POLISH NATIONAL AGENCY FOR ACADEMIC EXCHANGE

Publication financed by NAWA
as part of activities complementary to those undertaken by UJD within the COLOURS alliance

Fundamentals of databases

PostgreSQL: examples and practice exercises

Lidia Stępień

Marcin R. Stępień

Artur Gola

Jan Dlugosz University in Czestochowa

Fundamentals of databases
PostgreSQL: examples and practice exercises

Lidia Stępień, Marcin R. Stępień, Artur Gola

Częstochowa 2024

Reviewer
dr Paweł Róg

Translator
Katarzyna Stępień

Editor-in-chief
Paulina Piasecka-Florczyk

Proofreading
Bożena Woźna-Szcześniak

Technical editing and cover design
Bożena Woźna-Szcześniak

© Copyright by
Jan Dlugosz University in Częstochowa
Czestochowa 2024

ISBN 978-83-67984-25-6

The Publishing House of Jan Dlugosz University in Czestochowa
42-200 Częstochowa, al. Armii Krajowej 36A
www.ujd.edu.pl
e-mail: wydawnictwo@ujd.edu.pl

Contents

Preface **6**

1 Introduction to PostgreSQL System **7**

1.1 System Architecture . . . 7
1.2 Environment Preparation . . . 8
1.3 `psql` Client Commands . . . 9
1.4 Session Information Functions . . . 11
1.5 Comments . . . 12
1.6 SQL Data Types and Operators . . . 12
1.6.1 Data Types . . . 12
1.6.2 Mathematical Operators . . . 14
1.6.3 Logic and Comparison Operators . . . 15
1.6.4 Symbols and Operators for Building Regular Expressions . . . 16

2 Basic Table Operations **18**

2.1 Table Creation . . . 20
2.2 Modifying the Table Structure . . . 23
2.3 Adding a New Data to the Table . . . 25
2.4 Modifying and Deleting Data . . . 27
2.5 Dropping a Table . . . 29
2.6 Practice Exercises . . . 30

3 Simple Database Queries **33**

3.1 SELECT Query Syntax . . . 33
3.2 Fetching Data . . . 35
3.2.1 SELECT Clause . . . 35
3.2.2 WHERE Clause . . . 41
3.2.3 ORDER BY Clause . . . 44
3.3 Copying Data . . . 45

3.4 Practice Exercises 47

4 Data Joining 50

4.1 Joining Tables Horizontally 50

4.1.1 Inner Join 51

4.1.2 Outer Join 55

4.1.3 Cartesian Product - CROSS JOIN 57

4.2 Connecting Vertical Connection 58

4.3 Practice Exercises 59

5 Aggregate Functions 61

5.1 Functions Operating on Groups of Rows 61

5.1.1 `GROUP BY` Clause 63

5.1.2 HAVING Clause 66

5.2 Data Cleansing and Quality Control 67

5.3 Window Function 68

5.4 Practice Exercises 73

6 Query Nesting 75

6.1 Subquery Categories 75

6.1.1 Independent Subqueries 76

6.1.2 Correlated Subqueries 79

6.2 Recursive nesting 82

6.3 `WITH` Expression 83

6.4 Subqueries and Aggregate Functions in Practice 85

6.5 Practice Exercises 87

7 Creating and Using Views 89

7.1 Basic Operations Related to Views 89

7.2 Views in Action 92

7.3 Practice Exercises 95

8 Creating and Using Indexes 96

8.1 B-tree Index 97

8.2 Stored Functions 98

8.3 Sample Session 100

8.4 Effective Use of Indexes 107

8.5 Practice Exercises 108

9 Transactions **111**
9.1 Concurrency Control . . . 111
9.1.1 Read/Write Locks . . . 111
9.1.2 The Range of Locks . . . 112
9.2 Transactions . . . 113
9.2.1 ACID . . . 115
9.2.2 Isolation Levels . . . 115
9.2.3 Deadlocks . . . 117
9.2.4 Transactions in PostgreSQL - `AUTOCOMMIT` . . . 119
9.3 Practice Exercises . . . 120

10 Creating Database Applications in Java **121**
10.1 Setting Up the Work Environment . . . 121
10.2 Java JDBC . . . 123
10.3 Types of JDBC Drivers . . . 124
10.4 Structure of a Database Application in Java . . . 125
10.4.1 Example – Establishing a Connection to the Database . . . 125
10.5 Working with SQL Commands . . . 131
10.6 SQLException Exceptions . . . 134
10.7 Example – creating a table in the database . . . 136
10.8 Example – inserting data into the table . . . 136
10.9 Example – executing queries . . . 138
10.10 Example – creating an application with a GUI (Graphical User Interface) 139
10.11 Practice exercises . . . 146

Bibliografia **148**

Appendix A **149**

Appendix B **156**

Preface

Database is an entity that always applies to a certain fragment of reality (the area of analysis) and constitutes a collection of data, with specified internal structure, which represents said fragment of reality and makes data retaining easier. The description the database's inner structure is handled by data models, which allow for precise (usually formalised) depiction of the data's properties, with the use of mathematical language, eg. relational algebra in the relational database model. Aside from describing data structures, models are also used for determining permissible data operations and as a way of imposing constraints ensuring the correctness of the database.

To manage a database (understood as a collection of data), a Database Management System (DBMS) is used. It is an organised set of tools that enables the execution of operations on data that are essential for the user. So far, many database management systems (DBMS) have been developed based on various models, but those based on the relational model remain the most popular. Regardless of the choice, such software requires an understanding of the theoretical foundations of the model and the high-level language SQL (Structured Query Language). The difficulty in learning lies in the diversity and complexity of DBMS software offered by different vendors. The authors of this textbook propose learning based on PostgreSQL, one of the most popular open relational database management systems (as of 2023, according to `https://db-engines.com/en/blog_post/106`). For many years, this software has not fallen from the global popularity ranking of DBMS, consistently remaining in the top ten among nearly 400 DBMS of all types (the DB-Engines website: `https://db-engines.com/en/`).

The proposed textbook includes an introduction to basic SQL statements performed in relational databases, richly illustrated with examples, and tasks to be solved independently to consolidate the acquired skills. The reader will be introduced to how to create tables (basic data structures in the relational model), modify, delete and manipulate data. We will show how to improve query execution time by using indexes, while discussing selected basic issues related to the functions of the `plpgSQL` procedural language. In the last chapter we will present how to create a database application in Java we assume that the reader has already mastered programming skills in this language.

Chapter 1

Introduction to PostgreSQL System

The following chapter describes the architecture of the PostgreSQL database management systems, its installation on Linux and Windows operating systems and `psql` client's commands. Furthermore, we will also introduce basic data types and operators used in Structured Query Language (SQL).

1.1. System Architecture

PostgreSQL is one of the most popular database management system and one of the few offering object-oriented and relational approach to data.

The system architecture consists of `client-server` structure. In this model one client can simultaneously use services of multiple servers, and one server - be accessed by many users at the same time. That means splitting the roles between client and server as follows:

- The client is the one requesting database access - the client sent the request to the server and awaits the response, using dedicated interface for the operation. The client does not have any influence over server connection, protection of the data stored on it, nor their processing.
- The server is responsible for establishing the connection, processing the data and managing security. Its role is passive - it offers request-based services (providing adequately processed data)

For PostgreSQL there exist two kinds of client type tools - graphical user interface `pgAdmin` and `commandline` tool `psql`.

In this textbook, we will use `psql` – a terminal client that allows connecting to a PostgreSQL database and executing SQL queries interactively or from a previously prepared file. It can also be used to save query results to files.

1.2. Environment Preparation

Before starting a thorough reading of this textbook, specific software and tools need to be prepared. In the following subsections, we will explain how to do this on the most popular operating systems, Windows and Linux.

Download and installation of PostgreSQL system for Windows

To download PostgreSQL for Windows, follow these steps:

1. Go to `https://www.postgresql.org/download/` and select the *Windows* option from the *Packages and Installers* list.
2. After selecting *Download the Installer*, download the appropriate version of *PostgreSQL* (version 15.8 will be used in this textbook).
3. Run the interactive installer and click the *Next* button in most steps. When prompted to specify a data directory, provide a path that is easy to remember.
4. Enter the password for the *postgres* administrator.
5. Do not change the default port unless it conflicts with an application already installed on the system.
6. In the remaining steps, click the *Next* button and wait for the installation to complete.

To check if the *Path* variable is set correctly, open the command prompt, type the command `psql -U postgres` and press *Enter*.

If you receive the error `'psql' is not recognized as an internal or external command, operable program or batch file`, you need to add the PostgreSQL binaries directory to the *Path* variable.

To do this, follow these steps:

1. In the Windows search bar, type *environment variables*.
2. Select the *Environment Variables* option, highlight the *Path* variable, and click the *Edit* button.
3. Click the *New* button.
4. In File Explorer, locate the directory where PostgreSQL is installed and add the path to the *bin* directory of the PostgreSQL installation.
5. Confirm by clicking *OK* and restart your system.

After restarting your computer, open the command prompt and type the command `psql -U postgres` and press *Enter*. Launching the PostgreSQL shell requires entering

the password set during step 4 of the installation process and pressing *Enter*. Successful entry into the PostgreSQL shell is indicated by a change in the prompt. To exit the shell, type the command: `\q` and press *Enter*.

Download and installation of PostgreSQL system for Linux

Installing PostgreSQL on Ubuntu or Debian distribution-based systems:

1. Follow the installation instructions available at https://www.postgresql.org/download/linux/ubuntu/.
2. During the installation of PostgreSQL, a `postgres` user will be created, allowing you to launch the shell for this user by executing:

   ```
   sudo su postgres
   ```

   The change in the prompt will indicate access to the shell, where you can start the client by entering the command `psql`.

Instructions for installing PostgreSQL on other distributions can be found in the documentation. The download page for PostgreSQL on Linux systems is available at https://www.postgresql.org/download/.

1.3. psql Client Commands

Table 1.1: Selected client command - psql

Command	Description
\c[onnect] [DBNAME]	connect to database
\h [NAME]	help on syntax of SQL commands, * for all commands
\q	quit psql
\e [FILE]	edit the query buffer (or file) with external editor
\ef [FUNCNAME]	edit function definition with external editor
\p	show the contents of the query buffer
\r	reset (clear) the query buffer
\s [FILE]	display history or save it to file
\w FILE	write query buffer to file
\echo [STRING]	write string to standard output
\i FILE	execute commands from file

Continued on next page

Continued from previous page	
Command	**Description**
(options: S – show system objects, + – additional detail)	
\d[S+]	list tables, views, and sequences
\d[S+] NAME	describe table, view, sequence, or index
\da[+] [PATTERN]	list aggregates
\db[+] [PATTERN]	list tablespaces
\dc[S] [PATTERN]	list conversions
\dC [PATTERN]	list casts
\dd[S] [PATTERN]	show comments on objects
\dD[S] [PATTERN]	list domains
\des[+] [PATTERN]	list foreign servers
\deu[+] [PATTERN]	list user mappings
\dew[+] [PATTERN]	list foreign-data wrappers
\df[antw][S+] [PATRN]	list [only agg/normal/trigger/window] functions
\dF[+] [PATTERN]	list text search configurations
\dFd[+] [PATTERN]	list text search dictionaries
\dFp[+] [PATTERN]	list text search parsers
\dFt[+] [PATTERN]	list text search templates
\dg[+] [PATTERN]	list roles (groups)
\di[S+] [PATTERN]	list indexes
\dl	list large objects, same as \lo_list
\dn[+] [PATTERN]	list schemas
\do[S] [PATTERN]	list operators
\dp [PATTERN]	list table, view, and sequence access privileges
\ds[S+] [PATTERN]	list sequences
\dt[S+] [PATTERN]	list tables
\dT[S+] [PATTERN]	list data types
\du[+] [PATTERN]	list roles (users)
\dv[S+] [PATTERN]	list views
\l[+]	list all databases
\z [PATTERN]	same as \dp
\encoding [ENCODING]	show or set client encoding
\password [USERNAME]	securely change the password for a user
\prompt [TEXT] NAME	prompt user to set internal variable
\set [NAME [VALUE]]	set internal variable, or list all if no parameters
\unset NAME	unset (delete) internal variable

1.4. Session Information Functions

Table 1.2: Selected functions of the Session Information Functions

Name	Return Type	Description
current_catalog	name	name of current database (called ”catalog” in the SQL standard)
current_database()	name	name of current database
current_schema[()]	name	name of current schema
current_schemas(boolean)	name[]	names of schemas in search path optionally including implicit schemas
current_user	name	user name of current execution context
current_query	text	text of the currently executing query, as submitted by the client (might contain more than one statement)
pg_backend_pid()	int	Process ID of the server process attached to the current session
inet_client_addr()	inet	address of the remote connection
inet_client_port()	int	port of the remote connection
inet_server_addr()	inet	address of the local connection
inet_server_port()	int	port of the local connection
pg_my_temp_schema()	oid	OID of session’s temporary schema, or 0 if none
pg_is_other_temp_schema(oid)	boolean	is schema another session’s temporary schema?
pg_postmaster_start_time()	timestamp with time zone	server start time
pg_conf_load_time()	timestamp with time zone	configuration load time

Continued on next page

Continued from previous page

Name	Return Type	Description
session_user	name	session user name user name equivalent to current_user
version()	text	PostgreSQL version information

1.5. Comments

```
--        single line comment
/*  */    multiline comment
```

1.6. SQL Data Types and Operators

Relational databases and their standard language, SQL, are the two main technologies used for storing and processing large data sets. Despite the ANSI standard, vendors often introduce their own data types, frequently incorporating custom synonyms. Therefore, in this textbook, we will focus on the SQL data types and operators specific to the PostgreSQL system.

1.6.1. Data Types

Table 1.3: PostgreSQL - Data types

Name	Alias	Description
bigint	int8	8-byte integer of value from -9223372036854775808 to 9223372036854775807
bigserial	serial8	8-byte integer with automatic increment of value 1 to 9223372036854775807
bit [(n)]		bit string of constant length, ex. BIT(3) $\rightarrow$ B'101'
bit varying [(n)]	varbit	bit string of varying length
boolean	bool	logical value (true/false)
box		32-bit 2D rectangle ((x1,y1),(x2,y2))
byte		byte array

Continued on next page

Continued from previous page		
Name	**Alias**	**Description**
character varying [(n)]	varchar [(n)]	variable length character string
character [(n)]	char [(n)]	constant length character string
cidr		IPv4 or IPv6 address, 7 lub 19B
circle		2D circle, 24 bytes `<(x,y),r>` the centroid and radius
date		callendar date (year, month, day), 4 bytes, 4713BC to 5874897AD, one day precision
double precision	float8	floating-point number (8 bytes) of precision 15 decimal digits
inet		IPv4 or IPv6 host address, 7 or 19B
integer	int , int4	4-byte integer
interval [fields] [(p)]		time interval, 12 byte, precision 1μs, from -178000000 years to 178000000 years
line		infinite 2D line, 32 bytes, ((x1,y1),(x2,y2))
lseg		2D line segment, 32 bytes, ((x1,y1),(x2,y2))
macaddr		MAC Media Access Control address, 6B
money		currency value
numeric [(p,s)]	decimal[(p,s)]	floating-point number of chosen precision
path		2D geometrical path ((x1,y1),...)
point		2D geometrical point 16 bytes, (x,y)
polygon		2D polygon ((x1,y1),...)
real	float4	a single-precision floating-point number (4 bytes) with a precision of 6 decimal digits
smallint	int2	2-byte integer, value from -32768 to $+32767$
serial	serial4	4-byte integer with automatic increment, value from 1 to 2147483647
text		variable length string
time [(p)] [without time zone]		time of day (without timezone), 8 bytes, from $00:00:00$ to $24:00:00$, precision 1μs, 14 digits
		Continued on next page

Continued from previous page

Name	Alias	Description
time [(p)] with time zone	timetz	time with timezone, 12 bytes, from $00:00:00+1459$ to $24:00:00-1459$ precision 1μs, 14 digits
timestamp [(p)] [without time zone]		date and time without timezone, 8 bytes, from $4713BC$ to $294276AD$, precision 1μs, 14 digits
timestamp [(p)] with time zone	timestamptz	date and time with timezone, 8 bytes, from 4713BC to 294276AD, precision 1μs, 14 digits
tsquery		search query text
tsvector		search document text
txid_snapshot		transaction on the level of the user of the ID
uuid		universally unique identifier
xml		XML data

1.6.2. Mathematical Operators

Table 1.4: PostgreSQL mathematical operators

<table>
<tr><th>Operator</th><th>Description</th><th>Example</th><th>Return</th></tr>
<tr><td>+</td><td>addition</td><td>2 + 2</td><td>4</td></tr>
<tr><td>-</td><td>subtraction</td><td>2 - 3</td><td>-1</td></tr>
<tr><td>*</td><td>multiplication</td><td>2 * 3</td><td>6</td></tr>
<tr><td>/</td><td>integer division (truncates the result)</td><td>4 / 2</td><td>2</td></tr>
<tr><td>%</td><td>modulo</td><td>5%4</td><td>1</td></tr>
<tr><td>^</td><td>power</td><td>2.0^3.0</td><td>8</td></tr>
<tr><td>|/</td><td>square root</td><td>|/25.0</td><td>5</td></tr>
<tr><td>||/</td><td>cube root</td><td>||/27.0</td><td>3</td></tr>
<tr><td>!</td><td>factorial</td><td>5!</td><td>120</td></tr>
<tr><td>!!</td><td>factiorial (prefix operator)</td><td>!!5</td><td>120</td></tr>
<tr><td>@</td><td>absolute value</td><td>@ − 5.0</td><td>5</td></tr>
<tr><td>&</td><td>bitwise AND</td><td>91&15</td><td>11</td></tr>
<tr><td>|</td><td>bitwise OR</td><td>32|3</td><td>35</td></tr>
</table>

Continued on next page

Continued from previous page			
Operator	**Description**	**Example**	**Return**
#	bitwise XOR	17#5	20
~	bitwise NOT	~ 1	-2
<<	bitwise shift left	1 << 4	16
>>	bitwise shift right	8 >> 2	2

1.6.3. Logic and Comparison Operators

Comparison operators used in PostgreSQL: < (smaller), > (greater), <= (less than or equal to), >= (greater than or equal to), = (equal), <> or != (different).

Table 1.5: Truth tables for conjunction (**AND**) and disjunction (**OR**)

a	**b**	**a AND b**	**a OR b**
TRUE	TRUE	TRUE	TRUE
TRUE	FALSE	FALSE	TRUE
TRUE	NULL	NULL	TRUE
FALSE	FALSE	FALSE	FALSE
FALSE	NULL	FALSE	NULL
NULL	NULL	NULL	NULL

Table 1.6: Truth table for negation (**NOT**)

a	**NOT a**
TRUE	FALSE
FALSE	TRUE
NULL	NULL

Special operators to check whether an expression is one of the highlighted values:

- whether the expression is/is not an undefined value `NULL`:

```
<expression> IS NULL
<expression> IS NOT NULL
```

- whether the expression `expression_1` is different/or not from the expression indicated in the `FROM` phrase:

```
<expression_1> IS DISTINCT FROM <expression_2>
<expression_1> IS NOT DISTINCT FROM <expression_2>
```

- whether the expression evaluates to `TRUE` or `FALSE`:

```
<expression> IS TRUE
<expression> IS NOT TRUE
<expression> IS FALSE
<expression> IS NOT FALSE
```

- whether the expression is unknown or not:

```
<expression> IS UNKNOWN
<expression> IS NOT UNKNOWN
```

1.6.4. Symbols and Operators for Building Regular Expressions

`Regular expressions` (`regex`) are used to search for specific strings of characters in text that satisfy the rules defined in a regular expression. They create patterns that are constructed using the operators and symbols presented in the 1.7 and 1.8 tables, respectively.

Table 1.7: Regular expression (regex) operators

Operator	Usage	Description
~	`'string' ~ 'regex'`	Regular expression match, returns *true* if the pattern **matches** the string
!~	`'string' !~ 'regex'`	Regular expression match, returns *true* if the pattern **does not match** the string
~*	`'string' ~* 'regex'`	Case-insensitive regular expression match, returns *true* if the pattern **matches** the string
!~*	`'string' !~* 'regex'`	Case-insensitive regular expression match, returns *true* if the pattern **does not match** the string

Table 1.8: Regular expressions (regex) symbols

Symbol	Usage	Description
^	`^expression`	Matches the beginning of a string
$	`expression$`	Matches the end of a string
.	.	Matches any single character

Continued on next page

Continued from previous page		
Symbol	**Usage**	**Description**
[ ]	$[abc]$	Matches any single character listed within the brackets (e.g., `a`, `b`, or `c`)
[^]	$[\ {}^\wedge abc]$	Matches any single character not listed within the brackets (e.g., not `a`, `b`, or `c`)
[−]	$[a - z]$	Matches any single character not in the range specified within the brackets and separated by a hyphen (e.g., outside the range from `a` to `z`)
[^−]	$[\ {}^\wedge a - z]$	Matches any single character in the range specified within the brackets and separated by a hyphen (e.g., from `a` to `z`)
?	$a?$	Denotes zero or one occurrence of the preceding character or preceding question mark sequence of a regular expression
*	$a*$	Denotes zero or more occurrences of the preceding character or preceding asterisk sequence of a regular expression
+	$a+$	Denotes one or more occurrences of the preceding character or preceding plus sequence of a regular expression
$\vert$	$expr1 \vert expr2$	Denotes expression on the left or right side of the $\vert$ character (e.g., `expr1` or `expr2`)
()	$(expr1)expr2$	Explicitly groups expressions to determine the precedence of special character symbols
{}	$\{m\}$ $\{m,\}$ $\{m,n\}$	Exactly `m` times sequence At least `m` times sequence Range of occurrences, where $m < n$
$\backslash d$	$\backslash d\{2\}$	Any digit (e.g., exactly two digits)

Chapter 2

Basic Table Operations

Relational data model has been created in 1970 by Edgar F. Codd, based on relational algebra. In this model the data is packed into organised by relations collections of tuples. A tuple is a specifically ordered set of attributes. Tuples are usually called ”records” in database system context.

In this textbook, we will consider an example relational database representing book sales in an online ”bookstore”. This database includes information about customers (relation `t_client`). The attributes describing each record include details about individual customers such as the customer’s name and surname, delivery address, and contact phone number. Sample data contained in the `t_client` relation is presented in the table 2.1.

