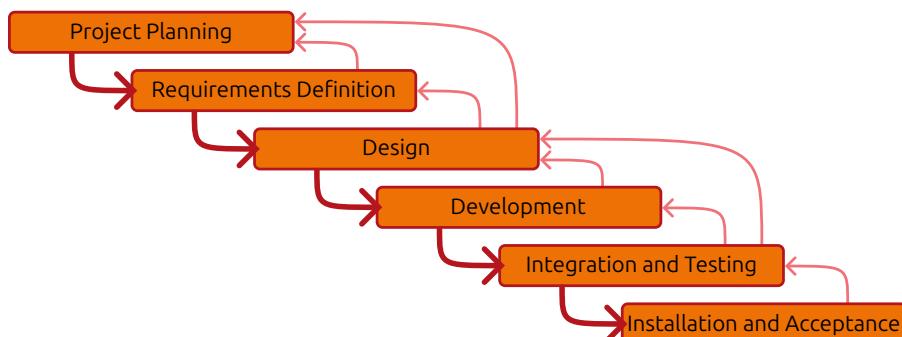


Figure 16.1: The waterfall model [191, 225, 298, 342, 369].

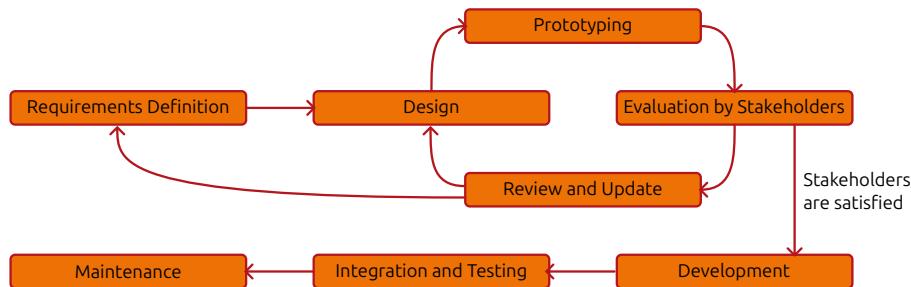


Figure 16.2: The prototyping model [191, 275, 329].

structure. At each phase of the project, we know where we are standing and there thus will be few misunderstandings.

This model may sometimes be misunderstood as being strictly sequential. If we indeed were to follow the steps sequentially, then we would fail to anticipate that we may make errors in each phase, that there can still be misunderstandings, and that we may still discover unexpected issues. However, Royce explicitly acknowledges that feedback between succeeding and preceding phases takes place. Sometimes even larger changes are necessary, e.g., when the testing phase does not yield the anticipated success. One idea already proposed in [369] is to do the whole design and development process twice for entirely new applications. The first time would be a brief simulation phase, maybe with a fourth of the scheduled project time. Its purpose would be to gather the experience that will then lead the second iteration of the process to success. I personally am a fan of this.

Either way, a weakness of the model may be that the requirements are specified early in the process and there is no predefined mechanism to introduce changes later in the project.

Another idea or maybe a formalization and extension of the “do it twice” concept is (rapid) prototyping [275]. Here we embrace that both the stakeholders and the developers do not clearly know what the final product should look like. We put a cycle of prototypes and discussions into the center of the development process [40, 275]. This prototyping process is illustrated in Figure 16.2. First, a limited version of the requirements are collected and a prototype of the DB and software is designed right away. The prototype may even be a mockup which only shows the basic functionality of the project and does not interact with other tools [40]. It serves as basis for discussions and we may need to redesign it a few times. In some variants of this approach, the requirements may be finalized early on [191, 329] and in others they can be updated later on [275]. The prototyping model provides the ability of iterative enhancement of the project specification. In turn, it sacrifices some clarity of process sequence.

The spiral model [44, 45, 191, 329] illustrated in Figure 16.3 could be considered as another take on the multi-pass waterfall model. Here, the development process begins with the requirements analysis and a development plan. An initial pass through a standard waterfall lifecycle based on a subset of the requirements is performed to develop a first prototype. The prototype is evaluated and then the cycle begins again in order to add new functionality and to create the next prototype. In each iteration, a risk analysis is performed before the prototype is developed. Each new prototype thus helps to reduce the risks.

One idea of the spiral model is that requirements are of hierarchical nature. In each iteration, additional requirements are built on top of the first set of requirements implemented. This may not be very suitable when we design a DB, where different functions can be more or less independent from each other while using the same data (e.g., keeping track of stock in a warehouse and booking customer orders).

The problem to be solved is defined at the project start and therefore the project scope is limited. However, since the requirements are finalized early, the model does not encourage progressive enhancement of the design. Also, the risk analysis steps can be quite complex and so is the overall structure of the spiral model, which are downsides of the method [191, 374].

Rapid Application Development (RAD) [38, 283], as illustrated in Figure 16.4, is similar to prototyping. The user gets to try the (prototypic) application *before* it is delivered [191, 287, 374]. If stakeholders can play around with a live system, then they can probably give much better feedback compared to a situation where they only work with specification documents. Thus, a prototype is created and installed as soon as possible and made available to the user. RAD-projects are often implemented by small, highly-collaborative teams and within a short time frame. The projects are very interactive

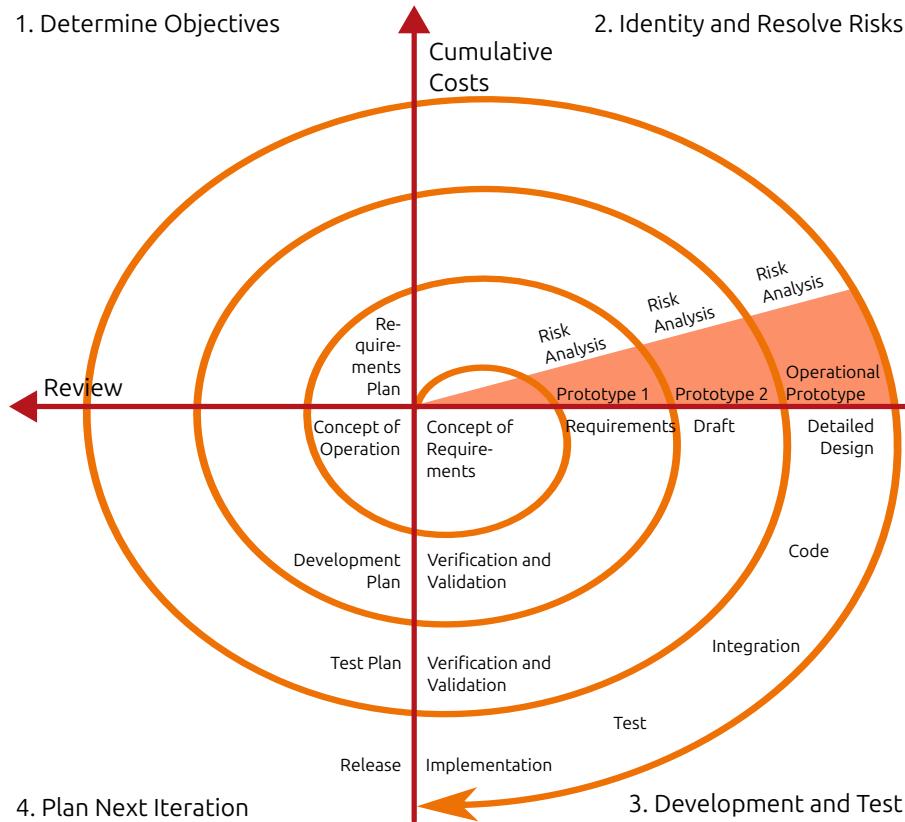


Figure 16.3: The spiral model [44, 45, 191, 374].

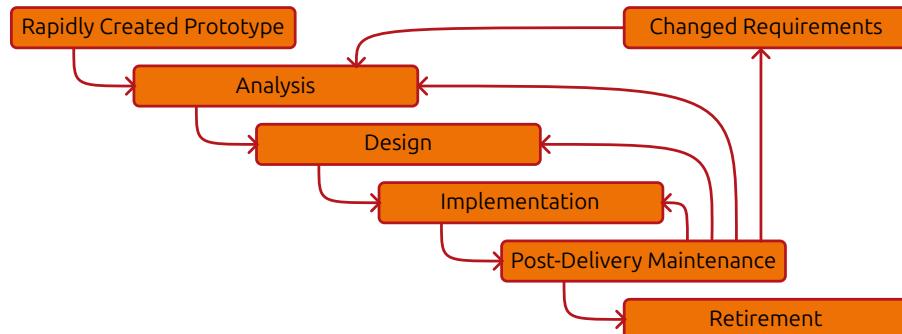


Figure 16.4: The Rapid Application Development (RAD) model [191, 225, 283, 298, 374].

and often of comparatively low complexity. RAD therefore also introduces a social component and team building into software development [38].

RAD-based projects tend to have a lower level of rejection when the application is placed into production. The downside is that they are also likely to exhibit scope creep: The stakeholders perceive modifying the application as easy and therefore are more likely to ask for more features. The final goal of having a fully operational product may drift out of focus and the projects may overdraw budget and schedule.

16.2 Databases Design Processes

DB systems differ from normal software applications. First, they usually are the foundation for several software projects. A student management DB for a university, for example, may offer the students to log in and join modules or view their grades. It may also allow the administrative staff to create new modules and even manage the module structure of curricula. It may furthermore handle the room planning for classes. And it may even help to manage the important dates in the semester, such as times for exams, the schedule for students. These things may be handled by different applications with

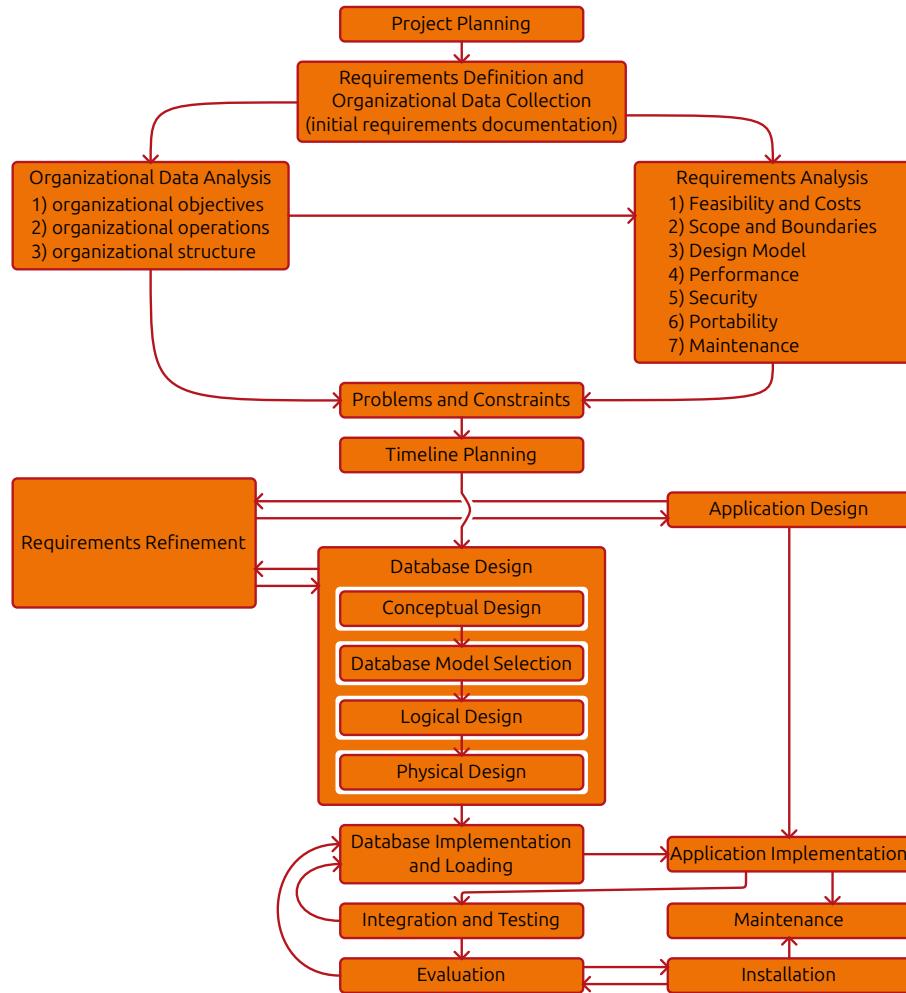


Figure 16.5: The lifecycle model by Gupta, Mata-Toledo, and Monger [191].

different user interfaces. But all of these applications would access the same DB backend.

Second, we normally want to incorporate the development succession over the three schemas: We will at some point develop the conceptual schema modelling the real-world entities and processes. Then we derive a logical schema by mapping the conceptual schema to a concrete technology. Finally we may introduce a physical schema with specific performance-improving configurations.

Thirs, we must keep on our radar that **DBs** are long-living artifacts. They must be managed, maintained, and improved over many years.

As a result, a wide variety of design processes and life cycle management structures for DBs have been developed. They often are adaptations of different **SDLC** models and can offer different levels of detail to the project managers.

Personally, I like the comprehensive approach proposed by Gupta, Mata-Toledo, and Monger [191] and illustrated in Figure 16.5. In it, we find many activities that a good DB architect needs to consider. In this model, the lifecycle begins with a planning stage. Initially, there often are too many uncertainties to lay out a realistic timeline for the project. Therefore, the goal of this first phase is to plan two activities: (1) the collection of the necessary information about the organizational processes that should be mirrored by the DB application and (2) the requirements analysis. The result of this phase is a plan document.

In the next stage, we collect all the necessary data and information about the organization. The developers and designers interact with all stakeholders in the project at all levels, from the intended users of the DB applications up to the management of the organization. The documents and processes in the organization are explored, as well as the flow of data through the organization. We check which systems and frameworks already exist, what input they require, what output they produce, who uses them and how and why. Interviews and questionnaires can be used to gather more data. The activities

of personnel at all levels of the organization can directly be observed and documented. The present needs and potential future expansions of the DB project are documented. A software requirement specification is produced.

In the requirements analysis, we can now analyze the collected data to find out whether the project is feasible and to approximate the costs. The goals of the project and the objectives of all proposed systems are specified. The scope and boundaries (budget, equipment, available software, ...) of the project are defined, as well as requirements regarding performance, security, portability, and maintenance. All of these issues are very important for both the short term and long term success of the project.

As the result, potential problems and constraints that could arise later will be identified. After the requirements collection and analyses are completed, a timeline for the rest of the project can be established.

DBs and the applications working on the data cannot be entirely separated. The DB and the application(s) are therefore designed as two parallel strands of a project. Both process update each other. The internal DB design block then follows the steps also given in [157]. The first step of the DB design is to create a conceptual design, i.e., a high-level and technology-independent overview of the DB. The goal is to outline the big picture at an abstract level. The model is based on the worldview and processes of the stakeholders. This design can be visualized using ERDs [23, 484]. It is common to *not* just draw one single huge ERD, which would be overwhelming and hard to understand. Instead, often, several smaller and easily readable ERDs are drawn with no more than ten or so entity types [484].

Then, the data model is chosen. In the context of this book, our focus is on relational databases, where data is stored in tables that can reference each other. This may not always be the right choice. For example, maybe we have to deal with images or video data, maybe we have to deal with results of simulations or computational experiments. Then, other formats or DB paradigms may be more suitable. Nevertheless, let us assume that relational databases are what we decide to use. At this stage, we may also decide which DBMS to use. This decision may be based on aspects such as costs, maintenance aspects, licenses, and available training. In our book, we chose PostgreSQL, simply because it is a free (open source) and very mature SQL DBMS.

In the next step, the conceptual schema is converted into a logical schema. This can mean to map the entity types from the ERDs in the conceptual models to tables and the relationships between them [378, 380]. During this step, we may perform adaptations to improve the practicality and efficiency. We may, for example, sometimes split one entity type from the ERDs into multiple tables, join multiple entity types into a single table, or use different primary keys (e.g., sequential integers instead of a multi-part key constructed of strings...). In other words, we map the conceptual model to the internal structures provided by the DBMS. At this stage, we basically have an unoptimized design of the DB.

The logical schema is then translated to a physical schema. This stage emphasizes the internal aspects of the DB, e.g., the creation of access paths, indexes, the creation of partitions, and the implementation of business rules. After this, we have a complete DB design, optimized for the planned operations.

Side note: The division in conceptual, logical, and physical design is not always defined like this. Some approaches only distinguish the conceptual and the physical model [177, 469]. In this case, the conceptual model is also called logical model and focuses on the specifications of the entity types and relations of the data. The physical model then comprises the two aspects that we called physical and logical design above, i.e., the definition of tables, relations, and access paths. From my perspective, which of the two approaches are chosen does not really matter. Whether we have three-step or a two-step design method – the important point is that we first model the data in an technological-independent and abstract way and later map this abstract model to a concrete design.

During the DB design phase, it is important to communicate and interact with the project stakeholders. At this stage, the requirements may change again, which must be properly documented in the specifications. While the DB is being designed, the user-facing applications are designed in parallel. Here, too, changes may be found necessary during discussions with the future users.

After the design phase, the DB is implemented based on the physical design documents developed earlier. The DB tables are created, populated with data, and constraints and queries are implemented.

Now that the DB exists, the application(s) can be implemented as well. They are then integrated with the DB. The system is tested.

Finally, the system is installed. The users can now evaluate it. The stakeholders can try the

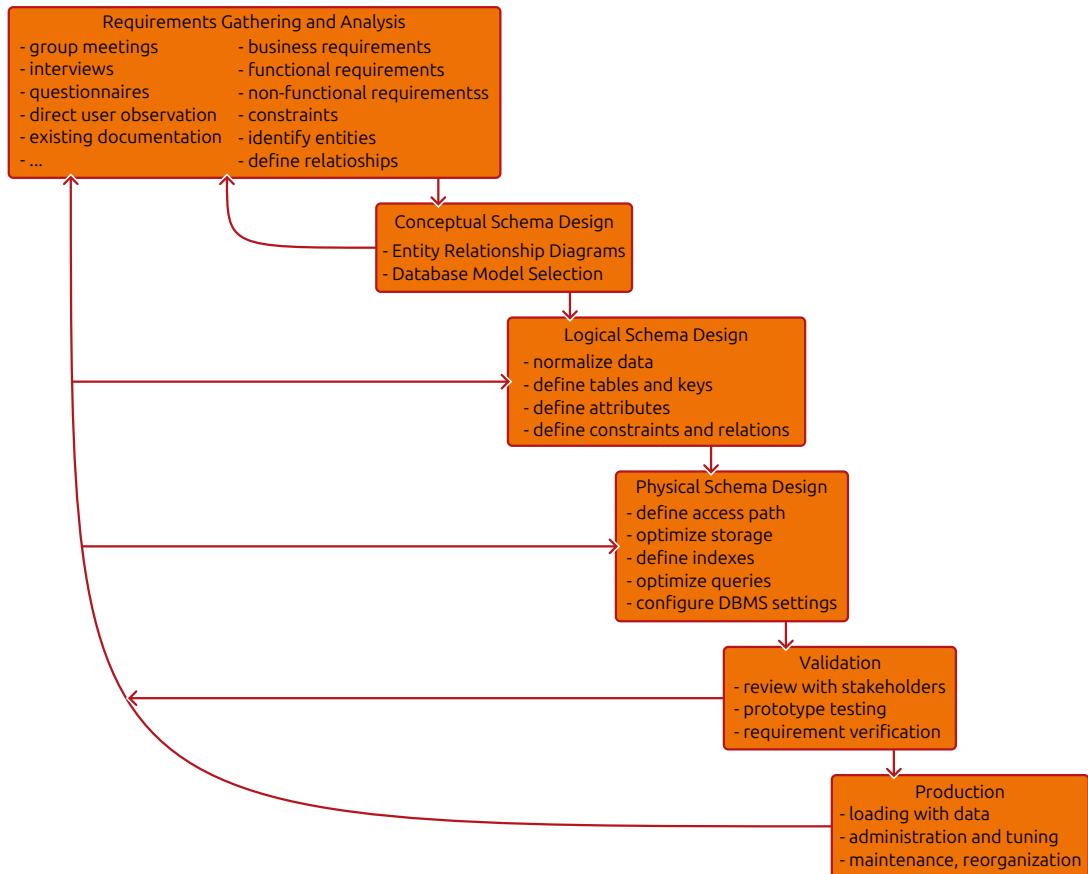


Figure 16.6: A simpler DB lifecycle model.

applications and judge its functionality and performance. After the system has been accepted and enters the productive stage, the maintenance phase is entered. The system is continuously upgraded, improved, and extended until it eventually reaches its end of life.

The method by Gupta, Mata-Toledo, and Monger [191] sketched in Figure 16.5 is a comprehensive approach for DB and software co-design in a large-scale project. The main goal is to make DB projects more predictable. Most discussions on DBs-design focus more on the core of the development process, labeled *Database Design* in Figure 16.5. They also often combine several of the framework activities such as installation and maintenance into one activity.

When we explore the design phases of DBs, we, too, will use a model with fewer steps. Figure 16.6 illustrates such a simplified lifecycle model that combines ideas from several sources [139, 378, 380]. Indeed, we will hinge the more comprehensive DB design example project discussed in this part of our book on this process.

This approach to the DB development life cycle begins with a phase of requirements gathering and analysis. Then, the three models (conceptual, logical, and physical) are designed one after the other [157]. We start with the conceptual model: The entities and their relationships are sketched with ERDs. It makes sense to discuss this model with the stakeholders and, if any inconsistencies are discovered, to adapt the requirements specification.

In this step, we also choose the proper data model.

The entities in the conceptual model are then translated to fit to the data model in the logical model design process. In case of a relational data model (as used here), this means to create tables, define keys, attributes, and relations, and to normalize the data. This model can be defined either in SQL or using any other suitable formal mechanism. It could be bound to a specific DBMS or to a group of similar DBMSes.

Finally, the logical model, i.e., the perspective of the users on the data, is mapped to a physical model. The physical model is the internal representation of the data as provided by the DBMS. The logical model defines how the data is accessed. The physical model defines how the access is made

efficient and can contain information about the physical layout of the data as well as the use of indexes and other performance tweaks.

Now a prototype can be developed and verified with the stakeholders. Finally, the DB enters the production stage. From here on, the focus is on maintenance, regular backups, fine-tuning, and the addition of features and adaptations to changed situations. The last two stages can create feedback to the earlier stages, i.e., lead to changes in the physical or logical models as well as the requirements.

16.3 Summary

Databases are long-lived and important assets of an organization. They are somewhere in the gray area between documents and software. They are the foundation for many of most important applications in an organization. Their design and performance therefore impact many of the most important processes of organizations. And they do so over a very long time.

If we design a DB, then we cannot just begin doing so aimlessly or without a clear plan. Instead, it makes sense to follow one of the well-known design processes. Which one is probably not very important. The important point is that all of them help us to bring structure into our work. They provide us guidance. They make projects plannable. And maybe most importantly, they reduce risks. There are plenty of risks, ranging from running out of budget or running out of time to not meeting the user requirements. To mitigate such risks, we follow established engineering approaches.

Most DB design processes boil down to iteratively working our way from a conceptual over a logical to a physical schema. They usually preface this process with requirements gathering and analysis. During the actual design, close contact with stakeholders is often emphasized. Here, prototypes come in handy and allow us to collect feedback and to discover misunderstandings. With this we can then update design documents and schemas if need be. As epilogue after the DB enters the productive stage, periodic activities like maintainance, update, and backup are usually prescribed.

In the following, we will work our way step-by-step through a DB development lifecycle. By doing so, we will not just explore the different steps in more detail, but also discuss several important topics, such as ERDs, and normalization, based on a more or less realistic scenario.

Chapter 17

Requirements Analysis

The requirements analysis is one of the most important steps of the DB development lifecycle. It is this point where we gain the understanding of the project. Requirements engineering mediates between the users and customers of a project on one side and the developers and suppliers on the other [426]. The results of the requirements analysis are **Software Requirements Specification (SRS)** documents that [426]

- enable an agreed understanding of all stakeholders (acquirers, users, customers, operators, developers, and suppliers),
- have been validated against the real-world needs,
- can be implemented, and
- provide a reference for verifying designs and solutions.

On one hand, during the requirements analysis, we build a clear understanding of the purpose, goals, and limits of the project. On the other hand, we also need to learn about the organizational structure and processes that should be embodied by our DB and the applications on top of it. Indeed, studies show that for many companies, more than half of the problems of systems [503] and costs of software development [62] are based on poor requirements definition. Inefficient requirements management is considered a top-cause for project failure [151]. Good requirements engineering can increase the developer productivity and lead to improved project planning [115]. Fixing errors in the requirement cost 10 to 200 times as much once the application is deployed compared to discovering them during the requirements analysis [46, 286, 367]. The collection and analysis of requirements therefore is very important.

17.1 Types of Requirements

Requirements can roughly be divided into business requirements, functional requirements, non-functional requirements, and constraints [225].

First, the business requirements are the high-level goals, objectives, and sought outcomes of the project. They define the motivation of the organization for why the system is being developed [426]. An organization usually has some overall initiative or plan to improve some of its metrics. The project is initiated to support this initiative. These requirements therefore defined such that the project will contribute to improving the organization. They are more general and abstract.

Second, the functional requirements are more concrete and define what the system should do. They define the functions, features, and the behavior that the system must offer to fulfill the business requirements. They may be defined as

- the input given to the system,
- the expected operation to be performed by the system on that input, and
- the expected output to be produced.

Third, the non-functional requirements define how the system should perform. They define the service quality that the system should offer, as well as the required performance, usability, scalability, reliability, etc. This includes the computational environment in which it must be possible to execute the system.

Fourth and finally, the constraints are the factors that limit which solutions are viable. They define the boundaries in which we operate. While requirements define properties that our system should have, the constraints rule out methods to get there. They can appear in form of budget or time limits. They may also appear in the form of demands like ‘*Your system must work with version XYZ of software ABC.*’

17.2 Requirements Gathering

Several different methods exist that can be used to collect requirements [503].

The first method, interviewing stakeholders to gather information about the system that we will design, is considered one of the most efficient requirements gathering techniques [128, 503]. For each interview, careful preparation is needed, which includes making an appropriate appointment. While stakeholders may spontaneously share information on some topics, they may not discuss others unless explicitly prompted with questions [60]. Each interview meeting should therefore have a proper agenda, pre-prepared questions, and follow a checklist [479]. If possible and if the interview partner(s) consent, then the meeting should be recorded. The recordings and notes should be evaluated within two days after the interview. The interviews should neutral, not push the interviewee into any direction. The goal is to collect diverse views on the project.

A second, more standardized method to collect information are questionnaires [430, 503]. This way, we can collect much data from many stakeholders within a brief time. Questionnaires are easy to evaluate and process, but designing them properly is important. Different types of stakeholders will use different parts of our system. Therefore, it may be necessary to design several different questionnaires, one for each group of future users. The questions should be clear and unambiguous. For some aspects, multiple-choice questions or range-based ratings are good, while others may require open-ended questions where the users can fill in their opinions.

The third method is to directly observe users doing their work [390, 503]. Stakeholders may not always be able to accurately describe their function and how they fulfill it. Observing them performing their day-to-day processes can thus provide helpful additional information. It is also possible to record such real-world examples instead of personally observing them [197]. Of course, under observation, people may behave somewhat differently from normal, so such information is to be taken with a grain of salt.

Fourth, we can also analyze both organization-internal and external regulations, as well as published procedures, processes, and other documents [367, 503]. Usually, an organization will have their own written regulations, announcements, and otherwise standardized procedures. Additionally, there are regulations and laws imposed upon an organization and the processes within. Gathering such information can be crucial and complements our understanding of what the system is supposed to do, and why current systems work the way they do. Sometimes, the official documentation of organization-internal methods and the practical realization of processes may differ, though.

A fifth method are group meetings and workshops [503]. Here, under the guidance of a session leader, stakeholders at all levels meet, from management to end user, from system analyst to data entry personnel. The group then jointly discusses the current situation and the planned system, which can be a highly efficient way to gather requirements. Different variants of this method exist since the 1970s, under names such as **Joint Application Development (JAD)** [67, 287] and **Participatory Design (PD)** [67, 165].

Several of these techniques can be combined, often with other approaches such as brainstorming sessions, surveys, reverse engineering of existing systems, or prototyping [225, 503]. In the context of DBs, it is particularly important to properly define the data structures and entities when analysing the requirements and, later, when developing the conceptual model [280].

17.3 Requirements Specification Document

After the requirements have been collected, they are stored in a formal specification document, the so-called **Software Requirements Specification (SRS)** document [400, 494]. The SRS is the most important document in the software development process. The document structure should follow the IEEE 830-1998 [219] standard or the newer ISO/IEC/IEEE 29148-2018 [426] standard. While the lifecycle of software or systems in general can be managed by ISO/IEC/IEEE 15288 [427] and ISO/IEC/IEEE 12207 [425], respectively, ISO/IEC/IEEE 29148 [426] provides the guidelines for their

requirements-related processes. Generally, it would be a good idea to simply follow these standards when gathering and analyzing requirements for software or DB projects.

17.4 Example: Teaching Management Platform

We will now explore the gathering and analyzing of requirements by using an example. Our example project is the development of a teaching management platform for a university. Sadly, doing this at a realistic level would go far beyond what we can do as a reasonable example in the context of a course book. We cannot really implement complete outline of ISO/IEC/IEEE 29148-2018 [426]. We cannot even specify the complete and exact requirements of any realistic system without exceeding reasonable time and length limits. Therefore, we will try to discuss the requirements partially. We will pick some more or less interesting issues while leaving others to the imagination of the reader. Also, we write down everything in a very informal way. In actual documents, a much more formal language would be used.

17.4.1 Business Requirements

Our imaginary university has several goals that it wants to achieve by introducing a new teaching management platform. At the start of our project, we had several meetings with the university leadership. During these meetings we gathered and understood the business purpose of the project. We also get a good idea about the roles of the stakeholders involved.

17.4.1.1 Business Purpose

First, by migrating processes into a DB and designing applications to access them via the web, all stakeholders will benefit: The students can more easily access their courses, obtain transcripts, register for classes, etc. The professors can manage their classes more efficiently. The workload of the management of the university can significantly be reduced and simplified.

Second, the administrative personnel so far does all the management of its students and courses using Microsoft Excel sheets and pen and paper. This has several drawbacks, such as the lack of centrally controlled backups, the possibility of errors, the lack of traceability of processes, problems when this duty is eventually handed over from one teacher to another, and so on. By developing a centralized system, the imaginary university wants increase the control over as well the accountability, traceability, and documentation of its processes. This could be seen as a tool for supporting quality management, maybe along the lines of ISO 9001 [346, 347].

Third, the long term goal is the digital transformation of our whole imaginary university. All processes inside the imaginary university would be managed by online platforms. This would significantly reduce administrative efforts and costs. All processes would automatically be documented and backed up. The quality of the services rendered to students would improve. Auditing becomes easier. The teaching management platform is the first building block of this digital transformation. It will allow the university to gather experience with systems that are not handled just by a few administrative personnel (such as the HR or financial accounting system), but accessed by thousands of users in different roles.

17.4.1.2 Major Stakeholders

There are five main groups of stakeholders on the university side of our imaginary university management system.

First, there are the students. The students need to register online for modules (courses and exercises). They want to print their schedule, which includes which courses they will attend at which days and times and in which rooms. They want to view their scores and progress.

Second, there are the faculty members, i.e., the professors and teachers. A professor can chair a module, meaning that they will teach a certain class. Other faculty members, say lecturers or assistant professors, may teach lab classes or practice classes. They have access to the list of students in their respective modules. Like students, teachers can see their schedule.

Third, the university administration can create and manage schools in the system. The university administration also manages the professors, teachers, and students. They are the only ones who can

enter new persons into the system. They are the only ones who can create new curricula for the different schools in the system. They assign buildings and rooms to schools.

Fourth, there is the administration of the different schools of the university. The school administration assigns professors to modules, lecturers to practice classes. They are allowed to create and schedule exams and major deliverables. They manage their rooms. They manage the students belonging to their school.

Fifth and finally, there are system administrators, including the **DBAs**. Their most important task is to keep the system running. This means that they will run regular backups. They also need to practice regularly how to restore the system from the backups. They also need to update all the involved components, such as the **OS**, the **DBMS**, the web **servers**, and so on.

17.4.2 Functional Requirements

After the stakeholders and motivation of the project have been reasonably clarified, we begin by interviewing several involved personnel. We visit the education department of the university. We interview the deans, vice deans for teaching, and secretaries of three different schools. We discuss with five professors. We also meet with the students union and several students at different academic performance levels. Our goal is to understand the academic processes from the points of view of different sides.

We want to clarify several questions. What is the basic functionality that we need to provide? What features could we offer that, currently, are unavailable? Based on our findings, we create different questionnaires and distribute them more widely among the above peer groups. Finally, we collect all the information and present them in workshops where, again, members of all the above groups take part. It is our goal to build a view on the requirements that can be agreed to by all stakeholders. We then continue writing our **SRS** document as follows.

The system has to be available through one or multiple websites. Several processes must be supported, for example:

17.4.2.1 Person Management

Only the university administration can create student or faculty member records in the system. Such a record must store information such as name, ID, mobile phone number, gender, highest academic degree, academic rank, role (student, staff, ...) etc. The university administration must be able to change these information. They can assign the people to schools. The administration of a school can access only the data of the people assigned to them.

17.4.2.2 Date Management

The university administration sets dates such as semester begin and end, begin and end of the exam period, holidays, etc. The university administration provides ranges for special dates such as the start of a graduation projects (开题), the graduation project midterm evaluations (中期), or the graduation project defenses (毕业). The schools then can adapt these date ranges to their situation. Professors who chair modules can then, for example, schedule an exam in the time range defined by their school, which, in turn, is a sub-range of the exam schedule provided by the university.

17.4.2.3 Curriculum Management

The university administration can create and change curricula. They do this of course in cooperation with the schools, but only they have the right to make decisions. Each curriculum contains different modules at different semesters. Modules can be compulsory or optional and belong to certain types, such as Subject Basic Courses (学科基础课), General Basic Courses (公共基础课), Professional Basic Courses (专业基础课), Professional Elective Courses (专业选修课), or General Elective Courses (公共选修课) and such and such. Modules can consist of only lectures, of lectures and practical training, only practical training, or deliverables (such as BSc and MSc theses). The university administration then assigns curricula to schools. The school administration can enroll their students to curricula.

17.4.2.4 Module Management

The school administration can, in each semester, create implementations of the modules. An implementation assigns a teacher to course and has an upper limit for student enrollment. Teachers are

notified about their assigned courses. Teachers can print their teaching schedule.

17.4.2.5 Room Management

The university administration manages, creates, changes, or deletes records for lecture rooms. Rooms have locations in buildings, a capacity for students, and features such as equipment (overhead projectors, blackboards, lab equipment for computer science, chemistry, biology, . . .). The university administration can assign rooms to schools. The school administration can assign courses to rooms and timeslots.

17.4.2.6 Module Enrollment

Students can enroll to the modules in their respective curricula. For compulsory modules that are offered only by a single professor, they are automatically assigned. Otherwise, the school administration can manually assign them. For any module in their curriculum for which they are not assigned by the school, they can choose by themselves. Students are notified automatically about enrollment options. Students can print their schedule.

17.4.2.7 Exams and Deliverables

The school administration can create exams for each module. An exam has an assigned room and time in the exam period. Professors can also request the school to schedule midterm exams, if they want. A midterm exam has an assigned room and time outside the exam period. For each module they teach, professors can also create deliverable records, for example for homework. They can then assign scores for the students for the exams and deliverables. Students are automatically notified about exam dates and locations for their modules. Students can print their current transcripts as well as the scores of all deliverables for each module they take at any time.

17.4.2.8 Communication

The system offers a facility for communication between students, teachers, and their school. The communication records are immutable and will be preserved. While this channel is likely not used often, it may be useful for things such as reminders, notifications, but also warnings or objections.

17.4.2.9 Administration and Backup

The DBAs of the university can create backups of the platform. They can update the platform. There always is a second instance of the platform to test updates and to verify that the backup procedure is working. The DBAs can also install the platform on a new computer and load the backups.

17.4.3 Non-Functional Requirements

The websites must render correctly and be usable both on desktop computers as well as mobile phones. They must work under the out-of-the-box default web browsers provided by Microsoft Windows, Linux, macOS, iOS, iPadOS, and Android.

The system must handle 50 000 new student records, 2000 new staff records, 200 curricula, 8000 modules, and 1 000 000 new exam/deliverable results per year. It must be able to handle this load over 20 years, i.e., twenty times the above. After twenty years, we assume that either the hardware is upgraded or that old records are removed from the system and backed up elsewhere. The response time for any query of any application must never exceed 2 seconds.

Communication should be secured over Hypertext Transfer Protocol Secure (HTTPS). Proper data protection must be offered, i.e., people can only access data relevant to them and this access is secured.

17.4.4 Constraints

The system must be set up as a set of Docker containers in the computational center of the university. The system should be composed entirely of open source software, in order to increase reliability, availability, and to reduce costs. The computational center of the university will provide three computational nodes, each with a 32 core processor with 4 GHz clock speed and 64 GiB of RAM. The computational center of the university will provide 100 TiB of centralized storage space. A first prototype must be

developed within six months. The system must enter thorough test and validation within one year. The project budget is limited to 5 000 000 RMB.

17.4.5 Summary

From this requirements analysis, we learned a lot. We understand that this is actually quite a complicated system. We also see that it does involve a lot of different aspects. Many DB projects do.

Like in many DB projects, there is not just the DB. There also are web-based graphical user interfaces (GUIs). Our course and book, however, are not about user interface design. We will just assume that we are part of a team here, and somebody else will take care of the GUI. We also do not really care about the Docker software environment structure either. We also just assume that the budget, the other constraints, and the non-functional requirements are OK. In the following, we will focus entirely on the DB part of this project.

This project would be immensely more complicated than our simple example from back in Part II (A Simple Example: The Factory Database). However, if you think back on what we have already touched, SQL, forms, and reports ... then you may have some rough idea on how this project here could be tackled. OK, the tables that we are going to need will be more complicated. But they will still be tables. The JOIN and REFERENCES keywords will still work and be our basic tools to connect and merge data.

Of course, the system should have a web-based GUI facing the students – but, as said, we will ignore this and assume that this can somehow be done via Python. At least we can imagine to use LibreOffice Base to construct at least some raw prototype for the administrative staff, where data could be entered and where schedules could be printed as reports.

Clearly, this undertaking will require much more brain power from our side. But if we tackle this in a well-structured way step-by-step ... then maybe we do have a decent chance. Actually, doing things in an well-organized ... that's all what we have been talking about recently. First we begin with a conceptual model, then we derive a logical model ... that's exactly this kind of stuff. Let's do it like this!

Chapter 18

Conceptual Model Design

Let us now begin with the conceptual modelling of applications. The requirements analysis has provided us with an understanding of the entities in our scenario and the relationships between them. In the case of our example, we know the entities of the teaching management platform and how they interact. However, so far, we discussed them only very informally. It is now time to put the information about the entities together into consistent models. At the conceptual design step, these models will be independent from any concrete technology. This step is called *entity relationship modeling* [23, 184, 377, 378, 468].

It should be noted that creating such models makes a lot of sense for larger DB applications like our example here. However, there are also many possible smaller situations where we may want to use a DB, say, to manage our literature reference, to manage a collection of books or musical records. For such smaller projects, one may directly skip this step and move on to the logical model [393]. Either way, we are now working on a beautiful and big project, so we definitely want to take this step.

18.1 Entities and Attributes

The first major component to model are the datastructures to be stored inside the DB. For this purpose, the following modeling primitives have emerged. The most basic elements for conceptual modeling are entities, attributes, and entity types.

Definition 18.1: Entity

An *entity* is an object or thing with an independent existence in the world. It can be distinguished from all other objects.

Examples of entities are maybe the student Mr. Bibbo, the module *Programming with Python* [482], the professor Mrs. Bebba 教授, room #36 202, or the curriculum *Computer Science and Technology*. Entities can be spotted easily in the requirement specification or when viewing the meeting or interview notes: They correspond to *proper nouns* [78], i.e., nouns that actually name one specific thing and that are usually capitalized [154].

Definition 18.2: Attribute

An *attribute* a is characterized by a name and a domain $\text{dom}(a)$ of values that it can take on.

Attributes are features or characteristics of an entity. Entities in our model are represented by the values of their attributes. A student could be defined by their name, ID, student ID, mobile phone number, home address, DOB, etc. A module could be described by its title, syllabus, and abstract. Additionally to such features, *adjectives* in the requirements text often can be interpreted as attribute values [78], e.g., red, young, successful, heavy, fast.

Definition 18.3: Domain

The *domain* $\text{dom}(a)$ of an attribute a is the set of possible values that it can take on.

We can distinguish the concepts of domain and datatype in the context of conceptual models [395]. A datatype is a mathematical concept whereas a domain is a logical concept. For example, `VARCHAR(100)`, `SMALLINT`, and `REAL` are datatypes. The name of a student, while being represented as a text string, is a logical concept. Mobile phone numbers and IDs are maybe also represented as text strings, but probably are of a fixed length and limited to certain character ranges at certain positions. A DOB, on the other hand, is a special date whose year is from a certain range of reasonable years. A score in an exam may be an integer number between 0 and 100. The attribute domain therefore can be considered as the combination of a datatype with semantics limiting its valid range.

Definition 18.4: Entity Type

The set of all entities that have the same attributes is called an *entity type*.

So while the entity Mr. Bebbo is a single student, the set of all possible students would form an entity type. While the module *Programming with Python* [482] is a single entity, the set of all possible modules would form an entity type. The professor Mrs. Bebba 教授 is a single entity, but the set of all possible professors represents an entity type. Entity types are *common nouns* that stand for groups or types of things [78] and that are usually written with lowercase letters [154].

Also notice the plural *attributes* in Definition 18.4:

Best Practice 13

Only things with multiple attributes should become entity types.

Indeed, it makes little sense, for example, to consider *year* as an entity type, because it does not have multiple attributes.

Best Practice 14

Each entity (type) should model exactly one (type of) object from reality (and not more than one) [381].

If we model the setup of, let's say, a car race, then drivers and cars should be separate entity types, although each driver is assigned to one car. If instead we would include the name of the driver and the type and features of a car into a single entity type, then we would likely encounter redundancy: If multiple cars of the same type participate in the race, we would store information about them several times (each time together with other driver data).

Definition 18.5: Entity Set

An *entity set* is a subset of an entity type. It is a set of some entities of a type that exist at one point in time.

For example, Mr. Bebbo is a single student entity, the set of all possible student entities forms an entity type, but the students Mr. Bebbo, Mr. Bibbo, and Mr. Bobbo together form an entity set. The modules *Programming with Python* [482] and *Databases* [481] form an entity set. This entity set is a subset of an entity type for modules. Notice that the mathematical notion of *set* is indeed correct here: All entities have a unique identity and, hence, can be differentiated from all possible other objects. There are no two identical Mr. Bibbos. Therefore, students can be grouped in a set and entity types are sets, too.

Viewing the conceptual design of DBs through this lense, we notice a few things. First, we always model only a tiny window to the real world. When we talk about students, modules, and curricula as entity types, this only concerns our particular application. Of course, in our real big wide world, students exist in other universities and in other countries. These students may have different attributes from ours. They probably do not have an Chinese ID (中华人民共和国居民身份证), for example. They do not matter in the model of our small part of the world. Also, we only model that aspects of

the students that are relevant to our application. We do not model, for example, their hobbies, favorite songs, shoe size, favorite food, hair color, etc. These do not matter in our window to the world either. Therefore, the entity type *student* does not reflect the full real-world concept of a student. It is a simplified projection of this concept into our application.

Second, if you attended or do attend a course on programming (such as [482]), then you will feel that this way of modeling things is a bit related to **Object-Oriented Programming (OOP)**. Entity types could be thought of as **classes** and entities could be their instances. Attributes could be their, well, attributes. While **DB** theorists may dislike this way of thinking, I believe that it is not wrong. It is a viable analogy. However, later, when we model the relationships between entities, it may no longer be helpful.

We now know that entity types with their attributes basically correspond to datastructures in programming. They form one of the important components of the conceptual model. But how do we actually write them down? How do we specify them?

For this, a graphical method has been introduced: **Entity relationship diagrams (ERDs)** are the most commonly used tool to model the entity types and their relationships in a DB [23, 79–81, 240, 484]. There exists a wide variety of graphical notations that can be used for ERDs. The original notations by Bachman [13] and Chen [80, 81] are still in use, the Crow's Foot notation [66, 159] is very common, the more comprehensive and standardized **Integration Definition for Information Modeling (IDEF1X)** syntax [54, 223], and the **Unified Modeling Language (UML)** [48, 309, 459]. Indeed, there are many different flavors of diagrams that can serve as ERDs. Shamshin [393], for example, presents nine slightly different variations. Scheweppe and Scholz [378] has two baseline variants and includes several slight variations of a UML-based approach (which is not listed in [393]). The notation used by Vandenberg [468] is yet again slightly different. Therefore, ERDs may look slightly different depending on who drew them and which tools they used. However, understanding them is not really hard, so these differences are not that important.

In the following sections, we will look at several different ERDs. The goal of this course is to teach you *actionable* knowledge. So, we will not just look at ERDs, we will also *draw* them.

That is fairly easy: Once you understand the basic syntax, you can draw them with almost arbitrary drawing software. Then again, this can also become tedious. You could, for example, use the **Draw** program which is part of **LibreOffice**, or a vector graphics program like **Inkscape**. This would mean that you have to draw all the shapes of the diagrams independently, which would yield inconsistent designs and be generally tedious.

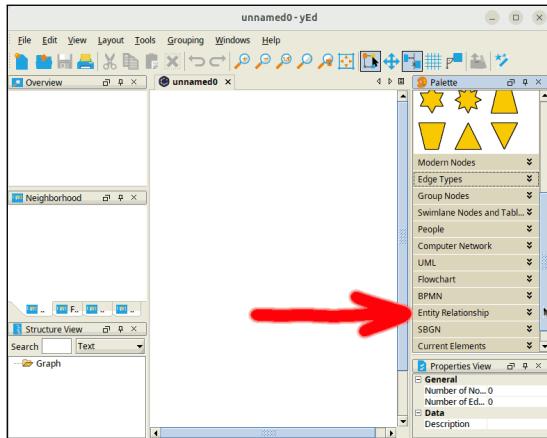
Then there are programs like **PgModeler** or **MySQL Workbench** which offer much better capabilities to draw ERDs. These tools, however, are bound to certain technologies, such as **PostgreSQL** and **MySQL**, respectively. They would be useful for the development of logical models, but it feels awkward to apply them at a conceptual level, which should be technology agnostic. There exists many possible tools that we could use. Brumm lists over 70 in [56].

After searching for a while, I have settled for **yEd** [384, 497]. **yEd** is a free graph editor that offers a convenient ability to draw and layout ERDs while being entirely independent from any data model. In [Chapter 6](#), we provide instructions on how to obtain and install this program. We will use it for all of the conceptual-level ERDs in the rest of the book. As an example on how to use **yEd**, we will give some instructions on how to draw the most simplest ERD with only a single entity type inside in [Figure 18.1](#). This program is rather easy to use, so after that example, we will assume that you can figure out how to draw more advanced ERDs on your own... At least we do not just paint some ERDs and leave you entirely to your own devices when the time comes where you should draw some as well...

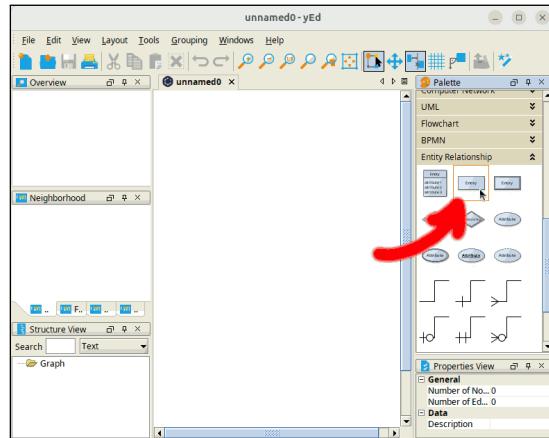
Before we really get into this, just a quick note: In the context of ERDs, an *entity type* are sometimes called a *entity*. In other words, the meaning of the term *entity* is shifted. But let this not bother us too much.

The very first thing that we want to model is the entity (type) *Student*. From our requirements analysis, we know that students have names, they have an ID (中国公民身份号码[507], issued by the government), they have a student ID (issued by our university), they have a mobile phone number, they have an address, and they have a **DOB**. So let us model this.

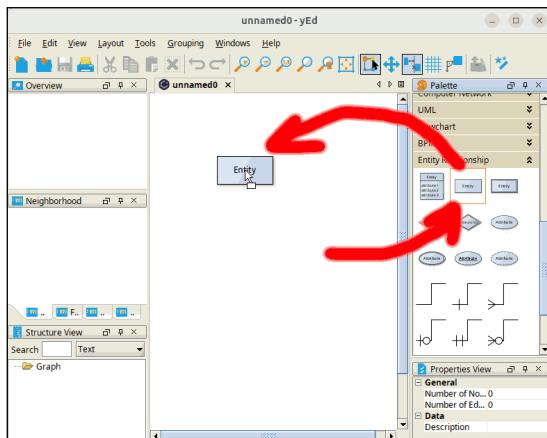
After installing **yEd** as discussed in [Chapter 6](#), we open it. We scroll down the **Palette** pane on the right-hand side until we find the **Entity Relationship** tab. We click on the tab and it opens in [Figure 18.1.1](#), offering us all the symbols and connectors commonly used in ERDs. Entity (types) in ERDs are represented by rectangles. We can now click on the **Entity** symbol and drag it into the empty



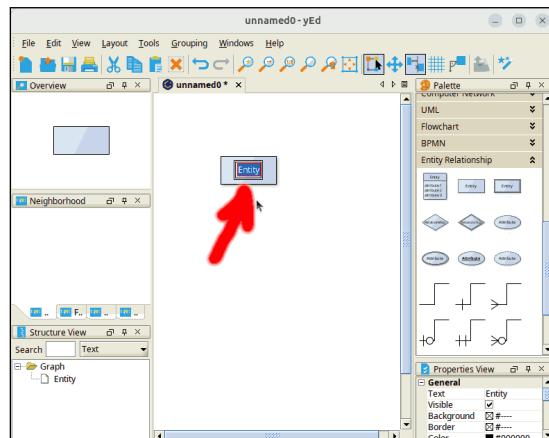
(18.1.1) We open the yEd editor and click on the Entity Relationship tab in the Palette view on the right-hand side.



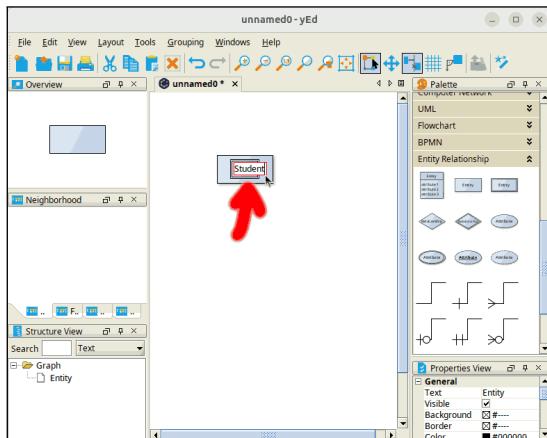
(18.1.2) We can now click on the Entity symbol and drag it into the empty document.



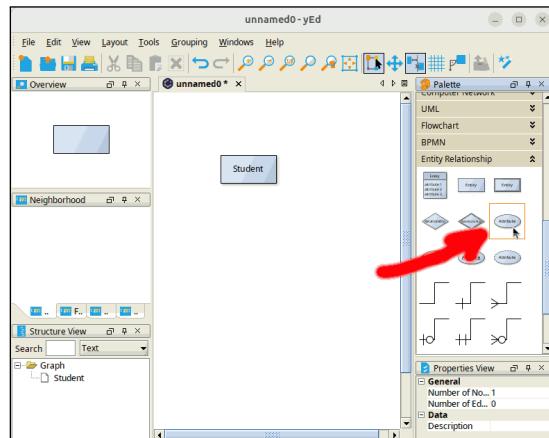
(18.1.3) We dragged the entity symbol into the diagram document.



(18.1.4) We double-click into the new entity symbol in the document in order to edit its name.

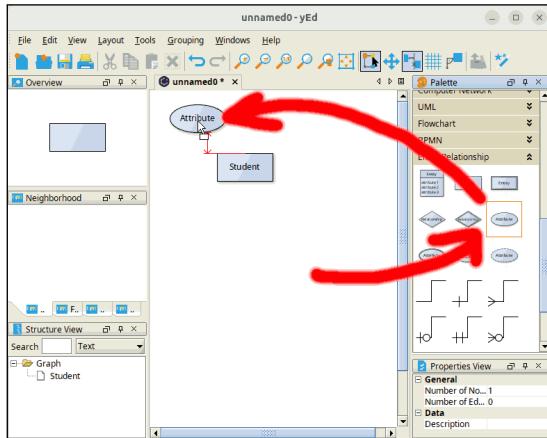


(18.1.5) We change the entity type name to "Student" and press **Enter**.

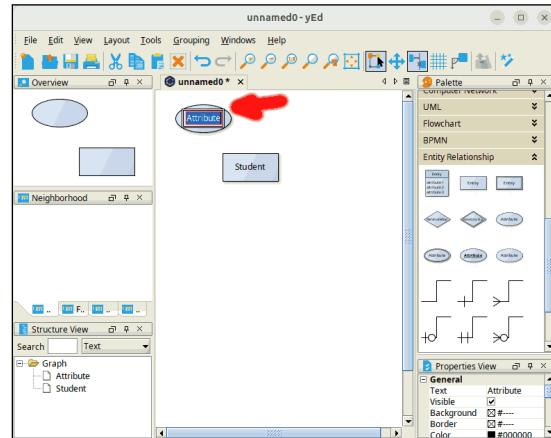


(18.1.6) The name has changed. We now click on the Attribute symbol in the palette.

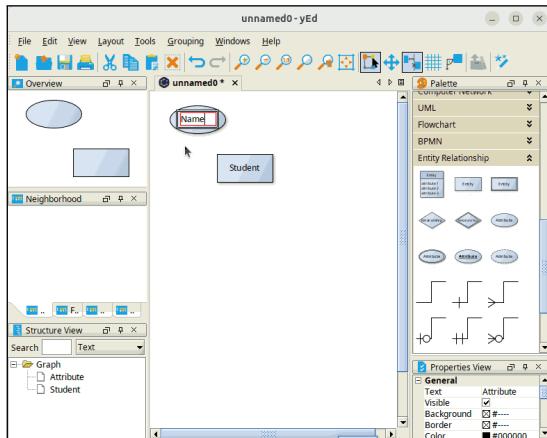
Figure 18.1: Drawing an ERD for the *Student* entity type using yEd.



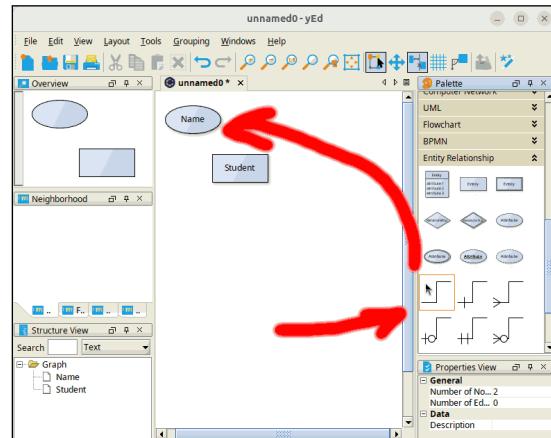
(18.1.7) We drag the attribute symbol into our document.



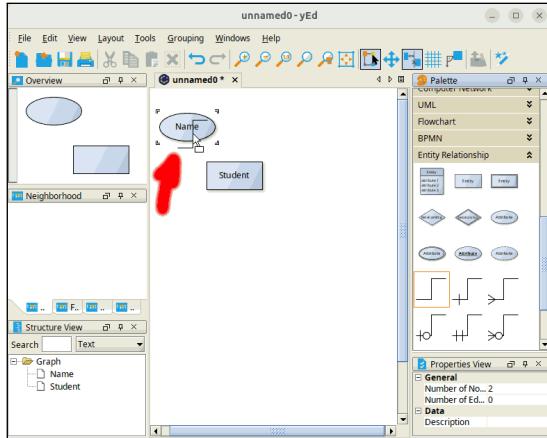
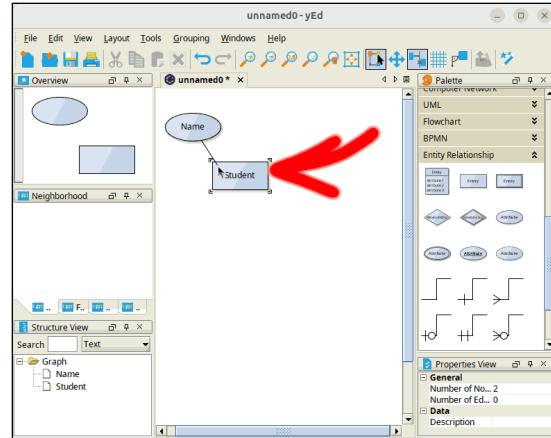
(18.1.8) We double-click into it to change its name.

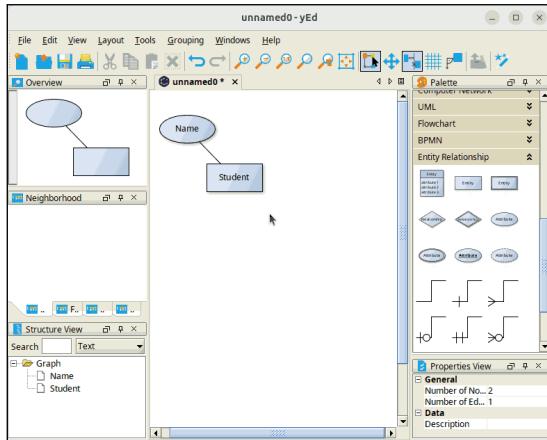


(18.1.9) We want its name to be, well, "Name."

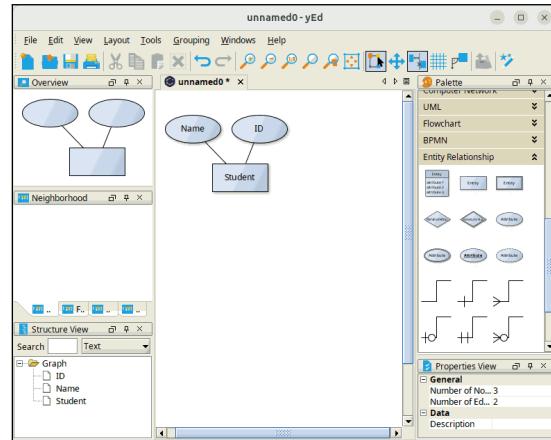


(18.1.10) We changed the attribute name. Now we click on the connection symbol in the palette and drag it right onto the attribute.

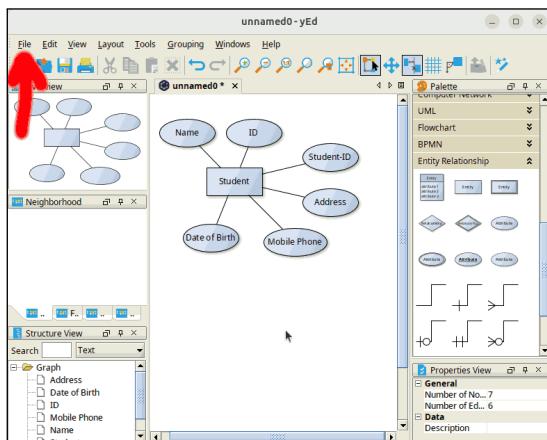
(18.1.11) We drop the connection symbol onto the *Name* attribute.(18.1.12) We then click into the entity type to connect the attribute *Name* to the *Student* entity type.Figure 18.1: Drawing an ERD for the *Student* entity type using yEd (Continued).



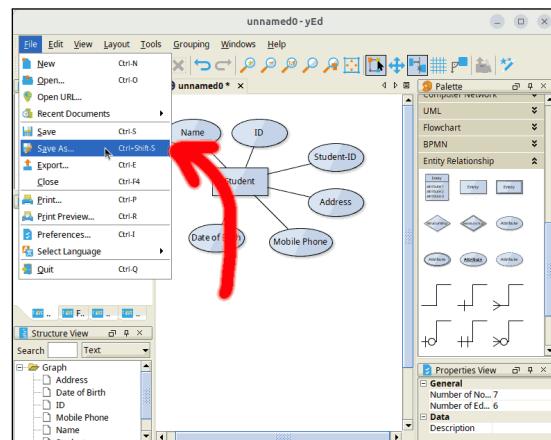
(18.1.13) The *Name* attribute is now connected to the *Student* entity type.



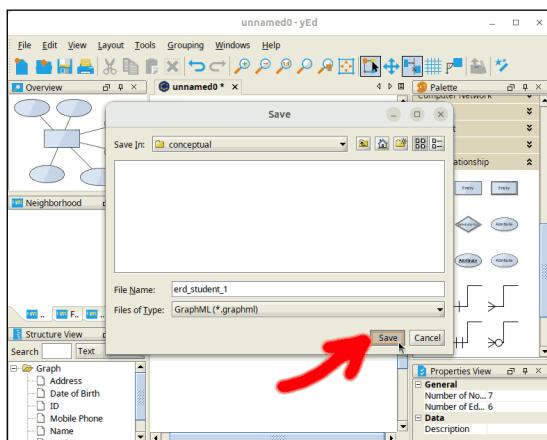
(18.1.14) We add an attribute *ID* in exactly the same way.



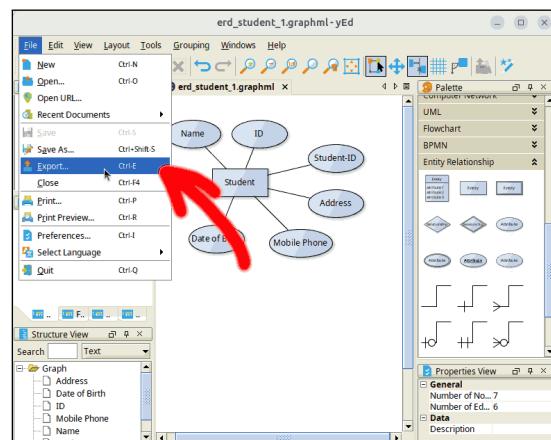
(18.1.15) We add several more attributes. Next, we click on the **File** menu.



(18.1.16) It is now time to save our document. We click on **Save As**.

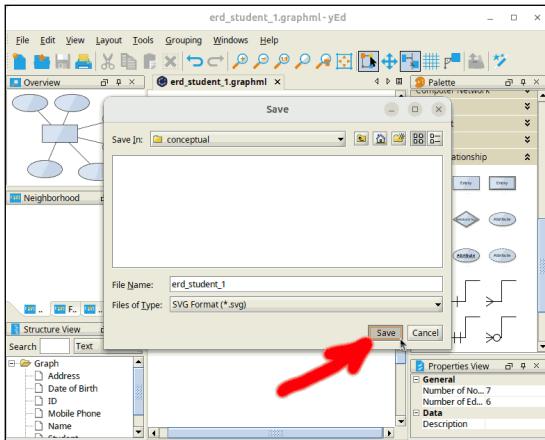


(18.1.17) We save the diagram under the name *erd_student_1* in the *graphml* format by entering this name and clicking **Save**.

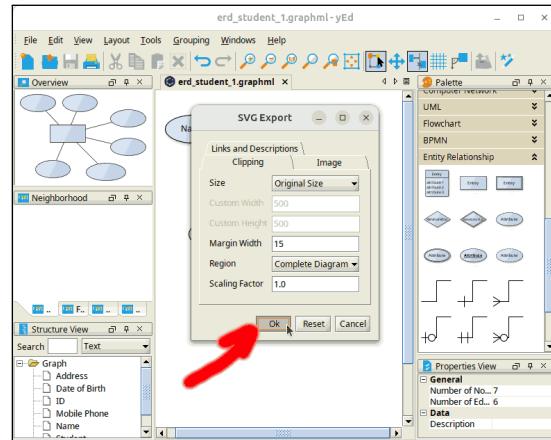


(18.1.18) We can also export the diagram in a format that we can use in other documents. For this, we click on **Export** in the **File** menu.

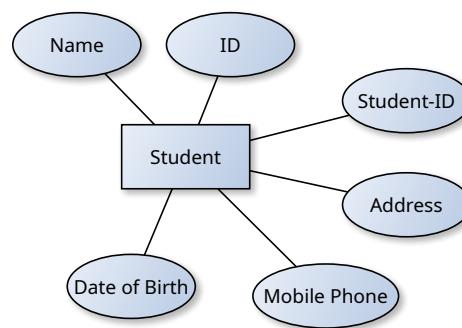
Figure 18.1: Drawing an ERD for the *Student* entity type using yEd (Continued).



(18.1.19) We choose to export in the Scalable Vector Graphics (SVG) format and click **Save**.



(18.1.20) We leave all settings as-is and click **OK**.



(18.1.21) This produces this beautiful ERD.

Figure 18.1: Drawing an ERD for the *Student* entity type using yEd (Continued).

document in Figure 18.1.2. We dragged the entity symbol into the diagram document in Figure 18.1.3.

Inside the rectangle, the name of the entity (type) is written. Let's change it to "Student". We therefore double-click into the new entity symbol in order to edit its name in Figure 18.1.4. The text inside is marked. We change the entity type name to "Student" and press **Enter** in Figure 18.1.5. The name has changed.

Let us now add the attributes of the *Student* entity type step by step. Attributes are represented as oval bubbles in ERDs that are connected to their owning entities by straight lines. We now click on the **Attribute** symbol in the element palette in Figure 18.1.6. We drag the attribute symbol into our document in Figure 18.1.7.

Of course, in Figure 18.1.8, we want to change its name as well. So we double-click into it to change its name. We want its name to be, well, "Name," i.e., we want to create an attribute that represents the name of the student in Figure 18.1.9. We successfully changed the attribute name in Figure 18.1.10. The attribute, however, is still floating in the diagram all by itself. Attributes belong to entities, so we need to attach it to the entity type *Student*.

We thus now we click on the connection symbol in the palette and drag it right onto the attribute. We drop the connection symbol onto the *Name* attribute in Figure 18.1.11. Our mouse cursor now marks the end of a connecting line and we can drag the connection to whatever object we want to. We click into the entity rectangle to connect the attribute *Name* to the *Student* entity type in Figure 18.1.12. The *Name* attribute is now connected to the *Student* entity type in Figure 18.1.13.

We add an attribute *ID* in exactly the same way in Figure 18.1.14. We then go on and add several more attributes, *Student-ID*, *Address*, *Mobile Phone*, *DOB*, in Figure 18.1.15. Notice that these are names that contain dashes and spaces, i.e., things that we would normally avoid when working with SQL. But we are not working with SQL. We are making a conceptual model. We want this model to be easily readable and beautiful. And it has to be. Because we want to discuss it with our stakeholders.

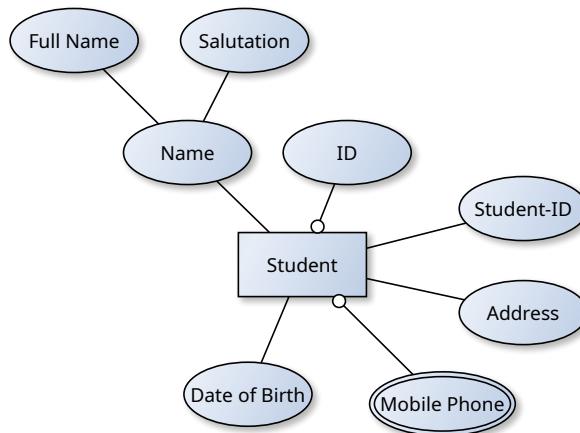


Figure 18.2: A new version of the *Student* ERD from Figure 18.1.21, this time with *Name* as composite attribute and *Mobile Phone* as multivalued attribute.

So we do not need to heed to restrictions of a particular technology at this point in time. Instead, we focus on readability.

Having finished drawing our very first ERD, it is time to save it. Next, we click on the `File` menu. We click on `Save As` in Figure 18.1.16. We save the diagram under the name `erd_student_1` in the `graphml` format by entering this name and clicking `Save` in Figure 18.1.17.

We can open this file later in order to keep editing it. However, we often want to also place the diagram into some document. For this purpose, it makes sense to convert it to an image. For this purpose, we click on `Export` in the `File` menu in Figure 18.1.18. We choose to export in the **Scalable Vector Graphics (SVG)** format and click `Save` in Figure 18.1.19. In the dialog that pops up, we leave all settings as-is and click `OK` in Figure 18.1.20.

This produces this beautiful ERD graphic shown in Figure 18.1.21. Notice that we can also open the SVG graphic in other programs, such as **Inkscape**, to further edit and beautify it. With this, we have completed our very first ERD. (Although we did not yet really touch the *Relationship* part symbolized by the *R* in ERD.)

Useful Tool 5

yEd [384, 497] is a free graph editor that can be used to draw ERDs. It is useful for the conceptual modeling stage in DB design as discussed in Chapter 18. Installation instructions are provided in Chapter 6 and a small hands-on tutorial is given in Section 18.1.

Anyway, let us continue our modeling adventure. We take our ERD from Figure 18.1.21 to our partners in the university, say, an administrator of one of the schools. We want to discuss whether this model for students is appropriate. During our discussion with the administrators, a few issues come up.

First, *Names* are more complicated than we thought. Students have a given name, a family name, and a salutation. For example, the name of “Mr. Bebbo” would actually be “Mr. Fred Bebbo”, where “Bebbo” is the family name, “Fred” is the given name. To accommodate this, *Name* attribute should not be a single attribute. We could turn it into a *composite attribute*, consisting of given name, family name, and a salutation. The system then could combine these components appropriately when addressing the student or when issuing documents.

Definition 18.6: Composite Attribute

A *composite attribute* is an attribute which consists of several parts with their own names and domains.

Definition 18.7: Simple Attribute

A *simple attribute* is an attribute that only has a single name and domain. Simple attributes do not have any components.

When thinking about this, we encounter a problem: In Western culture, the full name is usually composed by first writing the given name and then the family name. This is why the full name of Mr. Bebbo is Fred Bebbo. In Eastern cultures, it is often the other way around: The name of the famous Chinese mathematician 刘徽 is transcribed as LIU Hui in the Latin alphabet [303, 422, 500]. LIU (刘) here is the family name and it comes first. Hui (徽) is the given name and it comes second. Depending on the cultural background, the system would need to decide how to compose the name correctly. This sounds like an awful problem that would probably become the source of many interesting errors or, at least, complaints.

We decide to not compose the name of given name and family name. Instead, we provide two different components as part of the *Name* attribute: The *Full Name* will be the complete and official name, as written on the ID card. This would be something like 刘徽 for a Chinese person and Fred Bebbo for the Westerner Mr. Bebbo. As second component, we store the complete *Salutation*. For 刘徽, this could be 刘教授 and for Fred Bebbo, it could be Mr. Bebbo. Maybe we could later permit the students to change their salutation in the system to whatever they feel comfortable with. The full name, however, would be entered by an administrator exactly as it is spelled on the student's ID document. If our system would ever send an automated message to Mr. Bebbo, it will use the salutation. If we print the name on a certificate, we will use the full name. A composite attribute in the ERD is simply drawn with the components as attributes connected to the attribute.

Definition 18.8: Multi-Valued Attribute

An entity may have multiple values for a *multivalued attribute*.

Definition 18.9: Single-Valued Attribute

An entity has (at most) one value for a *single-valued attribute*.

Definition 18.10: Optional Attribute

An entity may or may not have a value for an *optional attribute*. If an attribute has no value, this is often represented as **NULL**.

A single-valued optional attribute has either one value or no value. A single-valued non-optional attribute has exactly one value. A multi-value optional value has either no value, one value, or multiple values. A multivalued non-optional attribute has one value or multiple values.

A multivalued attribute can be modeled as an ellips with a smaller ellipse inside in an ERD. Optional attributes can be signified in the ERD by small empty bubbles at the end of the lines connecting them to the entity types.

For example, we notice that a student may have multiple mobile phone numbers. This can be modeled with a multi-valued attribute. Upon closer inspection, we notice that the mobile phone number attribute should probably be modeled as optional multivalued attribute. While it seems very unlikely nowadays, there may be students who do not have a mobile phone number. Well, maybe a foreign exchange student, i.e., a 留学生, who enters China does not yet have a mobile phone number when enrolling into our university. So making this attribute optional is reasonable.

While we are on the subject of foreign exchange students: They probably do not have a Chinese ID (中国公民身份号码[507]). So we turn ID into an optional single-valued attribute. We update our ERD in [Figure 18.2](#). The DBS could now decide whether to fully spell out the name, e.g., in graduation certificates, or whether to just print the salutation and family name, e.g., when sending messages addressed to the students. Now we also are able to represent that fact that a student can have no or multiple mobile phone numbers. They also can have no or one ID.

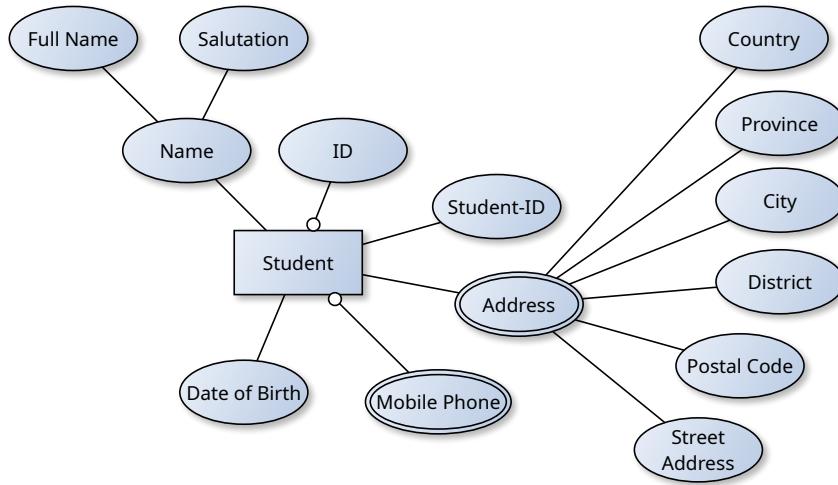


Figure 18.3: A new version of the *Student* ERD from Figure 18.2, this time with *Address* as multivalued and composite attribute.

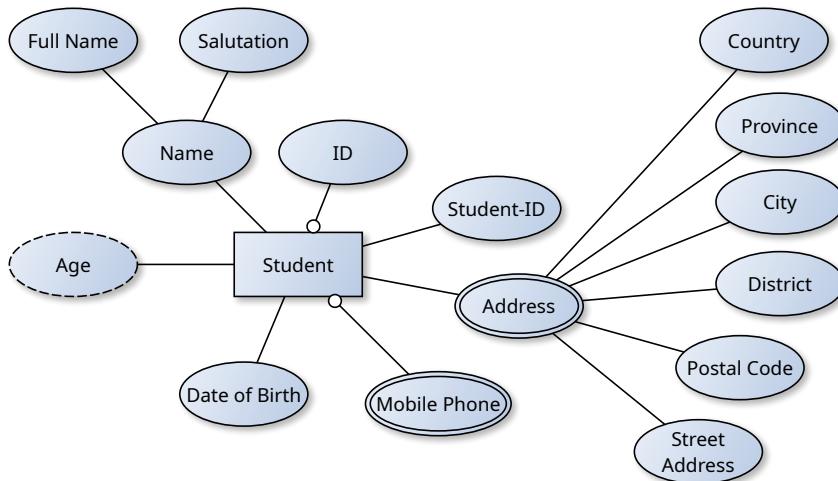


Figure 18.4: A new version of the *Student* ERD from Figure 18.3, where we added the derived attribute *Age*.

While we are at this, we realize that *Address* should also be a composite attribute. An address is not just a line of text, but consists of components such as country, province, city, district, postal code, street and street number, quarter, building, and flat number. Modeling that would be cumbersome. We propose that everything after the postal code could simply be merged into one string named *street address*, because an automated system probably cannot really handle information at a finer granularity than postal code in any meaningful way. While discussing the subject with our partners, we also realize that *Address* should be a multivalued attribute, i.e., a student can have multiple addresses. This is illustrated in Figure 18.3.

Finally, there are also so-called derived attributes.

Definition 18.11: Derived Attribute

A *derived attribute* is calculated based on the values of other attributes.

Derived attributes do not need to be stored in the DB. A typical example is the age of person. If we know the DOB and the current date, the age can directly be calculated. This can be done very quickly. There is no reason to store the age in the DB. Derived attributes are illustrated in ERDs by ellipses with dashed lines around their perimeter, as shown in Figure 18.4.

At this stage, we have learned several things about entities and attributes. An entity models one real-world object, such as one person, one thing, one location, one event, or one concept. An entity does not just exist. It is a tuple of its attribute values. While entities represent single objects, entity types represent classes of objects, such as people, addresses, a type of events, or a set of concepts.

Attributes can have different properties as well [393]: Attributes can be *simple*, which means that they have atomic values that cannot be subdivided any further, like phone numbers. Attributes can be *composite*, which means that they consist of parts. An address, for example, can be divided into country, province, city, etc.

It is not always immediately clear how an attribute could be modeled. For example, our students have the attribute *Date of Birth (DOB)*. Technically, we have learned back in Section 9.3 (The Table "demand") that dates are atomic datatypes in *SQL*. So naturally, we would model the DOB as an atomic attribute. Of course, we could also model it as a composite attribute consisting of year, month, and day. No sane person would ever do that, though, because then you have to manually consider things like time zones and different calendars... Then again, an address could also be represented as a single string of text instead of using a composite attribute.

The decision of how to model attributes probably depends on which data we need. For example, if we store dates as, well, atomic dates, then it is extremely easy and fast to calculate the year of birth or the age of a person. Storing years separately would be useless and just complicate and slow down the DB. For an address, however, extracting the country from an address string could be tedious and error prone. And we would probably need the country in almost any use case where an address is required.

Anyway, besides being either simple or composite, attributes can also be single-valued or multi-values. Students can have multiple addresses, multiple phone numbers, but only a single DOB.

We are now able to model the different types of data that we want to store in our DB. However, a relational DB is not just *data*. It also comprises constraints on the data as well as the relationship between them. And next we will explore one special constraint...

18.2 Keys

In Definition 18.1, we stated that an entity can be distinguished from all other entities in the world. This means that it must be unique. Entities are characterized entirely and only by their attributes. They do not have an property beyond their attribute values. The only way it can be unique is because of its attributes.

The attributes that can serve as unique identifiers are called *keys* [393].

Definition 18.12: Super Key

A *super key* is an attribute or set of attributes of an entity type that uniquely identifies an entity in an entity set [184, 393].

A student entity, for example, can uniquely be identified by the student ID. It can also be uniquely identified by the combination of the address and mobile phone number(s). Or just by the mobile phone number... if these were not optional. Or maybe by an email address. Or by the government-issued ID... if these were not optional. Or we could try using the name, address, and DOB.

Definition 18.13: (Candidate) Key

A *key* (or *candidate key*) of an entity type is a minimal super key, i.e., a super key that either consists of a single attribute or that would lose its unique property if a single attribute was removed from it [184, 383, 393].

This does not necessarily mean that all candidate keys have the same number of attributes. For example, when identifying a person, one candidate key could be the government issued ID number. Another possible candidate key could be a combination of the name, place of birth, and DOB. Both would be minimal in the sense of the above definition. The government-issued ID, however, would consist of a single attribute whereas the other choice consists of three. The combination of all four attributes would be a super key, but not a candidate key. Another super key would be the combination

of name, gender, place of birth, and DOB. It would not be a candidate key, because we could remove the attribute “gender” without impairing the uniqueness property.

If we ignore the issue of some attributes being optional for now, then we have at least three different candidate keys for students: the student ID, the government-issued ID, and the mobile phone number.

Definition 18.14: Primary Key

The *primary key* of an entity type is a *candidate key* that is used as *the* identifying attribute or group of attributes of an entity when modeling relationships between different entity types.

Definition 18.15: Prime Attribute

An attribute is referred to as *prime* if it is part of the primary key.

An entity cannot be referenced in one place using some attributes and in another using other attributes. That would lead to all kinds of problems and inconsistencies. Thus, there can only be one such primary key for each entity type.

Later in our DB design, we will also model how students enroll into modules. Thus, we then also will have another entity type for modules (or module instantiations in specific semesters). It will be necessary to establish relationships between student and module-instance entities. We already did something like this back in our factory example. You do remember the [PRIMARY KEY](#) and [REFERENCES](#) keywords from back then. Of course, these are technology-specific things which we do not care about at the conceptual level. However, even at this level, we still must decide about which attributes we will *actually* use to identify entities.

We will need one primary key that is used to uniquely identify students. Which of the candidate keys makes the most sense?

How about mobile phone numbers? Phone numbers may change. Primary keys must never change. Also, a student may have multiple phone numbers. This would feel awkward to use a primary key. Actually, some students might not have a mobile phone number. This may be rare, but it could happen, as we already discussed before. Primary keys must never be [NULL](#). So we rule out phone numbers as primary keys.

This is a very similar reason to *not* use the government-issued ID number (中国公民身份号码[507]) candidate key as primary key: Foreign exchange students, so-called 留学生, do not have IDs issued by the Chinese government. As said, primary keys should never be [NULL](#), so we rule out government-issued IDs as well.

Another candidate key could be a combination of the name, place of birth, and [DOB](#). Another criterion for primary keys is that they should be reasonably small. For example, composite attributes or attributes that consist of longer texts are not very suitable. If we store relationships between entities, then we do this by storing their primary keys. Recall, for example, our [demand](#) table from back in [Section 9.3](#). This means that primary keys are not just stored as part of their original entities, but also as part of all of the relations they are involved in. While this is a technological aspect that does not really belong into the conceptual model design stage, it is something that we should keep in mind: Huge keys are bad. And the combination of name, place of birth, and DOB would be fairly impractical to store in multiple locations. So we rule it out, too.

We would naturally prefer the student ID that the university itself issues. The reason is as follows: A student joins a curriculum, maybe studies in the Bachelor Program “Computer Science and Technology” at our university. For this process, they receive a student ID. This student ID does not just represent them as a person, but it represents them “as a person in the function ‘BSc student of Computer Science and Technology.’” Later, after graduation, they may join a Master’s program and get a new, different student ID.

This realization makes us feel a bit anxious about or concept of modeling students... We may have two student entities referring to the same person, but at different stages of their educational process. Maybe we should model students differently? We will see. For now, we accept this and use the student ID as primary key. From the available candidate keys, it is the best option.

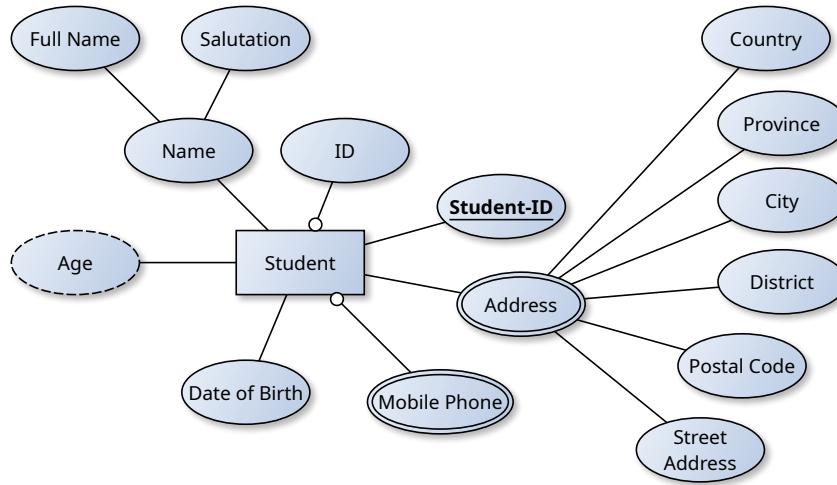


Figure 18.5: A new version of the *Student* ERD from Figure 18.4, this time with *Student-ID* marked as primary key.

We update our ERD from Figure 18.4. In the new Figure 18.5, the attribute *Student-ID* is marked as primary key. This is done by underlining the attribute name [184].

We learned a few things about primary keys. Let's re-iterate them:

Best Practice 15

Primary keys should:

1. be unique for each entity (obviously),
2. be immutable over the lifetime of an entity,
3. not be optional, i.e., they should never be allowed to be `NULL`,
4. not be derived attributes,
5. always be single-valued attributes, i.e., not be multivalued attributes,
6. consist of single attributes, i.e., not be based on candidate keys consisting of multiple attributes,
7. be simple attributes, i.e., not composed attributes,
8. be small in terms of the expected required storage size (see also Best Practice 9).

Sometimes, it can happen that we end up with entity types where *no* suitable primary key exists. Maybe all the attributes that form candidate keys are just too long. In such a case, we can use a technique we already learned in back in Section 9.1 (The Table “product”):

Definition 18.16: Surrogate Key

When no suitable candidate key for an entity type exists, an artificial key, such as an auto-incremented integer value, can be used as *surrogate key*.

Keys are a very important component when we model the real world. Each object in the real world must be unique in some aspect. Each object must be identifiable by some unique properties. The objects are represented as entities. Their properties are represented as attributes. Those attributes that we can use to identify them form the candidate keys. Those attributes that we *actually* use to identify them form the primary key. Primary keys must be unique and small.

18.3 Relationships

If we only had a single entity type in our DB, the using a DB makes no sense. In that case, we would be better off using simple documents, like CSV or XML files. However, clearly, in our teaching management platform application, we are going to have many different entity types. And again, if these entity types were unrelated, say "Student," "Weather," "Product," then we were still better off storing them in a bunch of files. But our entity types are going to be *very* related. So how do we model these relationships? Let us begin with some basic definitions [184].

Definition 18.17: Relationship

A *relationship* (instance) is an association of two or more entities.

An example of a relationship would be *Mr. Bibbo enrolls into the module Programming with Python*.

Definition 18.18: Relationship Type

A *relationship type* is the set of all relationships possible between two or more sets of entities.

The *Enrolls* relationship type could be defined between the *Student* entity type and the *Module* entity type. We notice that (transitive) verbs [156] in the requirements definition often represent relationship types [78].

Definition 18.19: Degree of Relationship

The degree of a relationship (instance or type) refers to the number of participating entities.

There can be binary relationships, i.e., relationships where two entities participate. For example, we could model the student-module relationship in a binary fashion: *Student enrolls into Module*. We could just as well use a ternary relationship with three participating entities instead, e.g., *Student enrolls into Module taught by Professor*. And so on.

Definition 18.20: Roles in a Relationship

Each entity participating in relationship may have a role, which defines the way in which the entity participates in the relationship.

Again imagine the ternary *Student enrolls into Module taught by Professor* relationship. Here, the student entity has the role *enrolls*. The professor entity would have the role *teaches*.

Definition 18.21: Relationship Attribute

A relationship type can have attributes describing properties of the relationships it embodies.

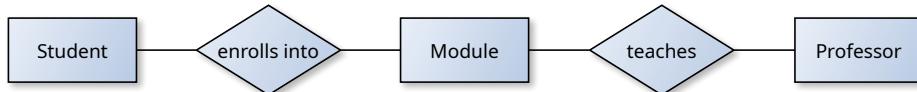
For example, we could write something like *Mr. Beppo enrolls into module Databases in summer semester 2025*. The attribute *Semester* of this relation only makes sense in this context. It neither belongs to the student *Mr. Beppo* nor does it belong to the module *Databases*. Different from entities, relationship types do not have key attributes. The single relationships are identified by the primary keys of the participating entities [184].

Let us start modelling relationships. We begin by representing the fact that a student can enroll into a module. Relationships (or better, relationship types) in ERDs are drawn as diamonds that are connected to the involved entities (or better, entity types). Figure 18.6.1 shows an ERD where the student entity type is linked to a module entity type by the relationship type *enrolls into*. This is a binary relationship, because two entities take part in it.

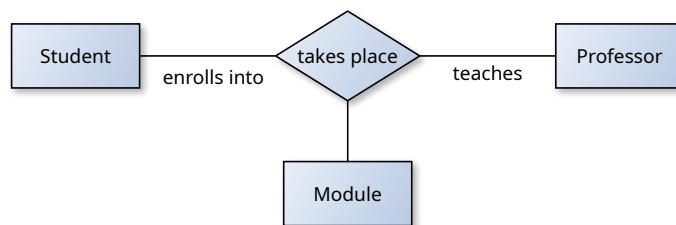
This diagram is interpreted as follows: Entities of type student can enroll into an entities of type module. At first glance, this sounds OK. However, then we notice several problems with this.



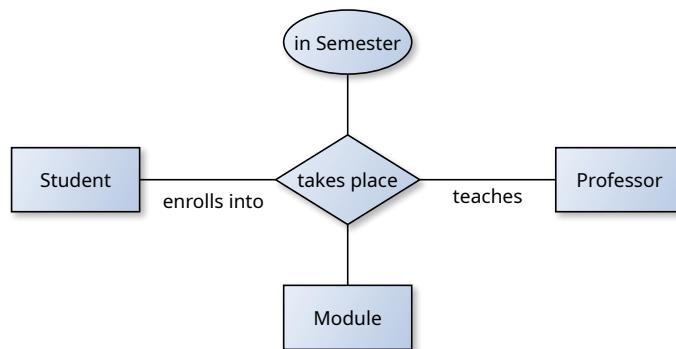
(18.6.1) The binary relationship of student and modules, which does not represent the relationship of professors to modules and students.



(18.6.2) Two binary relationships, i.e., the relationship of student to modules and the relationship of professors to modules. This does not represent that a student enrolls into a course taught by a professor.



(18.6.3) The ternary relationship of students, modules, and professors. This represents how students join a course taught by a specific professor. However, it would not permit the same student enroll into the same course for two years. It also does not give us the information when the course takes place.



(18.6.4) The ternary relationship of students, modules, and professors with the relationship attribute semester.

Figure 18.6: Modeling the relationship between students, professors, and modules.

First, modules are taught by professors. This issue is not represented. Matter of fact, the relationship makes no statement at all about this implicit student-professor relationship.

We try to fix this in Figure 18.6.2 by adding a second binary relationship: Professor teaches module. Sadly, this does not solve the problem at all. We now can properly represent that a student can enroll into a certain module. Multiple students can enroll into one module. We can also represent that a professor teaches a module. Maybe multiple professors can teach one Module. Maybe in this year, Professor Weise (汤卫思) teaches the module *Databases*, maybe next year Professor 李 teaches it. With this model, we cannot represent the information that a student joins a module taught by a specific professor. Because the two binary relationships we drew are independent from each other.

This can be fixed by using a single ternary relationship in our model. The ERD in Figure 18.6.3 shows a relationship where three entity types participate. The professor teaches the module and the student enrolls into the module that the professor teaches. This model is much better.

However, it is still a bit ambiguous. There is no real statement about when the module takes place. Also, we said before that a relationship is represented by the primary keys of all involved entities. What happens if a student takes part in a module but, sadly, fails the final exam or has to repeat the module for some other reason? What if the professor teaches it again? Then we have two relationships to the same entities with the same primary keys. To resolve these issues, we give our relationship the attribute

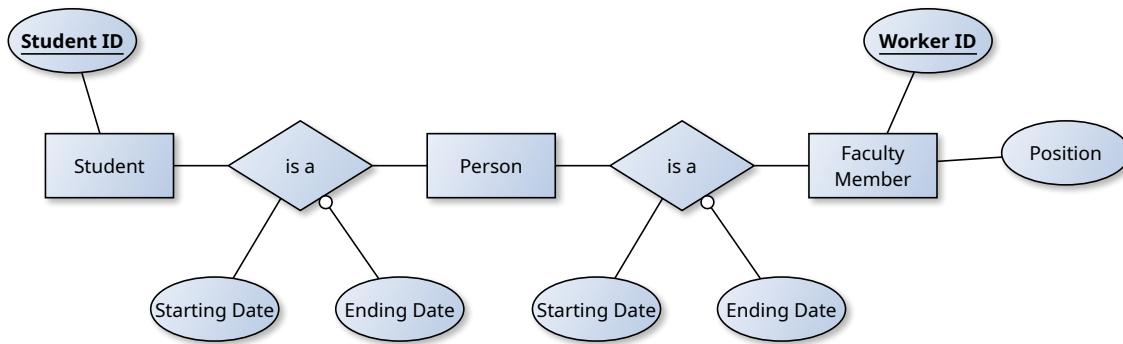


Figure 18.7: An ERD illustrating the new *Person* entity and how students and faculty members are related to it.

in Semester in Figure 18.6.4.

This was rather easy. But let us do something a bit more interesting. Today, we had a meeting with the stakeholders at the university again and showed them our entity model for students from Figure 18.5. It looks quite nice, we are told, but it does not withstand the harsh test of reality.

First, there is the issue with names again. See, people may have *multiple* names. Foreign exchange students, for example, may have their original name. However, they may also have a Chinese name. A *Ms. Elizabeth Prudence McDouglas* may thus also be known as 邓小花女士 in the university. Obviously, for official documents, only the original name of the person is to be used. University-internal files or maybe seating cards in meetings might use the Chinesified name. So the original name may be the name-for-documents, while others are there to allow us to match different documents.

“Original name,” we said, did we? Actually, in the West, it is not uncommon that people change their name when they marry. Let’s say that *Elizabeth* marries *Mr. Heinrich Gieselher von Görlitz-Zittau*. She may decide to keep her name unchanged. She may take on the family name of her husband and become *Mrs. Elizabeth Prudence von Görlitz-Zittau*. Or her new husband could change his family name to hers. The couple may also choose for a composite family name. This may result in a person named *Mrs. Elizabeth Prudence von Görlitz-Zittau-McDouglas* or maybe *Mrs. Elizabeth Prudence McDouglas-von-Görlitz-Zittau*. (Notice that this would look interesting on Chinese official documents where the space for names is usually calculated to not exceed five characters...) Either way, there can be a variety of reasons why a person may have multiple names or why the name of a person changes.

However, changing the names in the DB is not an option. Because this would mean that the old name “disappears”. Then we would eventually have older documents in the real world that can no longer be matched with the updated data in the DB. So we need to make names a multivalued attribute, too. When doing this, we may decide to assign names a valid-from and valid-to attribute, to emphasize the time when the names changed.

In the moment where wanted to begin modelling the names, we realize that this is not a student issue. The same issue will later reappear when we model the employees of our university. They, too, may have dodgy name issues. They also have mobile phones, IDs, and addresses.

Now we do not want to model the same stuff twice. That just makes the system more complicated and introduces the chance for inconsistencies. Thus, before moving on, we decide to make short work of this problem by introducing the new entity type *Person*. Students and employees are persons. We will put all the common attributes of students and employees into the person records.

This also makes sense because the same person could enroll in multiple curricula. A student could first do the BSc in computer science and later do an MSc. Theoretically, even a teacher might also enroll as a student. Maybe a chemistry lecturer wants to also do a MSc in computer science. They would still be the same person and there is no reason to store all their data separately multiple times.

We sort out all of these issues in Figure 18.7 by introducing the new entity type *Person*. Later, we will hang all the shared attributes on this entity type. For now, we do not worry about what primary key this entity type should get. The entity type *Student* for now only needs the single primary key attribute *Student ID*. A person can be a student, which we model with a relationship type. Such a relationship has a certain start and an optional end date. The end date can be **NULL** for all students

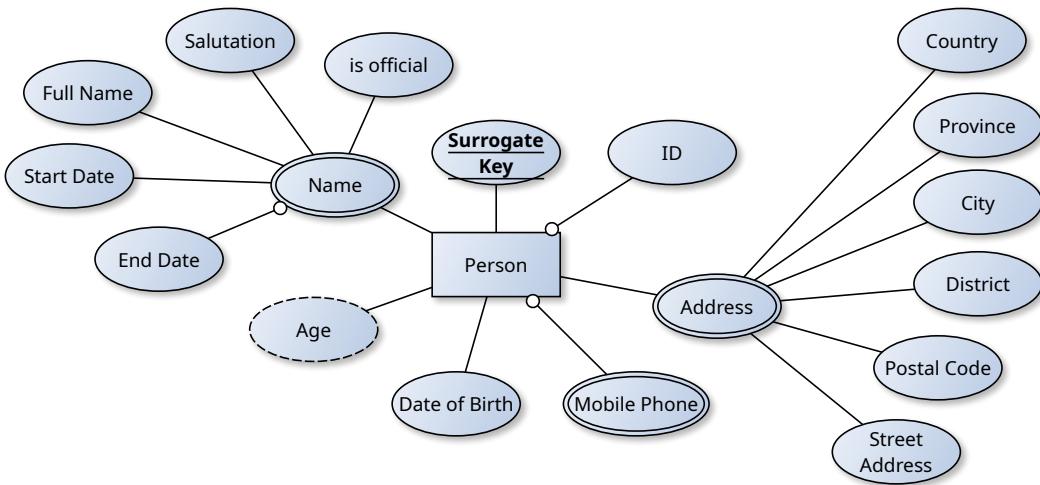


Figure 18.8: The *Person* entity type with the attributes of the *Student* entity type taken from Figure 18.5 and the improved *Name* attribute.

that currently are enrolled and therefore is an optional attribute.

For faculty members, the situation is quite similar. They are uniquely identified by a worker ID. They also have a position, e.g., lecturer, associate professor, or full professor. This function, too, has a start date and an optional end date.

In Figure 18.8, we copy the attributes from the old *Student* entity type in Figure 18.5 to the new *Person* entity type. Of course, we now do no longer include the student ID. We also fix the *Name* attribute. It is now a multivalued attribute. Each name now also has a start and an optional end date associated with it. A name can be *official* (True/False), which means that it is used for documents. This should help to sort out situations such as name changes nicely. We will also define the constraint that exactly one name of a person must be the *official* name and that this name must have a start date no later than the present date and no end date.

We imagine that our system will not bother the data-entry person too much with these options. They can just enter one name. Its start date will be set automatically to today and it will automatically be marked as official. The system will automatically set the salutation and the full name to be the same. The administrative person working with this data will be able to change either of them, to add new names, mark names as official, change dates, and so on. So there would not be much hassle when entering the data but the *option* to deal with all sorts of name-related issues. And since our university is an international university with students and faculty from all over the globe, it is important to consider such issues from the beginning.

To our dismay, we realize one problem, though: We can no longer spot a reasonable candidate for primary key in our *Person* entity type. *Name* and *Address* both are multivalued composite attributes that do not need to be unique. The government-issued *ID* is optional. The *Mobile Phone* number is both optional and multivalued. *Date of Birth* is certainly not unique and *Age* is additionally derived. None of them and neither combination is non-optional and necessarily unique.

Of course, later, we will add other ID values like passport numbers and email addresses. An application could enforce that each person has either a passport or a government-issued ID and that their numbers be unique. However, it would still feel awkward to somehow create a Frankenstein-esque primary key out of this. The easiest solution here is to use a surrogate key. In our model, we actually call it *Surrogate Key*, to avoid any form of misunderstanding. We will assume that whatever DBMS we will eventually use, it will be able to somehow generate unique values, like back in Section 9.1 (The Table “product”). Our *Person* in Figure 18.8 now looks fairly reasonable.

Having solved this problem, we now want to clean up the issue of IDs and ID documents. So far, we modeled the ID as the government-issued ID (中国公民身份号码 [507]). It is an optional attribute, because foreign exchange students as well as foreign employees do not have one. On one hand, even foreigners do have ID documents – just not Chinese ones. These foreign documents are, of course, useless in our context. On the other hand, a foreigner entering China must have a passport [279].

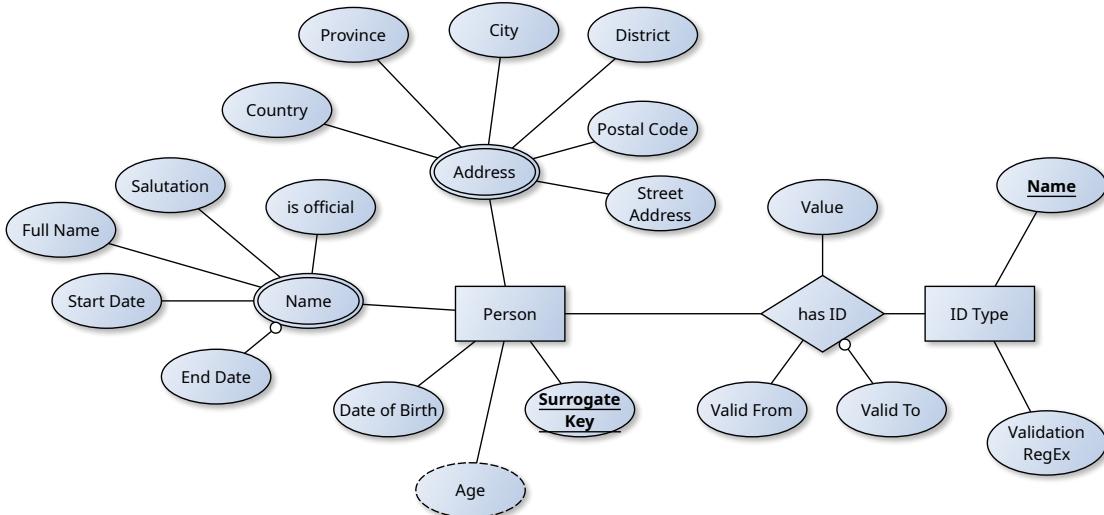


Figure 18.9: We now add the ID entity type to our *Person* ERD from Figure 18.8.