Table 2.1: Example data in relation `t_client`

client_id	name	surname	...	telephone
1	Jan	Kowalski	...	624-455-566
2	Tadeusz	Malinowski	...	624-424-332
3	Krystyna	Torbicka	...	624-212-111

Every relation is a 2D table, in which all rows are records, and the columns are the attributes describing them. Within a table, records must be unique. Therefore, although not technically required, most tables in the relational data model have a column (or group of columns) called the primary key (often also referred to as the base key), which uniquely identifies a single row. In table 2.1, `client_id` is the distinguished primary key. Data within columns belong to the same domain and have a data type describing the type of data stored in them. Figure 2.1 presents graphically in the form of an Entity-Relationship Diagram (ERD) the tables of the considered bookstore database along with their connections.

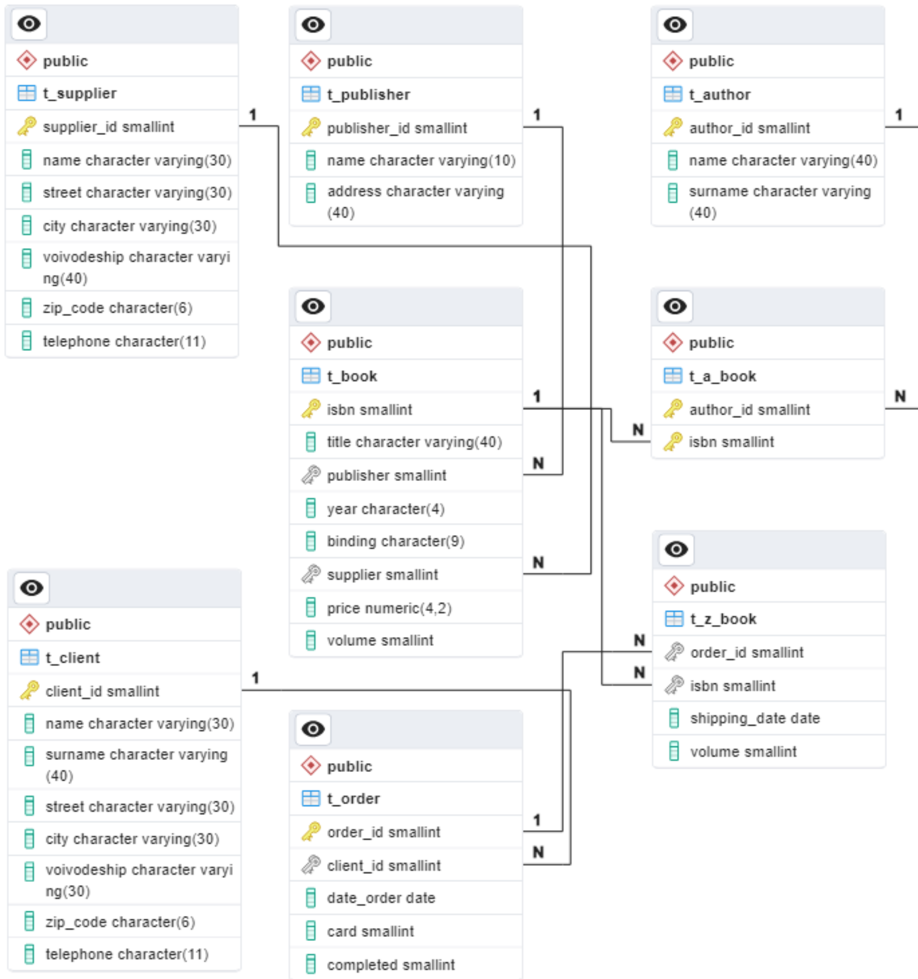

Figure 2.1: Database EDR diagram `bookstore`

Most operations in relational databases, and all data management systems in general, are related to tables and the data stored in them. In general, these operations can be divided into four groups: create, read, update, and delete. In order to operate on data, you must first prepare a definition of a data set, then supplement the set with data records to be able to modify, read, update, and when the data is no longer needed, also delete them. These operations are commonly known as **CRUD** (*Create, Read, Update, Delete*), the order of the letters indicating the order in which each operation is performed in the life cycle of the dataset.

In relational databases, the above-mentioned operations are performed using SQL under the control of a database management system. Individual CRUDs differ slightly in the way they handle data and even SQL syntax. There is a standard called ANSI

(*American National Standards Institute*) that describes SQL, which is largely followed by the CRUDs. However, each of the available operations has some specific interpretations and extensions of this standard that distinguish them from each other.

In this and later chapters, the reader will be introduced to all SQL instructions, often in an abbreviated form, corresponding to the operations mentioned above. For a full description of the SQL language instructions, the interested reader is referred to the PostgreSQL documentation at the link: `https://www.postgresql.org/docs/15/index.html`.

Writing SQL instructions, while it is not compulsory, one can adapt to the generally accepted rules given below, which facilitate both writing and compiling the code, and improve its readability.

- SQL commands can span multiple lines.
- It is a good practice to place clauses from new line.
- the end of the instruction is marked by semicolon.
- Words cannot be split between lines.
- Both lower- and uppercase letters can be used in the command, unless we are checking the value of a field.

The SQL voivodeshipment syntax presented in this manual will use square brackets ([]) to indicate optional elements, and curly brackets ({ }) to indicate a set of selectable options separated by a pipe (|).

```
COMMAND [OPTIONAL PART] {OPTION1 | OPTION2 | OPTION3}
```

2.1. Table Creation

To create a new, empty table in the database, use the `CREATE TABLE` command. Using the `IF NOT EXISTS` option in a command skips execution of the voivodeshipment when a table with the specified name already exists in the database. Failure to do so can result in an error: `ERROR: relation...already exists`.

Table creation consists of declaring names of the columns, their types, as well as the table's constraints.

```
CREATE TABLE [IF NOT EXISTS] table_name (
  column_name data_type [DEFAULT default_expr]
   [column_constraint[...]]
  [, ... ]
  | table_constraint
) [TABLESPACE tablespace];
```

where:

- `column_constraint`:

```
[CONSTRAINT constraint_name]
{NOT NULL | NULL | UNIQUE index_parameters |
PRIMARY KEY index_parameters |
CHECK (expression) | REFERENCES reftable [(refcolumn)]
[ON DELETE action][ON UPDATE action]}
```

- `table_constraint`:

```
[CONSTRAINT constraint_name]
{UNIQUE (column_name [,...]) index_parameters |
PRIMARY KEY (column_name [,...]) index_parameters |
CHECK (expression) |
FOREIGN KEY (column_name [,...] )
REFERENCES reftable [(refcolumn [,...])]
[ON DELETE action][ON UPDATE action]}
```

- `index_parameters` in constraints `UNIQUE` and `PRIMARY KEY`:

```
[WITH (storage_parameter [=value] [,...])]
[USING INDEX TABLESPACE tablespace]
```

It is necessary for the name of the table `table_name` to be unique in a single database. The uniqueness condition also applies to the column names `column_name` in the table. Each column created must be assigned one of the basic PostgreSQL data types (see the 1.3 table on the 12 page), such as:

- numeric types;
- character - note the difference between the fixed-length character type `char(n)`, padded with space characters up to a maximum length of n characters, and the character varying `varchar(n)`, matching the actual length of the string, but not exceeding the maximum declared length of n;
- logical;
- date and time - treated as strings;
- data structures (formatted as JSON and tables).

Columns defined in the newly created table can possess constrains `column_constraint`, that specify the special characteristics of the columns and guarantee that all rows in a given column will comply with the defined condition, e.g.

- `name varchar(20) NOT NULL` - this attribute must be provided (cannot be missing from the data);
- `price numeric(4,2) CHECK(price > 0)` - value constraints, in this case only positive numbers can be put into the price column;

- `st varchar(7) CHECK(st in ('mr', 'ms', 'mrs','dr'))` - only allows values from the given list to be put into column;
- `date_order date check(date_order <= current_date)` - order date cannot be later than the current date.

When building constraints, operators such as ($=, <, !=$, `is null`, `like` (% - string of length >= 0; _ - single character), `between...and`, `in (...)`, `and`, `or`, `not`) are used, among others. For a list of PostgreSQL mathematical operators, see 1.6.2 on 14 and logical operators and comparison 1.6.3 on 15. Additionally, constraints can be given a unique name `constraint_name`, which makes them easier to manipulate. PostgreSQL will give self-unnamed constraints their own names, which can be read, among other things, by issuing the command `\d table_name` in the `psql` client. A complete list of `psql` client commands can be seen in the 1.3 subsection on the 9 page.

To illustrate this, we shall show commands that create two new tables from the `bookstore` database shown in 2.1. Due to the foreign key defined within tables, the order in which tables are created is important. All tables that have only primary keys should be defined first. Then there are those in which, apart from the primary key, there is a foreign key, because the compiler checks whether there is a table and a column to which a reference is created. Finally, tables consisting only of foreign keys, so-called "associative (connective) tables".

For example, we create a table named `t_client`, which contains, among other things, the integer column `client_id`, which is its primary key, and the string columns `zip_code` and `telephone`, which are constrained on regular expression input. In the `zip_cod` column, only strings starting with two digits ($\hat{}\backslash d\{2\}$) followed by a minus sign ($-$) and ending with three digits ($\backslash d\{3\}\$$) are allowed. The condition uses the comparison operator for regular expressions, the tilde ($\sim$). In the case of a phone number, the format of the data entry requires grouping the digits of the number into 3 characters ($\hat{}\backslash d\{3\}$ at the beginning, $\backslash d\{3\}$ in the middle, $\backslash d\{3\}\$$ at the end) and separating them with a minus sign ($-$).

```
CREATE TABLE t_client(
  client_id int2 PRIMARY KEY,
  name varchar(30),
  surname varchar(40),
  street varchar(30),
  city varchar(30),
  voivodeship varchar(30),
  zip_code char(6) CHECK(zip_code ~ ('^\d{2}-\d{3}$')),
  telephone char(11) CHECK(telephone ~ ('^\d{3}-\d{3}-\d{3}$'))
);
```

The second example shows the command to create a table `t_order`, in which, in addition to defining the primary key, which is the column `order_id`, a table constraint is created named `key_from_client`, which defines a foreign key consisting of the `client_id` column of the `t_order` table as a reference to the `t_client` table and the primary key `client_id` defined in it with cascading deletion, i.e. when a record is deleted from a primary key table, records are automatically deleted from a relation in which the foreign key value is equal to the primary key value of the record being deleted. Therefore, you should use this option carefully so as not to lose the data associated with the master key being removed. If you do not use cascading table deletion, the table deletion operation must be performed in the reverse order in which the tables were created.

```
CREATE TABLE t_order (
  order_id int2 primary key,
  client_id int2,
  date_order DATE DEFAULT current_date,
  card int2,
  completed int2,
  CONSTRAINT key_from_client FOREIGN KEY(client_id)
  REFERENCES t_client(client_id) ON DELETE CASCADE
);
```

2.2. Modifying the Table Structure

After creating a table, you can freely modify its structure by using the `ALTER TABLE` command, among other things, allowing you to add new columns, new constraints, redefine existing columns, delete columns, remove constraints or temporarily activate/deactivate them.

```
ALTER TABLE [ONLY] name [*]
    action [,...];

ALTER TABLE [ONLY] name [*]
    RENAME [COLUMN] column TO new_column;

ALTER TABLE name
    RENAME TO new_name;

ALTER TABLE name
    SET SCHEMA new_schema;
```

where action:

```
ADD [COLUMN] column type [column_constraint [...]]
DROP [COLUMN] column [RESTRICT|CASCADE]
ALTER [COLUMN] column [SET DATA] TYPE type [USING expression]
ALTER [COLUMN] column SET DEFAULT expression
ALTER [COLUMN] column DROP DEFAULT
ALTER [COLUMN] column {SET|DROP} NOT NULL
ADD table_constraint
DROP CONSTRAINT constraint_name [RESTRICT|CASCADE]
SET TABLESPACE new_tablespace
```

For example, consider the `t_book` table that stores information about the books in the stock of the online `bookstore` shown in the diagram 2.1. We will use the `CREATE TABLE` command to create only selected columns of the table so that the rest of the columns will serve the purpose of illustrating how the `ALTER TABLE` command works.

```
CREATE TABLE t_book(
  ISBN int2 PRIMARY KEY,
  title varchar(40) NOT NULL,
  publisher int2,
  supplier int2,
  price decimal(4,2),
  volume int2,
  CHECK(price >= 0.0),
  CHECK(volume >= 0)
);
```

We will modify the initial structure of the `t_book` table using following commands:

- Add a column to the `t_book` table that stores the type of book binding, and limit the values stored in it to those from the given set:

  ```
  ALTER TABLE t_book
    ADD binding char(9),
    ADD CONSTRAINT t_book_binding_check
    CHECK(binding in ('paperback','hardcover'));
  ```

- Add a mandatory column `year` to the `t_book` table with a default value of 2024 with a 4-character regular expression match restriction:

  ```
  ALTER TABLE t_book
    ADD COLUMN year char(4),
  ```

```
    ALTER COLUMN year SET DEFAULT '2024',
    ALTER COLUMN year SET NOT NULL,
    ADD CONSTRAINT t_book_year_check
    CHECK (year ~ ('^\d{4}$'));
```

- Add foreign key definitions to the `t_book` table, provided that the `t_publisher` and `t_supplier` tables have been previously created:

```
ALTER TABLE t_book
 ADD CONSTRAINT key_from_supplier FOREIGN KEY(supplier)
 REFERENCES t_supplier(supplier_id)
 ON DELETE CASCADE;

ALTER TABLE t_book
  ADD CONSTRAINT key_from_publisher FOREIGN KEY (publisher)
  REFERENCES t_publisher(publisher_id)
  ON DELETE CASCADE;
```

2.3. Adding a New Data to the Table

There are multiple ways to add a new data to a table in SQL. One of them is directly inserting values into the table using the command `INSERT INTO`.

```
INSERT INTO table_name [(column [,...])]
{DEFAULT VALUES |
VALUES ({expression|DEFAULT}[,...])[,...] |
query}
[RETURNING * | output_expression [[AS] output_name][,...]]
```

Consider the example `t_order` table, which we showed in 2.1, which stores basic data about a single order: the identifier (`order_id`), which is the main key of the table, the identifier of the customer (`client_id`) who places the order, the date of the order placed (with the default value being the current system date) (`date_order`), a binary card payment confirmation field (`card`), and a binary field with order fulfilment completion confirmation (`completed`).

```
CREATE TABLE t_order (
  order_id int2 primary key,
  client_id int2,
  date_order DATE DEFAULT current_date,
  card int2,
```

```
  completed int2,
  CONSTRAINT key_from_client FOREIGN KEY(client_id)
  REFERENCES t_client(client_id) ON DELETE CASCADE
);
```

Inserting all column values (a complete row) into the table is done as follows:

```
INSERT into t_order VALUES(1, 1, '2007-01-10', 1, 1);
```

The order of the values provided must match the order of the columns created when the table was created. If you try to insert a row with values in a different order from the one created when you created the table, you will fail with an error message:

```
INSERT into t_order VALUES('2017-11-21', 1, 1, 2, 3);
ERROR: invalid input syntax for type smallint: "2017-11-21"
LINE 1: INSERT into t_order VALUES('2017-11-21', 1, 1, 2, 3);
```

An attempt to insert a row with a repeating value for the primary key:

```
INSERT into t_order VALUES(1, 2, '2012-10-13', 1, 1);
```

will yield following error:

```
ERROR: duplicate key value violates
       unique constraint "t_order_pkey"
DETAIL: Key (order_id)=(1) already exists.
```

If you want to insert all the values in a row into a table, you can explicitly define a list of columns to be completed, e.g.

```
INSERT INTO t_order
(order_id,client_id,date_order,card,completed)
VALUES(123, 3, DEFAULT, 1, 0);
```

Referencing `DEFAULT` will insert the default value defined upon table creation for the `date_order` column. By defining the list of columns to be completed, we also have the ability to change their order, different from the one set up when creating the table, e.g.

```
INSERT INTO t_order
(client_id,date_order,card,completed,order_id)
VALUES(1, DEFAULT, 0, 0, 124);
```

Moreover, instead of inserting a single record into the table multiple times, it is possible to modify the `VALUES` clause of the `INSERT INTO` command to include numerous records separated by commas, e.g.

```
INSERT into t_order VALUES
 (2, 2, '2004-01-10', 1,0),
 (3, 3, '2003-01-10', 0,0),
 (4, 2, '2002-01-11', 0,0),
 (5, 4, '2001-01-11', 1,1),
 (6, 5, '2008-01-11', 1,1),
 (7, 4, '2007-10-12', 1,1),
 (8, 4, '2010-01-12', 1,0);
```

By defining a list of columns to which values are to go, it is also possible to insert values only for selected columns in a row. It is important that mandatory value columns (`NOT NULL`), including the primary key column(s), are always completed, unless defined as `serial` (which automatically increments by one each time a new row is added) or default values are specified for these columns. The value inserted into columns that are not in the list and have not been defined as default values is `NULL`.
For example:

```
INSERT into t_order (order_id, client_id) VALUES(9, 3);
INSERT into t_order (order_id, date_order)
VALUES(10, '2012-01-12');
INSERT into t_order (order_id, card) VALUES(11, 1);
INSERT into t_order (order_id) VALUES(12);
```

In each of the above-mentioned records, only the order date (`date_order`) will take the default value of the current system date (`current_date`), the remaining columns not listed will be filled with `NULL` values. Attempting to insert a record omitting the value for the primary key will fail with an error message:

```
INSERT INTO t_order(client_id) VALUES(3);
ERROR: null value in column "order_id" violates not-null constraint
DETAIL: Failing row contains (null, 3, 2024-05-02, null, null).
```

The table 2.2 presents `t_order` table records after executing all previously presented, correct instructions `INSERT INTO`.

2.4. Modifying and Deleting Data

Often there is a need to modify or delete data already stored in the database. Command `UPDATE` is used to update data already in the table, while `DELETE` serves the purpose of deleting records.

Table 2.2: *t_ order* table records

order_id	client_id	date_order	card	completed
1	1	2007-01-10	1	1
123	3	2024-05-02	1	0
124	1	2024-05-02	0	0
2	2	2004-01-10	1	0
3	3	2003-01-10	0	0
4	2	2002-01-11	0	0
5	4	2001-01-11	1	1
6	5	2008-01-11	1	1
7	4	2007-10-12	1	1
8	4	2010-01-12	1	0
9	3	2024-05-02		
10		2012-01-12		
11		2024-05-02	1	
12		2024-05-02		

14 rows

```
UPDATE table_name [[AS] alias]
SET {column = {expression | DEFAULT} |
(column [,...]) = ({expression | DEFAULT } [,...])} [,...]
[WHERE condition]
```

Modifying the data of the table indicated by the name (`table_name`) is done by defining a new data set in the `SET` clause, for the columns indicated on the list with an optional condition that the data must meet in order to be changed. The lack of a data selection condition results in modifying the data in each table record in the columns indicated in the `SET` clause.

An example of using the `UPDATE` command would be to increase the price of each book from the selected three years by 5%. This is an example of using the value of an arithmetic expression.

```
UPDATE t_book
SET price = price * 1.05
WHERE year in ('1997','1999', '2012');
```

We are informed about the correctness by PostgreSQL with information about the number of modified records, e.g. `UPDATE 8`. To see how the data has been modified, run `SELECT` on it.

Sometimes we just need to remove a value from a row. We can use the UPDATE command for this purpose, simply assigning the value NULL to the fields in the selected columns. For example, we know that an error was made in the entered title of a book with an ISBN of 12. To avoid displaying incorrect data, until a valid title is entered, it will be changed to the undefined, empty value NULL.

```
UPDATE t_book
SET title = NULL
WHERE ISBN = 12;
```

However, there are also situations when we want to remove rows from the table. We can do this with the DELETE command.

```
DELETE FROM table_name
[WHERE condition];
```

Let's assume we want to delete records containing data about orders made before '1990-01-01':

```
DELETE FROM t_order
WHERE date_order < '1990-01-01';
```

If we want to delete all records from a table, but not the table itself, we can use the DELETE voivodeshipment without defining a condition. To illustrate deleting all rows from a table, we will create an example table Example, fill it with several records and then proceed to delete them.

```
CREATE TABLE Example (one int);
INSERT INTO Example VALUES(1),(2),(3);
DELETE FROM Example;
```

Another way to remove all data without deleting the table is to use the TRUNCATE TABLE voivodeshipment specifying the name of the table to be cleared of data:

```
TRUNCATE TABLE EXAMPLE;
```

2.5. Dropping a Table

To delete a table and all its rows, issue the DROP TABLE command in the psql client:

```
DROP TABLE [IF EXISTS] name [,...] [CASCADE | RESTRICT];
```

Here table_name is the name of the table to be deleted and the CASCADE option will delete all views associated with the table being deleted, but in the case of a foreign key

it will only delete the foreign key constraint and not the entire other table. This makes it possible to delete tables that are dependent on each other (in terms of foreign key) in any order. If you select RESTRICT, the table will be denied deletion if any objects depend on it. This is the default behaviour.

If we want to delete all data from the t_client table, including the table itself, and we know from the 2.1 diagram that it is related to the t_order table, we can only do it with the command:

```
DROP TABLE IF EXISTS t_client CASCADE;
```

Please note that using the cascading delete option (CASCADE) will remove the foreign key constraint in the t_order table, relating to the primary key client_id in the table being deleted, and all views associated with the table being deleted. Additionally, the IF EXISTS option will cause PostgreSQL to first check whether the table exists. If the table is not in the database, PostgreSQL skips the voivodeshipment and displays a table missing message, but does not report an error as in the voivodeshipment:

```
DROP TABLE t_client CASCADE;
ERROR:  table "t_client" does not exist
```

The DROP TABLE IF EXISTS voivodeshipment also allows you to automate the execution of SQL scripts, in which it is often used before the CREATE TABLE voivodeshipment. If the table exists, IF EXISTS will remove it before we recreate it. However, it is very important to make sure that the automatically deleted table is actually no longer in use.