A passport has a unique number ... it is not a *permanent* ID, though. While Chinese ID numbers identify one person and stay the same, the passport number is usually a number identifying the passport booklet [462]. A passport booklet usually stays valid for ten years and then a new passport is issued to the person, usually having a new passport number. When a foreigner enters China, they need a visa, which, in turn, also has a unique identification number and a time window of duration [509]. There are many different types of visa that are for different activities, for example X1 and X2 visa are for studying and Z visas are for working. Foreigners who work in China (e.g., as professors) furthermore need work permits [506], which also have unique identifying numbers. Of course, visas and work permits eventually expire and need to be renewed. This allows for an arbitrarily complex and ugly model.

How about this: We create an entity type to represent *ID Types*. An entity of this type will store the name of the ID type, which is the primary key of this entity type. This could be “Mobile Phone (China),” “Mobile Phone (Intl),” “ID (中华人民共和国居民身份证),” “Passport (护照),” “Work Permit (中华人民共和国外国人工作许可证),” “X1-Visa,” “X2-Visa,” “Z-Visa,” “Email Address,” “WeChat User Name,” whatever. Notice how this makes our system future-proof: While the importance of Emails is currently fading and WeChat has become the main communication device, maybe something new will emerge later. Or maybe there will be new visa types later on. We could then just add a new ID type. We could also extend this entity type with attributes, such as *is for communication*, *is ID document*, etc., to add more context.

Either way, using this idea, we have unified all communication and identification values into one entity type. We can also store multiple ID values and multiple phone numbers or email addresses for each person. Each ID value would be associated with an ID type.

This, however, creates the problem how to validate ID values. If a new ID-value is entered, there should be at least some basic sanity check. Mobile phone numbers and passport numbers and ID numbers are very different. Yet, we must be able to prevent, e.g., letters from occurring in a phone number. We could leave this, to some degree, to the application that we will build on top of our DB.

But we can also add some very basic method to check ID values. Back in Section 9.2 (The Table “customer”), we learned about **regexes**, i.e., text strings that describe patterns can be matched against other strings. We will simply store one regex as “Validation RegEx” attribute for each ID type entity. When the administrator enters a new ID of a given type, this value will be matched against this the corresponding pattern. While we cannot emulate checksums or other advanced validation techniques, we can this way at least ensure the right amount of characters or numbers for the IDs.

We can now introduce the new relationship type between the *Person* and *ID Type* entity types: A person has IDs. Each ID has a *Valid From* attribute and may have a *Valid To* date. Figure 18.9 provides an illustration of the new *Person* entity.

Now we can model relationships. Relationships connect two or more entities. With this, we can create models that actually require a DB. Without relationships, we could store our just as well data

in groups of flat files. But as soon as different entities are in relationships, this changes. It cannot be allowed that one entity that is part of a relationship is deleted without first deleting the relationship. Relationships can only be created between existing entities. Therefore, our models now require that the technology with which they will eventually be implemented supports this. And relational databases do.

18.4 Weak Entities

In the previous section, we made some big strides towards properly modelling people in our teaching management platform. A particularly interesting aspect was the modeling of the different types of IDs and communication monikers. Unifying IDs by using the new entity type *ID Type* was a nice idea and we are pleased with ourselves. Then we revisit our new ERD in Figure 18.9.

And we remember something: *Relationship types do not have key attributes. Relationships are identified by the primary keys of the participating entities.* Which entities are participating in our *Has ID* relationship? The *Person* entities, identified by their surrogate key and the *ID Type* entities, identified by their name. If we apply the above rule, then this means that, actually, each person can only have one ID of any given type. One mobile phone number. One passport number. One email address.

In other words: We were happy too early. Passports and visa expire and it is very possible that foreign members of our uni, during their stay, will have multiple different ones. We did not yet solve the problem completely.

At least, we did not solve it well in terms of the conceptual model. If we were to technically implement this model as a PostgreSQL DB, we could probably realize it in the way we “meant” it without too much of a hassle by using multiple tables and foreign keys and such and such. But we want to do this properly also on a conceptual level. Because the conceptual schema is a definition of the data that will go into our DB. It should be correct. The logical model that we design later must not diverge from the conceptual model. This would eventually result in a pure nightmare for maintenance. Luckily, the solution for this problem can be found in the different classes of entity types [393].

Definition 18.22: Strong Entity

A *strong entity* exists independently of the other entities and has an own primary key.

So far, we have modeled all of our data as strong entities. However, there are also weak entities [330, 377, 393].

Definition 18.23: Weak Entity

A *weak entity* is only identifiable by its attribute values *and* the primary key(s) of one (or multiple) entities.

A *weak entity* cannot exist on its own. Its existence depends on at least one other entity, which is called its *owner*. It does not have an own primary key. It has a partial key, i.e., a set of attributes that, combined with the primary keys of its owner(s), can be used to identify it in its entity set.

Definition 18.24: Identifying Relationship

An *identifying relationship* links a strong entity to a weak entity and is required for identifying the weak entity.

In an ERD, strong entities are symbolized by rectangles. So far, we only modeled strong entities. Weak entities are represented by double-lined rectangles. The identifying relationships that connect them to other entities are symbolized by double-lined diamonds. (Actually, we are talking here about entity types and relationship types, but writing this makes the text much harder to read...)

If we look at our “ID problem” again, we could model the *has ID* relationship as a weak entity. The weak entity would be identified by its own attribute values together with the primary key of the

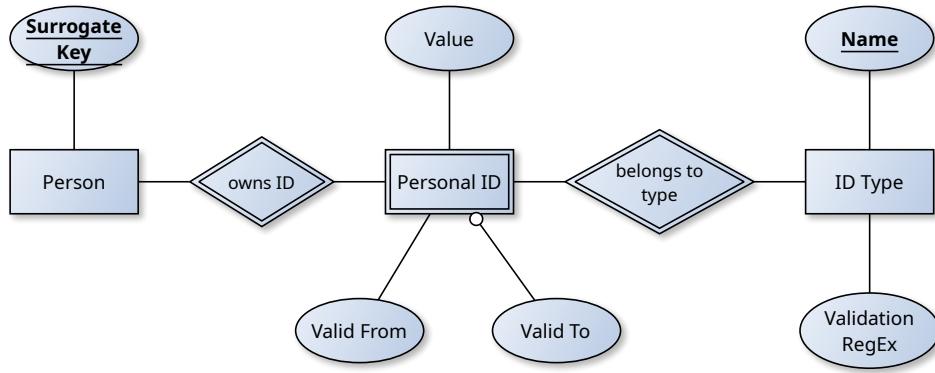


Figure 18.10: An improvement of Figure 18.9: We now use weak entities to represent the ID values of a person.

corresponding *Person* entity and the primary key of the corresponding *ID type* entity. It would be a weak entity depending on two strong entities.

Figure 18.10 illustrated this new situation. The new weak entity is called *Personal ID*. It has the same attributes as the relationship in Figure 18.9. It is linked with identifying relationships to both the *Person* and the *ID Type* entities. Each *Personal ID* entity is uniquely identified by its attributes *Value*, *Valid From*, and *Valid To* (which form the partial key) together with the primary keys *Surrogate Key* of the *Person* entity and *Name* of the *ID Type* entity. Now a person can have multiple phone numbers, provided that they are different or valid at different time ranges. A person can have different visas, because they will usually have different visa numbers. Now we indeed have unified communication-based IDs such as mobile phone numbers, email addresses, WeChat IDs together with document-based IDs such as, well, actual government issued IDs, visas, work permits, driver's licenses, etc. And we can create new forms of ID if need be.

Definition 18.25: Associative Entity

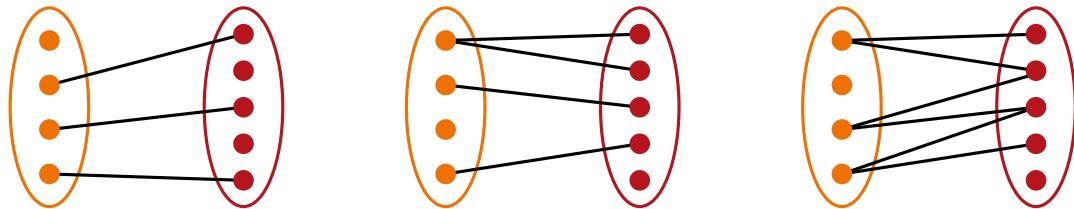
An *associative entity* (type) is an entity (type) that has attributes and a primary key, but also serves as relationship that can link entities together.

In an *ERD*, an associative entity is symbolized by a rectangle with a diamond inside. They are mainly used to normalize and simplify many-to-many relationships. They usually connect entities that have multiple interactions with each other. However, we skip them here, as I see them more as an intermediate step when mapping entity models to *relational database* models. At this stage, however, we do not want to concern ourselves with the *relational DB* structure (yet). It instead is our goal to more freely model the data structures that we will have to deal with (and, hence implement) in our teaching management platform.

Be that as it may, we again made an important step forward. By using weak entities, we can now cleanly model one-to-many and many-to-many relationships. As example for this, we had the “has ID” relationship. By using a weak entity type to replace this relationship, a person can have multiple IDs of the same type. Since we also model email addresses and mobile phone numbers as IDs, that is an important feature. Other candidates for using weak entities would be the relationships between professors, students, and modules. This would allow us to model a situation where a student enrolls into the same module taught by the same professor in different semesters more cleanly.

18.5 The Cardinality of Relationships

We already learned that attributes can have different cardinalities: There can be single-valued or multivalued attributes and either can be optional. Figure 18.11 sketches a set of basic examples for relationship cardinalities. Commonly, we distinguish one-to-one, one-to-many, and many-to-many relationships. This is embodied by Chen’s original *ERD* notation [80], where the relationship ends are simply annotated with 1, N, or M to express 1:1, 1:N, or M:N relationships. In notation by Bachman [13]



(18.11.1) A 1:1 or one-to-one relationship where participation on both ends is optional. An example would be the relation between students and graduation thesis topics.

(18.11.2) A 1:N or 1-to-many relationship where participation on both ends is optional. An example would be the relation between supervising professors and students.

(18.11.3) An N:M or many-to-many relationship where participation on both ends is optional. An example would be the relation between students and course enrollments.

Figure 18.11: Some simple examples for relationship cardinalities [377, 468].

from back in 1969, a relationship is represented by an arrow from one entity to another. The entity to which the arrow points may occur N times, then one from which the arrow line originates once.

When creating an entity-relationship model, we often want a finer granularity. We want to express whether relationships are optional or mandatory (required) on either end of the relationship. Therefore, an end of relationships can be annotated with its modality (is it optional or mandatory) and with its cardinality (the number of participating entities) [349].

Definition 18.26: Relationship Modality

The *modality* of a relationship end defines whether participation is optional or mandatory. It can be viewed as the minimum number of participating entities.

From a practical point of view, only the minimum participation numbers of 0 and 1 are really relevant. A modality of “optional participation” means that the minimum participation number is 0. In this case, one entity occurrence does not require a corresponding entity occurrence in a particular relationship. A modality of “mandatory participation” means that the minimum participation number is 1. In this case, one entity occurrence requires a corresponding entity occurrence in a particular relationship. This allows us to distinguish total and partial relationships [332, 468]. If all entities of an entity set *must* participate in at least one relationship, then this relationship end is *total*. If only some entities participate in a relationship, then this relationship end is *partial*.

Definition 18.27: Relationship Cardinality

The maximum number of participating entities of a relationship end is called *cardinality*.

Practically, we usually only distinguish the cardinalities *one* and *multiple*, i.e., unlimited. In the similar modelling language UML, cardinality is called multiplicity.

There are multiple conventions on how to express the relationship end modality and cardinality in an ERD. The top part of Figure 18.12 presents two of these methods. On one hand, there is the Crow’s Foot notation [66, 159, 393], where a graphical notation is used to express the cardinality and modality of a relationship end. It uses the four symbols $\text{+}\text{o}$, $\text{>}\text{o}$, ++ , and >+ to define optional one and many as well as mandatory one and many participations, respectively. On the other hand, we can also directly write the minimal and maximal permitted number of participants as labels on the ends of the relationships [332]. Here, an integer range $i..j$, where i is the minimum number of participating entities, j is the maximum, and $j = *$ stands for unlimited, multiple, many, or infinity. The second method has the advantage of permitting much more diverse ranges of cardinalities. The drawback is that it is slightly harder to read when printed on paper or from afar. More importantly, it is harder to draw, because we need to manually add labels to the relationship ends. Also, as said, a finer granularity of cardinalities is often not needed. Cardinalities like $7..11$ could potentially be nightmarish to implement and enforce later on.

In LibreOffice Base, ERDs can be drawn that are directly linked to the underlying DB. They are in the 1:1:n notation and the lowercase n represents “multiple”. Microsoft Access offers a similar

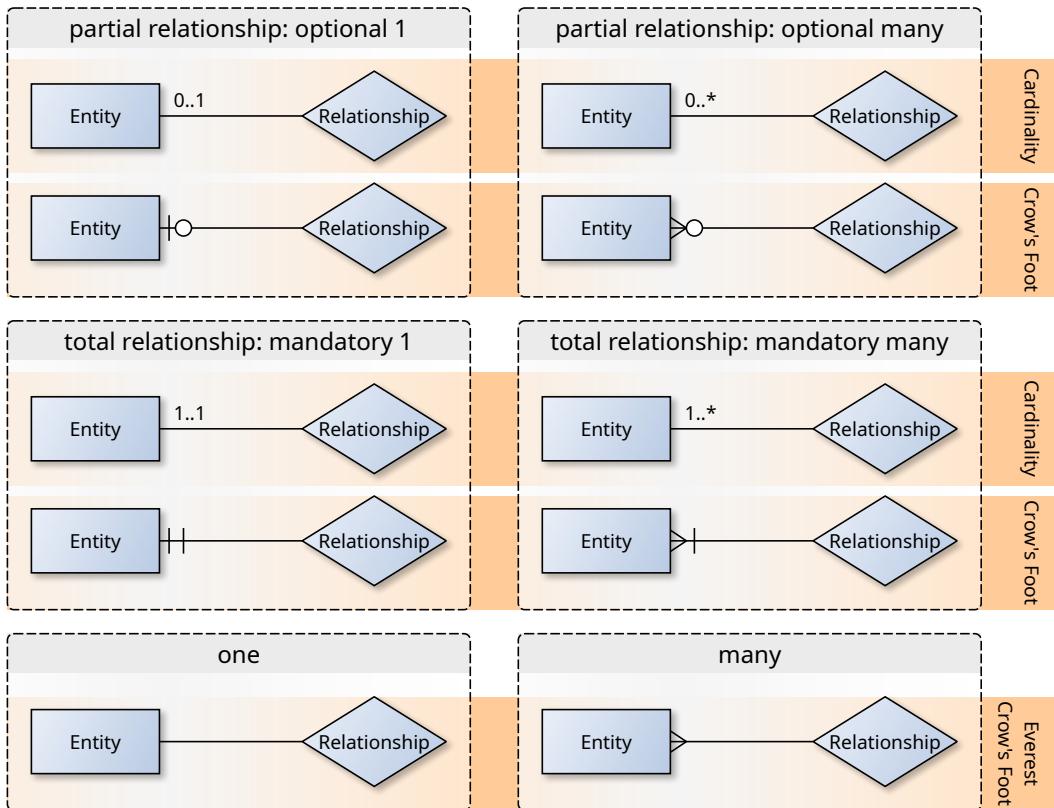


Figure 18.12: Three different ways to express possible modalities and cardinalities of relationship ends in ERDs.

functionality, but the relationship ends are annotated with 1 or ∞ . For the remainder of this text, we will stick to the Crow's Foot Notation to signify the relationship modalities and cardinalities, simply because we can paint this more easily with *yEd* without needing to add labels to relationship ends. For now, we keep using Chen's notation for the symbols, though.

Notice that the original Crow's Foot method by Everest [159], also sketched in Figure 18.12, which did not yet have the mandatory/optional symbolism. This method can be used when the modality is not important during modeling, or when we want to leave it open and settle it during a later discussion.

Of course, there are many more possible notations to express the cardinality of relationships. We could write (i, j) instead of $i..j$ to express the possible numbers of participating entities [184, 377]. Instead of using $*$ as symbol for unlimited / many, some write M or N. Different variations of the arrow-based notation are still in use as well. An arrow touching the relationship diamond means "one" in [468]. In [184], the arrow needs to instead touch the entity rectangle and means either ≥ 0 or ≥ 1 . Nice overviews can be found in [57, 410]. Which notation is used probably does not matter much, as long as all stakeholders agree upon and understand it.

As said, we will stick with the Crow's Foot notation. Let's write down all the possible combinations of "relationship ends" in Figure 18.13 for this notation. There are four possibilities to annotate the end of a relation:

- Optional 1: $+O$, equivalent to $0..1$,
- Mandatory 1: $++$, equivalent to $1..1$,
- Optional Many: $>O$, equivalent to $0..*$, $0..N$, and $0..\infty$, and
- Mandatory Many: $>+$, equivalent to $1..*$, $1..N$, and $1..\infty$.

Since each relationship has two ends, this gives us $10 = \frac{(4+2-1)!}{2!*(4-1)!}$ different combinations for binary

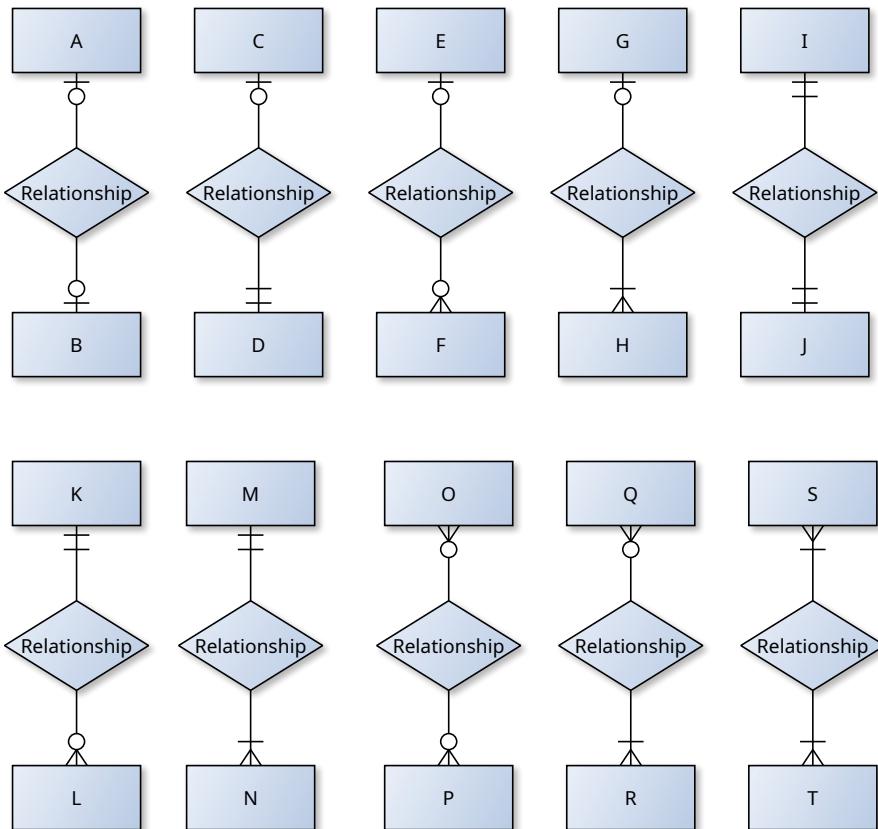


Figure 18.13: The ten possible combinations of cardinalities in binary relationship types expressed with the Crow's Foot notation.

relationships.¹ It is important to understand how to interpret the cardinalities and modalities of the relationship ends, because this can easily be mistaken: The participation of an entity depends on the *other* end. Let's take K $\text{++}\rightarrow\text{L}$ as example. First, place your finger on the K as say "Each K has...". Now move your finger towards the L and read the relationship end there, which says: "... zero, one, or multiple L." [212]. Then place your finger on the L and say "Each L has...". Move the finger to the K and read the relationship end touching the K. The symbol there means that "...exactly one K." Notice that the ++ touching K does *not* mean that "Every K must be related to some L." To be on the safe side, let's write down the meaning of each of the possible binary relationship types based on Figure 18.13.

- A $\rightarrow\!\!\!\rightarrow$ B: Each entity of type A may be linked to zero or one entity of type B. Each entity of type B may be linked to zero or one entity of type A. [295]
Example: When issuing an order for goods online, a customer may enter a discount code. Each discount code must be used at most once. At most one discount code can be used for an order. [295]
Example: One person maybe married to another person. [349]
See later in Section 19.2.2.1.
 - C $\rightarrow\!\!\!\rightarrow$ D: Each entity of type C must be linked to exactly one entity of type D. Each entity of type D may be linked to zero or one entity of type C. [452]
Example: Each office in an office building may host zero or one salespersons. Each salesperson must have one office. [452]
Example: A professor may be the dean of school or not be a dean. A school must have exactly one dean. [349]
See later in Section 19.2.2.2.

¹The number of multisets of size k with elements from a set of size n is $\frac{(m+k-1)!}{k!*(n-1)!}$ and that's the number of combinations of k out of n with repetitions but without order.

- E $\rightarrow\!\!\!\rightarrow$ F: Each entity of type E may be linked to zero, one, or multiple entities of type F. Each entity of type F may be linked to zero or one entity of type E. [273]

Example: Assume that for bank accounts, two addresses may be associated: A home address is required, which is not relevant for this example. What is relevant is that additionally, a bank account may be linked to zero or one postal address. Each address may be the postal address of zero, one, or multiple bank accounts. [273]

See later in Section 19.2.2.3.

- $G \rightarrow\!\!\!-\!\!\!-\! H$: Each entity of type G is related at least one, but maybe multiple entities of type H. Each entity of type H is linked to zero or one entity of type G.

Example: The trainer of a football club coaches multiple club members (but always at least one, otherwise they are not a trainer...). A club member can be coached by at most one trainer or not be coached at all, if they have other functions and do not play actively [210].

See later in Section 19.2.2.4.

- $I \rightarrow J$: Each entity of type I must be associated with exactly one entity of type J. Each entity of type J must be associated with exactly one entity of type I.

Example: Police patrols always are done by a team of two police officers (at least in classic movies and TV shows).

Example: In public bus transportation, it could be that there are always teams of one bus driver and one ticket inspector working together.

Example: In a certain company, each salesperson must be the backup for one other salesperson and vice versa. [452]

See later in Section 19.2.2.5.

- $K \dashv\!-\!\rightarrow L$: Each entity of type K may be linked to zero, one, or multiple entities of type L . Each entity of type L must be linked to exactly one entity of type K . [273, 295]

Example: A customer may issue zero or arbitrarily multiple orders for products. However, each order must be linked to exactly one customer. [295]

Example: A bank account may take part in zero, one, or multiple bank transfers as source account. However, each bank transfer must have exactly one source bank account. [273]

Example: Each customer is handled by exactly one salesperson. A salesperson may not have any customers, one customer, or multiple customers. [452]

$M \sqsubseteq \text{!}N$: Each entity of type M is related to at least one and possibly multiple entities of type N .

M \rightarrow N: Each entity of type M is related to at least one and possibly multiple entities of type N. Each entity of type N is related to exactly one entity of type M. [296]

Example: A patient can make a [redacted] appointment. [206]

- Example:* A school of a university must consist of at least one, but possibly multiple departments. Each department must belong to exactly one school. [349]

See later in Section 19.2.2.7.

$O \rightarrowtail \rightarrowtail P$: Each entity of type O may be related to zero, one, or multiple entities of type P. Each entity of type P may be related to zero, one, or multiple entities of type O.

Each entity of type P may be related to zero, one, or multiple entities of type Q.
Example: A pizza shop sells pizza to customers. A pizza may be ordered by zero, one, or multiple customers. A customer may not order pizza (maybe they want to eat something else), or order one pizza, or order multiple pizzas. [110]

one pizza, or order multiple pizzas later in Section 10.2.2.8.

- $Q \rightarrowtail R$: Each entity of type Q is linked to at least one, but maybe multiple entities of type R . Each entity of type R may be related to zero, one, or multiple entities of type Q . [205]

Example: When placing an order for products online, the order must be for at least one product. Each product may be referenced by zero, one, or multiple orders. [295]

Each product may be referenced by zero, one, or multiple orders. [295]
See later in Section 10.2.2.0.

See later in Section 19.2.2.9.

- **S $\rightarrow\!\!\!\rightarrow$ T:** Each entity of type S is linked to at least one, but maybe multiple entities of type T. Each entity of type T is linked to at least one, but maybe multiple entities of type S. [452]

Example: Each salesperson sells at least one product but may also sell more than one product. Each product must be sold by at least one salesperson, but may also be sold by multiple salespeople. [452]

See later in Section 19.2.2.10.

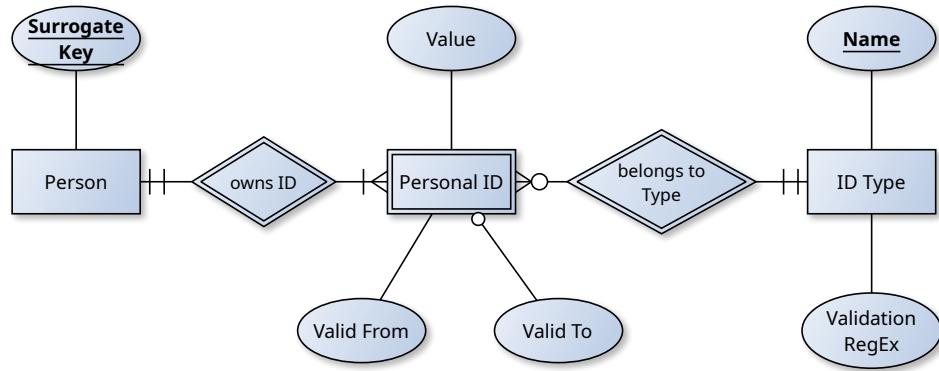


Figure 18.14: The ERD for *Person* from Figure 18.10, but annotated with relationship cardinalities.

So let us now annotate the cardinalities to the relationships of the *Person* entity type from Figure 18.10. This leads to the new ERD in Figure 18.14. We had the idea to have multiple different *ID Types*. Each actual *Personal ID* must belong to exactly one *ID Type*. There may be an arbitrary number of *Personal IDs* for each *ID Type* in our system, maybe none at all, maybe one, maybe many. Hence, the relationship is *Personal ID* \bowtie *ID Type*.

Each *Personal ID* belongs to exactly one *Person*. At the same time, there must be at least one *Personal ID* for every *Person* record in our DB. We cannot have any person in our system whose identify has not been confirmed in at least one official way. Thus, the relationship will be *Person* $\dashv\vdash$ *Personal ID*.

In Figure 18.15, we now expand our ERD regarding the two different types of persons, namely students and professors. We know that each student record must be associated with exactly one person record, because each student is a person and only one person. Then again, a person record may be associated with zero student roles (if they are not a student) or one student role (e.g., they enrolled as compute science Bachelor student). However, a person may also have *multiple* student roles. Maybe they enrolled as computer science Bachelor student, completed this curriculum and then graduated. Then they enrolled in the Master's program for computer science. In this case, they receive a new student ID. Each student role has a starting date and an end date.

Every student must be enrolled in exactly one curriculum instance. There may be zero, one, or many students enrolled in a curriculum instance. What is a curriculum instance? Well, we said that a student is maybe enrolled in the Bachelor program for computer science. When they do so, they enroll in a particular implementation of the program starting in a particular semester, say, the Bachelor program computer science starting in the Winter semester of 2024. From the perspective of entity relationship modeling, we could say: A curriculum may be executed by the university zero (unlikely), one, or many times as curriculum instance. Each curriculum instance is associated with exactly one curriculum. Also, it is associated with exactly one starting semester. Since curricula instances do not have any other identifying characteristics and cannot exist by themselves, they are weak entities. They only come to life through their identifying relationships with the curriculum they belong to and the semester in which they are launched.

We decided to model a semester as independent entity, because this allows us to tag attributes to it. For example, a semester can have a start date and end date, it may even have dates for the exam period. Since we give the start and end date, we could derive attributes such as whether a semester is a summer or a winter semester. Of course, in each semester, our university may launch zero, one, or many instances of curricula.

A curriculum has a unique name, serving as its primary key. Curricula also have further attributes, e.g., whether they are undergraduate or graduate programs. Each curriculum belongs to exactly one school of our university. Each school may have zero, one, or many curricula.

This model describes the situation of the students reasonably well. By tracing the relationships, we know which curriculum a student attends. We did not model this, but you can assume that, to each curriculum, we can relate the modules that it contains and in which semester of the curriculum they need to take place. Hence, we would know when which student should attend which module. We also know to which school a student belongs to. Based on this, the set of professors who could be their supervisor can be constructed.

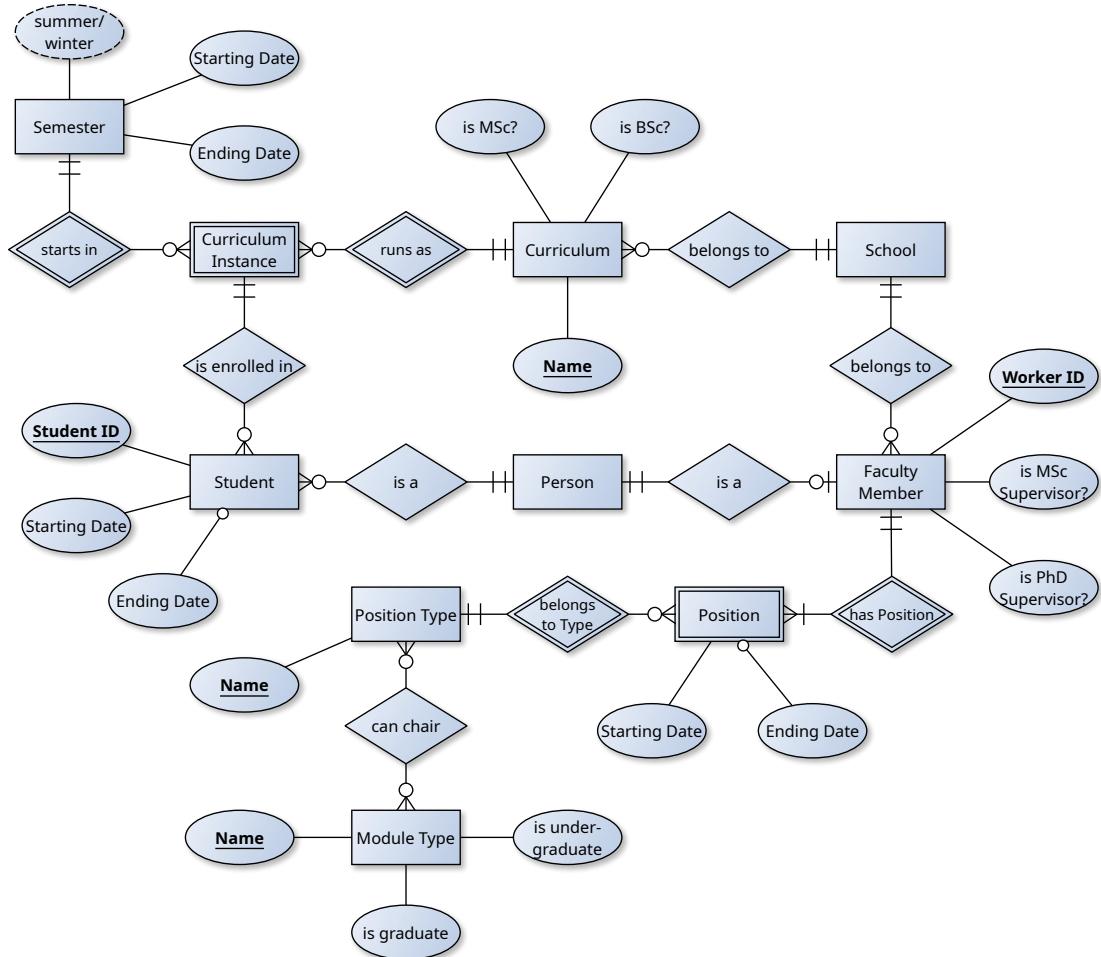


Figure 18.15: A significant extension of the Figure 18.7 ERD describing the relationship between students, professors, and persons. The new diagram also introduces curricula, schools, and positions.

Let us now model a bit of the situation of faculty members. A person can be at most one faculty member. Each faculty member is exactly one person and has a unique worker's ID. This is a bit different from the situation of students: A student gets a new student ID every time they enroll into a curriculum. The most common case is that a student will only study one curriculum in our university. A faculty member, however, will always retain the same worker's ID. A faculty member may be promoted or change their position, but the worker's ID will never change.

There are different types of positions in a university. For example, lecturer, assistant professor, associate professor, full professor, and even different levels of full professorship. Each such position type has a unique name and determines which kind of things a faculty member can do. For example, there are different types of modules, such as Subject Basic Courses (学科基础课), General Basic Courses (公共基础课), Professional Basic Courses (专业基础课), Professional Elective Courses (专业选修课), or General Elective Courses (公共选修课). A position grants the persons in that position the permission to chair modules of certain (an arbitrary number of) module types. (Modules of) each module type may be chaired by faculty members belonging to an arbitrary number of position types.

We introduce the weak entity **position** that we use to link faculty members to position types. Every instance of this weak entity type belongs to exactly one faculty member and to exactly one position type. Each faculty member does have at least one position. Usually they have only one position at a time. This is why each position has a starting date and an optional end date. Faculty members may have multiple different positions over time, for example, they may start as lecturer in 2022, then be promoted to associate professor in 2025. For each position type, there may be arbitrarily many corresponding position records.

Interestingly, whether a faculty member can supervise certain types of students is not necessarily only determined by the position type. These abilities are instead attributes of the faculty member.

They may require certain positions, such as full professor for graduate student supervision. But there may also be other factors, such as recent publications, fulfillment of teaching duties, etc.

Finally, we also model that each faculty member belongs to exactly one school. An each school can have an arbitrary number of faculty members.

When we look at the new [Figure 18.15](#), we find that it is truly beautiful. We can also see that many things are still missing. For example, we did not yet model the relationship between students and classes, classes and modules, modules and curricula, classes and rooms, the list of available rooms and their features, exams, exam results, graduation requirements, and so on. Still, we have managed to drag a good piece of reality into our model.

The ability to annotate relationship types with modalities and cardinalities is very important. It makes our models both clearer and more precise. Without it, we cannot express whether a person can study multiple curricula or not. Without it, we cannot express that a person can have multiple IDs of each ID type but that each ID belongs to exactly one ID type. We make big strides toward being able to model real-world scenarios.

18.6 Compact Crow's Foot Notation

From [Figure 18.15](#), we can draw two very general conclusions about the notation we used (entities as rectangles, attributes as ellipses, and relationships as diamonds, combined with the Crow's Foot method for expression cardinalities): First, it indeed allows us to model real-world situations as datastructures and their relationships. Second, it is also a bit verbose. Especially if we have many attributes, the ellipses get many and hard to read. The diamonds to expression relationships are also space consuming.

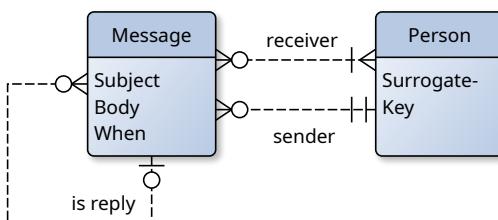
There also is a more compact method to express (almost) the same information: We can combine [UML class diagrams](#) [48, 309, 458, 459] with the Crow's Foot notation as well. This is a very commonly used method in several tools. Let us investigate it here and use it to further explore our example application, the teaching management platform. In [Figure 18.16](#), we sketch the messaging subsystem of our teaching management platform. The messaging system is a feature requested by students and teachers alike. The teachers want a way to provably and traceably inform students about requirements, tasks, problems, and reminders: *"You forgot your homework two times in a row. Do this again and I will have to deduct 10 points from your final score!"* Actually, the students want exactly the same thing, too – to be provably able to inform the teachers of issues and problems: *"I want to do the homework, but the PDF file you offer to download is damaged and cannot be displayed. Please re-upload it."* It is better to have such messages in our system, where they can provably be traced.

In the new compact notation, an entity type is still visualized a rectangle. In the top part of the rectangle, the entity type name is written. In the second part, we write the list of attributes. As you remember, *Person* is a strong entity type in our model. The new entity type *Message* is a strong entity type as well. It has three attributes, *Subject*, i.e., the title of the message, *Body*, i.e., the message text, and *When*, the date and time when the message was sent.

Relationships in this visualization approach are modeled just as lines directly connecting the entity types. No diamonds are used, but instead the relationship names are written as labels directly adjacent to the lines. Cardinalities and modalities are expressed using the Crow's Foot notation.

Each message has exactly one person as sender. Each person can be the sender of arbitrarily many messages. Each message has at least one, but potentially many persons as receiver(s). Each person can receive zero, one, or arbitrarily many messages. Additionally, a message can be the answer to no or exactly one previous message. There can be arbitrarily many answers to each message.

Before, we signified strong and weak entities by using normal or double-lined rectangles. Now



[Figure 18.16](#): The structure of the messaging subsystem of the teaching management platform.

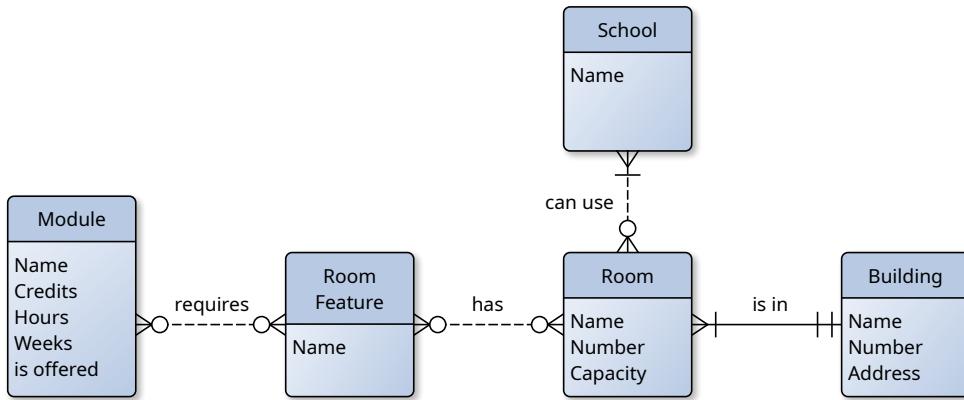


Figure 18.17: The room planning subsystem of the teaching management platform.

relationships can be weak or strong [319].

Definition 18.28: Identifying Relationship

An *identifying* (strong) relationship is connected to at least one weak entity and is required for identifying the weak entity (see also Definition 18.24).

Usually, a strong relationship connects a strong entity to a weak entity. The weak entity cannot exist without the strong entity. The primary key of the strong entity is then part of the key of weak entity. Strong relationships are signified by solid lines.

Definition 18.29: Non-Identifying Relationship

A *non-identifying* (weak) relationship is not needed to identify an entity.

Examples for this are, e.g., relationships that connect two strong entities. But also weak entities can be connected by weak relationships, as long as the connections are not required for identifying purposes.

We now develop the room management subsystem for our teaching management platform in Figure 18.17. Courses take place in rooms at certain times, so we need to know which rooms exist and what features they have. In this model, buildings are strong entities. They may have a name, maybe something like 合肥大学综合实验楼, and a number, let's say 53. We also offer an address string. No real addresses like those discussed before are needed, because we certainly do not need to model countries or provinces here. We assume that all classes of a curriculum take place in the same country, province, and city. However, maybe the university has different campuses, like 南一区 and 南二区, which would be something useful to write there.

Rooms exist within buildings and never without a building, so they are weak entities. They have a number and maybe a name. Each room has a capacity limit for students. Rooms are connected to the building entities via identifying relationships: A room must be in exactly one building, whereas a building should consist of one or many rooms (otherwise, we do not need to store it in our DB).

Rooms are also connected with non-identifying relationships to schools: One school may be permitted to use zero, one, or more rooms for teaching. Each room must be usable by at least one school (because otherwise, we simply don't need to store it in the DB).

We can easily expect that different teaching modules may have different requirements regarding the rooms. For normal teaching, it may be sufficient that an overhead projector is present. We could assume that this is always the case. However, for practical computer science lab classes, we may need one computer per desk. For chemistry experiment classes, we may need a smoke outlet and something for chemical waste disposal.

To model such things, we create the strong entity *Room Feature*, which just needs a descriptive name. Rooms are linked via non-identifying connections to such features: Each room may have zero or one or many such features. Each room feature may be provided by zero, one, or many rooms. At

the same time, a teaching module may require any number of room features. A room feature may be required by any number of teaching modules.

Wait. Suddenly we realize something interesting. We said “*A room must be in exactly one building, whereas a building should consist of one or many rooms.*” This is what we wrote in our **ERD** in [Figure 18.17](#) and this is what makes sense. However, that sentence is interesting, as it poses a somewhat philosophical problem. If we look at it from a technical perspective – which is not really relevant yet at this stage – it causes us to scratch our heads. What we have here is a **Building** \bowtie **Room** pattern. Actually, we have discussed several such relationship patterns before.

We are not yet at the stage to choose a data model and **DBMS**. Yet, regardless of what data model we use, we will be forced to instantiate the data structure for **Room** to, well, represent a room in our **DB**. Following the formal definition of the above rule, we would need to have an existing entity of type **Building** at this point in time. Because we must link each room to one building. So we must first instantiate **Building** and then we can instantiate **Room**. Except that we can’t. If we interpret my ERD strictly, we face the problem that each building must also be connected to at least one room. So this means that we would first need an instance of **Room** before we can instantiate the **Building** entity type.

This philosophical dilemma can be solved in three ways: First, we could say: “*Well, the conceptual model represents the real world. In the real world, each room is inside a building and a building has at least one room. That is true and that is what we model here. Whether or not this can be realized technically is not relevant at this stage of development.*” And this is true. Conceptual modelling should be technology-agnostic.

Second, we could say, maybe even as a corollary of the above, that: “*If not otherwise possible, on a technical level, we may choose to not enforce that buildings must contain rooms. We know that the user will add room records related to building records eventually. We just accept that there may be a temporary state of the DB where one of the mutual dependency constraints is violated. Just for a short time, between the moment when the user has created a building record and right before she adds a room record. It doesn’t matter.*”

The third solution is that we, well, actually implement it as specified. Most DBMSes support **transactions**, i.e., groups of instructions to the DBMS that are executed as one atomic unit and either fail together or succeed together. There is no intermediate state, as transactions are indivisible, atomically executed units. Thus, we could require that every time a new building is added to the DB, the first room is specified as well. The creation of the building and room record could be executed together as atomic transaction. If we do this, the conceptual constraints would perfectly map to logical / technical constraints. We would pay for this with more complex operations, because we now need to use transactions explicitly. **PostgreSQL**, for example, permits us to defer the constraints checking to the end of a transaction [297, 389]. Later, in [Section 19.2.2.7](#), we will learn that using **PostgreSQL**-specific extensions to **SQL**, we can group the necessary instructions together such that they form an implicit transaction.

Anyway, the point is that conceptual relationships that seem hard to realize in software may exist. We will see that all (binary) relationship models that can be represented with crow’s foot notation can be implemented in a relational DBMS. Whether it makes sense to do it – it can become a bit complicated – or not, this should not bother us during the conceptual modelling phase.

Let us re-design the interactions of students, curricula, faculty, and modules. Before making our initial designing of these systems, we discussed with our stakeholders in the university. We learned that, for example, some abilities of teachers are bound to their position, e.g., which kind of modules they can chair. Other abilities should be bound to their person, e.g., whether they can be Master’s supervisor. We did model this back in [Figure 18.15](#). There also are two different ways teachers and students can interact: Students can enroll in classes of professors and/or a professor can be their BSc or MSc supervisor. Basically, we would have two sets of interactions governed by two different forms credentials on the teacher’s side. Such situations are always bad. They make our models bigger. They have a smell of redundancy. They almost beg us to simplify them.

We have the idea to unify them both in [Figure 18.18](#). It is clear that not all teachers can chair all types of modules. Maybe our university will only permit full professors to chair core modules of a curriculum, while younger lecturers can propose and chair elective (voluntary) courses. This can easily be covered by relating position types to module types. Then again, some modules may require special certifications, such as chemistry safety certification or something. This does not fit well to the position type-module type approach, because “chemistry safety certified” is not a position. The same holds for “Master’s Supervisor”.

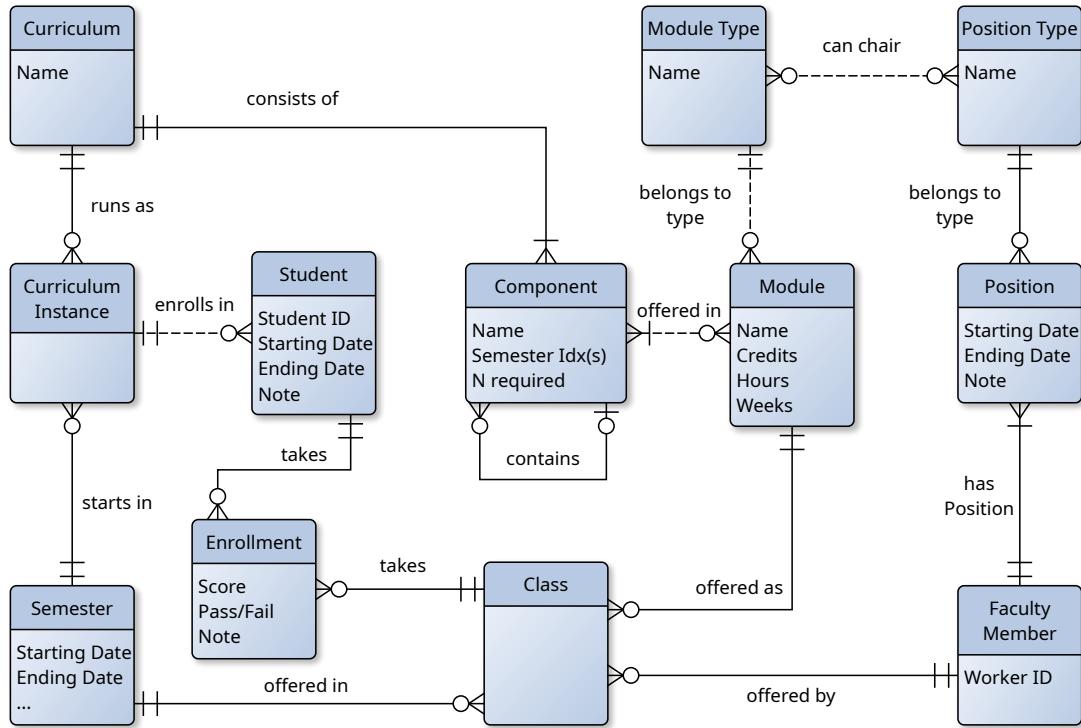


Figure 18.18: A re-design of the student/curriculum/faculty/module interactions in our teaching management platform.

Well, not necessarily: How about we permit that a person can hold multiple positions at a time. We already modelled the traditional career-based positions. Additionally, we could include things like “Chemistry Safety Certified,” “Master’s Supervisor,” and whatever else we need as position types. That positions are limited by starting and ending dates is also not a problem. This is exactly a feature that we would like to have. Interestingly, with our *Position* and *Position Type* entity types used like this, we could easily represent even more complex functions, such as Dean, Vice-Dean for Teaching, Department Head, Team Head, and so on. These functions could then be tied to what kind of changes the person can make to the data.

But back to the relationship between teaching and position. Each *Module Type* entity can now be linked to one or multiple *Position Type* entities that a teacher must hold to be permitted to chair them. Each position type, of course, can be the credential for chairing modules of multiple different module types. This approach works because we realize that Master’s and Bachelor’s projects are nothing but modules belonging special module types. They would automatically be covered by this permission system.

A school in our university may offer different curricula, e.g., a Master of Computer Science or a Bachelor of Engineering in Computer Science. Initially, we assumed that we could say that a curriculum consists of different modules and each module takes place in a certain semester of a curriculum. However, there are two different problems: First, sometimes we have elective modules, i.e., situations where a student needs to pick maybe two out of a set of three or four possible modules. Second, for a curriculum, there may be different specialization directions. A Master in Computer Science could offer the specializations **AI**, **DBMSes**, and **Computer Security**, for example. Each such specialization may (recursively) come with different compulsive and elective modules.

We will try to model this by introducing the *Component* entity type. A curriculum consists of at least one component and each component belongs to exactly one curriculum. A component can also be a sub-component of at most one other component. A component may also contain an arbitrary number of sub-components. Each component has a name and a set of semester indices in which it is offered (such as “7th and 8th semester”). A module can now be offered in one or multiple such components. Each component may offer zero or multiple modules. A component can also contain a number defining how many of the offered modules a student *must* complete. With this, we could now state that:

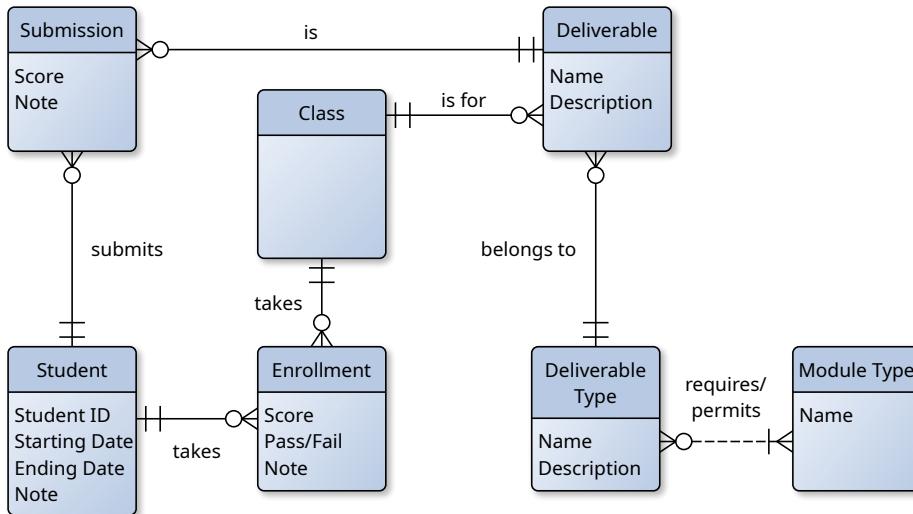


Figure 18.19: The handling of deliverables in our teaching management platform.

"Among the many components of the Master of Computer Science curriculum, there is the 'Specialization' component. It, in turn, offers the components 'AI', 'DBMS', and 'Computer Security.' A student must select exactly one such component. The 'DBMS' specialization then offers the compulsive module 'Databases'. It also offers the component 'Electives'. Both, the 'Databases' and 'Electives' component must be completed by students. The 'Electives' component offers the modules 'PostgreSQL,' 'Systems Security,' and 'DB Design,' two of which must be completed by the students."

This does not look very pretty, but at least it allows us to model even complicated situations within our DB. Either way, this brings us to the *Module* entity type. Modules must be offered by at least one component (otherwise they are useless). They also belong to a module type, which links them back to the position-based credential system for teachers discussed earlier. Each module has a name, a number of credits, and a number of teaching hours. Some modules, like internships or external practical training classes, may have weeks as duration. They also have a syllabus and abstract and other information, which we have omitted here.

Modules are basically the blueprint of the learning content that is offered. They are linked to certain semesters counted from the start of the curriculum.

A school can instantiate a given curriculum and create an entity of type *Curriculum Instance*. Such an instance is always linked to a starting semester, such as the Winter Semester 2025, or maybe the Summer Semester 2026. We retain the *Semester* entity type for this purpose, which contains all the necessary dates and meta-information for a semester date period as defined by the university. At the start of each semester, a faculty member can offer no, one, or multiple classes. A class is always an instantiation of one module. A module may be offered multiple times as class, maybe even within the same semester: Remember that we also treat Master's and Bachelor's projects as modules. (They start in one semester, but nobody said they need to end in the same semester.) Our system could easily check whether a teacher has the credentials required to offer a certain class. Of course, the administrative person of a school needs to enter the offered classes.

A student can now enroll into a class. Both classes and enrollments are weak entities. Each enrollment is linked to exactly one student, but a student may enroll into arbitrarily many classes. The enrollment record will later also contain information such as the score of the student, whether they passed a class or not, and maybe explanatory notes.

Every student is also enrolled into exactly one curriculum. Arbitrarily many students can enroll into one curriculum. Well, maybe there will be limits for this, but we assume that the administrative person of the school enters the students into the curricula and they will know what they do.

Actually, our platform could automatically enroll students into classes that they have to take. It sees which curriculum they belong to, knows which modules are required, and can auto-enroll them whenever there are no alternative choices. Based on the classes offered and the curriculum structure, it can also offer them choices where they exist via a web portal.

Another part for our platform will focus on deliverables in Figure 18.19. There are different types of deliverables, e.g., written exams, oral exams, midterm exams, homework, internship reports, or theses. Each module type may *require* any number of different deliverables types. Each module type may also *permit* any number of different deliverables types. (We modeled these two relationships as one to save space.)

We already have established that classes belong to modules belong to module types. So from the relationship between module types and deliverable types, we can infer what deliverable types are permitted and/or required for every class. For a Master's Project, a Master's Thesis may be a required deliverable. For a course "Databases", a written final exam may be required and a mid-term exam as well as homeworks may be permitted. The teacher then will create a (weak) *Deliverable* entity, which has a Name and a Description. Each student will make corresponding submission for the deliverable. These submissions will not go to our platform.

Instead they are sent by the student to the teacher. The teacher evaluates them and creates the corresponding (weak) *Submission* entities – one per student – in the system. This weak entity holds the result of the student and maybe a comment by the teacher. This way, the teacher can upload exam results, mid-term exam results, and so on. The students then can view them in the online system. Notice that we here just wrote *Score* as attribute, but we may as well imagine that other attributes like pass/fail would make sense here.

All of the elements of this conceptual draft of our teaching management DB are included in Figure 18.20. This model is still not very advanced. But it has already several good features. We can imagine that it will be at least somewhat usable.

We did not explicitly state this before, our model for the teaching platform partially follows the principle of an *Insert-Only Database* [334]:

Best Practice 16

In many application scenarios where historical information needs to be preserved, data in a DB should never be changed or deleted. Changes in the real world should instead be reflected by adding data to the DB.

Viewing this from the SQL perspective means that in this scenario, the operations `UPDATE` and `DELETE` should not be used. Instead, only new records should be added via `INSERT INTO`. And in our situation, we do want to preserve the history of the data.

For example, a curriculum consists of modules. Now we could have modeled that each module is assigned to a teacher. Once the teacher changes their workstation and another teacher takes over, we could just change the assignment. This would be much simpler than our current conceptual model. It would have a big drawback, though: It would be impossible to tell which teacher taught a module in the past.

Therefore, we chose a different path: Instead of assigning modules directly to teachers, we create instantiations of them (the class entities), which bind to semesters and teachers. This assignments never changes. Each semester, new class records are added. We always know who taught which course in which semester. Our DB will grow incrementally as changes are reflected by new records and not the modification of existing records. If a module is taken over by another professor, then this will yield a new class instance. We can see who taught which class (module) in which semester.

For this reason, the *Address* entity type has the *Valid From* and *Valid To* attributes. If the address of a student or faculty member changes, we create a new address record, set *Valid From* to today, while leaving the *Valid To* attribute at `NULL`. We then enter yesterday as *Valid To* value of the old address as no longer in use. The same also holds for our *Personal ID* and *Person Name* entity types. The administrators can therefore immediately see which address, ID, or name a person had in use when some action was taken in the past.

The conceptual model of our system also has some shortcomings. For example, there may be some more organisational levels in a university than just schools. A school may be subdivided into different departments and departments into teams.

Furthermore, we should be able to add timestamps and notes to most of the data items to be able to track when they have been created. Such a technique called *timestamping* – which we did not really model here. Also, we did not model rights management. We would probably need a table defining

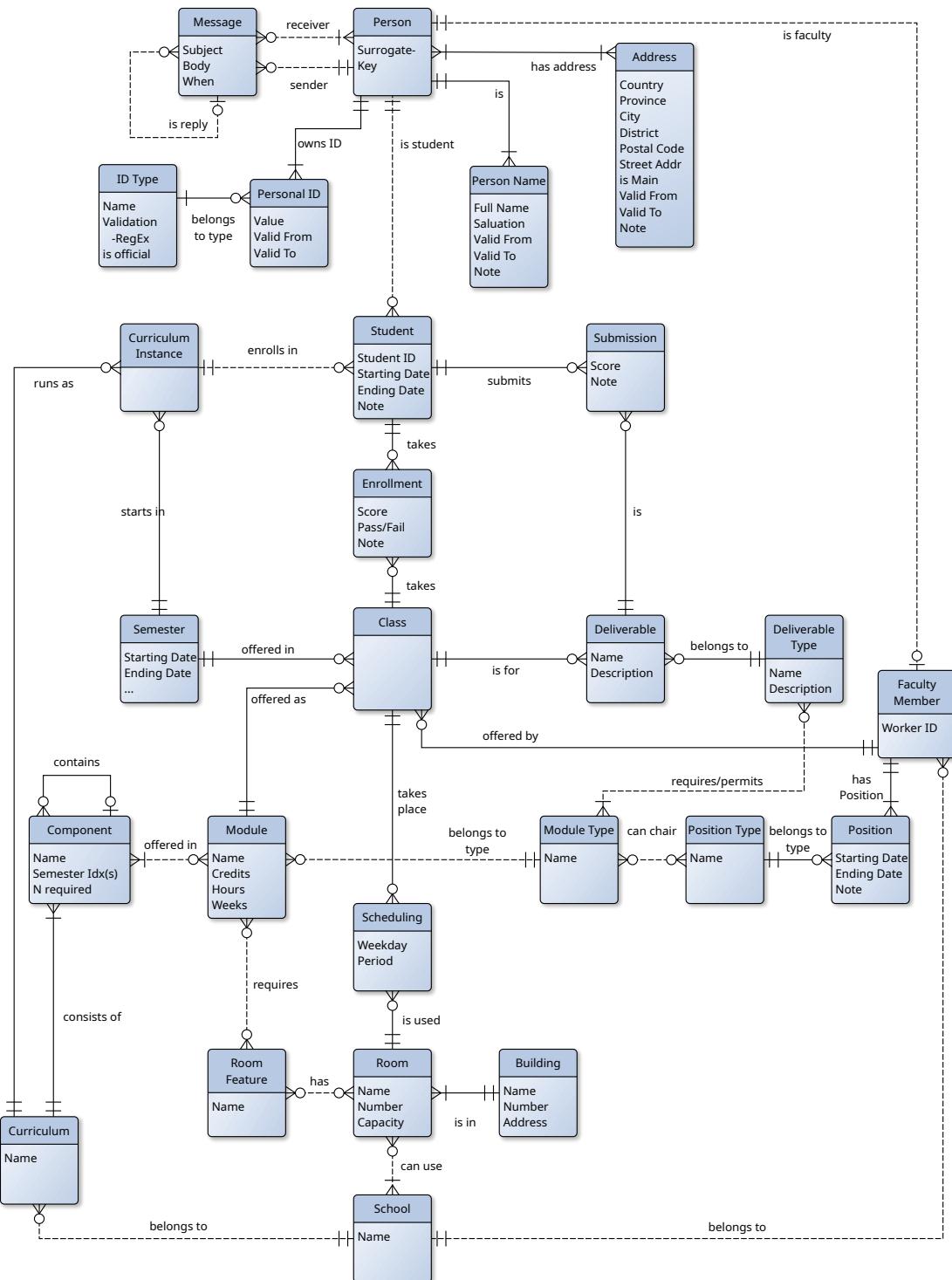


Figure 18.20: A complete overview of the conceptual model of our teaching management platform.

which person is permitted to enroll students into a class or curriculum, which person in a school can create new modules, classes, curricula, and so on. Or we could also handle this via the position entities that we already designed. This would probably a good idea.

Additionally, we may want to create a table creating an audit trail [252]. Such a table would store which user added/changed which record in which table at which point in time.

Finally, in our *Address* entity, countries and provinces are simple attributes. We could probably have another entity type to represent countries and country subdivisions (such as provinces) based on the ISO 3166 standard [105, 230], maybe with additional information such as the phone number prefix. Each address could be linked to at least one such entity via a relationship.

For now, we omit all of this. This is just a book, not a real system design. We have to draw the line somewhere to make this chapter not overly long. The system looks more or less reasonable, so our goal is to move on to create a logical model and then a prototype.

The important point is that we now can really design almost arbitrarily complex systems on the conceptual level. We know how to model entity type, relationship types, cardinalities, modalities, etc. We can do this in a compact notation such that we can easily fit ten entity types into one **ERD** while not sacrificing readability. In terms of conceptual modelling, we are good.

18.7 Data Model Selection

Between the design of the conceptual schema and the design of the logical schema, we have to choose a *data model* for our DB [380].

Definition 18.30: Data Model

The *data model* specifies the notation and language for defining datatypes and for accessing and updating the DB.

There are several important data models.

We could treat our data as single or weakly-interrelated tables. These could be stored in formats such as **CSV** or in the file formats offered by **Microsoft Excel** or **LibreOffice Calc**. Such formats can deal with thousands of similarly-structured records and perform efficient calculations on them. They are not suitable for data where records of different types are related and constraints are placed upon the relationships or the data. They are also not suitable for scenarios where multiple users concurrently work on the same documents. For our teaching management system, they are unsuitable.

Another choice would be hierarchically structured single documents, in formats such as **XML**, **JSON**, and **YAML**. These formats are able to deal with thousands of records that are structured similarly or differently. They are not suitable for data where records of different types are related and constraints are placed upon such relationships. They are also not suitable for scenarios where multiple users concurrently work on the same documents. Still, some **open source software (OSS)** such as **BaseX** [189, 190] exists that brings more of a DB flavor into this area.

The early developments in the DB field saw hierarchical DBs, such as **IMS** [33, 246, 254]. These offer the advantages of hierarchically structured documents combined with the ability to support concurrent access. The hierarchical structure of data may create redundancy: If a professor teaches two classes, then the data about the professor is stored twice. This approach to DBs is basically outdated. Still, some open source hierarchical **DBMSes** still exist, including **MUMPS** [304, 305]. The hierarchical key-value store **YottaDB** is also an implementation of **MUMPS** [39].

Another legacy approach is the network data model, which emerged from **IDS** [14, 15, 193] and **CODASYL** [432]. There still exist CODASYL- and COBOL-based DBMSes today [310]. They have the reputation of not being easy to use. However, as is the case for hierarchical DBs, these systems can be considered as in decline. I would avoid them.

Another data model are key-value stores. They are basically schema-less and appropriate if your data has a key-value structure. For our teaching management platform, they do not appear suitable.

The main focus of this book, however, is on the relational data model invented by Codd [90] and the **SQL** language. This model fits well to many application domains. If you can express your data in one of the previous models, you probably can do this with a relational model, too. Compared to the

other data models, there just exist infinitely more and well-maintained open source and commercial relational DBMSes.

Here, entity sets are represented as relation and stored in a table inside the DB. As we already practically explored, the tables can also be related and constraints can ensure that the relationships between them remain in a consistent state. We find that most of our entity types will probably nicely map two-dimensional tabular structure. Most of our attributes are simple. The relationships in the diagrams can probably be represented directly as foreign keys or in additional tables that relate foreign keys to each other other. Therefore, the relational model is what we decide to use for our teaching management platform. In [Section 19.1](#), we will introduce this model in more detail.

18.8 Summary

At this point, we finish our excursion into the exciting world of conceptual modeling. This important stage of the DB development lifecycle is somewhat the transition from requirements gathering to system implementation. Here, we try to convert the functional requirements into semi-formal system models. We can still discuss and verify them with stakeholders.

Conceptual models are supposed to be abstract. They are not focused on any specific DB technology and do not even have to comply with the relational data model. Instead, we try to represent the part of the world that is relevant for the system that we want to construct as clearly as possible. This gives us more freedom. We can model things in the most natural way and do not (yet) need to worry about how to actually implement the model.

At this stage, we can also hash out some basic principles for our system.

The conceptual model for our example, the teaching management platform, is still relatively simple. Yet, it already involves relationships between data over several levels indirection. It already involves lots of different entity types. Conceptual models of non-trivial systems have a certain complexity.

And this is also why we need them: Of course, we could have started designing the DB with [SQL](#) right away. That could even succeed.

However, this would have short-circuited our creative process. When you read this part of the book from beginning to end in one go, you may notice that we changed the way we modeled certain things. We could do that because we could look at simple visual representations of our DB. We even (imagined that we) discussed them with our (imaginary) stakeholders in our (imaginary) university and incorporated (imaginary) feedback. If you have downloaded [yEd](#) and maybe even loaded the [ERDs](#) that we designed with it, you may have found that working with this tool to design a model is even ... kind of fun. At least I found that. And I did not expect that it would be fun. Either way, it is clear that with relatively little effort, we can design a model or a part of a model as ERD. When we find the something is wrong with the idea, or maybe we get an idea to achieve the same results with a simpler approach, then we can conveniently change the model. Try doing that with a model based on SQL commands.

Chapter 19

Logical Model Design

The next phase of DB design is the transformation of the entity-relationship model into a logical schema obeying a certain data model [383]. We chose the relational data model in [Section 18.7](#).

19.1 The Relational Data Model

In the ERDs that we painted [before Section 18.6](#), there were three visual components: entity types (rectangles), attributes (ellipses), and relationship types (diamonds). When we moved to the more compact visualization style in [Section 18.6](#), the relationship diamonds disappeared. Instead, they were represented just by straight lines. This has two reasons: First, the relationship diamonds waste space. Second, in the relational data model, relationships do not exist as independent objects. In this model, we only have entity types (embodied by tables) and attributes (the columns of the tables). Relationships are realized as foreign keys, i.e., as special attributes, and as constraints.

19.1.1 Definitions

In the context of [relational databases](#), the same definitions for attributes and domains ([Definitions 18.2](#) and [18.3](#)) that we already discussed back in [Section 18.1](#) (Entities and Attributes) are used. The following additional definitions are commonly considered [90]:

Definition 19.1: Relation Schema

A *relation schema* Σ is the ordered sequences of n attributes (a_1, a_2, \dots, a_n) , i.e., is a sequence of attribute names and domains.

Definition 19.2: Relation

A *relation* R is a set of n -tuples $R \subseteq \text{dom}(a_1) \times \text{dom}(a_2) \times \dots \times \text{dom}(a_n)$ to which a relation schema $\Sigma(R)$ that specifies the attributes (a_1, a_2, \dots, a_n) is associated [382].

The definition of relation schemas in the relational model is therefore somewhat equivalent to the definition of entity types in the entity model (see [Definition 18.4](#)). When translating our conceptual model to a logical relational model, an entity type will become a relation schema. The difference to the conceptual is that, in the logical schema, we will use relations to implement both entities and relationships.

Also, at first glance, one may think that “Relations = Tables” in a DB. In other words, one may think that relations are implemented as tables. But this is only partially true: Relations can also be the result from a [SELECT](#) statement in [SQL](#). Relations can also be the parameter of an [INSERT INTO](#) statement. Thus, relations are a quite versatile concept to represent our data.

Notice that a relation is a *set* of tuples. Since a set cannot contain the same element twice, this means that duplicate tuples (rows, records) are not permitted in relations by definition [72]. As a deviation from the pure formalism, the SQL language does permit duplicate tuples in tables and query results [72]. Sets are also not ordered, so there is no default order of the tuples in relations either.

All attributes (columns) must have names, i.e., there are no anonymous attributes [395]. In the original works on relational databases [90], the order of the attributes (columns) in a relation mattered and it was permitted that two column have the same name. This idea was later abandoned. Today, the order of columns are unimportant and the columns of a table must have unique names [395]. The values of attributes are atomic, i.e., there are no multivalued attributes and no composite attributes [383, 395].

The degree of a relation is defined as follows (please to not mix this up with the degree of a *relationship* discussed in [Definition 18.19](#)):

Definition 19.3: Degree of a Relation

The *degree* of a relation is the number n of its attributes.

Relations are at the core of [relational databases](#).

The totality of data in a data bank may be viewed as a collection of time-varying relations. These relations are of assorted degrees. As time progresses, each n -ary relation may be subject to insertion of additional n -tuples, deletion of existing ones, and alteration of components of any of its existing n -tuples.

— Edgar Frank “Ted” Codd [90], 1970

19.1.2 Keys

In [Section 18.2](#) (Keys), we discussed the topic of *keys* in conceptual modelling. Just now, we stated that relations are sets of unique records. Keys are what make these records unique. Keys therefore play a very important role in relational DB design. It thus makes sense to revisit this topic here again.

Back in [Section 18.2](#), we learned that a (*candidate*) key is a minimal super key, i.e., a minimal set of attributes that can identify an entity. With the definitions given for the relational data model that we just discussed, [Definition 18.13](#) ((Candidate) Key) can also be expressed as follows [383]:

Definition 19.4: Key

A set of attributes $K \subseteq \Sigma(R)$ given as $K = \{k_i : i \in 1..m \wedge k_i \in \Sigma(R)\}$ of a relation R is a key if and only if

1. K is identifying, i.e., if the values $v_{1,i}$ and $v_{2,i}$ for all $i \in 1..m$ are the values of the attributes k_i for two rows r_1 and r_2 in R and $r_1 \neq r_2$, then there is at least one $j \in 1..m$ with $v_{1,j} \neq v_{2,j}$ and
2. there is no subset of $\{k_1, \dots, k_m\}$ with this identifying property, i.e., the key is minimal.

For a super key S , as introduced in [Definition 18.12](#) (Super Key), it then simply holds that $K \subseteq S \subseteq \Sigma(R)$. Each relation must have at least one key, because the records in a relation are unique. One of the keys is chosen as primary key (see [Definition 18.14](#)). Often, however, we instead use a surrogate key, i.e., an identifier automatically generated by the DBMS (see [Definition 18.16](#)), as primary key.

We already learned that the records of one relation can reference records in another relation via so-called *foreign keys*. Let us now formalize this concept [383]:

Definition 19.5: Foreign Key

A set of attributes F in the schema $\Sigma(R_1)$ of a relation R_1 is called a *foreign key* if

1. the attributes of F have the same domains as the attributes of primary key P of a different relation R_2 and
2. every value of F in a tuple $r_1 \in R_1$ either occurs as a value of P for some tuple $r_2 \in R_2$ or is [NULL](#).

With this, we have lifted the definitions of (candidate) keys, super keys, primary keys, and foreign keys from the conceptual level to the logical level under assumption of the relational data model.

19.1.3 Relational Database Management Systems

In the 1980s, many vendors of DBMS did not completely implement the relational data model as developed by Codd. Instead, they added mechanics that circumvent the relational characteristics, either because of laziness, to allow backwards compatibility to older systems, or in order to improve performance. In a response, Codd defined thirteen rules that govern a relational DBMS [91, 93, 395, 419, 420]. Since the first rule is called *Rule 0*, the thirteen rules are referred to as *the twelve rules*.

0. Foundation Rule: A relational DBMS must be able to manage DBs entirely through its relational capabilities.

1. Information Rule: All information in a relational database is represented explicitly at the logical level and in exactly one way – by values in tables. This includes even table names, column names, and column types, which, too, must be stored in a table. Such special tables that store the structure of a DB are usually part of the built-in system catalog. This system catalog holds the metadata of the system and is (or is part of) a relational DB itself.

Remember back when we first began working with PostgreSQL in our very first, very simple example? In Chapter 8, we started by creating a new user for the DBMSes called `boss`. We checked the list of existing users by writing `SELECT username FROM pg_catalog.pg_user`. `pg_catalog.pg_user` is the name of a table, which belongs to the system catalog. When we connected to our DB using LibreOffice Base, we saw lots of strange tables in Figure 13.1.12. These belong to the system catalog.

2. Guaranteed Access Rule: Each and every datum (atomic value) in a relational DB is guaranteed to be accessible via a combination of table name, primary key value, and column name.
3. Systematic Treatment of Missing Values Rule: `NULL` values (distinct from the empty character string or a string of blank characters, and distinct from zero or any other number) are supported in fully relational DBMS for representing missing information and inapplicable information in a systematic way (independent of data type).

This rule has been a point of arguments over many years [72]: Real data does include unspecified elements. There may be street addresses without house number, there may be people without phone number. So there is a need to represent such situations. However, having unspecified or missing values also violates the definition of tuples in relations. We could imagine that the domain of each attributes contains the additional value `NULL`, too, though.

4. Dynamic Online Catalog based on the Relational Model: The DB description is represented at the logical level just like ordinary data, so that authorized users can apply the same relational language to its interrogation as they apply to the regular data.
5. Comprehensive Data Sublanguage Rule: A relational system must support at least one language whose statements are expressible per some well-defined syntax as character strings; and which supports all of the following items:
 - a) data definition (e.g., creating tables),
 - b) view definition (creating views),
 - c) data manipulation (e.g., adding and deleting of data),
 - d) integrity constraints (e.g., limits on data range, foreign keys, . . .),
 - e) authorization (e.g., user management), and
 - f) transaction boundaries (begin, commit, and rollback).

In the case of our book, this language is SQL. Of course, one could conceive and support also other languages.

6. View Updating Rule: All views that are theoretically updatable are also updatable by the system.
7. Insert, Update, and Delete Rule: The capability of handling a base relation or a derived relation as a single operand applies not only to the retrieval of data but also to the insertion, update, and deletion of data. In other words, the operations do not just apply to single records (rows) but can concern multiple rows at once, because their inputs can be relations (whole tables, results of `SELECT`, . . .).

8. Physical Data Independence Rule: Application programs and terminal activities remain logically unimpaired whenever any changes are made in either storage representations or access methods. In other words, the way the DBMS actually stores the data has no impact on how an application accesses data via the text-based language.
9. Logical Data Independence Rule: Application programs and terminal activities remain logically unimpaired when information-preserving changes of any kind that theoretically permit unimpairment are made to the base tables. Let's say we split a table into two tables and distribute the rows into either part, leaving columns and primary keys intact. We can then design a view that merges the two tables (using `UNION`). An application sitting on top of that will not see any change.
10. Integrity Independence Rule: Integrity constraints specific to a particular relational database must be definable in the relational data sublanguage and storable in the catalog (not in the application programs). We did this several times, for example with the `PRIMARY KEY` constraint, the `REFERENCES` constraint, and the `CHECK` constraint as far back as in Chapter 9 (Creating Tables and Filling them with Data). All of them were defined in SQL.
11. Distribution Independence Rule: A relational DBMS has distribution independence. This means that a DBMS *may support* storing data distributed over several different computers (nodes) or a cluster. If the DBMS supports such distribution, then this should not affect the SQL programs sent to it. In this case, the DBMS must take care of dividing the queries to the corresponding nodes and re-assemble the results. If a DBMS does not support distribution of data over a network, then it automatically fulfills this rule.
12. Non-Subversion Rule: If a relational system has an additional low level (single-record-at-a-time) language, that low level cannot be used to subvert or bypass the integrity rules and constraints expressed in the higher level relational language (multiple-records-at-a-time). In other words, a DBMS must support at least one relational language (Rule 5), but it may support any other access languages or programming interfaces as well, some of which may work on single records, some may not be relational and so on. However, none of these access methods must be allowed to violate the integrity of the relational data.

These rules are implemented to a large degree by modern DBMSes. They also tell us relatively exactly what to expect, with what kind of features we can work.

We now have a basic understanding about how the relational data model works. We know the basic definitions. Even better, we can tie the definitions that we learned for conceptual modelling into the definitions for the relational model. We find that they are quite similar. We also learned the basic requirements for a relational DBMS. Combining these information with what we already learned in the initial example of this book, we have a pretty clear understanding of what relational DBMSes like PostgreSQL offer to us. And we may have some good ideas about how we can transform our conceptual models to relational ones tied to an SQL DB. Which, coincidentally, is what we will do next.

19.2 Mapping Conceptual Models to Logical Models

We now want to implement the conceptual model of our teaching management platform on a relational DBMS. This requires us to map entities and relationships to tables and constraints to objects in the relational realm. We further will need to design views, queries, as well as insertion rules for our data. Naturally, we choose PostgreSQL as the DBMS. PostgreSQL supports SQL, so most of the functionality we will use can be provided 1:1 by other systems, such as MySQL, MariaDB, or SQLite.

The question of how to translate the conceptual model to a logical model is interesting. There are several sources that say that entity relationship models can easily be converted to logical schemas based on the relational data model and that there are tools available that can automate this [383]. This, I believe, depends on how abstract your entity relationship models are. And it depends on how exact the translation should be.

There are different tools that we could use to create our ERDs. We used yEd, which is total independent from any underlying DB technology. It does not even have anything to do with the relational data model. Translating such models to logical model does require thinking, although often it is not that hard.

We could have used [PgModeler](#) to draw our ERDs as well. The [PgModeler](#) can output SQL or even directly connect to the [PostgreSQL](#) DBMS. Then, however, we would not have created an abstract conceptual model. We would have directly started with something that is more or less already a logical model.

But we do have an abstract conceptual model. And now we will learn how to translate such models to logical models.

19.2.1 Mapping Conceptual Entity Types to Logical Models

Translating entity types from the conceptual model to the logical model is fairly simple. Each entity type in the conceptual model becomes one table in the logical model. The entity type *Student* becomes the table `student`. Each simple single-valued attribute becomes one column of that table. The attribute *Name* becomes the column `name` with a specific [SQL](#) datatype for text and maybe an added sanity constraint, e.g., names should begin and end with printable (non-whitespace) characters.

Each component of a composite attribute becomes one column of that table. Assume that *Name* is not a simple attribute but a composite attribute. Let's say that it consists of the two components *Full Name* and *Salutation*. Then we will have two columns, one called `full_name` and one called `saluation`. Both would have reasonable datatypes and attached sanity constraints. In this case, the composite attribute *Name* in the conceptual model does not itself have a column in the table. Instead, its components have columns. Of course, if the components themselves are composite attributes, the process is repeated recursively. Then, the components are broken down until we arrive at simple attributes, for which we then have columns.

Multivalued attributes become separate tables. Each of their rows references the one row of the table of the entity type (by storing its primary key via a foreign key). For example, if the *Student* entity type in the conceptual model has the multivalued attribute *Mobile Phone*, this would mean that each entity of type *Student* can have multiple values of *Mobile Phone* associated with it. In the relational data model, all datatypes are atomic, i.e., we cannot have a column that is of type "list of something". Each attribute can only have a single value in each record. Thus, multivalued attributes need to become tables by themselves. So we would need to create a table `mobile` just for mobile phone numbers. This table would, at least, need a column for the actual phone number and a column that references the corresponding *Student* record via a foreign key. It may also need to have a surrogate key, but this we discuss later on.

Derived attributes normally are not included in the table.

OK, so our goal here would be to either transform the conceptual model of a [DB](#) to a logical mode or, in smaller-scale projects, to directly design the logical model. But lets first circle back to what a logical model is. In [Definition 15.2](#), we basically stated that the logical model is the collective view that users and applications have on the DB. In the relational model, this means that it defines all the tables, their attributes and constraints, as well as the queries.

In our small initial example in [Part II](#), we only worked with the logical model. We did not create a conceptual model and neither did we bother with a physical model. We just directly went for the action, we fired out SQL scripts to the [PostgreSQL server](#). Indeed, if we have chosen an [relational database](#) as DB type for our project, then the logical model can be specified in SQL – and that is what we did back in that example.

This time, we do have a conceptual model. We want to follow the DB design process properly, based on a software engineering perspective. Back in [Chapter 18](#), we designed our conceptual models based on a very loose syntax using the graphical editor [yEd](#). This editor is entirely unrelated to any [DBMS](#). If we wanted, we could have painted diagrams that make no sense at all. And this freedom is useful when designing conceptual models. We can quickly change entity types, attributes, and relationships. We do not need to worry about technical aspects. We can discuss our model with stakeholders who don't know anything about SQL.

The logical model, however, is bound to a technology. At this level, using a tool like [yEd](#) makes little sense. Instead, there are also tools that are tied closely to SQL or even to specific DBMSes. [MySQL Workbench](#) [289], for example, can connect to the [MySQL](#) DBMS and allows us to craft tables using an ERD-like syntax. [PgModeler](#) [8] allows us to do the same for the [PostgreSQL](#) DBMS. The idea here is that we can use a much clearer and more restricted syntax to draw a visual representation of our DB. This syntax can then be translated to SQL, which we can send to the [PostgreSQL](#) server, e.g., via the [psql client](#). Using such a [GUI](#) has two main advantages: First, diagrams are intuitive and

faster to understand than SQL scripts. Second, the different forms and dialogs that we use to create the ERDs help guiding us to create syntactically correct SQL.

Thus, we first install the PgModeler as discussed in [Chapter 7](#). And now, we will try to translate a simple conceptual ERD – with only a single entity type – to a logical model. We use the very first ERD we drew: the *Student* entity type from [Figure 18.1.21](#). Back when we just began discussing conceptual models, we tried to model the entity type *Student*. We drew an ERD for students as [Figure 18.1.21](#), which I here reproduce as [Figure 19.1.1](#). Here, each entity of type *Student* will have a name, an ID, a student-ID, an address, a mobile phone number, and a Date of Birth (DOB). Later on, we realized that this model has many shortcomings and is not suitable for our teaching management platform. Yet, it is fairly simple and suitable as an example for translating a single entity type from the conceptual model to the logical model.

We want to use the PgModeler for doing so. Under Ubuntu Linux, we can start this program by opening a terminal by hitting **Ctrl**+**Alt**+**T**, typing in `pgmodler`, and hitting **Enter**, as shown in [Figure 19.1.2](#). Under Microsoft Windows, you would instead proceed as shown in [Figure 7.2.18](#).

In the opened PgModeler window, we click on **New Model** in [Figure 19.1.3](#). An empty ERD opens that represents an (empty) DB. In a first step, we should choose a proper name for our DB. We right-click at some place in the empty ERD. In the context menu that opens up, we click on **Properties**, as shown in [Figure 19.1.4](#). A dialog called “Database Properties” opens. As said, we want to set a proper name for our new DB. We choose `student_database` – because the DB will only have a single table named `student` – and then click on **Apply** in [Figure 19.1.5](#).

Back in the ERD view it is now time for creating the table that will represent our *Student* entity type. We therefore again right-click into the (empty) diagram. In the popup-menu, we click on **New > Schema Object > Table**, as illustrated in [Figure 19.1.6](#).

The “Table properties” dialog opens in [Figure 19.1.7](#). We can choose a table name and, as said, we pick `student` and type this in. The attributes of an entity type become attributes in a relation in a relational logical schema, which are embodied as columns of a table. To add such columns, we click on the register **Columns**. The columns register is still empty. We click on the **Add Item** symbol  in [Figure 19.1.8](#).

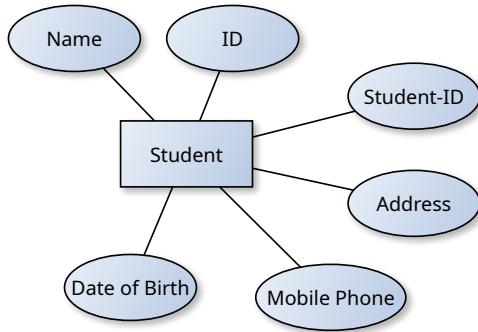
In a first step, we want to create a column for the university-issued student ID. As name for this column, we choose `student_id`. This is better than using `ID`, because it conveys a clear meaning that this is, in fact, the student ID. Everybody will immediately understand what this means. Student IDs are usually strings of a fixed length. We therefore choose the **Type** `character` in [Figure 19.1.9](#). In SQL, this is the datatype for fixed-length strings. As (fixed) length, we enter 11 in the **L:** field. This means that all student IDs that we store in our DB will be text strings consisting of eleven characters. We also mark the column as `NOT NULL`. This means that there cannot be a student record where the `student_id` is `NULL`. This, in turn, means that there cannot be a student record without student ID. `student_id` is a mandatory field that always needs to be provided. In [Figure 19.1.10](#), we click **Apply**.

Best Practice 17

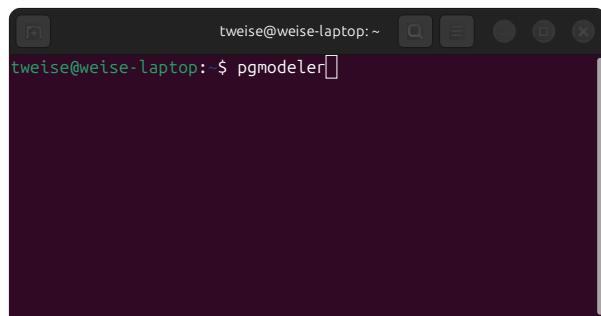
To avoid issues with quotations, it is best to use only lower case character names and underscores (`_`) to separate words for all named things in PgModeler, including tables, columns, and constraints.

The new column appears in the table creation dialog. We now want to add the next column, so we click again on **Add Item**  in [Figure 19.1.11](#). The next important piece of data of each student record is a national Chinese ID number (中国公民身份号码). We add the column `national_id` for storing Chinese ID numbers. As per standard GB11643-1999 公民身份号码 (*Citizen Identification Number*) [507], such numbers always consist of 18 characters. So we choose the datatype `character` with the fixed length 18. We here ignore the fact that there could be foreign exchange students (留学生) and demand that all records must have `national_id` field set by marking the column as `NOT NULL`. We click **Apply** in [Figure 19.1.12](#).

The new column appears in [Figure 19.1.13](#) and we click **Add Item** . We now define the column `name` for student names. Names are text strings of variable length, which corresponds to the SQL datatype `varchar`. We set the *maximum* length to 255 characters, which is fairly large and should be long enough for most sensible names. Each student must have a name, so we again specify `NOT NULL` and click **Apply** in [Figure 19.1.14](#).



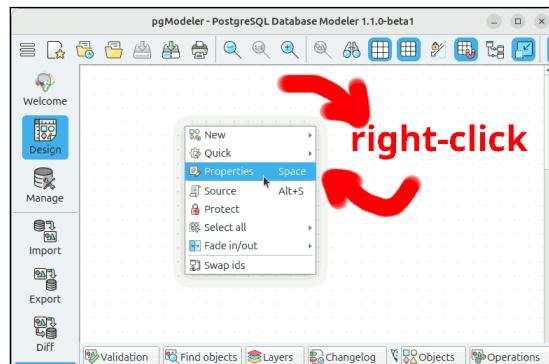
(19.1.1) A reproduction of the Student ERD from Figure 18.1.21. We want to translate this ERD into a logical model for a DB that only contains this single table.



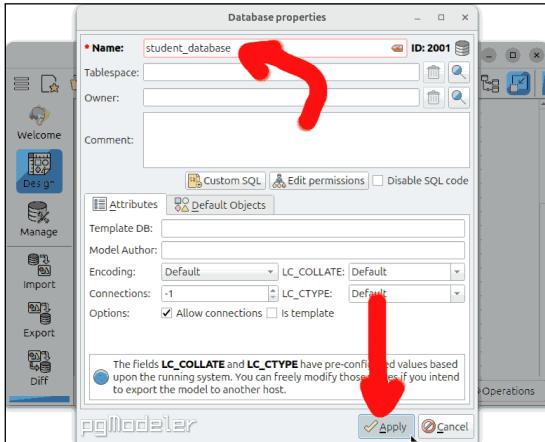
(19.1.2) To start the PgModeler, under Ubuntu Linux, we open a terminal by hitting **Ctrl**+**Alt**+**T**. We type in `pgmodeler` and hit **Enter**. Under Microsoft Windows, you would instead proceed as shown in Figure 7.2.18.



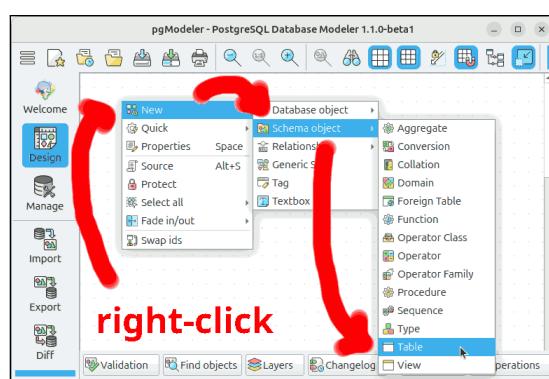
(19.1.3) In the PgModeler, we click on **New Model**.



(19.1.4) An empty ERD opens. We right-click somewhere in it. In the context menu that opens, we click on **Properties**.

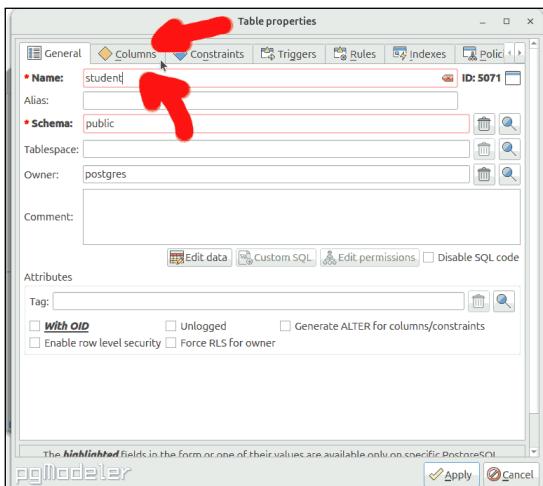


(19.1.5) A dialog called "Database Properties" opens. We want to set a proper name for our new DB. We choose `student_database` and then click on **Apply**.

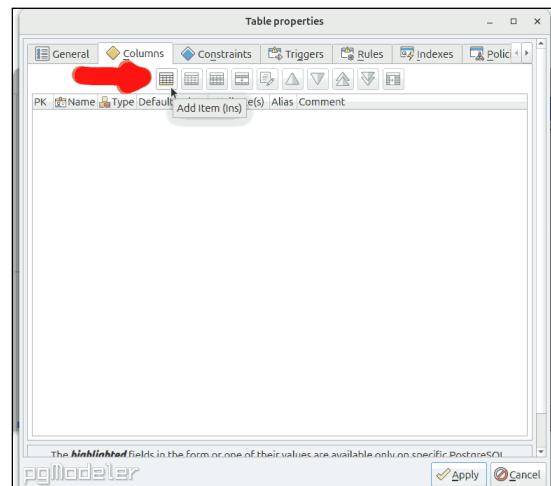


(19.1.6) Back in the ERD view, we again right-click into the (empty) diagram. In the popup-menu, we click on **New > Schema Object > Table**.

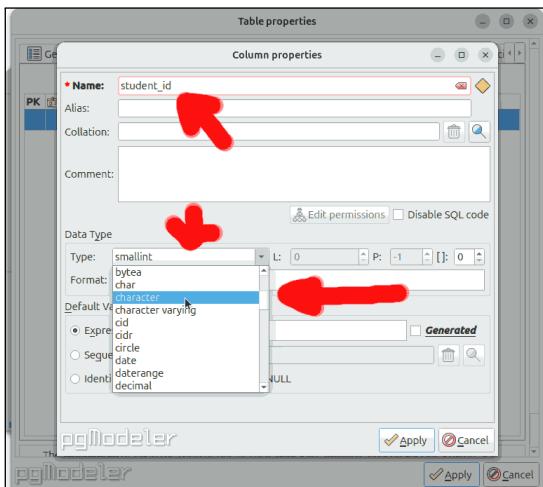
Figure 19.1: Developing logical models using PgModeler.



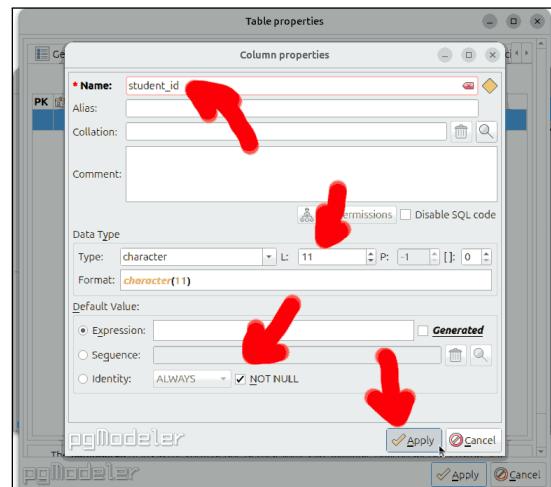
(19.1.7) The “Table properties” dialog opens. As table name, we enter `student`. Then we click on the register `Columns`.



(19.1.8) In the columns register, we click on the `Add Item` symbol



(19.1.9) We want to add a column for the university-issued student ID. As name for this column, we choose `student_id`. As type, we choose `character`, i.e., the SQL datatype for fixed-length strings (all student IDs have the same length).



(19.1.10) As (fixed) length, we enter 11 in the `L:` field. We also mark the column as `NOT NULL`, meaning that there cannot be a student record without student ID. We click `Apply`.

Figure 19.1: Developing logical models using PgModeler (continued).

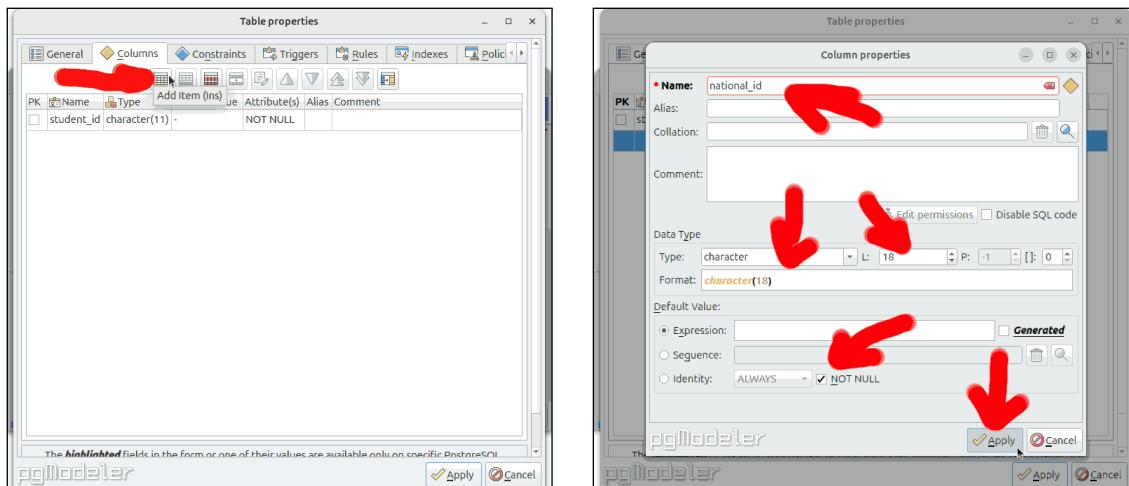
The new column appears in Figure 19.1.15 and we click again on `Add Item` . The next column we want to add is for storing the addresses of the students. We call this column `address`. Here, we again use strings of variable length (type `varchar`) as datatype. We again set the maximum length to 255 characters. We also again require the field to be `NOT NULL` and click `Apply` in Figure 19.1.16.

The new column appears in Figure 19.1.17 and we click `Add Item` .

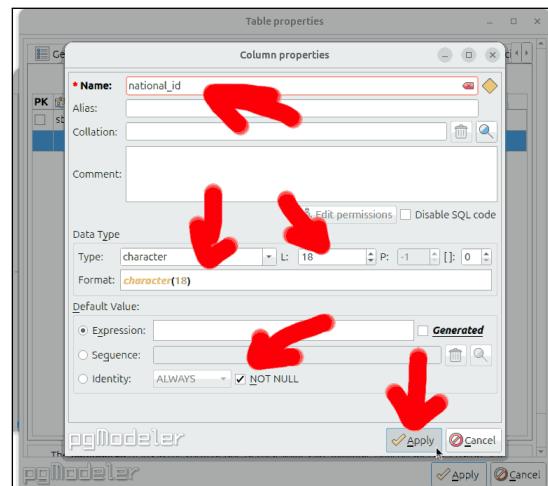
We now add a column for `mobile` phone numbers. Mobile phone numbers in China have 11 digits [508]. We can thus store them as strings (`character`) of the fixed length 11. We require that they must be specified (`NOT NULL`) and click `Apply` in Figure 19.1.18.

The new column appears in the table dialog and we again click on `Add Item` in Figure 19.1.19. Finally, we add the `DOB` in form of the `date_of_birth` column. The datatype here is `date`. Like all the columns so far, we require that DOBs to be `NOT NULL`. We click `Apply` in Figure 19.1.20.

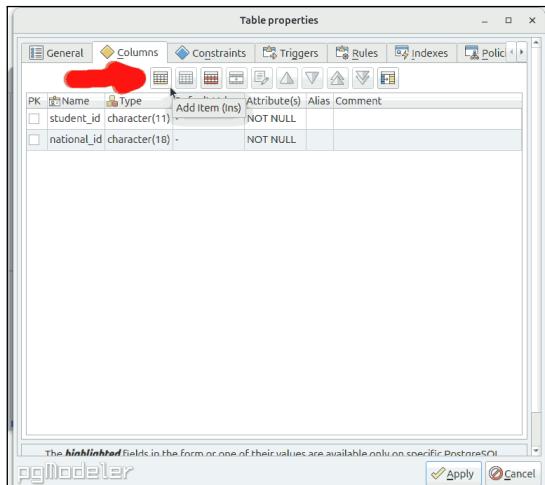
The new column appears in Figure 19.1.21. In the above text, you may have noticed that we are quite lenient with the data. For example, mobile phone numbers are not strings of arbitrary characters, but consist only of digits. Chinese ID numbers also are composed of digits, with the exception that the last character might be an `X`. Also, we should probably not permit arbitrary dates as DOBs. Even



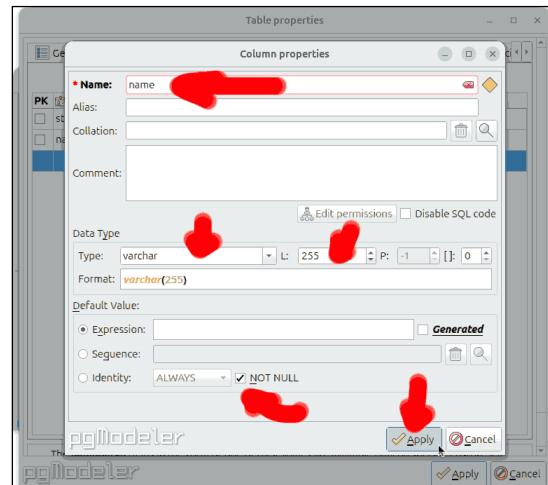
(19.1.11) The new column appears in the dialog. We click again on **Add Item**.



(19.1.12) We add the column **national_id** for storing Chinese ID numbers (中国公民身份号码). Such numbers are strings (**character**) of the fixed length 18. We also mark this column as **NOT NULL**, meaning that every record must have one. We click **Apply**.



(19.1.13) The new column appears and we click **Add Item**.



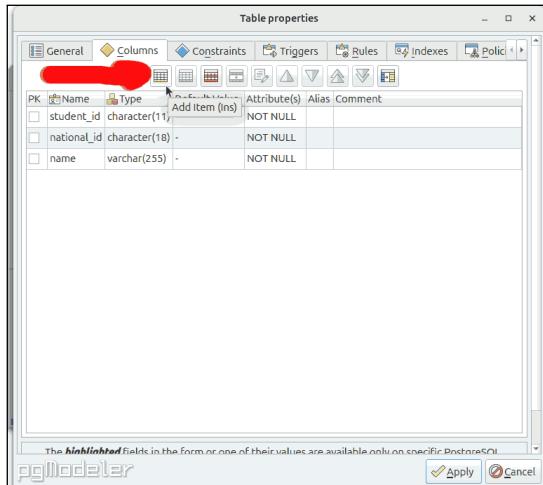
(19.1.14) We define the column **name** for student names. Names are of variable length (type **varchar**) and we set the **maximum** length 255. Each student must have a name, so we again specify **NOT NULL** and click **Apply**.

Figure 19.1: Developing logical models using PgModeler (continued).

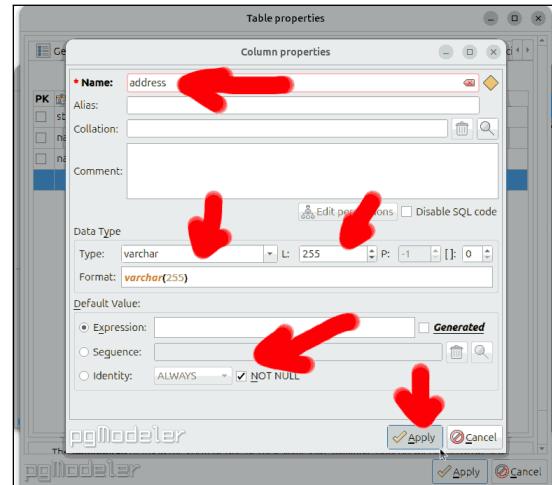
though September 23, 1811 would be a totally valid date, as the DOB of a student it would be unusual. Actually, we already learned how to deal with such restrictions on valid data back in [Section 9.2](#) (The Table “customer”): by using constraints. We also did not yet define a primary key (see [Definition 18.14](#)) for our table.

We click on the register **Constraints**, because now we want to add validity rules for our data. In the **Constraints** register, we click **Add Item**, as shown in [Figure 19.1.22](#). If you think about, we can consider the fact that a column is the *primary key* as a combination of a **UNIQUE** and a **NOT NULL** constraint (maybe together with some special indexing for fast access). So first, we want to choose a primary key.

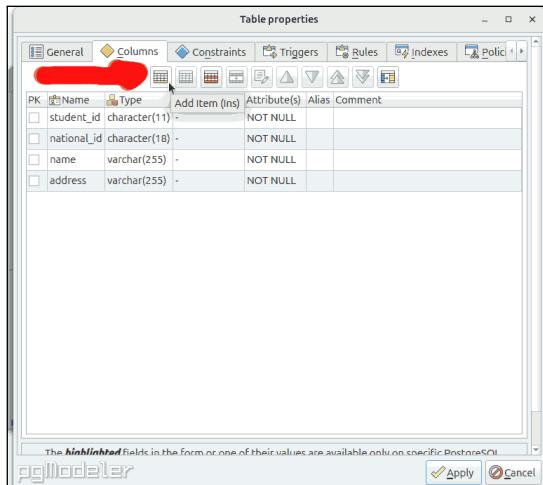
What would be the most suitable column for use as primary key? The columns **name**, **address**, and **date_of_birth** are unsuitable – if not for obvious reasons – then at least because they are not necessarily unique. The two columns **student_id** and **national_id** both look promising a primary keys. However, a person may enroll several times, maybe first as Bachelor and later as Master’s student. Hence, **national_id** is not necessarily unique. But for each enrollment, the person gets a



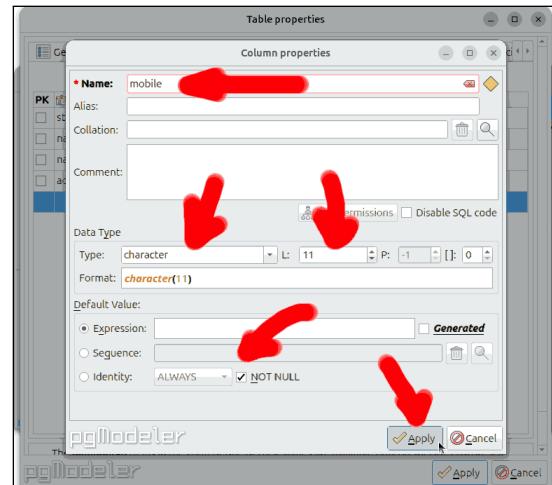
(19.1.15) The new column appears in the dialog. We click again on **Add Item**.



(19.1.16) Addresses, too, are strings of variable length (type `varchar`). We again set the maximum length to 255, require the field to be `NOT NULL`, and click **Apply**.



(19.1.17) The new column appears and we click **Add Item**.



(19.1.18) We now add a column for `mobile` phone numbers. In China, these are strings (`character`) of the fixed length 11. We require that they must be specified (`NOT NULL`) and click **Apply**.

Figure 19.1: Developing logical models using PgModeler (continued).

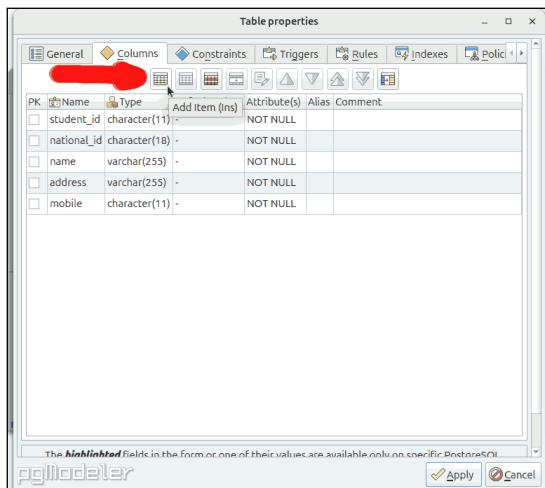
new `student_id`, which therefore is unique. As first constraint, we thus want to define `student_id` as the primary key of our table.

We call this constraint `student_student_id_pk` and select `PRIMARY KEY` as type. We select the column `student_id` in the `Column` drop-down box and click on **Add Item** in Figure 19.1.23. The column `student_id` appears in the `Columns` list. We click on **Apply** in Figure 19.1.24.

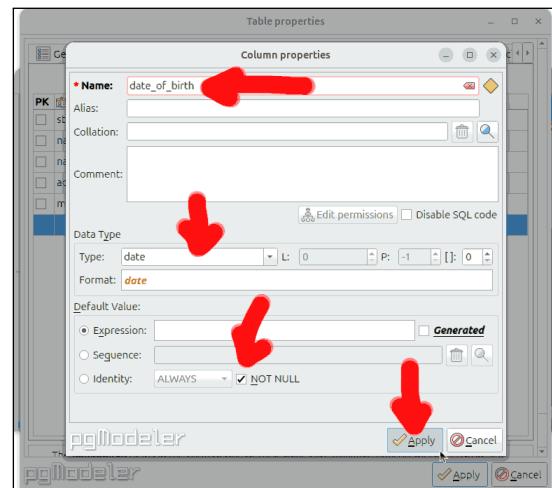
Best Practice 18

Constraints should have descriptive names [59]. If some table modification fails, we will see the name of the constraint that was violated. If the name makes sense and is easy to understand, then this makes it easier to find out what went wrong and why.

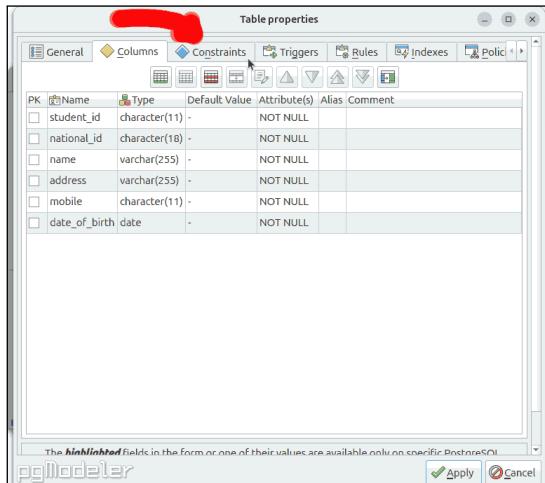
The new constraint appears and we click on **Add Item** in Figure 19.1.25. We now want to add a constraint checking that the national ID is correct. We call it `student_national_id_check` and select `CHECK` in the `Type` drop-down box in Figure 19.1.26. Indeed, we are going to create a `CHECK`



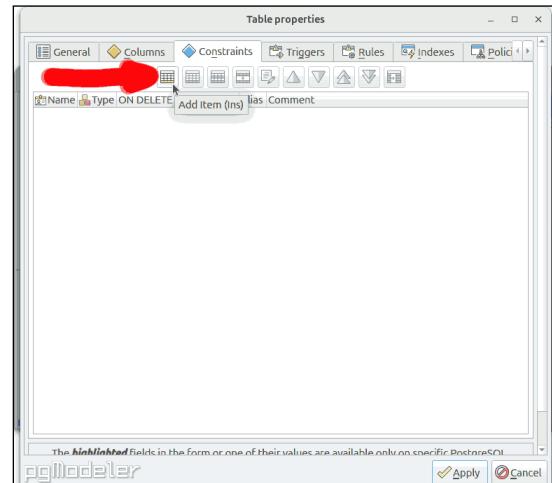
(19.1.19) The new column appears in the dialog. We click again on **Add Item**.



(19.1.20) Finally, we add the Date of Birth (DOB) in form of a `date_of_birth` column. The type here is `date` and DOBs are required to be `NOT NULL`. We click **Apply**.



(19.1.21) The new column appears. We click on the register **Constraints**, because now we want to add validity rules for our data.



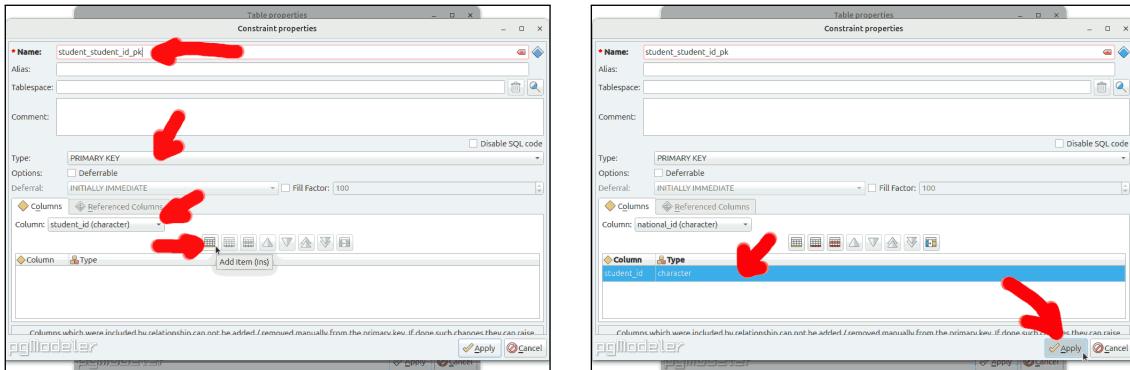
(19.1.22) In the **Constraints** register, we click **Add Item**.

Figure 19.1: Developing logical models using PgModeler (continued).

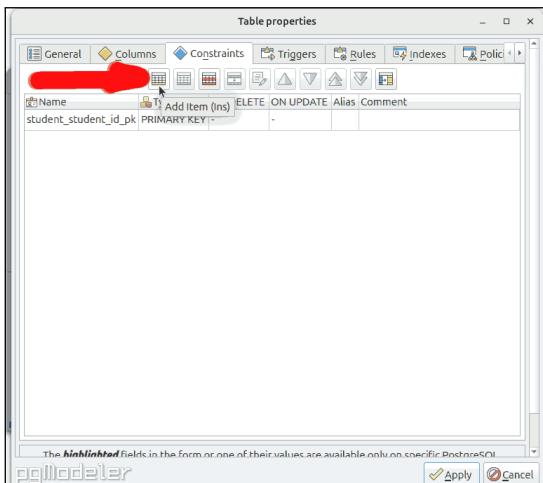
constraint. For this, we just need to provide an `SQL Expression`. This expression is evaluated whenever a row is added to the table or when a row is changed. If it then returns `TRUE`, everything is fine. If it returns `FALSE`, then the change will not be made.

So back to the Chinese ID numbers (中国公民身份号码). How do we check them? Standard GB11643-1999 公民身份号码 (*Citizen Identification Number*) [507] tells us that the first six digits are the administrative division code. The next eight digits are the DOB in format YYYYMMDD, followed by three digits of order code. The last character is a single checksum digit (which can be X). We could check this in a super fancy fashion. We could get our hands on a list of the actual valid values for the first digits, assuming that not all possible 1 000 000 possible administrative division codes are actually valid. We could even try to compute the checksum digit and check whether it matches.

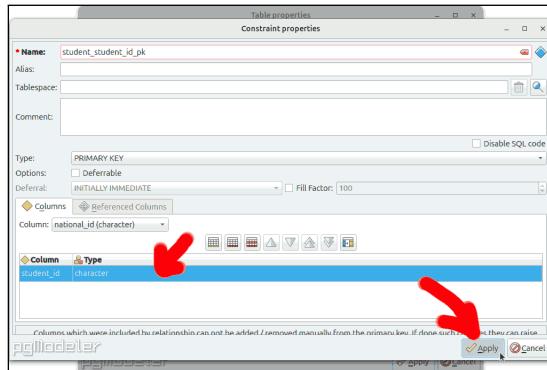
This is all too complicated for us. Instead, we will resort to a `regular expression` (`regex`) to check the field like back in Section 9.2 (The Table “customer”). We write `national_id ~ '^\d{6}((19)|(20))\d{9}[0-9X]$'`. The `national_id ~ xxx` means that the value of `national_id` must match to some regex `xxx`. In the regex, `\d` indicates the start of the text. `^\d{6}` means that six digits must immediately follow `\d`, i.e., be right at the start of the string. Then comes `((19)|(20))`, which means that the next two characters must be either ‘19’ or ‘20’. These two



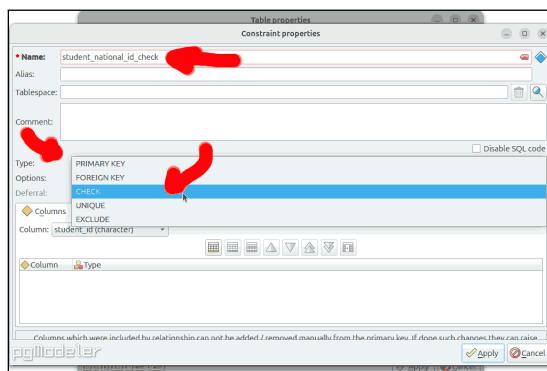
(19.1.23) As first constraint, we want to define `student_id` as the primary key of our table. We call this constraint `student_student_id_pk` and select `PRIMARY KEY` as type. We select the column `student_id` in the `Column` drop-down box and click on `Add Item`.



(19.1.25) The new constraint appears and we click on `Add Item`.



(19.1.24) The column `student_id` appears in the `Columns` list. We click on `Apply`.



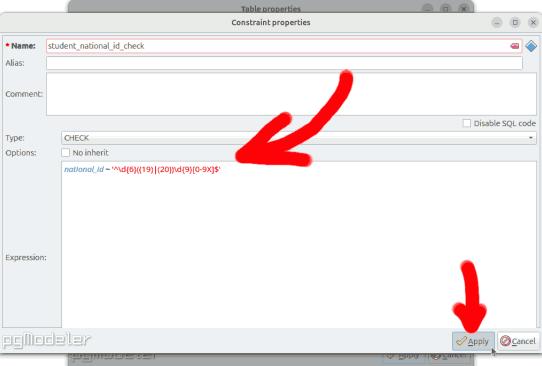
(19.1.26) We now want to add a constraint checking that the national ID is correct. We call it `student_national_id_check` and select `CHECK` in the `Type` drop-down box.

Figure 19.1: Developing logical models using PgModeler (continued).

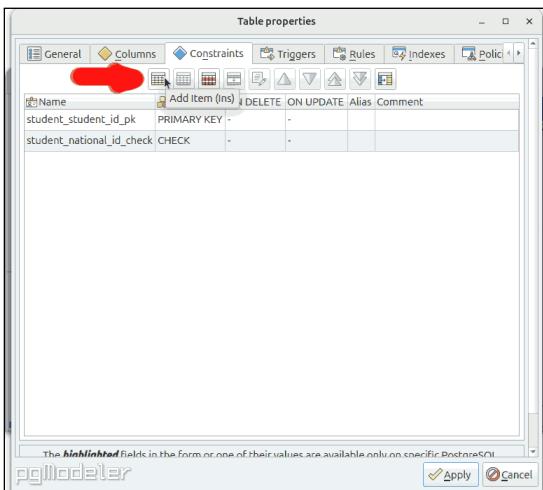
characters are the first two digits of the year of birth. We do not permit DOBs before the year 1900 or after 2099. After that, we require nine digits to follow via `\d{9}`. We could do this a bit better to enforce that next two digits would be valid months `01..12` and that the two digits following that be valid days-of-month, i.e., `01..31`. In a *real* application, we would totally do that ... but here I leave this as exercise to the interested reader. Anyway, this means we now have $6 + 2 + 9 = 17$ digits. This leaves only the final checksum character, which can be any digit from 0 to 9 or X. This is expressed by the `[0-9x]`. The textil\$ that follows marks the end of the string, which, hence, must come directly after the checksum digit.

You may ask: Why do we create such a trivial constraint? Well, this constraint would still guard against several possible typos. Verifying the checksum with a `regex` is probably not possible anyway, at least not with a regex of reasonable complexity. Eventually, we would create an application program through which the administrative staff enters student information. This program should then check the checksum of the `national_id` field.

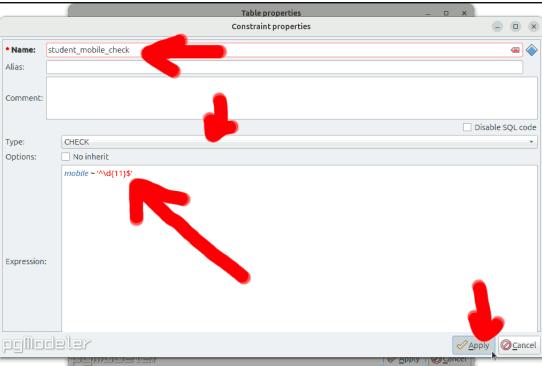
So you may ask: If we check the national ID value in the application anyway, then why do we put a constraint here? We could just leave it away and assume that the application will check the validity of the field. The answer is *defense in depth*.



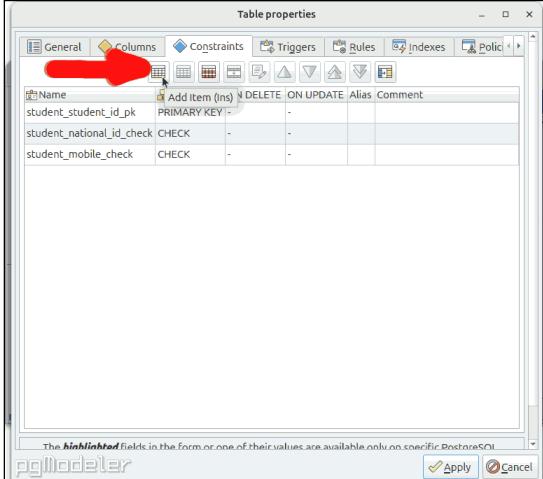
(19.1.27) **CHECK** constraints are specified as SQL **Expression**. To validate the field `national_id`, we specify the regex `national_id~'^\d{6}((19)|(20))\d{9}[0-9X]$'`. We click **Apply**.



(19.1.28) The new constraint appears and we click on **Add Item**.



(19.1.29) We create a **CHECK** constraint for the column `mobile`. The expression `mobile ~ '^\\d{11}$'` demands an 11 digit string. We click on **Apply**.



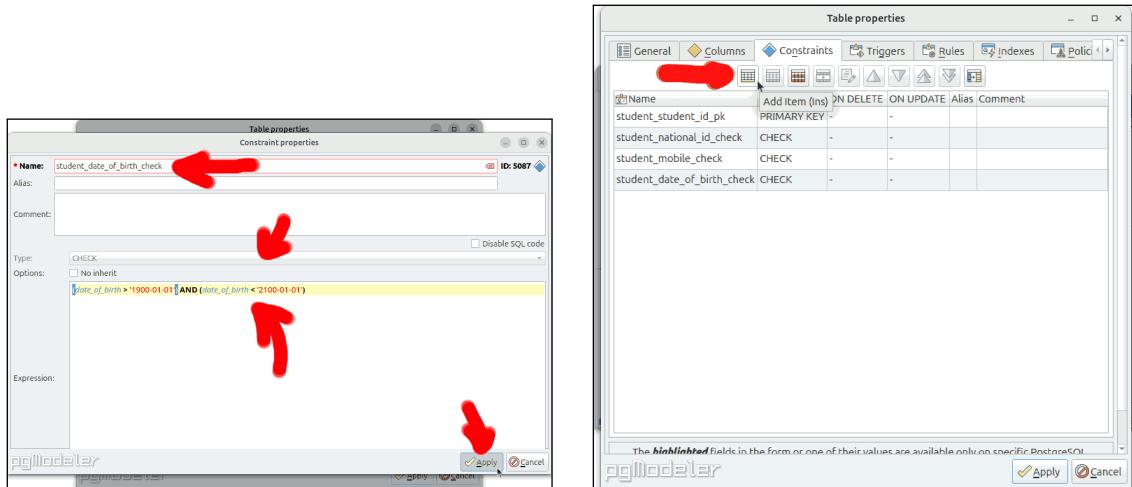
(19.1.30) The new constraint appears and we click on **Add Item**.

Figure 19.1: Developing logical models using PgModeler (continued).

Best Practice 19

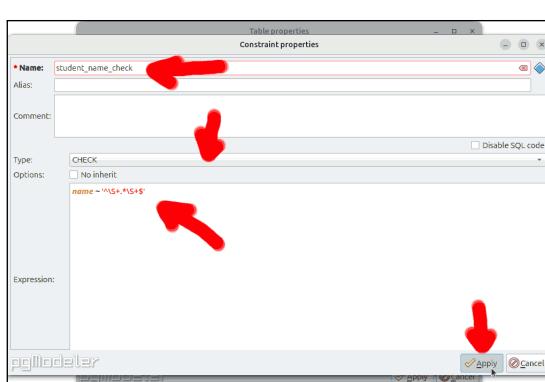
Data should be checked at all levels of an application, in the forms where it is entered, in the DB via constraints, and back in the application when it is loaded from the DB. The more lines of defense we create with constraints, static checks, and dynamic checks, the higher is our chance to discover errors early, to prevent them from propagating, and to pinpoint the reason of errors. This gives us the best chance to locate and fix the error if it is a problem with a program as well as to prevent errors resulting from typos to enter and pollute our DB.

This best practice also fits well to what we wrote in *Programming with Python* [482] for the Python programming language:



(19.1.31) We specify the **CHECK** constraint for the DOB and call it `student_date_of_birth_check`. We combine the condition `date_of_birth > '1900-01-01'` (demanding that students may not be born before the year 1900) and `date_of_birth < '2100-01-01'` (which prevents students born in the 22nd century) with **AND**. We click **Apply**.

(19.1.32) The new constraint appears and we click on **Add Item**.



(19.1.33) We create a **CHECK** constraint for the column `name` and call it `student_name_check`. The expression `name ~ '^\\S+.*\\S+$'` demands that names both start and end with printable characters and may contain an arbitrary number of characters in between. We click on **Apply**.

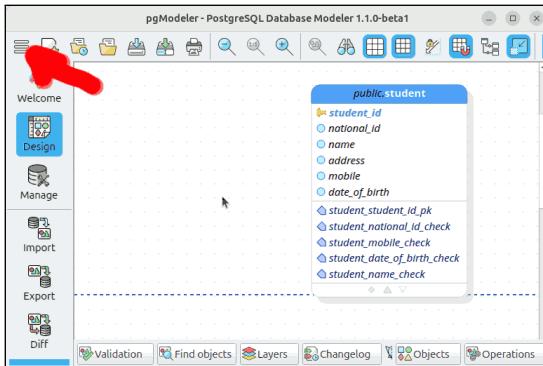
(19.1.34) The new constraint appears. We stop here and create the table model by clicking on **Apply**.

Figure 19.1: Developing logical models using PgModeler (continued).

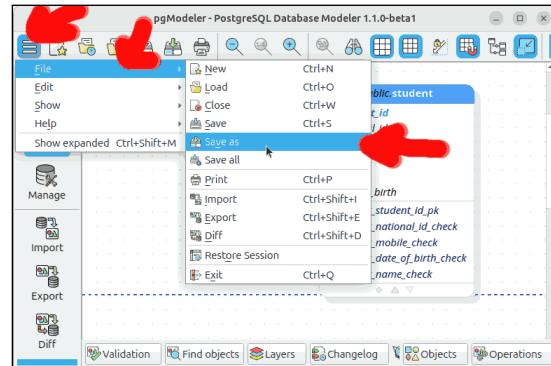
Best Practice 20

Errors should *not* be ignored and input data should *not* be artificially sanitized. Instead, the input of our functions should be checked for validity wherever reasonable. Faulty input should always be signaled by errors breaking the program flow. [In Python,]Exceptions should be raised as early as possible and whenever an unexpected situation occurs.

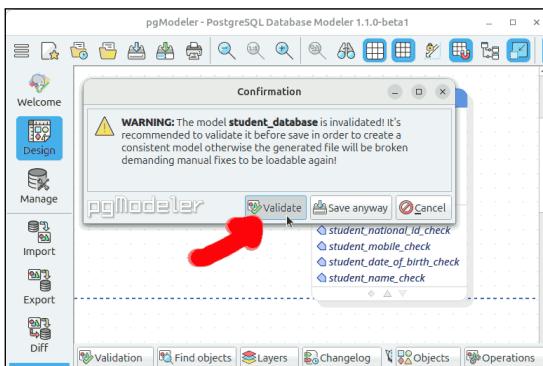
So it makes sense to specify as many constraints as possible wherever possible, even if they can only check some aspects of the data. Either way, after specifying the constraint, we click **Apply** in Figure 19.1.27 and the constraint is specified. The new constraint appears and we click on **Add Item**



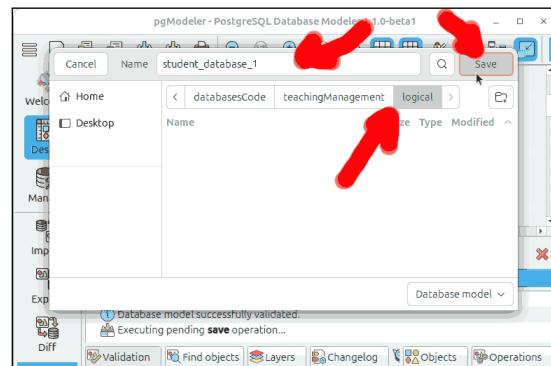
(19.1.35) The new table appears in our ERD, with a syntax similar to what we had in [Section 18.6](#). We click on the main menu \equiv .



(19.1.36) It is time to save the model to a file. We click on $\equiv \gg \text{File} \gg \text{Save as}$.



(19.1.37) Since our model is new and unchecked (or changed), we get asked to validate it. Heck, why not, we click on **Validate**.



(19.1.38) We can now select a file name and directory where the model should be stored. We choose the name **student_database_1** and click **Save**.

Figure 19.1: Developing logical models using PgModeler (continued).

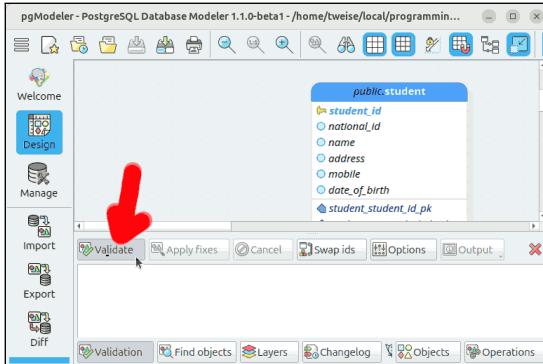
in [Figure 19.1.28](#).

We now create a similar **CHECK** constraint for the column **mobile** in [Figure 19.1.29](#). We call it **student_mobile_check**. The expression **mobile ~ '^\d{11}\$'** demands an 11 digit string: It states that the **mobile** value must, right at its begin (**^**), have eleven digits (**\d{11}**), and then the end of the string follows immediately (**\$**). We click on **Apply**.

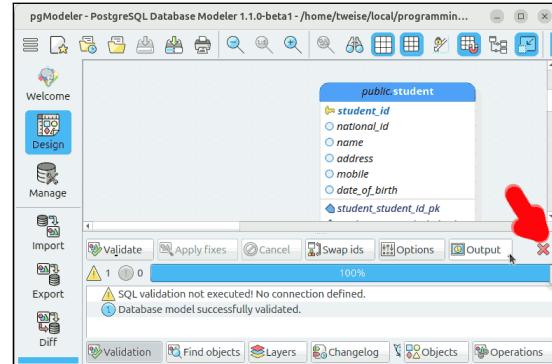
The new constraint appears and we click on **Add Item**  again in [Figure 19.1.30](#). We want to specify the **CHECK** constraint for the **DOB** and call it **student_date_of_birth_check**. One condition is that **date_of_birth > '1900-01-01'**, i.e., we demand that students may not be born before the year 1900. Another condition is that **date_of_birth < '2100-01-01'**, which prevents students born in the 22nd century. We combine both conditions with **AND**. We click **Apply** in [Figure 19.1.31](#). In [Figure 19.1.32](#), the new constraint appears and we click on **Add Item** .

It may be a cool idea to define a **CHECK** constraint that validates whether the DOB given as **date_of_birth** fits to the value provided in **national_id**. After all, the DOB is directly present in the national Chinese ID number, too. **SUBSTR(national_id, 7, 8)** should give us the eight digits of the birth date from the national ID [424]. **TO_CHAR(date_of_bith, 'YYYYMMDD')** would probably give us the DOB in the same format [116]. Hence, we could add a **CHECK**-constraint comparing the two and failing if they are different. This would be a pretty strong guard against typos. We leave this to the interested reader.

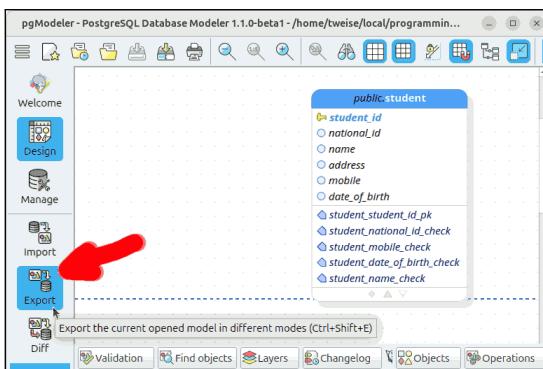
As final constraint, we want to set some restriction on valid names. We create a **CHECK** constraint for the column **name** and call it **student_name_check**. We specify the expression **name ~ '^\S+.*\S+\$'**. The **\S** matches a single character that is not whitespace, i.e., a character that is neither space nor a line break nor a tabulator. The **+** means “one or multiple repetitions of the previous”, so **\S+** means “one or multiple non-space characters”. We want the name to start (and end) with a letter



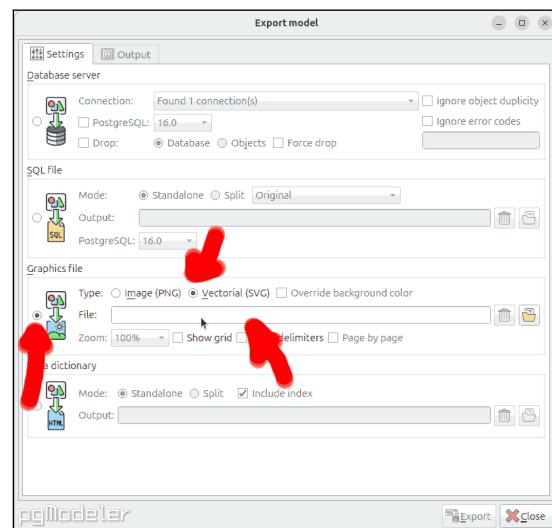
(19.1.39) This takes us back to the main window. We notice a bar with new buttons, including one called **Validate**. We click on it.



(19.1.40) Our model gets validated. It is OK. We can close the log.



(19.1.41) We now want to export the model and, thus, click on **Export**.



(19.1.42) We want to store it as graphic. So we click on **Graphics file** and select **Vectorial (SVG)**, which will store the model in SVG format. We then click into the **File** bar.

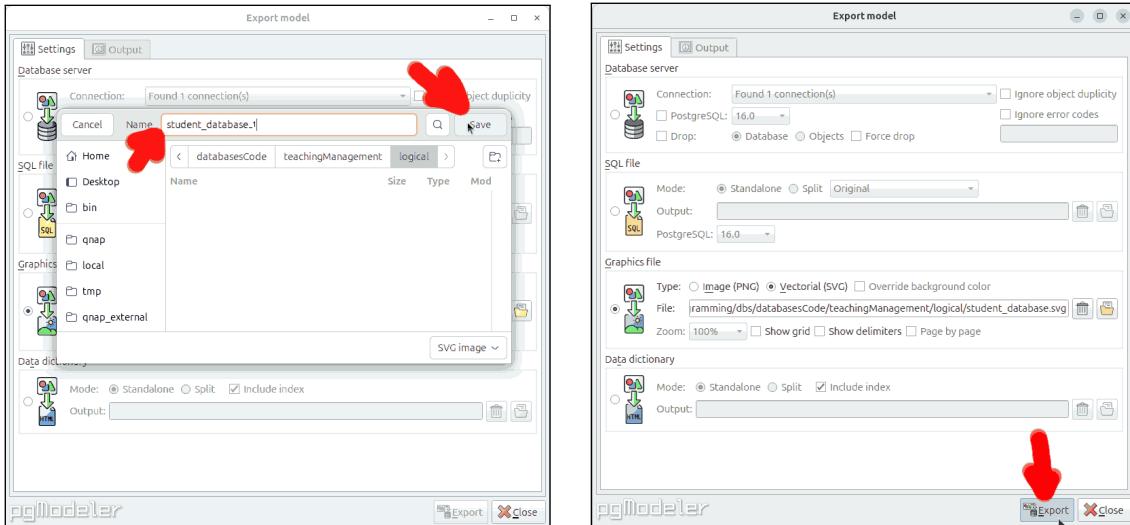
Figure 19.1: Developing logical models using PgModeler (continued).

or Chinese character or maybe Indian character or whatever, but no reasonable name starts with a space. We force such a non-space character to be at the beginning (`^\S+`) and at the end (`\S+$`) of the string `name`¹. Inbetween, we permit an arbitrary number (`*`) of arbitrary characters (`.`). Thus, this expression demands that names both start and end with printable characters and may contain an arbitrary number of characters in between. We click on **Apply** in Figure 19.1.33.

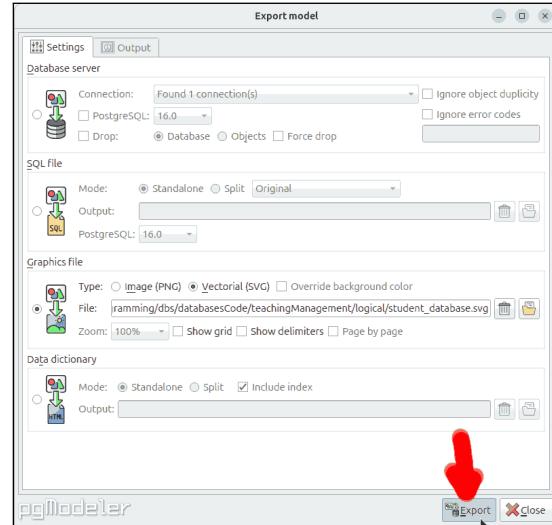
The new constraint appears. We stop here. Yes, we could add a similar constraint for the columns `address`. And indeed, we did not match the DOB stored as `date_of_birth` in the table against the DOBs encoded in the field `national_id`. We also did not impose a constraint upon the `student_id`, except that these values have to be `UNIQUE` and `NOT NULL`. Well, for this simple example, I think we are good. We now create the table model by clicking on **Apply** in Figure 19.1.34.

The new table appears in our ERD, with a syntax similar to what we had in Section 18.6. It is now time to save this logical model to a file. We click on `File > Save as` in Figure 19.1.36. Since our model is new and unchecked (or changed), we get asked to validate it. This seems to be a reasonable request and we click on **Validate** in Figure 19.1.37. Next we can select a file name and directory where the model should be stored. We choose the name `student_database_1` and click **Save** in Figure 19.1.38.

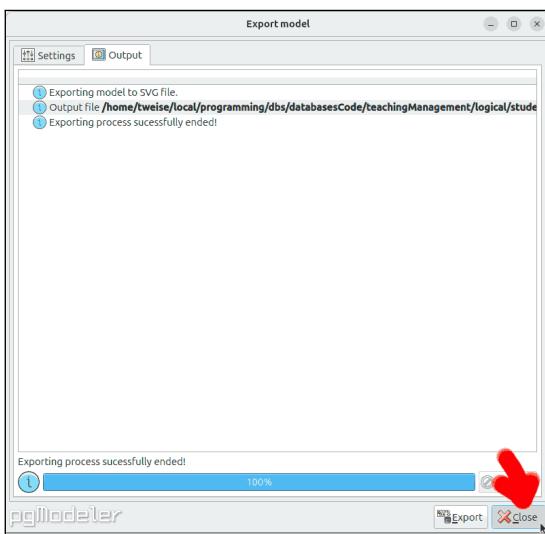
¹On second thought, we could have left the `^`s away.



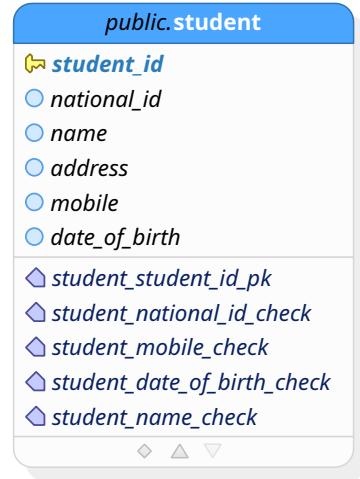
(19.1.43) We again get to select a file name and stick with `student_database_1`. We click on `Save`.



(19.1.44) We can now click on `Export`.



(19.1.45) The file has been exported, we can close the dialog.



(19.1.46) This is the exported vector graphic. It looks quite nice. If we had done a bigger model with many tables, it would probably look quite exciting.

Figure 19.1: Developing logical models using PgModeler (continued).

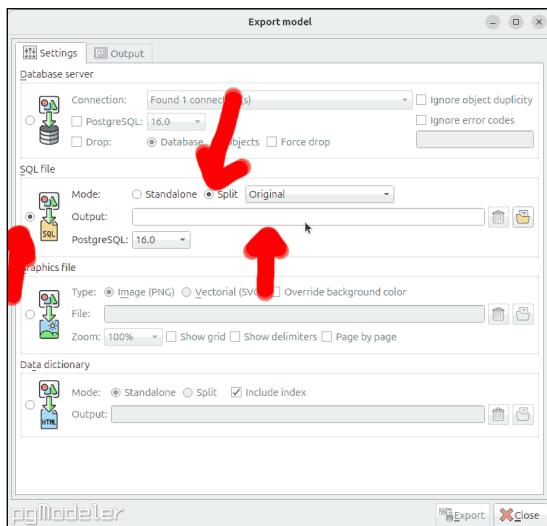
This takes us back to the main window. We notice a bar with new buttons, including one called `Validate`. This must be the meaning of the request to validate our model in Figure 19.1.37. So now we click on it in Figure 19.1.39. Our model gets validated. It is OK. We can close the log in Figure 19.1.40.

So far, however, we did not really do anything useful with this logical model. When we used yEd to draw our conceptual model, we could export it as Scalable Vector Graphics (SVG) graphic. We can also do this with models created in the PgModeler. We therefore click on `Export` in Figure 19.1.41.

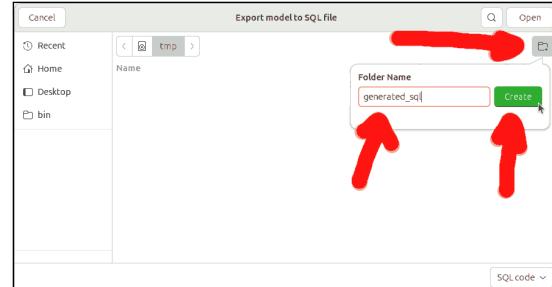
We want to store the model as SVG graphic. Therefore, we click on `Graphics file` and select `Vectorial (SVG)`. We then click into the `File` bar in Figure 19.1.42. We again get to select a file name and again stick with `student_database_1`. In Figure 19.1.43, we click on `Save`. This takes us back to the export dialog in Figure 19.1.44. Here we can now click on `Export`. The file has been exported, we can close the dialog in Figure 19.1.45.

In Figure 19.1.46, we illustrate the exported vector graphic. It looks quite nice. If we had done a bigger model with many tables, it would probably look quite exciting.

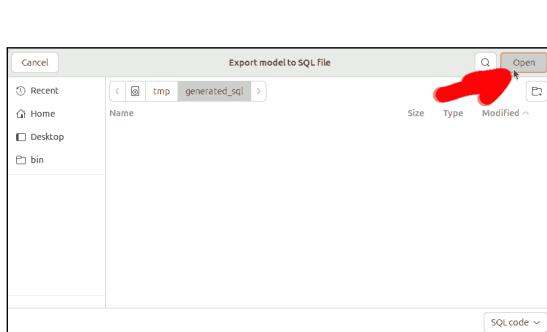
This is the extend of what we could do on the conceptual modelling level, too. There, we could



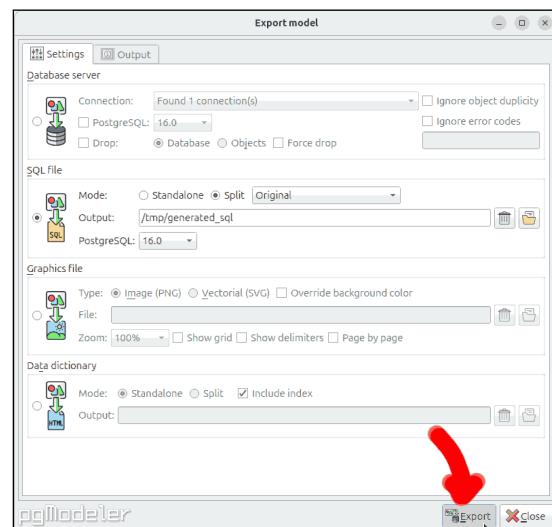
(19.1.47) We open the [Export] dialog again. This time, we want to export our model to SQL. We click on [SQL file]. **Important:** Mark the output as *Split*. Then click on the file bar.



(19.1.48) Because we want to output the model using the *split* method, this will create multiple SQL files. Instead of a file name, we need to choose a folder name. We therefore create a new folder and choose `generated_sql` as its name.



(19.1.49) The folder is created, we click on [Open].



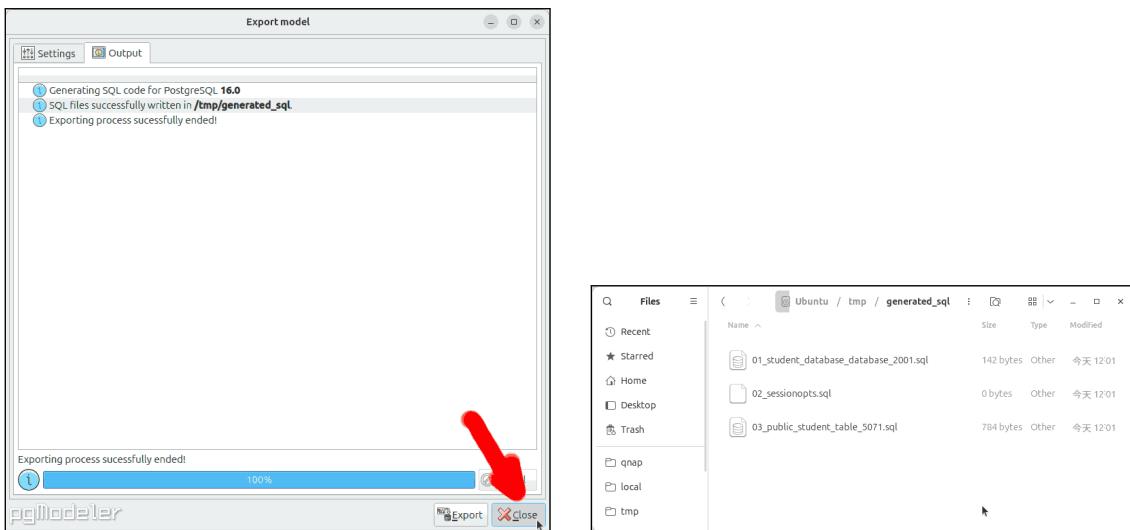
(19.1.50) This takes us back to the *Export model* dialog, where we click on [Export].

Figure 19.1: Developing logical models using PgModeler (continued).

paint a model and print it as graphic. We also painted a model now. The logical model we did paint had a much tighter syntax is a formal model. We used specific **SQL** datatypes, **SQL** constraints, and could do nothing that cannot be done with **SQL**. In stark contrast, yEd allows us to paint almost arbitrary graphics. We could have drawn stars and clouds into our **ERD** if we wanted to.

However, sticking to **SQL** (or, more precisely, the **PostgreSQL** flavor of it), has another advantage: We can actually create a **DB** directly from our model!

We therefore open the **Export** dialog again. This time, we want to export our model to **SQL** and we therefore select **[SQL file]** option in the **Export** dialog. It is very important to export the model to multiple files, i.e., to select the “*Split*” option. If we export everything into one file, then the commands to create of the `student_database` DB and the creation of the `student` table will in the same file. If we submit the contents of this file to `psql`, then the `student` table will not be created inside the `student_database` DB. Instead, the script will first create the `student_database` DB and then, in the public schema of the **PostgreSQL** server, also create the `student` table alongside it. The `student` table will not be inside the `student_database` DB, but in the public schema of the **DBMS**. It is therefore



(19.1.51) The logical model is exported to SQL. We can close the dialog by clicking on **Close**.

(19.1.52) We can browse to the folder using whatever file browser the OS offers. We find that it contains several files, whose contents are shown in Listings 19.1 and 19.3. We can execute them on the PostgreSQL server using `psql`. We do this in Listings 19.2 and 19.4.

Figure 19.1: Developing logical models using PgModeler (continued).

Listing 19.1: This auto-generated SQL script creates the DB `student_database`. (stored in file `01_student_database_database_2001.sql`; output in Listing 19.2)

```

1 -- object: student_database | type: DATABASE --
2 -- DROP DATABASE IF EXISTS student_database;
3 CREATE DATABASE student_database;
4 -- ddl-end --

```

Listing 19.2: The stdout of the program `01_student_database_database_2001.sql` given in Listing 19.1.

```

1 $ psql "postgres://postgres:XXX@localhost" -v ON_ERROR_STOP=1 -e bf 01
   ↪ _student_database_database_2001.sql
2 CREATE DATABASE
3 # psql 16.12 succeeded with exit code 0.

```

vital to click **SQL file**. After that is done, we click into the **Output** bar to select a target directory in Figure 19.1.47.

We now need to create a new directory where the **SQL** files should be stored. We click on the folder creation symbol, choose `generated_sql` as directory name in Figure 19.1.48 and then click on **Create**. We then click **Open** back in the directory selection dialog as shown in Figure 19.1.49. This takes us back to the **Export** dialog where click on **Export**, as shown in Figure 19.1.50.

The model is exported and we close the dialog in Figure 19.1.51. In Figure 19.1.52, we enter the folder into which we exported the files. We find three new files. The first one, named something like `01_student_database_database...sql`, is the SQL script for creating the DB. We list its contents in Listing 19.1. Besides some comments, we only find the `CREATE DATABASE student_database;` statement. This what we expect and we submit to to SQL using the `postgres` administrative account in Listing 19.2.

The second file is empty, so we can ignore it. The third file is named something like `03_public_student_table...sql`. This file contains the commands for creating the table `student`. Its contents are shown in Listing 19.3. There is not really anything there that goes beyond what we discussed in our initial example back in Chapter 9 (Creating Tables and Filling them with Data). We see the `CREATE TABLE` command together with statements for creating the constraints.

Listing 19.3: This auto-generated SQL script creates the table `student` inside the DB `student_database`. (stored in file `03_public_student_table_5071.sql`; output in Listing 19.4)

```

1  -- object: public.student | type: TABLE --
2  -- DROP TABLE IF EXISTS public.student CASCADE;
3  CREATE TABLE public.student (
4      student_id character(11) NOT NULL,
5      national_id character(18) NOT NULL,
6      name varchar(255) NOT NULL,
7      address varchar(255) NOT NULL,
8      mobile character(11) NOT NULL,
9      date_of_birth date NOT NULL,
10     CONSTRAINT student_student_id_pk PRIMARY KEY (student_id),
11     CONSTRAINT student_national_id_check CHECK (national_id ~ '^\d{6}((19)|'
12         ↪ (20))\d{9}[0-9X]$'),
13     CONSTRAINT student_mobile_check CHECK (mobile ~ '^\d{11}$'),
14     CONSTRAINT student_date_of_birth_check CHECK ((date_of_birth > '
15         ↪ 1900-01-01') AND (date_of_birth < '2100-01-01')),
16     CONSTRAINT student_name_check CHECK (name ~ '^\S+.*\S+$')
17 );
18 -- ddl-end --
19 ALTER TABLE public.student OWNER TO postgres;
20 -- ddl-end --

```

Listing 19.4: The stdout of the program `03_public_student_table_5071.sql` given in Listing 19.3.

```

1 $ psql "postgres://postgres:XXX@localhost/student_database" -v
2   ↪ ON_ERROR_STOP=1 -ebf 03_public_student_table_5071.sql
3 CREATE TABLE
4 ALTER TABLE
5 # psql 16.12 succeeded with exit code 0.

```

The only peculiarity is the `ALTER TABLE`, which can change a table after its creation [5]. Here, it assigns the owner of the table. We can ignore this for now.

When we pass the contents of this file to `psql`, we must make sure to send them to the DB `student_database`. We do so in Listing 19.4 and the execution completes successful.

We thus have created a DB from our logical model. In Listing 19.5, we test this new DB by inserting some records. The first two are OK, the third one violates the constraints on DOBs and on the `national_id`. The first two insertions succeed in Listing 19.6, but the third one fails, as expected. When the third request fails, we get a very clear and understandable output. The `student_date_of_birth_check` constraint was violated. Because we chose a clear name for our constraint, we can easily find out what went wrong and where and why.

Finally, in Listing 19.7, we provide the SQL code for deleting the DB again. After executing it in Listing 19.8, we can continue our work on a clean PostgreSQL server. Notice that this script, like the DB creation script, is executed with a connection URI that does not specify a DB to work on. Otherwise, if we would connect to the DB `student_database` and then attempt to delete while being connected to it, this would fail with the error message “cannot drop the currently open database.”

In summary, we can conclude that this approach works. The DB and the table were created exactly as expected. What does this mean? It means the following:

Useful Tool 6

With PgModeler, we have a tool in our hands that allows us to basically draw logical models for DBs as ERDs. These models are easy-to-understand graphics that follow crow’s foot notation.

PgModeler can connect to a PostgreSQL server and directly push the models to it or load a logical model from the server. It can also export logical models as SQL scripts that we then can execute. It therefore offers us a convenient GUI to design the logical schema of a DB.

Listing 19.5: We can insert some records into the table `student`. (stored in file `insert.sql`; output in Listing 19.6)

```

1  /** Insert some rows into the student database. */
2
3  -- Insert records that can be inserted correctly.
4  INSERT INTO student (student_id, national_id, name, address, mobile,
5                      date_of_birth) VALUES
6      ('1234567890', '123456199501021234', 'Bibbo', 'Hefei, China',
7       '12345678901', '1995-01-02'),
8      ('1234567891', '123456200508071234', 'Bebbo', 'Chemnitz, Germany',
9       '12345678902', '2005-08-07');
10
11 -- Print the records that were inserted.
12 SELECT student_id, name FROM student;
13
14 -- Try inserting an invalid record: The date of birth is way too early.
15 INSERT INTO student (student_id, national_id, name, address, mobile,
16                      date_of_birth) VALUES
17      ('1111111111', '123456022501011234', 'Liu Hui', 'Zouping, Shandong',
18       '12345678902', '0225-01-01');
```

Listing 19.6: The stdout of the program `insert.sql` given in Listing 19.5.

```

1  $ psql "postgres://postgres:XXX@localhost/student_database" -v
2      ↪ ON_ERROR_STOP=1 -e bf insert.sql
3  INSERT 0 2
4  student_id  | name
5  -----+-----
6  1234567890  | Bibbo
7  1234567891  | Bebbo
8  (2 rows)
9
10 psql:teachingManagement/logical/student_database_1/insert.sql:18: ERROR:
11     ↪ new row for relation "student" violates check constraint "
12     ↪ student_date_of_birth_check"
13 DETAIL:  Failing row contains (1111111111, 123456022501011234, Liu Hui,
14     ↪ Zouping, Shandong, 12345678902, 0225-01-01).
15 psql:teachingManagement/logical/student_database_1/insert.sql:18: STATEMENT
16     ↪ : INSERT INTO student (student_id, national_id, name, address,
17     ↪ mobile,
18             date_of_birth) VALUES
19      ('1111111111', '123456022501011234', 'Liu Hui', 'Zouping, Shandong',
20       '12345678902', '0225-01-01');
21 # psql 16.12 failed with exit code 3.
```

Of course, PgModeler is not the only such software. But it is quite nice, open source, and free. It is suitable for PostgreSQL, while other programs have been developed for other DBMSes. As said in Best Practice 1, good software engineers are both able and keen to learn new tools. To depart from this example with a clean slate, we execute the SQL script given in Listing 19.7 to delete the table and DB again.

Listing 19.7: Cleaning up after the student DB example. (stored in file `cleanup.sql`; output in Listing 19.8)

```
1 /* Cleanup after the example: Delete the student database. */
2
3 DROP DATABASE IF EXISTS student_database;
```

Listing 19.8: The stdout of the program `cleanup.sql` given in Listing 19.7.

```
1 $ psql "postgres://postgres:XXX@localhost" -v ON_ERROR_STOP=1 -eef cleanup.
  ↪ sql
2 DROP DATABASE
3 # psql 16.12 succeeded with exit code 0.
```

We currently are in the business of translating entity types to the relational data model, i.e., to tables. When entity types have composite attributes, these get recursively divided into their components. Each component of the composite attribute that cannot further be divided becomes an column in the table for the entity type. Multivalued attributes, i.e., attributes that can take on several values for each entity, instead need to go into their own, separate tables. Back when designing conceptual models, we had one variant of the *Student* entity type design that had both a composite and a multivalued attribute – see Figure 18.2.

We will now use this example to also exercise these steps of the conceptual-to-logical model mapping step. We therefore reprint the ERD from Figure 18.2 here as Figure 19.2.1. We already created a logical model for the *Student* entity type just now in Section 19.2.1. In that model, we had a simple attribute `name` and a single-valued attribute `mobile`. If we leave these two columns and the corresponding constraints away, then this model is a good starting point for translating Figure 18.2.

We either can create the exactly same model (without these columns and constraints) again in the PgModeler or we load the previous model and delete them. That's up to you. Either way, in Figure 19.2.2 we begin with this modified model variant. The model has the `student` table. We will first add the two new name-related columns. Therefore, we double-click on the table `student` to edit it.

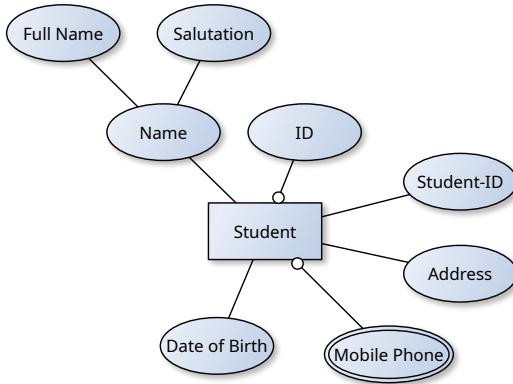
We want to add the columns `full_name` and `saltulation` to represent the flattened composite attribute `name` in Figure 19.2.3. First, we add a column `full_name`, which is a variable-length string with maximum length 255 that must be `NOT NULL`. We insist that full names must always be provided for students. Then, we add the column `saltulation`, which is a variable-length string with maximum length 255. For this one, we do not require the `NOT NULL` feature. If no salutation is provided, we simply assume that the full name can be used to address a student. After doing this Figure 19.2.4, we click `Apply`.

Multivalued attributes should go into their own table. We therefore need to create a second table in our logical model. We create a new table by right-clicking somewhere into our model (but *not* on the `student` table) and then selecting `New > Schema object > Table` in Figure 19.2.5. A suitable name for this new table is `mobile`. So we enter it as name. We then click on `Columns`, because we will now add several columns in Figure 19.2.6.

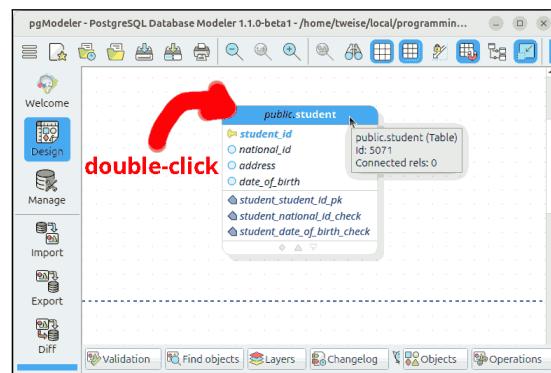
Our goal with this table is to store the mobile phone and a reference to the student in this table. Neither of them are necessarily unique: The same person may enroll first as Bachelor's and then, after graduation, as Master's student. This means that their mobile phone number may occur multiple times. Since each student can have multiple mobile phone numbers, the primary key `student_id` that we will need as foreign key is also not unique.

This means that we need a surrogate key. We already used surrogate keys in our factory example back in Chapter 9 (Creating Tables and Filling them with Data). We will do exactly the same here. However, for the sake of convenience, we will do so in the PgModeler. We create the column `id` of type `integer`. We mark it as `Identiy` which is (generated) `BY DEFAULT`. This will be roughly equivalent to how we generated IDs in Chapter 9. We do this in Figure 19.2.7 and then click `Apply`.

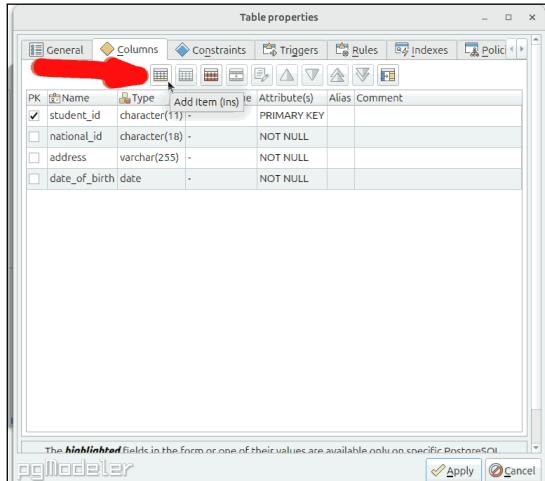
The next column we create is for the mobile phone numbers. To avoid calling it `mobile` as well, we here choose to call it `phone`. This column must have the same features as the same column in the



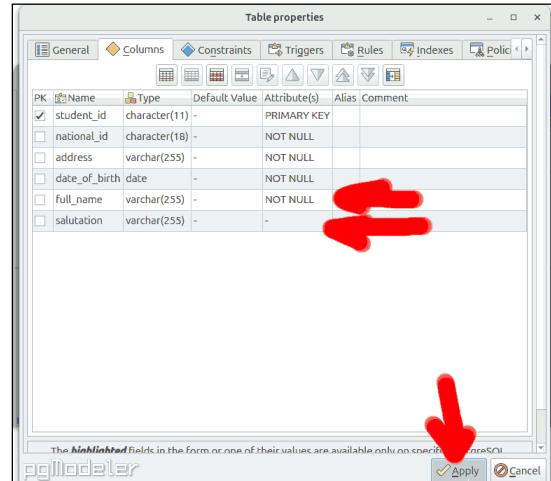
(19.2.1) We want to design a model where composite and multivalued attributes are represented. We therefore reprint the ERD from Figure 18.2 here.



(19.2.2) In PgModeler, we start with the same model as in Section 19.2.1, however without the columns `name` and `mobile` and without their corresponding constraints in the `student` table. We double-click on the table `student` to edit it.



(19.2.3) We want to add the columns `full_name` and `salutation` to represent the flattened composite attribute `name`.



(19.2.4) We add a column `full_name`, which is a variable-length string with maximum length 255 that must be `NOT NULL`. We add a column `salutation`, which is a variable-length string with maximum length 255. We click `Apply`.

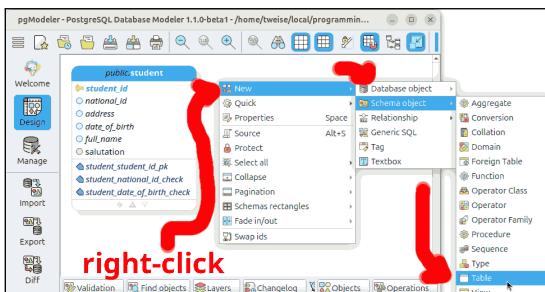
Figure 19.2: Creating a logical model represent students with a composite name attribute and multiple mobile phone numbers.

previous example: It is a fixed-length string of eleven characters and it must be `NOT NULL`. We create the column by clicking `Apply` in Figure 19.2.8.

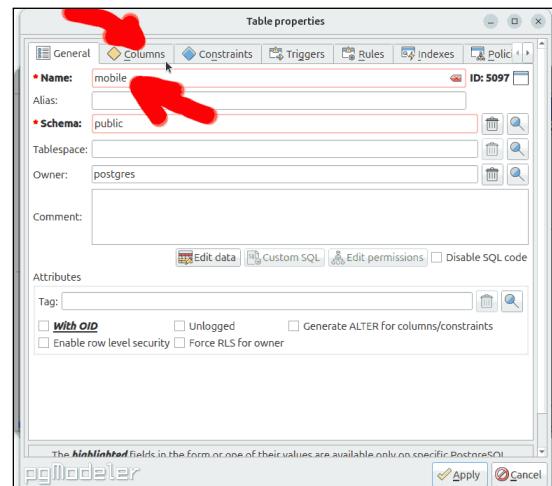
Finally, we need a column to hold the foreign key `student_id` pointing to rows in the `student` table. We therefore create a column `student`. Obviously, it must have the same datatype as our `student_id` column in the `student` table. This, too, happens to be a fixed-length string of eleven characters. Also obviously, it must be `NOT NULL`, because each mobile phone entry must be related to one student record. We create this column in Figure 19.2.9.

We have created the three columns in Figure 19.2.10 and move on to create `Constraints`. First we create a primary key constraint for the column `id` in Figure 19.2.11. There is nothing new about that. Then we re-create the mobile phone number checking constraint in Figure 19.2.12, which is the same as in the previous example as well.

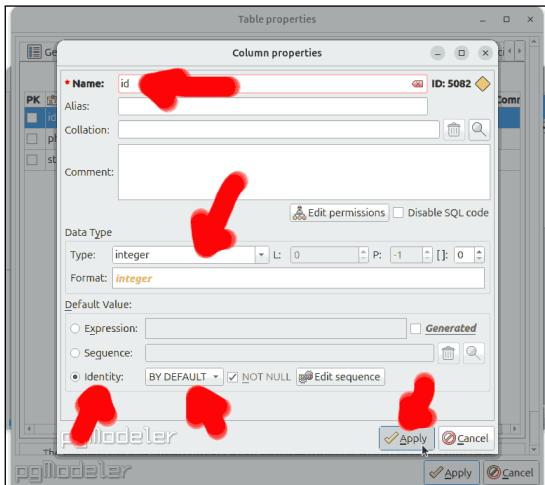
Finally, we want to link the rows in the table `mobile` to those in the table `student`. We create a `FOREIGN KEY` constraint and call it `mobile_student_id_fk`. We select `FOREIGN KEY` as `Type`. We then add the column `student` under `Columns` and then click on `Referenced Columns` in Figure 19.2.13. In the `Reference Columns` view, we click on the `Table` bar. In the dialog that pops up in Figure 19.2.14, we



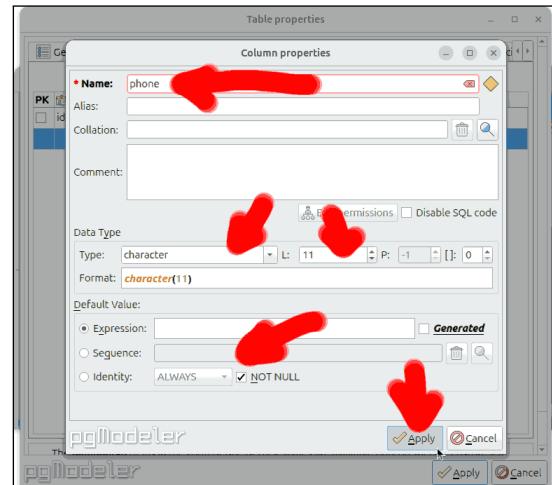
(19.2.5) Multivalued attributes should go into their own table. So we create a new table by right-clicking into our model and then selecting [New] > [Schema object] > [Table].



(19.2.6) We will call the table `mobile` and click on [Columns], because we will now add several columns.



(19.2.7) We will store the mobile phone and a reference to the student in this table. Neither are necessarily unique (since the same person may enroll multiple times). Thus, we need a surrogate key. We create the column `id` of type `integer` and mark it as `Identity` which is (generated) `BY DEFAULT`. This will be roughly equivalent to how we generated IDs in our factory example. We click `Apply`.



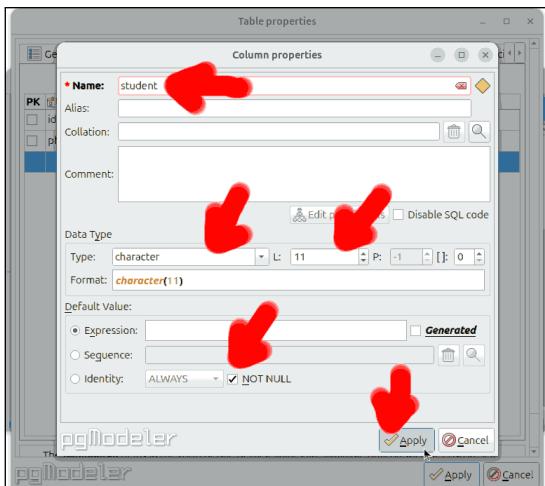
(19.2.8) We create a `phone` number column in the same style we did in the first student DB.

Figure 19.2: Creating a logical model represent students with a composite name attribute and multiple mobile phone numbers (continued).

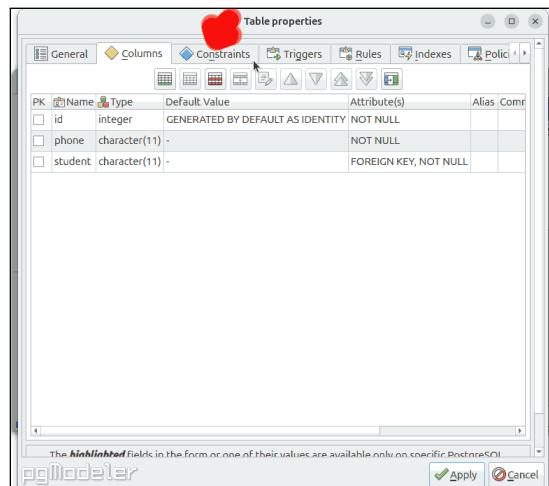
navigate to the `public` schema, open the `Table` list, and select the table `student`. We then click on `Column:`, select `student_id`, and add it via by pressing the `Add Items` button in Figure 19.2.15. The `student_id` column appears in the columns list. We are done and click `Apply` in Figure 19.2.16.

Figure 19.2.17 shows the three constraints that we have created. We click `Apply` to finally create our new table. The model appears in crow's foot notation in Figure 19.2.18. We can see both tables. We see that each row in the table `mobile` must be linked to exactly one row in the table `student`. We see that each row in the table `student` may be linked to arbitrarily many rows in the table `mobile`. This is a bit different from before, as now students are permitted to exist that do not have a mobile phone number associated with themselves. But this is also OK, so let's not fuss about it too much.

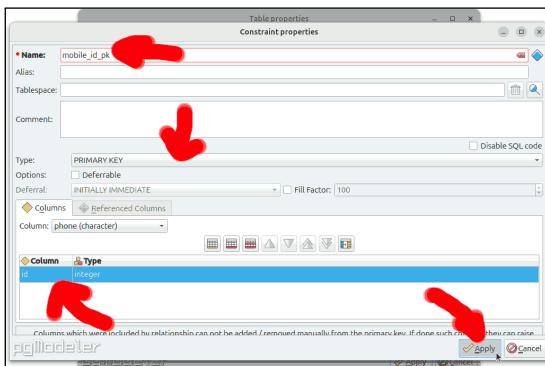
We can again export the model as `SVG` graphic by following the steps given in Figures 19.1.41 to 19.1.44. Figure 19.2.19 shows how nice it looks . . . just compare how much better a vector graphic



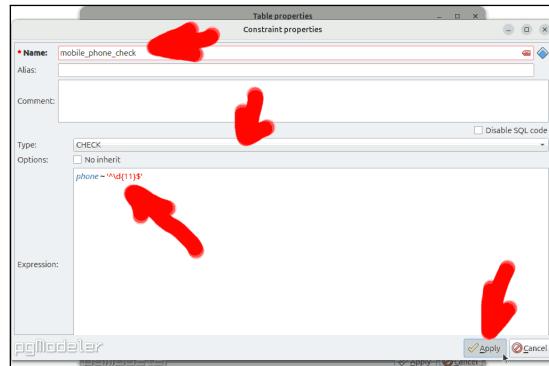
(19.2.9) We create a column `student`, which must have the same datatype as the `student_id` column that holds the primary key of the table `student`.



(19.2.10) We have created three columns and move on to create `Constraints`.



(19.2.11) First we create a primary key constraint for the column `id`.



(19.2.12) Then we re-create the mobile phone number checking constraint.

Figure 19.2: Creating a logical model represent students with a composite name attribute and multiple mobile phone numbers (continued).

looks compared to a pixel graphic (Figure 19.2.18).

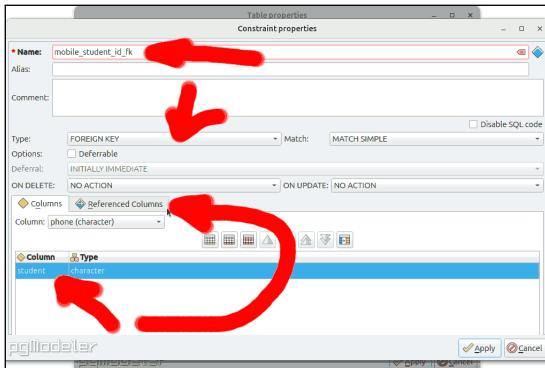
We also notice now that, while our model looks visually pleasing, the diagram lacks some details. For example, we cannot see the datatypes of columns. We cannot see whether they are annotated as `UNIQUE` or `NOT NULL`. This can easily be solved: In Figure 19.2.20, we click on the button at the top-right of the tools bar. This button lets the details appear.

This also makes the tables in the diagram larger, as shown in Figure 19.2.21. We drag them apart with the mouse, which gives us the much clearer layout in Figure 19.2.22. In this layout, we see, for example, that `student_id` is the primary key of the table `student` from the key symbol and the `<pk>` annotation. We see that it is of datatype `CHARACTER(11)`. We also see that `national_id` is a text string of the fixed length 18. and since it is annotated with `<nn>`, it must be `NOT NULL`.

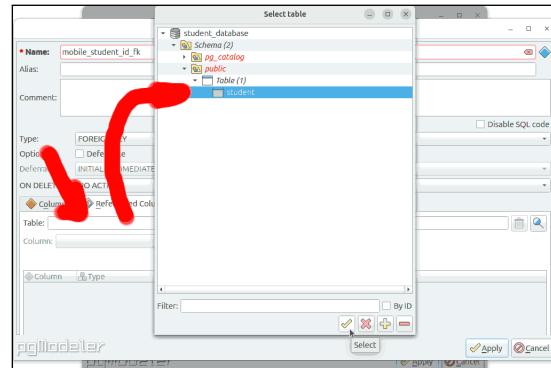
We can export the model again to a `SVG` graphic. This graphic, shown in Figure 19.2.23, now contains all the details as well. It clearly shows the structure and the most important information of our logical model at a glance.

We now export this model to `SQL`, exactly as we did before. This time, we get four scripts. The first one, Listing 19.9, again creates the `student_database` DB. The second one, Listing 19.11, creates the `student` table.

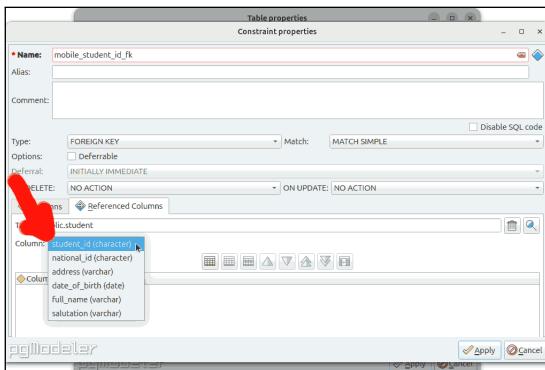
The third script, here given as Listing 19.13, creates the `mobile` table. We notice that the primary key is created as `id integer NOT NULL GENERATED BY DEFAULT AS IDENTITY`. This is almost exactly the same way in which we created the primary key for the `product` table back in Listing 9.1. The only



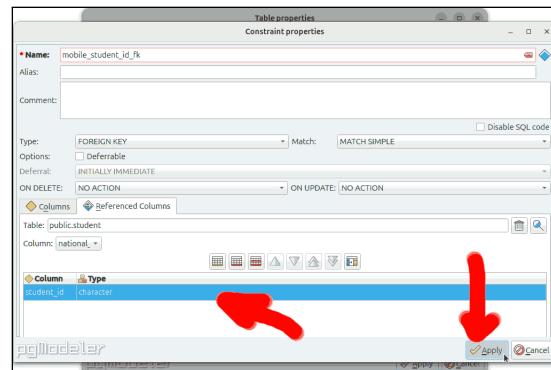
(19.2.13) Now we want to link the rows in this table to those in the table `student`. We create a `FOREIGN KEY` constraint and call it `mobile_student_id_fk`. We select `FOREIGN KEY` as `Type:`. We then add the column `student_id` under `Columns` and then click on `Referenced Columns`.



(19.2.14) There, we click on `Table` and select the table `student` in the dialog that pops up.



(19.2.15) We then click on `Column:`, select `student_id`, and add it via .



(19.2.16) It appears in the columns list. We are done and click `Apply`.

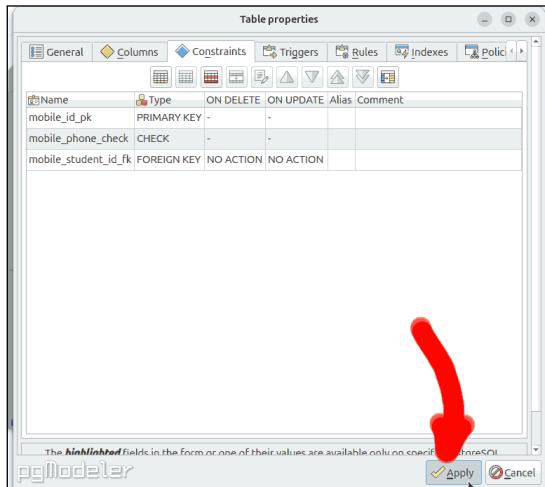
Figure 19.2: Creating a logical model represent students with a composite name attribute and multiple mobile phone numbers (continued).

difference is that PgModeler likes to express the integer type as `integer` and there we used `INT`. Both types are synonymous.

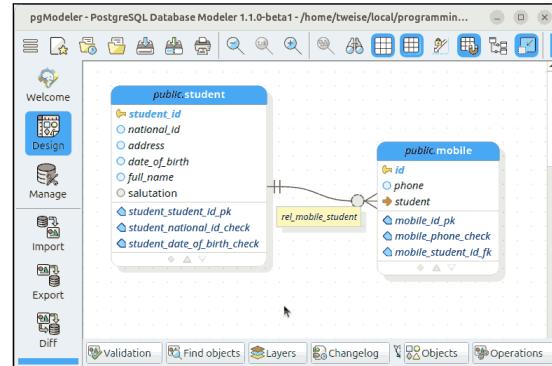
The foreign key constraint is not included in Listing 19.13. Instead, it went into its own script, here reproduced as Listing 19.15. Instead of directly including it when the table is created, the table is later changed (`ALTER TABLE`). The constraint is added via `ADD CONSTRAINT`. Apart from this and some additional behavior specifications that we will ignore here, it looks not much different from the `REFERENCES` statement we used when creating our factory's `demand` table back in Listing 9.17. Well, it looks different, because now it is explicitly defined as constraint instead of being declared inline. But it clearly has the same functionality and if you understand what one notation means, you can also infer what the other means.

We execute all four scripts. Their output in Listings 19.10, 19.14 and 19.16 shows that everything went successfully.

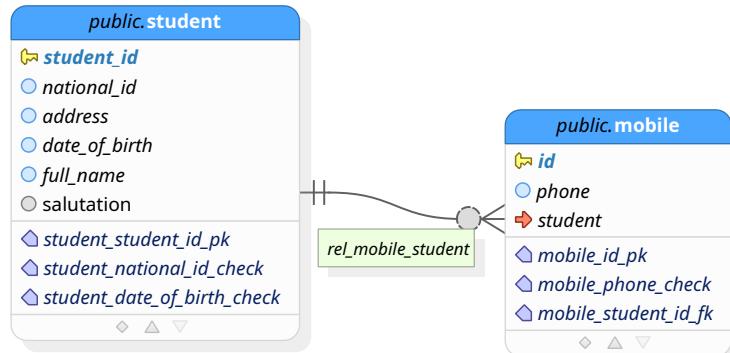
Let us now also use the DB for a bit. For this, we again manually write another SQL script. In Listing 19.17, we first insert some rows into the `student` table. Mr. Bibbo and Mr. Beppo enroll into our university. We also store three mobile phone numbers, two for Mr. Bibbo and one for Mr. Beppo. The rows in the `mobile` table reference the rows in the `student` table via the `student` column referencing the foreign key `student_id`. We do not need to specify values for the `id` column of the `mobile` table, as this one will automatically be filled with sequential values. Then, we `SELECT` the full names of the students associated with each mobile phone via an `INNER JOIN`. The output for this script, given in Listing 19.17, shows that both `INSERT INTO` commands were successful and that `SELECT` gives us the expected result.



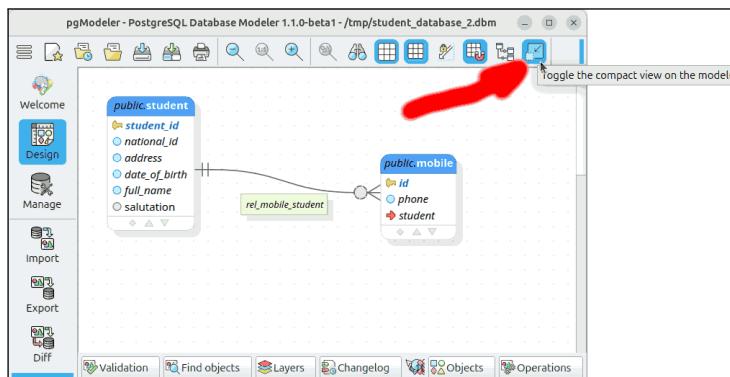
(19.2.17) We have created three columns and three constraints. We click **Apply** to finally create our new table.



(19.2.18) The model appears in crow's foot notation. We can see both tables. We see that each row in the table `mobile` must be linked to exactly one row in the table `student`. We see that each row in the table `student` may be linked to arbitrarily many rows in the table `mobile`.

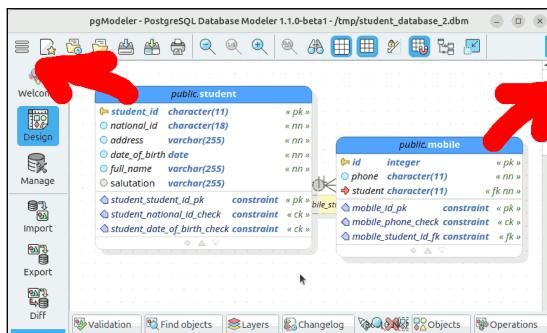


(19.2.19) This is the model if exported as SVG graphic by following the steps in Figures 19.1.41 to 19.1.44.

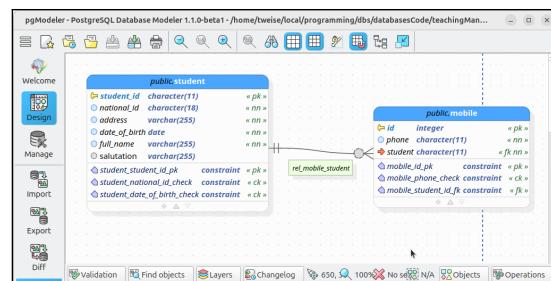


(19.2.20) In order to see more details in the diagram, we press the button at the top-right of the tools bar.

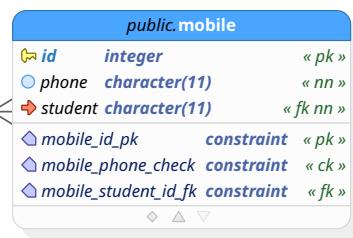
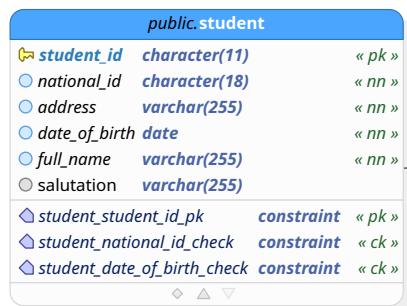
Figure 19.2: Creating a logical model represent students with a composite name attribute and multiple mobile phone numbers (continued).



(19.2.21) More details, such as the column types, appear in the diagram, causing the tables to overlap. We drag them apart with the mouse.



(19.2.22) The new layout looks much clearer.



(19.2.23) We export the model again to a SVG graphic, following the steps in Figures 19.1.41 to 19.1.44. This graphic now contains more details as well.

Figure 19.2: Creating a logical model represent students with a composite name attribute and multiple mobile phone numbers (continued).

Listing 19.9: This auto-generated SQL script creates the DB `student_database`. (stored in file `01_student_database_database_2001.sql`; output in Listing 19.10)

```

1 -- object: student_database | type: DATABASE --
2 -- DROP DATABASE IF EXISTS student_database;
3 CREATE DATABASE student_database;
4 -- ddl-end --

```

Listing 19.10: The stdout of the program `01_student_database_database_2001.sql` given in Listing 19.9.

```

1 $ psql "postgres://postgres:XXX@localhost" -v ON_ERROR_STOP=1 -ebsf 01
2   ↗ _student_database_database_2001.sql
3 CREATE DATABASE
4 # psql 16.12 succeeded with exit code 0.

```

Finally, we delete the DB again in Listing 19.19. With this, we are able to translate single entity types to the relational data model. We are then also able to create the corresponding logical model in a comfortable editor ([PgModeler](#)), that offers us an ERD-like visual syntax. We can export these models to SQL. And we can then push these SQL scripts to the [PostgreSQL](#) server.

Wow, that was a lot. We learned how we can translate an entity type from the conceptual model to a logical schema based on the relational data model. Each entity type becomes one relation, i.e., one table. Its attributes become the columns of the relation. This, however, only holds for simple, single-valued attributes. If the attribute is composite, we break it down to its atomic components and these become columns. Multi-values attributes need to be separated from the entity type. In relational DBs, all datatypes are single-valued. So multi-valued attributes become their own tables. They are then linked over foreign key constraints back to the main table of the entity type. Multiple rows in such a table can relate to one row of the main table, thus realizing the multi-valued-ness.

Listing 19.11: This auto-generated SQL script creates the table `student`. (stored in file `03_public_student_table_5071.sql`; output in [Listing 19.12](#))

```

1  -- object: public.student | type: TABLE --
2  -- DROP TABLE IF EXISTS public.student CASCADE;
3  CREATE TABLE public.student (
4      student_id character(11) NOT NULL,
5      national_id character(18) NOT NULL,
6      address varchar(255) NOT NULL,
7      date_of_birth date NOT NULL,
8      full_name varchar(255) NOT NULL,
9      salutation varchar(255),
10     CONSTRAINT student_student_id_pk PRIMARY KEY (student_id),
11     CONSTRAINT student_national_id_check CHECK (national_id ~ '^\d{6}((19|'
12         ↪ (20))\d{9}[0-9X]$'),
13     CONSTRAINT student_date_of_birth_check CHECK ((date_of_birth > '
14         ↪ 1900-01-01') AND (date_of_birth < '2100-01-01'))
15 );
16 -- ddl-end --
15 ALTER TABLE public.student OWNER TO postgres;
16 -- ddl-end --

```

Listing 19.12: The stdout of the program `03_public_student_table_5071.sql` given in [Listing 19.11](#).

```

1 $ psql "postgres://postgres:XXX@localhost/student_database" -v
2   ↪ ON_ERROR_STOP=1 -ebf 03_public_student_table_5071.sql
3 CREATE TABLE
4 ALTER TABLE
4 # psql 16.12 succeeded with exit code 0.

```

Luckily, we learned the new tool `PgModeler`. This tool allows us to draw logical models for the PostgreSQL DBMS in a way quite similar to what we are used with `yEd`. The models we generated by `yEd` were technology-independent conceptual models. The logical models created with `PgModeler` are bound to `SQL` and the PostgreSQL DBMS. We can export them as `SVG` graphics, exactly as we did with models we created in `yEd`. However, we can also export them to `SQL` directly. This is actually pretty awesome. We can now use a comfortable `GUI` to help us to build the DB schema. And then we can directly feed it into PostgreSQL.

Listing 19.13: This auto-generated SQL script creates the table `mobile`. (stored in file `04_public_mobile_table_5081.sql`; output in Listing 19.14)

```

1 -- object: public.mobile | type: TABLE --
2 -- DROP TABLE IF EXISTS public.mobile CASCADE;
3 CREATE TABLE public.mobile (
4     id integer NOT NULL GENERATED BY DEFAULT AS IDENTITY ,
5     phone character(11) NOT NULL ,
6     student character(11) NOT NULL ,
7     CONSTRAINT mobile_id_pk PRIMARY KEY (id),
8     CONSTRAINT mobile_phone_check CHECK (phone ~ '^\d{11}$')
9 );
10 -- ddl-end --
11 ALTER TABLE public.mobile OWNER TO postgres;
12 -- ddl-end --

```

Listing 19.14: The stdout of the program `04_public_mobile_table_5081.sql` given in Listing 19.13.

```

1 $ psql "postgres://postgres:XXX@localhost/student_database" -v
   ↪ ON_ERROR_STOP=1 -ebf 04_public_mobile_table_5081.sql
2 CREATE TABLE
3 ALTER TABLE
4 # psql 16.12 succeeded with exit code 0.

```

Listing 19.15: This auto-generated SQL script adds the foreign key constraint to the table `mobile`. (stored in file `05_public_mobile_mobile_student_id_fk_constraint_5087.sql`; output in Listing 19.16)

```

1 -- object: mobile_student_id_fk | type: CONSTRAINT --
2 -- ALTER TABLE public.mobile DROP CONSTRAINT IF EXISTS mobile_student_id_fk
   ↪ CASCADE;
3 ALTER TABLE public.mobile ADD CONSTRAINT mobile_student_id_fk FOREIGN KEY (
   ↪ student)
4 REFERENCES public.student (student_id) MATCH SIMPLE
5 ON DELETE NO ACTION ON UPDATE NO ACTION;
6 -- ddl-end --

```

Listing 19.16: The stdout of the program `05_public_mobile_mobile_student_id_fk_constraint_5087.sql` given in Listing 19.15.

```

1 $ psql "postgres://postgres:XXX@localhost/student_database" -v
   ↪ ON_ERROR_STOP=1 -ebf 05
   ↪ _public_mobile_mobile_student_id_fk_constraint_5087.sql
2 ALTER TABLE
3 # psql 16.12 succeeded with exit code 0.

```

Listing 19.17: We now insert some data into the `student` and `mobile` tables and use a `JOIN` to select data from both. (stored in file `insert_and_select.sql`; output in Listing 19.18)

```

1  /** Insert data into the student database and join the two tables. */
2
3  -- Insert several student records.
4  INSERT INTO student (student_id, national_id, full_name, salutation,
5                      address, date_of_birth) VALUES
6      ('1234567890', '123456199501021234', 'Bibbo', 'The Bib-Man',
7       'Hefei, China', '1995-01-02'),
8      ('1234567891', '123456200508071234', 'Bebbo', 'Bebbo Machine',
9       'Chemnitz, Germany', '2005-08-07');
10
11 -- Insert several mobile phone numbers
12 INSERT INTO mobile (phone, student) VALUES
13     ('111111111111', '1234567890'), ('222222222222', '1234567891'),
14     ('333333333333', '1234567890');
15
16 -- Print the mobile phone numbers of the students.
17 SELECT student.full_name, mobile.phone, mobile.id FROM mobile
18   INNER JOIN student ON mobile.student = student.student_id;

```

Listing 19.18: The stdout of the program `insert_and_select.sql` given in Listing 19.17.

```

1  $ psql "postgres://postgres:XXX@localhost/student_database" -v
2    ↪ ON_ERROR_STOP=1 -e bf insert_and_select.sql
3
4  INSERT 0 2
5  INSERT 0 3
6  full_name |   phone   | id
7  -----+-----+-----
8  Bibbo    | 111111111111 | 1
9  Bebbo    | 222222222222 | 2
10 Bibbo    | 333333333333 | 3
11 (3 rows)
12
13 # psql 16.12 succeeded with exit code 0.

```

Listing 19.19: We delete the DB again, so we can start with a clean slate in the next experiment. (stored in file `cleanup.sql`; output in Listing 19.20)

```

1  /* Cleanup after the example: Delete the student database. */
2
3  DROP DATABASE IF EXISTS student_database;

```

Listing 19.20: The stdout of the program `cleanup.sql` given in Listing 19.19.

```

1  $ psql "postgres://postgres:XXX@localhost" -v ON_ERROR_STOP=1 -e bf cleanup.
2    ↪ sql
3  DROP DATABASE
4  # psql 16.12 succeeded with exit code 0.

```

19.2.2 Mapping Conceptual Relationships to Logical Models

In Section 18.5, we discussed ten different types of binary relationships between two entity types that can occur in an [ERD](#) created during conceptual modeling. Back then, we worked our way through these types and tried to find examples in existing sources.

We can view the binary relationship types as requirements that are imposed on the elements of two entity types. In a $C \rightarrow\!\!\!-\!\!\!-\! D$ relationship, for example, it is required that any entity of type C *must* be related to exactly one entity of type D . Relationships are bi-directional, i.e., if an entity of type C is related to an entity of type D , then that very same entity of type D is obviously also related to the entity of type C . Vice versa, the $C \rightarrow\!\!\!-\!\!\!-\! D$ pattern also permits an entity of type D to either be linked to one or no entity of type C .

Definition 19.6: Referential Integrity

A [DB](#) where the relationship constraints between entities are correctly maintained has the property of *referential integrity*.

If we map a conceptual models, say given as [ERD](#), to the relational data model, we must also map the relationship patterns. This means essential to translate crow's foot notation to [SQL](#). SQL offers us four major tools to implement relationship constraints:

- the primary key constraint [PRIMARY KEY](#),
- the foreign key constraint [REFERENCES](#),
- the [NOT NULL](#) constraint that prevents an attribute to ever be undefined ([NULL](#)),
- the [UNIQUE](#) constraint that prevents a value from occurring twice in a column.

In most courses that I have on DB, the relationship types are discussed in the conceptual levels. Then, there is only a very approximate and incomplete explanation on how these can actually be realized. Maybe something along the lines of "For this type, you use two tables...". It is usually discussed how the multiplicity and modality can be actually enforced, especially for more complex constellations like $S \triangleright\!\!\!-\!\!\!-\! T$.

Well, you are lucky. We will now set out to find how *each* of the ten binary conceptual relationship types that we discussed can be implemented in SQL. We will do this in plain SQL and not in the [PgModeler](#), because we start from the visual representation of the relationships and want to transform them to SQL. Using [PgModeler](#), we would practically do the same, just in a convenient [GUI](#). [PgModeler](#) is also more suitable for managing larger models, whereas we will slash and hammer our way through several small models with two entity types each.

Please also consider this as an exercise in SQL. This is not so much about whether all of these relationship types do occur in practice. It is also not about memorizing the different approaches how they can be implemented. It is mainly about getting some feeling and understanding how the utilities that SQL offers us, mainly [NOT NULL](#), [REFERENCES](#), [UNIQUE](#), and [PRIMARY KEY](#) constraints [100] together with [INNER JOIN](#) queries [233], can be used to enforce referential integrity between tables. And also, for some relationship types ... it is even fun to figure out how they can be implemented.

First Time Readers and Novices: It is totally OK to skip over a few of the following subsections. Once you understand the basic concepts, it may not be necessarily to reproduce all ten scenarios. You can revisit the section later if you are looking for a particular setup.

Of course, keeping with our practical "This is what it looks like when we execute it on the [PostgreSQL server](#)." attitude, we spin up a DB to really see some of the concepts in action in Listing 19.21.

Listing 19.21: We spin up a DB for running our example SQL codes when mapping conceptual relationship between entity types to tables in SQL on the PostgreSQL server. (stored in file `init.sql`; output in Listing 19.22)

```

1 /* Initialize the database for our examples */
2
3 -- Create the database.
4 CREATE DATABASE relationships;
```

Listing 19.22: The stdout of the program `init.sql` given in Listing 19.21.

```

1 $ psql "postgres://postgres:XXX@localhost" -v ON_ERROR_STOP=1 -e bf init.sql
2 CREATE DATABASE
3 # psql 16.12 succeeded with exit code 0.
```

19.2.2.1 A +o---o B

We have the two entity types A and B. Each entity of type A is connected to zero or one entity of type B. Each entity of type B is connected to zero or one entity of type A.

We saw examples of this relationship pattern back in Section 18.5 (The Cardinality of Relationships): Let's we are a pizzeria that allows customers to order pizzas online. We also hand out discount vouchers that allow can reduce the cost of an online order by 20%. Each voucher can be redeemed for one such online order or it may also not be redeemed at all. For each online order, you may use one voucher to reduce the cost, or you may not use any voucher at all.

Another example could be integrated into our teaching management platform. As you will remember, we used the entity type *Person* as central model component. If we wanted, we could add information about the current marital status of all people in form of, fittingly, relationships. Each person can be married to zero or one (other) person. These two examples are illustrated in Figure 19.3.

There are two possible ways to implement this relationship pattern in a relational DBMS:

1. We can use a three-table solution, where each entity type gets one table and the relationship between the entity types A and B is managed in a third table. This solution makes sense if we have only few pairs of A and B entities that are related.
2. We can use a two-table solution, where we manage the relationship with an additional column in one of the tables. This is the better solution if we have many related pairs of A and B entities.

Regardless which solution we pick, we need at least the following two tables. First, table `a` is used for the entity type A. As primary key, we here use a surrogate primary key `aid` which is an automatically generated integer sequence. Let us assume that the entity type A also has a feature `x`, for illustration purposes, a string composed of three characters. Second, table `b` for the entity type B. We here use a surrogate primary key `bid`, again an automatically generated integer sequence. Assume that there also is an attribute `y`, which is a string of two characters. (In all the remaining examples in this section, we will use similar table patterns).

We start with the three-table approach in Listing 19.23, as suggested in, e.g., [396]. There, we have a third table, `relate_a_and_b`, relating the entities of type A to those of type B. This table has two columns, the first one, `fkaid`, holding the primary key of the A entities as foreign key and the second one, `fkbid`, holding the primary key of the B entities as foreign key. Since we only store the pairs that exist, both columns have the `NOT NULL` constraint. Both also have `REFERENCES` constraints to their

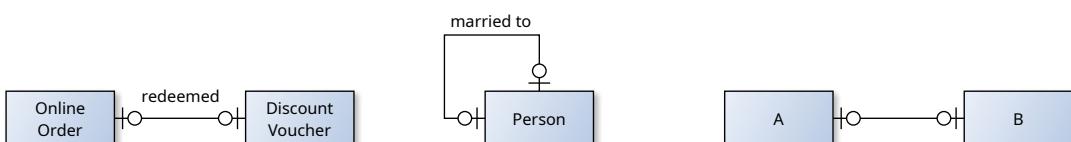


Figure 19.3: Examples of the A +o---o B relationship pattern from back in Section 18.5 (The Cardinality of Relationships).

Listing 19.23: The three-table realization of an A $\rightarrow\!\!\!\rightarrow$ B conceptual relationship. (stored in file `AB_1_tables.sql`; output in Listing 19.24)

```
1  /* Create the tables for an A-|o----o|-B relationship. */
2
3  -- Table A: Each row in A is related to zero or one row in B.
4  CREATE TABLE a (
5      aid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
6      x    CHAR(3)    -- example of other attributes
7 );
8
9  -- Table B: Each row in B is related to zero or one row in A.
10 CREATE TABLE b (
11     bid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
12     y    CHAR(2)    -- example of other attributes
13 );
14
15 -- The table for managing the relationship between A and B.
16 CREATE TABLE relate_a_and_b (
17     fkaid INT NOT NULL UNIQUE PRIMARY KEY REFERENCES a (aid),
18     fkbid INT NOT NULL UNIQUE                      REFERENCES b (bid)
19 );
```

Listing 19.24: The stdout of the program `AB_1_tables.sql` given in Listing 19.23.

```
1 $ psql "postgres://postgres:XXX@localhost/relationships" -v ON_ERROR_STOP=1
2   ↳ -ebl AB_1_tables.sql
3 CREATE TABLE
4 CREATE TABLE
5 CREATE TABLE
6 # psql 16.12 succeeded with exit code 0.
```

respective foreign keys. Since every entity of type A can only be linked to at most one entity of type B, each value of the primary key of table `a` must only appear in column `fkaid` of table `relate_a_and_b` *at most once*. The same holds for the entities of type B vice versa, which also means that each value (of `bid`) can occur at most once in column `fkbid` *at most once*. In other words: Since on both relationship ends there can only be one entity, both columns also have `UNIQUE` constraints. Either of them may be used as `PRIMARY KEY`.

We would probably choose the key belonging from the most common direction in which we access the table. If we most likely look for fitting entities of type B coming from a row representing an entity of type A (as is the case in Listing 19.25), then we would probably use `fkaid` as `PRIMARY KEY`. If it was the other way around, we would pick `fkbid`.

In Listing 19.25, we fill some data into the tables. Since both relationship ends are optional, we can begin by inserting data into tables `a` and `b` without specifying any references to the other table. We do not need to specify the primary keys, since they are automatically generated, and, as said, we can omit the foreign keys, because they are allowed to be `NULL`. So we only need to provide values for the attributes `x` and `y`. After this, we have filled both tables with some data. We can then establish relationships between the rows of tables `a` and `b` by adding entries to `relate_a_and_b` with the primary keys of the related rows. `INSERT INTO relate_a_and_b (fkaid, fkbid)VALUES (2, 3);`, for example, would relate the row in table `a` with `aid = 2` to the row in table `b` with `bid = 3`. The contents of the tables after executing Listing 19.25 is illustrated in Figure 19.4.

Now we want to query some information about an entity A, using `SELECT`, but also need information about the potentially related B entity. We use an `INNER JOIN` coming from table `a` on the third table `relate_a_and_b` based on the primary key `aid` of table `a`. We then need another `INNER JOIN` with the table for B on its primary key `bid`, as shown in Listing 19.25. This way, we can reconstruct the related data in two steps.

Notice that it is impossible to have any row of table `a` that is related to more than one row in table `b`, as demonstrated in Listing 19.27. The `UNIQUE` constraint on the column `fkaid` of `relate_a_and_b`

Listing 19.25: Inserting into and selecting data from the three-table realization of an A to-Of B conceptual relationship given in Listing 19.23. (stored in file `AB_1_insert_and_select.sql`; output in Listing 19.26)

```

1  /* Inserting data into the tables for the A-to-Of-B relationship. */
2
3  -- Insert some rows into the table for entity type A.
4  INSERT INTO a (x) VALUES ('123'), ('456'), ('789'), ('101');
5
6  -- Insert some rows into the table for entity type B.
7  INSERT INTO b (y) VALUES ('AB'), ('CD'), ('EF'), ('GH');
8
9  -- Create the relationships between the A and B rows.
10 INSERT INTO relate_a_and_b (fkaid, fkbid) VALUES (1, 1), (2, 3), (3, 4);
11
12 -- Combine the rows from A and B. This needs two INNER JOINS.
13 SELECT aid, x, bid, y FROM relate_a_and_b
14   INNER JOIN a ON a.aid = relate_a_and_b.fkaid
15   INNER JOIN b ON b.bid = relate_a_and_b.fkbid;

```

Listing 19.26: The stdout of the program `AB_1_insert_and_select.sql` given in Listing 19.25.

```

1  $ psql "postgres://postgres:XXX@localhost/relationships" -v ON_ERROR_STOP=1
2    ↪ -eef AB_1_insert_and_select.sql
3
4  INSERT 0 4
5  INSERT 0 4
6  INSERT 0 3
7  aid | x   | bid | y
8  ----+---+---+---+
9  1   | 123 | 1   | AB
10  2  | 456 | 3   | EF
11  3  | 789 | 4   | GH
12  (3 rows)
13
14 # psql 16.12 succeeded with exit code 0.