2.6. Practice Exercises

Create a script named bookstore.create containing following voivodeshipments:

1. Create bookstore database.
2. Connect with the previously created bookstore database.
3. Cascading delete all tables shown on 2.1 diagram, if they exist in the database.
4. Create tables from the 2.1 diagram, as described below:

```
t_book:
  ISBN int2 primary key
  title varchar(40) mandatory attribute
  publisher int2
  year char(4) default '2024', mandatory attribute
    only the value is allowed in the year column
    in 4 digit format
  binding char(9) allowed values are 'paperback' or 'hardcover'
```

```
    supplier int2
    price decimal(4,2) wartość >= 0,0
    volume int2 wartość >= 0

  t_author:
    author_id int2 primary key
    name varchar(40)
    surname varchar(40)

  t_a_book:
    author_id int2
    ISBN int2

  t_publisher:
    publisher_id int2 primary key
    name varchar(10)
    address varchar(40)

  t_order:
    order_id int2 primary key
    client_id int2
    date_order DATE wartość domyślna current_date
    card int2
    completed int2

  t_z_book:
    order_id int2
    isbn int2
    shipping_date DATE default value current_date
    volume int2

  t_client:
    client_id int2 primary key
    name varchar(30)
    surname varchar(40)
    street varchar(30)
    city varchar(30),
    voivodeship varchar(30)
```

```
  zip_code char(6) limit on value
  to the format '00-000' ('0' means any digit character)
  telephon char(11) limit on value
  to the format '000-000-000' ('0' means any digit character)

t_supplier:
  supplier_id int2 primary key
  name varchar(30)
  street varchar(30)
  city varchar(30)
  voivodeship varchar(40)
  zip_code char(6) limit on value
  to the format '00-000' ('0' means any digit character)
  telephon char(11) limit on value
  to the format '000-000-000' ('0' means any digit character)
```

5. Modify the structure of the `t_a_book` table by adding a primary key consisting of two columns `author_id` and `ISBN` and foreign keys for the `ISBN` columns, which is a reference to the primary key of the table `t_book` and `author_id` which is a reference to the primary key of the `t_author` table.
6. Modify the structure of the `t_book` table by adding foreign keys for the columns: `publisher` which is a reference to the primary key of the table `t_publisher` and `supplier` which is a reference to the primary key of the `t_supplier` table.
7. Modify the structure of the `t_order` table by adding a foreign key for the `client_id` column, which is a reference to the primary key of the `t_client` table.
8. Modify the structure of the `t_z_book` table by adding foreign keys for the columns `order_id`, which is a reference to the primary key of the `t_order` table, and `ISBN`, which is a reference to the primary key of the `t_book` table.
9. Insert 10 rows into every created table.
10. Execute the script using `psql` client.

Chapter 3

Simple Database Queries

SQL language provides `SELECT` instruction allowing the user to fetch data from the database. It is most likely the most commonly used SQL query. It consists of five elementary parts:

- Operation - begins the `SELECT` statement and describes the operation that will be performed on the data indicated in the list of columns and functions.
- Data - the second clause, `FROM`, indicates the table(s) from which the data is retrieved, along with reserved keywords specifying what data should be included for calculation, filtering, and retrieval purposes.
- Condition - used to filter data that meets the given condition, `WHERE` clause.
- Grouping - the `GROUP BY` clause provides a key against which data is retrieved and combined, and then the result is calculated based on the values from all rows with the same key.
- Post-processing - the resulting data is retrieved and processed, often using the `ORDER BY` and `LIMIT` keywords.

In this chapter, we will discuss each of the above-mentioned parts, illustrating their operation with examples, and provide exercises to solve on your own. Since the query result (of the `SELECT` statement) will most often be directed to the standard output, i.e. the screen, we will often formulate queries verbally in such a way that they display the retrieved data (on the screen by default).

3.1. SELECT Query Syntax

We will now present the syntax of the `SELECT` command, limiting ourselves to those elements that appear most frequently in database queries and which will be discussed in this and subsequent chapters. For a detailed description of the `SELECT` command, the

interested reader is referred to the PostgreSQL documentation located at the following link: `https://www.postgresql.org/docs/15/index.html`.

```
[WITH [RECURSIVE] with_query [,...]]
SELECT [ALL | DISTINCT [ON (expression[,...])]]
* | expression [[AS] output_name][,...]
[FROM from_item [, ...] ]
[WHERE condition ]
[GROUP BY expression [, ...] ]
[HAVING condition [, ...] ]
[{ UNION | INTERSECT | EXCEPT } [ALL] select ]
[ORDER BY expression [ASC | DESC | USING operator]
[NULLS {FIRST | LAST}] [,...]]
[LIMIT {count | ALL}]
[OFFSET start [ROW | ROWS]];
```

where `from_item`:

```
[ONLY] table_name [*] [[AS] alias [(column_alias[,...])]]
(select) [AS] alias [(column_alias [,...])]
with_query_name [[AS] alias [(column_alias [,...])]]
function_name ([argument [,...]]) [AS] alias
[(column_alias [,...] | column_definition [,...])]
function_name([argument [,...]]) AS (column_definition [,...])
from_item [NATURAL] join_type from_item
[ON join_condition | USING (join_column [,...])];
```

a `query_name`:

```
with_query_name [(column_name [,...])] AS (select)
TABLE {[ONLY] table_name [*] | with_query_name}
```

Each query returns a set of elements - a virtual table (`Virtual Table`, VT). Sometimes it is an empty set (when the query returns nothing), sometimes it is a single-element set (it can be described by one or many attributes), but usually it contains a number of rows (records) and columns (described by many attributes). The shape of the result set is determined by the `SELECT` clause. This is where we predefine how we want to define the elements of the result set (what attributes, i.e. column names, will be used to describe them). The rows (records) may be described by all columns from the source tables (* symbol) or only by a subset of them. The simplest possible query consists only of the `SELECT` clause (it does not even refer to any table), in which a reference is made to a scalar value, e.g. a literal, a function (e.g. `current_date`), and the returned set is also

a scalar value - a single-element set, described by one attribute (one column). Thanks to this, we can, for example, check what date format is used in the PostgreSQL system by issuing the command:

```
SELECT current_date;
```

As a result we receive a single row depicted by a single column:

```
current_date
--------------
 2024-05-03
(1 row)
```

3.2. Fetching Data

The `SELECT` command allows you to search for information in a database using relational algebra operators. The query contains:

- the `SELECT` clause indicating the columns (attributes) whose values are to be displayed,
- the `FROM` clause indicating the table (relation) to which the command applies,
- the `WHERE` clause enabling a selection operation from the relational algebra,
- the `ORDER BY` clause enables the result sorting operation,
- the `LIMIT` clause allows downloading only the number of records limited to the number specified in the clause.

3.2.1. SELECT Clause

In the most general form, to retrieve all data from the `t_book` table, use the following query:

```
SELECT *
FROM t_book;
```

In the `SELECT` clause, the asterisk symbol (`*`) is a shortcut that allows you to retrieve all columns from the `t_book` table from the database in the order they were created in the `CREATE TABLE` command. The semicolon (;) marks the end of a statement and is mandatory in SQL syntax. (In the case of the `pgAdmin` client, a semicolon is automatically added to each executed SQL statement. Unfortunately, this may cause problems when we forget about it when writing queries, e.g. in the `psql` client.) Figure 3.1 shows the first few lines from the output of the above query.

If we want to query only specific columns, we must replace the asterisk (*) with the names of the selected columns. You must specify the columns in the order in which

isbn	title	publisher	year	binding	supplier	price	volume
1	Kontrabasista	1	1997	hardcover	1	20.40	5
2	Mercedes Benc	3	2001	hardcover	2	30.00	5
3	Tomik wierszy	2	1998	paperback	3	20.50	4
4	Cesarz laleczka	1	1978	hardcover	4	25.50	2
5	Lapidarium	1	1995	hardcover	5	35.10	2
6	Pan Tadeusz	4	1997	hardcover	6	20.00	5
7	Potop	5	2001	hardcover	7	30.00	5
8	Mazurek	6	1998	paperback	1	20.00	4

Figure 3.1: A fragment of all data displayed from the t_book table

you want the query to return data. For example, if we want to get the publisher column followed by the supplier column from the t_book table, we would execute the query:

```
SELECT publisher, supplier
FROM t_book;
```

The result of a projection limited to a small set of columns (or a single column) may be displaying the same information (multiple repetitions of the same records), as shown in figure 3.2.

publisher	supplier
1	1
3	2
2	3
1	4
1	5
4	6
5	7
6	1
6	2
5	3
4	4
3	5

Figure 3.2: A fragment of data displayed from the t_book table's selected columns

To eliminate repetition of lines we use in the SELECT clause the DISTINCT operator, e.g. by using it to retrieve unique values of the publisher column from the t_book table in the query. The query results are shown in figure 3.3

```
SELECT DISTINCT publisher
FROM t_book;
```

The SELECT clause can contain additional elements for column operations. As a result of their use, the columns should be given a new, alternative name - an alias, which appears immediately after the column name (the alias may be preceded by the keyword AS). An alias can be used for any column appearing in the SELECT clause, not necessarily appearing in expressions.

```
 publisher
-----------
         3
         5
         4
         6
         2
         1
(6 rows)
```

Figure 3.3: Unique values for the `publisher` column from the `t_book` table

Additionally, in the `SELECT` clause additional elements may occur such as:

a) **Literals** – any string, date or number; it is automatically attached to every displayed row, as placed in the `SELECT` clause, and to the heading of the column to which it applies.
As an example, we will formulate a query that will retrieve the book titles and the year of their publication from the `t_book` table, preceded by the literal: `’ published in the year ’`, which is a separate column. The query results are shown in the figure 3.4.

```
SELECT title, ’ published in the year ’ AS "Literal", year
FROM t_book;
```

```
         title           |         Literal          | year
-------------------------+--------------------------+------
 Kontrabasista           |  published in the year  | 1997
 Mercedes Benc           |  published in the year  | 2001
 Tomik wierszy           |  published in the year  | 1998
 Cesarz laleczka         |  published in the year  | 1978
 Lapidarium              |  published in the year  | 1995
 Pan Tadeusz             |  published in the year  | 1997
 Potop                   |  published in the year  | 2001
 Mazurek                 |  published in the year  | 1998
 Fortepian Chopina       |  published in the year  | 1978
```

Figure 3.4: Using the literal in the query

b) **Arithmetic expressions** – are constructed from column names and numeric literals using operators available in PostgreSQL (see the 1.6.2 subsection on the page 14).
By using a mathematical expression, we can, for example, retrieve information about the title of a book and the value of the product of their price (`price` column) and the number of copies (`volume` column) from the `t_book` table. The results are shown in the figure 3.5.

```
SELECT title, price*volume "Arithmetic expression"
FROM t_book;
```

```
          title          | Arithmetic expression
-------------------------+----------------------
 Kontrabasista           |                102.00
 Mercedes Benc           |                150.00
 Tomik wierszy           |                 82.00
 Cesarz laleczka         |                 51.00
 Lapidarium              |                 70.20
 Pan Tadeusz             |                100.00
 Potop                   |                150.00
 Mazurek                 |                 80.00
 Fortepian Chopina       |                 50.60
```

Figure 3.5: Using arithmetic expression in the query

c) **Functions** – let the user transform the values of the columns and literals to which they are applied. Detailed information about PostgreSQL functions can be found in the system documentation at the following link: `https://www.postgresql.org/docs/current/functions.html`
We will illustrate the use of the function using the example of the `COALESCE` function, which is used to replace the `NULL` value with another value.

Result type	Description	Example
`COALESCE(list)`	returns the first non-empty list value from (`list`); if all values are `NULL` will return `NULL`;	`COALESCE(NULL,1,2)`

To do this, we will add a record to the `t_publisher` table, providing only the value for the primary key. This way the remaining columns will assume the value `NULL`.

```
INSERT INTO t_publisher (publisher_id)
VALUES (7);
```

Next, we fetch the publishers' names.

```
SELECT name
FROM t_publisher;
```

Figure 3.6 shows the query result - among the six records returned, there is a single one with the value `NULL`, which is empty.
By using the `COALESCE` function, we can replace each `NULL` value with a different value in the returned result.
Apart from the fact that the `NULL` character has no graphical interpretation, it also "spoils" the calculations. So, in mathematical expressions, let's use a function that replaces each value of `NULL` with the value 0. The query result after using the `COALESCE` function is shown in the figure 3.7.

```
   name
-----------
 Czytelnik
 PIW
 Znak
 Helion
 Robomatic
 Znak

(7 rows)
```

Figure 3.6: NULL in the query result

```
SELECT COALESCE(name,'NULL was here') AS name
FROM t_publisher;
```

```
     name
---------------
 Czytelnik
 PIW
 Znak
 Helion
 Robomatic
 Znak
 NULL was here
(7 rows)
```

Figure 3.7: COALESCE function in the query result

Another way to manipulate the return value is to use the CASE select statement. The general format is:

```
CASE
  WHEN condition1 THEN value1
  WHEN condition2 THEN value2
  ...
  WHEN conditionN THEN valueN
  ELSE value
END;
```

Here condition1, condition2,..., conditionN are logical conditions, the truth of which determines the appropriate value. The select statement also includes the ELSE clause as a condition opposite to all previous conditions. For each record in the query, the conditions in the CASE statement are checked, and when the first condition with a value of True is detected, the select statement returns the value associated with that condition. If neither condition is met, the value associated with the ELSE block is returned.

Figure 3.8 shows the results of a query using `CASE` displaying book titles (`title` column) and their current price calculated as the price difference (`price` column) and the discount granted in depending on the number of copies (`volume` column) of a given book.

```
SELECT title, price, (price - CASE WHEN volume IS NULL THEN 0
                                   WHEN volume < 5 THEN 10
                                   ELSE -10 END) AS price_discount
FROM t_book;
```

```
         title            | price | price_discount
--------------------------+-------+----------------
 Kontrabasista            | 20.40 |          30.40
 Mercedes Benc            | 30.00 |          40.00
 Tomik wierszy            | 20.50 |          10.50
 Cesarz laleczka          | 25.50 |          15.50
 Lapidarium               | 35.10 |          25.10
 Pan Tadeusz              | 20.00 |          30.00
 Potop                    | 30.00 |          40.00
 Mazurek                  | 20.00 |          10.00
 Fortepian Chopina        | 25.30 |          15.30
```

Figure 3.8: The result of using `CASE` in query

d) **Concatenation function** (`||`, `concat()`, `concat_ws(separator, ...)`) – allows combining the values of various attributes (columns) displayed in the `SELECT` command into single strings.
Figures 3.9, 3.10 and 3.11 show the results of queries using the concatenation function for `||`, `concat()` and respectively `concat_ws(separator, ...)`, displaying concatenated column values from the table `t_book`: `ISBN`, `title` and `year`.

```
SELECT ISBN || ': ' || title || ' published in: ' ||
       year AS Concatenation
FROM t_book;
```

```
              concatenation
-----------------------------------------
 1: Kontrabasista published in: 1997
 2: Mercedes Benc published in: 2001
 3: Tomik wierszy published in: 1998
 4: Cesarz laleczka published in: 1978
 5: Lapidarium published in: 1995
 6: Pan Tadeusz published in: 1997
 7: Potop published in: 2001
 8: Mazurek published in: 1998
 9: Fortepian Chopina published in: 1978
```

Figure 3.9: The result of using || in query

```
SELECT CONCAT(ISBN, ': ', title, ' published in: ', year)
FROM t_book;
```

```
concat
----------------------------------------
1: Kontrabasista published in: 1997
2: Mercedes Benc published in: 2001
3: Tomik wierszy published in: 1998
4: Cesarz laleczka published in: 1978
5: Lapidarium published in: 1995
6: Pan Tadeusz published in: 1997
7: Potop published in: 2001
8: Mazurek published in: 1998
9: Fortepian Chopina published in: 1978
```

Figure 3.10: The result of using `CONCAT` in query

```
SELECT CONCAT_WS(' ', ISBN, title, 'published in:', year)
FROM t_book;
```

```
concat_ws
---------------------------------------
1 Kontrabasista published in: 1997
2 Mercedes Benc published in: 2001
3 Tomik wierszy published in: 1998
4 Cesarz laleczka published in: 1978
5 Lapidarium published in: 1995
6 Pan Tadeusz published in: 1997
7 Potop published in: 2001
8 Mazurek published in: 1998
9 Fortepian Chopina published in: 1978
```

Figure 3.11: The result of using `CONCAT_WS` in query

3.2.2. WHERE Clause

The `WHERE` clause allows you to select records that must meet the condition of this clause. The condition is usually a logical expression that evaluates to either `True` or `False` for each line. Logical expressions can use comparison operators, logical operators that combine more than one logical expression, and special operators to handle `NULL` values.
Among others, logical expressions use:

- The `IN` and `NOT IN` operators, which check whether a value retrieved from a column belongs to a set of one or more specified values:
  - The price should be one of the following values: 10, 20 or 30

    ```
    SELECT price
    FROM t_book
    WHERE price IN (10, 20, 30);
    ```

equivalently, the above condition can be written as follows:

```
SELECT price
FROM t_book
WHERE price = 10 OR price = 20 OR price = 30;
```

– When using logical operators in a logical expression, it is important to remember the operator priorities and the order in which they are executed when using a complex logical expression. When necessary, you should use round brackets () to determine the correct order of operations, e.g.

```
SELECT title, price
FROM t_book
WHERE title ilike '%a%' AND
      price = 10 OR price = 20 OR price = 30;
```

```
     title      | price
----------------+-------
 Mercedes Benc  | 30.00
 Pan Tadeusz    | 20.00
 Potop          | 30.00
 Mazurek        | 20.00
 Egzorcysta     | 30.00
(5 rows)
```

Figure 3.12: The result of the query with wrong order of operations

In the 3.12 figure, we present all the records returned by the above query from the `t_book` table, whose title contains the letter 'a' anywhere and whose price is 10, and additionally those records where the price is 20 or 30.
Meanwhile, a query with a designated order of operations in brackets:

```
SELECT title, price
FROM t_book
WHERE title ilike '%a%' AND
      (price = 10 OR price = 20 OR price = 30);
```

will return those records from the `t_book` table whose title contains the letter 'a' anywhere and whose price is one of the amounts 10, 20 or 30 (figure 3.13).

```
    title     | price
--------------+-------
 Pan Tadeusz  | 20.00
 Mazurek      | 20.00
 Egzorcysta   | 30.00
(3 rows)
```

Figure 3.13: The result of the query with the designated order of actions

- The selection of values from the given range is done by building a logical expression using appropriate comparison operators and logical operators and laws.
  - The price should be contained within the range of 5 to 15:

    ```
    SELECT title, price
    FROM t_book
    WHERE price >= 5 AND price <= 25;
    ```

    equivalently, the condition for a mutually closed interval can be constructed using the operator `BETWEEN AND`:

    ```
    SELECT title, price
    FROM t_book
    WHERE price BETWEEN 5 AND 25;
    ```

  - The price should be an amount outside the mutually closed range from 5 to 15:

    ```
    SELECT title, price
    FROM t_book
    WHERE price < 5 OR price > 25;
    ```

    or equivalently with the use of the `NOT` operator:

    ```
    SELECT title, price
    FROM t_book
    WHERE NOT (price >= 5 AND price <= 25);
    ```

    ```
    SELECT title, price
    FROM t_book
    WHERE price NOT BETWEEN 5 AND 25;
    ```

- Comparison with a case-sensitive pattern is made using the `LIKE` or `NOT LIKE` operators (the `ILIKE` or `NOT ILIKE` operators, respectively, ignore case), and the pattern itself is built between others using the following characters: `'_'` - representing any single character (mandatory) and `'%'` - representing any (especially empty) string of characters.

  Figure 3.14 shows the results of a case-sensitive query selecting the names and surnames of those customers who have three-letter names and the letter `'i'` at the end of their surname, and figure 3.15 the same query returning results ignoring case.

```
SELECT name, surname
FROM t_client
WHERE name like '___' and surname like '%I';
```

```
 name | surname
------+---------
(0 rows)
```

Figure 3.14: The result of using LIKE in query

```
SELECT name, surname
FROM t_client
WHERE name like '___' and surname ilike '%I';
```

```
 name | surname
------+----------
 Jan  | Kowalski
(1 row)
```

Figure 3.15: The result of using ILIKE in query

3.2.3. ORDER BY Clause

The ORDER BY clause appears as the penultimate clause in the SELECT statement and sorts rows resulting from the query according to the value of the attribute(s) indicated in it. If there are more sorting attributes, we separate them with commas. Sorting is then performed according to the first column in the list of columns in the ORDER BY clause, and only in the case of the same values appearing in it, sorting of the result records according to the next column is started. A separate sorting order can be set for each of the columns on the list against which records are sorted. By default the sort order is ascending ASC and to change the sort order to descending use DESC. Since the columns in the SELECT clause are assigned numbers starting from the value 1 for the first column in the list of columns of this clause and increasing by one for each subsequent one, in the ORDER BY clause instead of the column names, you can use the column names assigned to them. numbers.

```
SELECT ...
FROM ...
WHERE ...
ORDER BY nazwa_kolumn(y)[ASC/DESC][,...];
```

or

```
ORDER BY numer_kolumny[ASC/DESC][,...];
```

Figure 3.16 shows the results of a query showing the unique names and surnames of 5 authors in ascending order of their name and descending order of their surname when the names are the same.

```
SELECT DISTINCT name, surname
FROM t_author
ORDER BY name, surname
LIMIT 5;
```

```
  name   |  surname
---------+-----------
 Adam    | Mickiewicz
 Adam    | Zielony
 Beata   | Powstaniec
 Cyprian | Norwid
 Dariusz | Port
(5 rows)
```

Figure 3.16: The result of querying a limited number of retrieved records sorted by more than one column

Figure 3.17 shows the results of a query showing unique `name` and `surname` of 5 authors sorted in ascending order by name (1), for the same names descending sorting by surname (2 DESC) is performed.

```
SELECT DISTINCT name, surname
FROM t_author
ORDER BY 1, 2 DESC
LIMIT 5;
```

```
  name   |  surname
---------+-----------
 Adam    | Mickiewicz
 Adam    | Zielony
 Beata   | Powstaniec
 Cyprian | Norwid
 Dariusz | Port
(5 rows)
```

Figure 3.17: Query result with complex sorting of records by columns identified by numbers

3.3. Copying Data

In SQL, a table can also be created as a copy of existing data based on a `SELECT` query. Then the `CREATE TABLE` command has the form:

```
CREATE TABLE [IF NOT EXISTS] table_name [(col_name[, ...])]
AS QUERY;
```

Similarly, by using a `SELECT` query on data and obtaining the result set, we can insert this data as rows into the table using the `INSERT INTO` command in the form:

```
INSERT INTO tabel_name [(col_name[,...])]
QUERY;
```

As an example, we will create a table called `A`, but this time it will be a copy of the existing data selected with the `SELECT` command, which returns the title and price of the books, as long as the price is different from 20.

```
CREATE TABLE A AS
SELECT title, price
FROM t_book
WHERE price != 20;
```

We can check the structure of the newly created table (figure 3.18) by running command `\d A` in `psql` client and display its contents with the SELECT command (figure 3.19):

```
                     Table "public.a"
 Column |          Type          | Collation | Nullable | Default
--------+-----------------------+-----------+----------+---------
 title  | character varying(40) |           |          |
 price  | numeric(4,2)          |           |          |
```

Figure 3.18: The table `A` structure

```
SELECT *
FROM A;
```

```
          title           | price
--------------------------+-------
 Kontrabasista            | 20.40
 Mercedes Benc            | 30.00
 Tomik wierszy            | 20.50
 Cesarz laleczka          | 25.50
 Lapidarium               | 35.10
 Potop                    | 30.00
 Fortepian Chopina        | 25.30
 Pole dla wszystkich      | 35.00
 ...
 Madagaskar               | 28.80
 Niebo w ogniu            | 39.99
(43 rows)
```

Figure 3.19: Sample records from the `A` table

We also append the previously rejected data to the `A` table.

```
INSERT INTO A
SELECT title, price
FROM t_book
WHERE price = 20;
```

3.4. Practice Exercises

1. Display the unique values of the supplier column from the `t_book` table.
2. Display the ISBN of the books and the margin amount, which is 15% of the book price, sort the results descending by margin.
3. Display the ISBN and publishers of books supplied by supplier ID 2, sort the results ascending by the ISBN column.
4. Display the ISBN, price and volume of books, as long as the volume does not exceed 5 copies.
5. Display the ISBN, title and price of books whose number of copies is between 1 and 4.
6. Display the name and surname of those clients whose ID is 1 or 3.
7. Display the surname of those clients whose name contains the letter `'a'` anywhere in the string, sorting the results in ascending order.
8. Display the surname and address details of those clients whose name is 'Adam' and whose ID is not greater than 5.
9. Display the ISBN of those books published by `'PWN'` or `'HELION'` whose price ranges from PLN 8 to PLN 30 and do not have a paperback cover.
10. Display the names of clients whose name begins and ends with `'a'` in descending order.
11. Display book titles and prices. Moreover, if the price:
    - `< 30`, display the text: `'price below thirty'`,
    - `= 30` – display the text: `'price exactly thirty'`,
    - `> 30`, display the text: `'price over thirty'`.

    Sort the results ascending by book title and limit the results to 10 records.
12. Display ISBN, title and year information for books that were not published in either 1997 or 1998.
13. Display those records from the `t_publisher` table for which a name or address was not provided when inserting data into the table.
14. Display all book information and sort first in ascending order by publisher then descending by supplier.
15. Display the ID and name of publishers based on `'Woronicza'` Street, sort the results descending by ID.

16. Display the title of each book and the calculated product of its price and volume (name the column as `'stock status'`). Arrange the rows in descending order of status, books with the same status arrange in ascending order of titles.
17. Display the ID and name of publishers based on `'Woronicza'` Street, sort the results descending by ID.
18. Display information about orders (`id_order`, `date_order`) that were placed between November 20 and December 31, 2015.

To solve the practice exercises starting from task 19, you should use the built-in PostgreSQL functions. See the documentation under the link https://www.postgresql.org/docs/current/functions.html.

19. View client information (`client_id`, `surname`) who have a two-part surname. You can assume that the two-part surname is separated by a hyphen (the `'-'` character, you can use e.g. the `POSITION` function).
20. Display the ISBN and titles of books with zero copies.
21. Display all information about clients who live in the `'Mazowieckie'` Voivodeship (`state`) or whose name and surname are strings of the same length. (`LENGTH`)
22. Display data about clients whose surnames start with the letter `'M'`. Displayed text is to be padded to 20 characters with the character sequence `'#*.'` Use the `RPAD` function. What is the difference between `LPAD` and `RPAD`?
23. Display clients surnames in the first column. In the second column display only 3 characters from the surname starting with first character. Use the `SUBSTR` function.
24. Display data about authors whose surnames are longer than 6 characters and in a separate column to confirm their length. Sort the result by: the length of the surname in order from the largest to the smallest.
25. Display clients names in the first column and surnames in the second column, but replace each letter of `'a'` with `'X'`. Use the `REPLACE` function.
26. Display the authors' surnames in the first column and their surnames in the second column, but written backwards. Use the `REVERSE` function.
27. Display all information about authors whose surnames start with `'K'`. All letters in the surname must be lowercase and all letters in the name must be uppercase. Use the `LOWER` and `UPPER` functions.
28. Display the authors' surnames in the first column. In the second column, write down where the letter `'a'` first appeared. Use the `POSITION` function.
29. Display the ISBN and price of the books rounded to the nearest whole number. Use the `ROUND` function.
30. Display the order date in the default display format and in the format you set yourself. Use the `TO_CHAR` function.

31. Display the final shipment date of the order, which cannot exceed 7 days from the order placement. For example, use the `INTERVAL` function.

Chapter 4

Data Joining

Data needed for a task is usually spread across multiple tables. In that case a simple `SELECT` statement referring to a single table is not enough to retrieve data. SQL provides methods for horizontally joining related data from different tables and methods for vertically joining the results of several queries (at least two), which we will discuss in this chapter.

4.1. Joining Tables Horizontally

We connect tables based on the value of a common attribute, for example the value of the primary key - foreign key pair. In this case, it is necessary to use a unique column identifier. Because the names of columns containing primary and foreign keys are most often identical, the column names should be preceded with the table names, separated by a dot, e.g. `t_book.ISBN`. We can improve query readability by using aliases for the table names.

We combine tables according to the following guidelines:

1. We try to connect tables using columns storing a primary-foreign key pair.
2. We use the entire primary keys of the tables to join. If for some table a complex primary key (consisting of several attributes) has been defined, when connecting such a table we refer to the entire key.
3. We combine objects using columns of the same type, from the same domain (by domain we mean a set of data with the same meaning, e.g. the domain for the `name` column will be a set of names).
4. We prefix column names with an alias of the source object name, even if the names are unique - this way we will improve the readability of the query.
5. We limit the number of connected objects to the necessary minimum.

Regardless of the selected join type, as a result of processing the FROM clause, we always obtain a set of elements - a virtual table, described by all columns of the input tables. At this point, it does not matter whether we connect two or more tables with an internal or external join. The elements (records, rows) of the result table will always be determined by all attributes (columns) of the combined sets. In the SELECT clause, we narrow down the resulting set of attributes to those that interest us. After selecting the tables, we need to define the join type and their conditions. Here we put into practice knowledge about relational databases and ways of relating tables to each other. Please remember that regardless of the number of tables (three or more), combining them always comes down to performing the operation of joining two of them multiple times.

Since database theory is not the purpose of this textbook – we assume that the reader already knows the theory. We would only like to draw the attention and interest the reader to the fact that foreign keys defined within the `bookstore` database, which are constraints on columns in tables, increase data integrity by ensuring that a foreign key will never contain a value that is not present in the specified table. This is called `referential integrity`. Additionally, adding a foreign key constraint helps further improve database performance. However, most analytical databases do not use foreign keys.

4.1.1. Inner Join

An inner join joins different tables together based on a condition called a predicate. As a result of the join, we obtain a result table consisting of all the columns of the input tables, containing only those records for which the inner join conditions are met (in three-valued logic, the result must be `True`). Otherwise, the combination is omitted from the results.

Inner joins are typically written as follows:

```
SELECT column, [,...]
FROM T1 INNER JOIN T2 ON bool_exp | USING (join_column_list);
```

`INNER JOIN` is a symmetric join and it does not particularly matter whether we join table `T1` with `T2` or vice versa. Similarly with conditions in `ON`. For the sake of order, however, it makes sense to keep the order of attributes in `ON` on the same side as specifying the source tables to the `JOIN` operator. Of course, we can combine more than two tables in one query, which will be illustrated in the last example of this section.

Let's consider an example of retrieving data from different tables. As the result of joining `t_book` with `t_publisher`, in the result table, limited to only two columns `t_publisher.name` and `t_book.title` only those elements for which values of columns `t_book.publisher` (foreign key) and `t_publisher.publisher_id` (primary key) are the same will be included. Figure 4.1 shows the first five lines of the output.

```
SELECT t_publisher.name, t_book.title
FROM t_book INNER JOIN t_publisher
     ON t_book.publisher = t_publisher.publisher_id
LIMIT 5;
```

```
   name    |      title
-----------+-----------------
 Czytelnik | Kontrabasista
 Znak      | Mercedes Benc
 PIW       | Tomik wierszy
 Czytelnik | Cesarz laleczka
 Czytelnik | Lapidarium
(5 rows)
```

Figure 4.1: The table `t_book` joined to the table `t_publisher`

You can give aliases to tables in a join so that you don't have to type their entire names each time.

```
SELECT surname, ISBN
FROM t_author ta INNER JOIN t_a_book AS tak
ON ta.author_id = tak.author_id
LIMIT 5;
```

Referring to column names with reference to the name of the table in which they are located is important when there are columns with the same names in different tables. PostgreSQL must be clearly informed which column it is dealing with.

```
SELECT author_id, surname, ISBN
FROM t_author ta INNER JOIN t_a_book AS tak
ON ta.author_id = tak.author_id
LIMIT 5;
```

In this case, using the `ON` phrase in the `INNER JOIN` join will result in an error and the following message will appear on the screen:

```
ERROR:  column reference "author_id" is ambiguous
LINE 1: SELECT author_id, surname, ISBN
```

Signaling the need to change the query by specifying the name of the table in the `SELECT` clause from which the `author_id` column comes. In the case of coinciding column names in a join condition, `INNER JOIN` provides the syntax using `USING`, which includes the name of the join column in parentheses (). Here, the repeating column is removed from the result table and the query fetches the data and shows no errors. Figure 4.2 shows 5 records of the result table.

```
SELECT author_id, surname, ISBN
FROM t_author INNER JOIN t_a_book USING (author_id)
LIMIT 5;
```

```
 author_id | surname | isbn
-----------+---------+------
         2 | Suskind |    1
         2 | Suskind |    5
         2 | Suskind |   13
         2 | Suskind |   31
         3 | Kapusta |    4
(5 rows)
```

Figure 4.2: Join using the `USING` keyword

All examples presented so far have combined tables by columns which are also the foreign/primary keys of the tables.

The general principle of combining tables is that it can be done on any columns. Only one condition must be met - compatibility of the data types (domains) of the combined attributes. How we write the condition and whether it makes sense depends only on user - the SQL language does not introduce any restrictions here. Additionally, we can perform any operations on the values of the attributes on which we connect. Process them using scalar functions, perform arithmetic operations, string concatenation, etc..

PostgreSQL introduces yet another method of internal join, based on the system's own selection of columns against which tables are joined. This is `NATURAL JOIN`, which does not require a reference or condition for link columns, but when executed it looks for columns in the tables with the same name and type matching. Figure 4.3 shows the 5 records of the natural join result table.

```
SELECT author_id, surname, ISBN
FROM t_author NATURAL JOIN t_a_book
LIMIT 5;
```

```
 author_id | surname | isbn
-----------+---------+------
         2 | Suskind |    1
         2 | Suskind |    5
         2 | Suskind |   13
         2 | Suskind |   31
         3 | Kapusta |    4
(5 rows)
```

Figure 4.3: `NATURAL JOIN` example

When using an inner join, `NATURAL JOIN`, you must be sure that the join columns are selected correctly. Due to the possibility of more than one column with the same name

and the same type of data stored in it, a natural join may lead to an incorrect connection of tables (incompatible with the ERD diagram) and retrieve incorrect data. Therefore, it is the responsibility of the designer and user of databases to ensure the correct reflection of real data and the relationships between them (database integrity).

`INNER JOIN`s are `ANSI X3.135` compliant and should be used in production databases. However, there is another, older method of writing table joins, which has not been recommended since the `ANSI SQL:92` standard, and is simply said not to be used. This is an inner join in which the condition is defined in the `WHERE` clause instead of in `ON/USING`. Tables connected by a Cartesian product are listed sequentially in the `FROM` clause, separated by commas. The result table obtained as a result of this operation contains all the columns from each table listed and is filled with records that combine the record of one table with all the records of the other table, etc. Then, for each record of the result set, the join condition in the `WHERE` clause is checked and records that do not meet this condition are rejected.

Figure 4.4 shows the first five records of the result table of a query returning all information about the author and the title of the book he wrote. When joining tables, the `ta.*` reference allows all columns from the table aliased `ta`, i.e. the `t_author` table, to be included in the result table.

```
SELECT ta.*, title
FROM t_author ta, t_a_book tak, t_book tk
WHERE ta.author_id = tak.author_id AND tak.ISBN = tk.ISBN
LIMIT 5;
```

```
 author_id |  name   | surname |      title
-----------+---------+---------+-----------------
         2 | Patrick | Suskind | Kontrabasista
         2 | Patrick | Suskind | Lapidarium
         2 | Patrick | Suskind | Zimna woda
         2 | Patrick | Suskind | M jak masakra
         3 | Ryszard | Kapusta | Cesarz laleczka
(5 rows)
```

Figure 4.4: Joining tables using a Cartesian product with a predicate in the `WHERE` clause

The rewritten query above using an inner join looks like this:

```
SELECT ta.*, title
FROM t_author ta INNER JOIN  t_a_book tak USING(author_id)
     INNER JOIN t_book tk ON tak.ISBN = tk.ISBN
LIMIT 5;
```

Figure 4.5 shows selected columns of the result table created by joining all tables of the `bookstore` database. The data was limited to books published in a specific year and customers whose surname ends in `'ski'`.

```
SELECT t_client.surname AS Client, ISBN, t_author.surname AS Author,
       t_supplier.name AS Supplier, t_publisher.name AS Publisher
FROM t_client JOIN t_order USING(client_id)
     JOIN t_z_book USING (order_id)
     JOIN t_book USING (ISBN)
     JOIN t_a_book USING (ISBN)
     JOIN t_author USING (author_id)
     JOIN t_supplier ON supplier = supplier_id
     JOIN t_publisher ON publisher = publisher_id
WHERE year in ('1997','2002','1998')
      AND t_client.surname ILIKE '%ski';
```

```
   client   | isbn | author  | supplier | publisher
------------+------+---------+----------+-----------
 Kowalski   |    1 | Suskind | Goniec   | Czytelnik
 Kowalski   |    3 | Kundera | Konik    | PIW
 Malinowski |    3 | Kundera | Konik    | PIW
 Kowalski   |    8 | Wybicki | Goniec   | Znak
(4 rows)
```

Figure 4.5: Selected data from joined database tables `bookstore`

4.1.2. Outer Join

In an outer join, rows that do not meet the connection condition can be displayed in the query result, e.g. the outer join operator is used when records are missing on one side of the table allowing display of all records of the second table (after joining). An outer join creates additional empty records in an incomplete table. The number of empty records is equal to the number of records that do not meet the conditions for a classic join.

To show how the outer join works, we will add two records to the `t_publisher` tables:

```
INSERT INTO t_publisher
VALUES(8, 'PWN','02-460 Warszawa, ul. Daimlera 2');
INSERT INTO t_publisher
VALUES(9, 'Novae Res','81-368 Gdynia, ul. Nowa 9/4');
```

Outer joins can be divided into three categories:

- **Left outer join -** all rows of the table specified on the left side of the `JOIN` clause are returned. If the statement does not find a matching row from the second table

(given to the right of the operator), it returns the row `NULL`. These types of joins are created using the keywords `LEFT OUTER JOIN` followed by a join predicate. You can also use the shortened notation `LEFT JOIN`.
In figure 4.6, we show the result table of a left-side outer join of the `t_publisher` tables (on the left side of the `LEFT JOIN` operator) with the `t_book` table, in which the records are limited only to those that are not referenced in the `t_book` table and a `NULL` record is generated for them.

```
SELECT title, name
FROM t_publisher LEFT OUTER JOIN t_book
ON (publisher_id = publisher)
WHERE title is NULL;
```

```
 title |   name
-------+-----------
       | PWN
       | Novae Res
       |
(3 rows)
```

Figure 4.6: Left outer join

- **Right outer join** - all rows of the table specified on the right side of the `JOIN` clause are returned. If the statement does not find a matching row from the second table (given on the left), it returns the row `NULL`. These types of joins are created using the keywords `RIGHT OUTER JOIN` followed by a join predicate. You can also use the shortened notation `RIGHT JOIN`.
  Figure 4.7 shows the result table of a right-hand outer join of the `t_publisher` tables (to the right of the `RIGHT JOIN` operator) with the `t_book` table, in which the records are limited only to those that are not referenced in the `t_book` table and a `NULL` record is generated for them. Changing the order of the tables in the right join, symmetrically to the left join, generated the same result table.

```
SELECT title, name
FROM t_book RIGHT OUTER JOIN t_publisher
ON (publisher_id = publisher)
WHERE title is NULL;
```

- **Full outer join** - returns all rows from the left and right tables regardless of whether the join predicate is satisfied. For records for which the predicate is met, records from both tables are combined accordingly. Records for which the predicate is not met are populated with `NULL`. A full outer join is performed using the `FULL OUTER`

```
 title |   name
-------+-----------
       | PWN
       | Novae Res
       |
(3 rows)
```

Figure 4.7: Right outer join

JOIN clause followed by a join predicate. Once again we repeat the join of the t_book and t_publisher tables, this time in a full outer join (figure 4.8).

```
SELECT title, name
FROM t_book FULL OUTER JOIN t_publisher
ON (publisher_id = publisher)
WHERE title is NULL;
```

```
 title |   name
-------+-----------
       | PWN
       | Novae Res
       |
(3 rows)
```

Figure 4.8: Full outer join

4.1.3. Cartesian Product - CROSS JOIN

In SQL, there is a CROSS JOIN, which is a cartesian product without conditions. A rather rarely used combination of sets. It connects every row of table A with every row of table B. It is the only one that cannot create connection conditions in ON|USING, because it is supposed to connect everything to everything.

As an example, consider the need to perform a basket analysis of sales patterns across multiple books. This is a situation in which combining two titles and selling them at a promotional price brings benefits to the bookstore. For affinity analysis, we need to determine by cross join all combinations of two books that we can create based on the data set from the t_book table. In the example, we will limit ourselves to only 5 records of the result table shown in figure 4.9.

```
SELECT tk1.ISBN, tk1.title, tk2.ISBN, tk2.title
FROM t_book tk1 CROSS JOIN t_book tk2
LIMIT 5;
```

The statement, without the LIMIT limit, returns all possible combinations of specified values from the same table. The query results (according to the number of rows in the

```
 isbn |     title      | isbn |      title
------+----------------+------+-----------------
    1 | Kontrabasista  |    1 | Kontrabasista
    1 | Kontrabasista  |    2 | Mercedes Benc
    1 | Kontrabasista  |    3 | Tomik wierszy
    1 | Kontrabasista  |    4 | Cesarz laleczka
    1 | Kontrabasista  |    5 | Lapidarium
(5 rows)
```

Figure 4.9: Cross join example

t_book table of the sample database bookstore - 44 rows) include 1936 records. In practice, cross joins are not used due to the risk of blocking the database and causing it to crash as a result of creating a large (hundreds of billions!) number of records. We leave the use of cross joins to the discretion of the database system user, counting on their extreme caution.

A CROSS JOIN join is also implemented when we specify the tables in the FROM clause, separating them only with a comma.

```
SELECT *
FROM t_publisher, t_supplier;
```

4.2. Connecting Vertical Connection

In order to vertically connect relations, we use one of the collective operators:

```
<SELECT . . . > UNION [ALL] <SELECT . . . >
<SELECT . . . > INTERSECT [ALL] <SELECT . . . >
<SELECT . . . > EXCEPT [ALL] <SELECT . . . >
```

These operators operate on the results of two or more SELECT operations. The same number of columns of compatible types must appear in SELECT clauses combined with the collective operator. The query results in attribute names only from the first SELECT clause. SELECT commands are performed in the order they occur (top to bottom). To change this order, we use parentheses. If you need to use the ORDER BY clause, it must appear as the last clause in the query. Moreover, in the ORDER BY clause we do not use column names, but their ordinal numbers.

```
<SELECT ...> OPERATOR <SELECT ...> <ORDER BY ...>;
```

In the example below, we combine the results of two queries into one table: the first one identifying the titles and publishers of books from the supplier with ID 1 and the second one limited to the same columns (it is important that the columns in the SELECT

clause are listed in the same order in both queries), but for the supplier with ID 3. We arrange the data in descending order by book title.

```
SELECT title, publisher FROM t_book WHERE supplier = 1
UNION ALL
SELECT title, publisher FROM t_book WHERE supplier = 3
ORDER BY 1 DESC;
```

4.3. Practice Exercises

1. Display all information about authors whose book price is in the range of 10 up to 15. Arrange the lines in descending order by author's surname.
2. Display for each book the title, its publisher, and its supplier. Rows arrange in descending order of titles.
3. Display the ISBN, title, supplier name, and client surname of these books, which has been written by an author named `'Adam'`.
4. Display the name of the publisher who published books with an ISBN in the range `⟨1,10⟩`.
5. Display the titles of books for which the order has already been completed.
6. Display the names of publishers whose books are not in the `bookstore`.
7. View the books' ISBN, author's name and surname, and the name and surname of those clients who ordered more than one book.
8. Display data about clients who have selected paperback books and their order was submitted before January 1, 2001.
9. Display information about those suppliers whose headquarters are on the same street as the street where the client lives.
10. Display `client_id`, `name`, and `surname` of clients who ordered books published in 2012.
11. Display the ISBN, title, author name and surname of these books were published by 'PIW', 'Znak' or 'Helion'.
12. List the ISBN, title, name and surname of the author of those books that are not were delivered by 'UPS' or 'Konik'.
13. Display book titles and prices. Also, if the price is less than 30, display the text: 'below 30', if the price is equal to 30 – display the text: 'equal to 30', and if the price is greater than 30, display the text: 'above 30'. Sort the rows ascending by price. Use the appropriate collective operator.
14. Display the names of publishers whose books are not yet available in the bookstore. Use the appropriate collective operator.

15. Display all information about books whose price is in the range <11.50,33.50> or in the telephone number of the client who bought them there is a 3 or 4 sign in any position.
16. Display information for each client about how many days have passed since the date of the order placed by him.
17. Display the client's surname and the name of the publishing house, the name of the supplier, as long as the client's place of residence, the publisher's headquarters and the supplier's headquarters are in the same city.
18. Display `client_id`, `name` and `surname` of those clients from the 'podlaskie' Voivodeship who ordered more than one copy of the book.
19. Display how many books were ordered by a customer named 'Anna' or 'Tadeusz'.
20. Print `ISBN`, title, first and surname of the author of those books whose title length does not exceed the sum of the lengths of the author's name and surname.

Chapter 5

Aggregate Functions

This chapter will present the logic of aggregation functions and the possibility of modifying their operation using the keywords `GROUP BY` and `HAVING`. For details, see the PostgreSQL documentation at `https://www.postgresql.org/docs/15/functions-aggregate.html`.