```

Table a		Table b		Table relate_a_and_b	
aid	x	bid	y	fkaid	fkbid
1	"123"	1	"AB"	1	1
2	"456"	2	"CD"	2	3
3	"789"	3	"EF"	3	4
4	"101"	4	"GH"		

Figure 19.4: The contents of the the tables in the three-table implementation of the A to-Of B conceptual relationship after executing Listing 19.25.

prevents this. Vice versa, the `UNIQUE` constraint on column `fkbid` in `relate_a_and_b` prevents that any row in table `b` is related to more than one row in table `a`. This is demonstrated in Listing 19.29.

However, there are two problems with this three-table-method: First, we need two `INNER JOIN` statements to combine the data from the entities A and B. Second, this approach makes sense only if comparatively few pairs of related A and B entities exist. Performance and space-wise, in the worst case, all entities of type A are related to an entity of type B or vice versa. In other words, our third table can be about as big as the smaller one of the two other tables. If the tables are big, then the query may not be fast. Also, if all entities of type A were related to an entity of type B, then we could just as well reference these B entities directly from the table for entity type A. Actually, in that case, using the third table would just be a waste of space and query time.

Thus, if many or most A entities are related to B entities, then instead of using a third table, we could alternatively add a column `fkbid` to the table for A and store the primary keys of the related B entities in that column. We delete the tables we just created in Listing 19.31 to explore the two-table

Listing 19.27: The schema illustrated in Listing 19.23 prevents entities of type A to be related to more than one entity of type B. (stored in file AB_1_insert_error_1.sql; output in Listing 19.28)

```

1  /* Insert a wrong row into tables for A-|o----o|-B relationship. */
2
3  -- Create an error in the relationships between the A and B rows.
4  -- This fails because an A entry is already assigned to an B entry.
5  -- The A entity with ID 1 is already related to B entity with ID 1.
6  INSERT INTO relate_a_and_b (fkaid, fkbid) VALUES (1, 2);

```

Listing 19.28: The stdout of the program AB_1_insert_error_1.sql given in Listing 19.27.

```

1  $ psql "postgres://postgres:XXX@localhost/relationships" -v ON_ERROR_STOP=1
2    ↪ -e bf AB_1_insert_error_1.sql
3  psql:conceptualToRelational/AB_1_insert_error_1.sql:6: ERROR:  duplicate
4    ↪ key value violates unique constraint "relate_a_and_b_pkey"
5  DETAIL:  Key (fkaid)=(1) already exists.
6  psql:conceptualToRelational/AB_1_insert_error_1.sql:6: STATEMENT:  /*
7    ↪ Insert a wrong row into tables for A-|o----o|-B relationship. */
8  -- Create an error in the relationships between the A and B rows.
9  -- This fails because an A entry is already assigned to an B entry.
10 -- The A entity with ID 1 is already related to B entity with ID 1.
11 INSERT INTO relate_a_and_b (fkaid, fkbid) VALUES (1, 2);
12 # psql 16.12 failed with exit code 3.

```

solution in Listing 19.33. The new column `fkbid` added to the table `a`. It can be allowed to be `NULL`, because not all entities of type A need to be related to an entity of type B. It must be marked as `UNIQUE`, though, because no entity of type B can be related to more than one entity of type A. This constraint only applies to `fkbid` values that are *not* `NULL`.

It also needs a `REFERENCES` constraint. In Listing 19.33, we add this constraint later via `ALTER TABLE`, very much like PgModeler does it (see back in Listing 19.15). Otherwise, we would need to create table `b` before table `a`. That is totally OK, but if we had many tables, the required order of table creation could make our SQL code harder to read. Also, the probability of making errors that are hard to figure out would be higher. It sometimes is just easier to first create the tables and then add the constraints via . I guess this is why PgModeler does it like this, too.

We could also do this vice versa, i.e., add an column `fkaid` to table `b` instead. Either way, this removes the need for a third table.

In Listing 19.35, we insert the data into the two tables `a` and `b` exactly as before. The relationships are now no longer inserted into a third table. Instead, we use `UPDATE` statements [461], as illustrated in Listing 19.35. `UPDATE a SET fkbid = 3 WHERE aid = 2;`, for example, relates the row in table `a` with `aid = 2` to the row in table `b` with `bid = 3`. The contents of the tables after creating the data is illustrated in Figure 19.5. We can now recombine the data from the two tables using a single `INNER JOIN`.

Relating entities of type A to multiple entities of type B is impossible, because the column `fkbid` in table `a` can only have one value. Relating entities of type B to multiple entities of type A is impossible, because of the `UNIQUE` constraint imposed on it the column `fkbid` in table `a`. A failed attempt to do so anyway is shown in Listing 19.37.

Of course, the table for entity type A now needs more space. If only few entities of type A are related to entities of type B, maybe the first solution, based on three tables, is maybe better. I think most often, the second solution, the two-table approach, is the way to go. Still, it depends on how many of the A and B entities are related, from which “side” we most likely navigate the relationship, and how big the tables are.

Listing 19.29: The schema illustrated in Listing 19.23 prevents entities of type B to be related to more than one entity of type A. (stored in file AB_1_insert_error_2.sql; output in Listing 19.30)

```

1  /* Insert a wrong row into tables for A-|o----o|-B relationship. */
2
3  -- Create an error in the relationships between the A and B rows.
4  -- This fails because a B entry is already assigned to an A entry.
5  -- The B entity with ID 3 is already related to A entity with ID 1.
6  INSERT INTO relate_a_and_b (fkaid, fkbid) VALUES (4, 3);

```

Listing 19.30: The stdout of the program AB_1_insert_error_2.sql given in Listing 19.29.

```

1  $ psql "postgres://postgres:XXX@localhost/relationships" -v ON_ERROR_STOP=1
   ↵ -ebf AB_1_insert_error_2.sql
2  psql:conceptualToRelational/AB_1_insert_error_2.sql:6: ERROR:  duplicate
   ↵ key value violates unique constraint "relate_a_and_b_fkbid_key"
3  DETAIL:  Key (fkbid)=(3) already exists.
4  psql:conceptualToRelational/AB_1_insert_error_2.sql:6: STATEMENT:  /*
   ↵ Insert a wrong row into tables for A-|o----o|-B relationship. */
5  -- Create an error in the relationships between the A and B rows.
6  -- This fails because a B entry is already assigned to an A entry.
7  -- The B entity with ID 3 is already related to A entity with ID 1.
8  INSERT INTO relate_a_and_b (fkaid, fkbid) VALUES (4, 3);
9  # psql 16.12 failed with exit code 3.

```

Listing 19.31: Deleting the three tables again, because we want to try another realization of the A $\rightarrow\!\!\rightarrow$ B conceptual relationship. (stored in file AB_cleanup.sql; output in Listing 19.32)

```

1  /* Drop the tables for the A-|o----o|-B relationship. */
2
3  DROP TABLE IF EXISTS relate_a_and_b;
4  DROP TABLE IF EXISTS a;
5  DROP TABLE IF EXISTS b;

```

Listing 19.32: The stdout of the program AB_cleanup.sql given in Listing 19.31.

```

1  $ psql "postgres://postgres:XXX@localhost/relationships" -v ON_ERROR_STOP=1
   ↵ -ebf AB_cleanup.sql
2  DROP TABLE
3  DROP TABLE
4  DROP TABLE
5  # psql 16.12 succeeded with exit code 0.

```

Listing 19.33: The two-table realization of an A $\rightarrow\!\!\!\rightarrow$ B conceptual relationship. (stored in file `AB_2_tables.sql`; output in Listing 19.34)

```

1  /* Create the tables for an A  $\rightarrow\!\!\!\rightarrow$  B relationship. */
2
3  -- Table A: Each row in A is related to zero or one row in B.
4  CREATE TABLE a (
5      aid      INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
6      fkbid   INT UNIQUE,    -- foreign key to B, see later <-- can be NULL.
7      x       CHAR(3)        -- example of other attributes
8  );
9
10 -- Table B: Each row in B is related to zero or one row in A.
11 CREATE TABLE b (
12     bid      INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
13     y       CHAR(2)        -- example of other attributes
14 );
15
16 -- To table A, we add the foreign key reference constraint to table B.
17 ALTER TABLE a ADD CONSTRAINT a_fkbid_fk FOREIGN KEY (fkbid)
18     REFERENCES b (bid);

```

Listing 19.34: The stdout of the program `AB_2_tables.sql` given in Listing 19.33.

```

1  $ psql "postgres://postgres:XXX@localhost/relationships" -v ON_ERROR_STOP=1
2      ↪ -e bf AB_2_tables.sql
3  CREATE TABLE
4  CREATE TABLE
5  ALTER TABLE
5  # psql 16.12 succeeded with exit code 0.

```

Listing 19.35: Inserting into and selecting data from the two-table realization of an A $\rightarrow\!\!\!\rightarrow$ B conceptual relationship given in Listing 19.33. (stored in file AB_2_insert_and_select.sql; output in Listing 19.36)

```

1  /* Inserting data into the tables for the A  $\rightarrow\!\!\!\rightarrow$  B relationship. */
2
3  -- Insert some rows into the table for entity type A.
4  INSERT INTO a (x) VALUES ('123'), ('456'), ('789'), ('101');
5
6  -- Insert some rows into the table for entity type B.
7  INSERT INTO b (y) VALUES ('AB'), ('CD'), ('EF'), ('GH');
8
9  -- Create the relationships between the A and B rows.
10 UPDATE a SET fkbid = 1 WHERE aid = 1;
11 UPDATE a SET fkbid = 3 WHERE aid = 2;
12 UPDATE a SET fkbid = 4 WHERE aid = 3;
13
14 -- Combine the rows from A and B. Only one INNER JOIN is needed.
15 SELECT aid, x, bid, y FROM a INNER JOIN b ON a.fkbid = b.bid;
```

Listing 19.36: The stdout of the program AB_2_insert_and_select.sql given in Listing 19.35.

```

1  $ psql "postgres://postgres:XXX@localhost/relationships" -v ON_ERROR_STOP=1
2      ↪ -eef AB_2_insert_and_select.sql
3  INSERT 0 4
4  INSERT 0 4
5  UPDATE 1
6  UPDATE 1
7  UPDATE 1
8  aid | x   | bid | y
9  ----+---+-----+---
10 1  | 123 | 1   | AB
11 2  | 456 | 3   | EF
12 3  | 789 | 4   | GH
13 (3 rows)
14 # psql 16.12 succeeded with exit code 0.
```

Table a			Table b	
aid	fkbid	x	bid	y
4	NULL	"101"	1	"AB"
1	1	"123"	2	"CD"
2	3	"456"	3	"EF"
3	4	"789"	4	"GH"

Figure 19.5: The contents of the the tables in the two-table implementation of the A $\rightarrow\!\!\!\rightarrow$ B conceptual relationship after executing Listing 19.35.

Listing 19.37: The schema illustrated in [Listing 19.33](#) prevents entities of type B to be related to more than one entity of type A via constraints. The entities of type A can only be associated with one entity of type B because of the table structure. (stored in file `AB_2_insert_error_2.sql`; output in [Listing 19.38](#))

```

1  /* Insert a wrong row into tables for A|o----o|B relationship. */
2
3  -- Create an error in the relationships between the A and B rows.
4  -- This fails because a B entry is already assigned to an A entry.
5  -- The B entity with ID 3 is already related to A entity with ID 2.
6  UPDATE a SET fkbid = 3 where aid = 4;

```

Listing 19.38: The stdout of the program `AB_2_insert_error_2.sql` given in [Listing 19.37](#).

```

1  $ psql "postgres://postgres:XXX@localhost/relationships" -v ON_ERROR_STOP=1
2    ↪ -eef AB_2_insert_error_2.sql
3  psql:conceptualToRelational/AB_2_insert_error_2.sql:6: ERROR:  duplicate
4    ↪ key value violates unique constraint "a_fkbid_key"
3  DETAIL:  Key (fkbid)=(3) already exists.
4  psql:conceptualToRelational/AB_2_insert_error_2.sql:6: STATEMENT:  /*
5    ↪ Insert a wrong row into tables for A|o----o|B relationship. */
5  -- Create an error in the relationships between the A and B rows.
6  -- This fails because a B entry is already assigned to an A entry.
7  -- The B entity with ID 3 is already related to A entity with ID 2.
8  UPDATE a SET fkbid = 3 where aid = 4;
9  # psql 16.12 failed with exit code 3.

```

19.2.2.2 C + O → CO₂

We have the two entity types C and D. Each entity of type C must be connected to exactly one entity of type D. Each entity of type D is connected to zero or one entity of type C.

We encountered this pattern when making a conceptual model of the relationship of persons, faculty, and students back in Figure 18.15. In our conceptual model, each entity of type *Person* could be a *Faculty* member or not. Each entity of type *Faculty* must be associated with one and exactly one record of type *Person*, as sketched in Figure 19.6.

A three-table solution makes no sense here. We *know* that every entity of type C *must* be connected to one entity of type D. The solution for implementing this relationship pattern in a **relational database** is therefore similar to the two-table variant in the previous section. We need a table `c` for the entities of type C. And we need a table `d` for the entities of type D. Both tables are also structured as in the previous section, including surrogate primary keys `cid` and `did` as well as attributes `x` and `y`, respectively.

Since each entity of type C must be connected to exactly one entity of type D, we add one

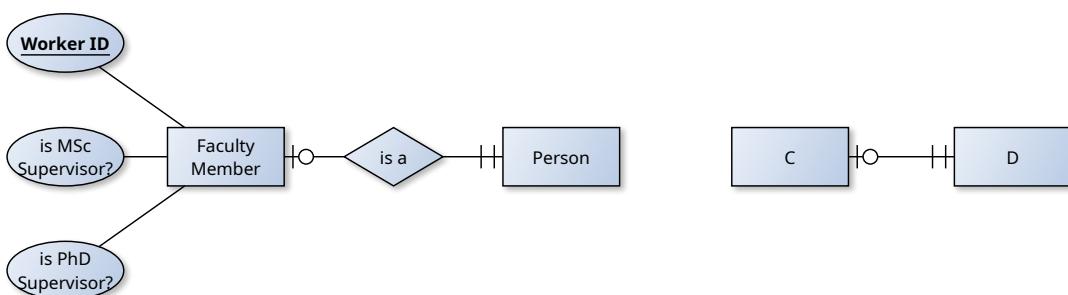


Figure 19.6: We encountered the C \rightarrow D relationship pattern in Figure 18.15.

Listing 19.39: The three-table realization of a C $\rightarrow\!-\!\!-\! D$ conceptual relationship. (stored in file `CD_tables.sql`; output in [Listing 19.40](#))

```

1  /* Create the tables for a C-----| | -D relationship. */
2
3  -- Table C: Each row in C is related to exactly one row in D.
4  CREATE TABLE c (
5      cid    INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
6      fkdid INT NOT NULL UNIQUE,   -- the foreign key to D, see later
7      x     CHAR(3)   -- example of other attributes
8 );
9
10 -- Table D: Each row in D is related to zero or one row in C.
11 CREATE TABLE d (
12     did    INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
13     y     CHAR(2)   -- example of other attributes
14 );
15
16 -- To table C, we add the foreign key reference constraint to table D.
17 ALTER TABLE c ADD CONSTRAINT c_fkdid_fk FOREIGN KEY (fkdid)
18     REFERENCES d (did);

```

Listing 19.40: The std::out of the program `CD tables.sql` given in Listing 19.39.

```
1 $ psql "postgres://postgres:XXX@localhost/relationships" -v ON_ERROR_STOP=1
   ↵ -ebs CD_tables.sql
2 CREATE TABLE
3 CREATE TABLE
4 ALTER TABLE
5 # psql 16.12 succeeded with exit code 0.
```

Listing 19.41: Inserting into and selecting data from the three-table realization of a C $\rightarrow\!\!\!\rightarrow$ D conceptual relationship given in Listing 19.39. (stored in file `CD_insert_and_select.sql`; output in Listing 19.42)

```

1  /* Inserting data into the tables for the C-----||-D relationship. */
2
3  -- Insert some rows into the table for entity type D.
4  -- We first must create the D elements, because the C rows cannot
5  -- exist without referencing one row in D each.
6  INSERT INTO d (y) VALUES ('AB'), ('CD'), ('EF'), ('GH'), ('IJ');
7
8  -- Insert some rows into the table for entity type C.
9  INSERT INTO c (fkdid, x) VALUES (1, '123'), (3, '456'), (4, '789'),
10   (2, '101');
11
12 -- Combine the rows from C and D.
13 SELECT cid, x, did, y FROM c INNER JOIN d ON c.fkdid = d.did;

```

Listing 19.42: The `stdout` of the program `CD_insert_and_select.sql` given in Listing 19.41.

```

1  $ psql "postgres://postgres:XXX@localhost/relationships" -v ON_ERROR_STOP=1
2    ↪ -ebs CD_insert_and_select.sql
3  INSERT 0 5
4  INSERT 0 4
5  cid | x   | did | y
6  ----+---+---+---+
7  1   | 123 | 1   | AB
8  4   | 101 | 2   | CD
9  2   | 456 | 3   | EF
10  3  | 789 | 4   | GH
11  (4 rows)
12 # psql 16.12 succeeded with exit code 0.

```

Table <code>c</code>			Table <code>d</code>	
<code>cid</code>	<code>fkdid</code>	<code>x</code>	<code>did</code>	<code>y</code>
1	1	"123"	1	"AB"
2	3	"456"	2	"CD"
3	4	"789"	3	"EF"
4	2	"101"	4	"GH"
			5	"IJ"

Figure 19.7: The contents of the the tables in the implementation of the C $\rightarrow\!\!\!\rightarrow$ D conceptual relationship after executing Listing 19.41.

column `fkdid` to the table for C which holds the primary key to the D entities as foreign key. This column thus has a `REFERENCES` constraint to the foreign key, which we add at the end of the script Listing 19.39 via `ALTER TABLE`. Different from the previous section, this column `fkdid` must also have a `NOT NULL` constraint, because each entity of type C must necessarily be related to one entity of type D. The column also has a `UNIQUE` constraint, because the entities of type D can only reference at most one entity of type C.

We can only insert rows into the table for entity type C that reference existing rows in the table for entity type D. The consequence is that these elements must be created first, as shown in Listing 19.41. Thus, after inserting data into table `d`, we then insert the rows into table `c`. Each of these rows must provide a proper and unique foreign key `fkdid` referencing the primary key value `did` in table `d`. The contents of the two tables `c` and `d` after executing Listing 19.41 is illustrated in Figure 19.7. The data from the two tables can be combined using a single `INNER JOIN`.

Relating entities of type C to multiple entities of type D is impossible, because the column `fkcid` in table `c` can only have one value. Relating entities of type D to multiple entities of type C is impossible,

Listing 19.43: It is impossible to insert a row into table **C** that references a row in table **D** that is already referenced by another row in table **C**. (stored in file `CD_insert_error_1.sql`; output in [Listing 19.44](#))

```

1  /* Insert a wrong row into tables for C---| | -D relationship. */
2
3  -- It is impossible to create a row in C that references a row in D
4  -- which is already referenced by another record.
5  INSERT INTO c (fkdid, x) VALUES (3, '555');
```

Listing 19.44: The `stdout` of the program `CD_insert_error_1.sql` given in [Listing 19.43](#).

```

1 $ psql "postgres://postgres:XXX@localhost/relationships" -v ON_ERROR_STOP=1
2   ↪ -ebf CD_insert_error_1.sql
3 psql:conceptualToRelational/CD_insert_error_1.sql:5: ERROR:  duplicate key
4   ↪ value violates unique constraint "c_fkdid_key"
3 DETAIL:  Key (fkdid)=(3) already exists.
4 psql:conceptualToRelational/CD_insert_error_1.sql:5: STATEMENT:  /* Insert
5   ↪ a wrong row into tables for C---| | -D relationship. */
6 -- It is impossible to create a row in C that references a row in D
7 -- which is already referenced by another record.
7 INSERT INTO c (fkdid, x) VALUES (3, '555');
8 # psql 16.12 failed with exit code 3.
```

Listing 19.45: It is impossible to insert a row into table **C** that does not reference a row in table **D**. (stored in file `CD_insert_error_2.sql`; output in [Listing 19.46](#))

```

1  /* Insert a wrong row into tables for C---| | -D relationship. */
2
3  -- It is impossible to create a row in C that references no row in D.
4  INSERT INTO c (x) VALUES ('365');
```

Listing 19.46: The `stdout` of the program `CD_insert_error_2.sql` given in [Listing 19.45](#).

```

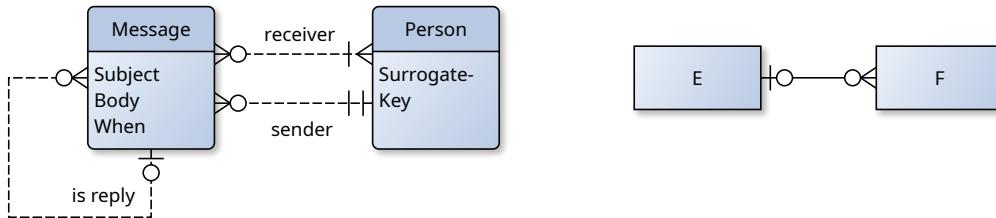
1 $ psql "postgres://postgres:XXX@localhost/relationships" -v ON_ERROR_STOP=1
2   ↪ -ebf CD_insert_error_2.sql
3 psql:conceptualToRelational/CD_insert_error_2.sql:4: ERROR:  null value in
4   ↪ column "fkdid" of relation "c" violates not-null constraint
3 DETAIL:  Failing row contains (6, null, 365).
4 psql:conceptualToRelational/CD_insert_error_2.sql:4: STATEMENT:  /* Insert
5   ↪ a wrong row into tables for C---| | -D relationship. */
6 -- It is impossible to create a row in C that references no row in D.
6 INSERT INTO c (x) VALUES ('365');
7 # psql 16.12 failed with exit code 3.
```

because of the `UNIQUE` constraint imposed on it the column `fkcid` in table **C**. It is impossible to insert a row into table **C** that references a row in table **D** that is already referenced by another row in table **C**, as shown in [Listing 19.43](#). Neither can we insert a row into table **C** that does *not* reference any row in table **D**, because of the `NOT NULL` constraint, as illustrated by [Listing 19.45](#). Notice that, together, this also means there can never be fewer entities of type D than entities of type C.

19.2.2.3 E $\rightarrow\!\!\!-\!\!\!$ F

We have the two entity types E and F. Each entity of type E may be connected to zero, one, or multiple entities of type F. Each entity of type F is connected to zero or one entity of type E.

We encountered this relationship pattern back in [Figure 18.16](#), where we tried to model the messaging subsystem for our teaching management platform. A message can either be a new message or a reply to exactly one existing message. There may be zero, one, or many replies to any existing message. This situation is sketched in [Figure 19.8](#).

Figure 19.8: We encountered the $E +o--< F$ relationship pattern in Figure 18.16.

Listing 19.47: The three-table realization of an $E +o--< F$ conceptual relationship. (stored in file `EF_1_tables.sql`; output in Listing 19.48)

```

1  /* Create the tables for an E +o----o-<- F relationship. */
2
3  -- Table E: Each row in E is related to zero or one or many rows in F.
4  CREATE TABLE e (
5      eid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
6      x    CHAR(3)    -- example of other attributes
7  );
8
9  -- Table F: Each row in F is related to zero or one row in E.
10 CREATE TABLE f (
11     fid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
12     y    CHAR(2)    -- example of other attributes
13 );
14
15 -- The table for managing the relationship between E and F.
16 CREATE TABLE relate_e_and_f (
17     fkeid INT NOT NULL                      REFERENCES e (eid),
18     fkfid INT NOT NULL UNIQUE PRIMARY KEY REFERENCES f (fid)
19 );
```

Listing 19.48: The stdout of the program `EF_1_tables.sql` given in Listing 19.47.

```

1  $ psql "postgres://postgres:XXX@localhost/relationships" -v ON_ERROR_STOP=1
2      ↪ -efb EF_1_tables.sql
3  CREATE TABLE
4  CREATE TABLE
5  CREATE TABLE
5  # psql 16.12 succeeded with exit code 0.
```

We need a table `e` for the entities of type E and a table `f` for the entities of type F. Both have their respective surrogate primary keys `eid` and `fid`. We also add the example attributes `x` and `y` to them, respectively. Apart from that, there can again be two solutions to implementing this relationship: We can use three tables or two, again, depending on how many entities of types E and F are actually related.

We can use a third table `relate_e_and_f` relating the entities of type E to those of type F. This approach is illustrated in Listing 19.47. There will be two columns in that table. The first one, `fkeid`, holds the reference to the primary key `eid` of the E entities. The second one, `fkfid`, holds the reference to the primary key `fid` of the F entities. Since we only store the pairs that exist, both columns are `NOT NULL` and have `REFERENCES` constraints to their respective foreign keys.

Since each entity of type F can only be connected to at most one entity of type E, there will be a `UNIQUE` constraint on the column `fkfid`. We also make it would be the primary key of table `relate_e_and_f`. It needs to be the primary key because the values in column `fkeid` do not need to be unique – one entity of type E can be related to many entities of type F.

In Listing 19.49, we insert some data into the tables. Since both relationship ends are optional, we can first populate tables `e` and `f`. Then we can establish the relationships by adding rows to

Listing 19.49: Inserting into and selecting data from the three-table realization of an $E \rightarrow\!\!\!-\!\!< F$ conceptual relationship given in Listing 19.47. (stored in file `EF_1_insert_and_select.sql`; output in Listing 19.50)

```

1  /* Inserting data into the tables for the E-----o<-F relationship. */
2
3  -- Insert some rows into the table for entity type E.
4  INSERT INTO e (x) VALUES ('123'), ('456'), ('789'), ('101');
5
6  -- Insert some rows into the table for entity type F.
7  INSERT INTO f (y) VALUES ('AB'), ('CD'), ('EF'), ('GH');
8
9  -- Create the relationships between the E and F rows.
10 INSERT INTO relate_e_and_f (fkeid, fkfid) VALUES (1, 1), (1, 2), (3, 4);
11
12 -- Combine the rows from E and F. This needs two INNER JOINS.
13 SELECT eid, x, fid, y FROM relate_e_and_f
14   INNER JOIN e ON e.eid = relate_e_and_f.fkeid
15   INNER JOIN f ON f.fid = relate_e_and_f.fkfid;

```

Listing 19.50: The `stdout` of the program `EF_1_insert_and_select.sql` given in Listing 19.49.

```

1  $ psql "postgres://postgres:XXX@localhost/relationships" -v ON_ERROR_STOP=1
2    ↪ -ebf EF_1_insert_and_select.sql
3
4  INSERT 0 4
5  INSERT 0 4
6  INSERT 0 3
7  eid | x   | fid | y
8  ----+---+----+---
9  1   | 123 | 1   | AB
10 1   | 123 | 2   | CD
11 3   | 789 | 4   | GH
12 (3 rows)
13
14 # psql 16.12 succeeded with exit code 0.

```

Table <code>e</code>		Table <code>f</code>		Table <code>relate_e_and_f</code>	
<code>eid</code>	<code>x</code>	<code>fid</code>	<code>y</code>	<code>fkeid</code>	<code>fkfid</code>
1	"123"	1	"AB"	1	1
2	"456"	2	"CD"	1	2
3	"789"	3	"EF"	3	4
4	"101"	4	"GH"		

Figure 19.9: The contents of the the tables in the three-table implementation of the $E \rightarrow\!\!\!-\!\!< B$ conceptual relationship after executing Listing 19.49.

table `relate_e_and_f`. We can re-assemble the data using two `INNER JOIN`s. The contents of the three tables after executing Listing 19.49 are given in Figure 19.9.

This is the recommended solution in [396], but we face a similar situation as back in Section 19.2.2.1 (A $\rightarrow\!\!\!-\!\!<$ B): It is efficient only if there are not too many E-F pairs that are related. If almost all entities of type F are related to one entity of type E, then the table `relate_e_and_f` is about as same as big as the table F. Instead of simply storing the keys of the related E entities in the table `f`, we now have another table that stores these keys *and* the primary keys of the `f` table. Also, we would again need two `INNER JOIN` statements instead of one when we merge the data, as shown in Listing 19.49.

We thus first delete the tables again in Listing 19.51. The two-table solution is illustrated in Listing 19.53. Here, we need to add a foreign key column `fkeid` to the table `f`. This column has a `REFERENCES` constraint pointing to column `eid` of table `e`. A value in this column can be `NULL`, because not all rows in table `f` need to be related to rows in table `e`. Different from the A $\rightarrow\!\!\!-\!\!<$ B situation

Listing 19.51: Deleting the tables created in Listing 19.47. (stored in file `EF_cleanup.sql`; output in Listing 19.52)

```

1  /* Drop the tables for the E-|o----o<-F relationship. */
2
3  DROP TABLE IF EXISTS relate_e_and_f;
4  DROP TABLE IF EXISTS e;
5  DROP TABLE IF EXISTS f;
```

Listing 19.52: The stdout of the program `EF_cleanup.sql` given in Listing 19.51.

```

1  $ psql "postgres://postgres:XXX@localhost/relationships" -v ON_ERROR_STOP=1
   ↪ -eef EF_cleanup.sql
2  DROP TABLE
3  DROP TABLE
4  DROP TABLE
5  # psql 16.12 succeeded with exit code 0.
```

Listing 19.53: A two-table realization of an $E \rightarrow\!\!-\!\!-\! F$ conceptual relationship. (stored in file `EF_2_tables.sql`; output in Listing 19.54)

```

1  /* Create the tables for an E-|o----o<-F relationship. */
2
3  -- Table E: Each row in E is related to zero or one or many rows in F.
4  CREATE TABLE e (
5      eid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
6      x     CHAR(3)    -- example of other attributes
7  );
8
9  -- Table F: Each row in F is related to zero or one row in E.
10 CREATE TABLE f (
11     fid  INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
12     fkeid INT REFERENCES e (eid),    -- the foreign key to E, can be NULL.
13     y     CHAR(2)    -- example of other attributes
14 );
```

Listing 19.54: The stdout of the program `EF_2_tables.sql` given in Listing 19.53.

```

1  $ psql "postgres://postgres:XXX@localhost/relationships" -v ON_ERROR_STOP=1
   ↪ -eef EF_2_tables.sql
2  CREATE TABLE
3  CREATE TABLE
4  # psql 16.12 succeeded with exit code 0.
```

in Listing 19.33, there *must not* be a `UNIQUE` constraint here: Each row of table `e` can be related to multiple rows in table `f`. This means that the primary key values from table `e` are allowed to occur multiple times as foreign key values in column `fkeid` in table `f`.

We store some data into the two tables in Listing 19.55. We can do this by first inserting rows into table `e`. Then we add rows to table `f`. For each such row, we can provide a valid foreign key `fkeid` to a row in table `e` identified by primary key `eid`. If we do this, then the new row in table `f` will be related to the existing row in table `e`. We can also store `NULL` as `fkeid`. In this case the row in table `f` is not related to any row in table `e`. The contents of the two tables after executing Listing 19.55 are given in Figure 19.10. We also only need a single `INNER JOIN` to merge the data, as illustrated in Listing 19.55.

This solution has another useful aspect: In the three-table solution, I can create an `INSERT` or `UPDATE` query to `relate_e_and_f` that fails, namely if I try to relate a row of `f` to more than one row in `e`. This is not possible in the two-table solution, because each row of `f` only has one attribute value `fkeid`.

Listing 19.55: Inserting into and selecting data from the two-table realization of an $E \rightarrow\!\!\!-\!\!\!< F$ conceptual relationship given in Listing 19.53. (stored in file `EF_2_insert_and_select.sql`; output in Listing 19.56)

```

1  /* Inserting data into the tables for the E-|o-----o<-F relationship. */
2
3  -- Insert some rows into the table for entity type E.
4  INSERT INTO e (x) VALUES ('123'), ('456'), ('789'), ('101');
5
6  -- Insert some rows into the table for entity type F.
7  INSERT INTO f (y, fkeid) VALUES ('AB', 1), ('CD', 1), ('EF', NULL),
8      ('GH', 3);
9
10 -- Combine the rows from E and F. This needs one INNER JOIN.
11 SELECT eid, x, fid, y FROM e INNER JOIN f ON f.fkeid = e.eid;

```

Listing 19.56: The `stdout` of the program `EF_2_insert_and_select.sql` given in Listing 19.55.

```

1 $ psql "postgres://postgres:XXX@localhost/relationships" -v ON_ERROR_STOP=1
2   ↳ -ef EF_2_insert_and_select.sql
3
4 INSERT 0 4
5 INSERT 0 4
6   eid | x    | fid | y
7   ----+----+----+----+
8   1   | 123 | 1   | AB
9   1   | 123 | 2   | CD
10  3   | 789 | 4   | GH
11  (3 rows)
12
13 # psql 16.12 succeeded with exit code 0.

```

Table <code>e</code>		Table <code>f</code>		
<code>eid</code>	<code>x</code>	<code>fid</code>	<code>fkeid</code>	<code>y</code>
1	"123"	1	1	"AB"
2	"456"	2	1	"CD"
3	"789"	3	NULL	"EF"
4	"101"	4	3	"GH"

Figure 19.10: The contents of the the tables in the two-table implementation of the $E \rightarrow\!\!\!-\!\!\!< B$ conceptual relationship after executing Listing 19.55.

19.2.2.4 $G \rightarrow\!\!\!-\!\!\!< H$

We have the two entity types G and H . Each entity of type G must be connected to at least one but maybe many entities of type H . Each entity of type H is connected to zero or one entity of type G .

An example of the $G \rightarrow\!\!\!-\!\!\!< H$ relationship pattern is given back in Section 18.5 (The Cardinality of Relationships): In a soccer club, each trainer coaches several club members. Each member can be coached by one trainer or, if they have other functions, not be coached at all. This example is illustrated in Figure 19.11.

We again create two tables `g` and `h`, respectively. They have the primary keys `gid` and `hid` as well as the example attributes `x` and `y`, respectively. Then we can approach this in the same way as the in the two-table manner in the $E \rightarrow\!\!\!-\!\!\!< F$ situation. However, enforcing this relationship pattern is a bit



Figure 19.11: An example of the $G \rightarrow\!\!\!-\!\!\!< H$ relationship pattern from back in Section 18.5 (The Cardinality of Relationships).

Listing 19.57: The realization of an $G \rightarrowtail H$ conceptual relationship. (stored in file `GH_tables.sql`; output in Listing 19.58)

```

1  /* Create the tables for a G-|o-----|<-H relationship. */
2
3  -- Table G: Each row in G is related to one or multiple rows in H.
4  -- We force that that row in H is also related to our row in G via
5  -- a foreign key constraint g_h_fk.
6  CREATE TABLE g (
7      gid     INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
8      fkhid  INT NOT NULL UNIQUE,    -- row is related >= 1 H.
9      x      CHAR(3)                -- example of other attributes
10 );
11
12 -- Table H: Each row in H is related to zero or one rows in G.
13 CREATE TABLE h (
14     hid     INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
15     fkgid  INT REFERENCES g (gid),   -- can be related to 0 or 1 G
16     y      CHAR(2),                  -- example of attributes
17     UNIQUE (hid, fkgid)            -- needed for g_fkhid_gid_fk
18 );
19
20 -- To table G, we add the foreign key reference constraint towards H.
21 ALTER TABLE g ADD CONSTRAINT g_fkhid_gid_fk FOREIGN KEY (fkhid, gid)
22     REFERENCES h (hid, fkgid)

```

Listing 19.58: The stdout of the program `GH_tables.sql` given in Listing 19.57.

```

1 $ psql "postgres://postgres:XXX@localhost/relationships" -v ON_ERROR_STOP=1
2   ↪ -ebs GH_tables.sql
3 CREATE TABLE
4 CREATE TABLE
5 ALTER TABLE
5 # psql 16.12 succeeded with exit code 0.

```

more complicated, but doable, as illustrated in Listing 19.57:

We begin by preparing the table `g`. Each entity of type G must be related to at least one entity of type H. We solve this problem in two steps: First, we enforce that each row in table `g` is connected to one row in table `h`. Later, we permit that it can be connected to more rows. We therefore begin by adding an attribute `fkhid`, which must be `NOT NULL`. This column should always point to the row in table `h` with the same value in `hid`. We will later add a proper `REFERENCE` constraint via `ALTER TABLE`, because we cannot add it yet, because table `h` does not yet exist. So for now, just imagine that we had added it and that this attribute always references on row in table `h`. We will consider this attribute `fkhid` to identify the “first H” entity to which the row in table `g` is related. We make this column also `UNIQUE`, because no row in table `h` can be related to more than one row in table `g`. Thus, no primary key value `hid` can thus occur more than once in column `fkhid`.

Now we prepare the table for entity type H. Since each entity of type H can be related to either zero or one entity of type G, we add a column `fkgid` to table `h`. The values in this column can be `NULL`, in which case the corresponding row in `h` is not related to any entity of type G. If it is not `NULL`, then it must be a proper foreign key reference to a row in table `g`. The values in this column do *not* need to be `UNIQUE`, as multiple entities of type H can be related to the very same entity of type G.

At this stage, we have enforced that each entity of type H can be related to zero or one entity of type G. We could now go and add a constraint `g_fkhid_fk` as `FOREIGN KEY (fkhid)REFERENCES h (hid)` to table `g` via `ALTER TABLE g ADD CONSTRAINT...`. This would enforce that each entity of type G must be related to at least one entity of type H. However, this would *not* enforced that, if an entity of type G is related to one entity of type H as its “primary H,” then that *very same* H entity also is related to the G entity.

We can enforce this in a somewhat weird way: By creating the foreign key `REFERENCES` constraint not

Listing 19.59: Inserting into and selecting data from the realization of an $G \rightarrowtail H$ conceptual relationship given in Listing 19.57. (stored in file `GH_insert_and_select.sql`; output in Listing 19.60)

```

1  /* Inserting data into the tables for the G----->H relationship. */
2
3  -- Insert some rows into the table for entity type H.
4  -- Not specifying `g` leave the references G as NULL for now.
5  INSERT INTO h (y) VALUES ('AB'), ('CD'), ('EF'), ('GH'), ('IJ'), ('KL');
6
7  -- Insert into G and relate to H. We do this three times.
8  WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (1, '123')
9    RETURNING gid, fkhid)
10 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
11
12 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (3, '456')
13   RETURNING gid, fkhid)
14 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
15
16 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (4, '789')
17   RETURNING gid, fkhid)
18 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
19
20 -- Link one H row to another G row. (We do this twice.)
21 UPDATE h SET fkgid = 3 WHERE hid = 2;
22 UPDATE h SET fkgid = 3 WHERE hid = 5;
23
24 -- Combine the rows from G and H.
25 SELECT gid, x, hid, y FROM h INNER JOIN g ON g.gid = h.fkgid;

```

Listing 19.60: The `stdout` of the program `GH_insert_and_select.sql` given in Listing 19.59.

```

1 $ psql "postgres://postgres:XXX@localhost/relationships" -v ON_ERROR_STOP=1
2   ↪ -ebs GH_insert_and_select.sql
3
4 INSERT 0 6
5 UPDATE 1
6 UPDATE 1
7 UPDATE 1
8 UPDATE 1
9
10  gid |  x  | hid | y
11  ----+---+----+---
12  1  | 123 | 1  | AB
13  2  | 456 | 3  | EF
14  3  | 789 | 4  | GH
15  3  | 789 | 2  | CD
16  3  | 789 | 5  | IJ
17 (5 rows)
18
19 # psql 16.12 succeeded with exit code 0.

```

on the single column `hid`, but by enforcing that the pair `(fkhid, gid)` from table `g` must also appear as pair `(hid, fkgid)` in table `h`. We know that each value of `gid` can only exist once in table `g`. We also know that each value of `hid` can only exist once in table `h`. Therefore, the first part of the column pair in the constraint, `fkhid` (or, looking at it from the other side, `hid`), already selects one unique row in table `h`. There can never be another row with the same `hid` value, because that's the primary key of table `h`. The second element of the pair, i.e., `gid` (or, looking from the other side, `fkgid`) thus forces that this single row in table `h` has the fitting value `gid` stored in `fkgid`. Since foreign key `REFERENCES` constraints can only reference `UNIQUE` column(s), we must add the constraint `UNIQUE (hid, fkgid)` to table `h`.

Let's go over this one more time: We specify the constraint `g_fkhid_gid_fk`

Table g			Table h		
gid	fkhid	x	hid	fkgid	y
1	1	"123"	6	NULL	"KL"
2	3	"456"	1	1	"AB"
3	4	"789"	3	2	"EF"
			4	3	"GH"
			2	3	"CD"
			5	3	"IJ"

Figure 19.12: The contents of the two tables in the implementation of the $G \rightarrowtail H$ conceptual relationship after executing Listing 19.59.

Listing 19.61: Trying to create a row into table `g` that is not related to any row in table `h` is not possible. (stored in file `GH_insert_error_1.sql`; output in Listing 19.62)

```
1 /* Can we create a row in G unrelated to any row in H? */
2
3 INSERT INTO g (x) VALUES ('777');
```

Listing 19.62: The stdout of the program `GH_insert_error_1.sql` given in Listing 19.61.

```
1 $ psql "postgres://postgres:XXX@localhost/relationships" -v ON_ERROR_STOP=1
2   ↪ -ebs GH_insert_error_1.sql
3 psql:conceptualToRelational/GH_insert_error_1.sql:3: ERROR:  null value in
4   ↪ column "fkhid" of relation "g" violates not-null constraint
5 DETAIL:  Failing row contains (4, null, 777).
6 psql:conceptualToRelational/GH_insert_error_1.sql:3: STATEMENT:  /* Can we
7   ↪ create a row in G unrelated to any row in H? */
8 INSERT INTO g (x) VALUES ('777');
9 # psql 16.12 failed with exit code 3.
```

as `FOREIGN KEY (fkhid, gid)REFERENCES h (hid, fkgid)`. On the side of the table `g`, this constraint looks at a row and takes the value of its foreign key `fkhid` to table `h` together with its own primary key `gid` as a tuple `(fkhid, gid)`. On the side of the table `h`, there must be a corresponding tuple with the primary key `hid` and the value of its corresponding foreign key `fkgid`. It is enforced that the pairs `(fkhid, gid) == (hid, fkgid)` always fit.

Of course, the primary keys of both tables are always unique. Let's say a row in table `g` has primary key `gid=u` and foreign key `fkhid=v`. Then the row in table `h` with primary key `hid=v` must have the foreign key `fkgid=u`. Of course, there could also be another row in table `h` which also has foreign key `fkgid=u`. That's OK, because multiple entities of type `H` can reference the same entity of type `G`.

With these constraints, given in Listing 19.57, we have implemented the relationship pattern. Let us check what we have done.

Can we insert a row into table `g` that is not related to at least one row in table `h`? No, we cannot, as shown in Listing 19.61. Inserting a row into table `g` requires us to specify a value for the foreign key `fkhid` and the corresponding row in table `h` must exist. We can insert rows into table `h` that are not related to any row in table `g`, but that is OK: Entities of type `H` are related to either zero or one entities of type `G`. But could we insert a row into table `g` that references a row in table `h` that is not already related to an entity of type `G`? No, because the constraint `g_fkhid_gid_fk` requires that the corresponding row in table `h` would reference back to the row in table `g` (Listing 19.63).

So how do we actually insert rows into table `g`? We cannot insert a row that does not reference an existing row in table `h`. We cannot insert a row that references a row in table `h` which is not already connected to another row in table `g`. And if we wanted to insert a row into table `g` that references a row in table `h` that already references some row in table `g`, then this would mean that we already need to have an existing row in table `g`. Which we do not have.

We have created our two tables and protected their referential integrity using fierce constraints. And these constraints need to be exactly like that, because we want to implement $G \rightarrowtail H$. Now we need to see that we can insert data into the tables. We have to solve this odd chicken-and-egg

Listing 19.63: Trying to create a row in table `g` that references a row in table `h` which is not referencing any row in table `g` is not possible. (stored in file `GH_insert_error_2.sql`; output in Listing 19.64)

```
1 /* Can we create a row in G related to a row in H unrelated to any G? */
2
3 INSERT INTO g (fkhid, x) VALUES (6, '888');
```

Listing 19.64: The stdout of the program `GH_insert_error_2.sql` given in Listing 19.63.

```
1 $ psql "postgres://postgres:XXX@localhost/relationships" -v ON_ERROR_STOP=1
   ↪ -eef GH_insert_error_2.sql
2 psql:conceptualToRelational/GH_insert_error_2.sql:3: ERROR: insert or
   ↪ update on table "g" violates foreign key constraint "g_fkhid_gid_fk"
3 DETAIL: Key (fkhid, gid)=(6, 5) is not present in table "h".
4 psql:conceptualToRelational/GH_insert_error_2.sql:3: STATEMENT: /* Can we
   ↪ create a row in G related to a row in H unrelated to any G? */
5 INSERT INTO g (fkhid, x) VALUES (6, '888');
6 # psql 16.12 failed with exit code 3.
```

Listing 19.65: Trying to create a row in table `g` that references a row in table `h` which is already referencing another row in table `g` is not possible. (stored in file `GH_insert_error_3.sql`; output in Listing 19.66)

```
1 /* Can we insert a G that is related to a H related to another G? */
2
3 -- H with id 4 is already related to G with id 3.
4 -- Can we make our new G row point to it as its "primary H" anyway?
5 INSERT INTO g (fkhid, x) VALUES (4, '999');
```

Listing 19.66: The stdout of the program `GH_insert_error_3.sql` given in Listing 19.65.

```
1 $ psql "postgres://postgres:XXX@localhost/relationships" -v ON_ERROR_STOP=1
   ↪ -eef GH_insert_error_3.sql
2 psql:conceptualToRelational/GH_insert_error_3.sql:5: ERROR: duplicate key
   ↪ value violates unique constraint "g_fkhid_key"
3 DETAIL: Key (fkhid)=(4) already exists.
4 psql:conceptualToRelational/GH_insert_error_3.sql:5: STATEMENT: /* Can we
   ↪ insert a G that is related to a H related to another G? */
5 -- H with id 4 is already related to G with id 3.
6 -- Can we make our new G row point to it as its "primary H" anyway?
7 INSERT INTO g (fkhid, x) VALUES (4, '999');
8 # psql 16.12 failed with exit code 3.
```

problem.

In Listing 19.59, we begin by first filling the table `h`. Since entities of type `H` do not necessarily be related any entity of type `G`, this can be done without worrying about constraints. However, when we insert an entity of type `G` into our table `g`, we must, at the same time, create a relationship to an entity of type `H` in table `h`. This sounds impossible, but it is not. We do this without problem by knowing that:

A constraint that is not deferrable will be checked immediately after every command. Checking of constraints that are deferrable can be postponed until the end of the transaction...

— [107], 2025

In PostgreSQL (and probably several other DBMSes), referential integrity constraints are checked at the end of the execution of a transaction. A transaction can group several commands together, that either all succeed or all fail (in which case, the DB will not be modified). A single command in PostgreSQL

Listing 19.67: Trying the same thing as in Listing 19.65, but this time also attempting to re-adjusting the row in `h` with an `UPDATE` instruction to make it reference the new row in table `g`, is also not possible. (stored in file `GH_insert_error_4.sql`; output in Listing 19.68)

```

1  /* Can we insert a G that is related to a H related to another G? */
2
3  -- H with id 4 is already related to G with id 3.
4  -- Can we make it point to the new G row instead?
5  WITH g_new AS (INSERT INTO g (fkhid, x) VALUES (4, '555')
6    RETURNING gid, fkhid)
7  UPDATE h SET fkgid = g_new.gid FROM g_new WHERE hid = fkhid;

```

Listing 19.68: The stdout of the program `GH_insert_error_4.sql` given in Listing 19.67.

```

1  $ psql "postgres://postgres:XXX@localhost/relationships" -v ON_ERROR_STOP=1
2    ↪ -ebs GH_insert_error_4.sql
3  psql:conceptualToRelational/GH_insert_error_4.sql:7: ERROR:  duplicate key
4    ↪ value violates unique constraint "g_fkhid_key"
5  DETAIL:  Key (fkhid)=(4) already exists.
6  psql:conceptualToRelational/GH_insert_error_4.sql:7: STATEMENT:  /* Can we
7    ↪ insert a G that is related to a H related to another G? */
8  -- H with id 4 is already related to G with id 3.
9  -- Can we make it point to the new G row instead?
10 WITH g_new AS (INSERT INTO g (fkhid, x) VALUES (4, '555')
11   RETURNING gid, fkhid)
12   UPDATE h SET fkgid = g_new.gid FROM g_new WHERE hid = fkhid;
# psql 16.12 failed with exit code 3.

```

also is a transactions [450]. This means that referential integrity constraints are checked at the end of the command execution. The changes are committed to the DB if the constraints are met and rolled back otherwise.

Now, to insert a row into table `g`, we also need to modify a record from table `h` that currently is unrelated to any row in table `g`. We need to relate that row in table `h` to that new row in table `g`. If we can wrap the insertion and the modification into a single `SQL` command, then things might be easy. Otherwise we would need to learn how to explicitly use transactions, which would also be OK.

In order to modify the `fkgid` value of a row in table `h`, we must know the primary key `gid` of the new row in table `g`. We declared the primary key `gid` of table `g` as `INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY`. This means that the values of this key are generated when the rows are created. This, in turn, means that before inserting the row into table `g`, we do not know the value that its primary key will have. Luckily, this is a very common problem: “What if we insert some data into a table with automatically generated primary keys and then need the key that was assigned to that data?” PostgreSQL offers an answer with the `RETURNING` keyword² [360].

The statement `INSERT INTO g (fkhid, x)VALUES (1, '123')RETURNING gid, fkhid` would insert a row into the table `g` where the value of the attribute `x` is `'123'` and the value of the attribute `fkhid` is `1`. It would return the value of the primary key column `gid` that was automatically generated and assigned to this row as well as the value of `fkhid`, which would be `1` in this case. Of course, this statement will fail, because it would violate our constraint `g_fkhid_gid_fk` from Listing 19.57. But we are one step closer to make things work.

One idea would be to build some Frankensteinian query trying to plug the first insert into the second in the form `UPDATE h SET gid = (INSERT INTO g (fkhid, x)VALUES (1, '123')RETURNING gid, fkhid)WHERE h.hid = fkhid;`. This is not a valid approach in SQL, and PostgreSQL does not like this either. It would be a single statement, but this does not work.

However, we can use another tool: `common table expressions (CTEs)`. A CTE allows us to assign a name to a sub-expression of a query. This sub-expression then works a bit like a temporary table. It is evaluated only once and can be used like a read-only table in other parts of the query. For

²This is a PostgreSQL addition to SQL [463] and does not seem to be part of the SQL standard, but it is also supported by MariaDB [226] and SQLite [361].

Listing 19.69: Changing a row in table `h` that references a row in table `g` to now reference another row in table `g` is not possible. (stored in file `GH_insert_error_5.sql`; output in Listing 19.70)

```

1  /* Can we change the relationship of a H away from its primary G? */
2
3  -- H with id 1 is used as "primary H" for G with ID 1.
4  -- Can we make it point to another G?
5  UPDATE h SET fkgid = 3 WHERE hid = 1;

```

Listing 19.70: The stdout of the program `GH_insert_error_5.sql` given in Listing 19.69.

```

1 $ psql "postgres://postgres:XXX@localhost/relationships" -v ON_ERROR_STOP=1
2   ↪ -ebs GH_insert_error_5.sql
3 psql:conceptualToRelational/GH_insert_error_5.sql:5: ERROR: update or
4   ↪ delete on table "h" violates foreign key constraint "g_fkhid_gid_fk"
5   ↪ on table "g"
6 DETAIL: Key (hid, fkgid)=(1, 1) is still referenced from table "g".
7 psql:conceptualToRelational/GH_insert_error_5.sql:5: STATEMENT: /* Can we
8   ↪ change the relationship of a H away from its primary G? */
9 -- H with id 1 is used as "primary H" for G with ID 1.
10 -- Can we make it point to another G?
11 UPDATE h SET fkgid = 3 WHERE hid = 1;
12 # psql 16.12 failed with exit code 3.

```

example, `WITH cats AS (SELECT age, name FROM animals WHERE type='cat')`, we would assign the result of the query `SELECT age, name FROM animals WHERE type='cat'` to the CTE `cats`. We could then use `cats` as if it was a read-only table and do something like `SELECT name FROM cats;` to get the cat names. It's a bit like a `VIEW`, but it is part of a single SQL command.

We now put everything together: Assume that we already inserted a row into table `h` that has the primary key 1. We can do that at any time, because these rows do not need to be related to rows in table `g`. Now, we put the row insertion into table `g` into a CTE `g_new` by writing `WITH g_new AS (INSERT INTO g (fkhid, x)VALUES (1, '123')RETURNING id, fkhid)`. This CTE will insert a row into table `g` that references the row of table `h` that has primary key 1. It will also return the primary key of the newly generated row in table `g` as `gid` and the foreign key `fkhid` (with value 1). Then we use this CTE when updating the row with primary key `hid = fkhid` in table `h`. We do `UPDATE h SET fkgid = g_new.gid FROM g_new WHERE h.gid = g_new.fkhid;`. This would make the foreign key stored in this row point to our new row in table `g`. Since both sub-expression are part of a single command, the referential integrity constraints (the `REFERENCES` constraints) are checked only at the very end, when the `;` is reached. By this time, the referential integrity has been established, since we inserted one row into table `g` and made the corresponding row in table `h` reference it.

The contents of the tables `g` and `h` after inserting the data are shown in Figure 19.12. Relating the existing rows in table `g` to additional rows in table `h` is then much easier. This can be done with single `UPDATE` statements applied to table `h`.

We now conduct a few additional sanity tests to check whether our constraints work and really protect the referential integrity. First, we check whether it is possible to make a second row in table `g` relate to a row in table `h` that is already related to another row in table `g`. This is not allowed by our relationship model. The output of the execution of Listing 19.65 shows that this is not possible. We can also not redirect a row in table `h` to a new row in table `g` (Listing 19.67) nor can we make a row in table `h` that currently is related to a row in table `g` point to another one via an `UPDATE` (Listing 19.69). It indeed seems that our constraints properly protected the relationships.

19.2.2.5 I $\perp\!\!\!\perp$ J

We have the two entity types I and J. Each entity of type I must be linked to exactly one entity of type J. Each entity of type J must be linked to exactly one entity of type I.

We discussed a few examples of this relationship pattern from back in Section 18.5 (The Cardinality of Relationships). For example, in classical police movies or TV shows, there are always teams of two

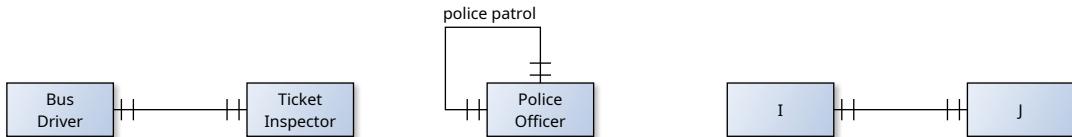


Figure 19.13: Examples of the $I \parallel\!\!\!-\!\!\!-|| J$ relationship pattern from back in [Section 18.5](#) (The Cardinality of Relationships).

Listing 19.71: The single-table realization of an $I \parallel\!\!\!-\!\!\!-|| J$ conceptual relationship. ([src](#))

```

1  /* Create the table for an I-||----||-J relationship. */
2
3  -- Table I_J: Each row holds one entity of type I and one of type J.
4  CREATE TABLE i_j (
5      id INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
6      x CHAR(3),    -- example of other attributes of entity type I
7      y CHAR(2)     -- example of other attributes of entity type J
8 );

```

police(wo)men together performing a police patrol. We can also imagine that there always is one bus driver working together with one ticket inspector together to service a bus route. These examples are illustrated in [Figure 19.13](#).

In this scenario, we only need a single table [396]. We just merge the attributes of the two entity types and put them all into one table. If each pair of related entities of types I and J forms a single row in table [i_j](#), then there can never be any issue with the referential integrity. Neither can we have a row without the I entity, nor can we have a row without entity of type J. This is illustrated in [Listing 19.71](#). Since this is so very straightforward, we will not exercise inserting data into this table and selecting it back in another listing.

19.2.2.6 $K \parallel\!\!\!-\!\!\!-○\!L$

We have the two entity types K and L. Each entity of type K may be connected to zero, one, or multiple entities of type L. Each entity of type L is connected to exactly one entity of type K.

We already encountered this relationship pattern in back in [Figure 18.14](#), when we modeled the relationship between the entity types *Personal ID* and *ID Type*. We found that there can be many different types of IDs, including the Chinese IDs (中国公民身份号码), passports, visas, or even mobile phone numbers. For each such ID-type, we may have zero, one, or many personal ID records stored in our DB. Each personal ID, however, must always belong to exactly one ID type. This is illustrated in [Figure 19.14](#).

We need a table [k](#) for the entities of type K and a table [l](#) for the entities of type L. We call the primary key for table [k](#) [kid](#) and also add the example attribute [x](#). The primary key for table [l](#) be [lid](#) and we again provide the example attribute [y](#). Every row of table [l](#) must be related to exactly one row of table [k](#). The three table solution from the E $\rightarrow\!\!\!-\!\!\!-○\!F$ scenario therefore makes no sense: The

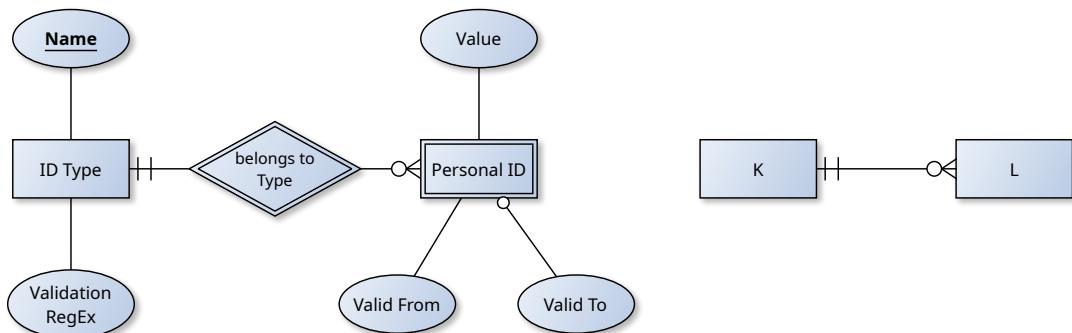


Figure 19.14: We encountered the $K \parallel\!\!\!-\!\!\!-○\!L$ relationship pattern in [Figure 18.14](#).

Listing 19.72: The realization of a $K \rightarrow\!\!\!-\!-\!o\leftarrow L$ conceptual relationship. (stored in file `KL_tables.sql`; output in Listing 19.73)

```

1  /* Create the tables for a K || -----o<-L relationship. */
2
3  -- Table K: Each row in K is related to zero or one or many rows in L.
4  CREATE TABLE k (
5      kid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
6      x    CHAR(3)   -- example of other attributes
7  );
8
9  -- Table L: Each row in L is related to exactly one row in K.
10 CREATE TABLE l (
11     lid    INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
12     fkkid INT NOT NULL REFERENCES k (kid),   -- the foreign key to K.
13     y    CHAR(2)   -- example of other attributes
14 );

```

Listing 19.73: The stdout of the program `KL_tables.sql` given in Listing 19.72.

```

1  $ psql "postgres://postgres:XXX@localhost/relationships" -v ON_ERROR_STOP=1
2      ↪ -eef KL_tables.sql
3  CREATE TABLE
4  CREATE TABLE
4  # psql 16.12 succeeded with exit code 0.

```

third table could never have just a few rows, it would always have exactly as many rows as Table 1.

The sensible solution is to add a foreign key column to table 1 that references the primary key of table k, as shown in Listing 19.72. Since the relationship is mandatory, the column must be `NOT NULL`. It does not need to be unique, because each row in table k can be related to arbitrarily many rows in table 1.

We insert some data into this structure in Listing 19.74. We can first create the rows for table k, because their relationship to rows in table 1 is optional. Then we insert rows in table 1. For each of them, we must provide a valid value for the column `fkkid`, i.e., for the foreign key to table k. The value in this column must match one existing value in column `kid` of table k. The reason is that each row in table 1 must be related to exactly one row in table k. Thus, providing `NULL` as `fkkid` is not possible. The contents of the tables k and 1 after inserting the data are shown in Figure 19.15.

It is impossible to add a row to table 1 without providing an existing value of the primary key attribute `kid` of table k for the foreign key column `fkkid`. In other words, because of the `REFERENCES` and the `NOT NULL` constraint, each row in table 1 must reference exactly one row in table k. It cannot reference more than one row, because the attribute `fkkid` can only take on a single value. In turn, the rows in table k can be referenced by arbitrarily many rows in table 1: maybe by no row at all, maybe by one, maybe by hundreds. This is exactly the meaning of the “optionally-many” relationship end pointing towards entity type L. In Listing 19.74, we insert some rows into both tables and combine the data together using a single `INNER JOIN`.

Listing 19.74: Inserting into and selecting data from the realization of an $K \bowtie L$ conceptual relationship given in Listing 19.72. (stored in file `KL_insert_and_select.sql`; output in Listing 19.75)

```

1  /* Inserting data into the tables for the K ||----o<-L relationship. */
2
3  -- Insert some rows into the table for entity type K.
4  INSERT INTO k (x) VALUES ('123'), ('456'), ('789'), ('101'), ('202');
5
6  -- Insert some rows into the table for entity type L, referencing K.
7  INSERT INTO l (y, fkkid) VALUES ('AB', 1), ('CD', 1), ('EF', 4),
8      ('GH', 3);
9
10 -- Combine the rows from K and L.
11 SELECT kid, x, lid, y FROM l INNER JOIN k ON l.fkkid = k.kid;

```

Listing 19.75: The stdout of the program `KL_insert_and_select.sql` given in Listing 19.74.

```

1 $ psql "postgres://postgres:XXX@localhost/relationships" -v ON_ERROR_STOP=1
2   ↪ -ebl KL_insert_and_select.sql
3
4 INSERT 0 5
5 INSERT 0 4
6
7
8
9
10
11
12 # psql 16.12 succeeded with exit code 0.

```

Table <code>k</code>		Table <code>l</code>	
<code>kid</code>	<code>x</code>	<code>lid</code>	<code>y</code>
1	"123"	1	"AB"
2	"456"	2	"CD"
3	"789"	3	"EF"
4	"101"	4	"GH"
5	"202"		

Figure 19.15: The contents of the two tables in the implementation of the $K \bowtie L$ conceptual relationship after executing Listing 19.74.

19.2.2.7 $M \bowtie\leftarrow N$

We have the two entity types M and N . Each entity of type M must be linked to at least one entity of type N , but can also be linked to many of them. Each entity of type N is connected to exactly one entity of type M .

We encountered this relationship pattern in Figure 18.14, when we tried to model the relationship between entities of type *Person* and *Personal ID*. Each person must have at least one personal ID. Each personal ID belongs to exactly one person. This is illustrated in Figure 19.16.

Implementing the referential integrity of this relationship pattern in SQL is a bit complicated but doable. I did struggle with the $G \rightarrowtail\leftarrow H$ relationship pattern back in Section 19.2.2.4 ($G \rightarrowtail\leftarrow H$), but in the end we figured it out. A similar structure can be implemented now. We only have one more constraint, namely the one that enforces that all entities of type N are linked to one entity of type M each. And this additional constraint is the same that we used back then, just “the other way around.” Writing down the table structures and constraints as shown in Listing 19.76 can thus be understood once we comprehend Listing 19.57 from back in Section 19.2.2.4. Well, almost.

We create a table `m` for the entities of type M in Listing 19.76. It has primary key `mid` and the additional example data column `x`. This also needs an attribute `fknid` that will later be used to

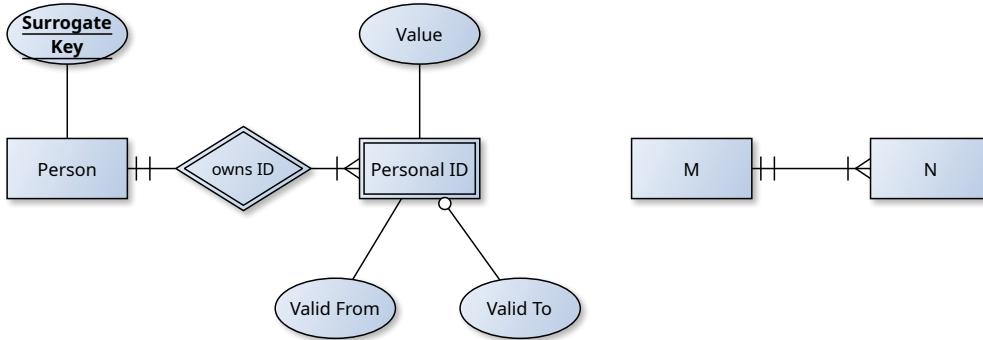


Figure 19.16: We encountered the $M \rightarrow\!\!\!-\!-\!- N$ relationship pattern in Figure 18.14.

reference one row in table `n`. Since each entity of type `M` must be linked to at least one entity of type `N`, this column will be `NOT NULL`. Since each entity of type `N` must be linked to exactly one entity of type `M` (and not more than one), we also mark the column as `UNIQUE`.

Now we create the table `n` for storing entities of type `N`. The primary key be `nid` and the additional example data column is `y`. Since each row in table `n` must be related to exactly one row in table `m`, we also add a column `fkmid` to this table. It `REFERENCES` the primary key `mid` of table `m`. The difference between Listing 19.76 and Listing 19.57 is that now `fkmid` of table `n` is `NOT NULL`, whereas the foreign key `fkgid` for table `h` could be `NULL`. This change is needed because every entity of type `N` must be related to an entity of type `M`, whereas back in Section 19.2.2.4, each entity of type `H` could be related to one or zero entities of type `G`. Apart from this difference, we now add the constraint `m_fknid_mid_fk` to table `m` which works exactly as `g_fkhid_gid_fk` in Listing 19.57.

It turns out that this single added `NOT NULL` constraint imposed on column `fkmid` of table `n` makes inserting data much harder. The “G end” of the relationship in Section 19.2.2.4 was an “optionally one”. Once we had created the tables and constraints, we could begin storing data by *first* populating the table for the entities of type `H`. Then we could take of creating the records for the table for the entities of type `G` and fix the referential integrity.

Regardless of how we look at it, we do not have such luck this time: Each entity of type `N` *must* be linked to exactly one existing entity of type `M`. Each entity of type `M` *must* be linked to at least one existing entity of type `N`. This is a much worse problem, because we need to know the primary key `nid` of a row in table `n` in order to create a new row in table `m`. And we need to know the primary key `mid` of a row in table `m` to create a new row in table `n`.

It is clear that we will have to use CTEs to approach this problem. The only solution I could find for this problem is based on the following idea: We need to, somehow, be able to get the value of the primary key that would be used when we create a new row in Tabelle `m` before creating this row. If we know the primary key value, then we can use its value as `fkmid` when inserting a row into table `n`. This way, get the primary key of that new row in Table `n`. Then we use this primary key as `fknid` and actually insert the row into table `m`.

This can be done if we generate the primary key for table `m` more explicitly than before. So far, we would write something like `mid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY`. This means that the column is a primary key is of type `INT`. It is `GENERATED BY DEFAULT`, which means that its values are generated automatically *unless specified* [175]. Thus, when inserting rows into the table, we could specify a value for `mid`, which is then used, or we do not specify a value, in which case one will be generated for us automatically. The `AS IDENTITY` means that, when a value is generated, it is taken from an implicit sequence [218].

Instead of using an implicit sequence without name, we also create a sequence by ourselves. This can be done with the command `CREATE SEQUENCE` [106]. We create a sequence `sqmid` to generate the primary key values for the table `m` in Listing 19.76 by writing `CREATE SEQUENCE sqmid AS INT;`. The values in such sequences will always be strictly monotonously increasing. The next value of `sqmid` would be obtained atomically by `NEXTVAL('sqmid')` [388]. We now define the primary key of table `m` as `mid INT DEFAULT NEXTVAL('sqmid') PRIMARY KEY`. This definition says that the value of the primary key `mid` can be provided when creating rows in table `m`. If it is *not* provided, then a `DEFAULT` value will be used [132]. This default value is computed by evaluating the expression `NEXTVAL('sqmid')`. It will

Listing 19.76: The realization of a M +---+ N conceptual relationship. (stored in file `MN_tables.sql`; output in Listing 19.77)

```

1  /* Create the tables for a M-+---+<-N relationship. */
2
3  -- The sequence for the primary keys of the rows in m.
4  CREATE SEQUENCE sqmid AS INT;
5
6  -- Table M: Each row in M is related to one or multiple rows in N.
7  CREATE TABLE m (
8      mid    INT DEFAULT NEXTVAL('sqmid') PRIMARY KEY,   -- Use ID sequence!
9      fknid INT NOT NULL UNIQUE,   -- UNIQUE: Each N is related to one M.
10     x      CHAR(3)           -- example of other attributes
11 );
12
13 -- Table N: Each row in N is related exactly one row in M.
14 CREATE TABLE n (
15     nid    INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
16     fkmid INT NOT NULL REFERENCES m (mid),
17     y      CHAR(2),           -- example of other attributes
18     UNIQUE (nid, fkmid)    -- needed for m_mid_fknid_fk
19 );
20
21 -- To table M, we add the foreign key reference constraint towards
22 -- table N. This enforces the mandatory-many part of the relationship.
23 ALTER TABLE m ADD CONSTRAINT m_fknid_mid_fk FOREIGN KEY (fknid, mid)
24     REFERENCES n (nid, fkmid);

```

Listing 19.77: The stdout of the program `MN_tables.sql` given in Listing 19.76.

```

1 $ psql "postgres://postgres:XXX@localhost/relationships" -v ON_ERROR_STOP=1
2   ↪ -ebs MN_tables.sql
3 CREATE SEQUENCE
4 CREATE TABLE
5 CREATE TABLE
6 ALTER TABLE
7 # psql 16.12 succeeded with exit code 0.

```

be the ever-increasing output of the sequence `sqmid`.

From a mechanical point of view, this works more or less like `mid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY`. The difference is that we now can also generate valid values for `mid` without creating rows in table `m`. Of course we can use `NEXTVAL('sqmid')` in any SQL query, `SELECT`, `INSERT`, `UPDATE` – whatever we want. And whenever `NEXTVAL('sqmid')` is invoked, it will step the sequence `sqmid`. It will never return the same value (unless you evaluate it 2^{64} times). This means that we can generate a valid primary key value for table `m`, then insert a row into table `n` using this value as foreign key, and then insert a row into table `m` actually using it as primary key.

In Listing 19.78, we thus insert a pair of rows into tables `m` and `n` as follows: First, we generate the primary key value for the row that we want to insert into table `m` as CTE that just invokes `NEXTVAL('sqmid')` and remembers the result, i.e., `m_id AS (SELECT NEXTVAL('sqmid') AS new_mid)`. Then we insert a row into table `n`, also as CTE, by doing `new_n AS (INSERT INTO n (y, fkmid) SELECT 'AB', new_mid FROM m_id RETURNING nid, fkmid)`. This CTE returns the primary key `nid` of the new row in table `n` as well as the previously generated primary key `new_mid` for the row that we will insert into table `m` (as `fkmid`). We use both values in to finally actually insert the row into table `m` by calling `INSERT INTO m (mid, x, fknid) SELECT fkmid, '123', nid FROM new_n;`.

After this, we want to append another row to table `n` and link it to the now existing row in table `m`. This can be done more easily via `INSERT INTO n (y, fkmid) VALUES ('CD', 1);`. The creation of new rows for table `m`, however, requires us to have the two CTEs. The contents of the tables `m` and `n` after

inserting the data are shown in [Figure 19.17](#). Reassembling the data from both tables can be achieved with a single `INNER JOIN`, as illustrated at the bottom of [Listing 19.78](#).

This solution requires several PostgreSQL-specific techniques, such as `RETURNING`, `NEXTVAL`, and the creation of sequences. Other DBMSes certainly have similar or slightly different extensions to the SQL standard that allow us to do similar things. In [Listings 19.80](#) to [19.89](#) we perform similar sanity tests as back in [Section 19.2.2.4](#). We confirm that our constraints protect the referential integrity of the data.

Listing 19.78: Inserting into and selecting data from the realization of an $M \leftrightarrow N$ conceptual relationship given in Listing 19.76. (stored in file `MN_insert_and_select.sql`; output in Listing 19.79)

```

1  /* Inserting data into the tables for the M-| |-----|<-n relationship. */
2
3  -- Insert into M and N at the same time.
4  -- This creates M entry with id 1, and N entry with id 1, and
5  -- relationship (1, 1).
6  WITH m_id AS (SELECT NEXTVAL('sqmid') AS new_mid),
7      new_n AS (INSERT INTO n (y, fkmid)
8          SELECT 'AB', new_mid FROM m_id RETURNING nid, fkmid)
9      INSERT INTO m (mid, x, fknid) SELECT fkmid, '123', nid FROM new_n;
10
11 -- Create a new row in N referencing an existing row in M.
12 INSERT INTO n (y, fkmid) VALUES ('CD', 1);
13
14 -- Insert into M and N at the same time.
15 WITH m_id AS (SELECT NEXTVAL('sqmid') AS new_mid),
16      new_n AS (INSERT INTO n (y, fkmid)
17          SELECT 'EF', new_mid FROM m_id RETURNING nid, fkmid)
18      INSERT INTO m (mid, x, fknid) SELECT fkmid, '456', nid FROM new_n;
19
20 -- Insert into M and N at the same time.
21 WITH m_id AS (SELECT NEXTVAL('sqmid') AS new_mid),
22      new_n AS (INSERT INTO n (y, fkmid)
23          SELECT 'GH', new_mid FROM m_id RETURNING nid, fkmid)
24      INSERT INTO m (mid, x, fknid) SELECT fkmid, '789', nid FROM new_n;
25
26 -- Create a new row in N referencing an existing row in M.
27 INSERT INTO n (y, fkmid) VALUES ('IJ', 1);
28
29 -- Combine the rows from M and N.
30 SELECT mid, x, nid, y FROM n INNER JOIN m ON n.fkmid = m.mid;

```

Listing 19.79: The std::cout of the program `MN_insert_and_select.sql` given in Listing 19.78.

```
1 $ psql "postgres://postgres:XXX@localhost/relationships" -v ON_ERROR_STOP=1
2   ↳ -ebf MN_insert_and_select.sql
3 INSERT 0 1
4 INSERT 0 1
5 INSERT 0 1
6 INSERT 0 1
7 INSERT 0 1
8   mid | x | nid | y
9   ----+---+----+---
10  1 | 123 | 1 | AB
11  1 | 123 | 2 | CD
12  2 | 456 | 3 | EF
13  3 | 789 | 4 | GH
14  1 | 123 | 5 | IJ
15
16 # psql 16.12 succeeded with exit code 0.
```

Table m			Table n		
mid	fknid	x	nid	fkmid	y
1	1	"123"	1	1	"AB"
2	3	"456"	2	1	"CD"
3	4	"789"	3	2	"EF"
			4	3	"GH"
			5	1	"IJ"

Figure 19.17: The contents of the two tables in the implementation of the M \bowtie N conceptual relationship after executing Listing 19.78.

Listing 19.80: Trying to create a row into table **m** that is not related to any row in table **n** is not possible. (stored in file `MN_insert_error_1.sql`; output in Listing 19.81)

```

1 /* Can we create a row in M unrelated to any row in N? */
2
3 INSERT INTO m (x) VALUES ('777');
```

Listing 19.81: The stdout of the program `MN_insert_error_1.sql` given in Listing 19.80.

```

1 $ psql "postgres://postgres:XXX@localhost/relationships" -v ON_ERROR_STOP=1
2   ↪ -ebf MN_insert_error_1.sql
3 psql:conceptualToRelational/MN_insert_error_1.sql:3: ERROR:  null value in
4   ↪ column "fknid" of relation "m" violates not-null constraint
5 DETAIL:  Failing row contains (4, null, 777).
6 psql:conceptualToRelational/MN_insert_error_1.sql:3: STATEMENT:  /* Can we
7   ↪ create a row in M unrelated to any row in N? */
8 INSERT INTO m (x) VALUES ('777');
9 # psql 16.12 failed with exit code 3.
```

Listing 19.82: Trying to create a row into table **n** that is not related to any row in table **m** is not possible. (stored in file `MN_insert_error_2.sql`; output in Listing 19.83)

```

1 /* Can we create a row in N unrelated to any row in M? */
2
3 INSERT INTO n (y) VALUES ('ZZ');
```

Listing 19.83: The stdout of the program `MN_insert_error_2.sql` given in Listing 19.82.

```

1 $ psql "postgres://postgres:XXX@localhost/relationships" -v ON_ERROR_STOP=1
2   ↪ -ebf MN_insert_error_2.sql
3 psql:conceptualToRelational/MN_insert_error_2.sql:3: ERROR:  null value in
4   ↪ column "fkmid" of relation "n" violates not-null constraint
5 DETAIL:  Failing row contains (6, null, ZZ).
6 psql:conceptualToRelational/MN_insert_error_2.sql:3: STATEMENT:  /* Can we
7   ↪ create a row in N unrelated to any row in M? */
8 INSERT INTO n (y) VALUES ('ZZ');
9 # psql 16.12 failed with exit code 3.
```

Listing 19.84: Trying to create a row in table `m` that references a row in table `n` which is referencing another row in table `m` is not possible. (stored in file `MN_insert_error_3.sql`; output in Listing 19.85)

```

1  /* Can we insert a M that is related to a N related to another M? */
2
3  -- N with id 4 is already related to M with id 3.
4  -- Can we make our new M row point to it as its "primary N" anyway?
5  INSERT INTO m (fknid, x) VALUES (4, '888');
```

Listing 19.85: The stdout of the program `MN_insert_error_3.sql` given in Listing 19.84.

```

1  $ psql "postgres://postgres:XXX@localhost/relationships" -v ON_ERROR_STOP=1
2    ↪ -ebf MN_insert_error_3.sql
3  psql:conceptualToRelational/MN_insert_error_3.sql:5: ERROR:  duplicate key
4    ↪ value violates unique constraint "m_fknid_key"
3  DETAIL:  Key (fknid)=(4) already exists.
4  psql:conceptualToRelational/MN_insert_error_3.sql:5: STATEMENT:  /* Can we
5    ↪ insert a M that is related to a N related to another M? */
5  -- N with id 4 is already related to M with id 3.
6  -- Can we make our new M row point to it as its "primary N" anyway?
7  INSERT INTO m (fknid, x) VALUES (4, '888');
8  # psql 16.12 failed with exit code 3.
```

Listing 19.86: Trying the same thing as in Listing 19.84, but this time also attempting to re-adjusting the row in `n` with an `UPDATE` instruction to make reference the new row in table `m`, is also not possible. (stored in file `MN_insert_error_4.sql`; output in Listing 19.87)

```

1  /* Can we insert a M that is related to a N related to another M? */
2
3  -- N with id 4 is already related to M with id 3.
4  -- Can we make it point to the new M row instead?
5  WITH m_new AS (INSERT INTO m (fknid, x) VALUES (4, '555')
6    RETURNING mid, fknid)
7    UPDATE n SET fkmid = m_new.mid FROM m_new WHERE nid = fknid;
```

Listing 19.87: The stdout of the program `MN_insert_error_4.sql` given in Listing 19.86.

```

1  $ psql "postgres://postgres:XXX@localhost/relationships" -v ON_ERROR_STOP=1
2    ↪ -ebf MN_insert_error_4.sql
2  psql:conceptualToRelational/MN_insert_error_4.sql:7: ERROR:  duplicate key
3    ↪ value violates unique constraint "m_fknid_key"
3  DETAIL:  Key (fknid)=(4) already exists.
4  psql:conceptualToRelational/MN_insert_error_4.sql:7: STATEMENT:  /* Can we
5    ↪ insert a M that is related to a N related to another M? */
5  -- N with id 4 is already related to M with id 3.
6  -- Can we make it point to the new M row instead?
7  WITH m_new AS (INSERT INTO m (fknid, x) VALUES (4, '555')
8    RETURNING mid, fknid)
9    UPDATE n SET fkmid = m_new.mid FROM m_new WHERE nid = fknid;
10 # psql 16.12 failed with exit code 3.
```

Listing 19.88: Changing a row in table `n` that references a row in table `m` to now reference another row in table `m` is not possible. (stored in file `MN_insert_error_5.sql`; output in Listing 19.89)

```

1 /* Can we change the relationship of a N away from its primary M? */
2
3 -- N with id 1 is used as "primary N" for M with ID 1.
4 -- Can we make it point to another M?
5 UPDATE n SET fkmid = 3 WHERE nid = 1;

```

Listing 19.89: The stdout of the program `MN_insert_error_5.sql` given in Listing 19.88.

```

1 $ psql "postgres://postgres:XXX@localhost/relationships" -v ON_ERROR_STOP=1
2   ↪ -ebf MN_insert_error_5.sql
3 psql:conceptualToRelational/MN_insert_error_5.sql:5: ERROR: update or
4   ↪ delete on table "n" violates foreign key constraint "m_fknid_mid_fk"
4   ↪ on table "m"
5 DETAIL: Key (nid, fkmid)=(1, 1) is still referenced from table "m".
6 psql:conceptualToRelational/MN_insert_error_5.sql:5: STATEMENT: /* Can we
6   ↪ change the relationship of a N away from its primary M? */
7 -- N with id 1 is used as "primary N" for M with ID 1.
8 -- Can we make it point to another M?
7 UPDATE n SET fkmid = 3 WHERE nid = 1;
8 # psql 16.12 failed with exit code 3.

```

19.2.2.8 $O \rightarrowtail \leftarrow P$

We have the two entity types O and P. Each entity of type O may be connected to zero, one, or multiple entities of type P. Each entity of type P may be connected to zero, one, or multiple entities of type O.

We encountered this pattern back in [Figure 18.15](#) when we modeled the relationship between the different types of modules and the different types of teacher position. For example, maybe a lecturer can propose and chair a module of type elective specialization course. The mandatory core module may only be chaired by a full professor, who is also permitted to chair elective specialization course. The Master's (thesis module) supervision may only be available to teachers that hold the position Master's supervisor. This means that each position may give a faculty member the credentials to chair zero, one, or many different types of modules. At the same time, a module type may be chaired by teachers belonging to zero, one, or many position types, as illustrated in [Figure 19.18](#). Granted, the situation where a module type cannot be chaired by any position type would be very strange. This maybe a shortcoming of our model back then...

When implementing this pattern, we need a table `o` for the entities of type O. The primary key of this table be `oid` and there also will be the example attribute `x`. We also need a table `p` for the entities of type P, which gets the primary key `pid` and the example attribute `y`. Now, since each row in table `o` can be related to multiple rows in table `p` and vice versa, no two-table solution is possible. This time, there is no way around it: We need three tables. But this three-table approach will look pretty much like the one in [Section 19.2.2.1](#) for the A $\rightarrowtail \leftarrow B$, but with different `UNIQUE` constraints.

In [Listing 19.90](#), we add the table `relate_o_and_p` which has two columns, `fkoid` and `fpid`. These are foreign keys pointing to the primary keys `oid` and `pid` of tables `o` and `p`, respectively. This is ensured with corresponding `REFERENCES` constraints.

Both columns are marked as `NOT NULL`, because we only store valid O-P pairs. Neither of them is `UNIQUE`, because each row of table `o` can be related to multiple rows of table `p` and vice versa. In the previous three-table solutions, we always had at least one `UNIQUE` foreign key column. That made sure that each pair of foreign key values could occur at most once. Since we cannot have a single `UNIQUE` column this time, it would now be possible that that the same `(fkoid, fpid)` values could occur multiple times. Obviously, we cannot have that. We must ensure that a pair `(fkoid, fpid)` can appear at most once in the table. Two specific rows in tables `o` and `p` can, of course, be related only once to each other.

We could enforce this with a constraint `UNIQUE (fkoid, fpid)`. However, the right solution here is to go directly for `PRIMARY KEY (fkoid, fpid)`. Our table needs a primary key, and here, the only possible primary key are the pairs of `(fkoid, fpid)`. Primary keys must be unique by default, so this constraint also covers the uniqueness.

In [Listing 19.92](#), we now insert data into the two tables. Since both relationship ends are optional, we can enter records in the two data tables arbitrary order. We first enter some data into the tables `o` and `p`. Then we can add the relationships between the rows of these tables by inserting

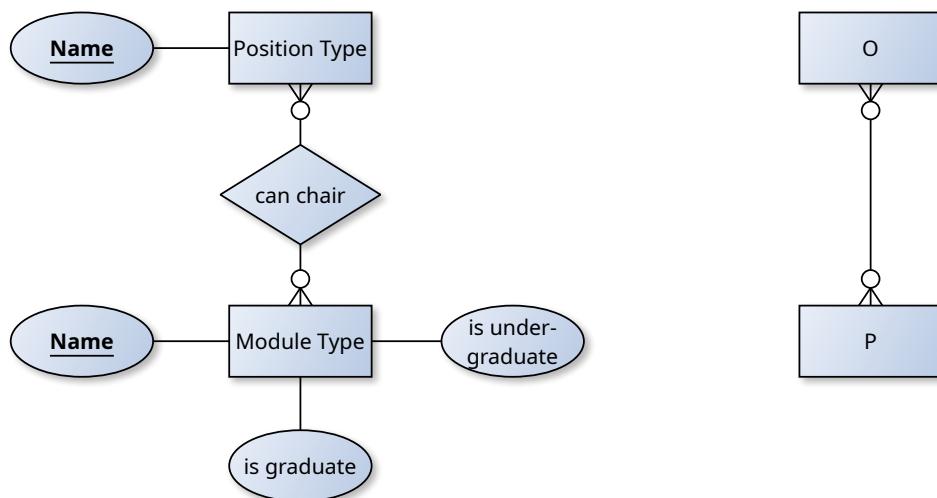


Figure 19.18: We encountered the $O \rightarrowtail \leftarrow P$ relationship pattern in [Figure 18.15](#).

Listing 19.90: The realization of a $O \rightarrowtail P$ conceptual relationship. (stored in file `OP_tables.sql`; output in Listing 19.91)

```

1  /* Create the tables for an O->----o<-P relationship. */
2
3  -- Table O: Each row in O is related to zero or one or many rows in P.
4  CREATE TABLE o (
5      oid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
6      x    CHAR(3)   -- example of other attributes
7  );
8
9  -- Table P: Each row in P is related to zero or one or many rows in O.
10 CREATE TABLE p (
11     pid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
12     y    CHAR(2)   -- example of other attributes
13 );
14
15 -- The table for managing the relationship between O and P.
16 CREATE TABLE relate_o_and_p (
17     fkoid INT NOT NULL REFERENCES o (oid),
18     fkpid INT NOT NULL REFERENCES p (pid),
19     PRIMARY KEY (fkoid, fkpid)  -- Primary key includes both columns.
20 );
```

Listing 19.91: The stdout of the program `OP_tables.sql` given in Listing 19.90.

```

1  $ psql "postgres://postgres:XXX@localhost/relationships" -v ON_ERROR_STOP=1
2  ↪ -ebf OP_tables.sql
3  CREATE TABLE
4  CREATE TABLE
5  CREATE TABLE
# psql 16.12 succeeded with exit code 0.
```

rows into table `relate_o_and_p`. The contents of all three tables after inserting the data are displayed in Figure 19.19. If we want to recombine data from the two tables, we can do this with two `INNER JOIN` expressions.

During the above example, we inserted the row `(1, 1)` into table `relate_o_and_p`. This row establishes that the row with primary key `oid = 1` of table `o` is related to the row with primary key `pid = 1` in table `p`. In Listing 19.94, we try inserting the row again into `relate_o_and_p`. Of course, no two rows can be related twice. It would make, for example, no sense to assign the same address twice to the same person. This would violate that rows must be unique, a basic requirement of the relational data model. If that was possible, these rows would also appear twice in the result of the `INNER JOIN`s when merging the data. So this should not be possible. And it is not possible. Thanks to the `PRIMARY KEY` constraint that we attached to table `relate_o_and_p`, this insertion fails.

Listing 19.92: Inserting into and selecting data from the realization of an O \rightarrowtail O \leftarrowtail P conceptual relationship given in Listing 19.90. (stored in file `OP_insert_and_select.sql`; output in Listing 19.93)

```

1  /* Inserting data into the tables for the O->----o<-P relationship. */
2
3  -- Insert some rows into the table for entity type O.
4  INSERT INTO o (x) VALUES ('123'), ('456'), ('789'), ('101');
5
6  -- Insert some rows into the table for entity type P.
7  INSERT INTO p (y) VALUES ('AB'), ('CD'), ('EF'), ('GH');
8
9  -- Create some relationships between O and P.
10 INSERT INTO relate_o_and_p (fkoid, fkpid) VALUES
11     (1, 1), (1, 2), (2, 2), (4, 1), (3, 2);
12
13 -- Combine the rows from O and P.
14 SELECT oid, x, pid, y FROM relate_o_and_p
15   INNER JOIN o ON o.oid = relate_o_and_p.fkoid
16   INNER JOIN p ON p.pid = relate_o_and_p.fkpid;

```

Listing 19.93: The stdout of the program `OP_insert_and_select.sql` given in Listing 19.92.

```

1  $ psql "postgres://postgres:XXX@localhost/relationships" -v ON_ERROR_STOP=1
2  ↪ -ebs OP_insert_and_select.sql
3
4  INSERT 0 4
5  INSERT 0 4
6  INSERT 0 5
7  oid | x   | pid | y
8  ----+---+-----+
9  1   | 123 | 1   | AB
10 1   | 123 | 2   | CD
11 2   | 456 | 2   | CD
12 4   | 101 | 1   | AB
13 3   | 789 | 2   | CD
14 (5 rows)
15
16 # psql 16.12 succeeded with exit code 0.

```

Table o		Table p		Table relate_o_and_p	
oid	x	pid	y	fkoid	fkpid
1	"123"	1	"AB"	1	1
2	"456"	2	"CD"	1	2
3	"789"	3	"EF"	2	2
4	"101"	4	"GH"	4	1
				3	2

Figure 19.19: The contents of the three tables in the implementation of the O \rightarrowtail O \leftarrowtail P conceptual relationship after executing Listing 19.92.

Listing 19.94: Trying to relate two entities twice, which is of course not permitted in *any* relationship pattern. (stored in file `OP_insert_error.sql`; output in Listing 19.95)

```
1 /* Try to create a relationship that already exists. */
2
3 -- The relationship (1, 1) already exists, so this will fail.
4 INSERT INTO relate_o_and_p (fkoid, fkpid) VALUES (1, 1);
```

Listing 19.95: The stdout of the program `OP_insert_error.sql` given in Listing 19.94.

```
1 $ psql "postgres://postgres:XXX@localhost/relationships" -v ON_ERROR_STOP=1
2   ↪ -ebf OP_insert_error.sql
3 psql:conceptualToRelational/OP_insert_error.sql:4: ERROR:  duplicate key
4   ↪ value violates unique constraint "relate_o_and_p_pkey"
5 DETAIL:  Key (fkoid, fkpid)=(1, 1) already exists.
6 psql:conceptualToRelational/OP_insert_error.sql:4: STATEMENT:  /* Try to
7   ↪ create a relationship that already exists. */
8 -- The relationship (1, 1) already exists, so this will fail.
9 INSERT INTO relate_o_and_p (fkoid, fkpid) VALUES (1, 1);
# psql 16.12 failed with exit code 3.
```

19.2.2.9 $Q \rightarrowtail \leftarrow R$

We have the two entity types Q and R. Each entity of type Q may be connected to zero, one, or multiple entities of type R. Each entity of type R must be connected to at least one entity of type P, but can be connected to many.

We came across this situation when modeling the messaging subsystem of our teaching management platform back in Figure 18.16. A message must have at least one receiver, but it could also be sent to multiple people at once. Each person may receive zero, one, or many messages. This situation is sketched in Figure 19.20.

Let us again begin by establishing the basic tables that we need. Then we sort out how to enforce

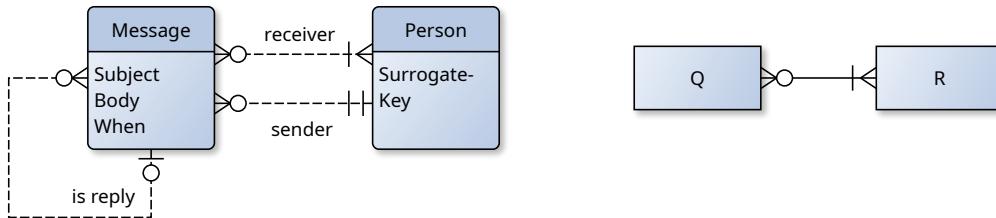


Figure 19.20: We encountered the $Q \rightarrowtailorleft; R$ relationship pattern in Figure 18.16.

Listing 19.96: The realization of a $Q \rightarrowtailorleft; R$ conceptual relationship. (stored in file `QR_tables.sql`; output in Listing 19.97)

```

1  /* Create the tables for a Q->0-----|<-R relationship. */
2
3  -- Table Q: Each row in Q is related to one or multiple rows in R.
4  CREATE TABLE q (
5      qid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
6      fkrid INT NOT NULL, -- later used to reference R via relate_q_and_r
7      x CHAR(3)           -- example for other attributes
8  );
9
10 -- Table R: Each row in R is related to zero, one, or many rows in Q.
11 CREATE TABLE r (
12     rid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
13     y CHAR(2) -- example for other attributes
14 );
15
16 -- The table for managing the relationship between Q and R.
17 CREATE TABLE relate_q_and_r (
18     fkqid INT NOT NULL REFERENCES q (qid),
19     fkrid INT NOT NULL REFERENCES r (rid),
20     PRIMARY KEY (fkqid, fkrid) -- Primary key includes both columns.
21 );
22
23 -- To table Q, we add the foreign key reference constraint towards
24 -- table relate_q_and_r. This enforces that one relation must exist.
25 ALTER TABLE q ADD CONSTRAINT q_qid_fkrid_fk FOREIGN KEY (qid, fkrid)
26   REFERENCES relate_q_and_r (fkqid, fkrid);
  
```

Listing 19.97: The stdout of the program `QR_tables.sql` given in Listing 19.96.

```

1 $ psql "postgres://postgres:XXX@localhost/relationships" -v ON_ERROR_STOP=1
2   ↵ -ebf QR_tables.sql
3 CREATE TABLE
4 CREATE TABLE
5 CREATE TABLE
6 ALTER TABLE
6 # psql 16.12 succeeded with exit code 0.
  
```

the referential integrity with proper constraints. We need a table `q` for the entities of type Q. The primary key of this table be `qid` and there also will be the example attribute `x`. We also need a table `r` for the entities of type R. It gets the primary key `rid` and the example attribute `y`. Since both ends of the relationship allow for a row to be connected to multiple rows in the other table, we definitely will need three tables.

We create another table `relate_q_and_r` to manage the relationships in Listing 19.96. This table has two columns, `fkqid` and `fkruid`, which are foreign keys pointing to the primary keys `qid` and `rid` of tables `q` and `r`, respectively. This is ensured with corresponding `REFERENCES` constraints. Both columns are marked as `NOT NULL`. Like in Section 19.2.2.8 ($O \rightarrow\!\!\!-\!\!-\!< P$), each pair `(fkqid, fkruid)` can appear only once in the table. This is because two specific rows in tables `q` and `r` can, of course, be related only once to each other. This is implemented via the constraint `PRIMARY KEY (fkqid, fkruid)`.

Compared to $O \rightarrow\!\!\!-\!\!-\!< P$, the problem that we need to solve is that we need to enforce that each row in table `q` *must be* related to (at least) one row in table `r`. Back in Section 19.2.2.4 ($G \rightarrow\!\!\!-\!\!-\!< H$), we faced a similar problem, i.e., “*How can we make sure that each entity of type G is connected to at least one entity of type H?*” The main problem there was not to make a new row in table `g` reference some row in table `h`, but that we then must make that row in table `h` reference back the same row in table `g`. We solved this by creating a constraint that enforced that the primary key of that row in table `h` together with the foreign key to the table `g` would need to be the same as the foreign key of the row in table `g` and its primary key.

Now our relationships are managed in the table `relate_q_and_r`. What we need to do is to make sure that, if we insert a row into table `q`, we must also – at the same time – insert a row into table `relate_q_and_r` that will link this row to a row in table `r`. We again store a “first R” foreign key to a row in `r` as column `fkruid` in table `q`. Then we add the constraint `q_qid_fkruid_fk` to table `q`. It enforces `FOREIGN KEY (qid, fkruid) REFERENCES relate_q_and_r (fkqid, fkruid)`.

In Listing 19.98, we now insert data into the two tables. For the entities of type R, the connection to entities of type Q is optional. Therefore, we can populate the table `r` first.

If we want to insert a row into table `q`, however, we must at the same time also insert a row into table `relate_q_and_r`. When we do this, the new row in table `relate_q_and_r` must use the primary key value of the new row in table `q` as foreign key. This means that we will again use a CTE, which we will call `q_new`. We define it as `INSERT INTO q (x, fkruid)VALUES ('123', 1)RETURNING qid, fkruid`. Here, we want to insert a row into table `q` whose `x` attribute has value `'123'` and that is related to the entry with primary key `1` in table `r`. Once this insert is executed, it will return the primary key of the new row (as `qid`) and the foreign key to the row in table `r`, which is named `fkruid` and obviously has value `1`.

This CTE is then used to insert a new row into table `relate_q_and_r`. We write `INSERT INTO relate_q_and_r (fkqid, fkruid)SELECT qid, fkruid FROM q_new;`. This is pretty straightforward: The new row in table `relate_q_and_r` gets the primary key value pointing to the new row in table `q`. Additionally, it gets the primary key value of the row in table `r` that it references. Both insertions are executed together as one single command and the referential integrity constraints are checked at its end. And it fits.

We insert a few rows into tables `q` and `relate_q_and_r` using this method. Once this is done, we can easily add more connections as rows in table `relate_q_and_r`. The contents of all three tables after we inserted the data are displayed in Figure 19.21. Finally, we can merge the data again using the same two `INNER JOIN` expressions we used several times before.

It is clear that we cannot insert any row into table `q` that does not provide a foreign key to a row in table `r`, as shown in Listing 19.100. Neither can we insert a row that does provide such a foreign key, but for which now row in table `relate_q_and_r` exists, as shown in Listing 19.102. Finally, Listing 19.104 shows that we can also not first insert a row into table `relate_q_and_r` and then create the corresponding row in table `q`, even if we know the right primary key for that second new row. In other words, our constraints properly protect the referential integrity of the relationship pattern.

Listing 19.98: Inserting into and selecting data from the realization of an $Q \rightarrowtail R$ conceptual relationship given in Listing 19.96. (stored in file `QR_insert_and_select.sql`; output in Listing 19.99)

```

1  /* Inserting data into the tables for the Q->-----|<-R relationship. */
2
3  -- Insert some rows into the table for entity type R first.
4  -- We can only create rows in Q related to existing R entities.
5  INSERT INTO r (y) VALUES ('AB'), ('CD'), ('EF'), ('GH'), ('IJ'), ('KL');
6
7  -- Insert into Q and relate_q_and_r at the same time.
8  -- This creates Q entry with id 1 and relationship (1, 1).
9  WITH q_new AS (INSERT INTO q (x, fkrid) VALUES ('123', 1)
10    RETURNING qid, fkrid)
11  INSERT INTO relate_q_and_r (fkqid, fkrid)
12    SELECT qid, fkrid FROM q_new;
13
14  -- Insert into Q and relate_q_and_r at the same time.
15  -- This creates Q entry with id 2 and relationship (2, 4).
16  WITH q_new AS (INSERT INTO q (x, fkrid) VALUES ('456', 4)
17    RETURNING qid, fkrid)
18  INSERT INTO relate_q_and_r (fkqid, fkrid)
19    SELECT qid, fkrid FROM q_new;
20
21  -- Insert into Q and relate_q_and_r at the same time.
22  -- This creates Q entry with id 2 and relationship (3, 1).
23  WITH q_new AS (INSERT INTO q (x, fkrid) VALUES ('789', 1)
24    RETURNING qid, fkrid)
25  INSERT INTO relate_q_and_r (fkqid, fkrid)
26    SELECT qid, fkrid FROM q_new;
27
28  -- We can now insert additional relationships
29  INSERT INTO relate_q_and_r VALUES (1, 2), (2, 3), (2, 5);
30
31  -- Combine the rows from Q and R. This needs two INNER JOINS.
32  SELECT qid, x, rid, y FROM relate_q_and_r
33    INNER JOIN q ON q.qid = relate_q_and_r.fkqid
34    INNER JOIN r ON r.rid = relate_q_and_r.fkrid;

```

Listing 19.99: The std::cout of the program `QR_insert_and_select.sql` given in Listing 19.98.

```
1 $ psql "postgres://postgres:XXX@localhost/relationships" -v ON_ERROR_STOP=1
2   ↵ -ebf QR_insert_and_select.sql
3
4 INSERT 0 6
5 INSERT 0 1
6 INSERT 0 1
7 INSERT 0 1
8 INSERT 0 3
9
10    qid | x   | rid | y
11    ----+----+----+----+
12      1 | 123 |    1 | AB
13      2 | 456 |    4 | GH
14      3 | 789 |    1 | AB
15      1 | 123 |    2 | CD
16      2 | 456 |    3 | EF
17      2 | 456 |    5 | IJ
18
19 (6 rows)
20
21
22 # psql 16.12 succeeded with exit code 0.
```

Table <u>q</u>		Table <u>r</u>	Table <u>relate_q_and_r</u>
qid	fkruid	rid	fkruid
1	1	"AB"	1
2	4	"CD"	2
3	1	"EF"	3
		"GH"	1
		"IJ"	2
		"KL"	2
			3
			5

Figure 19.21: The contents of the three tables in the implementation of the $Q \rightarrowtail R$ conceptual relationship after executing Listing 19.98.

Listing 19.100: Trying to create a row into table q without providing a foreign key to a row in r fails. (stored in file QR_insert_error_1.sql; output in Listing 19.101)

```

1 /* Can we create a row in Q unrelated to any row in R? */
2
3 INSERT INTO q (x) VALUES ('777');
```

Listing 19.101: The stdout of the program QR_insert_error_1.sql given in Listing 19.100.

```

1 $ psql "postgres://postgres:XXX@localhost/relationships" -v ON_ERROR_STOP=1
   ↵ -ebf QR_insert_error_1.sql
2 psql:conceptualToRelational/QR_insert_error_1.sql:3: ERROR:  null value in
   ↵ column "fkruid" of relation "q" violates not-null constraint
3 DETAIL:  Failing row contains (4, null, 777).
4 psql:conceptualToRelational/QR_insert_error_1.sql:3: STATEMENT:  /* Can we
   ↵ create a row in Q unrelated to any row in R? */
5 INSERT INTO q (x) VALUES ('777');
6 # psql 16.12 failed with exit code 3.
```

Listing 19.102: Trying to create a row into table q with creating a corresponding row in table relate_q_and_r also fails. (stored in file QR_insert_error_2.sql; output in Listing 19.103)

```

1 /* Can we create a row in Q ignoring `relate_q_and_r` table? */
2
3 INSERT INTO q (fkruid, x) VALUES (6, '888');
```

Listing 19.103: The stdout of the program QR_insert_error_2.sql given in Listing 19.102.

```

1 $ psql "postgres://postgres:XXX@localhost/relationships" -v ON_ERROR_STOP=1
   ↵ -ebf QR_insert_error_2.sql
2 psql:conceptualToRelational/QR_insert_error_2.sql:3: ERROR:  insert or
   ↵ update on table "q" violates foreign key constraint "q_qid_fkruid_fk"
3 DETAIL:  Key (qid, fkruid)=(5, 6) is not present in table "relate_q_and_r".
4 psql:conceptualToRelational/QR_insert_error_2.sql:3: STATEMENT:  /* Can we
   ↵ create a row in Q ignoring `relate_q_and_r` table? */
5 INSERT INTO q (fkruid, x) VALUES (6, '888');
6 # psql 16.12 failed with exit code 3.
```

Listing 19.104: Even if we can choose the primary key for the new row in table `q`, we cannot create the associated row in table `relate_q_and_r` first. (stored in file `QR_insert_error_3.sql`; output in Listing 19.105)

```

1  /* Can we create the row in `relate_q_and_r` first? */
2
3  INSERT INTO relate_q_and_r VALUES (10, 4);
4  INSERT INTO q (qid, fkrid, x) VALUES (10, 4, '999');

```

Listing 19.105: The stdout of the program `QR_insert_error_3.sql` given in Listing 19.104.

```

1 $ psql "postgres://postgres:XXX@localhost/relationships" -v ON_ERROR_STOP=1
   ↪ -ebf QR_insert_error_3.sql
2 psql:conceptualToRelational/QR_insert_error_3.sql:3: ERROR:  insert or
   ↪ update on table "relate_q_and_r" violates foreign key constraint "
   ↪ relate_q_and_r_fkqid_fkey"
3 DETAIL:  Key (fkqid)=(10) is not present in table "q".
4 psql:conceptualToRelational/QR_insert_error_3.sql:3: STATEMENT:  /* Can we
   ↪ create the row in `relate_q_and_r` first? */
5 INSERT INTO relate_q_and_r VALUES (10, 4);
6 # psql 16.12 failed with exit code 3.

```

19.2.2.10 $S \rightarrow\!\!\!-\!\!\!- T$

We have the two entity types S and T. Each entity of type S must be connected to at least one entity of type T, but can be connected to many. Each entity of type T must be connected to at least one entity of type S, but can be connected to many.

We encountered this relationship pattern back in [Figure 18.20](#) when constructing the overall model of our teaching management platform. Back then, we related address records to person entities. Each person must have at least one address, but may have more than one address. Each address stored in our system must be associated with at least one person, but it is possible that multiple people live at the same place. This part of the model is sketched in [Figure 19.22](#).

To implement this relationship pattern, we will combine what we learned when implementing the M $\rightarrow\!\!\!-\!\!\!- N$ pattern in [Section 19.2.2.7](#) with the method for implementing the Q $\rightarrow\!\!\!-\!\!\!- R$ pattern in [Section 19.2.2.9](#).

We first again create the basic tables that we are definitely going to need in [Listing 19.106](#). We need a table `s` for the entities of type S. The primary key of this table be `sid` and there also will be the example attribute `x`. We also need a table `t` for the entities of type T. It gets the primary key `tid` and the example attribute `y`.

We will also definitely need a third table to manage the relationships. We call it `relate_s_and_t`. This table has two columns, `fksid` and `fktid`, which are foreign keys pointing to the primary keys `sid` and `tid` of tables `s` and `t`, respectively. This is ensured with corresponding `REFERENCES` constraints. Both columns also are marked as `NOT NULL`, because neither value can be omitted in a row. Like in [Section 19.2.2.8](#) ($O \rightarrow\!\!\!-\!\!\!- P$), each pair `(fksid, fktid)` can appear only once in the table. This is because two specific rows in tables `s` and `t` can, of course, be related only once to each other. This is implemented via the constraint `PRIMARY KEY (fksid, fktid)`.

Like in the M $\rightarrow\!\!\!-\!\!\!- N$ pattern, both relationship ends are mandatory. We cannot create a row in table `s` without relating it to an existing row in table `t`. We also cannot create a row in table `t` without relating it to an existing row in table `s`. Back in [Section 19.2.2.7](#) (M $\rightarrow\!\!\!-\!\!\!- N$), we solved this chicken-and-egg problem by creating a sequence from which we could then generate values for the primary key of the table `m`. We used this sequence to first create a primary key value for table `m`. Then we used the primary key value as foreign key value in the table `n`. And we used the primary key value of that row together with the one for table `m` to finally insert a new row in table `m`.

We will follow a similar approach here. We first invoke `CREATE SEQUENCE sqsid AS INT;`. The primary key for table `s` is then defined as `sid INT DEFAULT NEXTVAL('sqsid') PRIMARY KEY`.

Both tables `s` and `t` will have a column referencing a primary key from the respective other table. For table `s`, this is column `fktid`. For table `t`, this is column `fksid`. These will be used to enforce that each row in table `s` is definitely related to one row in table `t` and vice versa. Of course, they can also be related to multiple rows, which is why we need to manage the relationships in table `relate_s_and_t`.

We also must make sure that for each row in table `s` with data `(sid, fktid)`, there exists a corresponding row `(fksid, fktid)` in table `relate_s_and_t`. We do this by adding the constraint `s_sid_fktid_fk` which is `FOREIGN KEY (sid, fktid)` that `REFERENCES relate_s_and_t (fksid, fktid)` to table `s`. This is the exactly same approach we used back in [Section 19.2.2.9](#) ($Q \rightarrow\!\!\!-\!\!\!- R$).

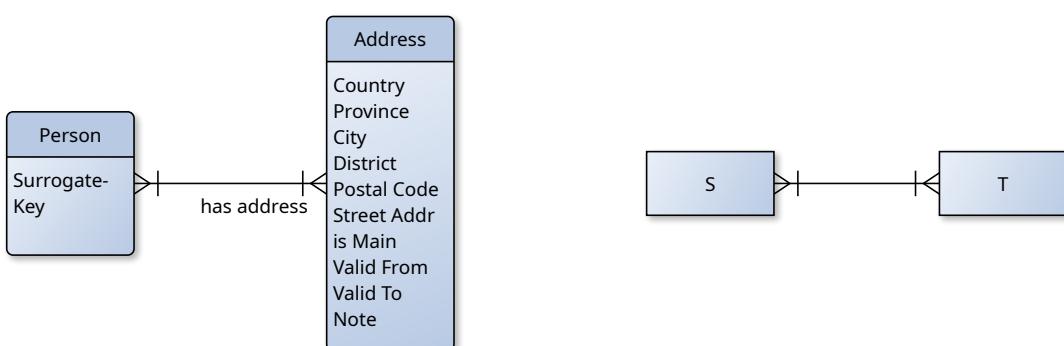


Figure 19.22: We encountered the $S \rightarrow\!\!\!-\!\!\!- T$ relationship pattern in [Figure 18.20](#).

Listing 19.106: The realization of a $S \rightarrow\!\!\!-\!\!\!- \leftarrow T$ conceptual relationship. (stored in file `ST_tables.sql`; output in Listing 19.107)

```

1  /* Create the tables for a S->|-----|<-T relationship. */
2
3  -- The sequence for the primary keys of the rows in s.
4  CREATE SEQUENCE sqsid AS INT;
5
6  -- Table S: Each row in S is related to one or multiple rows in T.
7  CREATE TABLE s (
8      sid    INT DEFAULT NEXTVAL('sqsid') PRIMARY KEY,
9      fktid  INT NOT NULL,   -- later used to reference T via relate_s_and_t
10     x      CHAR(3)        -- example for other attributes
11 );
12
13 -- Table T: Each row in T is related to one or multiple rows in S.
14 CREATE TABLE t (
15     tid    INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
16     fksid  INT NOT NULL,   -- later used to reference S via relate_s_and_t
17     y      CHAR(2)        -- example for other attributes
18 );
19
20 -- The table for managing the relationship between S and T.
21 CREATE TABLE relate_s_and_t (
22     fksid INT NOT NULL REFERENCES s (sid),
23     fktid INT NOT NULL REFERENCES t (tid),
24     PRIMARY KEY (fksid, fktid)  -- Primary key includes both columns.
25 );
26
27 -- To tables S and T, we add foreign key constraints that enforce that
28 -- the corresponding rows in relate_s_and_t exist.
29 ALTER TABLE s ADD CONSTRAINT s_sid_fktid_fk FOREIGN KEY (sid, fktid)
30     REFERENCES relate_s_and_t (fksid, fktid);
31 ALTER TABLE t ADD CONSTRAINT t_fksid_tid_fk FOREIGN KEY (fksid, tid)
32     REFERENCES relate_s_and_t (fksid, fktid);

```

Listing 19.107: The stdout of the program `ST_tables.sql` given in Listing 19.106.

```

1 $ psql "postgres://postgres:XXX@localhost/relationships" -v ON_ERROR_STOP=1
2   ↪ -e ST_tables.sql
3 CREATE SEQUENCE
4 CREATE TABLE
5 CREATE TABLE
6 ALTER TABLE
7 ALTER TABLE
8 # psql 16.12 succeeded with exit code 0.

```

The difference is that we must do this also for table `t`, because both relationship ends are mandatory. In other words, we add the constraint `t_fksid_tid_fk` to table `t`. It ensures that every tuple `(fksid, tid)` in that table must also appear as tuple `(fksid, fktid)` in table `relate_s_and_t`.

Defining the constraints that enforce referential integrity for pattern is one thing, finding a way to insert data into the tables under these tight constraints is another issue. In Listing 19.108, we do that.

Initially, the tables are empty. This means that we need to create three rows at once: We need to create a row in table `s`. We must immediately relate it to a newly created row in table `t`. And this relationship must also appear as row in table `relate_s_and_t`. Thanks to CTEs, this is possible.

First, we create a new primary key value for table `s` via the CTE `s_id`. This CTE does `SELECT NEXTVAL('sqsid') AS new_s_id`. We then use this new primary key value when inserting a row into table `t`. We therefore do `INSERT INTO t (y, fksid) SELECT 'z', new_s_id`. Of course, we do this as another CTE named `new_t` also doing `RETURNING tid, fksid`. This means

that this CTE will provide is the primary key value of the new row in table `t` and the primary key value `fksid` that we already allocated for the row in table `s` that we will create next.

Now we create this row. We invoke `INSERT INTO s (sid, x, fktid)SELECT fksid, '123', tid FROM new_t`. Notice how this uses the pre-created primary key value. It also uses the primary key value of the new row in table `t` as foreign key value. This is going to be our third CTE which we call `new_s`. It also returns both key values via `RETURNING sid, fktid`.

Finally, we can insert a row into `relate_s_and_t` by doing `INSERT INTO relate_s_and_t (fksid, fktid)SELECT sid, fk`. Thanks to CTEs, we could insert one row in three tables each – in a single SQL command. The referential integrity checks are performed at its end and thus work out positive.

Now there are existing records in the tables `s` and `t`. It is comparatively easy to create a new row for table `s` that is related to an existing row in table `t`. In this case, all we need to do is to insert the row into table `s` and, at the same time, insert a row into table `relate_s_and_t`. We just need a single CTE, which we will call `new_s`. It performs `INSERT INTO s (x, fktid)VALUES ('456', 1)RETURNING sid, fktid`. As result, we get the primary key value `sid` of the new row in table `s` as well as the foreign key value `fktid` to table `t`, `fktid`. This `fktid` is the same as the one we provided when creating the row in table `s`, namely `1`. We can then `INSERT INTO relate_s_and_t (fksid, fktid)SELECT sid, fktid FROM new_s` and are done.

Since the relationship pattern is symmetric, we can do exactly the same for table `t`. We can insert a new row into table `t` and relate it to an existing row in table `s`. For this, we would proceed the same way and also use one CTE.

Finally, we can also simply relate two existing rows in tables `s` and `t`. For this, we just need to create one new row in table `relate_s_and_t`. This can be like this: `INSERT INTO relate_s_and_t VALUES (1, 3)`. The contents of all three tables after inserting the data are shown in Figure 19.23.

Merging the data requires again two `INNER JOIN` expressions, exactly as before. The constraints prevent us from creating rows in `s` (or `t`) that are not related to rows in `t` (or `s`). We also cannot relate rows without creating the corresponding entry in `relate_s_and_t`, as shown in Listings 19.110 and 19.112.

Listing 19.108: Inserting into and selecting data from the realization of an $S \rightarrow\!\!\!-\!\!\!-| \leftarrow\!\!\!-\!\!\!- T$ conceptual relationship given in Listing 19.106. (stored in file `ST_insert_and_select.sql`; output in Listing 19.109)

```

1  /* Insert into the tables for the S->-----|<-T relationship. */
2
3  -- Create a pair of new and related S and T entities.
4  WITH s_id AS (SELECT NEXTVAL('sqsid') AS new_sid),
5      new_t AS (INSERT INTO t (y, fksid)
6                 SELECT 'AB', new_sid FROM s_id RETURNING tid, fksid),
7      new_s AS (INSERT INTO s (sid, x, fktid)
8                 SELECT fksid, '123', tid FROM new_t RETURNING sid, fktid)
9      INSERT INTO relate_s_and_t (fksid, fktid)
10     SELECT sid, fktid FROM new_s;
11
12 -- Create a new S record and relate it to an existing T entity.
13 WITH new_s AS (INSERT INTO s (x, fktid) VALUES ('456', 1)
14   RETURNING sid, fktid)
15   INSERT INTO relate_s_and_t (fksid, fktid)
16     SELECT sid, fktid FROM new_s;
17
18 -- Create a new T entity and relate it to an existing S entity.
19 WITH new_t AS (INSERT INTO t (y, fksid) VALUES ('CD', 2)
20   RETURNING tid, fksid)
21   INSERT INTO relate_s_and_t (fksid, fktid)
22     SELECT fksid, tid FROM new_t;
23
24 -- Create a new T entity and relate it to an existing S entity.
25 WITH new_t AS (INSERT INTO t (y, fksid) VALUES ('EF', 2)
26   RETURNING tid, fksid)
27   INSERT INTO relate_s_and_t (fksid, fktid)
28     SELECT fksid, tid FROM new_t;
29
30 -- We can now insert additional relationships.
31 INSERT INTO relate_s_and_t VALUES (1, 3);
32
33 -- Combine the rows from S and T. This needs two INNER JOINS.
34 SELECT sid, x, tid, y FROM relate_s_and_t
35   INNER JOIN s ON s.sid = relate_s_and_t.fksid
36   INNER JOIN t ON t.tid = relate_s_and_t.fktid;
```

Listing 19.109: The stdout of the program `ST_insert_and_select.sql` given in Listing 19.108.

```

1 $ psql "postgres://postgres:XXX@localhost/relationships" -v ON_ERROR_STOP=1
2   ↪ -e bf ST_insert_and_select.sql
3 INSERT 0 1
4 INSERT 0 1
5 INSERT 0 1
6 INSERT 0 1
7 INSERT 0 1
8   sid | x  | tid | y
9   ----+---+----+---
10  1  | 123 | 1  | AB
11  2  | 456 | 1  | AB
12  2  | 456 | 2  | CD
13  2  | 456 | 3  | EF
14  1  | 123 | 3  | EF
15
16 (5 rows)
17
18 # psql 16.12 succeeded with exit code 0.
```

Table s			Table t			Table relate_s_and_t	
sid	fktid	x	tid	fksid	y	fksid	fktid
1	1	"123"	1	1	"AB"	1	1
2	1	"456"	2	2	"CD"	2	2
			3	2	"EF"	2	3
						1	3

Figure 19.23: The contents of the three tables in the implementation of the $S \rightarrow\!\!-\!\!- T$ conceptual relationship after executing Listing 19.108.

Listing 19.110: Trying to insert new related rows into tables `s` and `t` without updating table `relate_s_and_t` does not work. (stored in file `ST_insert_error_1.sql`; output in Listing 19.111)

```

1  /* Trying to insert rows S and T without using the relationship table */
2
3  -- Create a pair of new related S and T entities, ignore relate_s_and_t.
4  WITH s_id AS (SELECT NEXTVAL('sqsid') AS new_sid),
5      new_t AS (INSERT INTO t (y, fksid)
6          SELECT 'GH', new_sid FROM s_id RETURNING tid, fksid)
7      INSERT INTO s (sid, x, fktid) SELECT fksid, '555', tid FROM new_t;

```

Listing 19.111: The stdout of the program `ST_insert_error_1.sql` given in Listing 19.110.

```

1  $ psql "postgres://postgres:XXX@localhost/relationships" -v ON_ERROR_STOP=1
2    ↪ -ebf ST_insert_error_1.sql
3  psql:conceptualToRelational/ST_insert_error_1.sql:7: ERROR:  insert or
4    ↪ update on table "t" violates foreign key constraint "t_fksid_tid_fk"
5  DETAIL:  Key (fksid, tid)=(3, 4) is not present in table "relate_s_and_t".
6  psql:conceptualToRelational/ST_insert_error_1.sql:7: STATEMENT:  /* Trying
7    ↪ to insert rows S and T without using the relationship table */
8  -- Create a pair of new related S and T entities, ignore relate_s_and_t.
9  WITH s_id AS (SELECT NEXTVAL('sqsid') AS new_sid),
10     new_t AS (INSERT INTO t (y, fksid)
11         SELECT 'GH', new_sid FROM s_id RETURNING tid, fksid)
12         INSERT INTO s (sid, x, fktid) SELECT fksid, '555', tid FROM new_t;
10 # psql 16.12 failed with exit code 3.

```

Listing 19.112: Trying to insert a new row into tables `s` and relate it to an existing row in table `t` without updating table `relate_s_and_t` does not work either. (stored in file `ST_insert_error_2.sql`; output in Listing 19.113)

```

1  /* Trying to insert rows S and T without using the relationship table */
2
3  -- Create relate a new S entity to an existing T entity without updating
4  -- relate_s_and_t.
5  INSERT INTO s (fktid, x) VALUES (3, '777');
```

Listing 19.113: The stdout of the program `ST_insert_error_2.sql` given in Listing 19.112.

```

1  $ psql "postgres://postgres:XXX@localhost/relationships" -v ON_ERROR_STOP=1
   ↵ -eef ST_insert_error_2.sql
2  psql:conceptualToRelational/ST_insert_error_2.sql:5: ERROR:  insert or
   ↵ update on table "s" violates foreign key constraint "s_sid_fktid_fk"
3  DETAIL:  Key (sid, fktid)=(4, 3) is not present in table "relate_s_and_t".
4  psql:conceptualToRelational/ST_insert_error_2.sql:5: STATEMENT: /* Trying
   ↵ to insert rows S and T without using the relationship table */
5  -- Create relate a new S entity to an existing T entity without updating
6  -- relate_s_and_t.
7  INSERT INTO s (fktid, x) VALUES (3, '777');
8  # psql 16.12 failed with exit code 3.
```

19.2.2.11 Summary

This was quite a journey. We have worked our way through every single binary conceptual relationship type that can be expressed with crow's foot notation. We listed all of these relationship types back in Section 18.5 (The Cardinality of Relationships).

We found that each of these types can be implemented in a relational DBMS. With *implemented*, we refer to the cardinality and modality of the relationship ends. If we want to implement, for example, a relationship following the pattern $K \text{ } \leftarrow\!\!\!-\!\!\rightarrow L$ in a DBMS, then we can do that. We can create a table `k` for the entities of type K and a table `l` for the entities of type L. Then we can use constraints that enforce that each row in table `l` must be related to exactly one row in table `k`. We can enforce that, at no time, there can ever be a row in table `l` that is not related to a row in table `k` or related to more than one such row. We can also enforce that each row in table `k` can be related to an arbitrary amount of rows in table `l`. Of course, we also enforce that if a row in table `l` is related to one row in table `k`, then that row in table `k` must be related to that row in table `l` and vice versa. No relationship end can be “open”. If these conditions are met, then our DB has the property of *referential integrity*.

What we also learned is that specifying the necessary constraints is not always easy. Especially if both relationship ends are mandatory, things can get complicated. We still managed to find solutions for all cases and also learned some additional SQL commands, such as [common table expressions \(CTEs\)](#). Sometimes, however, we used PostgreSQL-specific extensions to SQL. One particularly useful one is the [RETURNING](#) statement [360], which is also supported by MariaDB [226] and SQLite [361]. Another one were sequences and the [NEXTVAL](#) function [106, 388].

Due to the complexity, we may sometimes choose to change the relationship types when moving from the conceptual model to the logical model. We only considered binary relationship between two entity types, and doing something like $M \leftrightarrow N$ while also having $N \rightarrowtail X$ might prove too annoying to actually implement. One may also use more complex tools for that, like Stored Procedures and Transactions, which we do not discuss at this stage.

Creating examples for every single possible relationship type for two tables was fun, though. As final step, we can now get rid of our example DB in Listing 19.114.

Listing 19.114: We now tear down the example DB that we used to play around with when mapping conceptual relationships between entity types to tables in SQL. (stored in file `cleanup.sql`; output in [Listing 19.115](#))

```
1 /* Drop the database for our examples */
2
3 -- If the database 'relationships' exists, we delete it.
4 DROP DATABASE IF EXISTS relationships;
```

Listing 19.115: The stdout of the program `cleanup.sql` given in Listing 19.114.

```
1 $ psql "postgres://postgres:XXX@localhost" -v ON_ERROR_STOP=1 -e bf cleanup.  
2     ↕ sql  
2 DROP DATABASE  
3 # psql 16.12 succeeded with exit code 0.
```

19.2.3 Other Non-Relational Objects

When we begin designing an application, we develop the conceptual model. For this purpose, we usually use [entity relationship diagrams \(ERDs\)](#). ERDs give us lots of freedom in how we can represent the real world. We can use strong entities, weak entities, and relationships, and all three of these object types can have attributes. Attributes can be single-values or multivalued as well as atomic or composite.

We then use the relational data modelling approach for the logical model of our applications. At its core, the only type of objects offered by relational model for storing data are relations. Relations have attributes which are single-valued and atomic. Tables are the practical implementation of relations inside a [DBMS](#) and table columns represent attributes.

One of the beautiful things of the relational data model is that it feels very natural. Many components of the conceptual modeling level can directly be translated to the logical level. For example, Entities become tables.

Conceptual relationships are objects in the conceptual model that do not exist in the relational model as singular objects. Instead, they become either tables or columns and constraints, as we have discussed in the previous section. Multivalued attributes, which are useful at the conceptual level, also do not exist in the relational model. They become tables in their own right. Composite attributes are broken down to their elements which then become columns. All of this we have already discussed.

When scrolling over our section on conceptual modelling, we find that three types of objects are still “left over:”

1. We discussed strong entities, but not weak entities, see [Section 18.4](#) (Weak Entities).
2. We discussed relationships, but not relationship attributes, see [Definition 18.21](#) (Relationship Attribute).
3. We did not discuss derived attributes, see [Definition 18.11](#) (Derived Attribute).
4. While we exhaustively discussed all possible binary relationship patterns, we did not discuss relationships of a higher degree, i.e., situations where three or more entity types participate in a relationship, see [Definition 18.19](#) (Degree of Relationship).

To complete our treatment on the translation of conceptual models to logical models, let us also take a look how these are translated to the relational data model.

19.2.3.1 Weak Entities

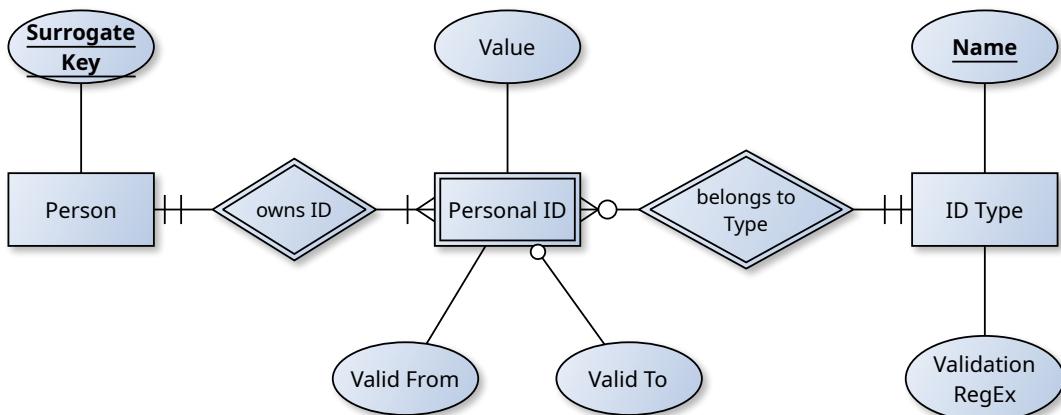
As introduced in [Section 18.4](#), weak entity types always appear in at least one identifying relationship with a strong entity type [396] (or with another weak entity type that can be identified.) In [Figure 18.14](#) back in [Section 18.5](#) (The Cardinality of Relationships), which is here reprinted as [Figure 19.24.1](#), we modeled IDs that belong to a person as such a weak entity type. Each *Personal ID* is in an identifying relationship with an entity of type *Person* and also in an identifying relationship with an entity of type *ID Type*.

If we map such identifying relationships to [SQL](#), then they must be represented with mandatory ends on the sides opposing the weak entity type. In other words, the relationships must be such that they enforce that the weak entity is connected to the strong entities. This is If we look at [Figure 19.24.1](#), then this would mean that we need to model the following two relationships:

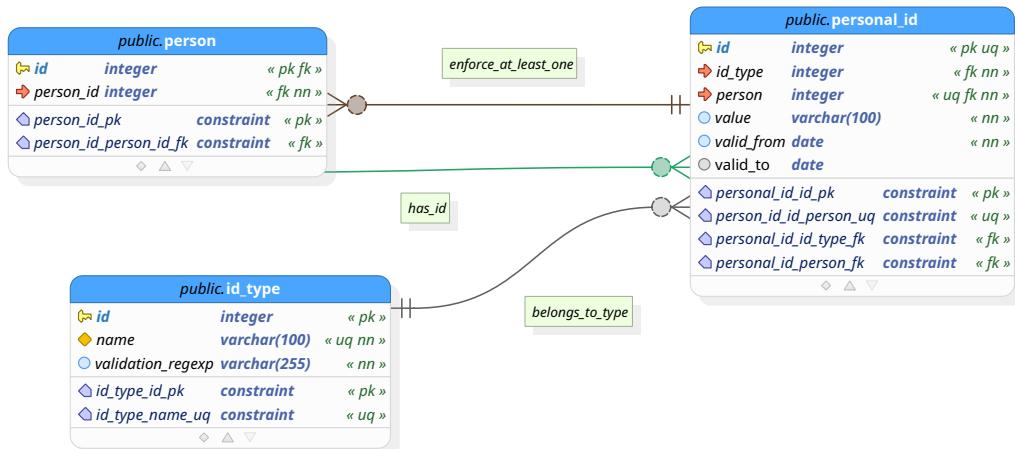
- Person $\text{+---}\text{+}$ Personal ID, which we can do based on [Section 19.2.2.7](#) and
- Personal ID $\text{+---}\text{+}$ ID Type, i.e., ID Type $\text{+---}\text{+}$ Personal ID, which we can do based on [Section 19.2.2.6](#).

For the modeling, we will this time use the [PgModeler](#). We will basically have to enter the same constraints and table formats as we would do via SQL, but can use a more convenient [GUI](#). The transformation of [Figure 19.24.1](#) to the logical model via [PgModeler](#) is illustrated in [Figure 19.24.2](#).

Back in [Section 19.2.2.7](#), we needed two foreign key [REFERENCES](#) constraints to enforce the M $\text{+---}\text{+}$ N. On the one hand, we referenced table [M](#) from table [N](#) with a single foreign key that was [NOT NULL](#). This enforced that there was at least one row in table [M](#) for each row in table [N](#). On the other hand, we referenced table [N](#) from table [M](#) with a compound foreign key. This enforced that each



(19.24.1) A reproduction of [Figure 18.14](#) from back in [Section 18.5](#) (The Cardinality of Relationships), which was created using yEd.



(19.24.2) A transformation of [Figure 18.14](#) to a logical model using [PgModeler](#).

Figure 19.24: The representation of weak entities as tables bound with mandatory relationships.

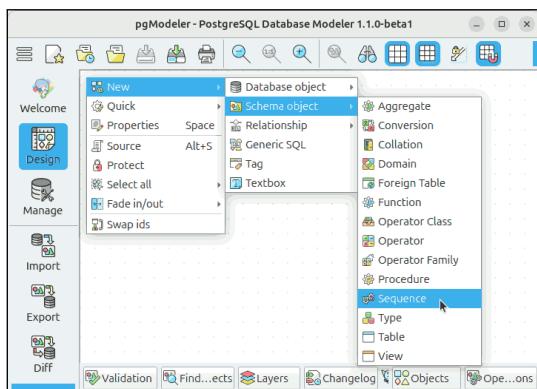
row in table `M` was connected to one row in table `N`, while not preventing that more rows in table `N` may exist that are related to it.

If we enter the same constraints into [PgModeler](#), it will display them separately. Indeed, this offers another perspective on the way we modeled the `M` $\text{---} \leftarrow \text{N}$ relationship in [SQL](#): Actually, we did realize it as a combination of a `M` $\text{---} \rightarrow \text{N}$ and a `M` $\text{---} \leftarrow \text{N}$ relationship.

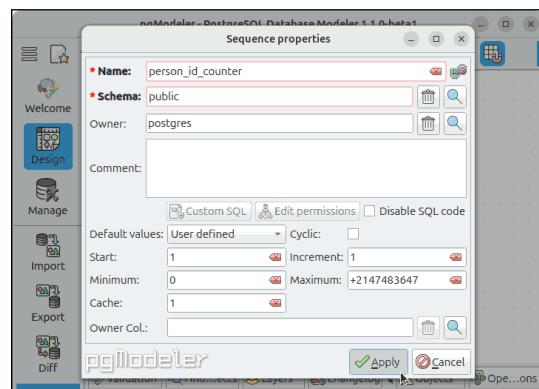
- The brown `person` $\text{---} \rightarrow \text{personal_id}$ relationship called `enforce_at_least_one` in [Figure 19.24.2](#) uses a composite foreign key and makes sure that, for each row in table `person`, at least one related row in table `personal_id` exists. It will be imposed to table `person`.
- The green `person` $\text{---} \leftarrow \text{personal_id}$ relationship called `has_id` in [Figure 19.24.2](#) uses a single foreign key in table `personal_id` and enforces that each row in that table is related to exactly one row in table `person`. It is imposed on table `personal_id`.

Notice that we used the `SEQUENCE` feature when creating the model for the `M` $\text{---} \leftarrow \text{N}$ setup in [Section 19.2.2.7](#). While I will not go into detail on how we create the whole logical model, let us briefly discuss how to use this feature in [PgModeler](#). It is rather easy. We begin by opening the [PgModeler](#) in the design view. We right-click into the workspace and select `New` \gg `Schema Object` \gg `Sequence`, as shown in [Figure 19.25.1](#).

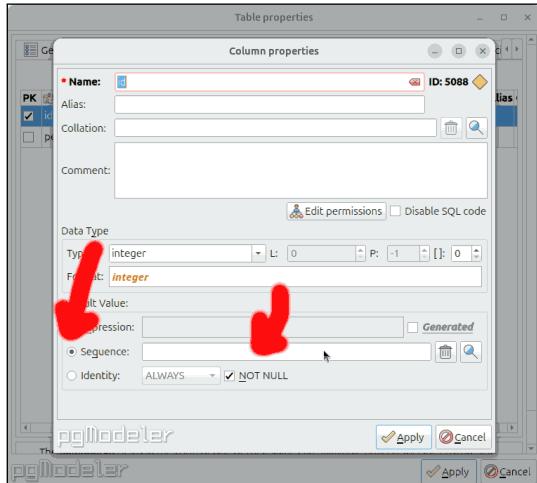
In the dialog that pops up, we first enter a reasonable name for our `SEQUENCE`. Here we choose `person_id_counter`, because we will use the sequence to generate the values for the `id` column of the `person` table. Then we click `Apply`, as sketched in [Figure 19.25.2](#).



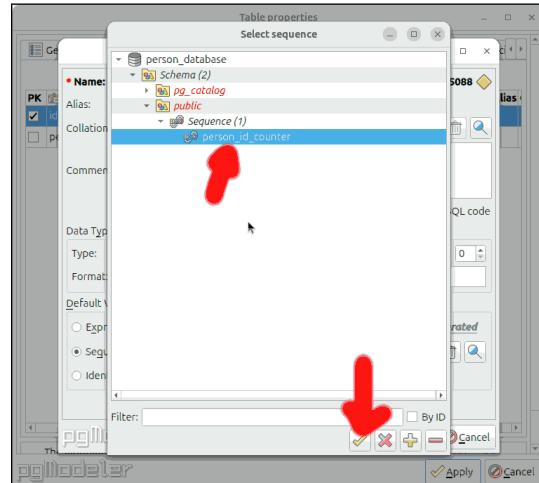
(19.25.1) In the PgModeler design view, right-click into the workspace and select [New] > [Schema Object] > Sequence.



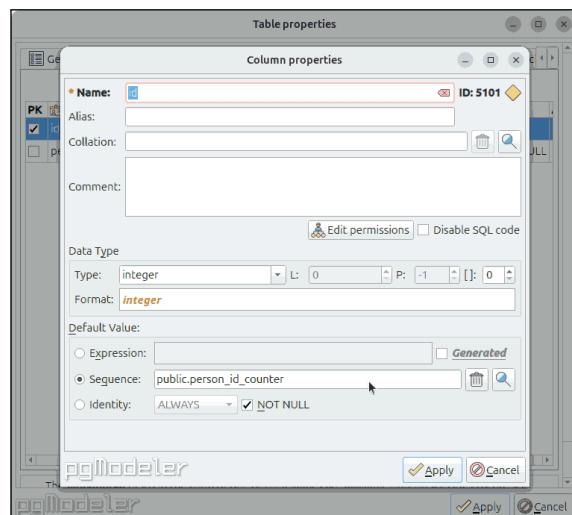
(19.25.2) In the dialog that pops up, we first enter a reasonable name for our SEQUENCE. Here we choose person_id_counter. Then we click [Apply].



(19.25.3) Later, when adding columns to a table, we can use the sequence. We select Sequence as Default Value and then click into the text box to the right of Sequence.



(19.25.4) Another dialog opens up where we click through the tree of objects, find our new SEQUENCE, and then click the check mark button at the bottom.



(19.25.5) The sequence is now selected and the column will take its default values from it.

Figure 19.25: Using the SEQUENCE feature in PgModeler.

Listing 19.116: The generated script to create the `person_database` DB. (src)

```

1 -- object: person_database | type: DATABASE --
2 -- DROP DATABASE IF EXISTS person_database;
3 CREATE DATABASE person_database;
4 -- ddl-end --

```

Listing 19.117: The generated script to create ID `SEQUENCE`. (src)

```

1 -- object: public.person_id_counter | type: SEQUENCE --
2 -- DROP SEQUENCE IF EXISTS public.person_id_counter CASCADE;
3 CREATE SEQUENCE public.person_id_counter
4     INCREMENT BY 1
5     MINVALUE 0
6     MAXVALUE 2147483647
7     START WITH 1
8     CACHE 1
9     NO CYCLE
10    OWNED BY NONE;
11
12 -- ddl-end --
13 ALTER SEQUENCE public.person_id_counter OWNER TO postgres;
14 -- ddl-end --

```

Listing 19.118: The generated script to create the `id_type` table. (src)

```

1 -- object: public.id_type | type: TABLE --
2 -- DROP TABLE IF EXISTS public.id_type CASCADE;
3 CREATE TABLE public.id_type (
4     id integer NOT NULL GENERATED ALWAYS AS IDENTITY ,
5     name varchar(100) NOT NULL,
6     validation_regex varchar(255) NOT NULL DEFAULT '.+',
7     CONSTRAINT id_type_id_pk PRIMARY KEY (id),
8     CONSTRAINT id_type_name_uq UNIQUE (name)
9 );
10 -- ddl-end --
11 ALTER TABLE public.id_type OWNER TO postgres;
12 -- ddl-end --

```

Later, when adding columns to a table `person`, we can use the sequence. We therefore select `Sequence` as `Default Value`. In Figure 19.25.3, we then click into the text box to the right of `Sequence`. As illustrated in Figure 19.25.4, another dialog opens up where we click through the tree of objects, find our new `Sequence`, and then click the check mark button at the bottom. In Figure 19.25.5, the sequence is now selected and the column will take its default values from it.

Listings 19.116 to 19.123 illustrate the generated SQL scripts corresponding to this logical model. In Listing 19.116, the empty new DB `person_database` is created. Listing 19.117 creates the sequence that we will use to generate the primary keys `id` of the `person` table.

Listing 19.118 creates the table for the ID types. This table has the primary key `id` and an attribute `name`, which must be `UNIQUE`. It also has a column `validation_regex` in which we will store a `regex` that applications can use to validate the input for a given ID type. This column must be `NOT NULL` and has the default value `'.+'`, which is the regex for “at least one character”. In other words, unless the DBA provides a better regex, applications are told to check that an ID always consists of at least one character.

Listing 19.119 creates that table in which we will store the *Personal ID* entities. Like all of our tables, it has a surrogate primary key `id`. The *Personal ID* entities are weak entities. They cannot exist without a identifying relationship to one *Person* entity and one *ID Type* entity. Therefore, this table stores two foreign keys: `id_type` and `person`. The corresponding constraints are added later in

Listing 19.119: The generated script to create the `personal_id` table. ([src](#))

```

1  -- object: public.personal_id | type: TABLE --
2  -- DROP TABLE IF EXISTS public.personal_id CASCADE;
3  CREATE TABLE public.personal_id (
4      id integer NOT NULL GENERATED BY DEFAULT AS IDENTITY ,
5      id_type integer NOT NULL,
6      person integer NOT NULL,
7      value varchar(100) NOT NULL,
8      valid_from date NOT NULL,
9      valid_to date,
10     CONSTRAINT personal_id_id_pk PRIMARY KEY (id),
11     CONSTRAINT person_id_id_person_uq UNIQUE (id, person)
12 );
13 -- ddl-end --
14 ALTER TABLE public.personal_id OWNER TO postgres;
15 -- ddl-end --

```

Listing 19.120: The generated script to create the `person` table. ([src](#))

```

1  -- object: public.person | type: TABLE --
2  -- DROP TABLE IF EXISTS public.person CASCADE;
3  CREATE TABLE public.person (
4      id integer NOT NULL DEFAULT nextval('public.person_id_counter'::
5          → regclass),
5      person_id integer NOT NULL,
6      CONSTRAINT person_id_pk PRIMARY KEY (id)
7 );
8 -- ddl-end --
9 ALTER TABLE public.person OWNER TO postgres;
10 -- ddl-end --

```

Listing 19.121: The generated script to create the constraint managing the relationship between the rows in table `personal_id` and those in table `id_type`. ([src](#))

```

1  -- object: personal_id_id_type_fk | type: CONSTRAINT --
2  -- ALTER TABLE public.personal_id DROP CONSTRAINT IF EXISTS
3  --   ↪ personal_id_id_type_fk CASCADE;
3  ALTER TABLE public.personal_id ADD CONSTRAINT personal_id_id_type_fk
4  --   ↪ FOREIGN KEY (id_type)
4  REFERENCES public.id_type (id) MATCH SIMPLE
5  ON DELETE NO ACTION ON UPDATE NO ACTION;
6  -- ddl-end --

```

Listings 19.121 and 19.122, respectively. The table has a `UNIQUE` constraint imposed on the tuples of the surrogate key `id` and the foreign key `person`, because later, the table `person` will use this tuple as composite foreign key.

In Listing 19.120, the table `person` is created. In this part of our example, we do not model much additional information about the *Person* entities. Therefore, this table only has a surrogate primary key `id`. However, we imposed the requirement that our system must have stored some form of identification for each entity of type *Person*. Therefore, we also have the foreign key `person_id` pointing to the table `personal_id` with the weak *Personal ID* entities. From our treatment of the M \bowtie N relationship pattern in Section 19.2.2.7, we know that this table must use a composition foreign key, which is created in Listing 19.123. This key makes sure that each row in table `person` has at least one corresponding row in table `personal_id`.

We create all the tables by executing all of the above scripts. Then, we can insert data into them by writing and running our own SQL script. Such a script is illustrated in Listing 19.124. We start out by inserting two rows into the table `id_type`, one for Chinese ID numbers (中国公民身份号码) and

Listing 19.122: The generated script to create the constraint that enforces that each row in table `personal_id` is related to exactly one row in table `person`. (src)

```

1 -- object: personal_id_person_fk | type: CONSTRAINT --
2 -- ALTER TABLE public.personal_id DROP CONSTRAINT IF EXISTS
   ↵ personal_id_person_fk CASCADE;
3 ALTER TABLE public.personal_id ADD CONSTRAINT personal_id_person_fk FOREIGN
   ↵ KEY (person)
4 REFERENCES public.person (id) MATCH SIMPLE
5 ON DELETE NO ACTION ON UPDATE NO ACTION;
6 -- ddl-end --

```

Listing 19.123: The generated script to create the constraint that enforces that, for each row in table `person`, at least one related row in table `personal_id` exists. (src)

```

1 -- object: person_id_person_id_fk | type: CONSTRAINT --
2 -- ALTER TABLE public.person DROP CONSTRAINT IF EXISTS
   ↵ person_id_person_id_fk CASCADE;
3 ALTER TABLE public.person ADD CONSTRAINT person_id_person_id_fk FOREIGN KEY
   ↵ (person_id,id)
4 REFERENCES public.personal_id (id,person) MATCH SIMPLE
5 ON DELETE NO ACTION ON UPDATE NO ACTION;
6 -- ddl-end --

```

one for Chinese mobile phone numbers. We store the same `regexes` that we used in [Section 19.2.1](#). Of course, now, we are only working on a part of a logical model of the teaching management system. So there is no application that actually uses these regexes. But at least in our imagination, we would have provided the means for checking ID values.

Then we use basically the same code as in [Listing 19.78](#) from back in [Section 19.2.2.7](#) ($M \bowtie \leftarrow N$) to fill the tables `person` and `personal_id`. First we create a `person` record with an associated Chinese national ID. Then we add a mobile phone number to that record. Then we create a second record, again with national ID. At the end, we combine the information back together using an `INNER JOIN`.

One may ask whether it is really useful to go through the hassle to enforce that each *Person* entity does have at least one associated *Personal ID* entity. This is a valid question. Creating the proper constraints to fully represent the logical relationship imposed by the conceptual model and the weak entity structure requires considerable effort. It also forces us to use the `SEQUENCE` feature, which is not supported by some `DBMSes`, and the `RETURNING` feature, which is also not supported by several systems.

It thus may be much easier to choose a *Person* $\bowtie \leftarrow$ *Personal ID* relationship structure instead of the *Person* $\bowtie \leftarrow$ *Personal ID* pattern. Doing this would allow us to remain in the realm of plain and simple standard `SQL`. We would have to trust the person tasked with entering data about students and faculty members that they will always require and properly enter some form of ID, even if the underlying `DB` would permit ID-less entries. We would not apply follow our conceptual model exactly. We would also reduce our “defense in depth” approach from [Best Practice 19](#), which emphasizes on using as many constraints and sanity checks as possible in all layers of our application. But we would get much simpler code, which is easier to check and easier to maintain. Also, if an application would later enter data, maybe via a library like `psycopg`, the complexity of the required access pattern would also be reduced. What to do here is a an interesting philosophical question, with no clear right or wrong choice.

In this book, we choose the harder method for two reasons. First, we want to see whether we can get it to work (we can). Also, we can learn more ... the hard way requires a deeper understanding of `SQL`. Anyway, after running this example we delete the `person_database` with a `DROP DATABASE` command.

So. What did we learn? We basically learned that weak entities can be implemented very much like the strong entities we use before. The main difference is that the identifying relationships must be mandatory. Which they should be in the conceptual model as well. So actually, we learned that we can

more or less ignore whether an entity is weak or strong.

Listing 19.124: Inserting into and selecting data from the tables in the `person_database`. (stored in file `insert_and_select.sql`; output in Listing 19.125)

```

1  /** Insert some data into the tables of our person database. */
2
3  -- Create two ID types: Chinese national ID and mobile phone numbers.
4  INSERT INTO id_type (name, validation_regex) VALUES
5      ('national ID', '^\\d{6}((19)|(20))\\d{9}[0-9X]$'),
6      ('mobile phone number', '^\\d{11}$');
7
8  -- Insert a new person record and a new ID record at the same time.
9  WITH pers_id AS (SELECT NEXTVAL('person_id_counter') AS person),
10    new_pers_id AS (INSERT INTO personal_id (
11        id_type, person, value, valid_from)
12        SELECT 1, person, '123456199501021234', '2024-12-01' FROM pers_id
13        RETURNING id, person)
14    INSERT INTO person (id, person_id) SELECT person, id FROM new_pers_id;
15
16  -- Insert a new personal ID for an existing person record.
17  INSERT INTO personal_id (id_type, person, value, valid_from) VALUES
18      (2, 1, '1234567890', '2023-02-07');
19
20  -- Insert a new person record and a new ID record at the same time.
21  WITH pers_id AS (SELECT NEXTVAL('person_id_counter') AS person),
22    new_pers_id AS (INSERT INTO personal_id (
23        id_type, person, value, valid_from)
24        SELECT 1, person, '123456200508071234', '2021-09-21' FROM pers_id
25        RETURNING id, person)
26    INSERT INTO person (id, person_id) SELECT person, id FROM new_pers_id;
27
28  -- Print the records that were inserted.
29  SELECT person, personal_id.id as pk, value, valid_from, name AS id_type
30      ↪ FROM personal_id
31      INNER JOIN id_type ON personal_id.id_type = id_type.id;

```

Listing 19.125: The stdout of the program `insert_and_select.sql` given in Listing 19.124.

```

1  $ psql "postgres://postgres:XXX@localhost/person_database" -v ON_ERROR_STOP
2  ↪ =1 -efb insert_and_select.sql
3
4  INSERT 0 2
5  INSERT 0 1
6  INSERT 0 1
7  INSERT 0 1
8  person | pk |          value          | valid_from |      id_type
9  -----+---+-----+-----+-----+
10   1 | 1 | 123456199501021234 | 2024-12-01 |  national ID
11   1 | 2 | 1234567890           | 2023-02-07 | mobile phone number
12   2 | 3 | 123456200508071234 | 2021-09-21 |  national ID
13 (3 rows)
14
15 # psql 16.12 succeeded with exit code 0.

```

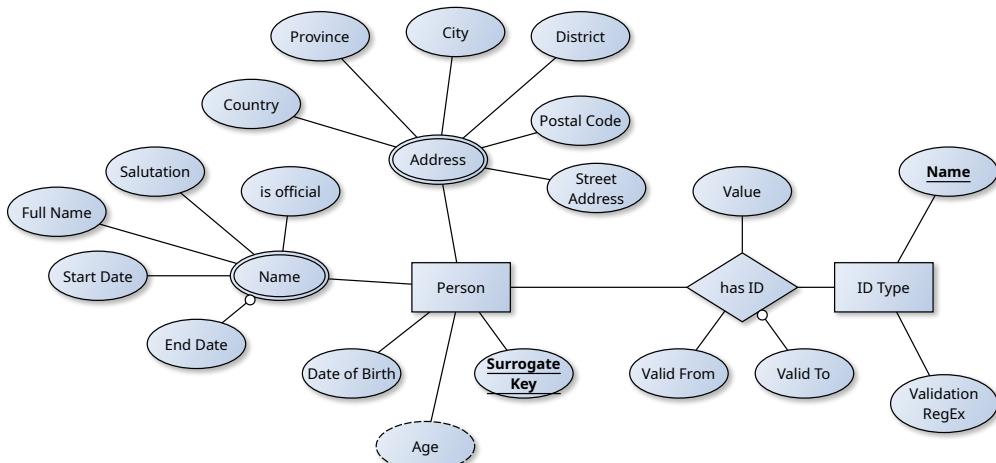
19.2.3.2 Relationship Attributes

Relationships in conceptual models may have attributes, as stated in [Definition 18.21](#). Of course, since relationships do not exist as distinct objects in the relational data model, we must find another way to express these attributes. Since only relations exist in the relational model and such relations become tables in a DB, the attributes of relationships also become table columns.

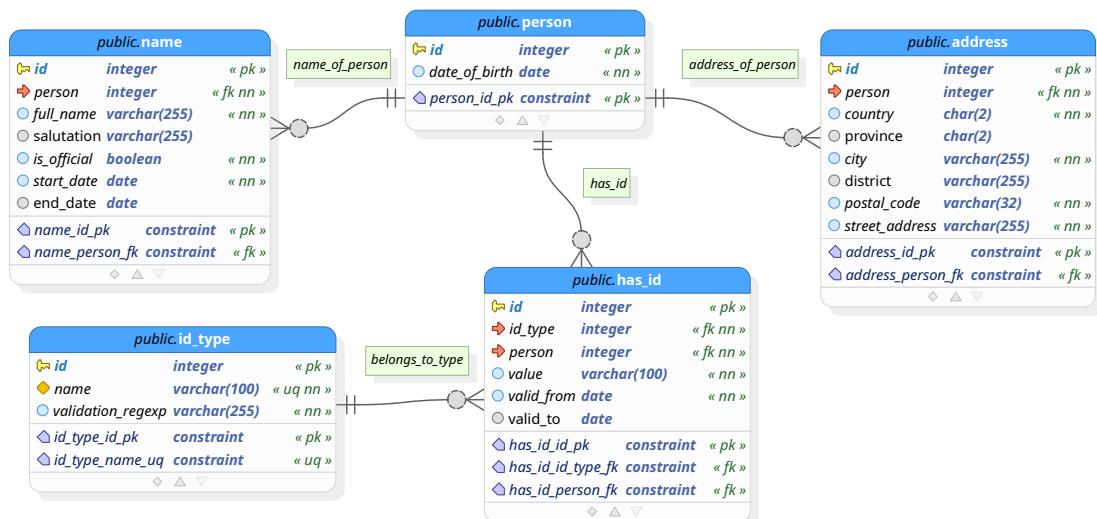
It will depend on the relationship pattern where we put them. To try this concept out, let us go back to an even earlier example of the *Person* entity: to [Figure 18.9](#) from back in [Section 18.3 \(Relationships\)](#). We created this figure using yEd and reprint it in [Figure 19.26.1](#). As you can see, in this figure, there is a relationship *has ID* that connects the *Person* entities with the entity type *ID Type*.

In the model, we did not annotate the relationships with cardinalities, because that was before we got to that topic. However, it is rather clear that this would either be a *Person* $\geq 0 - \infty$ *ID Type* or a *Person* $\geq 1 - \infty$ *ID Type* relationship. We can store arbitrarily many forms of ID for each person. Each form of ID may be used by arbitrarily many people. Since we went the hard way in the last section and modeled a relationship with the *mandatory many* pattern, we this time go easy and choose *Person* $0 - \infty$ *ID Type*. In other words, we follow the pattern $O \geq 0 - \infty P$ discussed in [Section 19.2.2.8](#) ($O \geq 0 - \infty P$).

For this pattern, we need an additional table. We follow exactly the same method as back in [Section 19.2.2.8](#), except that we use different table and column names. We also use [PgModeler](#) for the



(19.26.1) A reproduction of [Figure 18.9](#) from back in [Section 18.3 \(Relationships\)](#), which was created using yEd.



(19.26.2) A transformation of [Figure 19.26.1](#) to a logical model using [PgModeler](#).

Figure 19.26: The representation of relationship attributes as table for the relationship.

Listing 19.126: The generated script to create the `person_database` DB. (src)

```

1 -- object: person_database | type: DATABASE --
2 -- DROP DATABASE IF EXISTS person_database;
3 CREATE DATABASE person_database;
4 -- ddl-end --

```

Listing 19.127: The generated SQL script to create the table `person`. (src)

```

1 -- object: public.person | type: TABLE --
2 -- DROP TABLE IF EXISTS public.person CASCADE;
3 CREATE TABLE public.person (
4     id integer NOT NULL GENERATED BY DEFAULT AS IDENTITY ,
5     date_of_birth date NOT NULL,
6     CONSTRAINT person_id_pk PRIMARY KEY (id)
7 );
8 -- ddl-end --
9 ALTER TABLE public.person OWNER TO postgres;
10 -- ddl-end --

```

Listing 19.128: The generated SQL script to create the table `name`. (src)

```

1 -- object: public.name | type: TABLE --
2 -- DROP TABLE IF EXISTS public.name CASCADE;
3 CREATE TABLE public.name (
4     id integer NOT NULL GENERATED BY DEFAULT AS IDENTITY ,
5     person integer NOT NULL,
6     full_name varchar(255) NOT NULL,
7     salutation varchar(255),
8     is_official boolean NOT NULL DEFAULT True,
9     start_date date NOT NULL DEFAULT CURRENT_DATE,
10    end_date date,
11    CONSTRAINT name_id_pk PRIMARY KEY (id)
12 );
13 -- ddl-end --
14 ALTER TABLE public.name OWNER TO postgres;
15 -- ddl-end --

```

sake of convenience (and because it allows me to create nice ERDs...). We call the additional table `has_id` to properly reflect the conceptual model. The relationship attributes will therefore become columns of this table. This is illustrated in Figure 19.26.2.

Notice how we again gain another perspective on how we modeled relationships: In Section 19.2.2.8, we implement an O \rightarrowtail O \leftarrow P relationship in SQL. We did so using the additional table `relate_o_and_p`. Another perspective would be that we actually implemented two relationships: `o` \rightarrowtail `relate_o_and_p` and `p` \rightarrowtail `relate_o_and_p`. Each row in table `relate_o_and_p` must be related to one row in table `o` and also to one row in table `p`. Each row in table `o` can be related to arbitrarily many rows in table `relate_o_and_p`. Each row in table `p` can be related to arbitrarily many rows in table `relate_o_and_p`. Indeed, that forms a O \rightarrowtail O \leftarrow P relationship. And the diagram for the logical model here reflects that.

Also, when implementing the full conceptual model given in Figure 19.26.1, we again encounter multivalued attributes: `Name` and `Address`. As we already learned before, these go into additional tables, which we call `name` and `address`, respectively. For these, we again use the proper foreign key REFERENCES constraints.

We can export the logical model created with PgModeler and we get Listings 19.126 to 19.135. Listing 19.126 creates the DB `person_database`. The table `person` is created in Listing 19.127, with a surrogate primary key and an attribute `date_of_birth` for the DOB.

In Listing 19.128, the table `name` is created for the multivalued composite attribute `Name`. Each row

Listing 19.129: The generated SQL script to create the table `address`. (src)

```

1  -- object: public.address | type: TABLE --
2  -- DROP TABLE IF EXISTS public.address CASCADE;
3  CREATE TABLE public.address (
4      id integer NOT NULL GENERATED BY DEFAULT AS IDENTITY ,
5      person integer NOT NULL,
6      country char(2) NOT NULL DEFAULT 'CN',
7      province char(2) DEFAULT 'AH',
8      city varchar(255) NOT NULL,
9      district varchar(255),
10     postal_code varchar(32) NOT NULL,
11     street_address varchar(255) NOT NULL,
12     CONSTRAINT address_id_pk PRIMARY KEY (id)
13 );
14 -- ddl-end --
15 ALTER TABLE public.address OWNER TO postgres;
16 -- ddl-end --

```

Listing 19.130: The generated SQL script to create the table `id_type`. (src)

```

1  -- object: public.id_type | type: TABLE --
2  -- DROP TABLE IF EXISTS public.id_type CASCADE;
3  CREATE TABLE public.id_type (
4      id integer NOT NULL GENERATED ALWAYS AS IDENTITY ,
5      name varchar(100) NOT NULL,
6      validation_regexp varchar(255) NOT NULL DEFAULT '.+',
7      CONSTRAINT id_type_id_pk PRIMARY KEY (id),
8      CONSTRAINT id_type_name_uq UNIQUE (name)
9 );
10 -- ddl-end --
11 ALTER TABLE public.id_type OWNER TO postgres;
12 -- ddl-end --

```

in table `name` will store a foreign key `person` that is linked to a row in table `person` using a `REFERENCES` constraint. This is added later in Listing 19.132 using `ALTER TABLE`. Apart from that, we added a small gimmick for your enjoyment: The column `start_date` is created `NOT NULL DEFAULT CURRENT_DATE`. This means that, for each row in table `name`, we must store a proper start date as per the `NOT NULL`. However, if we do not provide such a date when the row is created, a `DEFAULT` value is stored instead. This `DEFAULT` is `CURRENT_DATE`, which, as the name implies, will be the date at the very moment we enter the row into the DB [120]. Similar functions are `CURRENT_TIME` and, most importantly, `CURRENT_TIMESTAMP` [120], which is often used for timestamping.

But back to the topic at hand. Listing 19.129 creates the table `address`. There is nothing too special about this table. One interesting thing is maybe that we provide `DEFAULT` values also for `country` and `province`. This means that, if we do not provide these when we enter data, they are set to '`CN`' and '`AH`', respectively. The foreign key `person` is related to the table `person` using the `REFERENCES` constraint given in Listing 19.133.

Listing 19.130 creates the table `id_type` in pretty much the same shape and form already used in the previous section in Listing 19.118. The table `has_id` is created in Listing 19.131. It follows the same design as table `personal_id` from the previous section, which was given in Listing 19.119. It hosts the relationship attributes that we designed in our conceptual model. The foreign key constraints linking its rows to tables `id_type` and `person` are created using Listings 19.134 and 19.135, respectively. With this, we have constructed the `person_database`.

We fill the DB with data in Listing 19.136. We begin by creating three records for table `person`, for which only we need to specify the DOBs. We then insert the names of the three people in table `name`. Curiously, we have four name records. The reason is that, on February 2nd, 1989, a 26 year old Ms. Bibbi married and decided to take on the name of her husband. Her name thus changed to Mrs. Bebba. This is reflected in the last two rows we add to the table `name`.

Listing 19.131: The generated SQL script to create the table `has_id`. (src)

```

1 -- object: public.has_id | type: TABLE --
2 -- DROP TABLE IF EXISTS public.has_id CASCADE;
3 CREATE TABLE public.has_id (
4     id integer NOT NULL GENERATED BY DEFAULT AS IDENTITY ,
5     id_type integer NOT NULL,
6     person integer NOT NULL,
7     value varchar(100) NOT NULL,
8     valid_from date NOT NULL,
9     valid_to date,
10    CONSTRAINT has_id_id_pk PRIMARY KEY (id)
11 );
12 -- ddl-end --
13 ALTER TABLE public.has_id OWNER TO postgres;
14 -- ddl-end --

```

Listing 19.132: The generated SQL script to create the foreign key constraint ensuring that each row in table `name` is associated with one row in table `person`. (src)

```

1 -- object: name_person_fk | type: CONSTRAINT --
2 -- ALTER TABLE public.name DROP CONSTRAINT IF EXISTS name_person_fk CASCADE
3   ↵ ;
4 ALTER TABLE public.name ADD CONSTRAINT name_person_fk FOREIGN KEY (person)
5 REFERENCES public.person (id) MATCH SIMPLE
6 ON DELETE NO ACTION ON UPDATE NO ACTION;
7 -- ddl-end --

```

Listing 19.133: The generated SQL script to create the foreign key constraint ensuring that each row in table `address` is associated with one row in table `person`. (src)

```

1 -- object: address_person_fk | type: CONSTRAINT --
2 -- ALTER TABLE public.address DROP CONSTRAINT IF EXISTS address_person_fk
3   ↵ CASCADE;
4 ALTER TABLE public.address ADD CONSTRAINT address_person_fk FOREIGN KEY (
5   ↵ person)
6 REFERENCES public.person (id) MATCH SIMPLE
7 ON DELETE NO ACTION ON UPDATE NO ACTION;
8 -- ddl-end --

```

The same two ID types as in the previous example – Chinese ID and mobile phone number – are created and stored in table `id_type`. For each ID type and person, we then store one value in the table `has_id`. We make use of the relationship attributes stored in there. Finally, we provide one `address` record for each row in `person`. The data entry is completed.

In Listing 19.138, we use a query that combines data from several tables using `INNER JOIN` statements and then concatenates the data to single text strings per result row. This is maybe the biggest `INNER JOIN` we did to date. We want to combine the names of the people with their addresses and mobile numbers. This can be done using the string concatenation operator `||`.

To get the necessary data, we go through the table `name` row-by-row. Later, in a `WHERE` clause, we make sure to only use the currently valid names via `name.is_official = TRUE`. For each such row in table `name`, we pick the corresponding `person` record via the foreign key. Using the primary key of that record's primary key, we can find the corresponding rows in tables `address` and `has_id`. We keep only the IDs with ID-type `2` via `has_id.id_type = 2` in the `WHERE` clause, i.e., the mobile phone numbers.

If a person would not have a mobile phone number or address, they would not appear in this list. What would happen if a person had multiple mobile phone numbers and/or addresses? Feel encouraged to try it out!

Anyway, we sort the resulting rows by the `DOB` of the persons, meaning that the oldest person will

Listing 19.134: The generated SQL script to create the foreign key constraint ensuring that each row in table `has_id` is associated with one row in table `id_type`. (src)

```

1 -- object: has_id_id_type_fk | type: CONSTRAINT --
2 -- ALTER TABLE public.has_id DROP CONSTRAINT IF EXISTS has_id_id_type_fk
   ↪ CASCADE;
3 ALTER TABLE public.has_id ADD CONSTRAINT has_id_id_type_fk FOREIGN KEY (
   ↪ id_type)
4 REFERENCES public.id_type (id) MATCH SIMPLE
5 ON DELETE NO ACTION ON UPDATE NO ACTION;
6 -- ddl-end --

```

Listing 19.135: The generated SQL script to create the foreign key constraint ensuring that each row in table `has_id` is associated with one row in table `person`. (src)

```

1 -- object: has_id_person_fk | type: CONSTRAINT --
2 -- ALTER TABLE public.has_id DROP CONSTRAINT IF EXISTS has_id_person_fk
   ↪ CASCADE;
3 ALTER TABLE public.has_id ADD CONSTRAINT has_id_person_fk FOREIGN KEY (
   ↪ person)
4 REFERENCES public.person (id) MATCH SIMPLE
5 ON DELETE NO ACTION ON UPDATE NO ACTION;
6 -- ddl-end --

```

come first. The result is as expected.

So what did we learn this time? Relationships on the conceptual level can have attributes. Relationships do not exist as objects in logical schemas based on the relational data model. Here, we only have tables. So these relationship attributes from the conceptual schema become table columns in the logical schema. And they are located in the tables that represent the relationships. That was easy.

Listing 19.136: Inserting into the tables in the `person_database`. (stored in file `insert.sql`; output in Listing 19.137)

```

1  /** Insert some data into the tables of our person database. */
2
3  -- Create the persons Bebbo, Bibbo, and Bebba.
4  INSERT INTO person (date_of_birth) VALUES
5      ('2005-08-07'), ('1995-01-02'), ('1963-11-13');
6
7  -- Create the names of Bebbo, Bibbo, and Bebba.
8  INSERT INTO name (person, full_name, salutation, is_official,
9                  start_date, end_date) VALUES
10     (1, 'Bebbo', 'Bebbo Machine', TRUE, '2005-08-07', NULL),
11     (2, 'Bibbo', 'The Bib-Man', TRUE, '1995-01-02', NULL),
12     (3, 'Bibbi', 'Ms. Bibbi', FALSE, '1963-11-13', '1989-02-12'),
13     (3, 'Bebba', 'Mrs. Bebba', TRUE, '1989-02-12', NULL);
14
15 -- Create two ID types: Chinese national ID and mobile phone numbers.
16 INSERT INTO id_type (name, validation-regexp) VALUES
17     ('national ID', '^\\d{6}((19)|(20))\\d{9}[0-9X]$'),
18     ('mobile phone number', '^\\d{11}$');
19
20 -- Insert the personal IDs and mobile phone numbers of the people.
21 INSERT INTO has_id (id_type, person, value, valid_from) VALUES
22     (1, 1, '123456200508071234', '2021-09-21'),
23     (1, 2, '123456199501021234', '2024-12-01'),
24     (1, 3, '123456196311131234', '1983-10-03'),
25     (2, 1, '2222222222', '2020-09-12'),
26     (2, 2, '11111111111', '2012-07-30'),
27     (2, 3, '44444444444', '2012-07-30');
28
29 -- Provide an address for each person.
30 INSERT INTO address (person, country, province, city,
31                      district, postal_code, street_address) VALUES
32     (1, 'DE', 'SN', 'Chemnitz', 'Zentrum', '09111', 'Rathaus'),
33     (2, 'CN', 'AH', 'Hefei', 'Jinkaiqu', '230601', 'Hefei University'),
34     (3, 'US', 'NY', 'New York', 'Manhattan', '10036', 'Times Square');
```

Listing 19.137: The stdout of the program `insert.sql` given in Listing 19.136.

```

1 $ psql "postgres://postgres:XXX@localhost/person_database" -v ON_ERROR_STOP
2   ↪=1 -ebf insert.sql
3 INSERT 0 3
4 INSERT 0 4
5 INSERT 0 2
6 INSERT 0 6
7 INSERT 0 3
7 # psql 16.12 succeeded with exit code 0.
```

Listing 19.138: Selecting data from the tables in the `person_database`. (stored in file `select.sql`; output in Listing 19.139)

```

1  /** Combine people, names, addresses, and phone numbers. */
2
3  SELECT full_name || ':' || country || '-' || province || ', ' ||
4      city || ',', ' || street_address ||
5      ' (call ' || value || ')') AS contact FROM name
6    INNER JOIN person ON person.id = name.person
7    INNER JOIN address ON person.id = address.person
8    INNER JOIN has_id ON person.id = has_id.person
9    WHERE name.is_official = TRUE AND -- use official name only
10       has_id.id_type = 2 -- use ID type for mobile phone
11    ORDER BY person.date_of_birth; -- sort by age, oldest people first

```

Listing 19.139: The stdout of the program `select.sql` given in Listing 19.138.

```

1  $ psql "postgres://postgres:XXX@localhost/person_database" -v ON_ERROR_STOP
2      ↪=1 -ebf select.sql
3          contact
4  -----
5  Bebba: US-NY, New York, Times Square (call 444444444444)
6  Bibbo: CN-AH, Hefei, Hefei University (call 111111111111)
7  Bebbo: DE-SN, Chemnitz, Rathaus (call 222222222222)
8  (3 rows)
9  # psql 16.12 succeeded with exit code 0.

```

19.2.3.3 Derived Attributes

Derived attributes, introduced in [Definition 18.11](#), are attributes that are computed based on the values of other attributes. Whether or not they should be stored in a table depends on how hard it is to compute them. For example, imagine that we have a factory and package things in boxes, pretty much like back in [Section 9.1 \(The Table “product”\)](#). Maybe we have a table in our DB just for storing the types of boxes by their width, height, and depth. The volume of a box could be derived attribute, computed as the product of the three dimensions. It would not make sense wasting space by storing this value, because we can easily compute it in an [SQL](#) query.

Then again, we may sell multiple products as packages. For example, we may offer a vacuum cleaner together with a spare battery, two different nozzles, two brushes, an edge tool, and other attachments, and a pack of air filter papers. These package elements need to all fit together into a box and the order in which they are packed into the is important. This is a bit like a puzzle game. The best order could be computed based on the dimensions of the box and the dimensions of the package components. Doing so, however, means solving an [NP-hard](#) problem, which requires lots of time. Then, we would definitely store the corresponding data as attributes even though they could be re-computed.

A derived attribute that is stored is just a normal table column. If the value can be computed from values of other columns, as is the case in the box volume example above, the column can be annotated as [GENERATED](#) [175]. At the time of this writing, [PostgreSQL](#) 17 is the current major version. This major version only supports [STORED](#) generated columns [175], but in the future, [VIRTUAL](#) generated columns whose values are not stored in the DB, but which are computed when read, will be introduced.

In summary, there are three choices of how we can implement derived attributes:

- realize them as [GENERATED](#) table columns and store their values [175],
- realize them as virtual [GENERATED](#) table columns without storing their values, which is currently not yet supported by [PostgreSQL](#) [175], but will be implemented in the future and may be supported by other DBMS already, or
- realize them by computing their values in [SELECT](#) queries as needed.

Now, in the conceptual model illustrated as [Figure 19.26.1](#) in the previous section, there was one derived attribute, namely *Age*. This attribute is supposed to represent the age of a person. Of course, if we know the [DOB](#) of a the person, we can compute the age right away. So how would we realize this attribute: As a stored [GENERATED](#) column or compute it on the fly during [SELECT](#) queries?

Storing this value makes no sense: The age of a person is not a constant. It will change as time goes by. Also, even if [PostgreSQL](#) does support virtual (not stored) [GENERATED](#) columns, it would still be a mistake to use those to represent the age of people.

Several restrictions apply to the definition of generated columns and tables involving generated columns:

- The generation expression can only use immutable functions and cannot use sub-queries or reference anything other than the current row in any way.
- ...

— [175], 2025

To compute the age of a person, we need the DOB and we will somehow subtract it from the [CURRENT_DATE](#). [CURRENT_DATE](#) certainly is not an immutable or constant function. It changes daily.

So regardless of whether we have virtual [GENERATED](#) columns or not, the best way to compute the age of a person is by defining a query, maybe solidified as a [VIEW](#). Still having the [person_database](#) from the previous section lying around, we will do it in there.

In [Listing 19.140](#), we create a view with the name [person_age](#). [PostgreSQL](#) offers the function [AGE](#) [120] which computes the difference between [CURRENT_DATE](#) and the [TIMESTAMP](#) or [DATE](#) value provided as parameter. Our new view simply selects all the columns from table [person](#) and adds a new column called [age](#) computed as [AGE\(date_of_birth\)](#). In [Listing 19.141](#), we then first print the value of the [CURRENT_DATE](#) of the time when this book is compiled for reference. Then we combine the result of the new view [person_age](#) with the table [name](#) to print all the person records together with their name, DOBs, and ages. Having done that, we can delete the DB [person_database](#) again.

Listing 19.140: Create a view that represents the derived attribute `age` and execute it. (stored in file `view_person_age.sql`; output in Listing 19.141)

```

1  /* Create a view adding age information to person table. */
2
3  -- Create a view that adds the age to the fields of the person table.
4  CREATE VIEW person_age AS
5      SELECT *, AGE(date_of_birth) AS age FROM person;
6
7  -- Show current date.
8  SELECT CURRENT_DATE;
9
10 -- Execute the view.
11 SELECT name.full_name, person_age.date_of_birth, person_age.age
12     FROM person_age INNER JOIN name ON name.person = person_age.id
13     WHERE name.is_official = TRUE;

```

Listing 19.141: The stdout of the program `view_person_age.sql` given in Listing 19.140.

```

1  $ psql "postgres://postgres:XXX@localhost/person_database" -v ON_ERROR_STOP
2      ↪=1 -ebsf view_person_age.sql
3  CREATE VIEW
4      current_date
5  -----
6  2026-02-20
7  (1 row)
8
9      full_name | date_of_birth | age
10     -----+-----+-----
11     Bebbo    | 2005-08-07   | 20 years 6 mons 13 days
12     Bibbo    | 1995-01-02   | 31 years 1 mon 18 days
13     Bebba    | 1963-11-13   | 62 years 3 mons 7 days
14
15 # psql 16.12 succeeded with exit code 0.

```

19.2.3.4 Relationships of a Higher Degree

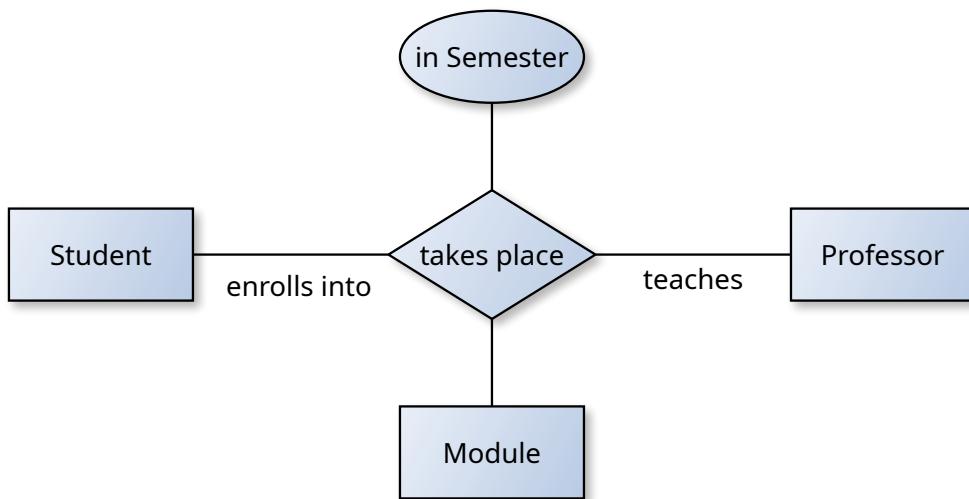
In Figure 19.27.1, we reproduce a figure from Section 18.3 (Relationships) where a ternary relationship is illustrated. The entity types *Professor*, *Student*, and *Module* are related with each other and this relationship even has a relationship attribute. The professor teaches a module in a certain semester. The student enrolls in that taught module in that semester.

We now want to create a logical model that fits to this scenario. First we sort out the easy things. The entity types become tables with surrogate primary keys. We thus get the tables `module`, `professor`, and `student`. In order to add a bit more sense to this example, we include the attribute `name` in the tables `student` and `professor` as well as `title` in `module`.

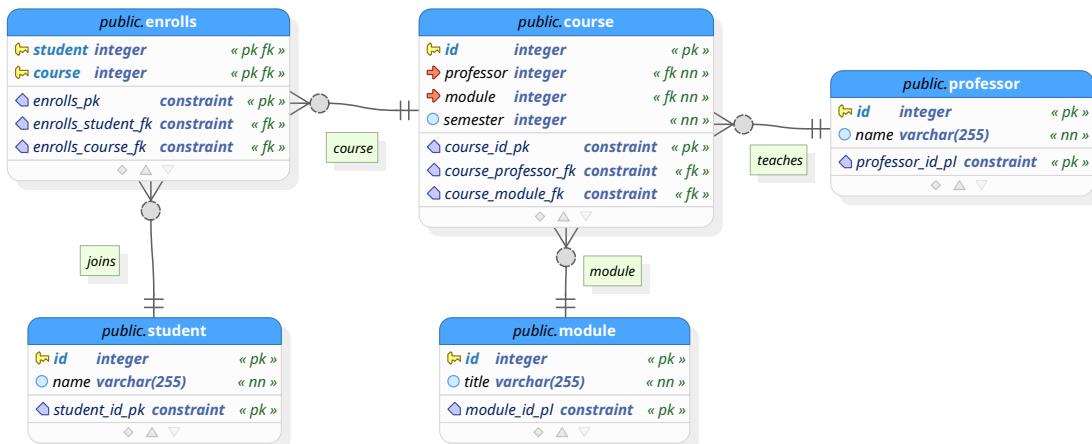
We then construct the ternary relationship. We basically have two choices to do this:

1. We can create one single table `takes_place`. It will have three foreign keys to the tables `professor`, `student`, and `module`, as well as a column `semester`. It will store all the data and participants of the ternary relationship.
2. We can create one table `course` with a surrogate key and use it to store the “module-instantiation”. An instantiation is a module being taught by a professor in one semester. This table thus will have foreign keys to the tables `professor` and `module` as well as the column `semester`. We then create a second table `enrolls` that has two foreign keys, one for the table `student` and one for the table `course`. It stores the “enrolls” relationship of students towards module-instantiations.

Both choices would be feasible, but we go with the second one. The idea is that we can also consider the realization of a module by a certain professor in a semester as an entity in its own right. This entity



(19.27.1) A reproduction of Figure 18.6.4, which illustrates a ternary relationship of students, modules, and professors with the relationship attribute semester. This graphic was painted using yEd.



(19.27.2) One possible transformation of Figure 18.6.4 to a logical model using PgModeler.

Figure 19.27: The representation of relationships with a degree higher than two in relational logical models.

would have the attribute `semester`, but could also have arbitrary other attributes that we may want to add later. Relating the students to that `course` table by using a separate table then makes sense.

This approach has two striking benefits. First, it avoids redundancy. The first idea, namely storing the student, module, and professor keys in a single table, would mean that we store the fact that “Prof. Mrs. Bibba” teaches “Mathematics 101” again and again. By using two tables, one for module instances and one for module enrollment, we only store it once.

Second, this second method can properly represent a situation in which a professor teaches the same module twice in one semester. Due to limited room capacity or scheduling conflicts of student time tables, it might indeed possible that “Prof. Mrs. Bibba” teaches “Mathematics 101” on Mondays for one group of students and on Tuesdays for a second group. In the first solution, there is no way to express this fact. In the second one, which we chose, we can simply instantiate the module twice.

Anyway, once we made up our mind, we can create the model. We do this using, of course, PgModeler and get the logical schema illustrated in Figure 19.27.2.

For this purpose, we extend the model further with reasonable assumptions about cardinalities and modalities. For example, it makes sense to assume that each row in table `course` must be related to exactly one row in table `professor` and exactly one row in table `module`. We do not permit a course that represents no module of the curriculum and we also do not permit a course that teaches the knowledge of multiple modules at once. Similarly, we cannot have a course without professor and also

Listing 19.142: The generated script to create the `teaching_database` DB. ([src](#))

```

1 -- object: teaching_database | type: DATABASE --
2 -- DROP DATABASE IF EXISTS teaching_database;
3 CREATE DATABASE teaching_database;
4 -- ddl-end --

```

Listing 19.143: The generated script to create the table `student`. ([src](#))

```

1 -- object: public.student | type: TABLE --
2 -- DROP TABLE IF EXISTS public.student CASCADE;
3 CREATE TABLE public.student (
4     id integer NOT NULL GENERATED BY DEFAULT AS IDENTITY ,
5     name varchar(255) NOT NULL,
6     CONSTRAINT student_id_pk PRIMARY KEY (id)
7 );
8 -- ddl-end --
9 ALTER TABLE public.student OWNER TO postgres;
10 -- ddl-end --

```

Listing 19.144: The generated script to create the table `professor`. ([src](#))

```

1 -- object: public.professor | type: TABLE --
2 -- DROP TABLE IF EXISTS public.professor CASCADE;
3 CREATE TABLE public.professor (
4     id integer NOT NULL GENERATED BY DEFAULT AS IDENTITY ,
5     name varchar(255) NOT NULL,
6     CONSTRAINT professor_id_pk PRIMARY KEY (id)
7 );
8 -- ddl-end --
9 ALTER TABLE public.professor OWNER TO postgres;
10 -- ddl-end --

```

Listing 19.145: The generated script to create the table `module`. ([src](#))

```

1 -- object: public.module | type: TABLE --
2 -- DROP TABLE IF EXISTS public.module CASCADE;
3 CREATE TABLE public.module (
4     id integer NOT NULL GENERATED BY DEFAULT AS IDENTITY ,
5     title varchar(255) NOT NULL,
6     CONSTRAINT module_id_pk PRIMARY KEY (id)
7 );
8 -- ddl-end --
9 ALTER TABLE public.module OWNER TO postgres;
10 -- ddl-end --

```

we do not permit more than one professor to teach a course. Well, the latter may actually be possible. Maybe that would be a good question to bring up during the requirements gathering process or later during conceptual modelling when sitting down with our stakeholders in the university. At least for now, we do not permit that (as it would also make our relationship patterns more complicated).

Professors can teach multiple modules and each module can be taught by multiple professors. We could add a `UNIQUE` constraint over the combination of the columns `module`, `professor`, and `semester`. This would mean that a professor cannot teach the same module twice in the same semester. Then again, we can also permit this. We can easily imagine some “service modules,” like “Mathematics for Engineers,” that many be offered by the School of Mathematics to several other schools, say the School of Computer Science, the School the Engineering, and the School of Agriculture. Then a professor may offer the same module several times in the same semester, just for different student groups.

Listing 19.146: The generated script to create the table `course` which relates the tables `professor` and `module`. (src)

```

1 -- object: public.course | type: TABLE --
2 -- DROP TABLE IF EXISTS public.course CASCADE;
3 CREATE TABLE public.course (
4     id integer NOT NULL GENERATED BY DEFAULT AS IDENTITY ,
5     professor integer NOT NULL,
6     module integer NOT NULL,
7     semester integer NOT NULL,
8     CONSTRAINT course_id_pk PRIMARY KEY (id)
9 );
10 -- ddl-end --
11 ALTER TABLE public.course OWNER TO postgres;
12 -- ddl-end --

```

Listing 19.147: The generated script to create the table `enrolls` which relates the tables `student` and `course`. (src)

```

1 -- object: public.enrolls | type: TABLE --
2 -- DROP TABLE IF EXISTS public.enrolls CASCADE;
3 CREATE TABLE public.enrolls (
4     student integer NOT NULL,
5     course integer NOT NULL,
6     CONSTRAINT enrolls_pk PRIMARY KEY (student, course)
7 );
8 -- ddl-end --
9 ALTER TABLE public.enrolls OWNER TO postgres;
10 -- ddl-end --

```

Listing 19.148: The generated script to create the constraint enforcing that each row in table `course` is related to exactly one row in table `professor`. (src)

```

1 -- object: course_professor_fk | type: CONSTRAINT --
2 -- ALTER TABLE public.course DROP CONSTRAINT IF EXISTS course_professor_fk
   ↪ CASCADE;
3 ALTER TABLE public.course ADD CONSTRAINT course_professor_fk FOREIGN KEY (
   ↪ professor)
4 REFERENCES public.professor (id) MATCH SIMPLE
5 ON DELETE NO ACTION ON UPDATE NO ACTION;
6 -- ddl-end --

```

We also enforce that each record in the table `enrolls` must relate exactly one row in table `student` and one row in table `course`. We use the combination of these two foreign keys as primary key. In other words, no student can enroll in the same course more than once. Courses are realizations of modules that emerge because a professor teaches them in a specific semester. So it makes no sense that a student takes part in the same module taught by the same professor in the same semester more than once. They can, however, take part in the same module in *other* semesters. Of course, students can also enroll into multiple different courses.

Exporting this model to SQL yields ten scripts. Listing 19.142 sets up the DB. Listing 19.143 is used to create the table `student` for the *Student* entities. It has a surrogate primary key `id` and stores the names of the students as variable-length strings.

The table `professor` is created by the script shown in Listing 19.144. This table has exactly the same structure as the table `student`. The table `module` has again the same structure, with the exception that it stores a `title` instead of a `name`, as shown in Listing 19.145.

The table `course`, constructed by Listing 19.146 is a bit more interesting. It, too, has a surrogate primary key `id`. Apart from this, it also stores two foreign keys – `professor` and `module` – that

Listing 19.149: The generated script to create the constraint enforcing that each row in table `course` is related to exactly one row in table `module`. (src)

```

1 -- object: course_module_fk | type: CONSTRAINT --
2 -- ALTER TABLE public.course DROP CONSTRAINT IF EXISTS course_module_fk
   ↪ CASCADE;
3 ALTER TABLE public.course ADD CONSTRAINT course_module_fk FOREIGN KEY (
   ↪ module)
4 REFERENCES public.module (id) MATCH SIMPLE
5 ON DELETE NO ACTION ON UPDATE NO ACTION;
6 -- ddl-end --

```

Listing 19.150: The generated script to create the constraint enforcing that each row in table `enrolls` is related to exactly one row in table `student`. (src)

```

1 -- object: enrolls_student_fk | type: CONSTRAINT --
2 -- ALTER TABLE public.enrolls DROP CONSTRAINT IF EXISTS enrolls_student_fk
   ↪ CASCADE;
3 ALTER TABLE public.enrolls ADD CONSTRAINT enrolls_student_fk FOREIGN KEY (
   ↪ student)
4 REFERENCES public.student (id) MATCH SIMPLE
5 ON DELETE NO ACTION ON UPDATE NO ACTION;
6 -- ddl-end --

```

Listing 19.151: The generated script to create the constraint enforcing that each row in table `enrolls` is related to exactly one row in table `course`. (src)

```

1 -- object: enrolls_course_fk | type: CONSTRAINT --
2 -- ALTER TABLE public.enrolls DROP CONSTRAINT IF EXISTS enrolls_course_fk
   ↪ CASCADE;
3 ALTER TABLE public.enrolls ADD CONSTRAINT enrolls_course_fk FOREIGN KEY (
   ↪ course)
4 REFERENCES public.course (id) MATCH SIMPLE
5 ON DELETE NO ACTION ON UPDATE NO ACTION;
6 -- ddl-end --

```

point to the tables of the same names, respectively. This is enforced by the two constraints given in Listings 19.148 and 19.149, again respectively. Additionally, it stores the attribute `semester` as integer number. We will use the “year * 10” plus 2 for the Fall semester and 1 for the Spring semester.

The last of the five tables is `enrolls` defined by Listing 19.147. This table is different: It does *not* have a surrogate primary key. It has two foreign keys – `student` and `course` – pointing to tables `student` and `course`, respectively. They are enforced via constraints created in Listings 19.150 and 19.151. And they, together, form the primary key. This means that the pairs of `(student, course)` must be `UNIQUE`. This means that the same student cannot enroll multiple times into the same course. Which makes a lot of sense.

After executing these scripts, we insert some data into the tables in Listing 19.152. We define the three students Bibbo, Beppo, and Bebba. Two professors are declared, namely Weise (yours truly) and Bobbo. We enter three modules, namely *Python*, *Databases*, and *Java*. Via the `courses` table, we specify that Prof. Weise teaches *Python* and *Databases*, both in semester 20252, which we interpret as the Fall semester in the year 2025. We also define that Prof. Bobbo teaches *Java* in the spring and fall semesters 2026.

Mr. Bibbo enrolls into Prof. Bobbo’s *Java* class as well as into both classes taught by Prof. Weise. Mr. Beppo takes *Java* and *Python*. Finally, Mrs. Bebba enrolls into *Java* and *Databases*.

Using four `INNER JOIN` expressions, we then merge all the data together. We order the results by the student name, professor names, module titles, and semesters. After all of that, we delete the DB again using `DROP DATABASE`.

In this example, we noticed that we could realize the conceptual ternary relationship given in Figure 19.27.1 in two different ways in a logical schema. In other scenarios, there may yet be other choices. It does not make much sense to iterate over all the $\frac{(4+3-1)!}{(4-1)!*3!} = 20$ possible ternary relationships here. Then we would have to also iterate over all 35 possible relationships that involve four entity types, all 56 relationship patterns of five entity types, and so on. We should also remember that relationship attributes may be present, which could mess up the patterns further.

The somewhat unsatisfying answer on how to implement relationship patterns of more than two entity types is *it depends*. We are equipped with the ability to enforce referential integrity of arbitrary binary relationship patterns. We can use this knowledge to reasonably realize patterns that are more complicated. We just have to build experience.

Either way, relationships of higher order will most often be broken down to several binary relationships. And we do know how to implement these.

Listing 19.152: Inserting into and selecting data from the tables in the `teaching_database`. (stored in file `insert_and_select.sql`; output in Listing 19.153)

```

1  /** Insert data into the teaching database and then merge it. */
2
3  -- Insert several student records.
4  INSERT INTO student (name) VALUES ('Bibbo'), ('Bebbo'), ('Bebba');
5
6  -- Insert several professor records.
7  INSERT INTO professor (name) VALUES ('Weise'), ('Bobbo');
8
9  -- Insert several module records.
10 INSERT INTO module (title) VALUES ('Python'), ('Databases'), ('Java');
11
12 -- Create the courses, i.e., the instances of the modules in semesters.
13 INSERT INTO course (professor, module, semester) VALUES
14     (1, 1, 20252), (1, 2, 20252), (2, 3, 20261), (2, 3, 20262);
15
16 -- Enroll students into the courses.
17 INSERT INTO enrolls (student, course) VALUES
18     (1, 1), (1, 2), (1, 3), (2, 1), (2, 4), (3, 2), (3, 3);
19
20 -- Print the enrollment information.
21 SELECT student.name AS student, professor.name AS teacher,
22       module.title AS module, semester FROM enrolls
23   INNER JOIN student ON enrolls.student = student.id
24   INNER JOIN course ON enrolls.course = course.id
25   INNER JOIN professor ON course.professor = professor.id
26   INNER JOIN module ON course.module = module.id
27 ORDER BY student.name, professor.name, module.title, semester;

```

Listing 19.153: The stdout of the program `insert_and_select.sql` given in Listing 19.152.

```

1  $ psql "postgres://postgres:XXX@localhost/teaching_database" -v
2      → ON_ERROR_STOP=1 -ebs insert_and_select.sql
3  INSERT 0 3
4  INSERT 0 2
5  INSERT 0 3
6  INSERT 0 4
7  INSERT 0 7
8  student | teacher | module    | semester
9  -----+-----+-----+-----
10 Bebba  | Bobbo   | Java      | 20261
11 Bebba  | Weise   | Databases | 20252
12 Bebbo  | Bobbo   | Java      | 20262
13 Bebbo  | Weise   | Python    | 20252
14 Bibbo  | Bobbo   | Java      | 20261
15 Bibbo  | Weise   | Databases | 20252
16 Bibbo  | Weise   | Python    | 20252
17 (7 rows)
18 # psql 16.12 succeeded with exit code 0.

```

19.2.3.5 Summary

At this stage, we now have the means to translate conceptual models to logical models that follow the relational principle. In **relational databases**, all the data is stored in tables. Each table as a primary key. One table can refer to another table via columns that can store the primary keys of that other table. These columns then are called the foreign keys. The referential integrity between the tables is maintained via **REFERENCES**, **NOT NULL**, and **UNIQUE** constraints. We have learned how to map weak entities, relationship attributes, derived attributes, and relationships of higher degrees to this data model as well. With this, we have the tools to transform all the elements that can draw into a **entity relationship diagram (ERD)** to **SQL** code.

19.3 Normalization

Designing the logical schema for a relational database is not just a transformation of a conceptual schema to SQL. While we now are able to translate all the elements of an ERD illustrating a conceptual model to a logical model, this does not necessarily mean that the resulting model will be efficient and well-designed. In order to achieve this, several guidelines and best practices can be followed. Some of the most important of these concern *normalization* [118, 157].

Normalization is a process that aims at minimizing redundancy and avoiding inconsistencies and anomalies [397, 398]. It does so usually at the trade-off of data retrieval speed: Data which, in unnormalized form, could be stored in a single table needs to be reassembled using **INNER JOIN** and similar constructs combining multiple tables in normalized form [242]. Therefore, whether to normalize data and to which degree is a question always to be answered with performance in mind [242].

There exist several **normal form (NF)**. Out of these, we shall here consider the **first normal form (1NF)**, the **second normal form (2NF)**, and the **third normal form (3NF)**. Higher normal forms are more restrictive. Therefore, if a part of a logical model is in a higher normal form, then it is also in all of the lower normal forms.

19.3.1 First Normal Form

The **first normal form (1NF)** dates back to Codd's seminal paper [90] where he presented the relational data model back in 1970. In its essence, it prescribes that the design of tables must follow the relational data model.

Definition 19.7: First normal form (1NF)

Under the **1NF**, all rows in a table must have the same number of fields and all fields must be atomic.

This excludes multivalued attributes as well as composite attributes. As we already discussed before, the relational data model does not support such attributes anyway. So why does this normal form even exist? If all tables would fulfill it by default, then there would be no need to even discuss it.

In Section 19.2.1, we discussed how entity types in the conceptual schema are translate to tables in the logical schema based on the requirements of the relational data model. We stated that multivalued attributes become separate tables. Composite attributes need to be recursively broken down into their atomic components, which then become separate columns. If we use the relational data model, then we would naturally produce logical models in 1NF.

However, this is only true if we *recognize* multivalued attributes and composite attributes as such. If a table does have attributes that are semantically composite but we implement them as single flat attributes, then it violates the 1NF. If an attribute of an entity is multivalued, but instead of placing it into an additional table, we try to represent it using multiple columns in the table for entity, we violate the 1NF. Let us explore what happens if we violate the 1NF.

19.3.1.1 Composite Attributes

Violation: The Use of Composite Attributes In Figure 19.28, we illustrate a part of a logical model that relates student records to address records. In this model, each student has exactly one address and a name. Addresses are text stored in a separate table.

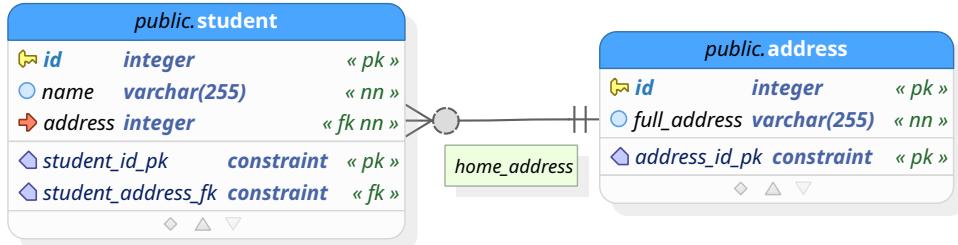


Figure 19.28: A violation of the 1NF: The `full_address` field is semantically a composite attribute, but represented as flat `VARCHAR`.

Listing 19.154: The generated SQL code for creating the `address` table that violates the 1NF based on Figure 19.28. (src)

```

1  -- object: public.address | type: TABLE --
2  -- DROP TABLE IF EXISTS public.address CASCADE;
3  CREATE TABLE public.address (
4      id integer NOT NULL GENERATED BY DEFAULT AS IDENTITY ,
5      full_address varchar(255) NOT NULL,
6      CONSTRAINT address_id_pk PRIMARY KEY (id)
7  );
8  -- ddl-end --
9  ALTER TABLE public.address OWNER TO postgres;
10 -- ddl-end --

```

Listing 19.155: The generated SQL code for creating the `student` table based on Figure 19.28. (src)

```

1  -- object: public.student | type: TABLE --
2  -- DROP TABLE IF EXISTS public.student CASCADE;
3  CREATE TABLE public.student (
4      id integer NOT NULL GENERATED BY DEFAULT AS IDENTITY ,
5      name varchar(255) NOT NULL,
6      address integer NOT NULL,
7      CONSTRAINT student_id_pk PRIMARY KEY (id)
8  );
9  -- ddl-end --
10 ALTER TABLE public.student OWNER TO postgres;
11 -- ddl-end --

```

The auto-generated script Listing 19.154 creates the table `address`. It has two columns, namely the surrogate primary key `id` and the column `full_address`, which is a variable-length string of up to 255 characters. As the name suggests, we will store the addresses of the students in this table.

The script Listing 19.155 then creates the table `student`. This table, too, has a surrogate primary key `id`. It also sports the column `name` storing the name of the students. Column `address` is a foreign key reference to the `id` column of table `address`. This is secured by a `REFERENCES` constraint that we create with another script. We do not print that one here and neither do we print the script for generating the DB, as they do not contribute much to the understanding of the scenario.

We then insert some data into them DB Listing 19.156. We first create four address records, one for our Hefei University (合肥大学) located in the beautiful city of Hefei (合肥市) in China, one address in my hometown Chemnitz, Germany, one address located in the Chinatown of New York, USA, and, finally, one address in Quanzhou city (泉州市), China. We then create four `student` records for Mr. Bibbo, Mr. Beppo, Mrs. Bibbi, and Mr. Beppo. Via their foreign key, these are linked to the above addresses in that order. At first glance, all looks well.

And all could be well, if we would treat the address of a student always as a single unstructured text string. However, this is not necessarily true, especially not true in our teaching management platform example. In our example, Mr. Bibbo lives directly in our Hefei University whereas Mr. Babbo comes from Quanzhou (泉州市) in the Fujian province (福建省). Mr. Beppo and Mrs. Bibbi, however, are

Listing 19.156: Inserting some data into the tables `student` and `address` in violation of the 1NF based on Figure 19.28. (src)

```

1  /** Insert data into the database. */
2
3  -- Insert several address records.
4  INSERT INTO address (full_address) VALUES
5      ('Hefei University, Hefei 230601 Jinkaiqu, Hefei, Anhui, China'),
6      ('Am Rathaus 1, 09111 Zentrum, Chemnitz, Sachsen, Deutschland'),
7      ('Canal Street 4, Chinatown, New York, NY, USA'),
8      ('West Street, Licheng District, Quanzhou 362002, Fujian, PRC');
9
10 -- Create the student records.
11 INSERT INTO student (name, address) VALUES
12     ('Babbo', 1), ('Bebbo', 2), ('Bibbi', 3), ('Babbo', 4);

```

foreign exchange students (留学生) from Germany and the USA, respectively. Assume that this table would be much larger. What would happen if we wanted to know who of our students have a valid address in China? How would we do that?

Matter of fact, we encountered this very same situation back in Section 9.2.2. Back then, we used the `ILIKE` expression [323] and we do so again here: In Listing 19.157, we combine the tables `student` and `address` by using an `INNER JOIN` statement. We then only keep the rows `WHERE full_address ILIKE '%china%'`. In other words, we retain only the rows where the word "china" occurs anywhere in the `full_address` columns, regardless of its case. "China," "china," "CHINA," "cHina" – all are OK. Doing this will yield the two students Mr. Bibbo and Mrs. Bibbi. Ms. Bibbi, however, is a foreign exchange student. She lives in *Chinatown*, New York. Also, Mr. Babbo was not listed, as he declared his address to be in the PRC, i.e., the People's Republic of China (中华人民共和国).

We are faced with two problems: The first one is that we have no method to decide what part of the `full_address` is the country and what not. The second problem is that there are many different ways to declare that the country of an address is China.

The first problem is caused directly by our violation of the 1NF. Here, we did not model the attribute for the address as a composite attribute. We modelled it as an atomic attribute, which turned out to be wrong, because now we want to access its components. An atomic attribute does not have components. So the error did occur in the conceptual modeling phase. It became apparent only after we finished implementing the logical model.

Now our DB can still "work". We can construct the second query in Listing 19.157, which deals with both of the special cases mentioned above. We exclude addresses that have China in their text but also Chinatown. And we include addresses that mention PRC and, for good measures, also those including P.R.C. These, however, are only crutches and no solutions. We can easily imagine addresses that still will be misclassified. For example, what do we do with the "Embassy of China in Berlin, Germany"?

Repair: Disassembling the Composite Attribute Let us now fix this problem. A proper solution can only be to model the address as a composite attribute. At least the country needs to be split off. Maybe also the province, because that could come in handy, too. Adn while we are at it, we probably also want to know the city and postal code. The components of this composite attribute then will become separate columns in a table. We apply these ideas to create the improved logical model in Figure 19.29.