5.1. Functions Operating on Groups of Rows

Often, the purpose of the query is gaining a better understanding of the properties of the entire column or table, and not only viewing respective rows of data. SQL provides functions that can be used to perform calculations on large groups of rows. These are known as aggregate functions:

- `COUNT([DISTINCT|ALL] *|col)` - returns the number of rows from the `col` column having a non-empty value (different from `NULL`. The `'*'` character - causes all rows in the group to be evaluated, even those containing the value `NULL` (number groups), in a special case returning the size of the table.
- `MAX([DISTINCT|ALL] col)` - returns the max value of the `col` column within the rows group. For text columns the last value in alphabetical order is returned.
- `MIN([DISTINCT|ALL] col)` - returns the min value of the `col` column within the rows group. For text columns the first value in alphabetical order is returned.
- `SUM([DISTINCT|ALL] col)` - calculates and returns the sum of all values in the `col` column within the rows group.
- `AVG([DISTINCT|ALL] col)` - calculates and returns the average of all values in the `col` column within the rows group.
- `STDDEV([DISTINCT|ALL] col)` - returns the standard deviation for a sample based on all values in the `col` column.

- `VARIANCE([DISTINCT|ALL] col)` - returns the variance for a sample based on all values in the `col` column.
- `REGR_SLOPE(col1, col2)` - returns the slope of the regression line for the `col1` column as the dependent variable and the `col2` column as the independent variable.
- `REGR_INTERCEPT(col1, col2)` - returns the intercept of the linear regression line with column `col1` as the dependent variable and column `col2` as the independent variable.
- `CORR(col1, col2)` - Calculates and returns the Pearson correlation coefficient for data from `col1` and `col2` columns.

Aggregate functions can be useful in efficient execution of certain tasks, e.g.

- Aggregate functions can be used for all records in a table, e.g.
  - in order to check the number of records stored in the table:

```
SELECT COUNT(ISBN) AS "Rows number"
FROM t_book;
```

  - in order to calculate the total value of all stored objects:

```
SELECT SUM(price*volume) AS "Total value"
FROM t_book;
```

  - in order to calculate the mean value of all stored objects:

```
SELECT ROUND(AVG(price),2) AS "Average Price"
FROM t_book;
```

  - to find the title of the book in the last position in alphabetical order:

```
SELECT MAX(title) AS "The last aplhabetically"
FROM t_book;
```

- Aggregate functions can also be used with the `WHERE` clause, thus limiting the calculation to a group of records that meet the condition of the clause `WHERE`, e.g.
  - to check how many books have a price higher than 25:

```
SELECT COUNT(*) AS "Number of books"
FROM t_book
WHERE price > 25.00;
```

  - to check how many different clients placed orders after January 1, 2000:

```
SELECT COUNT(DISTINCT client_id) AS "Number of clients"
FROM t_order
WHERE date_order > '2000-01-01';
```

  - to calculate the sum of the prices of books delivered by suppliers with IDs 2 or 5:

```
SELECT SUM(price) AS "Sum"
FROM t_book
WHERE supplier in (2,5);
```

- You can also combine aggregation functions using mathematical operations, e.g. to calculate the average price of books you can use the functions: SUM and COUNT:

  ```
  SELECT ROUND(SUM(price)/COUNT(*),2) AS "Average"
  FROM t_book;
  ```

  CAUTION: depending on the arguments the division operator (/) may return the integer part of division when the arguments are integers or a real number when at least one argument is a floating point type. Therefore, to make sure the result is a real number one can simply multiply one of the arguments by 1.0 or cast it to a floating point number.

  ```
  COUNT(DISTINCT publisher)*1.0 / COUNT(*)
  COUNT(DISTINCT publisher)::numeric / COUNT(*)
  ```

5.1.1. GROUP BY Clause

The GROUP BY clause allows you to divide the rows of a relationship into groups. Rows in the same group have the identical value of the grouping attribute that is indicated in the clause. After division, one group function can be applied to each group. The GROUP BY clause can be used recursively to separate subgroups within previously separated groups. The order in which relationships are divided into groups and subgroups corresponds to the order of the grouping columns. Using group functions in the SELECT clause prevents the ability to display the n column values of individual rows unless the n column names are listed in the GROUP BY clause.

For example, the GOUP BY clause separates a group of rows from the t_book table with the same value of the publisher column, thus grouping them by publisher IDs. The COUNT(*) grouping function is then executed independently for each group, counting the number of rows within the group.

```
SELECT publisher, COUNT(*) AS "Number of books"
FROM t_book
GROUP BY publisher;
```

It is also possible to use the number of the column to perform a grouping operation, instead of its name like in the previous example.

```
SELECT publisher, COUNT(*) AS "Number of books"
FROM t_book
GROUP BY 1;
```

The key value in the GROUP BY operation can also be the result of calling a function for a column (or columns), e.g. you can extract the year from the order date and count the number of orders in a given year:

```
SELECT TO_CHAR(date_order,'YYYY'), COUNT(*) AS "Number of orders"
FROM t_order
GROUP BY TO_CHAR(date_order,'YYYY');
```

The next example illustrates grouping for several columns. First, the lines representing the year '1997' are discarded. Then the rows are grouped by publisher ID (publisher column). Then, each group is divided into subgroups according to the supplier (supplier column). Finally, COUNT() is executed for each subgroup. In other words, we determine the size of different groups according to the supplier (excluding those from the year '1997'), within the previously separated groups according to publishers.

```
SELECT publisher, supplier, COUNT(*)
FROM t_book
WHERE year != '1997'
GROUP BY publisher, supplier
ORDER BY publisher, supplier DESC;
```

PostgreSQL also allows you to create several categories based on which values are grouped by using the GROUP BY clause with the GROUPING SETS subclause as:

```
SELECT c1, c2, aggregate_function(c3)
FROM table_name
GROUP BY
    GROUPING SETS (
        (c1, c2),
        (c1),
        (c2),
        ()
);
```

where in this case (c1, c2), (c1), (c2), () are the four possible groups (the order is not important).

An example of using the GROUPING SETS subclause is to count books by individual publishers and, at the same time, in the same aggregation function, determine the total number of books in particular years for each publisher. Rows containing NULL in the year column contain information about the total number of books by each publisher (figure 5.1).

```
SELECT publisher, year, COUNT(*)
FROM t_book
GROUP BY GROUPING SETS (
  (publisher),
  (publisher, year)
)
ORDER BY 1, 2;
```

```
 publisher | year | count
-----------+------+-------
         1 | 1978 |     6
         1 | 1995 |     3
         1 | 1997 |     2
         1 | 2007 |     1
         1 |      |    12
         2 | 1998 |     6
         2 |      |     6
         3 | 2001 |     6
         3 |      |     6
         4 | 1997 |     6
         4 |      |     6
         5 | 1995 |     3
         5 | 2001 |     3
         5 |      |     6
         6 | 1978 |     3
         6 | 1995 |     3
         6 | 1998 |     3
         6 |      |     9
(18 rows)
```

Figure 5.1: Resulting query with subclause `GROUPING SET`

The above solution can also be achieved by applying the collective operator `UNION ALL` to the result sets of two queries as follows:

```
(
  SELECT publisher, NULL as year, COUNT(*)
  FROM t_book
  GROUP BY 1, 2
)
  UNION ALL
(
  SELECT publisher, year, COUNT(*)
  FROM t_book
  GROUP BY 1, 2
)
ORDER BY 1, 2;
```

A sorted set will be returned, which is the sum of the data sets returned by two queries: the first query returns a set of data grouped by publishers, and the second query returns a set of data grouped by publishers and years of release.

Using the `UNION ALL` collective operator produces the same results as using the `GROUPING SETS` subclause, but the former may require writing very long queries and is therefore more prone to errors.

The aggregation functions described so far did not require sorted data as input. However, there are aggregate statistics whose calculation depends on the order of the values. For such functions, SQL provides a set of aggregate functions for ordered sets, including:

- `MODE()` - returns the value that occurs most often. If there are several such values, the one that appears earliest is returned.
- `PERCENTILE_CONT(position_as_fraction)` - returns a value from the ordered data set with the position given as a fraction. If necessary, the value is interpolated from adjacent inputs.
- `PERCENTILE_DISC(position_as_fraction)` - returns the first input data value whose position in the ordered data set is equal to or greater than the given fraction.

The syntax of such functions is:

```
SELECT fuction_for_ordered_data
WITHIN GROUP (ORDER BY classificatory_col)
FROM table_name;
```

For example, thanks to the `PERCENTILE_CONT` function, we can calculate the median of book prices for a function value of 0.5 (50% in fractional notation):

```
SELECT PERCENTILE_CONT(0.5)
WITHIN GROUP (ORDER BY price) AS Mediana
FROM t_book;
```

5.1.2. HAVING Clause

Sometimes we are only interested in certain rows from the results of an aggregate function that have certain characteristics, so we want to keep only those rows in the query output and remove the rest. To filter the results of aggregate functions, you need a special `HAVING` clause that always appears after the `GROUP BY` clause. It allows you to apply a filter to already aggregated data, and not to the original set of data to which we apply the `WHERE` clause condition.

We will use the `HAVING` clause to calculate the average price of books from each publisher, if the bookstore has more than two books published by him:

```
SELECT publisher, ROUND(AVG(price),2)
FROM t_book
GROUP BY publisher
HAVING COUNT(1) > 2;
```

5.2. Data Cleansing and Quality Control

By using aggregate functions you can determine not only which columns are missing values, but also how the missing data affects the calculations and whether this makes the missing column usable at all.

One way to check whether a column contains null values is to use the `CASE WHEN` statement with the `SUM` and `COUNT` functions. In this way, we will additionally determine the percentage of missing data.

```
SELECT SUM(
  CASE WHEN name is NULL OR name in ('Znak', 'PIW')
  THEN 1 ELSE 0
  END) :: float / COUNT(*) AS "Is empty?"
FROM t_publisher;
```

The data set obtained as a result of the query can be used in various ways, e.g. you can supplement the data if it is missing less than the designated limit or if there is more of it, you should eliminate the column from the analyses to obtain reliable conclusions.

If, on the other hand, you are only interested in `NULL` values, you can use only the `COUNT` function to determine both the percentage of values equal to and different from `NULL`.

```
SELECT ROUND(COUNT(name)*1.0/COUNT(*),4) as non_null_name,
1 - ROUND(COUNT(name)*1.0/COUNT(*),4) as null_name
FROM t_publisher;
```

Another common task is to check whether each value in a column is unique. For columns that are primary keys, uniqueness is ensured. However, we cannot always add `PRIMARY KEY` or `UNIQUE` constraints for columns. You can then use aggregation functions to check the uniqueness of the data.

One solution is to use the `COUNT` function on the rows grouped by column in the `GROUP BY` clause and add to the result table only those rows that have more than one in the group (`HAVING` clause. If the result set is empty, which means that all values in the column are unique.

```
SELECT publisher_id, COUNT(*)
FROM t_publisher
GROUP BY publisher_id
HAVING COUNT(1) > 1;
```

Another solution could be to use a Boolean expression in the `SELECT` clause. If the query returns the value `True`, it means that the column contains only unique values. Otherwise, there is one or more values that are repeated.

```
SELECT COUNT(DISTINCT publisher_id) = COUNT(*) as equal_ids
FROM t_publisher;
```

5.3. Window Function

When analyzing data stored in a database, we sometimes need to know the characteristics of a data point, such as its place in the data set. A typical example is a position that is calculated from both the measurement itself and the data set in which it is found. There may also be subgroups (partitions) in one data set on which the item is based. In a subgroup, you must select the rows included in the calculation so that you can determine the position based on the number of rows before the current row. These selected lines form a `window`. For a given set, the goal is to obtain a result for each row. This result depends on both the row value, the window, and the data set itself. The function that is used to perform this type of calculation is the `window function`.

The basic window function syntax consists of following elements:

```
SELECT [column [, column1],]
window_function OVER (
  PARTITION BY partition_key
  ORDER BY ordering_key
)
FROM tabel_name;
```

The command is used to select columns specified in the `SELECT` clause that come from the table or table joins specified in the `FROM` clause using the window function whose definition begins with the keyword `OVER`. The window function definition includes a column or columns used for partitioning (`PARTITION BY`) and a column or columns used to organize the data.

Any aggregation functions can be used as window functions. Let's consider an example of using the `COUNT` function as a window function.

```
SELECT client_id, name, surname,
COUNT(*) OVER () as Total
FROM t_client
ORDER by client_id;
```

```
 client_id |   name   |  surname   | total
-----------+----------+------------+-------
         1 | Jan      | Kowalski   |     5
         2 | Tadeusz  | Malinowski |     5
         3 | Krystyna | Torbicka   |     5
         4 | Anna     | Marzec     |     5
         5 | Adam     | Koper      |     5
(5 rows)
```

Figure 5.2: Query result with `COUNT` as window function

Figure 5.2 shows the result of the query, in which a column with information about the number of all clients has been added. This value corresponds to the query result:

```
SELECT COUNT(*)
FROM t_client;
```

Now we will use the window function and its `PARTITION BY` clause `city` column so that SQL splits the dataset into multiple subgroups based on the unique values of this column. For each subgroup, we then calculate the number of rows using the `COUNT(*)` function. Thanks to this, for each city in the `Total` column the number of lines with this value will be displayed, see figure 5.3.

```
SELECT client_id, name, surname, city,
COUNT(*) OVER (PARTITION BY city) as Total
FROM t_client
ORDER by client_id;
```

```
 client_id |   name   |  surname   |   city   | total
-----------+----------+------------+----------+-------
         1 | Jan      | Kowalski   | Warszawa |     3
         2 | Tadeusz  | Malinowski | Warszawa |     3
         3 | Krystyna | Torbicka   | Warszawa |     3
         4 | Anna     | Marzec     | Lipsk    |     2
         5 | Adam     | Koper      | Lipsk    |     2
(5 rows)
```

Figure 5.3: Query result with `PARTITION BY` clause

We will now use the `ORDER BY` clause in a window function ordering by `client_id`.

```
SELECT client_id, name, surname, city,
COUNT(*) OVER (ORDER BY client_id) as Total
FROM t_client
ORDER by client_id;
```

```
 client_id |   name    |  surname   |   city   | total
-----------+-----------+------------+----------+-------
         1 | Jan       | Kowalski   | Warszawa |     1
         2 | Tadeusz   | Malinowski | Warszawa |     2
         3 | Krystyna  | Torbicka   | Warszawa |     3
         4 | Anna      | Marzec     | Lipsk    |     4
         5 | Adam      | Koper      | Lipsk    |     5
(5 rows)
```

Figure 5.4: Query result with `ORDER BY` clause

Figure 5.4 shows the query results in which in the column `total` you can see a growing sum that determines the total number of clients. Since there is no `PARTITION BY` clause defined, the entire data set is included in the calculations. When a data set does not also have a `ORDER BY` clause specified, only one window covering the entire data set is used. However, if we use the `ORDER BY` clause, the rows in the group will be ordered according to it and for each unique value from the data so ordered, SQL will create a group for a given value. Such a group contains all rows in which this value appears. The query creates a window for each such group, including all rows in that group and all previous rows. In the example in Figure 5.4, the data set is ordered according to the primary key of the table `t_client`. Therefore, each row, having a unique value, creates a separate group. There are no other rows before the first group consisting of only one, first row, so a separate, single-line window is created. The window for the second group includes this group and the previous row. In this way, subsequent windows are created incrementally for each group. After determining the windows for each group, the number of rows included in it is calculated. The result is assigned to rows from individual value groups.

Now we will use both clauses in the window function: `PARTITION BY` and `ORDER BY`.

```
SELECT client_id, name, surname, city,
COUNT(*) OVER (
  PARTITION BY city
  ORDER BY client_id
) as Total
FROM t_client
ORDER by client_id;
```

Figure 5.5 shows the results of the query, where within a given group divided by city we received something like a ranking. `PARTITION BY` clause divides the data set into

```
 client_id |   name   |  surname   |   city   | total
-----------+----------+------------+----------+-------
         1 | Jan      | Kowalski   | Warszawa |     1
         2 | Tadeusz  | Malinowski | Warszawa |     2
         3 | Krystyna | Torbicka   | Warszawa |     3
         4 | Anna     | Marzec     | Lipsk    |     1
         5 | Adam     | Koper      | Lipsk    |     2
(5 rows)
```

Figure 5.5: Query result with `PARTITION BY` and `ORDER BY`

groups (called subgroups or partitions here) based on the values in the `city` column, then each subgroup is used to count rows and each has its own set of value groups. These groups of values are organized into a subgroup and based on them, windows are created, their order is determined, and the window function is run. The final results are assigned to individual rows of value groups.

We will now use the `SELECT` command syntax with the `WINDOW` clause, which shortens and simplifies the creation of window aliases. First, let's consider a query that uses the same window for different functions to calculate the cumulative count of customers and the sum of the cumulative count of customers whose surname contains the letter `'k'` anywhere in the name (case-insensitive).

```
SELECT client_id, name, surname, city,
COUNT(*) OVER (
  PARTITION BY city
  ORDER BY client_id
) as Total_Count,
SUM(CASE WHEN surname ILIKE '%k%' THEN 1 ELSE 0 END) OVER (
  PARTITION BY city
  ORDER BY client_id
) as Total_Sum
FROM t_client
ORDER by client_id;
```

We simplify the above code by using the `WINDOW` clause.

```
SELECT client_id, name, surname, city,
COUNT(*) OVER w as Total_Count,
SUM(CASE WHEN surname ILIKE '%k%' THEN 1 ELSE 0 END) OVER w as Total_Sum
FROM t_client
WINDOW w AS (
  PARTITION BY city
  ORDER BY client_id
) ORDER by client_id;
```

```
 client_id |   name   |  surname   |   city   | total_count | total_sum
-----------+----------+------------+----------+-------------+-----------
         1 | Jan      | Kowalski   | Warszawa |           1 |         1
         2 | Tadeusz  | Malinowski | Warszawa |           2 |         2
         3 | Krystyna | Torbicka   | Warszawa |           3 |         3
         4 | Anna     | Marzec     | Lipsk    |           1 |         0
         5 | Adam     | Koper      | Lipsk    |           2 |         1
(5 rows)
```

Figure 5.6: Query result with the `WINDOW` clause

Figure 5.6 shows the results of the query using the `WINDOW` clause (same as query without using this clause). In case `name` does not meet the selection condition in the `SUM` function, in the `Total_sum` column zero appears.

In addition to the aggregate functions that can appear as window functions, various statistical functions listed in the 5.1 table can also appear in this role. Typically, when any of these functions are called, the keyword `OVER` appears, followed by optional `PARTITION BY` clauses in parentheses. and `ORDER BY`.

Table 5.1: Statistical window functions

Function	**Explanation**
`ROW_NUMBER`	Specifies the current line number in the group; numbering starts from 1.
`RANK`	Returns the position of the row in the group; does not create gaps.
`DENSE_RANK`	Returns the position of the row in the group; creates gaps.
`LAG`	Returns the value obtained for the group line that precedes the current one by the offset.
`LEAD`	Returns the value obtained for the group row following the current one by the offset.
`NTILE`	Divides the rows in a group into as equal as possible bins and assigns integers from to the argument value to each row.

As an example, we will define a query that uses the `RANK` function to sort clients according to the date of orders, divided by the customer's city of residence. Figure 5.7 shows the query results.

```
SELECT client_id, name, surname, city, date_order,
RANK() OVER w as client_rank
FROM t_client JOIN t_order USING(client_id)
WINDOW w AS (
  PARTITION BY city
  ORDER BY date_order
)
ORDER by client_id;
```

```
 client_id |   name   |  surname   |   city   | date_order | client_rank
-----------+----------+------------+----------+------------+------------
         1 | Jan      | Kowalski   | Warszawa | 2007-01-10 |           4
         1 | Jan      | Kowalski   | Warszawa | 2012-05-12 |           6
         2 | Tadeusz  | Malinowski | Warszawa | 2002-01-11 |           1
         2 | Tadeusz  | Malinowski | Warszawa | 2004-01-10 |           3
         3 | Krystyna | Torbicka   | Warszawa | 2003-01-10 |           2
         3 | Krystyna | Torbicka   | Warszawa | 2011-01-12 |           5
         4 | Anna     | Marzec     | Lipsk    | 2001-01-11 |           1
         4 | Anna     | Marzec     | Lipsk    | 2007-10-12 |           2
         4 | Anna     | Marzec     | Lipsk    | 2010-01-12 |           4
         4 | Anna     | Marzec     | Lipsk    | 2012-04-12 |           6
         5 | Adam     | Koper      | Lipsk    | 2008-01-11 |           3
         5 | Adam     | Koper      | Lipsk    | 2012-01-12 |           5
(12 rows)
```

Figure 5.7: Data sorted by order date

5.4. Practice Exercises

1. Display the minimum, average and maximum price of books.
2. View the minimum and maximum prices for books from individual publishers.
3. Display the number of books shipped by supplier ID 1.
4. Display the average price and average price of all copies of books by individual publishers, arrange the data according to the increasing value of the average book price.
5. Display the difference between the highest and lowest price of a book.
6. Display the names of publishers who have published more than 3 books.
7. Check that all book numbers are unique.
8. Display the lowest book price in each author's group. Ignore groups where the minimum price is lower than 25. Sort the results by increasing prices.
9. Display the number of books shipped in each month of the year.
10. Calculate and display the maximum price of books published in any year in which Cyprian Norwid's books were also published.
11. Display `client_id`, `name` and `surname` of those clients from the 'mazowieckie' Voivodeship who ordered exactly one book.
12. Display any author name that is repeated.
13. Display the ID of each order whose value is greater than 200.
14. Display the lowest book price in each author's group. Ignore groups where the minimum price is lower than 25. Sort the results by increasing prices.
15. Display author information for books that have never been sold.
16. Display how many books were sold (we only take into account completed orders) in particular months of the year (we take into account the order date).
17. Display the minimum price for books published in a year other than: `'1978', '1995' or '1998'`.

18. Display the ISBN, title, author name and surname of those books ordered by clients with different name and surnames (e.g. `Adam Kowalski`).
19. Display the alphabetically oldest author's name that is repeated.
20. Display the order ID with the youngest order fulfilment date (`shipping_date` column).

Chapter 6

Query Nesting

In the relational data model, each query returns a two-dimensional result set. As one might expect, there is a way to use data from result tables in other queries. It is as easy as placing the query in round brackets in the appropriate clause of the `SELECT` command, give it an alias if necessary, and analyse the resulting data further. These queries are often called sub-`SELECT` statements, or subqueries, because they allow a `SELECT` statement to be executed within any SQL statement.

We can use subqueries in virtually any logical block of the query. The only limitation is the type of the returned set, which must match the place where we want to use it. For example, in the `FROM` clause, it can be any set (single-element, multi-element, etc.), while in the `SELECT` clause it must be a scalar value, i.e. a single-element set described by one attribute.

This chapter will discuss mechanisms that allow you to perform analyses in a query based on the results of the queries nested in them.

6.1. Subquery Categories

Subqueries, which are a child part of another query, can be divided into two categories based on their relationship to the parent query:

- **independent** – the subquery is executed first, once, and its results are passed to the outer query;
- **correlated** – the parent query is executed first, and only then the subquery correlated with it, which is executed for each row viewed by the parent query. In a correlated query, it is necessary to use aliases of the relations on which the parent query operates and refer to them in the subquery.

6.1.1. Independent Subqueries

We will use the fact that the subquery returns a data set, which we will treat as a fully specified named table appearing in the FROM clause. Hence the need to use aliases and unique column names within subqueries.

As an example, we will display using an independent subquery those names and surnames of clients who live in 'Warszawa' with a restriction to surnames starting with one of the letters from 'A' to 'K'.

```
SELECT *
FROM (
  -- preliminary data selection
  -- there can be any, even dangerous, query here
  SELECT name, surname
  FROM t_client
  WHERE city = 'Warszawa'
) AS my_question_1
WHERE surname ~ '[A-K].*';
```

In the next example, we complicate the subquery by using an aggregate function to calculate the average over two selected years, and then use the previous data to display only those years for which the average meets the specified condition.

```
SELECT year
FROM (
  SELECT year, AVG(price) AS Average
  FROM t_book
  WHERE year in ('1997','2001')
  GROUP BY year
) AS my_question_2
WHERE Average > 25;
```

Note that at any time we can run this subquery independently of the parent query. It will be executed once, during the entire logical processing of this query.

We determine the average value for all orders.

```
SELECT ROUND(AVG(t_book.volume*t_book.price),2) as AVG
FROM t_book INNER JOIN t_z_book USING (ISBN);
```

The returned data set can be placed anywhere in the query – as a subquery. Most often, we will use it in the conditions of the WHERE clause or in the SELECT clause. It

can also be used in other places where we create expressions, e.g. to filter groups in the `HAVING` clause or in the join conditions in `ON`.

We will now use the previously obtained dataset to filter out records in the `WHERE` clause and additionally display them in the `SELECT` clause as an additional column value.

```
SELECT order_id, date_order,
(
  SELECT ROUND(AVG(t_book.volume*t_book.price),2) as AVG
  FROM t_book INNER JOIN t_z_book USING (ISBN)
) AS AVG_Total
FROM t_order
WHERE card != 1
AND date_order between '1999-06-01' and '2024-05-30'
AND 25 < (
  SELECT ROUND(AVG(t_book.volume*t_book.price),2) as AVG
  FROM t_book INNER JOIN t_z_book USING (ISBN)
);
```

When a subquery selects a single row (usually limited to a single column in the `SELECT` clause of the subquery), we most often use one of the comparison operators in the selection condition of the parent query, e.g. $=, >=, <$. When a subquery selects more than one row, we most often use the `IN` operator in the selection condition of the parent query.

Additionally, the following operators have been introduced for subqueries:

- **ALL** – compares a single value to each value specified by the subquery. The selection condition of the parent query is met if all values in the list meet this condition.
- **ANY** – compares a single value (located to its left) to each value specified by the subquery. The selection condition of the parent query is met if the list of values specified by the subquery contains at least one element that satisfies it.
- **EXISTS** – checks if the subquery returns a non-empty result and returns `True` if it does, otherwise returns `False`.
- **NOT EXISTS** – checks if the result of the subquery is empty and returns `True` if it is, otherwise returns `False`.

It is crucial that the number of values determined by the subquery and their type match the number and type of columns used in the selection condition of the parent query.

The first example displays all information about the books that have the lowest price.

```
SELECT *
FROM t_book
WHERE price = (
```

```
  SELECT MIN(price)
  FROM t_book
);
```

The next example shows the titles and publisher IDs of books provided by a supplier called 'UPS'.

```
SELECT title, publisher
FROM t_book
WHERE supplier = (
  SELECT supplier_id
  FROM t_supplier
  WHERE name = 'UPS'
);
```

Unlike the previous examples where subqueries returned a single row, we will now show the use of the `IN` operator when the query returns more than one record, and the use of more than one column name (tuple, pair) to compare a pair of returned values in a single subquery record. We display all the information about books whose price is the minimum price in each supplier's group.

```
SELECT *
FROM t_book
WHERE (price, supplier) IN (
  SELECT MIN(price), supplier
  FROM t_book
  GROUP BY supplier
);
```

The above query can be equivalently written using the `ANY` operator.

```
SELECT *
FROM t_book
WHERE (price, supplier) = ANY (
  SELECT MIN(price), supplier
  FROM t_book
  GROUP BY supplier
);
```

We will now display some selected information about books whose price is greater than that of any book delivered by 'UPS'.

```
SELECT ISBN, title, publisher, supplier
FROM t_book
WHERE price > ALL (
  SELECT price
  FROM t_book inner join t_supplier on supplier = supplier_id
  WHERE name = 'UPS'
);
```

Subqueries can also be nested in the HAVING clause in order to either reject or accept specific groups of tuples depending on the subquery result.

We will display information about the total value of orders for individual clients, and we will reject those records for which the total value of the client's orders is less than or equal to the average value of all orders.

```
SELECT client_id, SUM(tk.price*tzk.volume) AS "Total_price"
FROM t_order INNER JOIN t_z_book tzk USING (order_id)
     INNER JOIN t_book tk USING (ISBN)
GROUP BY client_id
HAVING SUM(tk.price*tzk.volume) > (
  SELECT ROUND(AVG(t_book.volume*t_book.price),2)
  FROM t_book INNER JOIN t_z_book USING (ISBN)
);
```

6.1.2. Correlated Subqueries

The second type of subquery is a correlated subquery, which is directly related to the parent query. The connector is one or more columns passed from the parent query. From a performance perspective, this design usually (but not always) entails executing the subquery multiple times. Such operations can be quite expensive, although this may not necessarily be always the case.

The subquery determines the average price of books from the same publisher as the book analyzed by the parent query.

```
SELECT title, price, publisher
FROM t_book tk
WHERE price > (
  SELECT AVG(price)
  FROM t_book
  WHERE publisher = tk.publisher
);
```

This query determines the book's ISBN and the author's identifier, provided that the bookstore has only one title written by him.

Illustration of how to use the `NOT EXISTS` operator.

```
SELECT ISBN, author_id
FROM t_a_book tak
WHERE NOT EXISTS (
  SELECT ISBN
  FROM t_a_book
  WHERE author_id = tak.author_id and ISBN != tak.ISBN
)
ORDER BY 2;
```

We check the correctness of the obtained results by displaying only the identifiers of those authors for whom the number of book numbers (ISBN) equals 1.

```
SELECT author_id, COUNT(ISBN)
FROM t_a_book
GROUP BY author_id
HAVING COUNT(ISBN)=1
ORDER BY 1;
```

In many cases, either standalone or systemically (by the query optimizer), correlated subqueries can be reduced to simple table joins. For example, consider a query that displays unique titles of books published after '1995' and supplied by a supplier called 'Stolica', whose titles start with one of the listed letters.

```
SELECT distinct title
FROM t_book tk
WHERE tk.year > '1995'
    AND tk.supplier IN (
      SELECT td.supplier_id
      FROM t_supplier td
      WHERE td.name = 'Stolica'
      AND lower(substring(tk.title,1,1)) in ('a','b','d')
);
```

Such a query can easily be rewritten using table joins.

```
SELECT distinct title
FROM t_book, t_supplier
WHERE t_book.year > '1995'
```

```
  AND t_book.supplier = t_supplier.supplier_id
  AND t_supplier.name = 'Stolica'
  AND lower(substring(t_book.title,1,1)) in ('a','b','d');
```

There is another nested query transformation that uses defining a complement for the data set returned by the correlated subquery to rewrite the query using a table join, and finally using the collective operator EXCEPT, which returns the result of subtracting that complement from the entire data set.

Consider then a correlated query that displays the title of the most expensive paperback book and its vendor ID.

```
SELECT distinct x.title, x.supplier
FROM t_book x
WHERE   x.binding = 'paperback'
  AND x.price >= ALL (
    SELECT y.price
    FROM t_book y
    WHERE x.supplier = y.supplier
    AND y.binding ='paperback'
);
```

According to the order of operations in the transformation, we calculate the complement.

```
SELECT distinct x.title, x.supplier
FROM t_book x
WHERE   x.binding = 'paperback'
  AND x.price < ANY (
    SELECT y.price
    FROM t_book y
    WHERE x.supplier = y.supplier
    AND y.binding = 'paperback'
);
```

And then we rewrite the query using table joins.

```
SELECT distinct x.title, x.supplier
FROM t_book x, t_book y
WHERE   x.binding = 'paperback'
  AND x.supplier = y.supplier
  AND y.binding = 'paperback'
  AND x.price < y.price;
```

And finally, we subtract from the set of all blue products those for which the products are blue and have a higher price.

```
(
  SELECT x.title, x.supplier
  FROM t_book x
  WHERE x.binding = 'paperback'
)
EXCEPT --(MINUS)
(
  SELECT x.title, x.supplier
  FROM t_book x, t_book y
  WHERE   x.binding = 'paperback'
    AND x.supplier = y.supplier
    AND y.binding = 'paperback'
    AND x.price < y.price
);
```

From a performance standpoint, constructing a correlated query usually (but not always) involves the need to execute the subquery multiple times. There is no hard and fast rule that says every correlated query has to be expensive. In competent hands, correlated queries can be very useful and, contrary to common stereotypes, effective.

6.2. Recursive nesting

Independent subqueries (as opposed to correlated queries), regardless of nesting depth, are always executed in order from the most nested to the outermost. We delimit each subquery with parentheses and nest it to the right of the parent query condition (the query placed one level higher in the nest structure). Additionally, with multi-level nesting:

- the number and type of columns appearing in the `SELECT` clause of the subquery must match the number and type of columns used in the parent query condition (higher nested queries);
- we do not use the `ORDER BY` clause in subqueries;
- the `ORDER BY` clause can only appear as the last clause of the outermost query;
- collective operators can be used in a subquery; in external query conditions, apart from the `ANY`, `ALL EXISTS` and `NOT EXISTS` operators, any SQL operators can be used.

As an example, consider a query that displays all information about books whose prices are greater than the highest price of the book supplied by `'UPS'`.

```
SELECT *
FROM t_book
WHERE price > (
  SELECT MAX(price)
  FROM t_book
  WHERE supplier = (
    SELECT supplier_id
    FROM t_supplier
    WHERE name = 'UPS'
  )
);
```

The first supplier ID named 'UPS' is assigned. It is then used to determine the highest price for the books it supplies, and in turn display all information about books whose price is higher than the maximum amount set.

6.3. WITH Expression

Common table expression (CTE) named WITH expression is another type of subquery. Such expressions allow you to create temporary tables using the WITH clause. They simplify and improve the readability of the SQL code, and their use does not affect query performance. The CTE structure allows for recursion, but we will not discuss this issue in this manual. (We recommend the PostgreSQL documentation at the link https://www.postgresql.org/docs/15/queries-with.html.) Common array expressions can be used virtually anywhere - in views, functions, stored procedures, scripts, etc.

The CTE definition opens with the keyword WITH with the name of the set that will be created using it. We will refer to this named, temporary table in the query just as we would a regular table. In the example, we will define a common array expression called my_stat, which determines the first 5 records among all records containing information about the client's ID, the number of all his orders and the date of his last order. This expression gives each column a unique name. Columns can also be named in the body of the CTE expression as aliases and are not required to be named explicitly in the WITH clause. Once we define CTE, we can refer to it multiple times in the query it is associated with. The scope of visibility is very narrow and covers only the query that follows it, but in its full scope. Exactly as if we were querying a set, table or view, in this case specified by the CTE. In our example, the set of data determined by the expression CTE will be used to display, instead of the client's ID, which will be used to join tables, his name and surname along with information about the number of his total orders and the date of his last order.

```
-- definition of an array expression called my_stat
WITH my_stat(client_id, Total_number, Max_date)
AS (
  -- only CTE content can be tested,
  -- to check what it returns and what we will use later
  SELECT client_id, COUNT(*), MAX(date_order)
  FROM t_order
  GROUP BY client_id
  ORDER BY 2
  LIMIT 5
)
-- immediately after the definition,
-- query referring to, among others, this CTE
SELECT tk.name, tk.surname, ms.Total_number, ms.Max_date
FROM my_stat AS ms left join t_client AS tk
    ON ms.client_id = tk.client_id;
```

Using the WITH clause, the query presented on the page 76 (displaying the names and surnames of clients who live in 'Warszawa', limited to surnames starting with one of the letters 'A' to 'K') can be written in the following, more readable way:

```
-- definition of an array expression called my_question
WITH my_question AS (
  SELECT name, surname
  FROM t_client
  WHERE city = 'Warszawa'
)
SELECT *
FROM my_question
WHERE surname ~ '[A-K].*';
```

We can define several array expressions and, importantly, they are visible not only in the query associated with WITH, but also in the subsequently defined CTE (if there are more than 1 of them).

```
WITH
CTE_1 AS (
  SELECT client_id, COUNT(*) num_Order, MAX(date_order) Max_date
  FROM t_order
  GROUP BY client_id
),
```

```
CTE_2 AS (
  SELECT client_id, num_Order, Max_date
  FROM CTE_1
  WHERE num_Order < 4
)
SELECT CTE_1.*, CTE_2.*
FROM CTE_1 LEFT JOIN CTE_2 USING (client_id);
```

And the last example of using the `WITH` clause, additionally using the window function `RANK()` with the `PARTITION BY` clause (see PostgreSQL's documentation at the link `https://www.postgresql.org/docs/15/tutorial-window.html`) that assigns each record within a group a position number, which it occupies in this group according to a certain designated order.

```
WITH CTE_3 AS (
  SELECT
 client_id,
 order_id,
    date_order,
 RANK () OVER (
  PARTITION BY client_id
  ORDER BY date_order DESC
 ) date_order_rank
    FROM t_order
)
SELECT name, surname, CTE_3.*
FROM t_client INNER JOIN CTE_3 USING (client_id);
```

6.4. Subqueries and Aggregate Functions in Practice

We will now present some examples of using aggregate functions and subqueries in statements using existing data `CREATE TABLE AS` and `INSERT INTO ... QUERY`. We will create a table from the existing data called `New_table` containing data from the columns `client_id`, `name` and `surname` those clients whose surname contains the letter 'o' anywhere (ignore case).

```
CREATE TABLE New_table AS
SELECT client_id, name, surname
FROM t_client
WHERE surname ILIKE '%o%';
```

We add a column amount of the type 6-digit real number with 2 decimal places to the New_table table.

```
ALTER TABLE New_table ADD amount numeric(6,2);
```

If they do not exist, we add to the New_table table those clients who placed an order after '2008-01-01' and whose name is not 'Adam'.

```
INSERT INTO New_table
SELECT DISTINCT client_id, name, surname
FROM t_client JOIN t_order USING(client_id)
WHERE client_id NOT IN (SELECT client_id FROM New_table)
      AND name != 'Adam'
      AND date_order > '2008-01-01';
```

For each client with ID client_id in the New_table table, we will update the data in the amount column to the total cost of all his orders fulfilled.

```
UPDATE New_table AS N
SET amount  = (
    SELECT A.amount
    FROM (
        SELECT client_id, sum(tzk.volume*tk.price) amount
        FROM t_order JOIN t_z_book tzk USING(order_id)
             JOIN t_book tk USING (ISBN)
        WHERE completed = 1 AND client_id = N.client_id
        GROUP BY client_id ) AS A
);
```

We will then update the amount for those clients in the New_table table for which it is NULL to the average amount of all client orders in the New_table table whose amount is not NULL.

```
UPDATE New_table
SET amount = ( SELECT ROUND(AVG(amount),2)
              FROM New_table
              WHERE amount IS NOT NULL )
WHERE amount IS NULL;
```

6.5. Practice Exercises

Each of the following queries should use only subqueries:

1. Display titles of books whose author's name is `'Ryszard'`.
2. Display the names of publishers whose books are not available in the bookstore.
3. Display the ISBN, title, name and surname of the author of those books that were not provided by either `'Konik'` or `'UPS'`. Arrange the results in descending alphabetical order of the authors' surnames.
4. Display repeated author surnames, use correlated query.
5. Among all the books, display those whose price is the highest and those whose price is the second highest.
6. Calculate and enter the maximum price of books published in any year in which books ordered by a client with name `'Jan'` and surname `'Kowalski'` were published.
7. Display those authors whose surnames start with the first letter of name of the client with ID 1 or whose books cost more than the price of any book written by `'Cyprian Norwid'` and were ordered at `'2004-01-10'`.
8. Display the combined values of the name and surname columns of these authors, whose surname has the same letter anywhere as the first letter of the author's surname with identifier 16.
9. Display `client_id`, `name` and `surname` of those clients who ordered books published in the same year as the release year of the book with the title `'Tosca'`.
10. Display the `author_id`, `name` and `surname` of authors who published books in a year in which more than one book was published.
11. Display how many books there are that have never been sold (the order has not been completed yet or no one has ever ordered them).
12. Display the oldest order date of books ordered by a client with the surname `'Kowalski'`.
13. Display the ISBN, title, name and surname of the author of these books were published by `'PIW'`, `'Znak'` or `'Helion'`.
14. View IDs of completed orders with the highest value.
15. Display the smallest ID of the completed order with the highest value.
16. Display `client_id`, `name` and `surname` of those clients from the `'podlaskie'` Voivodeship (`state`) who ordered more than one book.
17. Display clients who have purchased books from publishers that have published more than two books.
18. Display the ISBN, title of the books, and the surnames of clients who purchased books by authors with a name starting with the letter `'K'` or `'W'`.
19. Display author information for books that have never been sold.

20. Search for the name of book suppliers whose publisher published more than two books in any year.
21. Search for how many books were sold (we only take into account completed orders) in particular years, as long as this number is between 2 and 4.
22. Increase the amount of those clients from the `New_table` table who did not exceed PLN 178 by the average of the total cost of not yet completed client orders from the `New_table` table.
23. Create a table `BookDescription` containing data about books from the following columns:
    - `ISBN (t_book)`,
    - `title (t_book)`,
    - `surname (t_author)`,
    - `name (t_publisher) as publisher`,
    - `name (t_supplier) as supplier`,

    which describe books from before the year 2000.
24. Add a `price` column of type `numeric(6,2)` to the `BookDescription` table.
25. Modify, for each record in the `BookDescription` table, the data in the `price` column, updating it to the actual value in the `t_book` table.

Chapter 7

Creating and Using Views

We create views to reuse SQL instructions we have written, especially those that are very long and complex, without the need to rewrite them. They are considered saved queries that allow for the creation of database objects that function similarly to tables but contain content that changes dynamically and directly pertains only to rows selected according to the view's definition. Views, ranging from simple queries on a single table to complex queries involving multiple tables, offer great flexibility. This chapter will discuss the creation and practical use of views.

7.1. Basic Operations Related to Views

The `CREATE VIEW` command allows you to create views, i.e logical windows containing a view of one or more tables. Views are objects similar to forms in that they do not they store data, and to tables as it is possible to use these the same SQL commands that can be applied to tables. Certain SQL statements must be met to execute specific requirements. This applies in particular `SELECT` command used to create the view and may contain a `GROUP BY` clause and `DISTINCT`.

Views are used to increase the security level of tables by limiting access only to selected table columns basic and allow you to hide complexity from the user data. It is safe to select data using views, but not them modifying. Data that is based on other views, multiple tables, or those in which the clauses `GROUP BY`, `DISTINCT`, `LIMIT`, `UNION`, `HAVING` and aggregate functions (`SUM`(), `MIN`(), `MAX`(), `COUNT`()) is used cannot be modified. If the view was created in this way, then you need to modify the data in the tables original.

The syntax for the create view command is:

```
CREATE [OR REPLACE]
[TEMP | TEMPORARY] [RECURSIVE] VIEW name [(column_name [,...])]
[WITH (view_option_name [= view_option_value] [,...])]
AS query
[WITH [CASCADED | LOCAL] CHECK OPTION];
```

- The `OR REPLACE` clause in the `CREATE VIEW` command syntax is optional and its addition causes if a view with this name exists, it will remain overwritten. Another method to change the query of a created view is `ALTER VIEW` statement.
- We can create a view called `name` by giving its columns the names listed in a list enclosed in round brackets, which can optionally be temporary or recursive.
- `[WITH (view_option_name [= view_option_value] [,...])]` specifies optional view parameters, such as: `check_option (enum)`, `security_barrier (boolean)` or `security_invoker (boolean)`.
- The `query` in the view you create determines the structure and data set. It can have almost any syntax and, for example, may contain joins on tables, intersections and subqueries, as well as refer to other views. The query cannot:
  - include subqueries in the `FROM` clause,
  - use system variables and those defined by user,
  - tables used in the query must exist at the time the view is created, temporary tables cannot be referenced.

  However, it is allowed to use the `ORDER BY` clause it is ignored if in the query to view this clause also appears.
- Clause `WITH [CASCADED | LOCAL] CHECK OPTION` - generally checks when using the `INSERT` and `UPDATE` statements whether the condition of the `WHERE` clause is met.

An example of creating a view in which, by default, the column names are the same as those that were part of the query.

```
CREATE VIEW V1 AS
SELECT isbn, title, volume
FROM t_book
WHERE volume > 3
WITH CHECK OPTION;
```

An example of creating a view in which the source column names have been changed.

```
CREATE VIEW V2 (kol1, kol2) AS
SELECT binding, count(*) as Count
FROM t_book
GROUP BY binding;
```

When renaming source columns, remember that the number of columns defined by the view must match the number of columns returned by the `SELECT` query.

To see the names of the created views, execute the command in the `psql` client: `\dv` or `\d+ view_name`.

`ALTER VIEW` provides the ability to modify the view definition and has the following syntax:

```
ALTER VIEW [IF EXISTS] name
ALTER [COLUMN] column_name SET DEFAULT expression

ALTER VIEW [IF EXISTS] name
ALTER [COLUMN] column_name DROP DEFAULT

ALTER VIEW [IF EXISTS] name
OWNER TO {new_owner | CURRENT_ROLE | CURRENT_USER | SESSION_USER}

ALTER VIEW [IF EXISTS] name
RENAME [COLUMN] column_name TO new_column_name

ALTER VIEW [IF EXISTS] name
RENAME TO new_name

ALTER VIEW [IF EXISTS] name
SET SCHEMA new_schema

ALTER VIEW [IF EXISTS] name
SET (view_option_name [= view_option_value] [,... ])

ALTER VIEW [IF EXISTS] name
RESET (view_option_name [,... ])
```

We refer to the view in the same way as if it were an ordinary table.

```
SELECT column_name [,...]
FROM view_name;
```

For example, for the V1 and V2 views defined earlier, their content will be displayed as follows:

```
SELECT title FROM V1;
SELECT * FROM V2;
```

To create a view, the user must have CREATE VIEW permissions (also for other instructions operating on the view) in the database and SELECT for the selected table. It is not possible to create a trigger (trigger) on a view. Also note that modifying the dataset (original data) affects the content of the data in the view.