The attribute `full_address` now longer exists when we create the table `address` in Listing 19.159. Instead, we have the columns `country`, `province`, `city`, `postal_code`, and `street_address`. All of them are of type `VARCHAR` of appropriate lengths. We permit `province` to be `NULL`, because some countries maybe do not have provinces. All other fields must be `NOT NULL`. Nothing else changes, the table `student` can stay as it is. We thus do not reproduce its creation here as a listing.

When we insert the data into our DB Listing 19.160, we of course also need to split the addresses properly over the columns. This also shows us a slight drawback that is inherent to all normal forms:

Listing 19.157: Trying to find all the students with an address in China, which is hard, because table `address` violates the 1NF. (stored in file `select.sql`; output in [Listing 19.158](#))

```

1  /** Get a list of students from China. */
2
3  -- Fails to get addresses from the PRC and includes addresses from
4  -- Chinatown, New York.
5  SELECT name, full_address AS address_attempt_1 FROM student
6    INNER JOIN address ON student.address = address.id
7    WHERE full_address ILIKE '%china%';
8
9  -- Gets everything right, but is still error prone.
10 -- For examples, what if China was written in Chinese?
11 -- What with other words or names that include China?
12 SELECT name, full_address AS address_attempt_2 FROM student
13   INNER JOIN address ON student.address = address.id
14   WHERE ((full_address ILIKE '%china%') AND NOT
15     (full_address ILIKE '%chinatown%'))
16   OR (full_address ILIKE '%PRC%')
17   OR (full_address ILIKE '%P.R.C.%');

```

Listing 19.158: The `stdout` of the program `select.sql` given in [Listing 19.157](#).

```

1  $ psql "postgres://postgres:XXX@localhost/anomalies" -v ON_ERROR_STOP=1 -
2    ↪ ebf select.sql
3
4  name | address_attempt_1
5  -----+
6  Bibbo | Hefei University, Hefei 230601 Jinkaiqu, Hefei, Anhui, China
7  Bibbi | Canal Street 4, Chinatown, New York, NY, USA
8  (2 rows)
9
10 name | address_attempt_2
11 -----+
12 Bibbo | Hefei University, Hefei 230601 Jinkaiqu, Hefei, Anhui, China
13 Babbo | West Street, Licheng District, Quanzhou 362002, Fujian, PRC
14 (2 rows)
15
16 # psql 16.12 succeeded with exit code 0.

```

They break compound data into independent pieces. If we later need the complete data again, we need to reassemble the pieces. Thus, if we need the full address string, we first must reassemble it, probably using the string concatenation operator `||` [\[424\]](#).

Anyway. This time, we create five student records, adding Ms. Bebbe to the mix. The addresses of the other students stay basically the same, but are broken down into their components. Ms. Bebbe lives in somewhere Beijing (北京), China.

As you can see in [Listing 19.162](#), we now can indeed obtain the list of all students with addresses in China much more easily. It is a given that we still have to deal with the fact that different people may use different names for the country. But at least we cannot accidentally classify someone from Chinatown in San Francisco as a PRC resident.

While we are here, let's do a small excursion that just fits nicely in this topic but is otherwise unrelated to the **1NF**. If you read [Listing 19.161](#), you notice that reassembling the full address was a bit complicated and went beyond simply using `||`. This is because we allowed the `province` column to be `NULL`.

We even have such a case in our table: A new student, Ms. Bebbe, has joined our university and she is from Beijing (北京市). Beijing that does not belong to any province but is a municipality directly under the central government of China. Therefore, when adding her address record, we left the `province` column as `NULL`.

In PostgreSQL, concatenating a string with `NULL` yields `NULL`. If we just combined all the address

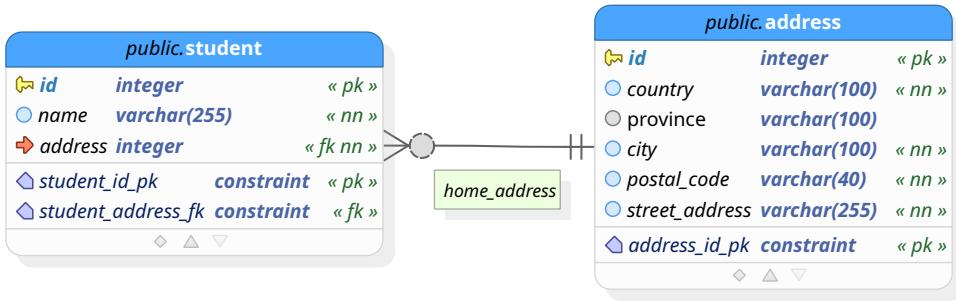


Figure 19.29: A new variant of Figure 19.28 that does no longer violate the 1NF. The address data has been disassembled in several columns, allowing us to extract the country of an address with ease.

Listing 19.159: The generated SQL code for creating the `address` table based on Figure 19.29, which no longer violates the 1NF. (src)

```

1 -- object: public.address | type: TABLE --
2 -- DROP TABLE IF EXISTS public.address CASCADE;
3 CREATE TABLE public.address (
4     id integer NOT NULL GENERATED BY DEFAULT AS IDENTITY ,
5     country varchar(100) NOT NULL,
6     province varchar(100),
7     city varchar(100) NOT NULL,
8     postal_code varchar(40) NOT NULL,
9     street_address varchar(255) NOT NULL,
10    CONSTRAINT address_id_pk PRIMARY KEY (id)
11 );
12 -- ddl-end --
13 ALTER TABLE public.address OWNER TO postgres;
14 -- ddl-end --

```

Listing 19.160: Inserting some data into the tables `student` and `address` as designed in Figure 19.29, which now comply with the 1NF. (src)

```

1 /** Insert data into the database. */
2
3 -- Insert several address records.
4 INSERT INTO address (
5     country, province, city, postal_code, street_address) VALUES
6     ('China', 'Anhui', 'Hefei', '230601', 'Jinkaiqu, Hefei University'),
7     ('Deutschland', 'Sachsen', 'Chemnitz', '09111', 'Am Rathaus 1'),
8     ('USA', 'NY', 'New York', '100013', 'Canal Street 4, Chinatown'),
9     ('PRC', 'Fujian', 'Quanzhou', '362002', 'West Street'),
10    ('P.R.C.', NULL, 'Beijing', '100084', 'Tsinghua University');
11
12 -- Create the student records.
13 INSERT INTO student (name, address) VALUES
14     ('Bibbo', 1), ('Bebbo', 2), ('Bibbi', 3), ('Babbo', 4), ('Bebbe', 5);

```

fields using `||`, we would yield `NULL` for the address of Ms. Bebbe. To deal with the potentially `NULL` in the `province` field, we use the `COALESCE` function [97]. This function takes arbitrarily many arguments and returns the first argument that is not `NULL` (or `NULL` if all of its arguments are `NULL`).

When reading the query, you will also find one additional change when checking the country: We could have used the logical `OR` to combine the three conditions `country ILIKE '%china%'`, `country ILIKE '%PRC%'`, and `country ILIKE '%P.R.C.%'`. Instead, we wrote `country ILIKE ANY(ARRAY['%china%', '%PRC%', '%P.R.C.%'])`, which is equivalent to that [9, 368]: We can declare an array of the values `a`, `b`, `c`, and `d` via `ARRAY[a, b, c, d]`. The expression `XXX operator ANY(ARRAY[...])` becomes `TRUE` if `XXX operator YYY` is `TRUE` for any, i.e., at least one,

Listing 19.161: Trying to find all the students with an address in China becomes easier now, because `country` is its own column. Also, reassembling the full address using the string concatenation operation `||`. (stored in file `select.sql`; output in [Listing 19.162](#))

```

1  /** Get a list of students from China. */
2
3  -- Much easier due to 1NF, but still a bit problematic due to multiple
4  -- names for same country.
5  SELECT name, country || ', ' || COALESCE(province || ', ', '') ||
6      postal_code || ' ' || city || ', ' ||
7      street_address AS address FROM student
8  INNER JOIN address ON student.address = address.id
9  WHERE country ILIKE ANY(ARRAY['%china%', '%PRC%', '%P.R.C.%']);

```

Listing 19.162: The `stdout` of the program `select.sql` given in [Listing 19.161](#).

```

1  $ psql "postgres://postgres:XXX@localhost/fixed" -v ON_ERROR_STOP=1 -e bf
2      ↪ select.sql
3
4  name | address
5  -----+
6  Bibbo | China, Anhui, 230601 Hefei, Jinkaiqu, Hefei University
7  Babbo | PRC, Fujian, 362002 Quanzhou, West Street
8  Bebbe | P.R.C., 100084 Beijing, Tsinghua University
9  (3 rows)

# psql 16.12 succeeded with exit code 0.

```

`YYY` in the array [368]. In our case, `XXX` is `country` and `operator` is `ILIKE`. (Similarly, the expression `XXX operator ALL(ARRAY[...])` becomes `TRUE` if `XXX operator YYY` is `TRUE` for every single, i.e., all, `YYY` in the array [368].) Thus, we can use the shorter expression to save a bit space.

Anyway, we have seen one anomaly that can occur when designing logical schemas. If we incorrectly model composite attributes as atomic attributes and later need to pull the components out of them, we can quickly descend into the hell of crutches and special cases. Properly recognizing the nature of attributes and strictly adhering to the [1NF](#) can reduce the potential errors very significantly. It comes at the cost of slightly more complicated queries when reassembling the compound data.

19.3.1.2 The Use of Multivalued Attributes

Violation: The Use of Multivalued Attributes Let us continue our example from the previous section. There, we developed a two-table structure for storing addresses of students. Each student had exactly one address. But maybe in reality, students can have more than one address. Let's say their current address, maybe in the university dormitory in a flat rented nearby, and their old home address, i.e., the flat of their parents. In the model from the previous section, this cannot be implemented. In [Figure 19.30](#), the DB developer had a very simple idea: Let's just have two columns for the addresses – `address_1` and `address_2` – in the table `student`. By doing so, they have violated the [1NF](#).

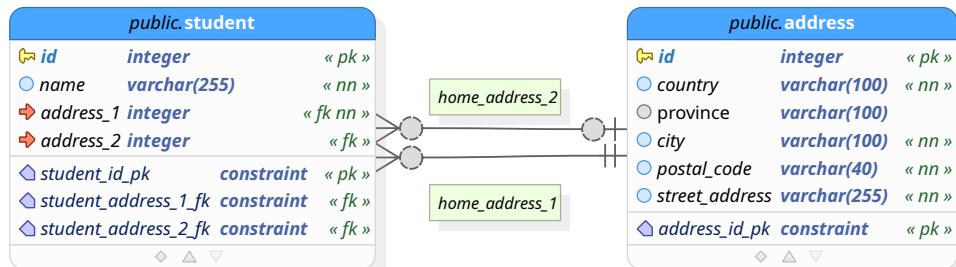


Figure 19.30: A violation of the [1NF](#): The `student` table has two columns with the same semantic, i.e., a repeating group. Both columns reference addresses to simulate a multivalued attribute.

Listing 19.163: The generated SQL code for creating the `student` table with two columns for addresses based on [Figure 19.30](#), which violates the 1NF. The corresponding `REFERENCES` constraints have been omitted here for the sake of brevity. ([src](#))

```

1 -- object: public.student | type: TABLE --
2 -- DROP TABLE IF EXISTS public.student CASCADE;
3 CREATE TABLE public.student (
4     id integer NOT NULL GENERATED BY DEFAULT AS IDENTITY ,
5     name varchar(255) NOT NULL,
6     address_1 integer NOT NULL,
7     address_2 integer,
8     CONSTRAINT student_id_pk PRIMARY KEY (id)
9 );
10 -- ddl-end --
11 ALTER TABLE public.student OWNER TO postgres;
12 -- ddl-end --

```

Listing 19.164: Inserting some data into the tables `student` and `address` in violation of the 1NF based on [Figure 19.30](#). ([src](#))

```

1 /** Insert data into the database. */
2
3 -- Insert several address records.
4 INSERT INTO address (
5     country, province, city, postal_code, street_address) VALUES
6     ('China', 'Anhui', 'Hefei', '230601', 'Jinkaiqu, Hefei University'),
7     ('China', 'Anhui', 'Hefei', '230026', 'USTC'),
8     ('Deutschland', 'Sachsen', 'Chemnitz', '09111', 'Am Rathaus 1'),
9     ('USA', 'NY', 'New York', '100013', 'Canal Street 4, Chinatown'),
10    ('Deutschland', 'Sachsen', 'Chemnitz', '09111', 'TU Chemnitz'),
11    ('PRC', 'Fujian', 'Quanzhou', '362002', 'West Street'),
12    ('P.R.C.', NULL, 'Beijing', '100084', 'Tsinghua University'),
13    ('Spain', 'Andalusia', 'Granada', '18009', 'Alhambra de Granada');
14
15
16 -- Create the student records.
17 INSERT INTO student (name, address_1, address_2) VALUES
18     ('Bibbo', 1, 2), ('Bebbo', 3, NULL), ('Bibbi', 4, NULL),
19     ('Babbo', 5, 6), ('Bebbe', 7, 8);

```

Listing 19.165: Trying to find all the students with at least one address in China, which is harder than necessary, because table `student` violates the 1NF. (stored in file `select.sql`; output in Listing 19.166)

```

1  /** Get a list of students with at least one address in China. */
2
3  -- Select the student id, student name, and address as three columns.
4  SELECT name, address_1 AS adr FROM student
5  UNION SELECT name, address_2 AS adr FROM student
6  WHERE address_2 IS NOT NULL;
7
8  -- Use the above as *subquery* to construct the overall result.
9  SELECT name, city || ', ' || street_address AS address FROM (
10      SELECT name, address_1 AS adr FROM student
11      UNION SELECT name, address_2 AS adr FROM student)
12  INNER JOIN address ON adr = address.id
13  WHERE country ILIKE ANY(ARRAY['%china%', '%PRC%', '%P.R.C.%']);
14
15  -- Remove double student entries.
16  SELECT DISTINCT ON (sid)
17      name, city || ', ' || street_address AS address FROM (
18          SELECT id AS sid, name, address_1 AS adr FROM student
19          UNION SELECT id AS sid, name, address_2 AS adr FROM student)
20  INNER JOIN address ON adr = address.id
21  WHERE country ILIKE ANY(ARRAY['%china%', '%PRC%', '%P.R.C.%']);

```

Listing 19.166: The stdout of the program `select.sql` given in Listing 19.165.

```

1  $ psql "postgres://postgres:XXX@localhost/anomalies" -v ON_ERROR_STOP=1 -
2  ↪ ebf select.sql
3
4  name | adr
5  -----+-----
6  Bebbe | 7
7  Babbo | 5
8  Bibbo | 1
9  Bibbo | 2
10  Bebbo | 3
11  Bebbe | 8
12  Bibbi | 4
13  Babbo | 6
14  (8 rows)
15
16  name | address
17  -----+-----
18  Bebbe | Beijing, Tsinghua University
19  Bibbo | Hefei, Jinkaiqu, Hefei University
20  Bibbo | Hefei, USTC
21  Babbo | Quanzhou, West Street
22  (4 rows)
23
24  name | address
25  -----+-----
26  Bibbo | Hefei, USTC
27  Babbo | Quanzhou, West Street
28  Bebbe | Beijing, Tsinghua University
29  (3 rows)
# psql 16.12 succeeded with exit code 0.

```

The table `student` is created based on this model in Listing 19.163. As you can see, we indeed have two foreign keys (`address_1` and `address_2`) now pointing to table `address`. We omitted printing the scripts for generating the foreign key `REFERENCES` constraints for the sake of brevity. We also did not print the SQL script for creating the table `address`, since it remains the same as in the last section.

This violation of the 1NF can cause various problems. First, there are design-level problems. What, for example, will we do if a student needs a third address? This is easily conceivable, maybe a student has parents who live separately, so they have two home addresses and one flat rented near the university. Sticking to the repeating-groups method, we would need to add a columns `address_3`. What if we have a person with four addresses? Will we keep adding columns when special cases that require more addresses appear? However, such a modification is not just a single change. There could be various queries and applications that make use of the address columns. We would need to update every single one of them.

Another problem is how do we know how many addresses a student has? In our logical model shown in Figure 19.30 and in its implementation in Listing 19.163, we constrained the `address_1` column to be `NOT NULL`. Column `address_2` must of course be permitted to be `NULL`. Therefore, a student has either one or two addresses. Let's that we extended our model to three addresses. Then it could happen that we get a row where the second address column is `NULL` while the third address column is not, or the other way around. So just to know the number of addresses, we would have a somewhat complex query.

Well, since we have multiple addresses now our queries get more complicated anyway. However, due to the repeated group emulating a multivalued attribute, they become even more complex. For example: What do we do if we want a list of students who have at least one address in China?

Before we try to construct such a query, let us first insert some example data into our DB in Listing 19.164. Mr. Bibbo now has two addresses in Hefei, one at our Hefei University (合肥大学) and one at the University of Science and Technology of China (中国科学技术大学, USTC). Mr. Beppo still has only one address and lives in Chemnitz city in Germany. Mrs. Bibbi lives only in Chinatown, New York, USA. Mr. Babbo, too, has an address in Chemnitz city, Germany, but also a secondary address in Quanzhou (福建省泉州市), China. The first address of Ms. Bebbe is in Beijing (北京), but she also has a secondary address in Spain. The example covers all the possible cases in this constellation: Some people have no address in China, for some the first address in China, for some the second address is in China, and for some, both addresses are in China. From the five people, clearly Mr. Bibbo, Mr. Babbo, and Ms. Bebbe have an address in China.

So how do we get the list of these three people? Regardless of how we slice this problem, we will need to somehow apply the same expression to both address columns. In Listing 19.165, we therefore begin by figuring out how to get the data into the shape of a relation that has a one student name and one corresponding address foreign key value. This can be done with a `UNION` query. The first query that we try out is thus one that selects the student `name` and the first address column `address_1`, which we rename to `adr` via `AS`. It then appends the result of a second query that does almost the same, but uses `address_2` as `adr`. The two queries are combined with the `UNION` keyword. As you can see in Listing 19.166, this produces eight rows. Actually, this is relation is in the 1NF.

Having re-created the 1NF, we can now go about selecting the people with addresses in China. We can do this almost exactly as in the previous section in Listing 19.161, by using an `INNER JOIN` and the array-based `ILIKE`. The difference is that we now cannot use the `student` table as data source. Instead, we use the `UNION` query that we constructed above as a *subquery*. In SQL, you can also use the result of another `SELECT...FROM` as source for a query instead of a table. All that needs to be done is to write the subquery in parentheses `(...)`!

The second query in Listing 19.165 therefore looks very much like the query that we designed in Listing 19.161. The only difference is that we have replaced the query source, the table `student`, with our `UNION` query in parentheses (and we renamed some columns). This works well, with one exception: As Listing 19.166 shows, Mr. Bibbo is listed twice. He has two addresses in China.

Luckily, the `DISTINCT` keyword exists [386]. `SELECT DISTINCT` deletes all duplicate rows resulting from a query. In other words, if two or more rows have all the same values, only one of them is preserved and the remaining rows are dropped. `SELECT DISTINCT ON (col1, col2, ...)` uses only a subset of columns (here `col1`, `col2`, ...) to decide whether a row is a duplicate or not. Well, names may be ambiguous. We therefore choose to change our subqueries to also pass along the student IDs (renamed to `sid`). By including a `DISTINCT ON (sid)` at the beginning of our query, we ensure that each student will only appear once in the result. The third query in Listing 19.165 and its result in Listing 19.166

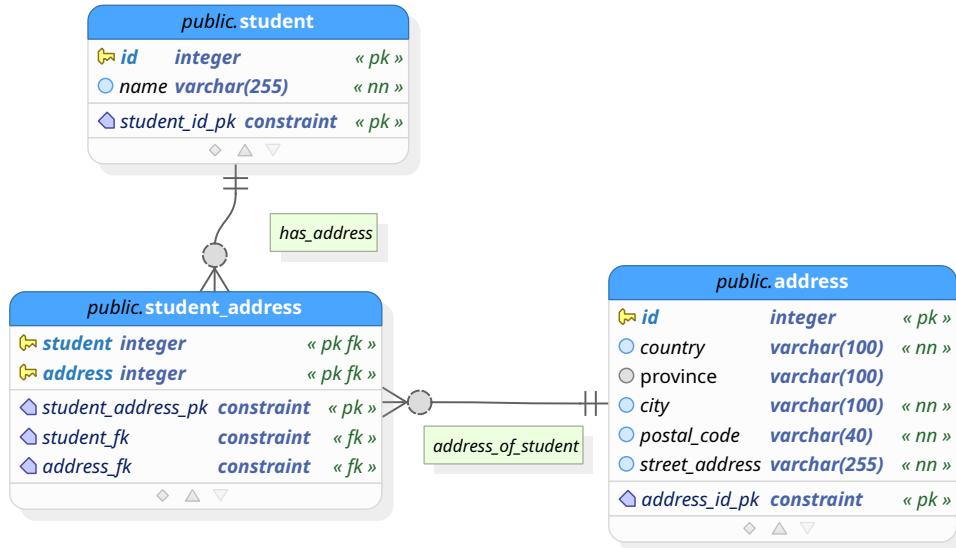


Figure 19.31: A redesign of the logical schema from Figure 19.30. Refactoring the multivalued attribute that was represented as repeating group into a separate table brings the model into the 1NF.

Listing 19.167: The generated SQL code for creating the `student` table, which now only has a primary key and the student name stored. ([src](#))

```
1 -- object: public.student | type: TABLE --
2 -- DROP TABLE IF EXISTS public.student CASCADE;
3 CREATE TABLE public.student (
4     id integer NOT NULL GENERATED BY DEFAULT AS IDENTITY ,
5     name varchar(255) NOT NULL,
6     CONSTRAINT student_id_pk PRIMARY KEY (id)
7 );
8 -- ddl-end --
9 ALTER TABLE public.student OWNER TO postgres;
10 -- ddl-end --
```

are now correct.

What we learned here is this: If we want to do anything useful with multivalued attributes represented as repeating groups ... then we have to rewire them be in **1NF**. This means that we need to represent them as separate relations, that is, in the form that we already learned back in [Section 19.2.1](#) (Mapping Conceptual Entity Types to Logical Models). So why not store them as such directly? The model in [Figure 19.30](#) is just a wrongheaded representation of a `student` $\triangleright\!\!\!\rightarrow\!\!\! \leftarrow$ `address` relationship. We learned about this in relationship pattern [Section 19.2.2.9](#) ($Q \triangleright\!\!\!\rightarrow\!\!\! \leftarrow R$).

Repair: Extracting Multivalued Attributes to Separate Table So let us fix this problem by extracting the multivalued attribute into a separate table. For the sake of simplicity, we will not enforce that each student must have at least one address. We instead create an $\text{student} \rightarrowtail \text{address}$ pattern, as discussed back in Section 19.2.2.8 ($O \rightarrowtail P$). But you do know how we could do this fully beautiful, so I hope that's OK.

As illustrated in Figure 19.31, we now need a third table, which we will call `student_address`. This table will have a composite primary key consisting of one foreign key reference to the table `student` and one foreign key reference to the table `address`. The table `student` is now reduced and its `address` attributes are removed. It now only has its primary key `id` and the student `name` attribute left. The table `address` stays as it is.

We omit the script for creating the DB. We also do not show the script for creating the table `address`, because it again was not changed. In Listing 19.167, we create the table `student`. It has been modified based on our new model and now really only has the two columns `id` and `name`. The table `student_address`, created via Listing 19.168, ties student records and address records together.

Listing 19.168: The generated SQL code for creating the `student_address` table, which has a composite primary key composed of the `student` and `address` foreign keys. The corresponding `REFERENCES` constraints have been omitted here for the sake of brevity. ([src](#))

```

1  -- object: public.student_address | type: TABLE --
2  -- DROP TABLE IF EXISTS public.student_address CASCADE;
3  CREATE TABLE public.student_address (
4      student integer NOT NULL,
5      address integer NOT NULL,
6      CONSTRAINT student_address_pk PRIMARY KEY (student,address)
7 );
8  -- ddl-end --
9  ALTER TABLE public.student_address OWNER TO postgres;
10 -- ddl-end --

```

Listing 19.169: Inserting some data into the tables `student`, `address`, and `student_address`. ([src](#))

```

1  /** Insert data into the database. */
2
3  -- Insert several address records.
4  INSERT INTO address (
5      country, province, city, postal_code, street_address) VALUES
6      ('China', 'Anhui', 'Hefei', '230601', 'Jinkaiqu, Hefei University'),
7      ('China', 'Anhui', 'Hefei', '230026', 'USTC'),
8      ('Deutschland', 'Sachsen', 'Chemnitz', '09111', 'Am Rathaus 1'),
9      ('USA', 'NY', 'New York', '10013', 'Canal Street 4, Chinatown'),
10     ('Deutschland', 'Sachsen', 'Chemnitz', '09111', 'TU Chemnitz'),
11     ('PRC', 'Fujian', 'Quanzhou', '362002', 'West Street'),
12     ('P.R.C.', NULL, 'Beijing', '100084', 'Tsinghua University'),
13     ('Spain', 'Andalusia', 'Granada', '18009', 'Alhambra de Granada');
14
15  -- Create the five student records.
16  INSERT INTO student (name) VALUES
17      ('Bibbo'), ('Bebbo'), ('Bibbi'), ('Babbo'), ('Bebbe');
18
19  -- Establish the relationship to the addresses.
20  INSERT INTO student_address (student, address) VALUES
21      (1, 1), (1, 2), (2, 3), (3, 4), (4, 5), (4, 6), (5, 7), (5, 8);

```

By using the composite primary key, we enforce that a student can be connected to each address at most once, and vice versa. This is because `PRIMARY KEY`-constraints imply `UNIQUE`. We omitted the `REFERENCES`-constraints here for brevity.

Let us now play with this newly normalized model. We therefore generate the same data as before. In Listing 19.169, we first create the `address` records. Then we create the `student` records. Then we insert the relationships between the `student` and the `address` records in table `student_address`. Notice that this step is now needed. When the table was not in the 1NF, we could directly link to addresses from the student records. Now we need to go the extra step and insert records into one additional table. This is the const of the 1NF.

Trying to find the students who have at least one address in China now becomes much easier. The corresponding query, shown in Listing 19.170, does no longer require an `UNION` statement. Instead, we already have the student-address relationships in perfect tabular form. With only two `INNER JOIN` constructs we can make the right connection. The `ILIKE`, `ANY`, `ARRAY`, and `DISTINCT ON` statements are used in the same way as before.

By bringing the data into the 1NF we have achieved two things: First, we made queries significantly simpler. Second, we also do no longer need to care about the actual number of addresses a student can have. A student can have one, two, three, four, ten addresses if they want to. Our query will work all the same. Of course, a DB will not just have one such query. Maybe there will be another query for students who live in our beautiful city Hefei (合肥), or for students who do not have any address

Listing 19.170: Finding all the students with addresses in China is now easier, as no `UNION` is required anymore. (stored in file `select.sql`; output in Listing 19.171)

```

1  /** Get a list of students with at least one address in China. */
2
3  SELECT DISTINCT ON (student)
4      name, city || ', ' || street_address AS address FROM student_address
5      INNER JOIN address ON address = address.id
6      INNER JOIN student ON student = student.id
7      WHERE country ILIKE ANY(ARRAY['%china%', '%PRC%', '%P.R.C.%']);

```

Listing 19.171: The stdout of the program `select.sql` given in Listing 19.170.

```

1 $ psql "postgres://postgres:XXX@localhost/fixed" -v ON_ERROR_STOP=1 -ebf
2   ↪ select.sql
3
4   name |           address
5   -----+
6   Bibbo | Hefei, Jinkaiqu, Hefei University
7   Babbo | Quanzhou, West Street
8   Bebbe | Beijing, Tsinghua University
9   (3 rows)

# psql 16.12 succeeded with exit code 0.

```

outside of our Anhui (安徽) province. If a student with three addresses had appeared under our old logical schema, then we would need to change the table `student` and then work our way through all of these queries to modify them. That would have been quite annoying.

That being say, we also have to admit one thing: I did cheat on you a little bit. The original schema, which violated the 1NF, required each student to have *at least one* address. The new schema which does not violate the 1NF does not. I said “*For the sake of simplicity, we will now not enforce that each student must have an address.*” Well, that’s a little bit of cheating. What we *actually* should have implemented is the pattern `student` \rightarrowtail `address`.

When it comes to selecting and querying the data, this changes nothing. All the advantages mentioned above remain true. We gain the ability to deal with arbitrary numbers of addresses per student.

However, *inserting* data becomes more complex in a $Q \rightarrowtail R$ relationship as compared to a $O \rightarrowtail P$ relationship. We would have needed to use a `WITH` statement to simultaneously creating a row in the `student` table *and* use the primary key of that row to link this new student record to a row in the table `address`. It still can be done. We have learned how to do that. But if we really want to enforce that each student must have at least one address, then the simple `NOT NULL` constraint of the column `address_1` in our original schema must be re-created. And it is recreated by another foreign key in the table `student`. For details, please refer back to Section 19.2.2.9.

The interesting aspect of this situation is this: Usually, normalization makes inserting and updating data easier at the cost of query complexity. For example, we normalized the composite address attribute into multiple tables. One annoying aspect here was that we found multiple different spellings of the country *China*. If we wanted, it would be very very easy to change all the countries to *China* that are either *PRC*, *P.R.C.*, or *中国* with an `UPDATE` statement. We could do that because we now have the column `country`. If you want to do that unnormalized `address` column than, well, good luck. We paid for this ease when we want to reassemble the address string, because now we needed to do string concatenation via `||` and `COALESCE` to deal with `NULL` values.

This time, it is the other way around. If we had implemented all constraints of the original model, then our `INSERT` statements would have become more complicated. The queries later become much easier. Either way, our data is certainly “cleaner” and easier to handle in the 1NF.

19.3.1.3 Summary

Normalizing data to obey the 1NF means to implement the relational data model consistently at the logical level. Most often, we will naturally produce data that conforms with the 1NF when we translate a conceptual model to a logical model. After all, we did learn exactly how to do that the right way. However, if our conceptual model does not fit to 1NF, we may end up with convoluted structures of repeating groups or attributes glued together into one that should actually be separate. This can indeed happen. We said so very often that the conceptual schema should be technology-agnostic. So we can hardly complain if such a model schema emerges that does not fit to our relational likings. Therefore, care must be taken during the logical model design. If we really need to create a logical model that deviates from the conceptual one to achieve proper normalization, we may wish to go back and also modify the conceptual model accordingly. Because we do not want to end up with design documents that contradict each other.

19.3.2 Second Normal Form

We already learned about the 1NF. It will not surprise you that this is not the only normal form.

The second normal form (2NF) deals with the relationship between key attributes and non-key attributes [88, 92, 118, 157, 242]. It applies to composite keys only, i.e., keys that consist of multiple columns of a table. Back in Definitions 18.13 and 19.4, we learned that a key can be used to uniquely identify entities. If a key X uniquely identifies an entity, then this means that all the other attributes of the entity basically provide additional information about that key. In a table in the relational model, a key is a unique identifier for each row. In other words, there can be at most one row for each value of X . All the other columns provide additional information about the real-world object represented by that key.

The 2NF is violated when a non-key attribute is a fact about a *proper subset* of any key [242]. A proper subset can only exist of a set with more than one element. Therefore, the 2NF is only relevant if our tables have a key that is composite, i.e., consists of more than one column. A table is in 2NF if it is in the 1NF *and* all columns that are not part of a key provide information about the complete key(s). Another perspective on the 2NF is offered by functional dependencies (FDs).

Definition 19.8: Functional Dependency

A *functional dependency* (FD) is a relationship between two *groups* of attributes X and Y , such that for each instance of X , the value of X determines the value of Y [398]. This can be written as $X \rightarrow Y$.

In other words, if $X \rightarrow Y$, then there are no two records with the same value of X but different values of Y [242]. A given value of the key X must always occur with the same associated value of Y . Also, if X is a key, then all other columns are by definition dependent on X , simply because there cannot be two rows in a table with the same value of X .

The relational schema of relation R be $\Sigma(R)$ and a key be $X \subseteq \Sigma(R)$. Of course, all attributes a in $\Sigma(R)$ depend on the key attributes X , i.e., it always holds that $X \rightarrow a$. Let us approach a definition of the 2NF using FDs. If the 2NF is observed, then for all attributes $a \in \Sigma(R)$ that are not part of the key X (meaning $a \notin X$), there does *not* exist a proper subset $X' \subset X$ such that a functionally depends on X' , i.e. $X' \rightarrow a$. (*Proper* means that $X' \neq X$.) In the 2NF, all attributes depend on the *complete* key X .

However, this definition is not fully correct. As we discussed before and will discuss again later, there can be multiple keys, i.e., multiple sets of attributes, that uniquely identify rows in R . Yes, we do choose one as primary key, but there may be more keys. This is especially true if we choose a surrogate primary key because the other candidate keys are too large. The non-key attributes must depend on all of these keys in their entirety. Under the 2NF, it is not permitted that they depend on any subset of any key. The correct definition for the 2NF there is given as follows [382]:

Definition 19.9: Second normal form (2NF)

A relation R with the primary key $P \subseteq \Sigma(R)$ is in the second normal form (2NF) if and only if it is in the 1NF and for all sets of attributes $X \subseteq \Sigma(R)$ and all attributes $a \in \Sigma(R)$ with $a \notin X$, $a \notin P$ and $X \rightarrow a$, it holds that X is either a key or a super key, but not a proper subset of any key of R .

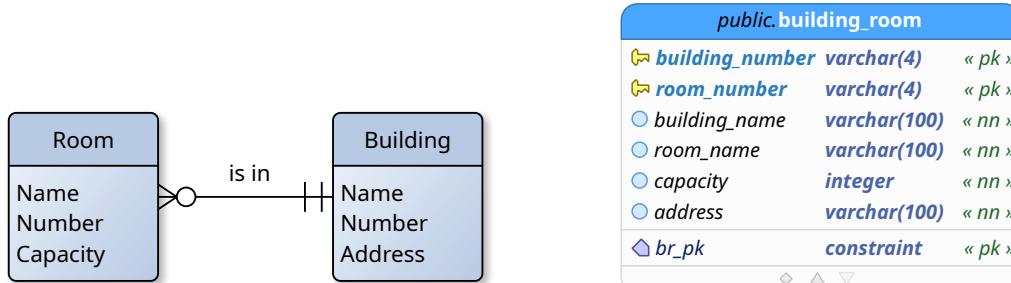
This definition may not be entirely clear at first glance. Let us do an example to understand it better.

19.3.2.1 Example: Room Management System

If a table is normalized to the 2NF, then all attributes depend on the entire primary key. This only matters if the primary key is composite, i.e., if it consists of multiple columns. Let's assume that the primary key is the only key. If a table violates the 2NF, then this means that some columns are functionally dependent only on a part of but not the whole primary key.

Let us explore why this can be a bad thing. Back in Figure 18.17 (The room planning subsystem of the teaching management platform), we presented an ERD for the room planning subsystem of our teaching management platform. We will now create two implementations that try to realize a part of such a system. The first one will violate the 2NF and the second one will not.

Violation: Attributes only Depending on Parts of a Composite Primary Key In Figure 19.32.2, we show only a part of this diagram, namely the two entity types *Room* and *Building*. As you can



(19.32.1) An ERD of a slightly modified subset of room planning subsystem of our teaching management platform that was specified in Figure 18.17.

(19.32.2) A realization of the ERD given in Figure 19.32.1 as a logical model that uses only a single table and that violates the 2NF.

Figure 19.32: An example for a violation of the second normal form (2NF): The table `building_room` has a composite primary key composed of the room number and building number. However, the other attributes each only depend on one part of this primary key.

Listing 19.172: he generated SQL code for creating the table `building_room` that violates the 2NF based on Figure 19.32.2. (src)

```

1 -- object: public.building_room | type: TABLE --
2 -- DROP TABLE IF EXISTS public.building_room CASCADE;
3 CREATE TABLE public.building_room (
4     building_number varchar(4) NOT NULL,
5     room_number varchar(4) NOT NULL,
6     building_name varchar(100) NOT NULL,
7     room_name varchar(100) NOT NULL,
8     capacity integer NOT NULL,
9     address varchar(100) NOT NULL,
10    CONSTRAINT br_pk PRIMARY KEY (building_number,room_number)
11 );
12 -- ddl-end --
13 ALTER TABLE public.building_room OWNER TO postgres;
14 -- ddl-end --

```

Listing 19.173: Inserting some data into the table `building_room` in violation of the 2NF. (src)

```

1  /** Insert data into the database. */
2
3  -- Insert several room records.
4  INSERT INTO building_room (building_number, room_number, building_name,
5                                room_name, capacity, address) VALUES
6      ('36', '305', 'CS Teaching Building', 'Meeting Room', 40,
7       'South Campus II'),
8      ('36', '105', 'CS Teaching Building', 'Lecture Room 1', 80,
9       'South Campus II'),
10     ('10', '100', 'Language Teaching Building', 'Teaching Room A', 30,
11      'South Campus 1'),
12     ('10', '102', 'Language Teaching Building', 'Teaching Room B', 30,
13      'South Campus 1'),
14     ('53', '904a', 'Comprehensive Experimental Building',
15      'Office 1', 10, 'South Campus 2'),
16     ('53', '904b', 'Comprehensive Experimental Building',
17      'Cluster Room', 3, 'South Campus 2'),
18     ('7', '200', 'Main Teaching Building', 'Auditorium', 120,
19      'South Campus 1'),
20     ('36', '106', 'CS Teaching Building', 'Lecture Room 2', 80,
21      'South Campus 2');

```

see, the two entity types are related based on the Room \rightarrow Building pattern. We showed how this pattern can be implemented in [Section 19.2.2.6](#) ($K \perp\!\!\!\perp O \Leftarrow L$).

Assume that a DB designer decided for a simpler implementation and chose to just put all the room and building data into a single table. The result is illustrated in [Figure 19.32](#). The table `building_room` was designed. It holds the columns `building_number`, `room_number`, `building_name`, `room_name`, `capacity`, and `address`. The `building_number` and `room_number` are both of type `VARCHAR` with a maximum length of 4. They do not necessarily need to be numbers but could also be something like '105b'. The type `VARCHAR` is for variable-length text strings. Together, the two columns form the primary key: Each room in our university is uniquely identified by the building and room number. There cannot be two different rooms that have the same building and room number.

This design violates the [2NF](#) because we have some columns that depend only on a part of the primary key. The columns `building_name` and `address` are facts about `building_number` alone. They do not functionally depend on `room_number`.

In [Listing 19.172](#), we illustrate the `SQL` code that generates the table. It creates the columns for the room and building names as well as the address as `VARCHAR` with a maximum length of 100. The capacity for students of a room is an `INTEGER` number. `building_number` and `room_number` together form the primary key.

In [Listing 19.173](#), we fill the table with some data. For example, we store the meeting room (room 305) of Building 36 (the Computer Science Teaching Building), located in South Campus 2. We also store information about a lecture room in the same building. We store information about two teaching rooms in the Language Teaching Building in South Campus 1, as well as some offices in Building 53 of South Campus 2, and some other rooms.

One thing immediately becomes clear when reading these `INSERT` statements: This structure creates a lot of redundancy. For example, the address of Building 36 is stored three times. Actually, the name of Building 36 is stored three times as well. This feels wrong.

Besides the redundancy, this structure has another problem. It becomes apparent when we try to change the data. With the first query in [Listing 19.174](#), we want to extract a list of all buildings from our DB. We will `SELECT` the building number and building name from our table `building_rooms`. Since these occur several times, we write `SELECT DISTINCT` instead of just `SELECT`. This only returns the unique rows. The output shows us that we have four buildings in our DB, as expected.

Assume that some time has passed and we decided that Rooms 904a and 904b are no longer used for small-group teaching. So with the second query in the listing, we delete them from our DB. We do this by using the `DELETE FROM` statement, with the `WHERE` criterion requiring that the building number

Listing 19.174: First, we select a list of all buildings from the DB, which yields 4 buildings. Then we delete the two offices 904a and 904b of Building 53 from our DB. This causes a deletion anomaly ([Definition 19.10](#)): all data about Building 53 to disappear. So when we select the list of all buildings again, now there are only 3. (stored in file `delete.sql`; output in [Listing 19.175](#))

```

1  /** Delete room 904a and 904b, which doesn't delete the building 53. */
2
3  -- Get the list of all buildings.
4  SELECT DISTINCT building_number, building_name FROM building_room;
5
6  -- Delete rooms 904a and 904b in building 53.
7  DELETE FROM building_room WHERE building_number = '53'
8      AND room_number LIKE '904%'
9      RETURNING building_number, room_number;
10
11 -- Get the list of all buildings again: Building 53 is still there.
12 SELECT DISTINCT building_number, building_name FROM building_room;
```

Listing 19.175: The stdout of the program `delete.sql` given in [Listing 19.174](#).

```

1  $ psql "postgres://postgres:XXX@localhost/violation" -v ON_ERROR_STOP=1 -
2      ↪ ebf delete.sql
3
4  building_number |          building_name
5  -----+-----
6  36           | CS Teaching Building
7  53           | Comprehensive Experimental Building
8  10           | Language Teaching Building
9  7            | Main Teaching Building
10 (4 rows)
11
12 building_number | room_number
13 -----+-----
14 53           | 904a
15 53           | 904b
16 (2 rows)
17
18 DELETE 2
19 building_number |          building_name
20 -----+-----
21 36           | CS Teaching Building
22 10           | Language Teaching Building
23 7            | Main Teaching Building
24 (3 rows)
25
26 # psql 16.12 succeeded with exit code 0.
```

must be 53 and the room number must be `LIKE '904%'`, meaning that it must start with 904 followed by an arbitrary character. PostgreSQL allows us to also add the `RETURNING` statement [360], that acts like a `SELECT` on the deleted rows. The output of the command therefore is the building number and room number of the deleted rooms. We see that, indeed, rooms 904a and 904b of Building 53 are deleted.

We now run the same query giving us the list of buildings again (as the third query in [Listing 19.174](#)). We notice that, by deleting the two rooms in Building 53, we inadvertently also deleted all information about the building. Its building number, building name, and building address are all lost. Because they were stored together with the room records. This is called *deletion anomaly*.

Definition 19.10: Deletion Anomaly

A situation where the deletion of unwanted information causes desired information to be deleted as well is called *deletion anomaly* [398].

Well, of course, you may argue that in some cases, that would actually be the desired effect. For example, in the original model given in [Figure 18.17](#) (The room planning subsystem of the teaching management platform), the relationship between buildings and rooms was specified as Room $\rightarrow\leftarrow$ Building. In this pattern, discussed in [Section 19.2.2.7](#) ($M \rightleftarrows N$), it would be required to delete the building data if all rooms in the building were removed. The problem is that this deletion also occurs if we try to follow the $K \rightarrow\leftarrow L$, where such deletion is not warranted.

[Listing 19.176](#) illustrates another issue that is encouraged (but not caused) by a violation of the 2NF: inconsistency. Assume that we wanted to get a list of all rooms available in South Campus 2. We could fire out the first query given in [Listing 19.176](#). We could simply `SELECT` the building number and room number where `address = 'South Campus 2'`. This query only returns a single room, namely Room 106 in Building 36. We know that this is wrong, we definitely entered more than one room for Building 36. Scrolling back to our insertion script in [Listing 19.173](#), we noticed that we sometimes wrote `'South Campus II'` instead of `'South Campus 2'`. Obviously, for a computer, these are two different things.

The fact that we need to store the same data again and again makes such mistakes more likely. Whenever a person enters some information into the computer, there is a certain probability that they make an error. The more often we enter data, the higher the chance that some error is made somewhere. Violating the 2NF forces us to write the same information more often. Hence it increases the chance of making an error. The violating the 2NF does not *cause* the error, but it makes it more likely.

Anyway, we can solve the above problem in two ways: Either we select all rows where `address = 'South Campus 2'` or `address = 'South Campus II'`, as done in the second query. Or we just fix the incorrectly entered data with an `UPDATE` command. We would set the building address to `'South Campus 2'` if it currently is `'South Campus II'`. We again enrich this query with a `RETURNING` statement [360] which will print out all the rows that were modified with this statement. After this update, the original query works as well.

Finally, let us try to change the name of Building 36 from “CS Teaching Building” to “Computer Science Building.” This can be done with the query shown in Listing 19.178. We again use an `UPDATE` statement applied to our table `building_room`. This time, we want to modify the rows where `address = 'South Campus 2'` and `building_number = '36'`. For all of these rows, we set `building_name = 'Computer Science Building'`. We again use the `RETURNING` statement [360] to print out all the rows that were modified with this command. As you can see, this changes three rows. Notice that we wanted to change only *one* single piece of data. But we did (or better: needed to) change *three* rows.

Definition 19.11: Update Anomaly

A situation where changing one piece of information requires that multiple rows must be updated is called *update anomaly*.

This anomaly is directly caused by the violation of the 2NF. And another anomaly results from it as well:

Definition 19.12: Insertion Anomaly

A situation where inserting data into the DB is not possible because other data is not already there is called *insertion anomaly* [398].

For example, it is simply not possible to insert information about a room without inserting information about a building as well (and vice versa). Of course, in our special case here, this would not make any sense anyway. But there can be many other cases where the violation of the 2NF causes this as a problem.

In [242], for example, another interesting case is presented from the field of warehousing. The names and addresses of warehouses are stored together with the names and quantities of the products in them *in one table*. This design of that single table violates the 2NF. The data was stored in exactly these four columns (name of warehouse, address of warehouse, name of product, quantity of product). Thus, it is not possible to create a record for a warehouse if no product is stored in it as well.

Listing 19.176: We want to see all rooms in South Campus 2. Due to some inconsistent spelling (South Campus II vs. South Campus 2), the first query misses some rooms. We then run a modified query, which gives us all the rooms. We can also fix the table by updating the corresponding rows, after which the first query also works correctly. (stored in file `update_a.sql`; output in Listing 19.177)

```

1  /** Find all the rooms available in South Campus 2. */
2
3  -- Several rooms are missing, because the data is inconsistent:
4  -- Their address was 'South Campus II'.
5  SELECT building_number, room_number FROM building_room
6    WHERE address = 'South Campus 2';
7
8  -- Now we get these rooms.
9  SELECT building_number, room_number FROM building_room
10   WHERE address = 'South Campus 2' OR address = 'South Campus II';
11
12 -- We can fix this problem with an UPDATE instruction.
13 UPDATE building_room SET address = 'South Campus 2'
14   WHERE address = 'South Campus II' RETURNING building_number;
15
16 -- This query now also works, because the addresses are now consistent.
17 SELECT building_number, room_number FROM building_room
18   WHERE address = 'South Campus 2';

```

Listing 19.177: The stdout of the program `update_a.sql` given in Listing 19.176.

```

1  $ psql "postgres://postgres:XXX@localhost/violation" -v ON_ERROR_STOP=1 -
2      ↪ ebf update_a.sql
3  building_number | room_number
4  -----+-----
5  36      | 106
6  (1 row)
7
8  building_number | room_number
9  -----+-----
10 36      | 305
11 36      | 105
12 36      | 106
13 (3 rows)
14
15 building_number
16 -----
17 36
18 36
19 (2 rows)
20
21 UPDATE 2
22 building_number | room_number
23 -----+-----
24 36      | 106
25 36      | 305
26 36      | 105
27 (3 rows)
28 # psql 16.12 succeeded with exit code 0.

```

Listing 19.178: We want to change the name of Building 36 to “Computer Science Building.” While we only want to change one single piece of information, we actually have to update three rows. This is an update anomaly ([Definition 19.11](#)) caused by the violation of the 2NF of our design. (stored in file `update_b.sql`; output in [Listing 19.179](#))

```

1  /** Change the name of Building 36 to Computer Science Building. */
2
3 UPDATE building_room SET building_name = 'Computer Science Building'
4   WHERE address = 'South Campus 2' AND building_number = '36'
5   RETURNING building_number, building_name, room_number;

```

Listing 19.179: The stdout of the program `update_b.sql` given in [Listing 19.178](#).

```

1 $ psql "postgres://postgres:XXX@localhost/violation" -v ON_ERROR_STOP=1 -
2   ↪ ebf update_b.sql
3      building_number |      building_name | room_number
4      +-----+-----+
5      36 | Computer Science Building | 105
6      36 | Computer Science Building | 106
7      36 | Computer Science Building | 305
8  (3 rows)
9
10 UPDATE 3
10 # psql 16.12 succeeded with exit code 0.

```

Fixed: Extracted Columns that Depend on Partial Key into Own Table The violation of the 2NF can – either directly or indirectly – cause several problems or anomalies. So let us normalize the DB from the previous section into the 2NF.

In our original design, we had a single table `building_room`. This table had a composite primary key consisting of the columns `building_number` and `room_number`. However, the columns `building_name` and `address` only depend on `building_number`. This led to several anomalies, to redundancy, and increased the probability to make errors.

To achieve normalization into the 2NF, we have to separate the columns `building_name` and `address` into an own table. We will call this table `building`. Of course, it also needs a primary key, which will be `building_number`. We can use `building_number`, because, in our university, the rule is that there can never be two buildings with the same number.

We rename the rest of the table `building_room` to `room`. Each row now, indeed, just represents a single room in a building. This table still has the composite primary key consisting of the columns `building_number` and `room_number`. There is no reason to change this. Matter of fact, keeping this composite key instead of using a surrogate key directly enforces that rooms can only be stored after the corresponding building record has been created.

There also still are the two fields `room_name` and `capacity` in this table. Clearly, both non-key fields

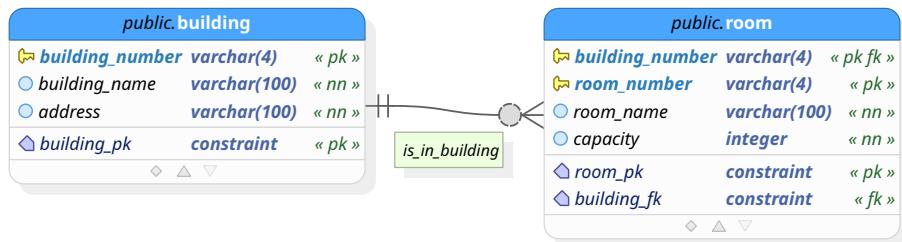


Figure 19.33: A different approach to Figure 19.32.2 that no longer violates the 2NF. The columns that only depend on the `building_number` have been extracted into their own table.

Listing 19.180: The generated SQL code for creating the table `building`. (src)

```

1  -- object: public.building | type: TABLE --
2  -- DROP TABLE IF EXISTS public.building CASCADE;
3  CREATE TABLE public.building (
4      building_number varchar(4) NOT NULL,
5      building_name varchar(100) NOT NULL,
6      address varchar(100) NOT NULL,
7      CONSTRAINT building_pk PRIMARY KEY (building_number)
8  );
9  -- ddl-end --
10 ALTER TABLE public.building OWNER TO postgres;
11 -- ddl-end --

```

Listing 19.181: The generated SQL code for creating the table `room`. (src)

```

1  -- object: public.room | type: TABLE --
2  -- DROP TABLE IF EXISTS public.room CASCADE;
3  CREATE TABLE public.room (
4      building_number varchar(4) NOT NULL,
5      room_number varchar(4) NOT NULL,
6      room_name varchar(100) NOT NULL,
7      capacity integer NOT NULL,
8      CONSTRAINT room_pk PRIMARY KEY (building_number, room_number)
9  );
10 -- ddl-end --
11 ALTER TABLE public.room OWNER TO postgres;
12 -- ddl-end --

```

Listing 19.182: The generated SQL code for creating the foreign key constraint linking the rows in table `room` to the rows in table `building`. (src)

```

1 -- object: building_fk | type: CONSTRAINT --
2 -- ALTER TABLE public.room DROP CONSTRAINT IF EXISTS building_fk CASCADE;
3 ALTER TABLE public.room ADD CONSTRAINT building_fk FOREIGN KEY (
    ↪ building_number)
4 REFERENCES public.building (building_number) MATCH SIMPLE
5 ON DELETE NO ACTION ON UPDATE NO ACTION;
6 -- ddl-end --

```

Listing 19.183: Inserting the very same data as in [Listing 19.173](#) into the tables `building` and `room`. (src)

```

1 /** Insert data into the database. */
2
3 -- Insert several room records.
4 INSERT INTO building (building_number, building_name, address) VALUES
5     ('36', 'CS Teaching Building', 'South Campus II'),
6     ('10', 'Language Teaching Building', 'South Campus 1'),
7     ('53', 'Comprehensive Experimental Building', 'South Campus 2'),
8     ('7', 'Main Teaching Building', 'South Campus 1');
9
10 -- Insert several room records.
11 INSERT INTO room (building_number, room_number, room_name,
12                     capacity) VALUES
13     ('36', '305', 'Meeting Room', 40),
14     ('36', '105', 'Lecture Room 1', 80),
15     ('10', '100', 'Teaching Room A', 30),
16     ('10', '102', 'Teaching Room B', 30),
17     ('53', '904a', 'Office 1', 10),
18     ('53', '904b', 'Cluster Room', 3),
19     ('7', '200', 'Auditorium', 120),
20     ('36', '106', 'Lecture Room 2', 80);

```

are functionally dependent on the primary key. They depend on the *complete* primary key, to be exact. As a bonus additionally to satisfying the 2NF, this logical design, illustrated in [Figure 19.33](#), represents our original conceptual model given in [Figure 19.32.1](#) much better.

We now implement this design using SQL. [Listing 19.180](#) creates the table `building`. It has the columns `building_number`, `building_name`, and `address`. `building_number` is the primary key. We could have added a `UNIQUE` constraint for `building_name`, thus enforcing that no two buildings can have the same name ... but we leave this as an exercise for the interesting reader.

[Listing 19.181](#) creates the table `room`. Its first two columns, `building_number` and `room_number`, form the composite primary key. The other two columns (`room_name` and `capacity`) functionally depend on this key in its entirety. [Listing 19.182](#) establishes the foreign key constraint linking each row of table `room` to one row of table `building`. This constraint turns `building_number` into a foreign key.

We can now insert the exactly as same data as in the last section into this DB. [Listing 19.183](#) therefore now needs two `INSERT INTO` statements. First we store all the building data into the table `building`. Then we store the information about the rooms into table `room`. We immediately notice that now, there is much less redundancy. The data about each building is stored exactly once. Before, we needed to store it for each room. Since we need to write data much less often, it becomes much less likely to make errors. And if we make errors, they are easier to spot, since there are fewer records that we would need to check.

In [Listing 19.184](#), we now reproduce the example from [Listing 19.174](#). Back then, we first got a list of all buildings and found that there were four of them. We then deleted the two rooms 904a and 904b. This led to Building 53 to disappear as well, as its existence only emerged from the two room records.

So we again start by getting a list of all buildings. This can be done using the query `SELECT building_number, building_na`

Listing 19.184: First, we select a list of all buildings from the DB, which yields 4 buildings. (Notice that we do no longer need the `DISTINCT` keyword, since each building is listed only once in table `building`.) Then we delete the two offices 904a and 904b of Building 53 from our DB. This no longer causes a deletion anomaly, as the data of Building 53 is not deleted. So when we select the list of all buildings again, it is still there. This is different from the situation in [Listing 19.175](#), where it disappeared. (stored in file `delete.sql`; output in [Listing 19.185](#))

```

1  /** Delete room 904a and 904b, which deletes the entire building. */
2
3  -- Get the list of all buildings.
4  SELECT building_number, building_name FROM building;
5
6  -- Delete rooms 904a and 904b in building 53, which deletes building 53.
7  DELETE FROM room WHERE building_number = '53'
8    AND room_number LIKE '904%'
9    RETURNING building_number, room_number;
10
11 -- Get the list of all buildings again: Building 53 disappeared.
12 SELECT building_number, building_name FROM building;
```

Listing 19.185: The stdout of the program `delete.sql` given in [Listing 19.184](#).

```

1 $ psql "postgres://postgres:XXX@localhost/fixed" -v ON_ERROR_STOP=1 -e bf
2   ↪ delete.sql
3
4   building_number |      building_name
5   -----+-----+
6     36          | CS Teaching Building
7     10          | Language Teaching Building
8     53          | Comprehensive Experimental Building
9     7           | Main Teaching Building
10  (4 rows)
11
12  building_number | room_number
13  -----+-----+
14  53            | 904a
15  53            | 904b
16  (2 rows)
17
18  DELETE 2
19  building_number |      building_name
20  -----+-----+
21  36          | CS Teaching Building
22  10          | Language Teaching Building
23  53          | Comprehensive Experimental Building
24  7           | Main Teaching Building
25  (4 rows)
26
27 # psql 16.12 succeeded with exit code 0.
```

Notice that, compared to [Listing 19.174](#), we do no longer need to write the `DISTINCT` keyword. Back then, in the table `building_room`, each building number and name could appear several times, as they were stored for each room. However, in our new table `building`, they can only appear once. Hence, the omission of duplicate rows, performed by `DISTINCT`, is now no longer necessary. Anyway, our query gives us a list of four buildings.

We then again delete the two offices 904a and 904b of Building 53 using a `DELETE` query. The `RETURNING` statement in the query again shows us the two deleted offices. Back in the previous section, deleting these rows also deleted all information about Building 53. This [Definition 19.10](#) was caused by our violation of the `2NF`. Back then, issuing the `SELECT` query again would yield only three buildings. However, as you can see in [Listing 19.185](#), this does not happen now. The building records are independently stored in table `building` and not affected by the deletion of rows in table `room`. All four

building records remain.

In Listing 19.186, we reproduce the example from Listing 19.176: We want to get a list of all rooms available in South Campus 2. When comparing the queries necessary to achieve this between the two listings, we notice the *drawback* of the 2NF. We now have a more complex query.

Before, the building address was stored together with the room number in a single row. This is no longer the case. The room information is stored in table `room`. However, the building address is stored in table `building`. If we want to find out whether a room is in South Campus 2, we must combine the data from these two tables. An `INNER JOIN` does the trick. The primary key of table `building` is `building_number`. The `building_number` is a foreign key of table `room` (and also part of that table's primary key). So the join condition `room.building_number = building.building_number` allows us to find the right row in table `building` for each row in table `room`.

So we execute this slightly more complex query. We are baffled to notice that this yields no result at all. There are no rooms in South Campus 2? OK, we deleted the two rooms in Building 53, which were in South Campus 2. So they are gone. But there still should be Building 36 with its three rooms.

Upon closer inspection of the data we inserted in Listing 19.183, we realize that we made the mistake to write the address of Building 36 as *South Campus II* instead of *South Campus 2*. Since we only store the building record once, this affected all rooms belonging to it.

This problem can be fixed exactly in the same ways as back in Listing 19.176. We can either just expand our selection condition to also include South Campus II. This works, but it also is a bit unsatisfying. Should we really leave inconsistent data in our DB?

No. Of course not. We decide to fix this by applying `SET address = 'South Campus 2'` to all rows of table `building` where `address = 'South Campus II'`. We also return the building number of the buildings that are affected by this update. As the result, we see that a single row in the table `building` is changed. Back in Listing 19.176, two rows needed to be changed. That was an example of the update anomaly. And that anomaly is gone now. After the update, the first query works as expected.

As a side note: The fact that this problem still persists hints that we should probably have a table `building_address` where we put the building addresses. This would avoid this issue.

An even clearer example of the update anomaly that occurs when the 2NF is violated was given in Listing 19.178. When we wanted to change the name of Building 36 from *CS Teaching Building* to *Computer Science Building*, we needed to update three rows. We wanted to change one piece of information, but three changes were actually required. Now that we observe the 2NF, doing the same thing in Listing 19.188 only affects a single row in table `building`. The anomaly has disappeared.

Listing 19.186: We want to see all rooms in South Campus 2. Compared to [Listing 19.176](#), the query is more complex as we now need an [INNER JOIN](#). Due to the inconsistent spelling (South Campus II vs. South Campus 2), the first query finds nothing. We then run a modified query, which gives us all the rooms. We can also fix the table [building](#) by updating the corresponding rows, after which the first query also works correctly. Notice that here, only one row is updated. In [Listing 19.177](#), two rows were affected. (stored in file [update_a.sql](#); output in [Listing 19.187](#))

```

1  /** Find all the rooms available in South Campus 2. */
2
3  -- Several rooms are missing, because the data is inconsistent:
4  -- Their address was 'South Campus II'.
5  SELECT room.building_number, room_number FROM room
6    INNER JOIN building ON
7      room.building_number = building.building_number
8    WHERE address = 'South Campus 2';
9
10 -- Now we get these rooms.
11 SELECT room.building_number, room_number FROM room
12   INNER JOIN building ON
13     room.building_number = building.building_number
14   WHERE address = 'South Campus 2' OR address = 'South Campus II';
15
16 -- We can fix this problem with an UPDATE instruction.
17 UPDATE building SET address = 'South Campus 2'
18   WHERE address = 'South Campus II' RETURNING building_number;
19
20 -- This query now also works, because the addresses are now consistent.
21 SELECT room.building_number, room_number FROM room
22   INNER JOIN building ON
23     room.building_number = building.building_number
24   WHERE address = 'South Campus 2';

```

Listing 19.187: The stdout of the program [update_a.sql](#) given in [Listing 19.186](#).

```

1 $ psql "postgres://postgres:XXX@localhost/fixed" -v ON_ERROR_STOP=1 -e bf
2   ↪ update_a.sql
3   building_number | room_number
4   -----+-----
5   (0 rows)
6
6   building_number | room_number
7   -----+-----
8   36           | 305
9   36           | 105
10  36           | 106
11 (3 rows)
12
13  building_number
14  -----
15  36
16 (1 row)
17
18  UPDATE 1
19  building_number | room_number
20  -----+-----
21  36           | 105
22  36           | 106
23  36           | 305
24 (3 rows)
25
26 # psql 16.12 succeeded with exit code 0.

```

Listing 19.188: We want to change the name of Building 36 to “Computer Science Building.” Different from the situation in [Listing 19.179](#), only a single row needs to be changed. (stored in file `update_b.sql`; output in [Listing 19.189](#))

```
1  /** Change the name of Building 36 to Computer Science Building. */
2
3  UPDATE building SET building_name = 'Computer Science Building'
4      WHERE address = 'South Campus 2' AND building_number = '36'
5      RETURNING building_number, building_name;
```

Listing 19.189: The stdout of the program `update_b.sql` given in [Listing 19.188](#).

```
1 $ psql "postgres://postgres:XXX@localhost/fixed" -v ON_ERROR_STOP=1 -e bf
2   ↪ update_b.sql
3   building_number |      building_name
4   -----+-----+
5   36          | Computer Science Building
6   (1 row)
7
8 UPDATE 1
9 # psql 16.12 succeeded with exit code 0.
```

19.3.2.2 Summary

The **2NF** is a rule that helps us to avoid redundancy. It prescribes that there should not be a functional dependency from non-key attributes on a part of any composite key. As the result, it reduces the chance of errors during data entry, because the volume of data entered becomes smaller.

The 2NF also mitigates problems that can be caused by deletion and update anomalies. It comes at the cost of slightly more complicated queries: To implement the 2NF often means to divide the data into more tables. Hence, when we need all of the information together, we need to merge it from these separate tables again using, e.g., **INNER JOIN** queries.

At the beginning of this section, we were talking about *keys*, but not just *primary keys*. We even gave the rather complicated **Definition 19.9** (Second normal form (2NF)), involving keys, the primary key, and super keys. This looked all quite complicated for a rather simple statement, namely that no attribute can depend on a part of a key only.

The reason for this complicated formulation is that a relation can have multiple different keys. Let's say we want to store information about people in table. Then a reasonable key of such a table would be the government-issued ID number. The name, place of birth, and **DOB** could be another key (although this would be a slightly more dangerous choice, because two people of the same name could be born in the same place on the same day). Yet another key could be the primary mobile phone number. Maybe the best choice in a practical **DB** implementation would be to use a surrogate key, i.e., an identifier that is unique and automatically generated by the **DBMS**. This last possible choice is interesting and relevant here.

We presented a design of the room planning subsystem of our imaginary teaching management platform that violated the 2NF. We put all the data of rooms and buildings into a single table. The primary key was composed of the building number and room number. The columns for building name and building address only depend on the building number, but not on the room number. This was the violation of the 2NF. We normalized the design to comply with the 2NF by separating these columns into a new table.

If the 2NF would only consider *primary keys*, then another "solution" to this problem would be to simply use a surrogate key for our original table **building_room**. Since the surrogate key would be a single column, the 2NF would then no longer be violated. If you refer back to our example and think about this method, you quickly realize: This would have solved none of the problems of the design. The deletion and update anomalies would have remained.

So having a definition for the 2NF that can easily be implemented making a useless change to the design that leads to no improvement at all would be useless. **NFs** are defined exactly with the goal to give us guidance towards good DB design. Observing them should yield a good structure and reduce the chance of anomalies.

For this purpose, the 2NF concerns *all* key attributes, not just the *primary key* attributes. Therefore, using a surrogate key would *not* lead to a normalization under the 2NF. The building number/room number attribute combination would still be key for the table. And the building name and building address would still only depend on a part of that key.

19.3.3 Third Normal Form

The **third normal form (3NF)** deals with the relationship between non-key attributes [88, 92, 118, 157, 242]. The **3NF** is violated when a non-key attribute is a fact about another non-key attribute [242]. A relation R is in the 3NF if no non-key attribute transitively depends on a key attribute.

Definition 19.13: Transitive Functional Dependency

Let A , B , and C be three distinct attributes (or distinct sets of attributes) of the relation R , i.e., $A \subseteq \Sigma(R)$, $B \subseteq \Sigma(R)$, and $C \subseteq \Sigma(R)$. The functional dependency $A \rightarrow C$ is a *transitive dependency*, if and only if $A \rightarrow B$ and $B \rightarrow C$ are true while $B \rightarrow A$ is *not* true.

Formally, the 3NF can be stated as follows [382]:

Definition 19.14: Third normal form (3NF)

A relation R is in 3NF if it is in 2NF and it is true for each attribute $a \in \Sigma(R)$ that is transitively dependent on a key $X \subseteq \Sigma(R)$ – i.e., for which it holds that $X \rightarrow Y \rightarrow a$ with $Y \subseteq \Sigma(R)$ – that either Y contains a key, a is part of the primary key, or $a \in X$.

A table is in 3NF if it is in the 2NF and all the attributes that are not part of any candidate key depend directly on the primary key. Every non-prime attribute is non-transitively dependent on every candidate key in the table.

19.3.3.1 Example: Student and Parent Information

Since, at first glance, the definition is again somewhat unclear, we explore it with an example. Imagine that we are working on the design of the teaching management platform. A colleague mentions that it would be a good idea to also have the contact information of one parent stored for each student. Sometimes there can be situations where such information could be useful. Maybe a student has an accident on campus. Maybe a student suddenly stops coming to lectures and is nowhere to be found. Then it would be good to be able to call a parent.

Assume that the following information about students should be stored in our DB:

- their university-assigned Student ID `student_id`,
- their name `student_name`,
- the name `parent_name` of one of their parents, and
- the mobile phone number `parent_mobile` of that parent.

The following FDs exist at first glance:

- $\text{student_id} \rightarrow \text{student_name}$,
- $\text{student_id} \rightarrow \text{parent_name}$,
- $\text{student_id} \rightarrow \text{parent_mobile}$, and
- $\text{parent_mobile} \rightarrow \text{parent_name}$.

Violation: Non-Key Attribute Transitively Depends on Primary Key In a first attempt, we again put all the data in one table. In Figure 19.34, we show a part of logical model which illustrates a DB table for storing information about students and their parents. The table has four columns. In the first column, we store the student ID, which is the primary key. The second column is the student name. It is not a key and not necessarily unique. For each student, we also store the name of one parent and their mobile phone number as a contact for emergencies.

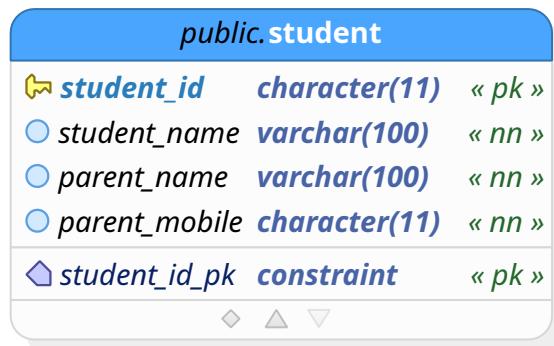


Figure 19.34: A table which stores data about students and their parents in violation of the 3NF, because the parent name depends on the parent phone number, which transitively depends on the student ID.

Listing 19.190: The generated SQL code for creating the table `student` that violates the 3NF based on [Figure 19.34.](#) (src)

```

1  -- object: public.student | type: TABLE --
2  -- DROP TABLE IF EXISTS public.student CASCADE;
3  CREATE TABLE public.student (
4      student_id character(11) NOT NULL,
5      student_name varchar(100) NOT NULL,
6      parent_name varchar(100) NOT NULL,
7      parent_mobile character(11) NOT NULL,
8      CONSTRAINT student_id_pk PRIMARY KEY (student_id)
9  );
10 -- ddl-end --
11 ALTER TABLE public.student OWNER TO postgres;
12 -- ddl-end --

```

Listing 19.191: Inserting some data into the table `student` in violation of the 3NF. (src)

```

1  /** Insert data into the database. */
2
3  -- Insert several student + parent records.
4  INSERT INTO student (student_id, student_name, parent_name,
5                      parent_mobile) VALUES
6  ('1234567890', 'Bibbo', 'Boddo', '55544466677'),
7  ('1234567891', 'Bebbo', 'Balla', '77788811122'),
8  ('1234567892', 'Bibboto', 'Boddo', '55544466677'),
9  ('1234567894', 'Bibboba', 'Boddo', '55544466677');

```

This table is in the [1NF](#), as there are neither compound attributes nor repeated groups. It is also in the [2NF](#), because there is no compound key and, hence, it is not possible that an attribute could depend on a part of such a key only.

However, this table violates the [3NF](#). The attribute `parent_name` is functionally dependent on the attribute `parent_mobile`. We write this as $\text{parent_mobile} \rightarrow \text{parent_name}$. A mobile phone number is associated with a single person, hence there can only be one name for a given mobile phone number.

The parent mobile phone number depends on the primary key `student_id`. This can be written as $\text{student_id} \rightarrow \text{parent_mobile}$. The chain $\text{student_id} \rightarrow \text{parent_mobile} \rightarrow \text{parent_name}$ exists.

It obviously does not hold that $\text{parent_mobile} \rightarrow \text{student_id}$. A parent could have multiple children who now study at our university. Therefore, [Definition 19.13](#) is fulfilled. The relationship $\text{student_id} \rightarrow \text{parent_name}$ is transitive. And since it occurs in the table `student`, it violates the [3NF](#).

Let us explore what consequences this has. We first create the table by executing the script given in [Listing 19.190](#). The primary key `student_id` be a string of exactly length 11 (`CHARACTER(11)`), because this is the length of our student IDs. The student name `student_name` is a variable-length string of maximal length 100 (`VARCHAR(100)`). The same holds for the name `parent_name` of the parent. The mobile phone number of the parent, `parent_mobile`, is stored again stored as fixed-length string with length 11. None of the four columns is optional, hence they are all marked as `NOT NULL`.

Then we insert four student records into the DB using the script given in [Listing 19.191](#). There are four students, Mr. Bibbo, Mr. Bebbo, Mr. Bibboto, and Ms. Bibboba. Their IDs are not important. What is important is that Mr. Bibbo, Mr. Bibboto, and Ms. Bibboba happen to be siblings. Their proud father is Mr. Böddö.

Now it happened that, being located in China, the teacher in the administrative office did not really know how to enter the letter ö. So they chose, for the time being, to call the dad of the three kids simply "Mr. Boddo". A few days later, they found a solution on how to enter the letter ö. They asked a German colleague to send them the letter via [WeChat](#) and copy-pasted it into the system. In order to fix the data, they created the script [Listing 19.192](#).

They use the mobile phone number 555 444 666 77 of Mr. Böddö to identify him in the table.

Listing 19.192: We noticed that the name of the father of Mr. Bibbo, Mr. Bibotto, and Ms. Bibboba is actually Mr. Böddö, not Mr. Boddo. To change it, we need to touch three records, which is a typical update anomaly. (stored in file `update.sql`; output in [Listing 19.193](#))

```

1  /** Find the parent Boddo and change his name to Böddö. */
2
3  -- Get the names of all students whose parent has mobile 55544466677.
4  SELECT student_name, parent_name FROM student
5    WHERE parent_mobile = '55544466677';
6
7  -- Change the name of the parent with mobile 55544466677 to Böddö.
8  UPDATE student SET parent_name = 'Böddö'
9    WHERE parent_mobile = '55544466677'
10   RETURNING student_name, parent_name;

```

Listing 19.193: The stdout of the program `update.sql` given in [Listing 19.192](#).

```

1 $ psql "postgres://postgres:XXX@localhost/violation" -v ON_ERROR_STOP=1 -
2   ↪ ebf update.sql
3   student_name | parent_name
4   -----+-----
5   Bibbo       | Boddo
6   Bibboto     | Boddo
7   Bibboba     | Boddo
8   (3 rows)
9
10  student_name | parent_name
11  -----+-----
12  Bibbo       | Böddö
13  Bibboto     | Böddö
14  Bibboba     | Böddö
15  (3 rows)
16 UPDATE 3
17 # psql 16.12 succeeded with exit code 0.

```

To test this, they first use a `SELECT` statement to see whether they really get the corresponding rows and only the corresponding rows. This works perfectly well, three student/parent name pairs associated with this mobile phone number are obtained. The output of this command is as expected, so next they launch an `UPDATE` command. The `parent_name` is `SET` to `Böddö` for each record where `parent_mobile = '55544466677'`. The updated records are returned via the `RETURNING` statement. As we can see, the correct three rows are affected.

So to change one piece of information, three rows needed to be touched. This is a classical example of an update anomaly. Similarly, if we would delete the records of Mr. Bibbo, Mr. Bibotto, and Ms. Bibboba, all the data about their dad would disappear as well. Even worse, if we would want to delete one parent from the dataset, e.g., Mr. Böddö, then the data of all their student children would be removed as well. That would be an deletion anomaly. Finally, since we cannot insert the data of a student without also inserting data about a parent and vice versa. That would be an insertion anomaly.

And of course we also have redundancy again. The data of Mr. Böddö is stored three times. It can be said that this redundancy causes the update anomaly. And it means that we again need to enter the same information several times. This very much increases the probability of typos and other errors that could lead to data inconsistency.

For example, it could have well been possible that the information about the students were entered by different teachers. Then, maybe one teacher would know how to enter an ö while another one would just use an o. And baam, our data would be inconsistent, as we would have same parent with two different names in our system.

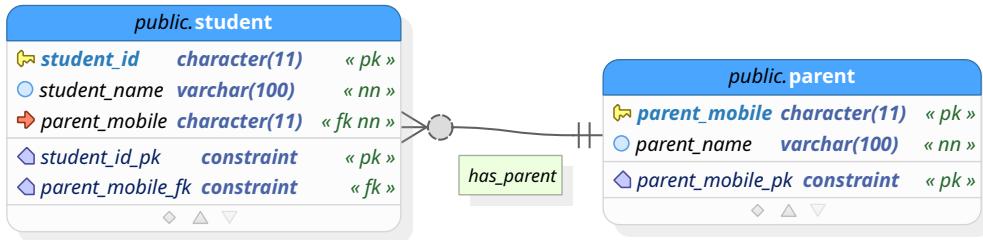


Figure 19.35: Two tables which store data about students and their parents and which, different from Figure 19.34, do not violate the 3NF.

Listing 19.194: he generated SQL code for creating the table `student`. (src)

```

1  -- object: public.student | type: TABLE --
2  -- DROP TABLE IF EXISTS public.student CASCADE;
3  CREATE TABLE public.student (
4      student_id character(11) NOT NULL,
5      student_name varchar(100) NOT NULL,
6      parent_mobile character(11) NOT NULL,
7      CONSTRAINT student_id_pk PRIMARY KEY (student_id)
8 );
9  -- ddl-end --
10 ALTER TABLE public.student OWNER TO postgres;
11 -- ddl-end --

```

Listing 19.195: The generated SQL code for creating the table `parent`. (src)

```

1  -- object: public.parent | type: TABLE --
2  -- DROP TABLE IF EXISTS public.parent CASCADE;
3  CREATE TABLE public.parent (
4      parent_mobile character(11) NOT NULL,
5      parent_name varchar(100) NOT NULL,
6      CONSTRAINT parent_mobile_pk PRIMARY KEY (parent_mobile)
7 );
8  -- ddl-end --
9  ALTER TABLE public.parent OWNER TO postgres;
10 -- ddl-end --

```

Listing 19.196: We add the foreign key constraint to table `student`. (src)

```

1  -- object: parent_mobile_fk | type: CONSTRAINT --
2  -- ALTER TABLE public.student DROP CONSTRAINT IF EXISTS parent_mobile_fk
3  --     ↪ CASCADE;
3  ALTER TABLE public.student ADD CONSTRAINT parent_mobile_fk FOREIGN KEY (
4      ↪ parent_mobile)
4  REFERENCES public.parent (parent_mobile) MATCH SIMPLE
5  ON DELETE NO ACTION ON UPDATE NO ACTION;
6  -- ddl-end --

```

Fixed: Transitive Dependency factored out into own Table Let us fix this problem. We need to bring the data into the 3NF. For this, we need to remove the transitive functional dependency $\text{student_id} \rightarrow \text{parent_mobile} \rightarrow \text{parent_name}$. This dependency is problematic because `parent_mobile` was no key of table `student`, but `parent_name` was determined by `parent_mobile`. As long as these three columns are in the same table, the 3NF will always be violated. So they need to be separated into different tables.

In Figure 19.35 we illustrate a logical model that no longer violates the 3NF. Different from our original sketch in Figure 19.34, we use two tables instead of one. The name of the parent depends on

Listing 19.197: Inserting some data into the tables `parent` and `student`. (src)

```

1  /** Insert data into the database. */
2
3  -- Insert several parent records.
4  INSERT INTO parent (parent_name, parent_mobile) VALUES
5      ('Boddo', '55544466677'),
6      ('Balla', '77788811122');
7
8  -- Insert several student records that are linked to parent records.
9  INSERT INTO student (student_id, student_name, parent_mobile) VALUES
10     ('1234567890', 'Bibbo', '55544466677'),
11     ('1234567891', 'Bebbo', '77788811122'),
12     ('1234567892', 'Bibboto', '55544466677'),
13     ('1234567894', 'Bibboba', '55544466677');
```

Listing 19.198: If we want to get the names of the parents of the students, we now need an `INNER JOIN`. (stored in file `select.sql`; output in Listing 19.199)

```

1  /** Get the names of all parents of all the students. */
2
3  SELECT student_name, parent_name FROM student
4      INNER JOIN parent ON student.parent_mobile = parent.parent_mobile;
```

Listing 19.199: The `stdout` of the program `select.sql` given in Listing 19.198.

```

1  $ psql "postgres://postgres:XXX@localhost/fixed" -v ON_ERROR_STOP=1 -e bf
2      ↳ select.sql
3  student_name | parent_name
4  -----+-----
5  Bibbo        | Boddo
6  Bebbo        | Balla
7  Bibboto      | Boddo
8  Bibboba      | Boddo
9  (4 rows)
10 # psql 16.12 succeeded with exit code 0.
```

their mobile phone number. So we remove this data from the original `student` table and put it into a table `parent`. This table has the primary key `parent_mobile`. This value is unique and identifies a parent. The second column of this table `name`, which obviously depends on that primary key.

The modified table `student` now uses the attribute `parent_mobile` as foreign key. It must be `NOT NULL`, meaning that each row in table `student` is linked to one (and exactly one) row in table `parent`. We now no longer store the name of the parent in the `student` table.

Both tables observe the **1NF**, as there are neither composite attributes nor repeated groups. Both also observe the **2NF**, because there is no compound key and, hence, it is not possible that an attribute could depend on a part of such a key only. They also both observe the **3NF**, because there is no transitive functional dependency.

As a side note, observe that we could also have used a surrogate primary key here for the table `parent`, which would probably be the better solution. This would have allowed us to later change the mobile phone number of a parent, without the need to touch the table `student`. Changing the structure for this would be a nice exercise for the keen reader.

Let us look at this system in action. We first create the table `student` by executing the script given in Listing 19.194. It retained the three columns `student_id`, `student_name`, and `parent_mobile`. But it lost the column `parent_name`. The primary key is still `student_id`. The column `parent_mobile` will become a foreign key to the table for the parent records.

Listing 19.195 then creates the new table `parent` for exactly these parent records. This table has

Listing 19.200: We noticed that the name of the father of Mr. Bibbo, Mr. Bibotto, and Ms. Bibboba is actually Mr. Böddö, not Mr. Boddo. Since we now observe the 3NF, we need to touch only a single records to change this. (stored in file `update.sql`; output in [Listing 19.201](#))

```

1  /** Find the parent Boddo and change his name to Böddö. */
2
3  -- Change the name of the parent with mobile 55544466677 to Böddö.
4  UPDATE parent SET parent_name = 'Böddö'
5    WHERE parent_mobile = '55544466677'
6    RETURNING parent_name, parent_mobile;
7
8  -- Get the names of all students whose parent has mobile 55544466677.
9  SELECT student_name, parent_name FROM student
10   INNER JOIN parent ON student.parent_mobile = parent.parent_mobile
11   WHERE student.parent_mobile = '55544466677';

```

Listing 19.201: The stdout of the program `update.sql` given in [Listing 19.200](#).

```

1 $ psql "postgres://postgres:XXX@localhost/fixed" -v ON_ERROR_STOP=1 -e bf
2   ↪ update.sql
3
4 parent_name | parent_mobile
5 -----
6 Böddö       | 55544466677
7 (1 row)
8
9 UPDATE 1
10  student_name | parent_name
11  -----
12 Bibbo        | Böddö
13 Bibotto      | Böddö
14 Bibboba      | Böddö
15 (3 rows)
16
17 # psql 16.12 succeeded with exit code 0.

```

only two columns: the primary key `parent_mobile`, which is still a string of the fixed length 11, and the parent name.

Script [Listing 19.196](#), adds the foreign key constraint to table `student`. This step is what actually turns `parent_mobile` into a foreign key of table `student`. This constraint enforces that each row in table `student` is related to exactly one row in table `parent`.

We can now insert data into this new DB structure. In [Listing 19.191](#), we begin by storing the two parent records into the table `parent`. The mobile phone numbers and names of Ms. Balla, the mom of Mr. Beppo, as well as Mr. Böddö, the dad of the other three students are stored. Like last time, our data entry specialist did, at first, not know how to write the fancy ö in name Mr. Böddö and resorted to just call him Mr. Boddo. Then we insert four student records for Mr. Bibbo, Mr. Beppo, Mr. Bibotto, and Ms. Bibboba. Here, we do not provide their parent's names anymore. Only their mobile phone numbers, linking to table `parent`, are needed.

At first glance, this new structure looks more complicated. We now have two tables instead of one. If we want to know the names of the parents of the students, a simple `SELECT` will no longer be enough. Instead, we need to merge the data from two tables by using an `INNER JOIN`, as shown in [Listing 19.198](#).

One could argue that the advantage is the reduced redundancy: The name of each parent is entered once and only one. Then again, you may argue that in this simple example, this advantage is more or less offset by the fact that we need to enter their mobile phone numbers in both tables. This only matters in this example because the example is a minimal corner case. We only extracted a single column. Often, there could be many columns. Imagine that we also stored the parent address and ID number and so on. Generally, the redundancy reduction is a good argument.

The usefulness of the new design becomes clearer when we change data. Like in our original

example, a few days after originally entering the data, our data entry specialist found a solution on how to enter the letter ö. In order to fix the data, they created the script Listing 19.192. The `UPDATE` command is now applied to table `parent`. It only touches a single row, as you can see by the result of the `RETURNING` statement. In the same script, we use another `SELECT` to check whether the parent names of Mr. Bibbo, Mr. Bibboto, and Ms. Bibboba have changed. And indeed, they have. The update anomaly has disappeared.

19.3.3.2 Summary

The 3NF is again a rule that reduces the redundancy. This time it deals with the redundancy that emerges if some columns of our table depend on other columns which are not keys. Normally, all the information in the row of a table should be information about the key columns K .

Let's say that there exists a functional dependency of some columns A on some columns B and that it is transitive to the key K , i.e., if we have $K \rightarrow B \rightarrow A$. Then, for any row that has a value β of B , the same value α of A is stored. The value of B determines the value of A . If the same value of B occurs in multiple rows, which it well may, then we will also store the same value of A multiple times. Which we would actually not need to do, because if we know the value of B , we also know the value of A .

So, in order to restore the 3NF, we may separate the table into two. We could use B as primary key in a new table and store the corresponding values of A therein. Then the old B column in the original table becomes a foreign key. This way, we have reduced the redundancy and improved the clarity and cleanliness of our data. It of course also comes with the drawback that we need to re-assemble the data with `INNER JOIN` if we want to know the value of A for a given value of K . Still, this often is a good idea.

Backmatter

Best Practices

Here we discuss the best practices to be followed when working with DBs.

Best Practice 1: A computer science professional is able and always keen to learn new tools. A computer science professional should know dozens if not hundreds of different software tools for different tasks. A software engineer is a craftsman and their knowledge of software is their tool belt.

Best Practice 2: Any professional computer scientist, software developer, software architect, DBA, or system administrator should be familiar with the Linux OS.

Best Practice 3: Regardless which programming language you are using (and we can count SQL as a programming language for DBs), it is important to write code and scripts in a consistent style, to use a consistent naming scheme for all things that can be named, and to follow the generally established best practices and norms for that language.

Best Practice 4: Keywords in SQL should always be written completely in uppercase [124].

Best Practice 5: Table names should be singular nouns written in lowercase without any prefix (i.e., no “tbl_” in front) [59].

Best Practice 6: Always assume that any `float` value is imprecise. Never expect it to be exact [28, 341, 482].

Best Practice 7: Never represent monetary data with floating point numbers [376, 485].

Best Practice 8: Store monetary data using the `DECIMAL` datatype [65, 485].

Best Practice 9: Prefer using surrogate primary keys based on automatically incremented integers [59]. See also [Definition 18.16](#).

Best Practice 10: Whenever using an automatically incremented integer as primary key for a table, name it `id`. While there is some controversy about this topic [213], anybody accessing your DB will immediately understand the meaning of the `id` columns and this practice is used in many sources [59, 338].

Best Practice 11: Never use SQL keywords or reserved words as names, e.g., for columns or tables [59].

Best Practice 12: Regardless which programming language or tool you use to access a DB, you must **never** construct SQL commands using string operations such as concatenation or (string) interpolation. Otherwise, you open the door to [SQLi attacks](#). Always use the proper tools, such as query parameters, for dynamic queries.

Best Practice 13: Only things with multiple attributes should become entity types.

Best Practice 14: Each entity (type) should model exactly one (type of) object from reality (and not more than one) [381].

Best Practice 15: Primary keys should:

1. be unique for each entity (obviously),
2. be immutable over the lifetime of an entity,
3. not be optional, i.e., they should never be allowed to be `NULL`,

4. not be derived attributes,
5. always be single-valued attributes, i.e., not be multivalued attributes,
6. consist of single attributes, i.e., not be based on candidate keys consisting of multiple attributes,
7. be simple attributes, i.e., not composed attributes,
8. be small in terms of the expected required storage size (see also [Best Practice 9](#)).

Best Practice 16: In many application scenarios where historical information needs to be preserved, data in a DB should never be changed or deleted. Changes in the real world should instead be reflected by adding data to the DB.

Best Practice 17: To avoid issues with quotations, it is best to use only lower case character names and underscores (`_`) to separate words for all named things in PgModeler, including tables, columns, and constraints.

Best Practice 18: Constraints should have descriptive names [59]. If some table modification fails, we will see the name of the constraint that was violated. If the name makes sense and is easy to understand, then this makes it easier to find out what went wrong and why.

Best Practice 19: Data should be checked at all levels of an application, in the forms where it is entered, in the DB via constraints, and back in the application when it is loaded from the DB. The more lines of defense we create with constraints, static checks, and dynamic checks, the higher is our chance to discover errors early, to prevent them from propagating, and to pinpoint the reason of errors. This gives us the best chance to locate and fix the error if it is a problem with a program as well as to prevent errors resulting from typos to enter and pollute our DB.

Best Practice 20: Errors should *not* be ignored and input data should *not* be artificially sanitized. Instead, the input of our functions should be checked for validity wherever reasonable. Faulty input should always be signaled by errors breaking the program flow. [In Python,]Exceptions should be raised as early as possible and whenever an unexpected situation occurs.

Useful Tools

Here we discuss some useful tools for working with DBs.

Useful Tool 1: PostgreSQL [161, 306, 339, 433] is an advanced relational DBMS. It is free and open source and the basis for all hands-on examples in our course.

Useful Tool 2: `psql` is a text-based console program that can be used to connect to a PostgreSQL server. From the `psql` console, we can send SQL commands to the PostgreSQL server and receive its answers.

Useful Tool 3: `psycopg` [473] is a library that allows us to connect to the PostgreSQL DBMS from Python code. This way, we can design complex applications in Python that interact with a PostgreSQL DB.

Useful Tool 4: LibreOffice Base [160, 385] offers us a simple GUI that can connect to a DBMS and provides capabilities such as executing SQL queries as well as designing and executing forms and reports.

Useful Tool 5: yEd [384, 497] is a free graph editor that can be used to draw ERDs. It is useful for the conceptual modeling stage in DB design as discussed in Chapter 18. Installation instructions are provided in Chapter 6 and a small hands-on tutorial is given in Section 18.1.

Useful Tool 6: With PgModeler, we have a tool in our hands that allows us to basically draw logical models for DBs as ERDs. These models are easy-to-understand graphics that follow crow's foot notation. PgModeler can connect to a PostgreSQL server and directly push the models to it or load a logical model from the server. It can also export logical models as SQL scripts that we then can execute. It therefore offers us a convenient GUI to design the logical schema of a DB.

Glossary

i! The factorial $a!$ of a natural number $a \in \mathbb{N}_1$ is the product of all positive natural numbers less than or equal to a , i.e., $a! = 1 * 2 * 3 * 4 * \dots * (a - 1) * a$ [71, 148, 276].

i..j with $i, j \in \mathbb{Z}$ and $i \leq j$ is the set that contains all integer numbers in the inclusive range from i to j . For example, $5..9$ is equivalent to $\{5, 6, 7, 8, 9\}$

1NF The first **normal form (NF)** in relational DBs [90, 118, 157, 242]. See also [Section 19.3.1](#).

2NF The second normal form (NF) in relational DBs [88, 92, 118, 157, 242]. See also [Section 19.3.2](#).

3NF The third normal form (NF) in relational DBs [88, 92, 118, 157, 242]. See also [Section 19.3.3](#).

AI Artificial Intelligence, see, e.g., [371]

Android is a common operating system for mobile phones [403].

API An *Application Programming Interface* is a set of rules or protocols that enables one software application or component to use or communicate with another [181].

Bash is the shell used under **Ubuntu Linux**, i.e., the program that “runs” in the **terminal** and interprets your commands, allowing you to start and interact with other programs [51, 299, 502]. Learn more at <https://www.gnu.org/software/bash>.

BCE The time notation *before Common Era* is a non-religious but chronological equivalent alternative to the traditional *Before Christ (BC)* notation, which refers to the years *before* the birth of Jesus Christ [68]. The years BCE are counted down, i.e., the larger the year, the farther in the past. The year 1 BCE comes directly before the year 1 **Common Era (CE)** [387, 504].

C is a programming language, which is very successful in system programming situations [140, 344].

CE The time notation *Common Era* is a non-religious but chronological equivalent alternative to the traditional *Anno Domini (AD)* notation, which refers to the years *after* the birth of Jesus Christ [68]. The years CE are counted upwards, i.e., the smaller they are, the farther they are in the past. The year 1 CE comes directly after the year 1 **BCE** [387, 504].

client In a **client-server architecture**, the **client** is a device or process that requests a service from the **server**. It initiates the communication with the **server**, sends a request, and receives the response with the result of the request. Typical examples for **clients** are web browsers in the internet as well as **clients** for **DBMSes**, such as **psql**.

client-server architecture is a system design where a central **server** receives requests from one or multiple **clients** [37, 272, 313, 354, 363]. These requests and responses are usually sent over network connections. A typical example for such a system is the **World Wide Web (WWW)**, where web **servers** host websites and make them available to web browsers, the **clients**. Another typical example is the structure of **DB** software, where a central **server**, the **DBMS**, offers access to the **DB** to the different **clients**. Here, the **client** can be some **terminal** software shipping with the DBMS, such as **psql**, or the different applications that access the **DBs**.

CSV *Comma-Separated Values* is a very common and simple text format for exchanging tabular or matrix data [391]. Each row in the text file represents one row in the table or matrix. The elements in the row are separated by a fixed delimiter, usually a comma (“,”), sometimes a semicolon (“;”). **Python** offers some out-of-the-box CSV support in the **csv** module [111].

CTE *Common Table Expressions* are [SQL](#) constructs for simplifying complex queries by allowing us to break them into smaller parts which are evaluated only once and, hence, can be reused. CTEs act as temporary named result sets created during query execution and discarded after query completion [27, 161, 492].

DB A *database* is an organized collection of structured information or data, typically stored electronically in a computer system. Databases are discussed in our book [Databases](#) [481].

DB2 developed by IBM is one of the older and popular relational DBMSes [17, 84, 192].

DBA A *database administrator* is the person or group responsible for the effective use of database technology in an organization or enterprise.

DBMS A *database management system* is the software layer located between the user or application and the [DB](#). The DBMS allows the user/application to create, read, write, update, delete, and otherwise manipulate the data in the [DB](#) [496].

DBS A *database system* is the combination of a [DB](#) and a the corresponding [DBMS](#), i.e., basically, an installation of a DBMS on a computer together with one or multiple [DBs](#). DBS = [DB](#) + [DBMS](#).

DOB Date of Birth

Docker provides [OS](#)-level virtualization. A Docker container is something like a more lightweight variant of a virtual machine. Docker containers offer a [Linux](#) runtime environment into which software can be installed and run isolated from the rest of the system. Such containers can be created by scripts. They offer a simple, reproducible, and portable way to configure and ship software components. Learn more at <https://docker.com> or in [238].

dom(a) the domain of an attribute a , see [Definition 18.3](#).

ERD Entity relationship diagrams show the relationships between objects, e.g., between the tables in a [DB](#) and how they reference each other [23, 57, 79–81, 240, 393, 484]

escape sequence Escaping is the process of presenting “forbidden” characters or symbols in a sequence of characters or symbols. In [Python](#) [482], string escapes allow us to include otherwise impossible characters, such as string delimiters, in a string. Each such character is represented by an *escape sequence*, which usually starts with the backslash character (“\”) [158]. In Python strings, the escape sequence `\\"`, for example, stands for `"`, the escape sequence `\\\` stands for `\`, and the escape sequence `\n` stands for a newline or linebreak character. In Python [f-strings](#), the escape sequence `\{\}` stands for a single curly brace `{`. In [PostgreSQL](#) [481], similar C-style escapes (starting with “\”) are supported [423].

exit code When a process terminates, it can return a single integer value (the exit status code) to indicate success or failure [234]. Per convention, an exit code of 0 means success. Any non-zero exit code indicates an error. Under Python, you can terminate the current process at any time by calling `exit` and optionally passing in the exit code that should be returned. If `exit` is not explicitly called, then the interpreter will return an exit code of 0 once the process normally terminates. If the process was terminated by an uncaught [Exception](#), a non-zero exit code, usually 1, is returned.

f-string let you include the results of expressions in strings [55, 169, 174, 182, 285, 407]. They can contain expressions (in curly braces) like `f"a{6-1}b"` that are then transformed to text via [\(string\) interpolation](#), which turns the string to `"a5b"`. F-strings are delimited by `f"..."`.

FD A *functional dependency* exists between two groups of attributes Y and X if the values of X determine the values of Y . This is written as $X \rightarrow Y$. See also [Definition 19.8](#).

Flask is a lightweight Python framework that allows developers to quickly and easily build web applications [3, 85, 449]. It is based on the Python WSGI standard [152]. Learn more at <https://flask.palletsprojects.com>.

Git is a distributed [Version Control Systems \(VCS\)](#) which allows multiple users to work on the same code while preserving the history of the code changes [404, 454]. Learn more at <https://git-scm.com>.

GitHub is a website where software projects can be hosted and managed via the [Git VCS](#) [327, 454]. Learn more at <https://github.com>.

GUI graphical user interface

HR human resources department

HTTP The Hyper Text Transfer Protocol (HTTP) is the protocol linking web browsers to web servers in the [WWW](#) [34, 35, 162, 163, 188].

HTTPS The Hypertext Transfer Protocol Secure (HTTPS) is the encrypted variant of [Hyper Text Transfer Protocol \(HTTP\)](#) where data is sent over [Transport Layer Security \(TLS\)](#) [163, 418].

IDE An *Integrated Developer Environment* is a program that allows the user do multiple different activities required for software development in one single system. It often offers functionality such as editing source code, debugging, testing, or interaction with a distributed version control system. For [Python](#), we recommend using [PyCharm](#). On Apple systems, [Xcode](#) is often used.

IDEF1X The Integration Definition for Information Modeling (IDEF1X) is a standardized syntax for [ERDs](#). It was originally developed by the United States Air Force for data modeling, focusing on entities, relationships, and key structures [54, 223].

IDS Integrated Data Store was developed by Bachman in 1961/62 at General Electric was the first [DBMS](#) using a network-based data model [14, 15]

IMS The Information Management System developed by IBM for the Apollo space program was the first DBMS using a hierarchical data model [33, 246, 254]

Inkscape is a free and open source vector graphics editor, which primarily works with the [SVG](#) format [245, 365]. This is the tool that I would recommend for professional graphic design. Learn more at <https://inkscape.org>.

iOS is the operating system that powers Apple iPhones [69, 406]. Learn more at <https://www.apple.com/ios>.

iPadOS is the operating system that powers Apple iPads [69]. Learn more at <https://www.apple.com/ipados>.

IT information technology

JAD Joint Application Development [67, 287]

Java is another very successful programming language, with roots in the [C](#) family of languages [42, 274].

JavaScript JavaScript is the predominant programming language used in websites to develop interactive contents for display in browsers [153].

JSON *JavaScript Object Notation* is a data interchange format [52, 438] based on [JavaScript](#) [153] syntax. An example of a [JSON](#) document is given in [Figure 1.3.2](#).

JSSP The *Job Shop Scheduling Problem* [41, 264] is one of the most prominent and well-studied scheduling tasks. In a JSSP instance, there are k machines and m jobs. Each job must be processed once by each machine in a job-specific sequence and has a job-specific processing time on each machine. The goal is to find an assignment of jobs to machines that results in an overall shortest makespan, i.e., the schedule which can complete all the jobs in the shortest time. The JSSP is NP -complete [77, 264].

LAMP Stack A system setup for web applications: [Linux](#), Apache (a web server), [MySQL](#), and the server-side scripting language [PHP](#) [64, 201].

LibreOffice is an open source office suite [171, 270, 385] which is a good and free alternative to Microsoft Office. It offers software such as LibreOffice Writer, LibreOffice Calc, and LibreOffice Base. Installation instructions for LibreOffice are given in Chapter 4.

LibreOffice Base is a DBMS that can work on stand-alone files but also connect to other popular relational databases [160, 385]. It is part of LibreOffice [171, 270, 385] and has functionality that is comparable to Microsoft Access [31, 86, 457]. See also Useful Tool 4.

LibreOffice Calc is a spreadsheet software that allows you to arrange and perform calculations with data in a tabular grid. It is a free and open source spread sheet software [270, 385], i.e., an alternative to Microsoft Excel. It is part of LibreOffice [171, 270, 385]. An example of how a LibreOffice Calc table looks like is given in Figure 1.2.

LibreOffice Writer is a free and open source text writing program [501] and part of LibreOffice [171, 270, 385]. It is a good alternative to Microsoft Word.

Linux is the leading open source operating system, i.e., a free alternative for Microsoft Windows [25, 198, 401, 448, 467]. We recommend using it for this course, for software development, and for research. Learn more at <https://www.linux.org>. Its variant Ubuntu is particularly easy to use and install.

literal A literal is a specific concrete value, something that is written down as-is [266, 443]. In Python, for example, `"abc"` is a string literal, `5` is an integer literal, and `23.3` is a `float` literal. In contrast, `sin(3)` is not a literal. Also, while `5` is an integer literal, if we create a variable `a = 5` then `a` is not a literal either (it is a variable). Hence, literals are values that the Python interpreter reads directly from the source code and creates as objects in memory. They are not something that is the result from a computation or the result of a variable lookup. Python supports some type hints for literals, including the type `LiteralString` for string literals and the type `Literal[xyz]` for arbitrary literals `xyz`.

locale A locale corresponds basically to a selection of country or culture for a system or application [235]. The locale then determines a set of country- or culture-dependent settings, such as the text representation for money or dates, or what decimal separators are used. A good example is, for example, that American English readers will interpret the date 7/4/2000 as July 4th, 2000, whereas British English readers will read this as the 7th of April, 2000 [122]. It is therefore important that software printing dates knows whether it is running on an American or British English PC.

localhost is the hostname of the current computer [82, 150]. It is equivalent to the IP address `127.0.0.1`. Any message or package sent to localhost will be sent to the current computer itself.

LTS Many types of software for which new versions appear regularly offer so-called *long-term support* (LTS) versions. Such versions are guaranteed to be supported for a longer-than-usual time. The Ubuntu OS releases a new LTS version every two years [444], for example.

macOS or Mac OS is the operating system that powers Apple Mac(intosh) computers [366, 406]. Learn more at <https://www.apple.com/macos>.

MacPorts is an abstraction layer that allows you to compile, install, upgrade, and use OSS software, often from the Linux realm, on macOS [366]. As such, it is a bit similar to MSYS2, which is for Microsoft Windows. Learn more at <https://www.macports.org>.

MariaDB An open source relational database management system that has forked off from MySQL [11, 12, 26, 149, 281, 355]. See <https://mariadb.org> for more information.

Matplotlib is a Python package for plotting diagrams and charts [216, 217, 232, 316]. Learn more at <https://matplotlib.org> [217].

Microsoft Access is a DBMS that can work on DBs stored in single, stand-alone files but also connect to other popular relational databases [31, 86, 288, 457]. It is part of Microsoft Office. A free and open source alternative to this commercial software is LibreOffice Base.

Microsoft Excel is a spreadsheet program that allows users to store, organize, manipulate, and calculate data in tabular structures [43, 187, 261]. It is part of Microsoft Office. A free alternative to this commercial software is LibreOffice Calc [270, 385].

Microsoft Office is a commercial suite of office software, including Microsoft Excel, Microsoft Word, and Microsoft Access [261]. LibreOffice is a free and open source alternative.

Microsoft SQL Server The Microsoft SQL Server is a successful commercial relational/SQL-based DBMS [6, 328, 488]. Learn more at <https://www.microsoft.com/sql-server> and <https://learn.microsoft.com/en-us/sql>.

Microsoft Windows is a commercial proprietary operating system [50]. It is widely spread, but we recommend using a Linux variant such as Ubuntu for software development and for our course. Learn more at <https://www.microsoft.com/windows>.

Microsoft Word is one of the leading text writing programs [141, 293, 501] and part of Microsoft Office. A free alternative to this commercial software is the LibreOffice Writer.

ML Machine Learning, see, e.g., [392]

MSYS2 Minimal SYStem 2 (MSYS2) is a collection of tools and libraries from the Linux world providing an environment for building, installing, and running native Microsoft Windows software [455]. It is a bit similar to MacPorts, which is for macOS. Learn more at <https://www.msys2.org>.

Mypy is a static type checking tool for Python [267] that makes use of type hints. Learn more at <https://github.com/python/mypy> and in [482].

MySQL An open source relational database management system [49, 149, 358, 433, 489]. MySQL is famous for its use in the LAMP Stack. See <https://www.mysql.com> for more information.

MySQL Workbench is a visual tool for DB designers that offers tools ranging from graphical modeling to performance analysis [289]. Learn more at <https://www.mysql.com/products/workbench>.

N₁ the set of the natural numbers *excluding* 0, i.e., 1, 2, 3, 4, and so on. It holds that $\mathbb{N}_1 \subset \mathbb{Z}$.

NF The *normal forms* define guidelines for the design of relational DBs with the goal to avoid redundancy and to prevent inconsistencies and anomalies [118, 157, 242, 397, 398]. There are several normal forms, 1NF, 2NF, 3NF, and so on, each more restrictive than the other. See also Section 19.3.

NP is the class of computational problems that can be solved in polynomial time by a non-deterministic machine and can be verified in polynomial time by a deterministic machine (such as a normal computer) [180].

NP-complete A decision problem is NP-complete if it is in NP and all problems in NP are reducible to it in polynomial time [180, 350]. A problem is NP-complete if it is NP-hard and if it is in NP.

NP-hard Algorithms that guarantee to find the correct solutions of NP-hard problems [77, 101, 264] need a runtime that is exponential in the problem scale in the worst case. A problem is NP-hard if all problems in NP are reducible to it in polynomial time [180].

NumPy is a fundamental package for scientific computing with Python, which offers efficient array datastructures [137, 196, 232]. Learn more at <https://numpy.org> [302].

OOP Object-Oriented Programming [374]

OR Operations Research (or Operational Research) is the application of sciences such as mathematics and computer science to the management and organization of systems, organizations, enterprises, factories, or projects. It encompasses the development and application of problem-solving methods and techniques (such as mathematical optimization, simulation, queueing theory and other stochastic models) with the goal to improve decision-making and efficiency [133].

Oracle Database The Oracle Database was the first commercial SQL-based relational database [72].

It is still a highly successful proprietary product with many features [32, 256]. Learn more at <https://www.oracle.com/database>.

OS Operating System, the system that runs your computer, see, e.g., Linux, Microsoft Windows, macOS, and Android.

OSS Open source software, i.e., software that can freely be used, whose source code is made available in the internet, and which is usually developed cooperatively over the internet as well [206]. Typical examples are Python, Linux, Git, and PostgreSQL.

Pandas is a Python data analysis and manipulation library [29, 269]. Learn more at <https://pandas.pydata.org> [320].

PD Participatory Design [67, 165]

PDF The Portable Document Format [168, 486] is the format in which provide this book. It is the standard format for the exchange of documents in the internet.

PgModeler the PostgreSQL DB modeler is a tool that allows for graphical modeling of logical schemas for DBs using an ERD-like notation [8]. Learn more at <https://pgmodeler.io>.

pip is the standard tool to install Python software packages from the PyPI repository [227, 333]. To install a package `thepackage` hosted on PyPI, type `pip install thepackage` into the terminal. Learn more at <https://packaging.python.org/installing>.

port A port in networking is a software-defined number associated to a network protocol that receives or transmits communication for a specific service [259]. In the client-server architecture, the server listens for incoming communication connections at a specific port. The clients connect to the network address and port number of the server to establish such connections. port numbers range from 0 to 65535, where the port numbers from 0 to 1023 are so-called well-known ports corresponding to the most common services, e.g., HTTP, the protocol underpinning of the WWW, uses normally port 80.

PostgreSQL An open source object-relational DBMS [161, 306, 339, 433]. See <https://postgresql.org> and Useful Tool 1 for more information.

psql is the client program used to access the PostgreSQL DBMS server. See also Useful Tool 2.

psycopg or, more exactly, `psycopg 3`, is the most popular PostgreSQL adapter for Python, implementing the Python DB API 2.0 specification [268]. Learn more at <https://www.psycopg.org> [473] and Useful Tool 3.

PyCharm is the convenient Python IDE that we recommend for this course [465, 490, 495]. It comes in a free community edition, so it can be downloaded and used at no cost. Learn more at <https://www.jetbrains.com/pycharm>.

PyPI The Python Package Index (PyPI) is an online repository that provides the software packages that you can install with `pip` [47, 440, 464]. Learn more at <https://pypi.org>.

Python The Python programming language [215, 265, 277, 482], i.e., what you will learn about in our book [482]. Learn more at <https://python.org>.

PyTorch is a Python library for deep learning and AI [322, 353]. Learn more at <https://pytorch.org>.

\mathbb{R} the set of the real numbers.

RAD Rapid Application Development [38, 287, 374]

regex A Regular Expression, often called “regex” for short, is a sequence of characters that defines a search pattern for text strings [221, 255, 294, 300]. In Python, the `re` module offers functionality work with regular expressions [255, 356]. In PostgreSQL, regex-based pattern matching is supported as well [336].

relational database A relational DB is a database that organizes data into rows (tuples, records) and columns (attributes), which collectively form tables (relations) where the data points are related to each other [90, 194, 195, 408, 431, 481, 487].

Scikit-learn is a Python library offering various machine learning tools [326, 353]. Learn more at <https://scikit-learn.org>.

SciPy is a Python library for scientific computing [232, 477]. Learn more at <https://scipy.org>.

SDLC Software Development Life Cycle [225, 298]

server In a client-server architecture, the server is a process that fulfills the requests of the clients. It usually waits for incoming communication carrying the requests from the clients. For each request, it takes the necessary actions, performs the required computations, and then sends a response with the result of the request. Typical examples for servers are web servers [64] in the internet as well as DBMSes. It is also common to refer to the computer running the server processes as server as well, i.e., to call it the “server computer” [260].

$\Sigma(R)$ the relation schema of relation R [382], see Definition 19.1.

SQL The Structured Query Language is basically a programming language for querying and manipulating relational databases [72, 117, 123, 131, 222, 291, 412, 419, 420, 431]. It is understood by many DBMSes. You find the SQL commands supported by PostgreSQL in the reference [412].

SQLi attack A SQL injection attack is an attack that is used to target data stored in DBMS by injecting malicious input into code that constructs SQL queries by string concatenation in order to subvert application functionality and perform unauthorized operations [109, 257, 324, 375, 481]. In order to prevent such attacks, queries to DBs should never be constructed via string concatenation or the likes of Python f-strings. Assume that `user_id` was a string variable in a Python program and we construct the query `f"SELECT * FROM data WHERE user_id = {user_id}"`. Notice that the `{user_id}` will be replaced with the value of variable `user_id` during (string) interpolation. If `user_id == "user123; DROP TABLE data;"`, mayhem would ensue when we execute the query [415]. Some programming languages, like Python, offer built-in datatypes (such as `LiteralString` [415]) to annotate string constants that can be used by static type-checkers. At the time of this writing, Mypy does not support this yet [475, 505].

SQLite is an relational DBMS which runs as in-process library that works directly on files as opposed to the client-server architecture used by other common DBMSes. It is the most wide-spread SQL-based DB in use today, installed in nearly every smartphone, computer, web browser, television, and automobile [72, 170, 203, 491]. Learn more at <https://sqlite.org> [411].

SRS Software Requirements Specification document [219, 400, 426, 494]

stderr The standard error stream is one of the three pre-defined streams of a console process (together with the standard input stream (`stdin`) and the `stdout`) [237]. It is the text stream to which the process writes information about errors and exceptions. If an uncaught `Exception` is raised in Python and the program terminates, then this information is written to `stderr`. If you run a program in a terminal, then the text that a process writes to its `stderr` appears in the console.

stdin The standard input stream is one of the three pre-defined streams of a console process (together with the `stdout` and the `stderr`) [237]. It is the text stream from which the process reads its input text, if any. The Python instruction `input` reads from this stream. If you run a program in a terminal, then the text that you type into the terminal while the process is running appears in this stream.

stdout The standard output stream is one of the three pre-defined streams of a console process (together with the `stdin` and the `stderr`) [237]. It is the text stream to which the process writes its normal output. The `print` instruction of Python writes text to this stream. If you run a program in a terminal, then the text that a process writes to its `stdout` appears in the console.

(string) interpolation In Python, string interpolation is the process where all the expressions in an f-string are evaluated and the final string is constructed. An example for string interpolation is turning `f"Rounded {1.234:.2f}"` to `"Rounded 1.23"`.

sudo In order to perform administrative tasks such as installing new software under [Linux](#), root (or “super”) user privileges as needed [87]. A normal user can execute a program in the [terminal](#) as super user by pre-pending `sudo`, often referred to as “super user do.” This requires the root password.

SVG The Scalable Vector Graphics (SVG) format is an [XML](#)-based format for vector graphics [113]. Vector graphics are composed of geometric shapes like lines, rectangles, circles, and text. As opposed to raster / pixel graphics, they can be scaled seamlessly and without artifacts. They are stored losslessly.

TensorFlow is a [Python](#) library for implementing machine learning, especially suitable for training of neural networks [2, 262]. Learn more at <https://www.tensorflow.org>.

terminal A terminal is a text-based window where you can enter commands and execute them [25, 87]. Knowing what a terminal is and how to use it is very essential in any programming- or system administration-related task. If you want to open a terminal under [Microsoft Windows](#), you can press  + , type in `cmd`, and hit . Under [Ubuntu Linux](#),  +  +  opens a terminal, which then runs a [Bash](#) shell inside.

timestamping is a technique used in [DBs](#) to keep track of the creation and/or modification time of data [214, 447]. For each table in a [relational database](#) to which it is applied, an additional column with datatype `TIMESTAMP` [121] is added for the creation timestamp. This column is usually annotated as `NOT NULL DEFAULT CURRENT_TIMESTAMP` [120], meaning that the [DBMS](#) will automatically store the current date and time when a row is created. If data can be changed, then another column for the last update time can be added to the table as well.

TLS Transport Layer Security, a protocol for encrypted communication over the internet [147, 359, 418], used by, e.g., [HTTPS](#).

type hint are annotations that help programmers and static code analysis tools such as [Mypy](#) to better understand what type a variable or function parameter is supposed to be [263, 466]. Python is a dynamically typed programming language where you do not need to specify the type of, e.g., a variable. This creates problems for code analysis, both automated as well as manual: For example, it may not always be clear whether a variable or function parameter should be an integer or floating point number. The annotations allow us to explicitly state which type is expected. They are *ignored* during the program execution. They are basically a piece of documentation.

Ubuntu is a variant of the open source operating system [Linux](#) [87, 201]. We recommend that you use this operating system to follow this class, for software development, and for research. Learn more at <https://ubuntu.com>. If you are in China, you can download it from <https://mirrors.ustc.edu.cn/ubuntu-releases>.

UCS Universal Coded Character Set, see [Unicode](#)

UML The Unified Modeling Language (UML) is a graphical language for visualizing, specifying, constructing, and documenting the artifacts of distributed object systems [48, 309, 458, 459]

Unicode A standard for assigning characters to numbers [224, 445, 460]. The Unicode standard supports basically all characters from all languages that are currently in use, as well as many special symbols. It is the predominantly used way to represent characters in computers and is regularly updated and improved.

unit test Software development is centered around creating the program code of an application, library, or otherwise useful system. A *unit test* is an *additional* code fragment that is not part of that productive code. It exists to execute (a part of) the productive code in a certain scenario (e.g., with specific parameters), to observe the behavior of that code, and to compare whether this behavior meets the specification [30, 307, 308, 317, 370, 446]. If not, the unit test fails. The use of unit tests is at least threefold: First, they help us to detect errors in the code. Second, program code is usually not developed only once and, from then on, used without change indefinitely. Instead, programs are often updated, improved, extended, and maintained over a long time. Unit tests can help us to detect whether such changes in the program code, maybe after years, violate the specification or, maybe, cause another, depending, module of the program to violate its specification. Third, they are part of the documentation or even specification of a program.

URI A *Uniform Resource Identifier* is an identifier for an abstract or physical resource in the internet [499]. It can be a **URL**, a name, or both. URIs are supersets of **URLs**. The connection strings of the **PostgreSQL** DBMS are examples for URIs.

URL A *Uniform Resource Locator* identifies a resource in the **WWW** and a way to obtain it by describing a network access mechanism. The most notable example of URLs is the text you write into web browsers to visit websites [36]. URLs are subsets of **URIs**.

UTF-8 The *UCS Transformation Format 8* is one standard for encoding **Unicode** characters into a binary format that can be stored in files [224, 499]. It is the world wide web's most commonly used character encoding, where each character is represented by one to four bytes. It is backwards compatible with ASCII.

VCS A *Version Control System* is a software which allows you to manage and preserve the historical development of your program code [454]. A distributed VCS allows multiple users to work on the same code and upload their changes to the server, which then preserves the change history. The most popular distributed VCS is **Git**.

virtual environment A virtual environment is a directory that contains a local **Python** installation [292, 478]. It comes with its own package installation directory. Multiple different virtual environments can be installed on a system. This allows different applications to use different versions of the same packages without conflict, because we can simply install these applications into different **virtual environments**.

WeChat 微信, produced by Tencent (腾讯公司) is the most-used messenger application in China. It integrates payment capabilities, video and voice chats, as well as social plugins and location-based services. See also <https://weixin.qq.com/>.

WWW World Wide Web [34, 134]

Xcode is offers the tools for developing, **testing**, and distributing applications as well as an **IDE** for Apple platforms such as **macOS** and **iOS** [373].

XML The *Extensible Markup Language* is a text-based language for storing and transporting of data [53, 95, 251]. It allows you to define elements in the form `<myElement myAttr="x">...text..</myElement>`. Different from **CSV**, elements in XML can be hierarchically nested, like `<a><c>test</c>bla`, and thus easily represent tree structures. XML is one of most-used data interchange formats. To process XML in Python, use the `defusedxml` library [200], as it protects against several security issues. Another example of an **XML** document is given in [Figure 1.3.1](#).

YAML *YAML Ain't Markup Language™* is a human-friendly data serialization language for all programming languages [94, 142, 251]. It is widely used for configuration files in the DevOps environment. See <https://yaml.org> for more information. An example of a **YAML**-document is given in [Figure 1.3.3](#).

yEd is a graph editor for high-quality graph-based diagrams [384, 497], suitable to draw, e.g., technology-independent **ERDs**, control flow charts, or **UML** class diagrams. An online version of the editor is available at <https://www.yworks.com/yed-live>. Learn more at <https://www.yworks.com/products/yed>. See also [Useful Tool 5](#).

\mathbb{Z} the set of the integers numbers including positive and negative numbers and 0, i.e., $\dots, -3, -2, -1, 0, 1, 2, 3, \dots$, and so on. It holds that $\mathbb{Z} \subset \mathbb{R}$.

SQL Commands

(...), 338
*, 108, 111
. , 115
\$, 115, 116
%, 111
^, 115, 116
~, 115, 116
||, 131, 162, 318, 333–335,
 341
\d, 116
\q, 95, 97, 99, 104
\w, 115

ADD CONSTRAINT, 253
AGE, 322
ALL, 335
ALTER
 ROLE, 30, 31
 TABLE, 247, 253, 263,
 269, 317
 USER, 30, 31
ALTER TABLE, 263
AND, 122, 126, 132–134,
 241, 242
ANY, 334, 340
ARRAY, 334, 340
AS, 111, 126, 131, 338
AVG, 111, 120

BY DEFAULT, 105, 106,
 113, 121

CHARACTER, 233,
 235–237, 358

CHECK, 115, 231, 237,
 239–242

COALESCE, 334, 341

COMMENT ON, 96, 98

CONSTRAINT, 115, 116,
 246

 CHECK, 114–116, 122,
 125, 231, 237–242

COUNT, 120, 126

CREATE
 DATABASE, 96, 97,
 246

 ROLE, 94

 SEQUENCE, 284

TABLE, 101–103, 106,
 113, 114, 246

USER, 94, 95

VIEW, 131, 322

CREATE ROLE, 91, 94

CREATE USER, 91, 94

CURRENT_DATE, 317,
 322

CURRENT_TIME, 317

CURRENT_TIMESTAMP,
 317, 375

DATABASE, 96, 246

 DROP, 171, 327

DATE, 122, 125, 235, 238,
 322

datname, 96, 98

DEFAULT, 317

DECIMAL, 105, 122, 125,
 365

DEFAULT, 284, 317

DELETE, 137, 352

DELETE FORM, 344

DELETE FROM, 136

DISTINCT, 338, 344, 352

 ON, 338

DISTINCT ON, 340

DROP, 171

 DATABASE, 171, 173,
 327

 IF EXISTS, 171–173

 TABLE, 143, 171, 173

 USER, 171, 173

 VIEW, 171, 172

DROP DATABASE, 312

ENCRYPTED, 94

f, 119, 120

FALSE, 119, 120

FOREIGN KEY, 250, 253

GENERATED, 105, 106,
 113, 121, 322

GENERATED...AS
 IDENTITY, 106

GROUP BY, 120, 126

IDENTITY, 105, 106, 113,
 121

IF EXISTS, 171, 173

ILIKE, 119, 332, 334, 335,
 338, 340

INNER JOIN, 253, 282,
 312, 318, 327,
 330, 332, 338,
 340, 353, 354,
 356, 361, 362

INSERT, 137, 341, 344

INSERT INTO, 107, 108,
 116, 122, 141,
 142, 144, 228,
 253, 351

INT, 105, 106, 113, 121,
 122, 253

INTEGER, 105, 344

IS, 98

JOIN, 133, 192, 258

 INNER, 128

 LEFT, 125, 126, 128

 LEFT OUTER, 125

LEFT OUTER JOIN, 125

LIKE, 111, 115, 119, 345

LIMIT, 111

NEXTVAL, 284–286

NOT, 122

NOT NULL, 104, 105,
 113–116, 122,
 125, 233,
 235–238, 243,
 252, 310, 317,
 330, 332, 338,
 341, 358, 361

NULL, 125, 126, 201, 204,
 205, 208, 229,
 230, 233,
 332–334, 341, 365

ON, 126

OR, 122, 334

ORDER BY, 111, 120, 126,
 129, 162

 ASC, 111

 DESC, 111, 131, 132

OWNER, 96, 97

- PASSWORD, 30, 94
pg_catalog.pg_tables, 106, 116
pg_catalog.pg_user, 94, 95
pg_database, 96
pg_databases, 98
postgres, 95
PRIMARY KEY, 105, 106, 113, 121, 231, 237, 239
REAL, 194
REFERENCES, 121, 122, 125, 128, 129, 135, 173, 192, 231, 253, 316, 317, 330, 336, 338, 340
RETURNING, 135–137, 279, 286, 312, 345, 346, 352, 359, 363
SELECT, 92, 104, 322, 344–346, 352, 359, 362, 363
SELECT DISTINCT, 344
SELECT FROM, 253
SELECT...FROM, 43, 94–96, 98, 106, 108, 111, 119, 123, 131, 140, 141, 156, 162, 228, 338
SEQUENCE, 308–310, 312 CREATE, 284
SET, 359
SMALLINT, 194
STORED, 322
SUBSTR, 242
SUM, 131, 132
t, 119, 120
TABLE
ALTER, 247, 253, 263, 269, 317
CREATE, 101, 106, 113, 114, 246
DROP, 171
tablename, 106
tableowner, 106
TIMESTAMP, 322, 375
TO_CHAR, 242
TRUE, 114, 119, 120
UNION, 231, 338, 340, 341
UNION ALL, 113
UNIQUE, 104, 108, 110, 113–116, 125, 131, 162, 236, 243, 252, 263, 310, 311, 325, 327, 330
UPDATE, 137, 263, 280, 341, 346, 353, 359, 363
UPDATE...SET, 134
username, 95
USER, 94
DROP, 171
VALUES, 107, 142
VARCHAR, 102, 104, 106, 113, 115, 233, 235–237, 331, 332, 344, 358
VIEW, 280, 322
CREATE, 131, 322
DROP, 171
VIRTUAL, 322
WHERE, 94, 106, 108, 111, 119, 130, 132–134, 136, 141, 318, 344
WITH, 280, 341
ENCRYPTED
PASSWORD, 94

Bibliography

- [1] *A Timeline of Database History & Database Management*. Boston, MA, USA: Quickbase, Inc., Feb. 18, 2022–Aug. 25, 2023. URL: <https://www.quickbase.com/articles/timeline-of-database-history> (visited on 2025-01-11) (cit. on p. 8).
- [2] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Gordon Murray, Benoit Steiner, Paul A. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. “TensorFlow: A System for Large-Scale Machine Learning”. In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI’2016)*. Nov. 2–4, 2016, Savannah, GA, USA. Ed. by Kimberly Keeton and Timothy Roscoe. Berkeley, CA, USA: USENIX Association, 2016, pp. 265–283. ISBN: 978-1-931971-33-1. URL: <https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi> (visited on 2024-06-26) (cit. on pp. 139, 375).
- [3] Olatunde Adedeji. *Full-Stack Flask and React*. Birmingham, England, UK: Packt Publishing Ltd, Nov. 2023. ISBN: 978-1-80324-844-8 (cit. on p. 369).
- [4] “Aggregate Functions”. In: *PostgreSQL Documentation*. 17.4. The PostgreSQL Global Development Group (PGDG), Feb. 20, 2025. Chap. 9.21. URL: <https://www.postgresql.org/docs/17/functions-aggregate.html> (visited on 2025-02-27) (cit. on pp. 111, 131).
- [5] “`ALTER TABLE`”. In: *PostgreSQL Documentation*. 17.4. The PostgreSQL Global Development Group (PGDG), Feb. 20, 2025. URL: <https://www.postgresql.org/docs/17/sql-altertable.html> (visited on 2025-11-18) (cit. on p. 247).
- [6] Dirk Angermann. *T-SQL-Abfragen für Microsoft SQL-Server 2022*. Blaufelden, Schwäbisch Hall, Baden-Württemberg, Germany: mitp Verlags GmbH & Co. KG, June 2024. ISBN: 978-3-7475-0633-2 (cit. on pp. 15, 17, 101, 372).
- [7] Database Management Systems ANSI/X3/SPARC Study Group. *Framework Report on Database Management Systems*. Montvale, NJ, USA: American Federation of Information Processing Societies (AFIPS) Press, 1978. Also published as [453] (cit. on pp. 6, 409).
- [8] Raphael “rkhaotix” Araújo e Silva. *pgModeler – PostgreSQL Database Modeler*. Palmas, Tocantins, Brazil, 2006–2025. URL: <https://pgmodeler.io> (visited on 2025-04-12) (cit. on pp. iv, 20, 72, 232, 373).
- [9] “Arrays”. In: *PostgreSQL Documentation*. 17.4. The PostgreSQL Global Development Group (PGDG), Feb. 20, 2025. Chap. 8.15. URL: <https://www.postgresql.org/docs/17/arrays.html> (visited on 2025-05-08) (cit. on p. 334).
- [10] David Ascher, ed. *Python Cookbook*. 1st ed. Sebastopol, CA, USA: O'Reilly Media, Inc., July 2002. ISBN: 978-0-596-00167-4 (cit. on p. 20).
- [11] Adam Aspin and Karine Aspin. *Query Answers with MariaDB – Volume I: Introduction to SQL Queries*. Tetras Publishing, Oct. 2018. ISBN: 978-1-9996172-4-0. See also [12] (cit. on pp. 14, 371, 379).
- [12] Adam Aspin and Karine Aspin. *Query Answers with MariaDB – Volume II: In-Depth Querying*. Tetras Publishing, Oct. 2018. ISBN: 978-1-9996172-5-7. See also [11] (cit. on pp. 14, 371, 379).
- [13] Charles William “Charlie” Bachman. “Data Structure Diagrams”. *DATA BASE – ACM SIGMIS Database: The DATABASE for Advances in Information Systems* 1(2):4–10, Sum. 1969. New York, NY, USA: Association for Computing Machinery (ACM). ISSN: 0095-0033. doi:[10.1145/1017466.1017467](https://doi.org/10.1145/1017466.1017467) (cit. on pp. 12, 195, 212, 213).

- [14] Charles William "Charlie" Bachman. "Software for Random Access Processing". *Datamation* 9(4):36–41, Apr. 1965. Chicago, IL, USA: Technical Publishing Co. and Boston, MA, USA: Cahners Publishing Company. ISSN: 0011-6963 (cit. on pp. 10, 226, 370).
- [15] Charles William "Charlie" Bachman. "The Origin of the Integrated Data Store (IDS): The First Direct-Access DBMS". *IEEE Annals of the History of Computing* 31(4):42–54, Oct.–Dec. 2009. Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers (IEEE). ISSN: 1058-6180. doi:10.1109/MAHC.2009.110. URL: <https://tschwarz.msccs.mu.edu/Classes/DB23/HW/bachmanIDS.pdf> (visited on 2025-01-08) (cit. on pp. 10, 11, 226, 370).
- [16] Peter Bailis, Joseph M. Hellerstein, and Michael Stonebraker, eds. *Readings in Database Systems*. 5th ed. Stanford, CA, USA, 2015. URL: <http://www.redbook.io> (visited on 2025-04-04) (cit. on p. 17).
- [17] Jim Bainbridge, Hernando Bedoya, Rob Bestgen, Mike Cain, Dan Cruikshank, Jim Denton, Doug Mack, Tom McKinley, and Simona Pacchiarini. *SQL Procedures, Triggers, and Functions on IBM DB2 for i*. Durham, NC, USA: IBM Redbooks, Apr. 2016. ISBN: 978-0-7384-4164-1 (cit. on pp. 15, 18, 369).
- [18] Paul Baran. *A Briefing on the Distributed Adaptive Message-Block Network*. Tech. rep. P-3127. Santa Monica, CA, USA: The RAND Corporation, Apr. 1965. URL: <https://www.rand.org/content/dam/rand/pubs/papers/2008/P3127.pdf> (visited on 2025-01-20) (cit. on p. 11).
- [19] Paul Baran. *On a Distributed Command and Control System Configuration*. Memorandum RM-2632. Santa Monica, CA, USA: The RAND Corporation, Dec. 31, 1960. URL: https://www.rand.org/content/dam/rand/pubs/research_memoranda/2009/RM2632.pdf (visited on 2025-01-20) (cit. on p. 11).
- [20] Paul Baran. *On Distributed Communications Networks*. Tech. rep. P-2626. Santa Monica, CA, USA: The RAND Corporation, Sept. 1962. URL: <https://www.rand.org/content/dam/rand/pubs/papers/2005/P2626.pdf> (visited on 2025-01-20) (cit. on pp. 11, 12).
- [21] Paul Baran. *On Distributed Communications: I. Introduction to Distributed Communications Networks*. Memorandum RM-3042-PR. Santa Monica, CA, USA: The RAND Corporation, Aug. 1964. URL: https://www.rand.org/content/dam/rand/pubs/research_memoranda/2006/RM3420.pdf (visited on 2025-01-20) (cit. on p. 11).
- [22] Paul Baran. *On Distributed Communications: V. History, Alternative Approaches, and Comparisons*. Memorandum RM-3097-PR. Santa Monica, CA, USA: The RAND Corporation, Aug. 1964. URL: https://www.rand.org/content/dam/rand/pubs/research_memoranda/2008/RM3097.pdf (visited on 2025-01-20) (cit. on p. 11).
- [23] Richard Barker. *Case*Method: Entity Relationship Modelling (Oracle)*. 1st ed. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., Jan. 1990. ISBN: 978-0-201-41696-1 (cit. on pp. 184, 193, 195, 369).
- [24] G. A. Barnard III and Louis Fein. "Organization and Retrieval of Records Generated in a Large-Scale Engineering Project". In: *Papers and Discussions Presented at the 1958 Eastern Joint Computer Conference. Modern Computers: Objectives, Designs, Applications (AIEE-ACM-IRE) 1958 (Eastern)*. Dec. 3–5, 1958, Philadelphia, PA, USA. Ed. by John M. Broomall. New York, NY, USA: Association for Computing Machinery (ACM), 1958, pp. 59–63. ISBN: 978-1-4503-7866-6. doi:10.1145/1458043.1458058 (cit. on p. 9).
- [25] Daniel J. Barrett. *Efficient Linux at the Command Line*. Sebastopol, CA, USA: O'Reilly Media, Inc., Feb. 2022. ISBN: 978-1-0981-1340-7 (cit. on pp. 21, 371, 375).
- [26] Daniel Bartholomew. *Learning the MariaDB Ecosystem: Enterprise-level Features for Scalability and Availability*. New York, NY, USA: Apress Media, LLC, Oct. 2019. ISBN: 978-1-4842-5514-8 (cit. on pp. 14, 371).
- [27] Daniel Bartholomew. *MariaDB and MySQL Common Table Expressions and Window Functions Revealed*. New York, NY, USA: Apress Media, LLC, Nov. 2017. ISBN: 978-1-4842-3120-3 (cit. on p. 369).
- [28] Gary Baumgartner, Danny Heap, and Richard Krueger. "Numerical Systems". In: *Course Notes for CSC165H: Mathematical Expression and Reasoning for Computer Science*. Toronto, ON, Canada: Department of Computer Science, University of Toronto, Aut. 2006. Chap. 7. URL: <https://www.cs.toronto.edu/~krueger/csc165h/f06/lectures/ch7.pdf> (visited on 2024-07-27) (cit. on pp. 105, 365).

- [29] David M. Beazley. "Data Processing with Pandas". *:login: Usenix Magazin* 37(6), Dec. 2012. Berkeley, CA, USA: USENIX Association. ISSN: 1044-6397. URL: <https://www.usenix.org/publications/login/december-2012-volume-37-number-6/data-processing-pandas> (visited on 2024-06-25) (cit. on pp. 139, 373).
- [30] Kent L. Beck. *JUnit Pocket Guide*. Sebastopol, CA, USA: O'Reilly Media, Inc., Sept. 2004. ISBN: 978-0-596-00743-0 (cit. on p. 375).
- [31] Ben Beitler. *Hands-On Microsoft Access 2019*. Birmingham, England, UK: Packt Publishing Ltd, Mar. 2020. ISBN: 978-1-83898-747-3 (cit. on pp. 15, 18, 45, 146, 371).
- [32] Peter Belknap, John Beresniewicz, Benoît Dageville, Karl Dias, Yakov Shafranovich, and Khaled Yagoub. "A Decade of Oracle Database Manageability". 34(4):20–27, Dec. 2011. URL: http://sites.computer.org/debull/A11dec/DODM_V2.pdf (cit. on pp. 15, 373).
- [33] Uri Berman, Jackie Berman, and Bob Patrick. *The Birth of IMS/360*. Tech. rep. 102762458. Mountain View, CA, USA: Computer History Museum (CHM), Apr. 2007. URL: <https://archive.computerhistory.org/resources/text/2016/12/102762458-05-01-acc.pdf> (visited on 2025-01-08) (cit. on pp. 11, 226, 370).
- [34] Tim Berners-Lee. *Re: Qualifiers on Hypertext links...* Geneva, Switzerland: World Wide Web project, European Organization for Nuclear Research (CERN) and Newsgroups: alt.hypertext, Aug. 6, 1991. URL: <https://www.w3.org/People/Berners-Lee/1991/08/art-6484.txt> (visited on 2025-02-05) (cit. on pp. 370, 376).
- [35] Tim Berners-Lee, Roy T. Fielding, and Henrik Fristyk Nielsen. *Hypertext Transfer Protocol -- HTTP/1.0*. Request for Comments (RFC) 1945. Wilmington, DE, USA: Internet Engineering Task Force (IETF), May 1996. URL: <https://www.ietf.org/rfc/rfc1945.txt> (visited on 2025-02-05) (cit. on p. 370).
- [36] Tim Berners-Lee, Larry Masinter, and Mark P. McCahill. *Uniform Resource Locators (URL)*. Request for Comments (RFC) 1738. Wilmington, DE, USA: Internet Engineering Task Force (IETF), Dec. 1994. URL: <https://www.ietf.org/rfc/rfc1738.txt> (visited on 2025-02-05) (cit. on p. 376).
- [37] Alex Berson. *Client/Server Architecture*. 2nd ed. Computer Communications Series. New York, NY, USA: McGraw-Hill, Mar. 29, 1996. ISBN: 978-0-07-005664-0 (cit. on pp. 12, 368).
- [38] Paul Beynon-Davies, Chris Carne, Hugh Mackay, and Doug Tudhope. "Rapid Application Development (RAD): An Empirical Review". *European Journal of Information Systems (EJIS)* 8(3), Sept. 1999. London, England, UK: Taylor and Francis Ltd. ISSN: 0960-085X. doi:10.1057/palgrave.ejis.3000325. URL: <https://www.researchgate.net/publication/31978101> (visited on 2025-10-07) (cit. on pp. 181, 182, 373).
- [39] K.S. Bhaskar. *The Heritage and Legacy of M (MUMPS) – and the Future of YottaDB*. Malvern, PA, USA: YottaDB, LLC., Feb. 14, 2018. URL: <https://yottadb.com/heritage-legacy-m-mumps-future-yottadb> (visited on 2025-04-04) (cit. on p. 226).
- [40] Elizabeth Bjarnason, Franz Lang, and Alexander Mjöberg. "An Empirically based Model of Software Prototyping: A Mapping Study and a Multi-Case Study". *Empirical Software Engineering: An International Journal* 28(5):115, Aug. 2023. London, England, UK: Springer Nature Limited. ISSN: 1382-3256. doi:10.1007/S10664-023-10331-W. URL: <https://www.researchgate.net/publication/373517644> (visited on 2025-10-07) (cit. on p. 181).
- [41] Jacek Błażewicz, Wolfgang Domschke, and Erwin Pesch. "The Job Shop Scheduling Problem: Conventional and New Solution Techniques". *European Journal of Operational Research* 93(1):1–33, Aug. 1996. Amsterdam, The Netherlands: Elsevier B.V. ISSN: 0377-2217. doi:10.1016/0377-2217(95)00362-2 (cit. on p. 370).
- [42] Joshua Bloch. *Effective Java*. Reading, MA, USA: Addison-Wesley Professional, May 2008. ISBN: 978-0-321-35668-0 (cit. on p. 370).
- [43] Bernard Obeng Boateng. *Data Modeling with Microsoft Excel*. Birmingham, England, UK: Packt Publishing Ltd, Nov. 2023. ISBN: 978-1-80324-028-2 (cit. on pp. 2, 372).
- [44] Barry W. Boehm. "A Spiral Model of Software Development and Enhancement". In: *International Workshop on the Software Process and Software Environments*. Mar. 27–29, 1985, Coto de Caza, Trabuco Canyon, CA, USA. New York, NY, USA: Association for Computing Machinery (ACM), 1985, pp. 22–42. ISBN: 978-0-89791-202-0. doi:10.1145/12944.12948. Also contained in [343] (cit. on pp. 181, 182).

- [45] Barry W. Boehm. "A Spiral Model of Software Development and Enhancement". *Computer* 21(5):61–72, May 1988. Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers (IEEE). ISSN: 0018-9162. doi:10.1109/2.59 (cit. on pp. 181, 182).
- [46] Barry W. Boehm and Philip N. Papaccio. "Understanding and Controlling Software Costs". *IEEE Transactions on Software Engineering* 14(10):1462–1477, Oct. 1988. Los Alamitos, CA, USA: IEEE Computer Society. ISSN: 0098-5589. doi:10.1109/32.6191 (cit. on p. 187).
- [47] Ethan Bommarito and Michael Bommarito. *An Empirical Analysis of the Python Package Index (PyPI)*. arXiv.org: Computing Research Repository (CoRR) abs/1907.11073. Ithaca, NY, USA: Cornell University Library, July 26, 2019. doi:10.48550/arXiv.1907.11073. URL: <https://arxiv.org/abs/1907.11073> (visited on 2024-08-17). arXiv:1907.11073v2 [cs.SE] 26 Jul 2019 (cit. on p. 373).
- [48] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language Reference Manual*. 1st ed. Reading, MA, USA: Addison-Wesley Professional, Jan. 1999. ISBN: 978-0-201-57168-4 (cit. on pp. 195, 219, 375).
- [49] Silvia Botros and Jeremy Tinley. *High Performance MySQL*. 4th ed. Sebastopol, CA, USA: O'Reilly Media, Inc., Nov. 2021. ISBN: 978-1-4920-8051-0 (cit. on pp. 14, 372).
- [50] Ed Bott. *Windows 11 Inside Out*. Hoboken, NJ, USA: Microsoft Press, Pearson Education, Inc., Feb. 2023. ISBN: 978-0-13-769132-6 (cit. on pp. 22, 372).
- [51] Ron Brash and Ganesh Naik. *Bash Cookbook*. Birmingham, England, UK: Packt Publishing Ltd, July 2018. ISBN: 978-1-78862-936-2 (cit. on p. 368).
- [52] Tim Bray. *The JavaScript Object Notation (JSON) Data Interchange Format*. Request for Comments (RFC) 8259. Wilmington, DE, USA: Internet Engineering Task Force (IETF), Dec. 2017. URL: <https://www.ietf.org/rfc/rfc8259.txt> (visited on 2025-02-05) (cit. on pp. 2, 370).
- [53] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, and Eve Maler, eds. *Extensible Markup Language (XML) 1.0 (Fifth Edition)*. W3C Recommendation. Wakefield, MA, USA: World Wide Web Consortium (W3C), Nov. 26, 2008–Feb. 7, 2013. URL: <http://www.w3.org/TR/2008/REC-xml-20081126> (visited on 2024-12-15) (cit. on pp. 2, 376).
- [54] Ronald H. Brown, Mary L. Good, Arati Prabhakar, and James H. Burrows. *Integration Definition for Information Modeling (IDEF1X)*. Federal Information Processing Standards Publication (FIPS PUB) 184. Gaithersburg, MD, USA: U.S. Department of Commerce, National Institute of Standards and Technology (NIST), Dec. 21, 1993. URL: <https://nvlpubs.nist.gov/nistpubs/Legacy/FIPS/fipspub184.pdf> (visited on 2025-03-29). Software Standard (cit. on pp. 195, 370).
- [55] Florian Bruhin. *Python f-Strings*. Winterthur, Switzerland: Bruhin Software, May 31, 2023. URL: <https://fstring.help> (visited on 2024-07-25) (cit. on p. 369).
- [56] Ben Brumm. "79 Data Modeling Tools Compared". In: *Database Star*. Armadale, VIC, Australia: Elevated Online Services PTY Ltd., Oct. 20, 2018–Mar. 26, 2023. URL: <https://www.databasestar.com/data-modeling-tools> (visited on 2025-04-05) (cit. on p. 195).
- [57] Ben Brumm. "A Guide to the Entity Relationship Diagram (ERD)". In: *Database Star*. Armadale, VIC, Australia: Elevated Online Services PTY Ltd., July 30, 2019–Dec. 23, 2023. URL: <https://www.databasestar.com/entity-relationship-diagram> (visited on 2025-03-29) (cit. on pp. 214, 369).
- [58] Ben Brumm. *Database Star*. Armadale, VIC, Australia: Elevated Online Services PTY Ltd., Dec. 2024. URL: <https://www.databasestar.com> (visited on 2025-03-29) (cit. on p. 18).
- [59] Ben Brumm. "SQL Best Practices and Style Guide". In: *Database Star*. Armadale, VIC, Australia: Elevated Online Services PTY Ltd., Dec. 10, 2024. URL: <https://www.databasestar.com/sql-best-practices> (visited on 2025-02-26) (cit. on pp. 94, 101, 106, 121, 237, 365, 366).
- [60] Corentin Burnay, Ivan Jureta, and Stéphane Faulkner. "What stakeholders will or will not say: A theoretical and empirical study of topic importance in Requirements Engineering elicitation interviews". *Information Systems: Databases: Their Creation, Management and Utilization* 46:61–81, Dec. 2014. Oxford, Oxfordshire, England, UK: Pergamon Press Ltd., now Amsterdam, The Netherlands: Elsevier B.V. ISSN: 0306-4379. doi:10.1016/J.IS.2014.05.006 (cit. on p. 188).

- [61] Thomas Burns, Elizabeth N. Fong, David Jefferson, Richard Knox, Leo Mark, Christopher Reedy, Louis Reich, Nick Roussopoulos, and Walter Truszkowski. "Reference Model for DBMS Standardization, Database Architecture Framework Task Group (DAFTG) of the ANSI/X3/SPARC Database System Study Group". *ACM SIGMOD Record* 15(1):19–58, May 1985–Mar. 1986. New York, NY, USA: Association for Computing Machinery (ACM). ISSN: 0163-5808. doi:10.1145/16342.16343 (cit. on p. 6).
- [62] *Business Analysis Benchmark 2009: The Path to Success*. Wilmington, DE, USA: Information Architecture Group (IAG) Consulting, 2009–Nov. 8, 2011. URL: https://www.iag.biz/wp-content/uploads/Business_Analysis_Benchmark_Full_report_2009.pdf (visited on 2025-03-27) (cit. on p. 187).
- [63] Rudd H. Canaday, R.D. Harrison, Evan L. Ivie, J.L. Ryder, and L.A. Wehr. "A Back-End Computer for Data Base Management". *Communications of the ACM (CACM)* 17(10):575–582, Oct. 1974. New York, NY, USA: Association for Computing Machinery (ACM). ISSN: 0001-0782. doi:10.1145/355620.361172 (cit. on p. 12).
- [64] Jason Cannon. *High Availability for the LAMP Stack*. Shelter Island, NY, USA: Manning Publications, June 2022 (cit. on pp. 14, 370, 374).
- [65] Cardinal ("cardinalby"). *Storing Currency Values: Data Types, Caveats, Best Practices*. San Francisco, CA, USA: GitHub Inc, Jan. 8, 2023. URL: <https://cardinalby.github.io/blog/post/best-practices/storing-currency-values-data-types> (visited on 2025-02-27) (cit. on pp. 105, 365).
- [66] John Vincent Carlis and Joseph D. Maguire. *Mastering Data Modeling: A User Driven Approach*. Reading, MA, USA: Addison-Wesley Professional, Nov. 2000. ISBN: 978-0-201-70045-9 (cit. on pp. 17, 195, 213).
- [67] Erran Carmel, Randall D. Whitaker, and Joey F. George. "PD and Joint Application Design: A Transatlantic Comparison". *Communications of the ACM (CACM)* 36(6):40–48, June 1993. New York, NY, USA: Association for Computing Machinery (ACM). ISSN: 0001-0782. doi:10.1145/153571.163265 (cit. on pp. 188, 370, 373).
- [68] Antonio Cavacini. "Is the CE/BCE notation becoming a standard in scholarly literature?" *Scientometrics* 102(2):1661–1668, July 2015. London, England, UK: Springer Nature Limited. ISSN: 0138-9130. doi:10.1007/s11192-014-1352-1 (cit. on p. 368).
- [69] Josh Centers. *Take Control of iOS 18 and iPadOS 18*. San Diego, CA, USA: Take Control Books, Dec. 2024. ISBN: 978-1-990783-55-5 (cit. on p. 370).
- [70] Cfwlrl. "Multics". In: *BetaWiki: An Open Encyclopedia of Software History*. Mar. 20, 2023–June 1, 2024. URL: <https://betawiki.net/wiki/Multics> (visited on 2025-05-23) (cit. on p. 10).
- [71] Noureddine Chabini and Rachid Beguenane. "FPGA-Based Designs of the Factorial Function". In: *IEEE Canadian Conference on Electrical and Computer Engineering (CCECE'2022)*. Sept. 18–20, 2022, Halifax, NS, Canada. Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers (IEEE), 2022, pp. 16–20. ISBN: 978-1-6654-8432-9. doi:10.1109/CCECE49351.2022.9918302 (cit. on p. 368).
- [72] Donald D. Chamberlin. "50 Years of Queries". *Communications of the ACM (CACM)* 67(8):110–121, Aug. 2024. New York, NY, USA: Association for Computing Machinery (ACM). ISSN: 0001-0782. doi:10.1145/3649887. URL: <https://cacm.acm.org/research/50-years-of-queries> (visited on 2025-01-09) (cit. on pp. 9, 12–15, 89, 228, 230, 373, 374).
- [73] Donald D. Chamberlin, Morton M. Astrahan, Kapali P. Eswaran, Patricia P. Griffiths, Raymond A. Lorie, James W. Mehl, Phyllis Reisner, and Bradford W. Wade. "SEQUEL 2: A Unified Approach to Data Definition, Manipulation, and Control". *IBM Journal of Research and Development* 20(6):560–575, Nov. 1976. Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers (IEEE) and Armonk, NY, USA: International Business Machines Corporation (IBM). ISSN: 0018-8646. doi:10.1147/RD.206.0560 (cit. on p. 12).
- [74] Donald D. Chamberlin and Raymond F. Boyce. "SEQUEL: A Structured English Query Language". In: *1974 ACM SIGFIDET (now SIGMOD) Workshop on Data Description, Access and Control*. May 1–3, 1974, Ann Arbor, MI, USA. Ed. by Gene Altshuler, Randall Rustin, and Bernard D. Plagman. Vol. 1. New York, NY, USA: Association for Computing Machinery (ACM), 1974, pp. 249–264. ISBN: 978-1-4503-7415-6. doi:10.1145/800296.811515 (cit. on p. 12).

- [75] Lois Mai Chan and Joan S. Mitchell. *Dewey Decimal Classification: Principles and Application*. Dublin, OH, USA: Ohio College Library Center (OCLC), Jan. 2003. ISBN: 978-0-910608-72-5 (cit. on p. 8).
- [76] "Character Types". In: *PostgreSQL Documentation*. 17.4. The PostgreSQL Global Development Group (PGDG), Feb. 20, 2025. Chap. 8.3. URL: <https://www.postgresql.org/docs/17/datatype-character.html> (visited on 2025-02-27) (cit. on p. 102).
- [77] Bo Chen, Chris N. Potts, and Gerhard J. Woeginger. "A Review of Machine Scheduling: Complexity, Algorithms and Approximability". In: *Handbook of Combinatorial Optimization*. Ed. by Panos Miltiades Pardalos, Ding-Zhu Du, and Ronald Lewis Graham. 1st ed. Boston, MA, USA: Springer, 1998, pp. 1493–1641. ISBN: 978-1-4613-7987-4. doi:[10.1007/978-1-4613-0303-9_25](https://doi.org/10.1007/978-1-4613-0303-9_25). See also pages 21–169 in volume 3/3 by Norwell, MA, USA: Kluwer Academic Publishers. (Cit. on pp. 370, 372).
- [78] Peter Pin-Shan Chen. "English, Chinese and ER Diagrams". *Data & Knowledge Engineering (DKE)* 23(1):5–16, June 1997. Amsterdam, The Netherlands: Elsevier B.V. ISSN: 0169-023X. doi:[10.1016/S0169-023X\(97\)00017-7](https://doi.org/10.1016/S0169-023X(97)00017-7). URL: https://www.csc.lsu.edu/~chen/pdf/ER_C.pdf (visited on 2025-04-06) (cit. on pp. 13, 193, 194, 206).
- [79] Peter Pin-Shan Chen. "Entity-Relationship Modeling: Historical Events, Future Trends, and Lessons Learned". In: *Software Pioneers: Contributions to Software Engineering*. Ed. by Manfred Broy and Ernst Denert. Berlin/Heidelberg, Germany: Springer-Verlag GmbH Germany, Feb. 2002, pp. 296–310. doi:[10.1007/978-3-642-59412-0_17](https://doi.org/10.1007/978-3-642-59412-0_17). URL: http://bit.csc.lsu.edu/%7Echen/pdf/Chen_Pioneers.pdf (visited on 2025-03-06) (cit. on pp. 13, 195, 369).
- [80] Peter Pin-Shan Chen. "The Entity-Relationship Model – Toward a Unified View of Data". *ACM Transactions on Database Systems (TODS)* 1(1):9–36, Mar. 1976. New York, NY, USA: Association for Computing Machinery (ACM). ISSN: 0362-5915. doi:[10.1145/320434.320440](https://doi.org/10.1145/320434.320440) (cit. on pp. 12, 195, 212, 214, 369, 384).
- [81] Peter Pin-Shan Chen. "The Entity-Relationship Model: Toward a Unified View of Data". In: *1st International Conference on Very Large Data Bases (VLDB'1975)*. Sept. 22–24, 1975, Framingham, MA, USA. Ed. by Douglas S. Kerr. New York, NY, USA: Association for Computing Machinery (ACM), 1975, p. 173. ISBN: 978-1-4503-3920-9. doi:[10.1145/1282480.1282492](https://doi.org/10.1145/1282480.1282492). See [80] for a more comprehensive introduction. (Cit. on pp. 12, 13, 195, 369).
- [82] Stuart Cheshire and Marc Krochmal. *Special-Use Domain Names*. Request for Comments (RFC) 6761. Wilmington, DE, USA: Internet Engineering Task Force (IETF), Feb. 2013. URL: <https://www.ietf.org/rfc/rfc4253.txt> (visited on 2025-02-27) (cit. on p. 371).
- [83] Junghoo "John" Cho. *CS143: Data Management Systems*. Davis, CA, USA: University of California – Los Angeles (UCLA), Sept. 2016–Aut. 2021. URL: <http://oak.cs.ucla.edu/classes/cs143> (visited on 2025-04-04) (cit. on p. 16).
- [84] Raul F. Chong, Xiaomei Wang, Michael Dang, and Dwaine R. Snow. *Understanding DB2®: Learning Visually with Examples*. 2nd ed. Indianapolis, IN, USA: IBM Press, Dec. 2007. ISBN: 978-0-7686-8177-2 (cit. on pp. 13, 15, 18, 369).
- [85] Aakash Choudhury, Arjav Choudhury, Umashankar Subramanium, and S. Balamurugan. "Health-Saver: A Neural Network based Hospital Recommendation System Framework on Flask Webapplication with Realtime Database and RFID based Attendance System". *Journal of Ambient Intelligence and Humanized Computing* 13(10):4953–4966, Oct. 2022. London, England, UK: Springer Nature Limited. ISSN: 1868-5137. doi:[10.1007/S12652-021-03232-7](https://doi.org/10.1007/S12652-021-03232-7) (cit. on p. 369).
- [86] Christmas, FL, USA: Simon Sez IT. *Microsoft Access 2021 – Beginner to Advanced*. Birmingham, England, UK: Packt Publishing Ltd, Aug. 2023. ISBN: 978-1-83546-911-8 (cit. on pp. 15, 18, 45, 146, 371).
- [87] David Clinton and Christopher Negus. *Ubuntu Linux Bible*. 10th ed. Bible Series. Chichester, West Sussex, England, UK: John Wiley and Sons Ltd., Nov. 10, 2020. ISBN: 978-1-119-72233-5 (cit. on pp. 22, 375).

- [88] Edgar Frank "Ted" Codd. "Normalized Data Base Structure: A Brief Tutorial". In: *ACM SIGFIDET Workshop on Data Description, Access, and Control*. Nov. 11–12, 1971, San Diego, CA, USA. Ed. by Edgar Frank "Ted" Codd and Albert L. Dean Jr. New York, NY, USA: Association for Computing Machinery (ACM), 1971, pp. 1–17. ISBN: 978-1-4503-7300-5. doi:10.1145/1734714.1734716. See also [89] (cit. on pp. 342, 356, 368).
- [89] Edgar Frank "Ted" Codd. *Normalized Data Base Structure: A Brief Tutorial*. IBM Research Report RJ935. San Jose, CA, USA: IBM Research Laboratory, 1971. URL: [https://www.fsmwarden.com/Codd/Normalized%20data%20base%20structure_%20a%20brief%20tutorial\(1971,%20nov\).pdf](https://www.fsmwarden.com/Codd/Normalized%20data%20base%20structure_%20a%20brief%20tutorial(1971,%20nov).pdf) (visited on 2025-05-04) (cit. on p. 385).
- [90] Edgar Frank "Ted" Codd. "A Relational Model of Data for Large Shared Data Banks". *Communications of the ACM (CACM)* 13(6):377–387, June 1970. New York, NY, USA: Association for Computing Machinery (ACM). ISSN: 0001-0782. doi:10.1145/362384.362685. URL: <https://www.seas.upenn.edu/~zives/03f/cis550/codd.pdf> (visited on 2025-01-05) (cit. on pp. 3, 11, 12, 226, 228, 229, 330, 368, 374).
- [91] Edgar Frank "Ted" Codd. "An Evaluation Scheme for Database Management Systems that are claimed to be Relational". In: *Second International Conference on Data Engineering (ICDE'1986)*. Feb. 5–7, 1986, Los Angeles, CA, USA. Los Alamitos, CA, USA: IEEE Computer Society, 1986, pp. 720–729. ISBN: 978-0-8186-0655-7. doi:10.1109/ICDE.1986.7266284. URL: [https://www.fsmwarden.com/Codd/An%20evaluation%20scheme%20\(1986\).pdf](https://www.fsmwarden.com/Codd/An%20evaluation%20scheme%20(1986).pdf) (visited on 2025-04-10). Keynote Address. ©1985 CW Communications, Inc., excerpted from *Computerworld* [93] (cit. on pp. 230, 385).
- [92] Edgar Frank "Ted" Codd. *Further Normalization of the Data Base Relational Model*. IBM Research Report RJ909. San Jose, CA, USA: IBM Research Laboratory, Aug. 31, 1971. URL: <https://forum.thethirdmanifesto.com/wp-content/uploads/asgarosforum/987737/00-efc-further-normalization.pdf> (visited on 2025-05-04). Reprinted in and presented at [372] (cit. on pp. 342, 356, 368).
- [93] Edgar Frank "Ted" Codd. "Is your DBMS really relational? (Part 1)". *Computerworld* 19:ID1, Oct. 14, 1985. Framingham, MA, USA: CW Communications, Inc. and Needham, MA, USA: Foundry (formerly IDG Communications, Inc.) ISSN: 0010-4841. URL: <https://thaumatorium.com/articles/the-papers-of-ef-the-coddfather-codd/1985a-is-your-dbms-really-relational> (visited on 2025-04-10). See also the Keynote Address [91] (cit. on pp. 230, 385).
- [94] Coding Gears and Train Your Brain. *YAML Fundamentals for DevOps, Cloud and IaC Engineers*. Birmingham, England, UK: Packt Publishing Ltd, Mar. 2022. ISBN: 978-1-80324-243-9 (cit. on pp. 2, 376).
- [95] Timothy W. Cole and Myung-Ja K. Han. *XML for Catalogers and Metadata Librarians (Third Millennium Cataloging)*. 1st ed. Dublin, OH, USA: Libraries Unlimited, May 23, 2013. ISBN: 978-1-59884-519-8 (cit. on pp. 2, 376).
- [96] "Combining Queries (`UNION`, `INTERSECT`, `EXCEPT`)". In: *PostgreSQL Documentation*. 17.4. The PostgreSQL Global Development Group (PGDG), Feb. 20, 2025. Chap. 7.4. URL: <https://www.postgresql.org/docs/17/queries-union.html> (visited on 2025-02-27) (cit. on p. 113).
- [97] "Conditional Expressions". In: *PostgreSQL Documentation*. 17.4. The PostgreSQL Global Development Group (PGDG), Feb. 20, 2025. Chap. 9.18. URL: <https://www.postgresql.org/docs/17/functions-conditional.html> (visited on 2025-05-06) (cit. on p. 334).
- [98] "Connection URLs". In: *PostgreSQL Documentation*. 17.4. The PostgreSQL Global Development Group (PGDG), Feb. 20, 2025. Chap. 32.1.1.2. URL: <https://www.postgresql.org/docs/17/libpq-connect.html#LIBPQ-CONNSTRING-URIS> (visited on 2025-02-25) (cit. on p. 90).
- [99] *Connections: Local Networks*. Mountain View, CA, USA: Computer History Museum (CHM), 1996–2025. URL: <https://www.computerhistory.org/revolution/networking/19/381> (visited on 2025-01-21) (cit. on p. 12).
- [100] "Constraints". In: *PostgreSQL Documentation*. 17.4. The PostgreSQL Global Development Group (PGDG), Feb. 20, 2025. Chap. 5.5. URL: <https://www.postgresql.org/docs/17/ddl-constraints.html> (visited on 2025-02-28) (cit. on pp. 102, 104, 114, 125, 259).

- [101] Stephen Arthur Cook. "The Complexity of Theorem-Proving Procedures". In: *Third Annual ACM Symposium on Theory of Computing (STOC'1971)*. May 3–5, 1971, Shaker Heights, OH, USA. Ed. by Michael A. Harrison, Ranan B. Banerji, and Jeffrey D. Ullman. New York, NY, USA: Association for Computing Machinery (ACM), 1971, pp. 151–158. ISBN: 978-1-4503-7464-4. doi:10.1145/800157.805047 (cit. on p. 372).
- [102] Fernando J. Corbató, Marjorie Merwin-Daggett, Robert C. Daley, R. J. Creasy, J. D. Hellwig, Richard H. Orenstein, and L. K. Korn. *The Compatible Time-Sharing System: A Programmer's Guide*. Cambridge, MA, USA: Massachusetts Institute of Technology (MIT) Computation Center and MIT Press, 1963–1964. URL: https://www.ibiblio.org/apollo/Documents/CTSS_ProgrammersGuide.pdf (visited on 2025-01-08). Second Printing 5, 1964 (cit. on p. 9).
- [103] Fernando J. Corbató and Victor A. Vyssotsky. "Introduction and Overview of the Multics System". In: *1965 Fall Joint Computer Conference (AFIPS'1965, Fall, Part 1)*. Nov. 30–Dec. 1, 1965, Las Vegas, NV, USA. Ed. by Robert W. Rector. New York, NY, USA: Association for Computing Machinery (ACM), 1965, pp. 185–196. ISBN: 978-1-4503-7885-7. doi:10.1145/1463891.1463912. URL: <https://www.multicians.org/fjcc1.html> (visited on 2025-01-08) (cit. on p. 10).
- [104] Carlos Coronel and Steven Morris. *Database Systems: Design, Implementation, & Management*. 13th ed. Boston, MA, USA: Cengage Learning, Jan. 2018. ISBN: 978-1-337-62790-0 (cit. on p. 17).
- [105] *Country Codes*. Tech. rep. ISO 3166. ISO 3166 Maintenance Agenc, c/o International Organization for Standardization (ISO): ISO 3166 Maintenance Agenc, c/o International Organization for Standardization (ISO), 2020. URL: <https://www.iso.org/iso-3166-country-codes.html> (visited on 2025-04-11) (cit. on p. 226).
- [106] "[CREATE SEQUENCE](#)". In: *PostgreSQL Documentation*. 17.4. The PostgreSQL Global Development Group (PGDG), Feb. 20, 2025. URL: <https://www.postgresql.org/docs/current/sql-createsequence.html> (visited on 2025-04-22) (cit. on pp. 284, 306).
- [107] "[CREATE TABLE](#)". In: *PostgreSQL Documentation*. 17.4. The PostgreSQL Global Development Group (PGDG), Feb. 20, 2025. URL: <https://www.postgresql.org/docs/17/sql-createtable.html> (visited on 2025-04-21) (cit. on pp. 102, 278).
- [108] "[CREATE VIEW](#)". In: *PostgreSQL Documentation*. 17.4. The PostgreSQL Global Development Group (PGDG), Feb. 20, 2025. Chap. Part VI. Reference. URL: <https://www.postgresql.org/docs/17/sql-createview.html> (visited on 2025-03-03) (cit. on p. 131).
- [109] Ignacio Samuel Crespo-Martínez, Adrián Campazas Vega, Ángel Manuel Guerrero-Higueras, Virginia Riego-Del Castillo, Claudia Álvarez-Aparicio, and Camino Fernández Llamas. "SQL Injection Attack Detection in Network Flow Data". *Computers & Security* 127:103093, Apr. 2023. Amsterdam, The Netherlands: Elsevier B.V. ISSN: 0167-4048. doi:10.1016/J.COSE.2023.103093 (cit. on pp. 143, 374).
- [110] "Crow's Foot Notation – Relationship Symbols and How to Read Diagrams". In: [#DATABASE](#). Oakland, CA, USA: Free Code Camp, Inc., June 6, 2022. URL: <https://www.freecodecamp.org/news/crows-foot-notation-relationship-symbols-and-how-to-read-diagrams> (visited on 2025-04-06) (cit. on p. 216).
- [111] "[csv](#) – CSV File Reading and Writing". In: *Python 3 Documentation. The Python Standard Library*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. URL: <https://docs.python.org/3/library/csv.html> (visited on 2024-11-14) (cit. on p. 368).
- [112] *Cuneiform Tablets: From the Reign of Gudea of Lagash to Shalmanassar III*. Washington, D.C., USA: Library of Congress. URL: <https://www.loc.gov/collections/cuneiform-tablets> (visited on 2025-01-08). See also [250, 456] (cit. on pp. 8, 396, 409).
- [113] Erik Dahlström, Patrick Dengler, Anthony Grasso, Chris Lilley, Cameron McCormack, Doug Schepers, Jonathan Watt, Jon Ferraiolo, Jun Fujisawa, and Dean Jackson, eds. *Scalable Vector Graphics (SVG) 1.1 (Second Edition)*. W3C Recommendation. Wakefield, MA, USA: World Wide Web Consortium (W3C), Aug. 16, 2011. URL: <http://www.w3.org/TR/2011/REC-SVG11-20110816> (visited on 2024-12-17) (cit. on p. 375).

- [114] Robert C. Daley and Peter G. Neumann. "A General-Purpose File System for Secondary Storage". In: *1965 Fall Joint Computer Conference (AFIPS'1965, Fall, Part 1)*. Nov. 30–Dec. 1, 1965, Las Vegas, NV, USA. Ed. by Robert W. Rector. New York, NY, USA: Association for Computing Machinery (ACM), 1965, pp. 213–229. ISBN: 978-1-4503-7885-7. doi:10 . 1145 / 1463891 . 1463915. URL: <https://www.multicians.org/fjcc4.html> (visited on 2025-01-08) (cit. on p. 10).
- [115] Daniela E. Damian, James Chisan, Lakshminarayanan Vaidyanathasamy, and Yogendra Pal. "Requirements Engineering and Downstream Software Development: Findings from a Case Study". *Empirical Software Engineering: An International Journal* 10(3):255–283, July 2005. London, England, UK: Springer Nature Limited. ISSN: 1382-3256. doi:10.1007/S10664-005-1288-4. URL: <https://www.researchgate.net/publication/220277938> (visited on 2025-03-27) (cit. on p. 187).
- [116] "Data Type Formatting Functions". In: *PostgreSQL Documentation*. 17.4. The PostgreSQL Global Development Group (PGDG), Feb. 20, 2025. Chap. Part II.9.8. URL: <https://www.postgresql.org/docs/17/functions-formatting.html> (visited on 2025-11-04) (cit. on p. 242).
- [117] *Database Language SQL*. Tech. rep. ANSI X3.135-1986. Washington, D.C., USA: American National Standards Institute (ANSI), 1986 (cit. on pp. 13, 89, 93, 374).
- [118] Christopher J. Date. *An Introduction to Database Systems*. 8th ed. Hoboken, NJ, USA: Pearson Education, Inc., July 2003. ISBN: 978-0-321-19784-9 (cit. on pp. 17, 330, 342, 356, 368, 372).
- [119] *Date and Time – Representations for Information Interchange – Part 1: Basic Rules*. International Standard ISO 8601-1:2019(E), Edition 1. Geneva, Switzerland: International Organization for Standardization (ISO), Feb. 2019 (cit. on pp. 122, 163).
- [120] "Date/Time Functions and Operators". In: *PostgreSQL Documentation*. 17.4. The PostgreSQL Global Development Group (PGDG), Feb. 20, 2025. Chap. 9.9. URL: <https://www.postgresql.org/docs/17/functions-datetime.html> (visited on 2025-05-01) (cit. on pp. 317, 322, 375).
- [121] "Date/Time Types". In: *PostgreSQL Documentation*. 17.4. The PostgreSQL Global Development Group (PGDG), Feb. 20, 2025. Chap. 8.5. URL: <https://www.postgresql.org/docs/17/datatype-datetime.html> (visited on 2025-02-28) (cit. on p. 375).
- [122] "Dates". In: *Cambridge Dictionary English (UK)*. Cambridge, England, UK: Cambridge University Press & Assessment, 2025. URL: <https://dictionary.cambridge.org/grammar/british-grammar/dates> (visited on 2025-01-21) (cit. on p. 371).
- [123] Matt David and Blake Barnhill. *How to Teach People SQL*. San Francisco, CA, USA: The Data School, Chart.io, Inc., Dec. 10, 2019–Apr. 10, 2023. URL: <https://dataschool.com/how-to-teach-people-sql> (visited on 2025-02-27) (cit. on pp. 89, 93, 374).
- [124] Matt David and Blake Barnhill. "Syntax Conventions". In: *How to Teach People SQL*. San Francisco, CA, USA: The Data School, Chart.io, Inc., July 28, 2020. URL: <https://dataschool.com/how-to-teach-people-sql/syntax-conventions> (visited on 2025-02-27) (cit. on pp. 94, 365).
- [125] Donald W. Davies. *Proposal for a Digital Communication Network*. Tech. rep. London, England, UK: National Physical Laboratory (NPL), June 1966. URL: <https://www.dcs.gla.ac.uk/~wpc/grcs/Davies05.pdf> (visited on 2025-01-21) (cit. on p. 12).
- [126] Donald W. Davies. *Proposal for the Development of a National Communications Service for On-Line Data Processing*. Tech. rep. London, England, UK: National Physical Laboratory (NPL), Dec. 8, 1965 (cit. on p. 12).
- [127] Donald W. Davies. *Remote On-line Data Processing and its Communication Needs*. Tech. rep. London, England, UK: National Physical Laboratory (NPL), Nov. 10, 1965 (cit. on p. 12).
- [128] Alan M. Davis, Óscar Dieste Tubío, Ann M. Hickey, Natalia Juristo Juzgado, and Ana María Moreno. "Effectiveness of Requirements Elicitation Techniques: Empirical Results Derived from a Systematic Review". In: *14th IEEE International Conference on Requirements Engineering (RE'2006)*. Sept. 11–15, 2006, Minneapolis/St. Paul, MN, USA. Los Alamitos, CA, USA: IEEE Computer Society, 2006, pp. 176–185. ISSN: 1090-705X. ISBN: 978-0-7695-2555-6. doi:10.1109/RE.2006.17 (cit. on p. 188).

- [129] *DB-Engines Ranking of Relational DBMS* (June 2025). Cambridge, England, UK: Red Gate Software Ltd., June 2025. URL: <https://db-engines.com/en/ranking/relational+dbms> (visited on 2025-06-01) (cit. on pp. 13, 15).
- [130] *Database Administrators*. New York, NY, USA: Stack Exchange Inc. URL: <https://dba.stackexchange.com> (visited on 2025-02-27) (cit. on p. 18).
- [131] *Database Language SQL*. International Standard ISO 9075-1987. Geneva, Switzerland: International Organization for Standardization (ISO), 1987 (cit. on pp. 13, 89, 93, 374).
- [132] “Default Values”. In: *PostgreSQL Documentation*. 17.4. The PostgreSQL Global Development Group (PGDG), Feb. 20, 2025. Chap. 5.2. URL: <https://www.postgresql.org/docs/current/ddl-default.html> (visited on 2025-04-23) (cit. on p. 284).
- [133] *Definition of Operations Research*. University of Western Ontario, London, ON, Canada: International Federation of Operational Research Societies (IFORS), 2020. URL: <https://www.ifors.org/what-is-or> (visited on 2026-01-01) (cit. on p. 372).
- [134] Paul Deitel, Harvey Deitel, and Abbey Deitel. *Internet & World Wide Web: How to Program*. 5th ed. Hoboken, NJ, USA: Pearson Education, Inc., Nov. 2011. ISBN: 978-0-13-299045-5 (cit. on p. 376).
- [135] “`DELETE`”. In: *PostgreSQL Documentation*. 17.4. The PostgreSQL Global Development Group (PGDG), Feb. 20, 2025. Chap. Part VI. Reference. URL: <https://www.postgresql.org/docs/17/sql-delete.html> (visited on 2025-03-07) (cit. on p. 136).
- [136] “`DELETE`”. In: *MariaDB Server Documentation*. Milpitas, CA, USA: MariaDB, 2025. URL: <https://mariadb.com/docs/server/reference/sql-statements/data-manipulation-changing-deleting-data/delete> (visited on 2025-07-06) (cit. on pp. 136, 137).
- [137] Justin Dennison, Cherokee Boose, and Peter van Rysdam. *Intro to NumPy*. Centennial, CO, USA: ACI Learning. Birmingham, England, UK: Packt Publishing Ltd, June 2024. ISBN: 978-1-83620-863-1 (cit. on pp. 139, 372).
- [138] Adam “djeada” Djellouli. *Database Notes*. Berlin, Germany, Feb. 2022–Mar. 2025. URL: https://adamdjellouli.com/articles/databases_notes (visited on 2025-03-27) (cit. on p. 18).
- [139] Adam “djeada” Djellouli. “Database Requirements Analysis”. In: *Database Notes*. Berlin, Germany, Mar. 11, 2025. URL: https://adamdjellouli.com/articles/databases_notes/02_database_design/01_requirements_analysis (visited on 2025-03-27) (cit. on p. 185).
- [140] Slobodan Dmitrović. *Modern C for Absolute Beginners: A Friendly Introduction to the C Programming Language*. New York, NY, USA: Apress Media, LLC, Mar. 2024. ISBN: 979-8-8688-0224-9 (cit. on p. 368).
- [141] Pooyan Doozandeh and Frank E. Ritter. “Some Tips for Academic Writing and Using Microsoft Word”. *XRDS: Crossroads, The ACM Magazine for Students* 26(1):10–11, Aut. 2019. New York, NY, USA: Association for Computing Machinery (ACM). ISSN: 1528-4972. doi:10.1145/3351470 (cit. on p. 372).
- [142] Ingry döt Net, Tina Müller, Pantelis Antoniou, Eemeli Aro, Thomas Smith, Oren Ben-Kiki, and Clark C. Evans. *YAML Ain’t Markup Language (YAML™) version 1.2*. Revision 1.2.2. Seattle, WA, USA: YAML Language Development Team, Oct. 1, 2021. URL: <https://yaml.org/spec/1.2.2> (visited on 2025-01-05) (cit. on pp. 2, 376).
- [143] “`DROP DATABASE`”. In: *PostgreSQL Documentation*. 17.4. The PostgreSQL Global Development Group (PGDG), Feb. 20, 2025. Chap. Part VI. Reference. URL: <https://www.postgresql.org/docs/17/sql-dropdatabase.html> (visited on 2025-03-05) (cit. on pp. 171, 173).
- [144] “`DROP TABLE`”. In: *PostgreSQL Documentation*. 17.4. The PostgreSQL Global Development Group (PGDG), Feb. 20, 2025. Chap. Part VI. Reference. URL: <https://www.postgresql.org/docs/17/sql-droptable.html> (visited on 2025-03-05) (cit. on pp. 171, 173).
- [145] “`DROP USER`”. In: *PostgreSQL Documentation*. 17.4. The PostgreSQL Global Development Group (PGDG), Feb. 20, 2025. Chap. Part VI. Reference. URL: <https://www.postgresql.org/docs/17/sql-dropuser.html> (visited on 2025-03-05) (cit. on pp. 171, 173).

- [146] “[DROP VIEW](#)”. In: *PostgreSQL Documentation*. 17.4. The PostgreSQL Global Development Group (PGDG), Feb. 20, 2025. Chap. Part VI. Reference. URL: <https://www.postgresql.org/docs/17/sql-dropview.html> (visited on 2025-03-05) (cit. on pp. 171, 172).
- [147] Paul Duplys and Roland Schmitz. *TLS Cryptography In-Depth*. Birmingham, England, UK: Packt Publishing Ltd, Jan. 2024. ISBN: 978-1-80461-195-1 (cit. on p. 375).
- [148] Jacques Dutka. “The Early History of the Factorial Function”. *Archive for History of Exact Sciences* 43(3):225–249, Sept. 1991. Berlin/Heidelberg, Germany: Springer-Verlag GmbH Germany. ISSN: 0003-9519. doi:10.1007/BF00389433. Communicated by Umberto Bottazzini (cit. on p. 368).
- [149] Russell J.T. Dyer. *Learning MySQL and MariaDB*. Sebastopol, CA, USA: O'Reilly Media, Inc., Mar. 2015. ISBN: 978-1-4493-6290-4 (cit. on pp. 14, 371, 372).
- [150] Donald E. Eastlake 3rd and Aliza R. Panitz. *Reserved Top Level DNS Names*. Request for Comments (RFC) 2606. Wilmington, DE, USA: Internet Engineering Task Force (IETF), June 1999. URL: <https://www.ietf.org/rfc/rfc2606.txt> (visited on 2025-02-27) (cit. on p. 371).
- [151] Christof Ebert and Jozef De Man. “Requirements Uncertainty: Influencing Factors and Concrete Improvements”. In: *27th International Conference on Software Engineering (ICSE'2005)*. May 15–21, 2005, St. Louis, MO, USA. Ed. by Gruia-Catalin Roman, William G. Griswold, and Bashar Nuseibeh. Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers (IEEE) and New York, NY, USA: Association for Computing Machinery (ACM), 2005, pp. 553–560. ISSN: 0270-5257. ISBN: 978-1-58113-963-1. doi:10.1109/ICSE.2005.1553601 (cit. on p. 187).
- [152] Phillip J. Eby. *Python Web Server Gateway Interface v1.0.1*. Python Enhancement Proposal (PEP) 3333. Beaverton, OR, USA: Python Software Foundation (PSF), Sept. 26–Oct. 4, 2010. URL: <https://peps.python.org/pep-3333> (visited on 2025-03-04) (cit. on p. 369).
- [153] *ECMAScript Language Specification*. Standard ECMA-262, 3rd Edition. Geneva, Switzerland: Ecma International, Dec. 1999. URL: https://ecma-international.org/wp-content/uploads/ECMA-262_3rd_edition_december_1999.pdf (visited on 2024-12-15) (cit. on p. 370).
- [154] “Common and Proper Nouns: What's the difference? Learn which ones get capitals.” In: *Merriam-Webster: America's Most Trusted Dictionary*. Ed. by Editors of Merriam-Webster. Apr. 6–May 12, 2023. URL: <https://www.merriam-webster.com/grammar/common-and-proper-nouns-whats-the-difference> (visited on 2025-03-29) (cit. on pp. 193, 194).
- [155] Editors of Merriam-Webster, ed. *Merriam-Webster: America's Most Trusted Dictionary*. 2005. URL: <https://www.merriam-webster.com> (visited on 2025-03-29).
- [156] “transitive (adjective)”. In: *Merriam-Webster: America's Most Trusted Dictionary*. Ed. by Editors of Merriam-Webster. Feb. 20, 2025. URL: <https://www.merriam-webster.com/dictionary/transitive> (visited on 2025-04-06) (cit. on p. 206).
- [157] Ramez Elmasri and Shamkant Navathe. *Fundamentals of Database Systems*. 7th ed. Hoboken, NJ, USA: Pearson Education, Inc., June 2015. ISBN: 978-0-13-397077-7 (cit. on pp. 16, 177, 184, 185, 330, 342, 356, 368, 372).
- [158] “Escape Sequences”. In: *Python 3 Documentation. The Python Language Reference*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. Chap. 2.4.1.1. URL: https://docs.python.org/3/reference/lexical_analysis.html#escape-sequences (visited on 2025-08-05) (cit. on p. 369).
- [159] Gordon C. Everest. “Basic Data Structure Models Explained with a Common Example”. In: *Fifth Texas Conference on Computing Systems (Computing Systems'1976)*. Oct. 18–19, 1976, Austin, TX, USA. Long Beach, CA, USA: IEEE Computer Society Publications Office, 1976, pp. 39–46. ISSN: 0730-8310. URL: <https://www.researchgate.net/publication/291448084> (visited on 2025-04-04) (cit. on pp. 195, 213, 214).

- [160] Steve Fanning, Vasudev Narayanan, “*flywire*”, Olivier Hallot, Jean Hollis Weber, Jenna Sargent, Pulkit Krishna, Dan Lewis, Peter Schofield, Jochen Schiffers, Robert Großkopf, Jost Lange, Martin Fox, Hazel Russman, Steve Schwettman, Alain Romedenne, Andrew Pitonyak, Jean-Pierre Ledure, Drew Jensen, and Randolph Gam. *Base Guide 7.3. Revision 1. Based on LibreOffice 7.3 Community*. Berlin, Germany: The Document Foundation, Aug. 2022. URL: <https://books.libreoffice.org/en/BG73/BG73-BaseGuide.pdf> (visited on 2025-01-13) (cit. on pp. iv, 15, 20, 45, 146, 170, 367, 371).
- [161] Luca Ferrari and Enrico Pirozzi. *Learn PostgreSQL*. 2nd ed. Birmingham, England, UK: Packt Publishing Ltd, Oct. 2023. ISBN: 978-1-83763-564-1 (cit. on pp. iv, 14, 17, 20, 25, 88, 367, 369, 373).
- [162] Roy T. Fielding, Jim Gettys, Jeffrey C. Mogul, Henrik Frystyk Nielsen, and Tim Berners-Lee. *Hypertext Transfer Protocol -- HTTP/1.1*. Request for Comments (RFC) 2068. Wilmington, DE, USA: Internet Engineering Task Force (IETF), Jan. 1997. URL: <https://www.ietf.org/rfc/rfc2068.txt> (visited on 2025-02-05) (cit. on p. 370).
- [163] Roy T. Fielding, Mark Nottingham, and Julian F. Reschke. *HTTP Semantics*. Request for Comments (RFC) 9110. Wilmington, DE, USA: Internet Engineering Task Force (IETF), June 2022. URL: <https://www.ietf.org/rfc/rfc9110.txt> (visited on 2025-02-05) (cit. on p. 370).
- [164] Eric Fischer. “A Brief History of the ‘ls’ Command”. *Linux Gazette*, Dec. 1999. Seattle, WA, USA: Specialized Systems Consultants, Inc. URL: <https://www.linuxdoc.org/LDP/LG/issue48/fischer.html> (visited on 2025-05-30) (cit. on p. 10).
- [165] Christiane Floyd, Wolf-Michael Mehl, Fanny-Michaela Reisin, Gerhard Schmidt, and Gregor Wolf. “Out of Scandinavia: Alternative Approaches to Software Design and System Development”. *Human-Computer Interaction* 4(4):253–350, 1989. London, England, UK: Taylor and Francis Ltd. ISSN: 0737-0024. doi:10.1207/S15327051HCI0404_1 (cit. on pp. 188, 373).
- [166] Keith D. Foote. *A Brief History of Database Management*. Studio City, CA, USA: Dataversity Digital LLC, Oct. 25, 2021. URL: <https://www.dataversity.net/brief-history-database-management> (visited on 2025-01-11) (cit. on p. 8).
- [167] “Foreign Keys”. In: *PostgreSQL Documentation*. 17.4. The PostgreSQL Global Development Group (PGDG), Feb. 20, 2025. Chap. 5.5.5. URL: <https://www.postgresql.org/docs/17/ddl-constraints.html#DDL-CONSTRAINTS-FK> (visited on 2025-02-28) (cit. on pp. 121, 125).
- [168] *PDF 32000-1:2008 – Document Management – Portable Document Format – Part 1: PDF 1.7*. 1st ed. San Jose, CA, USA: Adobe Systems Incorporated, July 1, 2008. URL: https://pdf-lib.js.org/assets/with_large_page_count.pdf (visited on 2024-12-12) (cit. on p. 373).
- [169] “Formatted String Literals”. In: *Python 3 Documentation. The Python Tutorial*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. Chap. 7.1.1. URL: <https://docs.python.org/3/tutorial/inputoutput.html#formatted-string-literals> (visited on 2024-07-25) (cit. on p. 369).
- [170] Kevin P. Gaffney, Martin Prammer, Laurence C. Brasfield, D. Richard Hipp, Dan R. Kennedy, and Jignesh M. Patel. “*SQLite: Past, Present, and Future*”. *Proceedings of the VLDB Endowment (PVLDB)* 15(12):3535–3547, Aug. 2022. Irvine, CA, USA: Very Large Data Bases Endowment Inc. ISSN: 2150-8097. doi:10.14778/3554821.3554842. URL: <https://www.vldb.org/pvldb/vol15/p3535-gaffney.pdf> (visited on 2025-01-12). All papers in this issue were presented at the *48th International Conference on Very Large Data Bases (VLDB 2022)*, 9–15, 2022, hybrid/Sydney, NSW, Australia (cit. on pp. 15, 374).
- [171] Jonas Gamalielsson and Björn Lundell. “Long-Term Sustainability of Open Source Software Communities beyond a Fork: A Case Study of LibreOffice”. In: *8th IFIP WG 2.13 International Conference on Open Source Systems: Long-Term Sustainability OSS’2012*. Sept. 10–13, 2012, Hammamet, Tunisia. Ed. by Imed Hammouda, Björn Lundell, Tommi Mikkonen, and Walt Scacchi. Vol. 378. Vol. 378 of *IFIP Advances in Information and Communication Technology (IFIP AICT)*. Berlin/Heidelberg, Germany: Springer-Verlag GmbH Germany, 2012, pp. 29–47. ISSN: 1868-4238. ISBN: 978-3-642-33441-2. doi:10.1007/978-3-642-33442-9_3 (cit. on pp. 15, 45, 101, 371).

- [172] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database Systems: The Complete Book*. 2nd ed. Hoboken, NJ, USA: Pearson Education, Inc., May 2008. ISBN: 978-0-13-187325-4 (cit. on p. 17).
- [173] Aakanksha Gaur, Gloria Lotha, Tara Ramanathan, Erik Gregersen, Emily Rodriguez, Anthony Lin, Parul Jain, and William L. Hosch. *List of Windows Versions*. Ed. by The Editors of Encyclopaedia Britannica. Chicago, IL, USA: Encyclopædia Britannica, Inc., Jan. 15, 2009–Feb. 19, 2025. URL: <https://www.britannica.com/technology/list-of-Windows-versions> (visited on 2024-12-14) (cit. on p. 179).
- [174] Bhavesh Gawade. “Mastering F-Strings in Python: Efficient String Handling in Python Using Smart F-Strings”. In: *C O D E B*. Mumbai, Maharashtra, India: Code B Solutions Pvt Ltd, Apr. 25–June 3, 2025. URL: <https://code-b.dev/blog/f-strings-in-python> (visited on 2025-08-04) (cit. on p. 369).
- [175] “Generated Columns”. In: *PostgreSQL Documentation*. 17.4. The PostgreSQL Global Development Group (PGDG), Feb. 20, 2025. Chap. 5.4. URL: <https://www.postgresql.org/docs/17/ddl-generated-columns.html> (visited on 2025-04-23) (cit. on pp. 106, 284, 322).
- [176] Sebastian Gerstl. “IBM 350 – Ein Kühlenschrankmonster mit unvorstellbarer Datenmenge: Die Festplatte wird 60”. In: *elektronikpraxis.de*. Würzburg, Bayern, Germany: Vogel Communications Group GmbH & Co. KG, Sept. 13, 2016. URL: <https://www.elektronikpraxis.de/ein-kuehlschrankmonster-mit-unvorstellbarer-datenmenge-die-festplatte-wird-60-a-549932> (visited on 2025-05-29) (cit. on p. 10).
- [177] Michael Gertz and Bertram Ludäscher. “Introduction to Relational Databases”. In: *ECS 165A Winter 2011 – Introduction to Database Systems*. Ed. by Todd J. Green. Davis, CA, USA: University of California, Davis, Win. 2011. Chap. 1. URL: <https://web.cs.ucdavis.edu/~green/courses/ecs165a-w11/1-intro.pdf> (visited on 2025-03-25) (cit. on p. 184).
- [178] Gleek.io. Prague, Czech Republic: Blocshop s.r.o., 2024. URL: <https://www.gleek.io/blog> (visited on 2025-04-05).
- [179] Alan Goldfine and Patricia Konig. *A Technical Overview of the Information Resource Dictionary System*. Tech. rep. NBSIR 85-3164. Gaithersburg, MD, USA: United States Department of Commerce, National Bureau of Standards, Center for Programming Science and Technology, Institute for Computer Sciences and Technology, Apr. 1985. URL: <https://nvlpubs.nist.gov/nistpubs/Legacy/IR/nbsir85-3164.pdf> (visited on 2025-03-29) (cit. on p. 12).
- [180] Michael T. Goodrich. *A Gentle Introduction to NP-Completeness*. Irvine, CA, USA: University of California, Irvine, Apr. 2022. URL: <https://ics.uci.edu/~goodrich/teach/cs165/notes/NPComplete.pdf> (visited on 2025-08-01) (cit. on p. 372).
- [181] Michael Goodwin. *What is an API?* Armonk, NY, USA: International Business Machines Corporation (IBM), Apr. 9, 2024. URL: <https://www.ibm.com/topics/api> (visited on 2024-12-12) (cit. on p. 368).
- [182] Olaf Górski. “Why f-strings are awesome: Performance of different string concatenation methods in Python”. In: *DEV Community*. Sacramento, CA, USA: DEV Community Inc., Nov. 8, 2022. URL: <https://dev.to/grski/performance-of-different-string-concatenation-methods-in-python-why-f-strings-are-awesome-2e97> (visited on 2025-08-04) (cit. on p. 369).
- [183] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. The Morgan Kaufmann Series in Data Management Systems. Burlington, MA, USA/San Mateo, CA, USA: Morgan Kaufmann Publishers, Sept. 1992. ISBN: 978-1-55860-190-1 (cit. on p. 7).
- [184] Todd J. Green. “Conceptual Modeling using the Entity-Relationship Model”. In: *ECS 165A Winter 2011 – Introduction to Database Systems*. Ed. by Todd J. Green. Davis, CA, USA: University of California, Davis, Win. 2011. Chap. 2. URL: <https://web.cs.ucdavis.edu/~green/courses/ecs165a-w11/2-er.pdf> (visited on 2025-03-27) (cit. on pp. 193, 203, 205, 206, 214).
- [185] Todd J. Green, ed. *ECS 165A Winter 2011 – Introduction to Database Systems*. Davis, CA, USA: University of California, Davis, Win. 2011. URL: <https://web.cs.ucdavis.edu/~green/courses/ecs165a-w11> (visited on 2025-03-25) (cit. on p. 16).

- [186] Todd J. Green, ed. *ECS 165B Spring 2011 – Database System Implementation*. Davis, CA, USA: University of California, Davis, Spr. 2011. URL: <https://web.cs.ucdavis.edu/~green/courses/ecs165b-s11> (visited on 2025-04-04) (cit. on p. 16).
- [187] Dawn Griffiths. *Excel Cookbook – Receipts for Mastering Microsoft Excel*. Sebastopol, CA, USA: O'Reilly Media, Inc., May 2024. ISBN: 978-1-0981-4332-9 (cit. on pp. 2, 372).
- [188] Ilya Grigorik. *HTTP Protocols*. Sebastopol, CA, USA: O'Reilly Media, Inc., Dec. 2017. ISBN: 978-1-4920-3046-1 (cit. on p. 370).
- [189] Christian Grün. "Pushing XML Main Memory Databases to their Limits". In: *Tagungsband zum 18. GI-Workshop über Grundlagen von Datenbanken. 18th GI-Workshop on the Foundations of Databases*. June 6–9, 2006, Wittenberg, Sachsen-Anhalt, Germany. Ed. by Stefan Brass and Alexander Hinneburg. 2006, pp. 60–64. URL: https://dbs.informatik.uni-halle.de/GvD2006/gvd06_gruen.pdf (cit. on p. 226).
- [190] Christian Grün, Alexander Holupirek, and Michael Seiferle. *BaseX*. Konstanz, Baden-Württemberg, Germany: BaseX GmbH, Jan. 2010–Mar. 2025. URL: <https://basex.org> (visited on 2025-04-09) (cit. on p. 226).
- [191] Pranshu Gupta, Ramon A. Mata-Toledo, and Morgan D. Monger. "Database Development Life Cycle". *Journal of Information Systems and Operations Management (JISOM)* 5(1):8–17, May 2011. Bucharest (Bucureşti), Romania: Romanian-American University (RAU), Scientific Research Department. ISSN: 1843-4711. URL: <http://www.rebe.rau.ro/RePEc/rau/jisomg/SP11/JISOM-SP11-A1.pdf> (visited on 2025-03-20) (cit. on pp. 179–183, 185).
- [192] Donald J. Haderle and Cynthia M. Saracco. "The History and Growth of IBM's DB2". *IEEE Annals of the History of Computing* 35(2):54–66, Apr. 2013–June 2014. Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers (IEEE). ISSN: 1058-6180. doi:10.1109/MAHC.2012.55 (cit. on pp. 13, 15, 369).
- [193] Thomas Haigh. "How Charles Bachman Invented the DBMS, A Foundation of Our Digital World". *Communications of the ACM (CACM)* 59(7):25–30, July 2016. New York, NY, USA: Association for Computing Machinery (ACM). ISSN: 0001-0782. doi:10.1145/2935880. URL: <https://cacm.acm.org/opinion/how-charles-bachman-invented-the-dbms-a-foundation-of-our-digital-world> (visited on 2025-05-08) (cit. on pp. 10, 11, 226).
- [194] Terry Halpin and Tony Morgan. *Information Modeling and Relational Databases*. 3rd ed. Burlington, MA, USA/San Mateo, CA, USA: Morgan Kaufmann Publishers, July 2024. ISBN: 978-0-443-23791-1 (cit. on pp. 3, 17, 374).
- [195] Jan L. Harrington. *Relational Database Design and Implementation*. 4th ed. Burlington, MA, USA/San Mateo, CA, USA: Morgan Kaufmann Publishers, Apr. 2016. ISBN: 978-0-12-849902-3 (cit. on pp. 3, 17, 374).
- [196] Charles R. Harris, K. Jarrod Millman, Stéfan van der Walt, Ralf Gommers, Pauli "pv" Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. "Array programming with NumPy". *Nature* 585:357–362, 2020. London, England, UK: Springer Nature Limited. ISSN: 0028-0836. doi:10.1038/S41586-020-2649-2 (cit. on pp. 139, 372).
- [197] Peter Haumer, Klaus Pohl, and Klaus Weidenhaupt. "Requirements Elicitation and Validation with Real World Scenes". *IEEE Transactions on Software Engineering* 24-12(12):1036–1054, 1998. Los Alamitos, CA, USA: IEEE Computer Society. ISSN: 0098-5589. doi:10.1109/32.738338 (cit. on p. 188).
- [198] Michael Hausenblas. *Learning Modern Linux*. Sebastopol, CA, USA: O'Reilly Media, Inc., Apr. 2022. ISBN: 978-1-0981-0894-6 (cit. on pp. 21, 371).
- [199] Lars Heide. "Shaping a Technology: American Punched Card Systems 1880-1914". *IEEE Annals of the History of Computing* 19(4):28–41, Oct.–Dec. 1997. Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers (IEEE). ISSN: 1058-6180. doi:10.1109/85.627897 (cit. on p. 8).
- [200] Christian Heimes. "[defusedxml](#) 0.7.1: XML Bomb Protection for Python stdlib Modules". In: Mar. 8, 2021. URL: <https://pypi.org/project/defusedxml> (visited on 2024-12-15) (cit. on p. 376).

- [201] Matthew Helmke. *Ubuntu Linux Unleashed 2021 Edition*. 14th ed. Reading, MA, USA: Addison-Wesley Professional, Aug. 2020. ISBN: 978-0-13-668539-5 (cit. on pp. 14, 22, 370, 375).
- [202] Michael J. Hernandez. *Database Design for Mere Mortals: 25th Anniversary Edition*. 4th ed. Reading, MA, USA: Addison-Wesley Professional, Dec. 2020. ISBN: 978-0-13-678813-3 (cit. on p. 17).
- [203] D. Richard Hipp et al. "Well-Known Users of *SQLite*". In: *SQLite*. Charlotte, NC, USA: Hipp, Wyrick & Company, Inc. (Hwaci), Jan. 2, 2023. URL: <https://www.sqlite.org/famous.html> (visited on 2025-01-12) (cit. on pp. 15, 374).
- [204] "History of Units". In: *PostgreSQL Documentation*. 17.4. The PostgreSQL Global Development Group (PGDG), Feb. 20, 2025. Chap. B.6. URL: <https://www.postgresql.org/docs/17/datetime-units-history.html> (visited on 2025-03-01) (cit. on pp. 122, 125).
- [205] Jeffrey A. Hoffer, Venkataraman Ramesh, and Heikki Topi. *Modern Database Management*. 13th ed. Hoboken, NJ, USA: Pearson Education, Inc., Mar. 2021. ISBN: 978-0-13-477365-0 (cit. on p. 16).
- [206] Manuel Hoffmann, Frank Nagle, and Yanuo Zhou. *The Value of Open Source Software*. Working Paper 24-038. Boston, MA, USA: Harvard Business School, Jan. 1, 2024. URL: https://www.hbs.edu/ris/Publication%20Files/24-038_51f8444f-502c-4139-8bf2-56eb4b65c58a.pdf (visited on 2025-06-04) (cit. on pp. 14, 373).
- [207] Steve Hollasch. "IEEE Standard 754 Floating Point Numbers". In: *CSE401: Introduction to Compiler Construction*. Seattle, WA, USA: University of Washington, Jan. 8, 1997. URL: <https://courses.cs.washington.edu/courses/cse401/01au/details/fp.html> (visited on 2024-07-05) (cit. on p. 105).
- [208] Herman Hollerith. *Apparatus for Compiling Statistics*. United States Patent US-0395783-A. Washington, D.C., USA: United States Patent Office, Sept. 23, 1884–Jan. 8, 1889. URL: <https://ppubs.uspto.gov/pubwebapp/static/pages/ppubsbasic.html> (visited on 2025-01-07) (cit. on p. 8).
- [209] Herman Hollerith. *Machine for Tabulating Statistics*. United States Patent 526,130. Washington, D.C., USA: United States Patent Office, Aug. 20, 1892–Sept. 18, 1894. URL: <https://ppubs.uspto.gov/pubwebapp/static/pages/ppubsbasic.html> (visited on 2025-01-07) (cit. on pp. 8, 9).
- [210] "Use the following business rules to create a Crow's Foot ERD". In: *Database Administrators*. Ed. by zz Z. New York, NY, USA: Stack Exchange Inc., Apr. 13, 2020–Oct. 27, 2024. URL: <https://dba.stackexchange.com/questions/264891> (visited on 2025-04-25) (cit. on p. 216).
- [211] "VARCHAR Primary Key – MySQL". In: *Database Administrators*. Ed. by marc_s. New York, NY, USA: Stack Exchange Inc., Oct. 22, 2014–Feb. 5, 2016. URL: <https://dba.stackexchange.com/questions/80806> (visited on 2025-02-27) (cit. on p. 106).
- [212] "How do I read ERD Notation (Crow's Feet) to convert to Natural Language?" In: *Database Administrators*. Ed. by raddevus. New York, NY, USA: Stack Exchange Inc., Feb. 17, 2016–June 22, 2018. URL: <https://dba.stackexchange.com/questions/129551> (visited on 2025-04-06) (cit. on p. 215).
- [213] "Why is naming a table's Primary Key column "Id" considered bad practice? [closed]". In: *Software Engineering*. Ed. by Jean-Philippe Leclerc. New York, NY, USA: Stack Exchange Inc., Oct. 17, 2011–Dec. 22, 2024. URL: <https://softwareengineering.stackexchange.com/questions/114728> (visited on 2025-02-27) (cit. on pp. 106, 365).
- [214] "When should a database table use timestamps?" In: *Software Engineering*. Ed. by GWed. New York, NY, USA: Stack Exchange Inc., Jan. 29, 2014–Sept. 16, 2016. URL: <https://softwareengineering.stackexchange.com/questions/225903> (visited on 2025-02-27) (cit. on p. 375).
- [215] John Hunt. *A Beginners Guide to Python 3 Programming*. 2nd ed. Undergraduate Topics in Computer Science (UTICS). Cham, Switzerland: Springer, 2023. ISBN: 978-3-031-35121-1. doi:10.1007/978-3-031-35122-8 (cit. on pp. 20, 56, 373).
- [216] John D. Hunter. "Matplotlib: A 2D Graphics Environment". *Computing in Science & Engineering* 9(3):90–95, May–June 2007. Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers (IEEE). ISSN: 1521-9615. doi:10.1109/MCSE.2007.55 (cit. on p. 371).