A view deleting executes the command:

```
DROP VIEW [IF EXISTS] nazwa_widoku;
```

The IF EXISTS clause is optional and is intended to do so checking whether the view we want to delete it definitely exists. If the instruction does not exist, it will be ignored and we will be informed about it with an appropriate message.

For example:

```
DROP VIEW IF EXISTS V1;
```

removes the view named V1 if it was previously created and still exists in the database.

7.2. Views in Action

There are two types of view:

- a straight line that provides data from a single table, and its definition does not use:
  - set operations,
  - function,
  - or grouping of lines;
- composite that provides data from multiple tables as well as set operations, table joins, functions, and row grouping. Then only SELECT can be directed to the view.

Let's define the view based on a query that will determine the number of books available in the bookstore for each publisher by grouping books by publisher's name and the lowest price in each group.

```
CREATE VIEW book_publisher(name_publisher, Number, min) AS
SELECT name, count(1), min(price)
FROM t_book INNER JOIN t_publisher ON (publisher = publisher_id)
GROUP BY name
ORDER BY 1 DESC;
```

Let's display the designated data by referring to the view.

```
SELECT * FROM book_publisher;


SELECT *
FROM book_publisher
ORDER BY min DESC LIMIT 3;
```

The second example further shows that view results can be sorted differently than done in the definition of view.

An updateable view allows you to update the base tables that make up it. As long as certain conditions are met, you can execute `UPDATE`, `DELETE`, and even `INSERT` in the view as in a regular table. A view cannot be updated if it contains grouping clauses, totals, an aggregate function, among others. A query that modifies data can include a join, but all the columns being changed must be in one table.

```
CREATE VIEW V3 AS
SELECT publisher_id, name
FROM t_publisher
WHERE name ilike '%a%';


UPDATE v3
SET name = upper(name)
WHERE publisher_id = 3;


SELECT name FROM v3 WHERE publisher_id = 3;


SELECT name FROM t_publisher WHERE publisher_id = 3;
```

```
INSERT INTO v3
VALUES (1000, 'Kongo');
```

After inserting a new record through the view, an attempt to refer also through the view to the record with publisher ID equal to 1000 ends in failure and an empty result set. Name publisher does not meet the condition in the `WHERE` clause of the view definition.

```
SELECT name FROM v3 WHERE publisher_id = 1000;
 name
-------
(0 rows)
```

Meanwhile, the reference to the source table shows the inserted data.

```
SELECT name FROM t_publisher WHERE publisher_id = 1000;
 name
-------
 Kongo
(1 row)
```

When inserting, it is important to ensure that all `NOT NULL` columns are completed or a default value should be defined for these columns.

Now we delete the lines through view.

```
DELETE FROM v3
WHERE publisher_id = 3;
```

```
SELECT name FROM v3 WHERE publisher_id = 3;
 name
-------
(0 rows)
```

We will not find the deleted record in the source table either.

```
SELECT name FROM t_publisher WHERE publisher_id = 3;
 name
-------
(0 rows)
```

Examples of views based on table joins.

```
CREATE VIEW v4 AS
SELECT ISBN, title, name
FROM t_book INNER JOIN t_publisher ON publisher = publisher_id
WHERE isbn < 50;
```

```
SELECT * FROM v4 WHERE ISBN = 45;
 isbn |     title     | name
------+---------------+------
   45 | Niebo w ogniu | Znak
(1 row)
```

```
CREATE VIEW author_book AS
SELECT name, surname, title, year
FROM t_author INNER JOIN t_a_book USING (author_id)
     INNER JOIN t_book USING (ISBN)
WHERE year > '1997';
```

```
SELECT * FROM author_book;

CREATE VIEW book_statistic(year, price_min,price_max,price_avg) AS
SELECT year, min(price), max(price), round(avg(price),2)
FROM t_book
GROUP BY year;

SELECT *
FROM book_statistic
WHERE year LIKE '_9%'
ORDER BY 1;
```

7.3. Practice Exercises

For each of the following tasks, define views whose names consist of a word task and the number of the task being solved. View data from every view.

1. Display the book titles and author ID of those books that have been ordered in quantities of 2 or more.
2. We assume that the author (or the author's descendant) receives 2% from the copy sold, display the amount of money earned from sales (we only take into account completed orders).
3. Display information about books whose price is no greater than the price of books supplied by a supplier located in a city containing the letter `'s'` anywhere.
4. Display the titles of books whose number of copies sold exceeds the number of copies owned (use a correlated query).
5. Display the book titles and author ID of those books that have been sold in quantities of less than 2 (use a correlated subquery).
6. Display the IDs of publishers who have published books by at least 3 different authors.
7. Nominate authors who published books in a year when no one else published anything. (Use the `NOT EXISTS` operator).

Readers who want to improve their skills in working with views are encouraged to define views based on the self-solving tasks proposed in other chapters.

Chapter 8

Creating and Using Indexes

We increase the efficiency of database queries by using indexes. A database index is a separate file that contains a pre-prepared and ordered set (or subset) of links to data that comply with specified conditions. When an index containing the data of interest is available when executing a query, the planner, using various mechanisms operating on the server to analyze the request and select the best way to execute it in terms of time and memory efficiency, can use pre-prepared and ordered data from this index. If the index is unavailable, the database must repeatedly scan all records to select those that contain the data we are interested in. Even if all the information is at the beginning of the searched set data, then all of them must be scanned records.

The usefulness of a given index is determined by its key. An index key is the column or group of columns to which the index is applied. An index can only speed up a query if its key contains one or more of the columns used in the query predicate, i.e. in its `WHERE` clause (the order of the columns is also important). It should be emphasized that in relational databases, an index on its primary key is always defined in the table being created. The remaining indexes are defined by the user and should be done carefully, realizing that each index increases database resources, and at the same time, if we incorrectly define the selection criteria in the query, it may never be used.

To increase search efficiency, PostgreSQL uses many different indexing mechanisms, such as B-tree, hash indexes, GIN (*Generalized Inverted Index*) indexes, and GiST (*Generalized Search Tree*) indexes. Since the most frequently used type of index is the B-tree index, in this chapter we will present the creation and use of this type of indexes on the example of a simple table but with a large number of rows. We will explain why they are needed and what advantages and disadvantages they have.

8.1. B-tree Index

B-tree indexes are a specific type of database index that organizes specific ranges of key values in the form of a tree structure - a balanced tree. Without delving into the theory of the B-tree structure and tree traversal algorithms, because they are managed by the database itself, we will try to simply explain the principle of operation of this type of indexes.

The B-tree index orders rows according to key values and divides the ordered list into ranges. These scopes are organized in a tree structure, where the root node defines a broad set of scopes and each level below narrows the scopes. Consider the example of a bookstore where books are arranged by author's name. Let's assume that we are looking for a book by Sienkiewicz (here the name does not matter, because the order is determined by the authors' surnames). There are rows of shelves in a bookstore, with each row marked with the initial letters of the authors' surnames, for example `A-F`, then `G-O`, then `P-Z`. We go to a specific shelf with the key value range `P-Z` and see that each shelf has a plate with a smaller range of author names, e.g. `PA-PZ`, then `SA-SZ` and so on. So we are dealing here with subranges of larger ranges. In turn, the books on each shelf are arranged alphabetically according to: authors' names, thanks to which we can quickly find Sienkiewicz's book.

The organization of the B-tree index reflects the bookstore example described above. Of course, in the case of a database index, we are dealing with a larger number of ranges, but the basic idea remains the same. Each node in the B-tree is called a block. The blocks at each level of the tree except the last are called branch blocks, and those at the last level are called leaf blocks. Entries in branch blocks consist of a scope identifier and a pointer to a block at the next level of the tree. Each entry in a leaf block represents a row and consists of a key value (e.g. "Sienkiewicz") and a row ID, which the database uses to obtain all other data contained in that row.

To create an index of a specific type (e.g. `BTREE, HASH, GIST`, etc.) for a data set, we use the `CREATE INDEX` statement along with listing the columns on which the index is applied within the given table. In addition, we may impose additional conditions and restrictions to make the index more selective.

```
CREATE INDEX index_name
ON table_name [USING index_type](col[,...])
[WHERE condition];
```

When running `CREATE INDEX`, a B-Tree index is created by default and works on all data types and can be used to retrieve NULL values.

8.2. Stored Functions

In this section, we will briefly discuss creating and using stored functions to query and reuse them, including creating and populating a sample table with data to demonstrate how indexes work.

Functions in SQL are isolated blocks of code that are stored in the database. They enable efficient code reuse (storing already compiled functions on the server, they only need to be run) to repeat and modify statements and queries. However, the most important advantage of functions is that they allow you to divide your code into smaller, testable portions.

To create functions stored in SQL, use the following command:

```
CREATE [OR REPLACE] FUNCTION function_name ([arg[,...]])
RETURNS returning_value_type AS $$
[DECLARE var_name var_type;]
BEGIN
  body_function;
  [RETURN value;]
END; $$
LANGUAGE plpgsql;
```

Individual function elements:

- `function_name` - name assigned to the function and used to run it later;
- `[arg[,...]]` - optional list of function arguments. It may be empty or contain a list of values of various data types (literals) or a list of named arguments;
- `returning_value_type` - data type returned by the function;
- optional statement `DECLARE` is a block of declarations of local function variables;
- `body_function` - function body;
- optional value returned by the function;
- `plpgsql` specifies the language used in the function, for this manual we only use this language.

For example, we will define the `count_order` function, which takes no arguments and returns an integer that determines the number of orders placed and completed.

```
CREATE FUNCTION count_order()
RETURNS integer AS $$
DECLARE
  total integer;
BEGIN
  SELECT COUNT(order_id) INTO total
```

```
  FROM t_order
  WHERE completed = 1;
  RETURN total;
END;
$$ LANGUAGE plpgsql;
```

In the function body, we used the `SELECT..INTO...` statement, which redirects the query result to the variable declared in the `DECLARE` block and returns it. We can use the defined function wherever necessary, such as any built-in, standard PostgreSQL function.

A call to the `count_order` stored function in its simplest form looks like this:

```
SELECT count_order();
```

We will now define the `max_order` function, which will enable us to calculate and return the highest order value among those already completed.

```
CREATE FUNCTION max_order()
RETURNS numeric AS $$
DECLARE
  maximum numeric;
BEGIN
  WITH A AS (
     SELECT order_id, sum(tk.price * tzk.volume) as total
     FROM t_book tk JOIN t_z_book tzk USING (ISBN)
          JOIN t_order USING (order_id)
     WHERE completed = 1
     GROUP by tzk.order_id
  )
  SELECT max(A.total) INTO maximum
  FROM A;
  RETURN maximum;
END;
$$ LANGUAGE plpgsql;
```

We have defined a function by running (`SELECT max_order()`) that we can obtain information about the current, highest amount of the completed order.

We will now create a function accepting arguments that allows us to calculate the sum of the prices of books found in accordance with the given parameters (arguments).

```
CREATE FUNCTION search_sum_book(y char(4), t char(1))
RETURNS numeric AS $$
DECLARE
```

```
  result numeric;
BEGIN
  SELECT sum(price) INTO result
  FROM t_book
  WHERE year = y AND title ilike '%'||t||'%';
  RETURN result;
END;
$$ LANGUAGE plpgsql;
```

And then we will call it to calculate the sum of the prices of books from 1997 in which the letter 't' appears anywhere in the title (case-insensitive).

```
select search_sum_book('1997', 't');
```

In the `psql` client, using the `\df` command we can retrieve a list of stored functions as well as variables and data types of arguments and the type of return value.

```
postgres=# \df
                              List of functions
 Schema |      Name       | Result    |     Argument       | Type
        |                 | data type |    data types      |
--------+-----------------+-----------+------------------+------
 public | count_order     | integer   |                  | func
 public | max_order       | numeric   |                  | func
 public | search_sum_book | numeric   | year character,  | func
        |                 |           | t character      |
(4 rows)
```

In turn, with the command `\sf function_name` we can display the definition of an already existing function, e.g. `postgres=# \sf max_order`.

Detailed information about functions and stored procedures can be found in the PostgreSQl documentation.

8.3. Sample Session

To be able to read the execution time of `SQL` commands, you can turn on or off the display of the duration of each SQL statement with the command `\timing [ON/OFF]` – default `OFF` or read `Execution Time` from information displayed by the `EXPLAIN` or `EXPLAIN ANALYZE` commands.

Running `EXPLAIN` or `EXPLAIN ANALYZE` will not execute the query and will not return a value. Instead, it will return a description along with the costs of processing each stage of the plan, and `ANALYZE` will also add the time needed to plan and execute the query.

```
EXPLAIN SELECT * FROM t_book;
                        QUERY PLAN
---------------------------------------------------------------
 Seq Scan on t_book  (cost=0.00..14.20 rows=420 width=166)
(1 row)
```

The query execution plan contains information about the type of scan used in the query. `Seq Scan` - sequential scan, a linear way of searching the entire data set, each record in the table is checked and compared with the sequential scan criteria to select those that meet the criterion specified in the `WHERE` clause. Most often used when the table is small, or the field used in the search contains many repeated values, or the scheduler determines that sequential scanning with the given criteria is equal to (or more) efficient than other scanning methods.

Additionally, we receive information about the costs of preparatory operations (`cost=0.00`), e.g. sorting data. This measure is given in cost units, not seconds, and is often the number of requests to disk or page downloads. We are then given a value (`14.20`) representing the total cost of executing the query if all rows are retrieved (this is not always the case). The next value (`rows=420`) in the plan specifies the total number of rows available to be returned if the plan is executed in its entirety. And the last one (`width=166`) is the length of each line in bytes.

In the example in this chapter, we will use the `EXPLAIN ANALYZE` command to compare query execution times.

We create a test table and insert `1,000,000` sample records into it, for this purpose we will create a file **indexes.sql** containing the definition of the argument-free function `indexes`, in which three nested iterative statements `FOR` generate values inserted into the test table.

```
CREATE FUNCTION indexes()
RETURNS int AS $$
DECLARE
   i int;
   j int;
   k int;
BEGIN
   DROP TABLE IF EXISTS indexes;
   CREATE TABLE indexes (
    id SERIAL PRIMARY KEY,
    x INT,
    y INT,
    z INT
```

```
   );
   FOR i IN 1..100 LOOP  -- FOR loop
     FOR j in 1..100 LOOP
       FOR k in 1..100 LOOP
         INSERT INTO indexes (x,y,z) VALUES(i,j,k);
       END LOOP;
     END LOOP;
   END LOOP;
   RETURN 0;
END;
$$;
SELECT indexes(); -- wywolanie funkcji
```

We check the number of rows in the `indexes` table.

```
postgres=# SELECT COUNT(*) FROM indexes;
+----------+
| count(*) |
+----------+
|  1000000 |
+----------+
(1 row)
```

We check what indexes we currently have on the **indexes** table - there is only one, the index for the primary key.

```
postgres=# \d indexes

 Table "public.indexes"
 Column |  Type   | Colla-  |          |
        |         | tion    | Nullable |              Default
--------+---------+---------+----------+-----------------------------------
 id     | integer |         | not null | nextval('indexes_id_seq'::regclass)
 x      | integer |         |          |
 y      | integer |         |          |
 z      | integer |         |          |
Indexes:
    "indexes_pkey" PRIMARY KEY, btree (id)
```

We make a sample query.

```
postgres=# SELECT * FROM indexes WHERE x=1 AND y=10 AND z=100;
```

```
+------+------+------+------+
| id   | x    | y    | z    |
+------+------+------+------+
| 1000 |    1 |   10 |  100 |
+------+------+------+------+
(1 row)
```

The SQL query execution plan shows that there is no index was used (because there are no indexes yet that could help with faster execution queries).

```
postgres=# EXPLAIN ANALYZE SELECT *
postgres-# FROM indexes WHERE x=1 AND y=10 AND z=100;
```

```
QUERY PLAN
------------------------------------------------------------------------
 Gather  (cost=1000.00..13697.77 rows=1 width=16)
         (actual time=2.827..105.610 rows=1 loops=1)
   Workers Planned: 2
   Workers Launched: 2
   -> Parallel Seq Scan on indexes (cost=0.00..12697.67 rows=1 width=16)
                            (actual time=52.130..82.390 rows=0 loops=3)
         Filter: ((x = 1) AND (y = 10) AND (z = 100))
         Rows Removed by Filter: 333333
 Planning Time: 0.400 ms
 Execution Time: 105.688 ms
(8 rows)
```

The first noteworthy aspect of the above query is that it involves several different steps. The lower-level query performs a sequential scan based on the provided values. Here, sequential scanning is performed in parallel, `Parallel Seq Scan`, and is planned to use two worker threads. Whether the PostgreSQL server uses parallel scanning depends on both the server configuration and the computer's hardware capabilities. In this particular example, PostgreSQL has determined that parallel scanning may provide higher performance, so it allocates two threads to this operation. At a higher level in the plan, you see the `Gather` step, which is executed at the beginning of the query. The cost of preparation operations is non-zero (`1000`), and the total cost is `13697.77`.

We create three indexes, one for each non-key attribute (on columns x, y and z, respectively).

```
postgres=# CREATE INDEX idx_x ON indexes (x);
CREATE INDEX
postgres=# CREATE INDEX idx_y ON indexes (y);
```

```
CREATE INDEX
postgres=# CREATE INDEX idx_z ON indexes (z);
CREATE INDEX
```

```
postgres=# \d indexes;
```

```
 Table "public.indexes"
 Col |  Type   | Colla- |          |
 umn |         | tion   | Nullable |              Default
-----+---------+--------+----------+-----------------------------------
 id  | integer |        | not null | nextval('indexes'::regclass)
 x   | integer |        |          |
 y   | integer |        |          |
 z   | integer |        |          |
Indexes:
    "indexes" PRIMARY KEY, btree (id)
    "idx_x" btree (x)
    "idx_y" btree (y)
    "idx_z" btree (z)
```

We execute the same query again and check in the query execution plan (`EXPLAIN ANALYZE`) whether the system used previously defined indexes.

```
postgres=# SELECT * FROM indexes WHERE x=1 AND y=10 AND z=100;
  id  | x | y  |  z
------+---+----+-----
 1000 | 1 | 10 | 100
(1 row)
postgres=# EXPLAIN ANALYZE SELECT *
postgres-#FROM indexes WHERE x=1 AND y=10 AND z=100;
```

```
                             QUERY PLAN
----------------------------------------------------------------------
 Index Scan using idx_x on indexes (cost=0.42..409.68 rows=1 width=16)
                            (actual time=2.095..12.810 rows=1 loops=1)
   Index Cond: (x = 1)
   Filter: ((y = 10) AND (z = 100))
   Rows Removed by Filter: 9999
 Planning Time: 0.449 ms
 Execution Time: 12.893 ms
(6 rows)
```

We create a composite index, built on three columns.

```
postgres=# CREATE INDEX idx_xyz ON indexes (x, y, z);
CREATE INDEX
```

We redisplay information about the indexes defined on the `indexes` table.

```
postgres=# \d indexes;
                               Table "public.indexes"
 Col- |  Type   | Colla- | Nullable |               Default
 umn  |         | tion   |          |
------+---------+--------+----------+-------------------------------------
 id   | integer |        | not null | nextval('indexes'::regclass)
 x    | integer |        |          |
 y    | integer |        |          |
 z    | integer |        |          |
Indexes:
    "indexes" PRIMARY KEY, btree (id)
    "idx_x" btree (x)
    "idx_xyz" btree (x, y, z)
    "idx_y" btree (y)
    "idx_z" btree (z)
```

In the `EXPLAIN ANALYZE` command you can see that this time the system has used one triple index, instead of the three previously defined single indexes.

```
postgres=# EXPLAIN ANALYZE SELECT *
postgres-# FROM indexes WHERE x=1 AND y=10 AND z=100;
                                   QUERY PLAN
----------------------------------------------------------------------
Index Scan using idx_xyz on indexes (cost=0.42..8.45 rows=1 width=16)
                            (actual time=0.170..0.178 rows=1 loops=1)
   Index Cond: ((x = 1) AND (y = 10) AND (z = 100))
 Planning Time: 1.514 ms
 Execution Time: 0.273 ms
(4 rows)
```

Now we remove single indexes:

```
postgres=# DROP INDEX idx_x;
DROP INDEX
postgres=# DROP INDEX idx_y;
DROP INDEX
postgres=# DROP INDEX idx_z;
DROP INDEX
```

Only the composite index remains (on three columns: x, y, z) and this index will be used only if the WHERE clause has the appropriate form:

```
postgres=# EXPLAIN ANALYZE SELECT COUNT(*)
postgres-# FROM indexes WHERE x=1 AND y=10 AND z=100;

                                QUERY PLAN
---------------------------------------------------------------------
Aggregate (cost=8.45..8.46 rows=1 width=8)
          (actual time=0.232..0.235 rows=1 loops=1)
-> Index Only Scan using idx_xyz on indexes (cost=0.42..8.45 rows=1 width=0)
                           (actual time=0.195..0.202 rows=1 loops=1)
         Index Cond: ((x = 1) AND (y = 10) AND (z = 100))
         Heap Fetches: 1
 Planning Time: 0.880 ms
 Execution Time: 0.379 ms
(6 rows)

postgres=# EXPLAIN ANALYZE SELECT COUNT(*)
postgres-# FROM indexes WHERE z=100;

                                QUERY PLAN
-----------------------------------------------------------------------
 Finalize Aggregate  (cost=11625.28..11625.29 rows=1 width=8)
                     (actual time=102.133..114.030 rows=1 loops=1)
   ->  Gather  (cost=11625.06..11625.27 rows=2 width=8)
               (actual time=101.974..114.014 rows=3 loops=1)
         Workers Planned: 2
         Workers Launched: 2
         ->  Partial Aggregate (cost=10625.06..10625.07 rows=1 width=8)
                             (actual time=87.578..87.579 rows=1 loops=3)
               ->  Parallel Seq Scan on indexes
                   (cost=0.00..10614.33 rows=4292 width=0)
                   (actual time=0.084..86.934 rows=3333 loops=3)
                     Filter: (z = 100)
                     Rows Removed by Filter: 330000
 Planning Time: 0.454 ms
 Execution Time: 114.184 ms
(10 rows)
```

Comparison of command execution times:

```
+----+---------+---------------------------------------------------------+
|ID  | Duration| Query                                                   |
|    |    (ms) |                                                         |
+----+---------+---------------------------------------------------------+
| 1 |    9.869 | SELECT COUNT(*) FROM indexes WHERE x=1                  |
| 2 |  111.066 | SELECT COUNT(*) FROM indexes WHERE y=10                 |
| 3 |  116.288 | SELECT COUNT(*) FROM indexes WHERE z=100                |
| 4 |  107.074 | SELECT COUNT(*) FROM indexes WHERE z=100 AND y=10       |
| 5 |    0.281 | SELECT COUNT(*) FROM indexes WHERE x=1 AND y=10 AND z=100 |
| 6 |    0.266 | SELECT COUNT(*) FROM indexes WHERE z=100 AND y=10 AND x=1 |
+----+---------+---------------------------------------------------------+
```

We will test the use of the LIMIT clause in queries. We will delete the existing composite index and then ask about the records about which we know that one occurs at the beginning of the table and the other at the end. Of course, in practical applications we most often do not know the physical location of the record or records we are looking for, and therefore using the LIMIT clause may (usually) not give such good results.

```
postgres=# DROP INDEX idx_xyz;
DROP INDEX
postgres=# EXPLAIN ANALYZE SELECT *
postgres-# FROM indexes WHERE x=1 AND y=1 AND z=1;
postgres=# EXPLAIN ANALYZE SELECT *
postgres-# FROM indexes WHERE x=100 AND y=100 AND z=100;
postgres=# EXPLAIN ANALYZE SELECT *
postgres-# FROM indexes WHERE x=1 AND y=1 AND z=1 LIMIT 1;
postgres=# EXPLAIN ANALYZE SELECT *
postgres-# FROM indexes WHERE x=100 AND y=100 AND z=100 LIMIT 1;
```

8.4. Effective Use of Indexes

Indexes may seem like an obvious way to speed up queries, but they don't always have the desired effect. Let's consider some typical situations:

- The field for which the index is created is often modified. Inserting or deleting rows from the table may cause the created index to become inefficient because it was created on data that no longer exists or has changed its value. This requires reorganizing the indexes. In SQL, data indexes can be recreated using the `REINDEX` command, but it requires analysis of costs, techniques, and strategies.