- [217] John D. Hunter, Darren Dale, Eric Firing, Michael Droettboom, and The Matplotlib Development Team. *Matplotlib: Visualization with Python*. Austin, TX, USA: NumFOCUS, Inc., 2012–2025. URL: <https://matplotlib.org> (visited on 2025-02-02) (cit. on p. 371).
- [218] “Identity Columns”. In: *PostgreSQL Documentation*. 17.4. The PostgreSQL Global Development Group (PGDG), Feb. 20, 2025. Chap. 5.3. URL: <https://www.postgresql.org/docs/17/ddl-identity-columns.html> (visited on 2025-02-27) (cit. on pp. 106, 284).
- [219] *IEEE Recommended Practice for Software Requirements Specifications*. IEEE Std 830-1998. New York, NY, USA: Institute of Electrical and Electronics Engineers (IEEE), Oct. 20, 1998. doi:10.1109/IEEEESTD.1998.88286. URL: <https://cse.msu.edu/~cse870/IEEEExplore-SRS-template.pdf> (visited on 2025-03-27). Superseded by [426] (cit. on pp. 188, 374, 407).
- [220] *IEEE Standard for Floating-Point Arithmetic*. IEEE Std 754™-2019 (Revision of IEEE Std 754-2008). New York, NY, USA: Institute of Electrical and Electronics Engineers (IEEE), June 13, 2019 (cit. on p. 105).
- [221] *IEEE Standard for Information Technology--Portable Operating System Interfaces (POSIX(TM))--Part 2: Shell and Utilities*. IEEE Std 1003.2-1992. New York, NY, USA: Institute of Electrical and Electronics Engineers (IEEE), June 23, 1993. URL: <https://mirror.math.princeton.edu/pub/oldlinux/Linux.old/Ref-docs/POSIX/all.pdf> (visited on 2025-03-27). Board Approved: 1992-09-17, ANSI Approved: 1993-04-05. See unapproved draft IEEE P1003.2 Draft 11.2 of 9 1991 at the url (cit. on p. 373).
- [222] *Information Technology – Database Languages – SQL – Part 1: Framework (SQL/Framework), Part 1*. International Standard ISO/IEC 9075-1:2023(E), Sixth Edition, (ANSI X3.135). Geneva, Switzerland: International Organization for Standardization (ISO) and International Electrotechnical Commission (IEC), June 2023. URL: [https://standards.iso.org/ittf/PubliclyAvailableStandards/ISO_IEC_9075-1_2023_ed_6_-_id_76583_Publication_PDF_\(en\).zip](https://standards.iso.org/ittf/PubliclyAvailableStandards/ISO_IEC_9075-1_2023_ed_6_-_id_76583_Publication_PDF_(en).zip) (visited on 2025-01-08). Consists of several parts, see <https://modern-sql.com/standard> for information where to obtain them. (Cit. on pp. 13, 89, 93, 374).
- [223] *Information Technology -- Modeling Languages -- Part 2: Syntax and Semantics for IDEF1X₉₇ (IDEF_{object})*. ISO/IEC/IEEE International Standard 31320-2-2012(E). Geneva, Switzerland: International Organization for Standardization (ISO), International Electrotechnical Commission (IEC), and New York, NY, USA: Institute of Electrical and Electronics Engineers (IEEE), Sept. 15–Oct. 30, 2012. doi:10.1109/IEEEESTD.2012.6357338 (cit. on pp. 195, 370).
- [224] *Information Technology – Universal Coded Character Set (UCS)*. International Standard ISO/IEC 10646:2020. Geneva, Switzerland: International Organization for Standardization (ISO) and International Electrotechnical Commission (IEC), Dec. 2020 (cit. on pp. 375, 376, 412).
- [225] Joseph Ingino. *Software Architect’s Handbook*. Birmingham, England, UK: Packt Publishing Ltd, Aug. 2018. ISBN: 978-1-78862-406-0 (cit. on pp. 179, 180, 182, 187, 188, 374).
- [226] “`INSERT...RETURNING`”. In: *MariaDB Server Documentation*. Milpitas, CA, USA: MariaDB, 2025. URL: <https://mariadb.com/docs/server/reference/sql-statements/data-manipulation/inserting-loading-data/insertreturning> (visited on 2025-07-06) (cit. on pp. 137, 279, 306).
- [227] *Python 3 Documentation. Installing Python Modules*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. URL: <https://docs.python.org/3/installing> (visited on 2024-08-17) (cit. on p. 373).
- [228] *International Workshop on the Software Process and Software Environments*. Mar. 27–29, 1985, Coto de Caza, Trabuco Canyon, CA, USA. New York, NY, USA: Association for Computing Machinery (ACM), 1985. ISBN: 978-0-89791-202-0. Also contained in [343] (cit. on p. 402).
- [229] *Introduction to the Dewey Decimal Classification*. Dublin, OH, USA: Ohio College Library Center (OCLC), May 17, 2019. URL: <https://www.oclc.org/content/dam/oclc/dewey/versions/print/intro.pdf> (visited on 2025-05-28) (cit. on p. 8).
- [230] *IP2Location™ ISO 3166-2 Subdivision Code*. Bayan Baru, Pulau Pinang, Malaysia: Hexasoft Development Sdn. Bhd., Apr. 16, 2025. URL: <https://www.ip2location.com/free/iso3166-2> (visited on 2025-04-29) (cit. on p. 226).

- [231] Jay E. Israel, James G. Mitchell, and Howard E. Sturgis. *Separating Data From Function in a Distributed File System*. Blue and White Series CSL-78-5. Palo Alto, CA, USA: Xerox Palo Alto Research Center (PARC), Sept. 1978 (cit. on p. 12).
- [232] Robert Johansson. *Numerical Python: Scientific Computing and Data Science Applications with NumPy, SciPy and Matplotlib*. New York, NY, USA: Apress Media, LLC, Dec. 2018. ISBN: 978-1-4842-4246-9 (cit. on pp. 139, 371, 372, 374).
- [233] “Joined Tables”. In: *PostgreSQL Documentation*. 17.4. The PostgreSQL Global Development Group (PGDG), Feb. 20, 2025. Chap. 7.2.1.1. URL: <https://www.postgresql.org/docs/17/queries-table-expressions.html#QUERIES-JOIN> (visited on 2025-03-01) (cit. on pp. 125, 128, 259).
- [234] “exit – Terminate a Process”. In: *POSIX.1-2024: The Open Group Base Specifications Issue 8, IEEE Std 1003.1™-2024 Edition*. Ed. by Andrew Josey. Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers (IEEE) and San Francisco, CA, USA: The Open Group, Aug. 8, 2024. URL: <https://pubs.opengroup.org/onlinepubs/9799919799/functions/exit.html> (visited on 2024-10-30) (cit. on p. 369).
- [235] “Locale”. In: *POSIX.1-2024: The Open Group Base Specifications Issue 8, IEEE Std 1003.1™-2024 Edition*. Ed. by Andrew Josey. Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers (IEEE) and San Francisco, CA, USA: The Open Group, Aug. 8, 2024. Chap. XBD 7. URL: https://pubs.opengroup.org/onlinepubs/9799919799/basedefs/V1_chap07.html (visited on 2025-01-21) (cit. on p. 371).
- [236] Andrew Josey, ed. *POSIX.1-2024: The Open Group Base Specifications Issue 8, IEEE Std 1003.1™-2024 Edition*. Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers (IEEE) and San Francisco, CA, USA: The Open Group, Aug. 8, 2024. URL: <https://pubs.opengroup.org/onlinepubs/9799919799> (visited on 2024-10-30).
- [237] “stderr, stdin, stdout – Standard I/O Streams”. In: *POSIX.1-2024: The Open Group Base Specifications Issue 8, IEEE Std 1003.1™-2024 Edition*. Ed. by Andrew Josey. Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers (IEEE) and San Francisco, CA, USA: The Open Group, Aug. 8, 2024. URL: <https://pubs.opengroup.org/onlinepubs/9799919799/functions/stdin.html> (visited on 2024-10-30) (cit. on p. 374).
- [238] Sean P. Kane and Karl Matthias. *Docker: Up & Running*. 3rd ed. Sebastopol, CA, USA: O'Reilly Media, Inc., Apr. 2023. ISBN: 978-1-0981-3182-1 (cit. on p. 369).
- [239] Bill Karwin. *SQL Antipatterns: Avoiding the Pitfalls of Database Programming*. Flower Mound, TX, USA: Pragmatic Bookshelf by The Pragmatic Programmers, L.L.C., June 2017. ISBN: 978-1-934356-55-5 (cit. on p. 17).
- [240] Shannon Kempe and Paul Williams. *A Short History of the ER Diagram and Information Modeling*. Studio City, CA, USA: Dataversity Digital LLC, Sept. 25, 2012. URL: <https://www.dataversity.net/a-short-history-of-the-er-diagram-and-information-modeling> (visited on 2025-03-06) (cit. on pp. 12, 195, 369).
- [241] Alfons Kemper and André Eickler. *Datenbanksysteme: Eine Einführung*. De Gruyter Studium. Berlin, Germany: Walter de Gruyter GmbH, 2015. ISBN: 978-3-11-044375-2 (cit. on p. 17).
- [242] William (Bill) Kent. “A Simple Guide to Five Normal Forms in Relational Database Theory”. *Communications of the ACM (CACM)* 26(2):120–125, Sept. 1982–Feb. 1983. New York, NY, USA: Association for Computing Machinery (ACM). ISSN: 0001-0782. doi:10.1145/358024.358054. URL: <https://www.cs.dartmouth.edu/~cs61/Resources/Papers/CACM%20Kent%20Five%20Normal%20Forms.pdf> (visited on 2025-05-03) (cit. on pp. 330, 342, 347, 356, 368, 372).
- [243] Tom Kilburn, R. Bruce Payne, and David J. Howarth. “The Atlas Supervisor”. In: *Eastern Joint Computer Conference: Computers – Key to Total Systems Control, (AFIPS'1961, Eastern)*. Dec. 12–14, 1961, Washington, D.C., USA. Ed. by Willis H. Ware. New York, NY, USA: Association for Computing Machinery (ACM), 1961, pp. 279–294. ISBN: 978-1-4503-7873-4. doi:10.1145/1460764.1460786. URL: <https://www.chilton-computing.org.uk/acl/technology/atlas/p019.htm> (visited on 2025-01-08) (cit. on p. 9).
- [244] Won Kim. “Relational Database Systems”. *ACM Computing Surveys (CSUR)* 11(3):187–211, Sept. 1979. New York, NY, USA: Association for Computing Machinery (ACM). ISSN: 0360-0300. doi:10.1145/356778.356780 (cit. on p. 11).

- [245] Dmitry Kirsanov. *The Book of Inkscape*. 2nd ed. San Francisco, CA, USA, Nov. 2021. ISBN: 978-1-7185-0175-1 (cit. on p. 370).
- [246] Barbara Klein, Richard Alan Long, Kenneth Ray Blackman, Diane Lynne Goff, Stephen P. Nathan, Moira McFadden Lanyi, Margaret M. Wilson, John Butterweck, and Sandra L. Sherrill. *Introduction to IMS: Your Complete Guide to IBM Information Management System*. 2nd ed. Indianapolis, IN, USA: IBM Press, Mar. 13, 2012. ISBN: 978-0-13-288687-1 (cit. on pp. 11, 226, 370).
- [247] Bernd Klein. *Einführung in Python 3 – Für Ein- und Umsteiger*. 3., überarbeitete. München, Bayern, Germany: Carl Hanser Verlag GmbH & Co. KG, 2018. ISBN: 978-3-446-45208-4. doi:10.3139/9783446453876 (cit. on p. 20).
- [248] Leonard Kleinrock. "An Early History of the Internet [History of Communications]". *IEEE Communications Magazine* 48(8):26–36, Aug. 2010. Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers (IEEE). ISSN: 0163-6804. doi:10.1109/MCOM.2010.5534584. URL: <https://www.researchgate.net/publication/262316090> (visited on 2025-05-29) (cit. on p. 12).
- [249] Leonard Kleinrock. *Information Flow in Large Communication Nets, Ph.D. Thesis Proposal*. Cambridge, MA, USA: Massachusetts Institute of Technology (MIT), May 31, 1961. URL: <https://www.lk.cs.ucla.edu/data/files/Kleinrock/Information%20Flow%20in%20Large%20Communication%20Nets.pdf> (visited on 2025-01-20) (cit. on pp. 11, 12).
- [250] Leah Knobel and Neely Tucker. "It's As If It Was Written in... Clay. For 4,200 Years". *Timeless Stories from the Library of Congress*, Nov. 22, 2021. Washington, D.C., USA: Library of Congress. ISSN: 2836-9459. URL: <https://blogs.loc.gov/loc/2021/11/its-as-if-it-was-written-in-clay-for-4200-years> (visited on 2025-01-08). See also [112] (cit. on pp. 8, 386).
- [251] Katie Kodes. *Intro to XML, JSON, & YAML*. London, England, UK: Payhip, 2019–Sept. 4, 2020 (cit. on pp. 2, 376).
- [252] Petr Kozelek. *Audit Trail – Tracing Data Changes in Database*. Toronto, ON, Canada: Code-Project, Aug. 30, 2010. URL: <https://www.codeproject.com/Articles/105768/Audit-Trail-Tracing-Data-Changes-in-Database> (visited on 2025-04-09) (cit. on p. 226).
- [253] Tim Kraska and Michael Cafarella. *6.5830/6.5831: Database Systems*. Cambridge, MA, USA: Massachusetts Institute of Technology (MIT), Aut. 2024. URL: <https://dsg.csail.mit.edu/6.5830> (visited on 2025-01-08) (cit. on p. 16).
- [254] Tim Kraska and Michael Cafarella. "Introduction to Databases". In: *6.5830/6.5831: Database Systems*. Cambridge, MA, USA: Massachusetts Institute of Technology (MIT), Aut. 2024. Chap. 1. URL: <https://dsg.csail.mit.edu/6.5830/lectures/lec1.pdf> (visited on 2025-01-08) (cit. on pp. 11, 226, 370).
- [255] Andrew M. Kuchling. *Python 3 Documentation. Regular Expression HOWTO*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. URL: <https://docs.python.org/3/howto/regex.html> (visited on 2024-11-01) (cit. on p. 373).
- [256] Darl Kuhn and Thomas Kyte. *Expert Oracle Database Architecture: Techniques and Solutions for High Performance and Productivity*. 4th ed. New York, NY, USA: Apress Media, LLC, Nov. 2021. ISBN: 978-1-4842-7499-6 (cit. on pp. 15, 18, 373).
- [257] Animesh Kumar, Sandip Dutta, and Prashant Pranav. "Analysis of SQL Injection Attacks in the Cloud and in WEB Applications". *Security and Privacy* 7(3), May–June 2024. Chichester, West Sussex, England, UK: John Wiley and Sons Ltd. ISSN: 2475-6725. doi:10.1002/SPY2.370 (cit. on pp. 143, 374).
- [258] E. A. Kurako and V. L. Orlov. "Database Migration from ORACLE to PostgreSQL". 49(5):455–463, Oct. 2023. doi:10.1134/S0361768823050055 (cit. on p. 15).
- [259] James F. Kurose and Keith Ross. *Computer Networking: A Top Down Approach*. 8th ed. Hoboken, NJ, USA: Pearson Education, Inc., Mar. 30, 2020. ISBN: 978-0-13-668155-7 (cit. on pp. 11, 12, 373).
- [260] Jay LaCroix. *Mastering Ubuntu Server*. 4th ed. Birmingham, England, UK: Packt Publishing Ltd, Sept. 2022. ISBN: 978-1-80323-424-3 (cit. on p. 374).

- [261] Joan Lambert and Curtis Frye. *Microsoft Office Step by Step (Office 2021 and Microsoft 365)*. Hoboken, NJ, USA: Microsoft Press, Pearson Education, Inc., June 2022. ISBN: 978-0-13-754493-6 (cit. on pp. 15, 372).
- [262] Charles Landau. *TensorFlow Deep Dive: Build, Train, and Deploy Machine Learning Models with TensorFlow*. Sebastopol, CA, USA: O'Reilly Media, Inc., Dec. 2023 (cit. on pp. 139, 375).
- [263] Łukasz Langa. *Literature Overview for Type Hints*. Python Enhancement Proposal (PEP) 482. Beaverton, OR, USA: Python Software Foundation (PSF), Jan. 8, 2015. URL: <https://peps.python.org/pep-0482> (visited on 2024-10-09) (cit. on p. 375).
- [264] Eugene Leighton Lawler, Jan Karel Lenstra, Alexander Hendrik George Rinnooy Kan, and David B. Shmoys. "Sequencing and Scheduling: Algorithms and Complexity". In: *Production Planning and Inventory*. Ed. by Stephen C. Graves, Alexander Hendrik George Rinnooy Kan, and Paul H. Zipkin. Vol. IV of Handbooks of Operations Research and Management Science. Amsterdam, The Netherlands: Elsevier B.V., 1993. Chap. 9, pp. 445–522. ISSN: 0927-0507. ISBN: 978-0-444-87472-6. doi:10.1016/S0927-0507(05)80189-6. URL: <http://alexandria.tue.nl/repository/books/339776.pdf> (visited on 2023-12-06) (cit. on pp. 370, 372).
- [265] Kent D. Lee and Steve Hubbard. *Data Structures and Algorithms with Python*. Undergraduate Topics in Computer Science (UTICS). Cham, Switzerland: Springer, 2015. ISBN: 978-3-319-13071-2. doi:10.1007/978-3-319-13072-9 (cit. on pp. 20, 56, 373).
- [266] Michael Lee, Ivan Levkivskyi, and Jukka Lehtosalo. *Literal Types*. Python Enhancement Proposal (PEP) 586. Beaverton, OR, USA: Python Software Foundation (PSF), Mar. 14, 2019. URL: <https://peps.python.org/pep-0586> (visited on 2024-12-17) (cit. on p. 371).
- [267] Jukka Lehtosalo, Ivan Levkivskyi, Jared Hance, Ethan Smith, Guido van Rossum, Jelle "JelleZijlstra" Zijlstra, Michael J. Sullivan, Shantanu Jain, Xuanda Yang, Jingchen Ye, Nikita Sobolev, and Mypy Contributors. *Mypy – Static Typing for Python*. San Francisco, CA, USA: GitHub Inc, 2024. URL: <https://github.com/python/mypy> (visited on 2024-08-17) (cit. on p. 372).
- [268] Marc-André Lemburg. *Python Database API Specification v2.0*. Python Enhancement Proposal (PEP) 249. Beaverton, OR, USA: Python Software Foundation (PSF), Apr. 12, 1999. URL: <https://peps.python.org/pep-0249> (visited on 2025-02-02) (cit. on pp. 56, 140, 141, 373).
- [269] Reuven M. Lerner. *Pandas Workout*. Shelter Island, NY, USA: Manning Publications, June 2024. ISBN: 978-1-61729-972-8 (cit. on pp. 139, 373).
- [270] *LibreOffice – The Document Foundation*. Berlin, Germany: The Document Foundation, 2024. URL: <https://www.libreoffice.org> (visited on 2024-12-12) (cit. on pp. 2, 15, 45, 101, 371, 372).
- [271] Joseph Carl Robnett "Lick" Licklider. *MEMORANDUM FOR: Members and Affiliates of the Intergalactic Computer Network*. Washington, D.C., USA: Advanced Research Projects Agency (ARPA), Apr. 23, 1963. URL: https://worrydream.com/refs/Licklider_1963_-Members_and_Affiliates_of_the_Intergalactic_Computer_Network.pdf (visited on 2025-01-20) (cit. on p. 12).
- [272] Gloria Lotha, Aakanksha Gaur, Erik Gregersen, Swati Chopra, and William L. Hosch. "Client-Server Architecture". In: *Encyclopaedia Britannica*. Ed. by The Editors of Encyclopaedia Britannica. Chicago, IL, USA: Encyclopædia Britannica, Inc., Jan. 3, 2025. URL: <https://www.britannica.com/technology/client-server-architecture> (visited on 2025-01-20) (cit. on pp. 12, 368).
- [273] Tony Loton. "Data Modeling: Entity-Relationship Diagram (ER Diagram)". In: *ModernAnalyst.com*. Calabasas, CA, USA: Modern Analyst Media, LLC, 2006–2025. URL: <https://www.modernanalyst.com/Resources/Articles/tabid/115/ID/2008/Data-Modeling-Entity-Relationship-Diagram-ER-Diagram.aspx> (visited on 2025-04-05) (cit. on p. 216).
- [274] Marc Loy, Patrick Niemeyer, and Daniel Leuck. *Learning Java*. 5th ed. Sebastopol, CA, USA: O'Reilly Media, Inc., Mar. 2020. ISBN: 978-1-4920-5627-0 (cit. on p. 370).
- [275] Luqi. "Software Evolution Through Rapid Prototyping". *Computer* 22(5):13–25, May 1989. Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers (IEEE). ISSN: 0018-9162. doi:10.1109/2.27953 (cit. on p. 181).

- [276] Peter Luschny. *A New Kind of Factorial Function*. Highland Park, NJ, USA: The OEIS Foundation Inc., Oct. 4, 2015. URL: <https://oeis.org/A000142/a000142.pdf> (visited on 2024-09-29) (cit. on p. 368).
- [277] Mark Lutz. *Learning Python*. 6th ed. Sebastopol, CA, USA: O'Reilly Media, Inc., Mar. 2025. ISBN: 978-1-0981-7130-8 (cit. on p. 373).
- [278] Stuart Macfarlane. *A Brief History of Databases*. Claymont, DE, USA: Sutori Website Administrator/HSTRY LTD., Nov. 1, 2024. URL: <https://www.sutori.com/en/story/a-brief-history-of-databases> (visited on 2025-01-10) (cit. on p. 8).
- [279] *Machine Readable Travel Documents. Part 3: Specifications Common to all MRTDs. Eighth Edition*. Tech. rep. Doc 9303. Montreal, QC, Canada: International Civil Aviation Organization (ICAO), 2021. URL: https://www.icao.int/publications/documents/9303_p3_cons_en.pdf (visited on 2025-04-03) (cit. on p. 209).
- [280] Robert W. Mantha. "Data Flow and Data Structure Modeling for Database Requirements Determination: A Comparative Study". *MIS Quarterly: Management Information Systems Quarterly (MISQ)* 11(4):531–545, Dec. 1987. Minneapolis, MN, USA: University of Minnesota – Twin Cities. ISSN: 0276-7783 (cit. on p. 188).
- [281] *MariaDB Server Documentation*. Milpitas, CA, USA: MariaDB, 2025. URL: <https://mariadb.com/kb/en/documentation> (visited on 2025-04-24) (cit. on p. 371).
- [282] "Ferranti Computing Systems Atlas 1 Brochure: 1962". In: *Chilton Computing*. Ed. by Victoria Marshall. Swindon, Wiltshire, England, UK: Science and Technology Facilities Council (STFC), UK Research and Innovation (UKRI), May 2, 2025. URL: <https://www.chilton-computing.org.uk/acl/technology/atlas/p002.htm> (visited on 2025-05-29) (cit. on pp. 9, 10).
- [283] James Martin. *Rapid Application Development*. Indianapolis, IN, USA: Macmillan Publishing Co., Inc., Jan. 1991. ISBN: 978-0-02-376775-3 (cit. on pp. 181, 182).
- [284] "Mathematical Functions and Operators". In: *PostgreSQL Documentation*. 17.4. The PostgreSQL Global Development Group (PGDG), Feb. 20, 2025. Chap. 9.3. URL: <https://www.postgresql.org/docs/17/functions-math.html> (visited on 2025-02-27) (cit. on p. 111).
- [285] Aaron Maxwell. *What are f-strings in Python and how can I use them?* Oakville, ON, Canada: Infinite Skills Inc, June 2017. ISBN: 978-1-4919-9486-3 (cit. on p. 369).
- [286] Steve McConnel. "From the Editor – An Ounce of Prevention". *IEEE Software* 18(3), May–June 2001. Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers (IEEE). ISSN: 0740-7459. doi:10.1109/MS.2001.922718 (cit. on p. 187).
- [287] Steve McConnel. *Rapid Development: Taming Wild Software Schedules*. Hoboken, NJ, USA: Microsoft Press, Pearson Education, Inc., July 1996. ISBN: 978-1-55615-900-8 (cit. on pp. 181, 188, 370, 373).
- [288] Ron McFadyen and Cindy Miller. *Relational Databases and Microsoft Access*. 3rd ed. Palatine, IL, USA: Harper College, 2014–2019. URL: <https://harpercollege.pressbooks.pub/relationaldatabases> (visited on 2025-04-11) (cit. on pp. 15, 18, 371).
- [289] Michael McLaughlin. *MySQL Workbench: Data Modeling & Development*. Oracle Press. New York, NY, USA: McGraw-Hill, Apr. 9, 2013. ISBN: 978-0-07-179188-5 (cit. on pp. 232, 372).
- [290] *MDS: Melvil Decimal System*. Portland, ME, USA: LibraryThing, Inc., 2025. URL: <https://www.librarything.com/mds> (visited on 2025-05-29) (cit. on p. 9).
- [291] Jim Melton and Alan R. Simon. *SQL: 1999 – Understanding Relational Language Components*. The Morgan Kaufmann Series in Data Management Systems. Burlington, MA, USA/San Mateo, CA, USA: Morgan Kaufmann Publishers, June 2001. ISBN: 978-1-55860-456-8 (cit. on pp. 17, 89, 93, 374).
- [292] Carl Meyer. *Python Virtual Environments*. Python Enhancement Proposal (PEP) 405. Beaverton, OR, USA: Python Software Foundation (PSF), June 13, 2011–May 24, 2012. URL: <https://peps.python.org/pep-0405> (visited on 2024-12-25) (cit. on p. 376).
- [293] *Microsoft Word*. Redmond, WA, USA: Microsoft Corporation, 2024. URL: <https://www.microsoft.com/en-us/microsoft-365/word> (visited on 2024-12-12) (cit. on p. 372).
- [294] Zsolt Nagy. *Regex Quick Syntax Reference: Understanding and Using Regular Expressions*. New York, NY, USA: Apress Media, LLC, Aug. 2018. ISBN: 978-1-4842-3876-9 (cit. on p. 373).

- [295] Constantine Nalimov. *Crow's Foot Notation in Entity-Relationship Diagrams*. Prague, Czech Republic: Blocshop s.r.o., Sept. 2, 2020. URL: <https://www.gleek.io/blog/crows-foot-notation> (visited on 2025-04-05) (cit. on pp. 215, 216).
- [296] Constantine Nalimov. *ER Diagram for a Hospital Management System (Crow's Foot Notation)*. Prague, Czech Republic: Blocshop s.r.o., Dec. 3, 2012. URL: <https://www.gleek.io/blog/erd-hospital-management> (visited on 2025-04-05) (cit. on p. 216).
- [297] Adrien “anayrat” Nayrat. “PostgreSQL: Deferrable Constraints”. In: *Select * from Adrien*. Valence, Drôme, Rhône-Alpes, France, Aug. 13, 2016. URL: <https://blog.anayrat.info/en/2016/08/13/postgresql-deferrable-constraints> (visited on 2025-04-10) (cit. on p. 221).
- [298] Catherine Nelson. *Software Engineering for Data Scientists*. Sebastopol, CA, USA: O'Reilly Media, Inc., Apr. 2024. ISBN: 978-1-0981-3620-8 (cit. on pp. 179, 180, 182, 374).
- [299] Cameron Newham and Bill Rosenblatt. *Learning the Bash Shell – Unix Shell Programming: Covers Bash 3.0*. 3rd ed. Sebastopol, CA, USA: O'Reilly Media, Inc., 2005. ISBN: 978-0-596-00965-6 (cit. on p. 368).
- [300] Thomas Nield. *An Introduction to Regular Expressions*. Sebastopol, CA, USA: O'Reilly Media, Inc., June 2019. ISBN: 978-1-4920-8255-2 (cit. on p. 373).
- [301] “Numeric Types”. In: *PostgreSQL Documentation*. 17.4. The PostgreSQL Global Development Group (PGDG), Feb. 20, 2025. Chap. 8.1. URL: <https://www.postgresql.org/docs/17/datatype-numeric.html> (visited on 2025-02-27) (cit. on pp. 105, 125).
- [302] NumPy Team. *NumPy*. San Francisco, CA, USA: GitHub Inc and Austin, TX, USA: NumFOCUS, Inc. URL: <https://numpy.org> (visited on 2025-02-02) (cit. on pp. 139, 372).
- [303] John J. O'Connor and Edmund F. Robertson. *Liu Hui*. St Andrews, Scotland, UK: University of St Andrews, School of Mathematics and Statistics, Dec. 2003. URL: https://mathshistory.st-andrews.ac.uk/Biographies/Liu_Hui (visited on 2024-08-10) (cit. on p. 201).
- [304] Kevin C. O’Kane. *The Mumps Programming Language*. North Charleston, SC, USA: CreateSpace Independent Publishing Platform, June 19, 2008. ISBN: 978-1-4382-4338-2 (cit. on p. 226).
- [305] Kevin C. O’Kane. *The Mumps Programming Language*. Cedar Falls, IA, USA: University of Northern Iowa, Apr. 4, 2025. URL: <https://www.cs.uni.edu/~okane> (visited on 2025-04-04) (cit. on p. 226).
- [306] Regina O. Obe and Leo S. Hsu. *PostgreSQL: Up and Running*. 3rd ed. Sebastopol, CA, USA: O'Reilly Media, Inc., Oct. 2017. ISBN: 978-1-4919-6336-4 (cit. on pp. iv, 14, 20, 25, 88, 367, 373).
- [307] A. Jefferson Offutt. “Unit Testing Versus Integration Testing”. In: *Test: Faster, Better, Sooner – IEEE International Test Conference (ITC'1991)*. Oct. 26–30, 1991, Nashville, TN, USA. Los Alamitos, CA, USA: IEEE Computer Society, 1991. Chap. Paper P2.3, pp. 1108–1109. ISSN: 1089-3539. ISBN: 978-0-8186-9156-0. doi:10.1109/TEST.1991.519784 (cit. on p. 375).
- [308] Michael Olan. “Unit Testing: Test Early, Test Often”. *Journal of Computing Sciences in Colleges (JCSC)* 19(2):319–328, Dec. 2003. New York, NY, USA: Association for Computing Machinery (ACM). ISSN: 1937-4771. doi:10.5555/948785.948830. URL: <https://www.researchgate.net/publication/255673967> (visited on 2025-09-05) (cit. on p. 375).
- [309] *OMG® Unified Modeling Language® (OMG UML®)*. Version 2.5.1. OMG Document formal/2017-12-05. Milford, MA, USA: Object Management Group, Inc. (OMG), Dec. 2017. URL: <https://www.omg.org/spec/UML/2.5.1/PDF> (visited on 2025-03-30) (cit. on pp. 195, 219, 375).
- [310] *Oracle CODASYL DBMS™ – Database Administration Reference Manual*. 7.4.1. Redwood Shores, CA, USA: Oracle Corporation, Aug. 2022. URL: <https://www.oracle.com/a/tech/docs/collateral/dbm0741-dbadmin-ref.pdf> (cit. on p. 226).
- [311] “Oracle Timeline: Highlighting the most important moments in Oracle’s history, with commentary from the people who made it happen”. *Profit – The Executive Guide to Oracle Applications* 12(2):26–33, May 2007. Redwood Shores, CA, USA: Oracle Corporation. ISSN: 1531-7455. URL: <https://www.oracle.com/us/corporate/profit/profit-may-07-151925.pdf> (visited on 2025-06-03) (cit. on pp. 13, 15).

- [312] Richard H. Orenstein and Robert C. Daley. *TO: Computation Center Staff. LDEDT and BPEDT, the CTSS Disk Editors.* Tech. rep. CC-208. Cambridge, MA, USA: Massachusetts Institute of Technology (MIT) Computation Center, May 9, 1963. URL: <https://people.csail.mit.edu/saltzer/CTSS/CTSS-Documents/CC-Memos/CC-208.pdf> (visited on 2025-01-08) (cit. on p. 10).
- [313] Robert Orfali, Dan Harkey, and Jeri Edwards. *Client/Server Survival Guide*. 3rd ed. Chichester, West Sussex, England, UK: John Wiley and Sons Ltd., Jan. 25, 1999. ISBN: 978-0-471-31615-2 (cit. on pp. 12, 368).
- [314] Christopher Painter-Wakefield. *A Practical Introduction to Databases*. Luck, WI, USA: Runestone Academy, 2022. URL: https://runestone.academy/ns/books/published/practical_db/index.html (visited on 2025-04-06) (cit. on p. 16).
- [315] Camila A. Paiva, Raquel Maximino, Frederico Paiva, Rafael Accetta Vieira, Nicole Espanha, João Felipe Pimentel, Igor Wiese, Marco Aurélio Gerosa, Igor Steinmacher, Leonardo Murta, and Vanessa Braganholo. “Analyzing the Adoption of Database Management Systems throughout the History of Open Source Projects”. *Empirical Software Engineering: An International Journal* 30(3):71, Feb. 2025. London, England, UK: Springer Nature Limited. ISSN: 1382-3256. doi:10.1007/S10664-025-10627-Z. URL: <https://www.authorea.com/users/677798/articles/674742> (visited on 2025-06-04) (cit. on pp. 14, 15, 88).
- [316] Ashwin Pajankar. *Hands-on Matplotlib: Learn Plotting and Visualizations with Python* 3. New York, NY, USA: Apress Media, LLC, Nov. 2021. ISBN: 978-1-4842-7410-1 (cit. on p. 371).
- [317] Ashwin Pajankar. *Python Unit Test Automation: Automate, Organize, and Execute Unit Tests in Python*. New York, NY, USA: Apress Media, LLC, Dec. 2021. ISBN: 978-1-4842-7854-3 (cit. on p. 375).
- [318] Charles C. Palmer. *COSC 61 Winter 2025: Database Systems*. Hanover, MD, USA: Dartmouth College, Jan.–Mar. 2025. URL: <https://www.cs.dartmouth.edu/~cs61> (visited on 2025-04-06) (cit. on p. 16).
- [319] Charles C. Palmer. “ER Modeling”. In: *COSC 61 Winter 2025: Database Systems*. Hanover, MD, USA: Dartmouth College, Jan. 15, 2025. Chap. 4. URL: <http://www.cs.dartmouth.edu/~cs61/Lectures/04%20-%20ER%20Modeling/04%20-%20ER%20Modeling.pdf> (visited on 2025-04-08) (cit. on p. 220).
- [320] Pandas Developers. *Pandas*. Austin, TX, USA: NumFOCUS, Inc. and Montreal, QC, Canada: OVHcloud. URL: <https://pandas.pydata.org> (visited on 2025-02-02) (cit. on pp. 139, 373).
- [321] Burt Parker. “Introducing ANSI-X3.138-1988: A Standard for Information Resource Dictionary System (IRDS)”. In: *Second Symposium on Assessment of Quality Software Development Tools*. May 27–29, 1992, New Orleans, LA, USA. Los Alamitos, CA, USA: IEEE Computer Society, 1992, pp. 90–99. ISBN: 978-0-8186-2620-3. doi:10.1109/AQSDT.1992.205841 (cit. on p. 12).
- [322] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Z. Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. “PyTorch: An Imperative Style, High-Performance Deep Learning Library”. In: *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems (NeurIPS'2019)*. Dec. 8–14, 2019, Vancouver, BC, Canada. Ed. by Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d’Alché-Buc, Emily B. Fox, and Roman Garnett. San Diego, CA, USA: The Neural Information Processing Systems Foundation (NeurIPS), 2019, pp. 8024–8035. ISBN: 978-1-7138-0793-3. URL: <https://proceedings.neurips.cc/paper/2019/hash/bdbca288fee7f92f2bfa9f7012727740-Abstract.html> (visited on 2024-07-18) (cit. on pp. 139, 373).
- [323] “Pattern Matching”. In: *PostgreSQL Documentation*. 17.4. The PostgreSQL Global Development Group (PGDG), Feb. 20, 2025. Chap. 9.7. URL: <https://www.postgresql.org/docs/17/functions-matching.html> (visited on 2025-02-27) (cit. on pp. 111, 119, 332).
- [324] Alan Paul, Vishal Sharma, and Oluwafemi Olukoya. “SQL Injection Attack: Detection, Prioritization & Prevention”. *Journal of Information Security and Applications* 85:103871, Sept. 2024. Amsterdam, The Netherlands: Elsevier B.V. ISSN: 2214-2126. doi:10.1016/J.JISA.2024.103871 (cit. on pp. 143, 374).

- [325] Linda Dailey Paulson. "Open Source Databases Move into the Marketplace". *Computer* 37(7):13–15, July 2004. Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers (IEEE). ISSN: 0018-9162. doi:10.1109/MC.2004.62 (cit. on p. 14).
- [326] Fabian Pedregos, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake VanderPlas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Edouard Duchesnay. "Scikit-learn: Machine Learning in Python". *Journal of Machine Learning Research (JMLR)* 12:2825–2830, Oct. 2011. Cambridge, MA, USA: MIT Press. ISSN: 1532-4435. doi:10.5555/1953048.2078195 (cit. on pp. 139, 374).
- [327] Yasset Pérez-Riverol, Laurent Gatto, Rui Wang, Timo Sachsenberg, Julian Uszkoreit, Felipe da Veiga Leprevost, Christian Fufezan, Tobias Ternent, Stephen J. Eglen, Daniel S. Katz, Tom J. Pollard, Alexander Konovalov, Robert M. Flight, Kai Blin, and Juan Antonio Vizcaíno. "Ten Simple Rules for Taking Advantage of Git and GitHub". *PLOS Computational Biology* 12(7), July 14, 2016. San Francisco, CA, USA: Public Library of Science (PLOS). ISSN: 1553-7358. doi:10.1371/JOURNAL.PCBI.1004947 (cit. on p. 370).
- [328] Dušan Petković. *Microsoft SQL Server 2019: A Beginner's Guide*. 7th ed. New York, NY, USA: McGraw-Hill, Jan. 2020. ISBN: 978-1-260-45888-6 (cit. on pp. 15, 17, 101, 372).
- [329] Shari Lawrence Pfleeger and Joanne M. Atlee. *Software Engineering: Theory and Practice*. Hoboken, NJ, USA: Pearson Education, Inc., Feb. 2009. ISBN: 978-0-13-606169-4 (cit. on p. 181).
- [330] Donnie Pinkston. "Converting E-R Diagrams to Relational Model". In: *CS101b – Introduction to Relational Databases*. Pasadena, CA, USA: California Institute of Technology (Caltech), Win. 2007. Chap. 17. URL: <http://users.cms.caltech.edu/~donnie/dbcourse/intro0607/lectures/Lecture17.pdf> (visited on 2025-04-04) (cit. on p. 211).
- [331] Donnie Pinkston. *CS101b – Introduction to Relational Databases*. Pasadena, CA, USA: California Institute of Technology (Caltech), Win. 2006–Spr. 2007. URL: <http://users.cms.caltech.edu/~donnie/dbcourse/intro0607> (visited on 2025-04-04) (cit. on p. 16).
- [332] Donnie Pinkston. "Entity-Relationship Model II". In: *CS101b – Introduction to Relational Databases*. Pasadena, CA, USA: California Institute of Technology (Caltech), Win. 2007. Chap. 15. URL: <http://users.cms.caltech.edu/~donnie/dbcourse/intro0607/lectures/Lecture15.pdf> (visited on 2025-04-04) (cit. on p. 213).
- [333] pip Developers. *pip Documentation v24.3.1*. Beaverton, OR, USA: Python Software Foundation (PSF), Oct. 27, 2024. URL: <https://pip.pypa.io> (visited on 2024-12-25) (cit. on p. 373).
- [334] Hasso Plattner. "Insert-Only". In: *A Course in In-Memory Data Management: The Inner Mechanics of In-Memory Databases*. 2nd ed. Berlin/Heidelberg, Germany: Springer-Verlag GmbH Germany, June 2014. Chap. Advanced Database Storage Techniques, 3, pp. 173–180. ISBN: 978-3-642-55269-4. doi:10.1007/978-3-642-55270-0_26 (cit. on p. 224).
- [335] Pope Gregory XIII. *Inter Gravissimas*. Proclamation / Papal Bull. Vatican: Catholic Church, Feb. 24, 1582 (cit. on pp. 122, 125).
- [336] "POSIX Regular Expressions". In: *PostgreSQL Documentation*. 17.4. The PostgreSQL Global Development Group (PGDG), Feb. 20, 2025. Chap. 9.7.3. URL: <https://www.postgresql.org/docs/17/functions-matching.html#FUNCTIONS-POSIX-REGEXP> (visited on 2025-02-27) (cit. on pp. 115, 373).
- [337] "libpq – C Library". In: *PostgreSQL Documentation*. 17.4. The PostgreSQL Global Development Group (PGDG), Feb. 20, 2025. Chap. Part IV. Client Interfaces, Chapter 32. URL: <https://www.postgresql.org/docs/17/libpq.html> (visited on 2025-03-05) (cit. on p. 145).
- [338] *PostgreSQL Documentation*. 17.4. The PostgreSQL Global Development Group (PGDG), Feb. 2025. URL: <https://www.postgresql.org/docs/17/index.html> (visited on 2025-02-25) (cit. on pp. 18, 94, 106, 365).
- [339] *PostgreSQL Essentials: Leveling Up Your Data Work*. Sebastopol, CA, USA: O'Reilly Media, Inc., Mar. 2024 (cit. on pp. iv, 14, 20, 25, 88, 367, 373).
- [340] *PostgreSQL JDBC Driver*. The PostgreSQL Global Development Group (PGDG), Jan. 14, 2025. URL: <https://jdbc.postgresql.org> (visited on 2025-03-05) (cit. on p. 145).

- [341] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. "1.1 Error, Accuracy, and Stability". In: *Numerical Recipes: The Art of Scientific Computing*. 3rd ed. Cambridge, England, UK: Cambridge University Press (CUP), 2007–2011. Chap. 1 Preliminaries, pp. 8–12. ISBN: 978-0-521-88068-8. URL: <https://numerical.recipes/book.html> (visited on 2024-07-27). Version 3.04 (cit. on pp. 105, 365).
- [342] Roger S. Pressman and Bruce R. Maxim. *Software Engineering: A Practitioner's Approach (SEPA)*. 9th ed. New York, NY, USA: McGraw-Hill, 2020. ISBN: 978-1-259-87297-6 (cit. on p. 180).
- [343] "Proceedings of the *International Workshop on the Software Process and Software Environments*, 1985 [228]". *ACM SIGSOFT Software Engineering Notes* 11(4), June 1985. New York, NY, USA: Association for Computing Machinery (ACM). ISSN: 0163-5948. URL: <https://dl.acm.org/action/showFmPdf?doi=10.1145%2F12944> (visited on 2025-10-07) (cit. on pp. 381, 394).
- [344] *Programming Languages – C, Working Document of SC22/WG14*. International Standard ISO/IEC9899:2017 C17 Ballot N2176. Geneva, Switzerland: International Organization for Standardization (ISO) and International Electrotechnical Commission (IEC), Nov. 2017. URL: <https://files.lhmouse.com/standards/ISO%20C%20N2176.pdf> (visited on 2024-06-29) (cit. on p. 368).
- [345] "psql – PostgreSQL Interactive Terminal". In: *PostgreSQL Documentation*. 17.4. The PostgreSQL Global Development Group (PGDG), Feb. 20, 2025. Chap. Part VI. Reference. URL: <https://www.postgresql.org/docs/17/app-psql.html> (visited on 2025-06-07) (cit. on pp. 95, 96).
- [346] *Quality Management Systems – Guidelines for the Application of ISO 9001 in Local Government*. International Standard ISO 18091:2019, Edition 2. Geneva, Switzerland: International Organization for Standardization (ISO), 2019. See [347] (cit. on p. 189).
- [347] *Quality Management Systems – Requirements*. International Standard ISO 9001:2015, Edition 5. Geneva, Switzerland: International Organization for Standardization (ISO), 2015 (cit. on pp. 189, 402).
- [348] Saty Raghavachary. *CSCI 585: Database Systems*. Los Angeles, CA, USA: University of Southern California (UCS), Spr. 2024. URL: <https://bytes.usc.edu/cs585/s24-d-a-t-aaa/lectures> (visited on 2025-04-06) (cit. on p. 16).
- [349] Saty Raghavachary. "ER". In: *CSCI 585: Database Systems*. Los Angeles, CA, USA: University of Southern California (UCS), Spr. 2024. URL: <https://bytes.usc.edu/cs585/s24-d-a-t-aaa/lectures/ER/slides.html> (visited on 2025-04-06) (cit. on pp. 213, 215, 216).
- [350] Sanatan Rai and George Vairaktarakis. "NP-Complete Problems and Proof Methodology". In: *Encyclopedia of Optimization*. Ed. by Christodoulos A. Floudas and Panos Miltiades Pardalos. 2nd ed. Boston, MA, USA: Springer, Sept. 2008, pp. 2675–2682. ISBN: 978-0-387-74758-3. doi:10.1007/978-0-387-74759-0_462 (cit. on p. 372).
- [351] *RAMAC: The first random-access disk drive revolutionized how businesses use computers and set the stage for everything from space flight to e-commerce*. IBM Heritage. Armonk, NY, USA: International Business Machines Corporation (IBM). URL: <https://www.ibm.com/history/ramac> (visited on 2025-01-09) (cit. on p. 9).
- [352] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. 3rd ed. New York, NY, USA: McGraw-Hill, Aug. 2002. ISBN: 978-0-07-246563-1 (cit. on p. 17).
- [353] Sebastian Raschka, Yuxi Liu, and Vahid Mirjalili. *Machine Learning with PyTorch and Scikit-learn*. Birmingham, England, UK: Packt Publishing Ltd, Feb. 2022. ISBN: 978-1-80181-931-2 (cit. on pp. 139, 373, 374).
- [354] Abhishek Ratan, Eric Chou, Pradeeban Kathiravelu, and Dr. M.O. Faruque Sarker. *Python Network Programming*. Birmingham, England, UK: Packt Publishing Ltd, Jan. 2019. ISBN: 978-1-78883-546-6 (cit. on pp. 12, 368).
- [355] Federico Razzoli. *Mastering MariaDB*. Birmingham, England, UK: Packt Publishing Ltd, Sept. 2014. ISBN: 978-1-78398-154-0 (cit. on pp. 14, 371).

- [356] “`re` – Regular Expression Operations”. In: *Python 3 Documentation. The Python Standard Library*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. URL: <https://docs.python.org/3/library/re.html#module-re> (visited on 2024-11-01) (cit. on p. 373).
- [357] Robert W. Rector, ed. *1965 Fall Joint Computer Conference (AFIPS'1965, Fall, Part 1)*. Nov. 30–Dec. 1, 1965, Las Vegas, NV, USA. New York, NY, USA: Association for Computing Machinery (ACM), 1965. ISBN: 978-1-4503-7885-7. doi:10.1145/1463891.
- [358] Mike Reichardt, Michael Gundall, and Hans D. Schotten. “Benchmarking the Operation Times of NoSQL and MySQL Databases for Python Clients”. In: *47th Annual Conference of the IEEE Industrial Electronics Society (IECON'2021)*. Oct. 13–15, 2021, Toronto, ON, Canada. Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers (IEEE), 2021, pp. 1–8. ISSN: 2577-1647. ISBN: 978-1-6654-3554-3. doi:10.1109/IECON48115.2021.9589382 (cit. on pp. 14, 372).
- [359] Eric Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.3*. Request for Comments (RFC) 8446. Wilmington, DE, USA: Internet Engineering Task Force (IETF), Aug. 2018. URL: <https://www.ietf.org/rfc/rfc8446.txt> (visited on 2025-02-05) (cit. on p. 375).
- [360] “Returning Data from Modified Rows”. In: *PostgreSQL Documentation*. 17.4. The PostgreSQL Global Development Group (PGDG), Feb. 20, 2025. Chap. 6.4. URL: <https://www.postgresql.org/docs/17/dml-returning.html> (visited on 2025-04-21) (cit. on pp. 279, 306, 345, 346).
- [361] “`RETURNING`”. In: *SQLite*. Charlotte, NC, USA: Hipp, Wyrick & Company, Inc. (Hwaci), May 8, 2024. URL: https://sqlite.org/lang_returning.html (visited on 2025-04-24) (cit. on pp. 135–137, 279, 306).
- [362] Luke Reynolds. “Comparison of Major Linux Package Management Systems”. In: *LinuxConfig.org*. Sydney, NSW, Australia: TOSID Group Pty Ltd, Jan. 21, 2025. URL: <https://linuxconfig.org/comparison-of-major-linux-package-management-systems> (visited on 2025-04-16) (cit. on p. 77).
- [363] Mark Richards and Neal Ford. *Fundamentals of Software Architecture: An Engineering Approach*. Sebastopol, CA, USA: O'Reilly Media, Inc., Jan. 2020. ISBN: 978-1-4920-4345-4 (cit. on pp. 12, 368).
- [364] Lawrence G. Roberts. “The ARPANET & Computer Networks”. In: *ACM Conference on The History of Personal Workstations (HPW'1986)*. Jan. 9–10, 1986, Palo Alto, CA, USA. New York, NY, USA: Association for Computing Machinery (ACM), 1986, pp. 51–58. ISBN: 978-0-89791-176-4. doi:10.1145/12178.12182 (cit. on p. 12).
- [365] Christopher Rogers. *Design Made Easy with Inkscape 1.3. A practical guide to your journey from beginner to pro-level vector illustration*. Birmingham, England, UK: Packt Publishing Ltd, Apr. 2023. ISBN: 978-1-80107-877-1 (cit. on p. 370).
- [366] Ernest E. Rothman, Rich Rosen, and Brian Jepson. *Mac OS X for Unix Geeks*. 4th ed. Sebastopol, CA, USA: O'Reilly Media, Inc., Sept. 2008. ISBN: 978-0-596-52062-5 (cit. on pp. 86, 371).
- [367] Quentin Rouland, Stojanche Gjorcheski, and Jason Jaskolka. “Eliciting a Security Architecture Requirements Baseline from Standards and Regulations”. In: *31st IEEE International Requirements Engineering Conference (RE'2023), Workshops*. Sept. 4–5, 2023. Ed. by Kurt Schneider, Fabiano Dalpiaz, and Jennifer Horkoff. Hannover, Niedersachsen, Germany: Institute of Electrical and Electronics Engineers (IEEE), 2023, pp. 224–229. ISSN: 2770-6826. ISBN: 979-8-3503-2692-5. doi:10.1109/REW57809.2023.00045 (cit. on pp. 187, 188).
- [368] “Row and Array Comparisons”. In: *PostgreSQL Documentation*. 17.4. The PostgreSQL Global Development Group (PGDG), Feb. 20, 2025. Chap. 9.25. URL: <https://www.postgresql.org/docs/current/functions-comparisons.html> (visited on 2025-05-08) (cit. on pp. 334, 335).
- [369] Winston Walker Royce. “Managing the Development of Large Software Systems”. In: *Papers Presented at the Western Electronic Show and Convention (IEEE WESCON'1970)*. Aug. 25–28, 1970, Los Angeles, CA, USA. Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers (IEEE), 1970, pp. 328–388. ISSN: 1095-791X. URL: <https://blog.jbrains.ca/assets/articles/royce1970.pdf> (visited on 2025-10-06) (cit. on pp. 180, 181).

- [370] Per Runeson. "A Survey of Unit Testing Practices". *IEEE Software* 23(4):22–29, July–Aug. 2006. Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers (IEEE). ISSN: 0740-7459. doi:10.1109/MS.2006.91 (cit. on p. 375).
- [371] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach (AIMA)*. 4th ed. Hoboken, NJ, USA: Pearson Education, Inc. ISBN: 978-1-292-40113-3. URL: <https://aima.cs.berkeley.edu> (visited on 2024-06-27) (cit. on p. 368).
- [372] Randall Rustin, ed. *Data Base Systems: Courant Computer Science Symposium 6*. May 24–25, 1971, New York, NY, USA. Englewood Cliffs, NJ, USA: Prentice-Hall, 1972. ISBN: 978-0-13-196741-0 (cit. on p. 385).
- [373] Ahmad Sahar. *iOS 26 Programming for Beginners*. 10th ed. Birmingham, England, UK: Packt Publishing Ltd, Nov. 2025. ISBN: 978-1-80602-393-6 (cit. on p. 376).
- [374] Stephen R. Schach. *Object-Oriented Software Engineering*. New York, NY, USA: McGraw-Hill, Sept. 2007. ISBN: 978-0-07-352333-0 (cit. on pp. 181, 182, 372, 373).
- [375] Stephan Scheuermann. "SQL Injection". In: *1st Kassel Student Workshop on Security in Distributed Systems (KaSWoSDS'2008)*. Feb. 13, 2008, Kassel, Hessen, Germany. Ed. by Thomas Weise (汤卫思) and Philipp Andreas Baer. Vol. 2008-1 of Kasseler Informatikschriften (KIS). Kassel, Hessen, Germany: Universität Kassel, Universitätsbibliothek, Apr. 14, 2008. Chap. 3, pp. 35–49. URL: <https://kobra.uni-kassel.de/items/a6134d61-d5c3-4cb8-804f-bbb89a3f59a7> (visited on 2025-08-23) (cit. on pp. 142, 143, 374).
- [376] "Why not use Double or Float to represent currency?" In: *Stack Overflow*. Ed. by Jan Schultke. New York, NY, USA: Stack Exchange Inc., Sept. 16, 2010–Jan. 13, 2025. URL: <https://stackoverflow.com/questions/3730019> (visited on 2025-02-27) (cit. on pp. 105, 365).
- [377] Heinz Schweppe and Manuel Scholz. "Conceptual Database Design: Integrity Constraints and Modeling Patterns". In: *Einführung in die Datenbanksysteme. Datenbanken für die Bioinformatik*. Berlin, Germany: Freie Universität Berlin, Apr.–Oct. 2005. Chap. 2.3/2.4. URL: <https://www.inf.fu-berlin.de/lehre/SS05/19517-V/FolienEtc/dbs05-03-ConceptualModeling2-2.pdf> (visited on 2025-03-27) (cit. on pp. 193, 211, 213, 214).
- [378] Heinz Schweppe and Manuel Scholz. "Conceptual Database Design: Requirement Analysis and Modeling Languages". In: *Einführung in die Datenbanksysteme. Datenbanken für die Bioinformatik*. Berlin, Germany: Freie Universität Berlin, Apr.–Oct. 2005. Chap. 2.1/2.2. URL: <https://www.inf.fu-berlin.de/lehre/SS05/19517-V/FolienEtc/dbs05-02-ConceptualModeling1-2.pdf> (visited on 2025-03-24) (cit. on pp. 184, 185, 193, 195).
- [379] Heinz Schweppe and Manuel Scholz. *Einführung in die Datenbanksysteme. Datenbanken für die Bioinformatik*. Berlin, Germany: Freie Universität Berlin, Apr.–Oct. 2005. URL: <https://www.inf.fu-berlin.de/lehre/SS05/19517-V> (visited on 2025-01-08) (cit. on p. 16).
- [380] Heinz Schweppe and Manuel Scholz. "Introduction". In: *Einführung in die Datenbanksysteme. Datenbanken für die Bioinformatik*. Berlin, Germany: Freie Universität Berlin, Apr.–Oct. 2005. Chap. 1. URL: <https://www.inf.fu-berlin.de/lehre/SS05/19517-V/FolienEtc/dbs05-01-Intro-2.pdf> (visited on 2025-01-08) (cit. on pp. 6, 179, 184, 185, 226).
- [381] Heinz Schweppe and Manuel Scholz. "Normalization: Quality of Relational Designs". In: *Einführung in die Datenbanksysteme. Datenbanken für die Bioinformatik*. Berlin, Germany: Freie Universität Berlin, Apr.–Oct. 2005. Chap. 5. URL: <https://www.inf.fu-berlin.de/lehre/SS05/19517-V/FolienEtc/dbs05-07-FA-1-2.pdf> (visited on 2025-05-06) (cit. on pp. 194, 365).
- [382] Heinz Schweppe and Manuel Scholz. "Schema Definition with SQL / DDL (II)". In: *Einführung in die Datenbanksysteme. Datenbanken für die Bioinformatik*. Berlin, Germany: Freie Universität Berlin, Apr.–Oct. 2005. Chap. 4. URL: <https://www.inf.fu-berlin.de/lehre/SS05/19517-V/FolienEtc/dbs05-06-DDLSQL-2-2.pdf> (visited on 2025-05-15) (cit. on pp. 228, 342, 356, 374).
- [383] Heinz Schweppe and Manuel Scholz. "Schema Design: Logical Design using the Relational Data Model". In: *Einführung in die Datenbanksysteme. Datenbanken für die Bioinformatik*. Berlin, Germany: Freie Universität Berlin, Apr.–Oct. 2005. Chap. 3. URL: <https://www.inf.fu-berlin.de/lehre/SS05/19517-V/FolienEtc/dbs05-04-RDM1-2.pdf> (visited on 2025-04-04) (cit. on pp. 203, 228, 229, 231).

- [384] Matthias Sedlmeier and Martin Gogolla. "Model Driven ActiveRecord with *yEd*". In: *25th International Conference on Information Modelling and Knowledge Bases XXVII (EJC'2015)*. June 8–12, 2015, Maribor, Štajerska, Podravska, Slovenia. Ed. by Tatjana Welzer, Hannu Jaakkola, Bernhard Thalheim, Yasushi Kiyoki, and Naofumi Yoshida. Vol. 280 of Frontiers in Artificial Intelligence and Applications. Amsterdam, The Netherlands: IOS Press BV, 2015, pp. 65–76. ISSN: 0922-6389. ISBN: 978-1-61499-610-1. doi:10.3233/978-1-61499-611-8-65 (cit. on pp. iv, 20, 64, 195, 200, 367, 376).
- [385] Winfried Seimert. *LibreOffice 7.3 – Praxiswissen für Ein- und Umsteiger*. Blaufelden, Schwäbisch Hall, Baden-Württemberg, Germany: mitp Verlags GmbH & Co. KG, Apr. 2022. ISBN: 978-3-7475-0504-5 (cit. on pp. iv, 2, 15, 20, 45, 101, 146, 170, 367, 371, 372).
- [386] "SELECT". In: *PostgreSQL Documentation*. 17.4. The PostgreSQL Global Development Group (PGDG), Feb. 20, 2025. Chap. Part VI. Reference. URL: <https://www.postgresql.org/docs/17/sql-select.html> (visited on 2025-05-08) (cit. on pp. 106, 108, 338).
- [387] Syamal K. Sen and Ravi P. Agarwal. "Existence of year zero in astronomical counting is advantageous and preserves compatibility with significance of AD, BC, CE, and BCE". In: *Zero – A Landmark Discovery, the Dreadful Void, and the Ultimate Mind*. Amsterdam, The Netherlands: Elsevier B.V., 2016. Chap. 5.5, pp. 94–95. ISBN: 978-0-08-100774-7. doi:10.1016/C2015-0-02299-7 (cit. on p. 368).
- [388] "Sequence Manipulation Functions". In: *PostgreSQL Documentation*. 17.4. The PostgreSQL Global Development Group (PGDG), Feb. 20, 2025. Chap. 9.17. URL: <https://www.postgresql.org/docs/17/functions-sequence.html> (visited on 2025-04-23) (cit. on pp. 284, 306).
- [389] "SET CONSTRAINTS". In: *PostgreSQL Documentation*. 17.4. The PostgreSQL Global Development Group (PGDG), Feb. 20, 2025. Chap. Part VI. Reference. URL: <https://www.postgresql.org/docs/17/sql-set-constraints.html> (visited on 2025-04-10) (cit. on p. 221).
- [390] Norbert Seyff, Florian Graf, Paul Grünbacher, and Neil A. M. Maiden. "The Mobile Scenario Presenter: A Tool for in situ Requirements Discovery with Scenarios". In: *15th IEEE International Requirements Engineering Conference (RE'2007)*. Oct. 15–19, 2007, New Delhi, India. Los Alamitos, CA, USA: IEEE Computer Society, 2007, pp. 365–366. ISSN: 1090-705X. ISBN: 978-0-7695-2935-6. doi:10.1109/RE.2007.27 (cit. on p. 188).
- [391] Yakov Shafranovich. *Common Format and MIME Type for Comma-Separated Values (CSV) Files*. Request for Comments (RFC) 4180. Wilmington, DE, USA: Internet Engineering Task Force (IETF), Oct. 2005. URL: <https://www.ietf.org/rfc/rfc4180.txt> (visited on 2025-02-05) (cit. on pp. 2, 368).
- [392] Shai Shalev-Shwartz and Shai Ben-David. *Understanding Machine Learning: From Theory to Algorithms*. Cambridge, England, UK: Cambridge University Press (CUP), July 2014. ISBN: 978-1-107-05713-5. URL: <http://www.cs.huji.ac.il/~shais/UnderstandingMachineLearning> (visited on 2024-06-27) (cit. on p. 372).
- [393] Yuriy Shamshin. "Conceptual Database Model. Entity Relationship Diagram (ERD)". In: *Databases*. Riga, Latvia: ISMA University of Applied Sciences, May 2024. Chap. 04. URL: https://dbs.academy.lv/lection/dbs_LS04EN_erd.pdf (visited on 2025-03-29) (cit. on pp. 193, 195, 203, 211, 213, 369).
- [394] Yuriy Shamshin. *Databases*. Riga, Latvia: ISMA University of Applied Sciences, May 2024. URL: <https://dbs.academy.lv> (visited on 2025-01-11) (cit. on p. 16).
- [395] Yuriy Shamshin. "Logical Data Models. Relation Model. Relation Algebra". In: *Databases*. Riga, Latvia: ISMA University of Applied Sciences, May 2024. Chap. 05. URL: https://dbs.academy.lv/lection/dbs_LS05EN_rm.pdf (visited on 2025-04-10) (cit. on pp. 194, 229, 230).
- [396] Yuriy Shamshin. "Mapping ER Diagrams to Relation Data Model". In: *Databases*. Riga, Latvia: ISMA University of Applied Sciences, May 2024. Chap. 06. URL: https://dbs.academy.lv/lection/dbs_LS06EN_er2rm.pdf (visited on 2025-04-20) (cit. on pp. 260, 272, 281, 307).
- [397] Yuriy Shamshin. "Normalization". In: *Databases*. Riga, Latvia: ISMA University of Applied Sciences, May 2024. Chap. 07a. URL: https://dbs.academy.lv/lection/dbs_LS07ENa_normalization.pdf (visited on 2025-05-03) (cit. on pp. 330, 372).

- [398] Yury Shamshin. "RDM Normalization. Data Anomalies. Functional Dependency. Normal Forms." In: *Databases*. Riga, Latvia: ISMA University of Applied Sciences, May 2024. Chap. 07. URL: https://dbs.academy.lv/lection/dbs_LS07EN_normalization.pdf (visited on 2025-05-03) (cit. on pp. 330, 342, 346, 372).
- [399] Yury Shamshin. "The History of *Databases*". In: *Databases*. Riga, Latvia: ISMA University of Applied Sciences, May 2024. Chap. 02a. URL: https://dbs.academy.lv/lection/dbs_LS02ENa_hist.pdf (visited on 2025-01-11) (cit. on p. 8).
- [400] Mingtao Shi. "Documenting Software Requirements Specification: A Revisit". *Computer and Information Science* 3(1):17–19, Feb. 2010. Richmond Hill, ON, Canada: Canadian Center of Science and Education (CCSE). ISSN: 1913-8989. doi:[10.5539/CIS.V3N1P17](https://doi.org/10.5539/CIS.V3N1P17) (cit. on pp. 188, 374).
- [401] Ellen Siever, Stephen Figgins, Robert Love, and Arnold Robbins. *Linux in a Nutshell*. 6th ed. Sebastopol, CA, USA: O'Reilly Media, Inc., Sept. 2009. ISBN: 978-0-596-15448-6 (cit. on p. 371).
- [402] Abraham "Avi" Silberschatz, Henry F. "Hank" Korth, and S. Sudarshan. *Database System Concepts*. 7th ed. New York, NY, USA: McGraw-Hill, Mar. 2019. ISBN: 978-0-07-802215-9 (cit. on p. 16).
- [403] Bryan Sills, Brian Gardner, Kristin Marsicano, and Chris Stewart. *Android Programming: The Big Nerd Ranch Guide*. 5th ed. Reading, MA, USA: Addison-Wesley Professional, May 2022. ISBN: 978-0-13-764579-4 (cit. on p. 368).
- [404] Anna Skoulikari. *Learning Git*. Sebastopol, CA, USA: O'Reilly Media, Inc., May 2023. ISBN: 978-1-0981-3391-7 (cit. on p. 370).
- [405] Ioannis Skoulis, Panos Vassiliadis, and Apostolos V. Zarras. "Open-Source *Databases*: Within, Outside, or Beyond Lehman's Laws of Software Evolution?" In: *26th International Conference on Advanced Information Systems Engineering (CAiSE'2014)*. June 16–20, 2014, Thessaloniki, Central Macedonia, Greece. Ed. by Matthias Jarke, John Mylopoulos, Christoph Quix, Colette Rolland, Yannis Manolopoulos, Haralambos Mouratidis, and Jennifer Horkoff. Vol. 8484 of Lecture Notes in Computer Science (LNCS). Cham, Switzerland: Springer, 2014, pp. 379–393. ISSN: 0302-9743. ISBN: 978-3-319-07880-9. doi:[10.1007/978-3-319-07881-6_26](https://doi.org/10.1007/978-3-319-07881-6_26) (cit. on p. 179).
- [406] Drew Smith. *Modern Apple Platform Administration – macOS and iOS Essentials* (2025). Birmingham, England, UK: Packt Publishing Ltd, Feb. 2025. ISBN: 978-1-80580-309-6 (cit. on pp. 370, 371).
- [407] Eric V. "ericvsmith" Smith. *Literal String Interpolation*. Python Enhancement Proposal (PEP) 498. Beaverton, OR, USA: Python Software Foundation (PSF), Nov. 6, 2016–Sept. 9, 2023. URL: <https://peps.python.org/pep-0498> (visited on 2024-07-25) (cit. on p. 369).
- [408] John Miles Smith and Philip Yen-Tang Chang. "Optimizing the Performance of a Relational Algebra Database Interface". *Communications of the ACM (CACM)* 18(10):568–579, Oct. 1975. New York, NY, USA: Association for Computing Machinery (ACM). ISSN: 0001-0782. doi:[10.1145/361020.361025](https://doi.org/10.1145/361020.361025) (cit. on pp. 3, 374).
- [409] *Software Engineering*. New York, NY, USA: Stack Exchange Inc. URL: <https://softwareengineering.stackexchange.com> (visited on 2025-02-27).
- [410] Il-Yeol Song and Kristin Froehlich. "Appendix A: A Practical Guide to Entity-Relationship Modeling". In: Jan. 19, 2000–May 20, 2005. URL: <https://cci.drexel.edu/faculty/song/courses/info%20605/appendix/AppendixA.PDF> (visited on 2025-04-05) (cit. on p. 214).
- [411] *SQLite*. Charlotte, NC, USA: Hipp, Wyck & Company, Inc. (Hwaci), 2025. URL: <https://sqlite.org> (visited on 2025-04-24) (cit. on p. 374).
- [412] "SQL Commands". In: *PostgreSQL Documentation*. 17.4. The PostgreSQL Global Development Group (PGDG), Feb. 20, 2025. Chap. Part VI. Reference. URL: <https://www.postgresql.org/docs/17/sql-commands.html> (visited on 2025-02-25) (cit. on pp. 89, 93, 94, 374).
- [413] "SQL Dump". In: *PostgreSQL Documentation*. 17.4. The PostgreSQL Global Development Group (PGDG), Feb. 20, 2025. Chap. 25.1. URL: <https://www.postgresql.org/docs/17/backup-dump.html> (visited on 2025-03-05) (cit. on p. 171).

- [414] "SQL Key Words". In: *PostgreSQL Documentation*. 17.4. The PostgreSQL Global Development Group (PGDG), Feb. 20, 2025. Chap. Appendix C. URL: <https://www.postgresql.org/docs/17/sql-keywords-appendix.html> (visited on 2025-07-04) (cit. on p. 122).
- [415] Pradeep Kumar Srinivasan and Graham Bleaney. *Arbitrary Literal String Type*. Python Enhancement Proposal (PEP) 675. Beaverton, OR, USA: Python Software Foundation (PSF), Nov. 30, 2021–Feb. 7, 2022. URL: <https://peps.python.org/pep-0675> (visited on 2025-03-04) (cit. on pp. 141, 143, 374).
- [416] *Stack Overflow*. New York, NY, USA: Stack Exchange Inc. URL: <https://stackoverflow.com> (visited on 2025-02-27).
- [417] "Stack Overflow 2024 Developer Survey". In: *Stack Overflow*. New York, NY, USA: Stack Exchange Inc., May–June 2024. URL: <https://survey.stackoverflow.co/2024> (visited on 2025-06-01) (cit. on pp. 13–15, 88).
- [418] Bogdan Stashchuk. *SSL Complete Guide 2021: HTTP to HTTPS*. Birmingham, England, UK: Packt Publishing Ltd, May 2021. ISBN: 978-1-83921-150-8 (cit. on pp. 370, 375).
- [419] Ryan K. Stephens and Ronald R. Plew. *Sams Teach Yourself SQL in 21 Days*. 4th ed. Sams Teach Yourself. Indianapolis, IN, USA: SAMS Technical Publishing and Hoboken, NJ, USA: Pearson Education, Inc., Oct. 2002. ISBN: 978-0-672-32451-2 (cit. on pp. 17, 89, 93, 230, 374, 407).
- [420] Ryan K. Stephens, Ronald R. Plew, Bryan Morgan, and Jeff Perkins. *SQL in 21 Tagen. Die Datenbank-Abfragesprache SQL vollständig erklärt (in 14/21 Tagen)*. 6th ed. Burghausen, Bayern, Germany: Markt+Technik Verlag GmbH, Feb. 1998. ISBN: 978-3-8272-2020-2. Translation of [419] (cit. on pp. 17, 89, 93, 230, 374).
- [421] Michael Stonebraker, ed. *The INGRES Papers: Anatomy of a Relational Database System*. Reading, MA, USA: Addison-Wesley Professional, 1986. ISBN: 978-0-201-07185-6 (cit. on p. 12).
- [422] Philip D. Straffin Jr. "Liu Hui and the First Golden Age of Chinese Mathematics". *Mathematics Magazine* 71(3):163–181, June 1998. London, England, UK: Taylor and Francis Ltd. ISSN: 0025-570X. doi:10.2307/2691200. URL: <https://www.researchgate.net/publication/237334342> (visited on 2024-08-10) (cit. on p. 201).
- [423] "String Constants". In: chap. 4.1.2.1. URL: <https://www.postgresql.org/docs/17/sql-syntax-lexical.html#SQL-SYNTAX-STRINGS> (visited on 2025-08-23) (cit. on p. 369).
- [424] "String Functions and Operators". In: *PostgreSQL Documentation*. 17.4. The PostgreSQL Global Development Group (PGDG), Feb. 20, 2025. Chap. 9.4. URL: <https://www.postgresql.org/docs/17/functions-string.html> (visited on 2025-05-06) (cit. on pp. 131, 242, 333).
- [425] *Systems and Software Engineering - Software Life Cycle Processes*. ISO/IEC/IEEE International Standard ISO/IEC/IEEE 12207-2017. Geneva, Switzerland: International Organization for Standardization (ISO), International Electrotechnical Commission (IEC), and New York, NY, USA: Institute of Electrical and Electronics Engineers (IEEE), Nov. 15, 2017 (cit. on p. 188).
- [426] *Systems and Software Engineering -- Life Cycle Processes -- Requirements Engineering, Second Edition*. ISO/IEC/IEEE International Standard ISO/IEC/IEEE 29148:2018(E). Geneva, Switzerland: International Organization for Standardization (ISO), International Electrotechnical Commission (IEC), and New York, NY, USA: Institute of Electrical and Electronics Engineers (IEEE), Nov. 30, 2018. doi:10.1109/IEEEESTD.2018.8559686. Supersedes [219] (cit. on pp. 187–189, 374, 394).
- [427] *Systems and Software Engineering -- System Life Cycle Processes, Second Edition*. ISO/IEC/IEEE International Standard ISO/IEC/IEEE 15288-2023(E). Geneva, Switzerland: International Organization for Standardization (ISO), International Electrotechnical Commission (IEC), and New York, NY, USA: Institute of Electrical and Electronics Engineers (IEEE), May 16, 2023. doi:10.1109/IEEEESTD.2023.10123367 (cit. on p. 188).
- [428] Mana Takahashi, Shoko Azuma, and Tokyo, Japan: Trend-Pro Co, Ltd. *The Manga Guide to Databases*. San Francisco, CA, USA: No Starch Press, Jan. 2009. ISBN: 978-1-59327-190-9 (cit. on p. 17).
- [429] Sinclair Target. *The IBM 029 Card Punch*. New York, NY, USA, June 23, 2018. URL: <https://twobithistory.org/2018/06/23/ibm-029-card-punch.html> (visited on 2025-05-29) (cit. on p. 9).

- [430] Sherif M. Tawfik and Marwa M. Abd-Elghany. "Investigating Software Requirements Through Developed Questionnaires To Satisfy The Desired Quality Systems (Security Attribute Example)". In: *2008 International Conference on Systems, Computing Sciences and Software Engineering (SCSS'2008), part of the International Joint Conferences on Computer, Information, and Systems Sciences, and Engineering (CISSE'2008)*. Dec. 5–13, 2008, Bridgeport, CT, USA. Ed. by Khaled M. Elleithy. Vol. II: Advanced Techniques in Computing Sciences and Software Engineering. Dordrecht, The Netherlands: Springer, Dec. 2009, pp. 245–250. ISBN: 978-90-481-3659-9. doi:10.1007/978-90-481-3660-5_41 (cit. on p. 188).
- [431] Allen Taylor. *Introducing SQL and Relational Databases*. New York, NY, USA: Apress Media, LLC, Sept. 2018. ISBN: 978-1-4842-3841-7 (cit. on pp. 3, 17, 89, 93, 374).
- [432] Robert W. Taylor and Randall L. Frank. "CODASYL Data-Base Management Systems". *ACM Computing Surveys (CSUR)* 8(1):67–103, Mar. 1976. New York, NY, USA: Association for Computing Machinery (ACM). ISSN: 0360-0300. doi:10.1145/356662.356666 (cit. on pp. 11, 226).
- [433] Alkin Tezuyosal and Ibrar Ahmed. *Database Design and Modeling with PostgreSQL and MySQL*. Birmingham, England, UK: Packt Publishing Ltd, July 2024. ISBN: 978-1-80323-347-5 (cit. on pp. iv, 14, 17, 20, 25, 88, 367, 372, 373).
- [434] *The Document Foundation Wiki: ReleasePlan*. Berlin, Germany: The Document Foundation, Jan. 21, 2011–Nov. 22, 2024. URL: <https://wiki.documentfoundation.org/ReleasePlan> (visited on 2025-03-21) (cit. on p. 179).
- [435] The Editors of Encyclopaedia Britannica, ed. *Encyclopaedia Britannica*. Chicago, IL, USA: Encyclopædia Britannica, Inc.
- [436] "The Evolution of Microsoft SQL Server". In: Feb. 15, 2025. URL: <https://peter-whyte.com/2025/02/the-evolution-of-microsoft-sql-server> (visited on 2025-06-03) (cit. on p. 15).
- [437] *The IBM Punched Card: The paper on-ramp to the Information Age once held most of the world's data*. IBM Heritage. Armonk, NY, USA: International Business Machines Corporation (IBM). URL: <https://www.ibm.com/history/punched-card> (visited on 2025-01-08) (cit. on p. 8).
- [438] *The JSON Data Interchange Syntax*. Standard ECMA-404, 2nd Edition. Geneva, Switzerland: Ecma International, Dec. 2017. URL: <https://ecma-international.org/publications-and-standards/standards/ecma-404> (visited on 2024-12-15) (cit. on pp. 2, 370).
- [439] *Python 3 Documentation. The Python Language Reference*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. URL: <https://docs.python.org/3/reference> (visited on 2025-04-27).
- [440] *The Python Package Index (PyPI)*. Beaverton, OR, USA: Python Software Foundation (PSF), 2024. URL: <https://pypi.org> (visited on 2024-08-17) (cit. on p. 373).
- [441] *Python 3 Documentation. The Python Standard Library*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. URL: <https://docs.python.org/3/library> (visited on 2025-04-27).
- [442] *Python 3 Documentation. The Python Tutorial*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. URL: <https://docs.python.org/3/tutorial> (visited on 2025-04-26).
- [443] "Literals". In: *Static Typing with Python*. Ed. by The Python Typing Team. Beaverton, OR, USA: Python Software Foundation (PSF), 2021. URL: <https://typing.python.org/en/latest/spec/literal.html> (visited on 2025-08-29) (cit. on p. 371).
- [444] *The Ubuntu Lifecycle and Release Cadence*. London, England, UK: Canonical Ltd., Oct. 2024. URL: <https://ubuntu.com/about/release-cycle> (visited on 2025-03-21) (cit. on pp. 179, 371).
- [445] *The Unicode Standard, Version 15.1: Archived Code Charts*. South San Francisco, CA, USA: The Unicode Consortium, Aug. 25, 2023. URL: <https://www.unicode.org/Public/15.1.0/charts/CodeCharts.pdf> (visited on 2024-07-26) (cit. on p. 375).

- [446] George K. Thiruvathukal, Konstantin Läufer, and Benjamin Gonzalez. “Unit Testing Considered Useful”. *Computing in Science & Engineering* 8(6):76–87, Nov.–Dec. 2006. Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers (IEEE). ISSN: 1521-9615. doi:10.1109/MCSE.2006.124. URL: <https://www.researchgate.net/publication/220094077> (visited on 2024-10-01) (cit. on p. 375).
- [447] Kristian Torp, Christian S. Jensen, and Richard T. Snodgrass. “Effective Timestamping in Databases”. 8(3-4):267–288, Feb. 2000. doi:10.1007/S007780050008. URL: <https://peo ple.cs.aau.dk/~csj/Thesis/pdf/chapter40.pdf> (visited on 2025-05-01) (cit. on p. 375).
- [448] Linus Torvalds. “The Linux Edge”. *Communications of the ACM (CACM)* 42(4):38–39, Apr. 1999. New York, NY, USA: Association for Computing Machinery (ACM). ISSN: 0001-0782. doi:10.1145/299157.299165 (cit. on pp. 21, 371).
- [449] Sherwin John C. Tragura. *Mastering Flask Web and API Development*. Birmingham, England, UK: Packt Publishing Ltd, Aug. 2024. ISBN: 978-1-83763-322-7 (cit. on p. 369).
- [450] “Transactions”. In: *PostgreSQL Documentation*. 17.4. The PostgreSQL Global Development Group (PGDG), Feb. 20, 2025. Chap. 3.4. URL: <https://www.postgresql.org/docs/17/tutorial-transactions.html> (visited on 2025-04-21) (cit. on p. 279).
- [451] Kevin Treu. *CSC-341: Database Management Systems*. Greenville, SC, USA: Furman University, Spr. 2025. URL: <https://cs.furman.edu/~ktreu/csc341> (visited on 2025-04-05) (cit. on p. 16).
- [452] Kevin Treu. “Entity Relationship Modeling”. In: *CSC-341: Database Management Systems*. Greenville, SC, USA: Furman University, Spr. 2025. Chap. 4. URL: <https://cs.furman.edu/~ktreu/csc341/lectures/chapter04.pdf> (visited on 2025-04-05) (cit. on pp. 215, 216).
- [453] Dennis Tsichritzis and Anthony Klug. “The ANSI/X3/SPARC DBMS Framework Report of the Study Group on Database Management Systems”. *Information Systems: Databases: Their Creation, Management and Utilization* 3(3):173–191, 1978. Oxford, Oxfordshire, England, UK: Pergamon Press Ltd., now Amsterdam, The Netherlands: Elsevier B.V. ISSN: 0306-4379. Also published as [7] (cit. on pp. 6, 379).
- [454] Mariot Tsitoara. *Beginning Git and GitHub: Version Control, Project Management and Teamwork for the New Developer*. New York, NY, USA: Apress Media, LLC, Mar. 2024. ISBN: 979-8-8688-0215-7 (cit. on pp. 370, 376).
- [455] Berik I. Tuleuov and Ademi B. Ospanova. “Minimal Systems”. In: *Beginning C++ Compilers: An Introductory Guide to Microsoft C/C++ and MinGW Compilers*. Berkeley, CA, USA: Apress Media, LLC, Jan. 2024. Chap. 9, pp. 75–83. ISBN: 978-1-4842-9562-5. doi:10.1007/978-1-4842-9563-2_8 (cit. on pp. 77, 372).
- [456] Christina Tyler. “Clay Tablets Reveal Accounting Answers”. *The Gazette* 10(36), Oct. 1, 1999. URL: <https://www.loc.gov/collections/cuneiform-tablets/articles-and-essays/clay-tablets-reveal-accounting-answers> (visited on 2025-01-08). Also part of [112] (cit. on pp. 8, 386).
- [457] Laurie A. Ulrich and Ken Cook. *Access For Dummies*. Hoboken, NJ, USA: For Dummies (Wiley), Dec. 2021. ISBN: 978-1-119-82908-9 (cit. on pp. 15, 18, 45, 146, 371).
- [458] *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. 3rd ed. The Addison-Wesley Object Technology Series. Reading, MA, USA: Addison-Wesley Professional, Sept. 2003. ISBN: 978-0-321-19368-1 (cit. on pp. 219, 375).
- [459] *UML Notation Guide*. Version 1.1. Santa Clara, CA, USA: Rational Software Corporation, Redmond, WA, USA: Microsoft Corporation, Palo Alto, CA, USA: Hewlett-Packard Company, Redwood Shores, CA, USA: Oracle Corporation, Dallas, TX, USA: Sterling Software, Ottawa, ON, Canada: MCI Systemhouse Corporation, Blue Bell, PA, USA: Unisys Corporation, Blue Bell, PA, USA: ICON Computing, Santa Clara, CA, USA: IntelliCorp, Burlington, MA, USA: i-Logix, Armonk, NY, USA: International Business Machines Corporation (IBM), Kanata, ON, Canada: ObjectTime Limited, Chicago, IL, USA: Platinum Technology Inc., Boston, MA, USA: Ptech Inc., Orlando, FL, USA: Taskon A/S, Paoli, PA, USA: Reich Technologies, and Paris, Île-de-France, France: Softeam, Sept. 1, 1997. URL: <https://web.cse.msu.edu/~cse870/Materials/uml-notation-guide-9-97.pdf> (visited on 2025-03-30) (cit. on pp. 195, 219, 375).

- [460] *Unicode®15.1.0*. South San Francisco, CA, USA: The Unicode Consortium, Sept. 12, 2023. ISBN: 978-1-936213-33-7. URL: <https://www.unicode.org/versions/Unicode15.1.0> (visited on 2024-07-26) (cit. on p. 375).
- [461] “**UPDATE**”. In: *PostgreSQL Documentation*. 17.4. The PostgreSQL Global Development Group (PGDG), Feb. 20, 2025. Chap. Part VI. Reference. URL: <https://www.postgresql.org/docs/17/sql-update.html> (visited on 2025-03-07) (cit. on pp. 134, 263).
- [462] “Use of Personal ID Number as Passport Number”. In: *Technical Advisory Group on Machine Readable Travel Documents. Fifteenth Meeting*. May 17–21, 2004, Montreal, QC, Canada. Vol. TAG-MRTD/15 WP/19 28/4/04. Montreal, QC, Canada: International Civil Aviation Organization (ICAO), pp. 1–2. URL: https://www.icao.int/Meetings/TAG-MRTD/Documents/Tag-Mrted-15/TagMrtd15_WP019_en.pdf (visited on 2025-04-03) (cit. on p. 210).
- [463] “What does standard SQL or any of the non-PostgreSQL SQLs do instead of **RETURNING**?”. In: *Database Administrators*. Ed. by user210271. New York, NY, USA: Stack Exchange Inc., June 7–8, 2020. URL: <https://dba.stackexchange.com/questions/268664> (visited on 2025-04-23) (cit. on pp. 135–137, 279).
- [464] Marat Valiev, Bogdan Vasilescu, and James D. Herbsleb. “Ecosystem-Level Determinants of Sustained Activity in Open-Source Projects: A Case Study of the PyPI Ecosystem”. In: *ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/SIGSOFT FSE’2018)*. Nov. 4–9, 2018, Lake Buena Vista, FL, USA. Ed. by Gary T. Leavens, Alessandro F. Garcia, and Corina S. Păsăreanu. New York, NY, USA: Association for Computing Machinery (ACM), 2018, pp. 644–655. ISBN: 978-1-4503-5573-5. doi:10.1145/3236024.3236062 (cit. on p. 373).
- [465] Bruce M. Van Horn II and Quan Nguyen. *Hands-On Application Development with PyCharm*. 2nd ed. Birmingham, England, UK: Packt Publishing Ltd, Oct. 2023. ISBN: 978-1-83763-235-0 (cit. on p. 373).
- [466] Guido van Rossum and Łukasz Langa. *Type Hints*. Python Enhancement Proposal (PEP) 484. Beaverton, OR, USA: Python Software Foundation (PSF), Sept. 29, 2014. URL: <https://peps.python.org/pep-0484> (visited on 2024-08-22) (cit. on p. 375).
- [467] Sander van Vugt. *Linux Fundamentals*. 2nd ed. Hoboken, NJ, USA: Pearson IT Certification, June 2022. ISBN: 978-0-13-792931-3 (cit. on p. 371).
- [468] Scott L. Vandenberg. “Conceptual Design using the Entity-Relationship Model”. In: *CSE 594: Database Management Systems*. Seattle, WA, USA: University of Washington, Aut. 1999. URL: <https://courses.cs.washington.edu/courses/csep544/99au/lectures/class2.pdf> (visited on 2025-03-29) (cit. on pp. 193, 195, 213, 214).
- [469] Scott L. Vandenberg. “Course Introduction”. In: *CSE 594: Database Management Systems*. Seattle, WA, USA: University of Washington, Aut. 1999. Chap. 1. URL: <https://courses.cs.washington.edu/courses/csep544/99au/lectures/class1.pdf> (visited on 2025-03-25) (cit. on p. 184).
- [470] Scott L. Vandenberg. *CSE 594: Database Management Systems*. Seattle, WA, USA: University of Washington, Aut. 1999. URL: <https://courses.cs.washington.edu/courses/csep544/99au> (visited on 2025-03-25) (cit. on p. 16).
- [471] Daniele “dvarrazzo” Varrazzo, Federico “fogzot” Di Gregorio, and Jason “jerickso” Erickson. “[Connection Classes]”. In: *Psycopg 3 – PostgreSQL Database Adapter for Python*. London, England, UK: The Psycopg Team, June 21, 2022. URL: <https://www.psycopg.org/psycopg3/docs/api/connections.html> (visited on 2025-03-04) (cit. on pp. 140, 141).
- [472] Daniele “dvarrazzo” Varrazzo, Federico “fogzot” Di Gregorio, and Jason “jerickso” Erickson. “[Cursor Classes]”. In: *Psycopg 3 – PostgreSQL Database Adapter for Python*. London, England, UK: The Psycopg Team, June 21, 2022. URL: <https://www.psycopg.org/psycopg3/docs/api/cursors.html> (visited on 2025-03-04) (cit. on p. 141).
- [473] Daniele “dvarrazzo” Varrazzo, Federico “fogzot” Di Gregorio, and Jason “jerickso” Erickson. *Psycopg*. London, England, UK: The Psycopg Team, 2010–2023. URL: <https://www.psycopg.org> (visited on 2025-02-02) (cit. on pp. iv, 20, 56, 140, 145, 367, 373).

- [474] Daniele “dvarrazzo” Varrazzo, Federico “fogzot” Di Gregorio, and Jason “jerickso” Erickson. *Psycopg 3 – PostgreSQL Database Adapter for Python*. London, England, UK: The Psycopg Team, June 21, 2022. URL: <https://www.psycopg.org/psycopg3/docs> (visited on 2025-03-04).
- [475] Daniele “dvarrazzo” Varrazzo, Federico “fogzot” Di Gregorio, and Jason “jerickso” Erickson. “Static Typing”. In: *Psycopg 3 – PostgreSQL Database Adapter for Python*. London, England, UK: The Psycopg Team, June 21, 2022. URL: <https://www.psycopg.org/psycopg3/docs/advanced/typing.html> (visited on 2025-03-04) (cit. on pp. 143, 374).
- [476] “pacman”. In: *Arch Linux*. Ed. by Judd Vinet, Aaron Griffin, and Levente Polyák. San Jose, CA, USA, Oct. 16, 2005–Apr. 13, 2025. URL: <https://wiki.archlinux.org/title/Pacman> (visited on 2025-04-16) (cit. on p. 77).
- [477] Pauli “pv” Virtanen, Ralf Gommers, Travis E. Oliphant, Matt “mdhaber” Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, CJ Carey, İlhan “ilayn” Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregos, Paul van Mulbregt, and SciPy 1.0 Contributors. “SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python”. *Nature Methods* 17:261–272, Mar. 2, 2020. London, England, UK: Springer Nature Limited. ISSN: 1548-7091. doi:10.1038/s41592-019-0686-2. URL: <http://arxiv.org/abs/1907.10121> (visited on 2024-06-26). See also arXiv:1907.10121v1 [cs.MS] 23 Jul 2019. (Cit. on pp. 139, 374).
- [478] “Virtual Environments and Packages”. In: *Python 3 Documentation. The Python Tutorial*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. Chap. 12. URL: <https://docs.python.org/3/tutorial/venv.html> (visited on 2024-12-24) (cit. on p. 376).
- [479] Abdullah Wahbeh, Surendra Sarnikar, and Omar F. El-Gayar. “Exploring the Impact of Analyst Knowledge of Socio-Technical Concepts on Requirements Questionnaires Quality”. In: *22nd Americas Conference on Information Systems (AMCIS’2016)*. Aug. 11–14, 2016, San Diego, CA, USA. Atlanta, GA, USA: Association for Information Systems (AIS), 2016. URL: <https://scholar.dsu.edu/cgi/viewcontent.cgi?article=1016&context=bispapers> (visited on 2025-03-27) (cit. on p. 188).
- [480] Gerhard Weikum and Gottfried Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. The Morgan Kaufmann Series in Data Management Systems. Burlington, MA, USA/San Mateo, CA, USA: Morgan Kaufmann Publishers, June 2001. ISBN: 978-1-55860-508-4 (cit. on p. 7).
- [481] Thomas Weise (汤卫思). *Databases*. Hefei, Anhui, China (中国安徽省合肥市): Hefei University (合肥大学), School of Artificial Intelligence and Big Data (人工智能与大数据学院), 2025. URL: <https://thomasweise.github.io/databases> (visited on 2025-01-05) (cit. on pp. iv, 2, 56, 143, 194, 206, 207, 369, 374).
- [482] Thomas Weise (汤卫思). *Programming with Python*. Hefei, Anhui, China (中国安徽省合肥市): Hefei University (合肥大学), School of Artificial Intelligence and Big Data (人工智能与大数据学院), 2024–2025. URL: <https://thomasweise.github.io/programmingWithPython> (visited on 2025-01-05) (cit. on pp. iv, 1, 4, 20, 24, 56, 57, 88, 104, 105, 115, 139, 140, 193–195, 206, 240, 365, 369, 372, 373).
- [483] John R. Weitzel and Larry Kerschberg. “Developing Knowledge-Based Systems: Reorganizing the System Development Life Cycle”. *Communications of the ACM (CACM)* 32(4):482–488, Apr. 1989. New York, NY, USA: Association for Computing Machinery (ACM). ISSN: 0001-0782. doi:10.1145/63334.63340 (cit. on p. 180).
- [484] Matthew West. *Developing High Quality Data Models*. Version: 2.0, Issue: 2.1. London, England, UK: Shell International Limited and European Process Industries STEP Technical Liaison Executive (EPISTLE); Burlington, MA, USA/San Mateo, CA, USA: Morgan Kaufmann Publishers, Dec. 8, 1995–Dec. 2010. ISBN: 978-0-12-375107-2. URL: <https://www.researchgate.net/publication/286610894> (visited on 2025-03-24). Edited by Julian Fowler (cit. on pp. 184, 195, 369).

- [485] Randolph West. *How should I store currency values in SQL Server?* Calgary, AB, Canada: Born SQL, June 3, 2020. URL: <https://bornsql.ca/blog/how-should-i-store-currency-values-in-sql-server> (visited on 2025-02-27) (cit. on pp. 105, 365).
- [486] *What does PDF mean?* San Jose, CA, USA: Adobe Systems Incorporated, 2024. URL: <https://www.adobe.com/acrobat/about-adobe-pdf.html> (visited on 2024-12-12) (cit. on p. 373).
- [487] *What is a Relational Database?* Armonk, NY, USA: International Business Machines Corporation (IBM), Oct. 20, 2021–Dec. 12, 2024. URL: <https://www.ibm.com/think/topics/relational-databases> (visited on 2025-01-05) (cit. on pp. 3, 374).
- [488] Peter Whyte. *Microsoft SQL Server DBA Blog*. Edinburgh, Scotland, UK, 2018–2025. URL: <https://peter-whyte.com/sql-dba-blog> (visited on 2025-06-03) (cit. on pp. 15, 18, 101, 372).
- [489] Ulf Michael “Monty” Widenius, David Axmark, and Uppsala, Sweden: MySQL AB. *MySQL Reference Manual – Documentation from the Source*. Sebastopol, CA, USA: O’Reilly Media, Inc., July 9, 2002. ISBN: 978-0-596-00265-7 (cit. on pp. 14, 372).
- [490] Kevin Wilson. *Python Made Easy*. Birmingham, England, UK: Packt Publishing Ltd, Aug. 2024. ISBN: 978-1-83664-615-0 (cit. on p. 373).
- [491] Marianne Winslett and Vanessa Braganholo. “Richard Hipp Speaks Out on SQLite”. *ACM SIGMOD Record* 48(2):39–46, June 2019. New York, NY, USA: Association for Computing Machinery (ACM). ISSN: 0163-5808. doi:10.1145/3377330.3377338 (cit. on pp. 15, 374).
- [492] “[WITH Queries \(Common Table Expressions\)](#)”. In: *PostgreSQL Documentation*. 17.4. The PostgreSQL Global Development Group (PGDG), Feb. 20, 2025. Chap. 7.8. URL: <https://www.postgresql.org/docs/17/queries-with.html> (visited on 2025-04-21) (cit. on p. 369).
- [493] “With Statement Context Managers”. In: *Python 3 Documentation. The Python Language Reference*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. Chap. 3.3.9. URL: <https://docs.python.org/3/reference/datamodel.html#with-statement-context-managers> (visited on 2024-12-15) (cit. on p. 140).
- [494] Bernard Wong. “A Study of the Metrics for Measuring the Quality of the Requirements Specification Document”. In: *International Conference on Software Engineering Research and Practice (SERP’2004)*. Vol. 2. June 21–24, 2004, Las Vegas, NV, USA. Ed. by Hamid R. Arabnia and Hassan Reza. USA: Computer Science Research, Education, and Applications (CSREA) Press, 2004, pp. 549–553. ISBN: 978-1-932415-29-2. URL: <https://opus.lib.uts.edu.au/bitstream/10453/6956/1/2004000890.pdf> (visited on 2025-03-27) (cit. on pp. 188, 374).
- [495] Martin Yanev. *PyCharm Productivity and Debugging Techniques*. Birmingham, England, UK: Packt Publishing Ltd, Oct. 2022. ISBN: 978-1-83763-244-2 (cit. on p. 373).
- [496] Kinza Yasar and Craig S. Mullins. *Definition: Database Management System (DBMS)*. Newton, MA, USA: TechTarget, Inc., June 2024. URL: <https://www.techtarget.com/searchdata-management/definition/database-management-system> (visited on 2025-01-11) (cit. on pp. 8, 369).
- [497] *yEd Graph Editor Manual*. Tübingen, Baden-Württemberg, Germany: yWorks GmbH, 2011–2025. URL: <https://yed.yworks.com/support/manual/index.html> (visited on 2025-03-31) (cit. on pp. iv, 20, 64, 195, 200, 367, 376).
- [498] Ka-Ping Yee and Guido van Rossum. [Iterators](#). Python Enhancement Proposal (PEP) 234. Beaverton, OR, USA: Python Software Foundation (PSF), Jan. 30–Apr. 30, 2001. URL: <https://peps.python.org/pep-0234> (visited on 2025-02-02) (cit. on p. 141).
- [499] François Yergeau. *UTF-8, A Transformation Format of ISO 10646*. Request for Comments (RFC) 3629. Wilmington, DE, USA: Internet Engineering Task Force (IETF), Nov. 2003. URL: <https://www.ietf.org/rfc/rfc3629.txt> (visited on 2025-02-05). See [Unicode](#) and [224] (cit. on p. 376).

- [500] Wenqi Ying (应雯棋), ed. *Commemoration of Ancient Chinese Mathematical Master Liu Hui for his Timeless Influence on Mathematics and Civilizational Exchange*. Vol. 48 (Special Issue) of CAST Newsletter. China, Beijing (中国北京市): 中国科学技术协会 (China Association for Science and Technology, CAST), Nov. 2024. URL: https://english.cast.org/cms_files/filemanager/1941250207/attach/202412/8f23655a82364d19ad7874eb37b23035.pdf (visited on 2025-08-24). Proofreader: Yumeng Wei (魏雨萌), Designer: Shan Zhang (张珊) (cit. on p. 201).
- [501] Pavlo V. Zahorodko and Pavlo V. Merzlykin. "An Approach for Processing and Document Flow Automation for Microsoft Word and LibreOffice Writer File Formats". In: *4th Workshop for Young Scientists in Computer Science & Software Engineering (CS&SE@SW'2021)*. Dec. 18, 2021, Virtual Event and Kryvyi Rih, Ukraine. Ed. by Arnold E. Kiv, Serhiy O. Semerikov, Vladimir N. Soloviev, and Andrii M. Striuk. Vol. 3077 of CEUR Workshop Proceedings (CEUR-WS.org). Aachen, Nordrhein-Westfalen, Germany: CEUR-WS Team, Rheinisch-Westfälische Technische Hochschule (RWTH) Aachen, 2022, pp. 66–82. ISSN: 1613-0073. URL: <https://ceur-ws.org/Vol-3077/paper12.pdf> (visited on 2025-10-04) (cit. on pp. 371, 372).
- [502] Giorgio Zarrelli. *Mastering Bash*. Birmingham, England, UK: Packt Publishing Ltd, June 2017. ISBN: 978-1-78439-687-9 (cit. on p. 368).
- [503] Zhenyu Zhu. "Requirements Determination and Requirements Structuring". In: *Information Systems Analysis: 6840 Papers*. Ed. by Vicki L. Sauter. St. Louis, MO, USA: University of Missouri-St. Louis, Aut. 2003. URL: https://www.umsl.edu/~sauterv/analysis/6840_f03_papers/zhu (visited on 2025-03-26) (cit. on pp. 187, 188).
- [504] Nicola Abdo Ziadeh, Michael B. Rowton, A. Geoffrey Woodhead, Wolfgang Helck, Jean L.A. Filliozat, Hiroyuki Momo, Eric Thompson, E.J. Wiesenber, and Shih-ch'ang Wu. "Chronology – Christian History, Dates, Events". In: *Encyclopaedia Britannica*. Ed. by The Editors of Encyclopaedia Britannica. Chicago, IL, USA: Encyclopædia Britannica, Inc., July 26, 1999–Mar. 20, 2024. URL: <https://www.britannica.com/topic/chronology/Christian> (visited on 2025-08-27) (cit. on p. 368).
- [505] Jelle "JelleZijlstra" Zijlstra, Mehdi "hmc-cs-mdrissi" Drissi, Alex "AlexWaygood" Waygood, Daniele "dvarrazzo" Varrazzo, Shantanu "hauntsaninja", François-Michel "FinchPowers" L'Heureux, and Rupesh "rupeshs" Sreeraman. *Issue #12554: Support PEP 675 (LiteralString)*. San Francisco, CA, USA: GitHub Inc, Apr. 9, 2022–Nov. 29, 2024. URL: <https://github.com/python/mypy/issues/12554> (visited on 2025-03-05) (cit. on p. 374).
- [506] "中华人民共和国外国人工作许可证 (Work Permit for Foreigners of the People's Republic of China)". In: 百度百科 (Baidu Baike). China, Beijing (中国北京市): Baidu (百度公司), Jan. 19–22, 2025. URL: <https://baike.baidu.com/item/%E4%B8%AD%E5%8D%8E%E4%BA%BA%E6%B0%91%E5%85%B1%E5%92%8C%E5%9B%BD%E5%A4%96%E5%9B%BD%E4%BA%BA%E5%B7%A5%E4%BD%9C%E8%AE%B8%E5%8F%AF%E8%AF%81> (visited on 2025-04-03) (cit. on p. 210).
- [507] 公民身份号码 (*Citizen Identification Number*). 中华人民共和国国家标准 (National Standard of the People's Republic of China, GB) GB11643-1999. China, Beijing (中国北京市): 中华人民共和国国家质量监督检验检疫总局 (General Administration of Quality Supervision, Inspection and Quarantine of the People's Republic of China), 中国国家标准化管理委员会 (Standardization Administration of the People's Republic of China, SAC), and 中国标准出版社 (Standards Press of China), Jan. 19–Nov. 3, 1999. URL: <https://openstd.samr.gov.cn/bzgk/gb/newGbInfo?hcno=080D6FBF2BB468F9007657F26D60013E> (visited on 2024-07-26) (cit. on pp. 5, 195, 201, 204, 209, 233, 238).
- [508] "手机号码 : 电话管理部门为手机设定的号码 (Mobile Phone Number: A Number Set by the Telephone Management Department for Mobile Phones)". In: 百度百科 (Baidu Baike). China, Beijing (中国北京市): Baidu (百度公司), Jan. 6, 2025. URL: <https://baike.baidu.com/item/%E6%89%8B%E6%9C%BA%E5%8F%B7%E7%A0%81> (visited on 2025-04-17) (cit. on pp. 115, 235).
- [509] 来华签证简介 (*Introduction to Chinese Visa*). China, Beijing (中国北京市): 中华人民共和国外交部 (Ministry of Foreign Affairs of the People's Republic of China), Nov. 20, 2019. URL: http://cs.mfa.gov.cn/wgrlh/lhqzjj_660596 (visited on 2025-04-03) (cit. on p. 210).
- [510] 百度百科 (Baidu Baike). China, Beijing (中国北京市): Baidu (百度公司), Apr. 20, 2006. URL: <https://baike.baidu.com> (visited on 2025-04-03).