- The index is out of date, the existing references are invalid, or there are segments of unindexed data against which the planner cannot use the index. PostgreSQL usually automatically updates indexes in response to table changes. Unfortunately, there may be situations where the automatic update does not run correctly. Then it is necessary to update it manually.
- Often, records are searched using the same search criteria for a specific field. Instead of applying the index to all the data in the specified column, you can then restrict it to create a partial index using a subset of the data that matches the selection criterion. In this way, we create a smaller, and therefore more efficient, index that is easier to maintain and can be used in more complex queries.
- The database is not large. Then the cost of creating, maintaining and using the index may exceed sequential scanning, which is quite fast for a small amount of data (especially if it concerns data already stored in RAM). Therefore, creating an index does not guarantee that it will be used in the query execution plan, it only brings load to the database.

This chapter presented plans for executing queries relating to only one table. As queries become more complex due to table joins, their execution plans also become more complex. Merging two tables complicates the query execution plan, which results from the need not only to retrieve more data from the hard disk, but also to match the data (based on the key used in the join) from one table to the data from the other. The ability to interpret and understand such execution plans is undoubtedly very important, but discussion of this topic is beyond the scope of this manual. The interested reader is referred to the PostgreSQL documentation for additional information.

8.5. Practice Exercises

1. From the `psql` PostgreSQL console, run the **indexes.sql** script again, using e.g.

   ```
   \i path\filename.
   ```

   The test procedure will consist in executing the same query several times, but for previously defined appropriate indexes (the order of creating indexes can be found in the 8.1 table):

   ```
   SELECT * FROM indexes WHERE x=1 AND y=10 AND z=100;
   ```

   and reading the response time (the response time in milliseconds, with an accuracy of 3 decimal places, should be written in the third column of table 8.1). In subsequent attempts, we only change the organization of the indexes established on the **indexes** table, without changing the query issued. For example, position number 5 in table 8.1 ("Simple indexes on columns **x**, **y**") means that only two indexes are created on

the test table at the time of executing the **SELECT** command simple, na columns **x** and **y** respectively. To move to position 6, remove the index on the **y** column and add an index on the **z** column, etc.

Table 8.1: Index creation order in the test procedure and SELECT command execution times

L.P.	Current index configuration	Command execution time SELECT
1	No indexes	
2	Simple index on column **x**	
3	Simple index on column **y**	
4	Simple index on column **z**	
5	Simple indexes on columns **x**, **y**	
6	Simple indexes on columns **x**, **z**	
7	Simple indexes on columns **y**, **z**	
8	Simple indexes on columns **x**, **y**, **z**	
9	Composite index on columns **x**, **y**	
10	Composite index on columns **x**, **z**	
11	Composite index on columns **y**, **z**	
12	Composite index on columns **x**, **y**, **z**	

2. Create a table:

```
DROP TABLE IF EXISTS indexes2;

CREATE TABLE indexes (
    id INT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(10),
    surname VARCHAR(15),
    gender CHAR(1),
    salary INT,
    date_employment DATE
);
```

Prepare a `sql` script (e.g. using the Python programming language), which will consist of instructions inserting 100,000 records into the `indexes2` table with values such that:

- column **name** was randomly filled with names from a list of 5 female names and 5 male names;

- column **surname** contained random phrases of length between 3 and 15; the first letter is uppercase;
- column **gender** was randomly filled with the letter M or K;
- column **salary** contained a random number between 1 and 5000;
- column **date_employment** contained a random date between the current date (the day the script was generated) and a date up to 5000 days earlier.

and conduct appropriate tests (analogously to task 1) showing the operation of various indices. What query will show the effectiveness of the defined indexes? Record in the table what index was created and what is the query execution time. Remember to perform tests for different indexes on the same query.

3. Define the `F1` function that will find and return how many books there are that have never been sold (the order has not been completed yet or no one has ever ordered them).
4. Define the `F2` function that will find and return how many books were ordered in the year given as a function parameter.
5. Define a function `F3` that will find and return the oldest date of ordering books by a client with the ID given as an argument to the function.
6. Define the function `F4`, which will calculate and return the sum of the products of prices and quantities of books ordered but not yet delivered.
7. Define the function `F5` which, for the supplier's ID given as an argument, will check and return how many different books they delivered to the bookstore.
8. Define the `F6` function, which will find and return the alphabetically oldest first letter of the name of those authors who have the letter passed as an argument to the function in the third place from the end.
9. Define a function `F7` that will find and return the smallest ID of the completed order with the highest value.

Chapter 9

Transactions

Transactions are one of the fundamental concepts of modern (relational) database systems. They enable concurrent access to database content by providing necessary synchronization mechanisms. In this chapter we will discuss concurrency control and transaction theory. It is not our goal recalling all the knowledge available on this topic, only introducing these necessary information so that the reader can complete the transaction on his own.

9.1. Concurrency Control

Concurrency control occurs when more than just one query must change the data at the same time. To illustrate the problem of concurrency control, for the purposes of this chapter, let's consider the example of an e-mail box used in Unix where the file `mbox` all news are connected to each other, i.e. they are located one behind the other. For this reason, reading i message processing is simple and the new message is appended at the end file. The question arises: what happens when two processes try to join at the same time messages to the same mailbox? At worst, maybe this lead to corruption of the mailbox and leaving its file at the end two mixed messages. To avoid this type of damage, well-designed email delivery systems use blockages. If the client tries to deliver the message while the mailbox is blocked, it must wait until this message is delivered release the lock. The locking scheme does not fully support concurrency, because only one process can modify the mailbox at a time, which becomes problematic in the case of huge mailboxes.

9.1.1. Read/Write Locks

Reading messages from the mailbox is not that problematic, even if multiple clients do this at the same time. However, the problem arises when when one of the clients wants

to delete, for example, the fifth message, while another the program reads this message at the same time - this program can then receive a damaged or inconsistent view of your mailbox. Therefore, in order ensuring security, even reading the mailbox requires special care caution. A similar situation may occur in a simple database table when several clients read, modify or delete its records at the same time. A system that supports simultaneous read and write operations usually has one implemented locking mechanism consisting of two types of locks called shared locks - disabled locks, or read locks - write locks. Read locks placed on the resource are shared, i.e. they do not block each other - many clients at the same time can read the same resource without disturbing each other. On the other hand pages, write locks are exclusive - i.e. they block both read and write locks and other write locks. At a time, only one customer registers in a specific one resource, at the same time it is not possible to read this resource in this same time. In databases, locking occurs almost constantly and management is carried out internally in a way that is virtually unnoticeable ("transparent"). When the customer performs modification of specific data to these A write lock is applied to the data, which prevents other clients from reading these data.

9.1.2. The Range of Locks

Concurrency can be improved by selecting blocked elements. Instead of blocking the entire resource, you can block only that part of it contains modified data. Minimizing the amount of data blocked in a given moment allows changes to a specific resource to occur simultaneously, under provided they do not conflict with each other. We would like to point out that blockages cause resource consumption - each locking operation involves retrieving the lock, checking, whether the lock is free, release the lock etc. So the locking strategy is there some kind of trade-off between the system load resulting from use blocking and data security.

Most commercially available (commercial) database servers provide lock mechanism (lock) at the record level in the table along with various, often complicated ways to maintain performance after application multiple blockages. It is important to properly determine the scope of blocking at a specific location level to obtain better operational performance for given operations, while leaving the engine slightly less suited to other purposes. The two most important locking strategies in databases are: table locking and record lock.

A table lock is the primary locking strategy available in a database constituting its smallest burden. In case of locking the entire client table may want to save data in a table (insert, delete, update, etc.) and then required there is a write lock that prevents other read operations from being performed and writing to modified data. When no one performs surgery write, it is possible to impose a read lock that does not interfere with

others read locks. The table lock offers variants that cause good behavior performance in certain situations, for example table lock `READ LOCK` allows you to perform some type of simultaneous write operations. Because write lock has higher priority than read lock, write lock request will be placed at the top of the lock queue even when queued read locks are already present. A write lock can get ahead of the lock queue read locks, while read locks cannot overtake write locks. Base data most often uses various locks that are effective at the table level in specific situations. For example, the server uses level locks table when executing commands such as `ALTER TABLE`, regardless of type data storage engine.

In turn, a record lock offers the highest level of concurrency (and at the same time constitutes the greatest load on the server). It is implemented in the engine data store, not on the server. Various data storage engines implement blockades in different, own ways. Readers interested in learning more about locks in PostgreSQL Please refer to the documentation: `https://www.postgresql.org/docs/current/explicit-locking.html`.

9.2. Transactions

A transaction is a group of SQL queries executed atomically as a single query unit. If the database engine can execute an entire group of queries within transaction, then he will do it. However, if at least one of the queries is not there could have been performed due to a failure or any other event, then none of them will be done - according to the "all or nothing" principle. A properly behaving transaction processing system retains four `ACID` properties (**A**tomic, **C**onsistency, **I**solation, **D**urability), which is indivisible, consistent, isolated and durable.

As an example, consider a banking application where the execution of specific operations require the existence of a transaction. We assume that in the database we have two tables (two accounts) `Account_X` and `Y_Account`. To transfer `200 PLN` from the account `Account_X` to the account `Account_Y` we need to follow these steps:

1. First, make sure that the account has `Account_X` are found sufficient funds (min. `PLN 200`).
2. Then we subtract `200 PLN` from the account balance `Account_X`.
3. To add `200 PLN` to the account balance `Account_Y`.

There is a high probability that while doing the above instructions will fail and then only some of them will be completed. What about the rest? What will happen to our money? So we see that it's whole the operation should be executed as a transaction in case of failure any step, all steps completed so far have been rolled back.

Generally, a transaction consists of:

- Start with the command:

  ```
  START TRANSACTION | BEGIN [TRANSACTION]
  ```

  When the user joins the database, starts a new transaction. From this moment, all operations performed by it are operations within transaction. The user can explicitly start a new transaction by requesting ending the current transaction.
- Execution of instructions.
- Commit with `COMMIT`. Once the transaction is approved, it takes place releasing locks created by operations within the transaction, removal of security points, checking of deferred integrity constraints. Changes introduced by operations within a transaction are permanently saved in the database and are visible to other transactions. Once the current transaction is committed, a new transaction is started.
- Rollbacks with `ROLLBACK`. When you roll back the current transaction, all changes made to the database are canceled by operations within the transaction and the established locks are released. When the current transaction is rolled back, a new transaction is started.
- Set optional savepoints with the command:

  ```
  SAVEPOINT <label>
  ```

  thanks to which we can roll back the transaction to a certain point:

  ```
  ROLLBACK TO [SAVEPOINT] <label>
  ```

  and not to the beginning; the effect of the command is to roll back the changes, entered by the operations of the active transaction since creation security point with the given label until the command is executed withdrawal. If we retreat to a point of safety earlier than last created, all security points created later remain deleted; a safety point identified by a label can be removed explicitly from the transaction history with the command:

  ```
  RELEASE SAVEPOINT <label>
  ```

Please note that when you withdraw to a specific save point, we must continue executing subsequent instructions in the transaction as if they typed and performed them for the first time. The advantage of using save points is the ability to roll back to a specific point in a transaction when an error is detected, rather than cancelling (rolling back to the beginning) the entire transaction.

We will now write down the three steps of the transfer described above in the form of a transaction amounts from one account to another.

```
START TRANSACTION;
SELECT balance FROM Account_X WHERE id = 12345;
UPDATE Account_X SET balance = balance - 200.0 WHERE id = 12345;
UPDATE Account_Y SET balance = balance + 200.0 WHERE id = 12345;
COMMIT;
```

9.2.1. ACID

The transaction must function as a single, indivisible unit so that the entire transaction was either approved or rejected. When is the transaction indivisible, then there is no such thing as a partially completed transaction. The rule is "all or nothing" and the transaction is said to be "atomic".

The execution of a transaction should be carried out by a single-state database coherent to the next one. Consistency ensures that a possible failure between instructions 3 and 4 in the 109 example will not cause the money to disappear from `Account_X` client. In case of failure, the transaction will not be confirmed, and no change made by the transaction will be recorded in the database. Until the transaction is completed, its outcome is usually invisible to others transaction. This solution makes it so that if between the execution of lines 3 and 4 from the example from the 109 page, the balance checking procedure will be launched account, the result will still show `200 PLN` in the account `Account_X`. Word use "usually" will become clearer once we discuss isolation levels. Once a transaction is approved, the changes it makes are permanent. Means is that changes must be saved so that data is not lost in the event system failure. With `ACID` compliant transactions the level of base security increases data. The disadvantage is that the server performs additional database processing complicated operations that the user often does not even realize exist realize. Therefore, the database server that handles ACID transactions generally requires more processor power, memory and hard disk space, than a server that does not support this type of transaction. In the case of databases, it is the user decides whether the application requires a transaction or another type security level.

9.2.2. Isolation Levels

The SQL standard defines four levels with specific descriptive rules changes that are or are not visible inside and outside the transaction. The lower level customarily allows for a level of concurrency and simultaneity It is a burden on the system. Every data store implements general information slightly differently, please refer to the manual the user engine to be used. For the purposes of this chapter, information about the four levels of isolation will be briefly and generally presented.

READ UNCOMMITTED

At the isolation level `read uncommitted` transactions have access to uncommitted results yet transaction. In this case, there is a possibility of occurrence an anomaly called `dirty read` when the transaction reads data that has been changed by another transaction, which is then rolled back. Because of this the first transaction read data that no longer exists. This level is very high rarely used in practice because the efficiency is not significantly greater than achieved at other levels and at the same time requires knowledge from the programmer and full awareness of what he is doing.

READ COMMITTED

The default isolation level for most (relational) database systems (including for PostgreSQL) the level is `read committed`. A transaction at this level isolation can only read changes made by other transactions, that have already been approved. However, changes to the transaction being performed will not be visible until it is approved. This level of insulation allows for `read unique` (non-repeatable read) - the same command issued twice may return different data each time because there is a different transaction in parallel modifies attribute values of the same record.

REPEATABLE READ

Level `repeatable read` solves the "dirty data" reading problem that occurs at the `read uncommitted` level. Guarantees that any record is read in a transaction will "look the same" on subsequent reads of it in this one the transaction itself. However, there is a danger of reading the so-called `phantoms`. Reading phantoms may occur when certain is selected range of records, another transaction will insert a new record in the selected range, a later the same range will be selected again. In such a case the user will see the new record that has just been inserted, i.e. the phantom.

SERIALIZABLE

The highest isolation level is `serializable`, which solves the reading problem phantoms by forcing transactions to be executed sequentially, so no they may conflict with each other. It therefore simulates the serial execution of a transaction (one on the other). This level imposes a lock on every record read by and through then a large number of timeout or conflict situations may occur blockages.

To change the isolation level for the current transaction, run the command:

```
SET TRANSACTION ISOLATION LEVEL <level>;
```

In the 9.1 table, we collectively present the isolation levels and the anomalies they represent may appear in them.

Table 9.1: Insulation levels and their disadvantages

Isolation Level	**Dirty reading?**	**Read unique?**	**Reading phantoms?**	**Read locks?**
READ UNCOMMITTED	Yes	Yes	Yes	No
READ COMMITED	No	Yes	Yes	No
REPEATABLE READ	No	No	Yes	No
SERIALIZABLE	No	No	No	Yes

9.2.3. Deadlocks

A serious problem that occurs in databases with concurrent sets transaction, there is a possibility of `deadlock`. Deadlock ("mutual lock" or "death embrace") - occurs when two or more transactions hold each other and request a block on the same resource, resulting in a cycle of dependencies. Deadlocks occur if transactions attempt to apply locks to a resource in different orders. They can also occur when multiple transactions place a lock on the same resource. This creates a situation in which at least two transactions are waiting for each other to free up resources, so that none of them can complete their operation.

We will illustrate a deadlock with an example – table 9.2. We assume two owners of bank accounts with IDs `12345` and `1234` they perform mutually money transfer. To do this, each of them runs a client on their computer `psql` (to simulate this, the reader should run the `psql` client in two separate windows (terminals). User `12345` started his transaction (`Transaction #1`), in which it executes the first command modifying the state of its account (`Account_X`) reducing it by the transfer amount. Before modifying the record is locked in exclusive mode by `Transaction #1`. In the meantime user `1234` also started its transaction (`Transaction #2`) and in the first command, he modified his account balance (`Account_Y`) by reducing it by the amount transfer. Here too the

Table 9.2: Transaction deadlock example

User: 12345 Transaction #1	User: 1234 Transaction #2
START TRANSACTION;	START TRANSACTION;
UPDATE Account_X	UPDATE Account_Y
SET balance = balance - 200	SET balance = balance - 200
WHERE id = 12345;	WHERE id = 1234;
UPDATE Account_Y	UPDATE Account_X
SET balance = balance + 200	SET balance = balance + 200
WHERE id = 1234;	WHERE id = 12345;
COMMIT;	COMMIT;

record is locked by `Transaction #2`. User `12345` in your `Transaction #1` continues to work and tries to modify account balance (`Account_Y`) of user `1234`, increasing it by the transfer amount. But the transaction cannot obtain a lock because this record has been previously locked by `Transaction #2`. `Transaction #1` is suspended, waiting to be released locks. Meanwhile, user `1234` in `Transaction #2` wants to increase account (`Account_X`) by user `12345` about the transfer amount. However, the operation cannot be done completed because the transaction does not obtain a lock on the record it is supposed to be modified – this record was previously locked by `Transaction #1`. `Transaction #2` is also suspended and waits for `Transaction #1` to complete. Both transactions are suspended waiting for the locks to be removed. Neither can continue work because it can't get blocks. Because of this, he also cannot slow down locks for which the second transaction is waiting. A deadlock has occurred. System database management detects the deadlock and resolves it by rolling back one from pending commands of suspended transactions. Next, the user must decide what should happen next with the transactions: whether to withdraw them or continue, but by executing other commands.

The existence of a deadlock is confirmed by the most commonly transmitted data questions. Database systems implement various types of specific forms deadlocks and operation timeout. More advanced base systems damage data and immediate error indication. Other systems canceling the operation after the time instance occurs or may be rolled back a transaction that has access to exclusively locked records (which is therefore easily withdrawn).

Blocking behavior and its sequence are specific to given storage engine, so some engines may get stuck when executing a specific sequence of commands, while others do not in this situation will get stuck. Some deadlocks are unavoidable due to actual problems data conflicts, while others arise as a result of the way we operate given data storage engine. The deadlock cannot be broken without rollback one of the transactions either partially or completely. Remember: Deadlocks happen reality in trading systems and the application should be prepared for their service. Many of them simply try to redo the transaction.

By recording transaction events, their operation is much more effective. Instead of updating the tables on your hard drive after each entry change, the data storage engine may make changes to the copy of the placed data in memory - this operation is very fast. Then the magazine engine data can save record changes in the transaction event log located there on the disk, and therefore permanent - this is also a relatively quick operation, that is due to the use of continuous I/O operations on a small area of the hard disk instead of many random I/O operations scattered across different places on the disk. The table saved on the hard drive is updated from time to time.

Most data storage engines use logging techniques write-ahead logging, saves changes twice on the hard drive. If the failure occurs after saving changes to the event log transactions, but before making these changes to the actual data, the engine Datastore can recover changes after restarting by applying various engine-dependent methods for this purpose.

9.2.4. Transactions in PostgreSQL - `AUTOCOMMIT`

PostgreSQL runs in `AUTOCOMMIT` mode by default - if the user clearly has not initiated a transaction, each query will be automatically executed as separate transaction. Turns the mode on (`on`) or off (`off`). `AUTOCOMMIT` For current connection is possible by setting the value of the variable:

```
postgres=# \set AUTOCOMMIT off;
postgres=# \echo :AUTOCOMMIT
postgres=# \set AUTOCOMMIT on;
```

After setting the variable to `AUTOCOMMIT` off the user will always be in the transaction until the `COMMIT` command is issued or `ROLLBACK`. Then the database will start another transaction. Full list of commands that cause automatic commit transaction should be included in the relevant documentation PostgreSQL version.

9.3. Practice Exercises

1. Add savepoints after each statement in the transaction from the 111 fulfilling side bank transfer. Test the functionality of the rollback transaction individual save points and finally roll back the transaction completely to its beginning.
2. Execute task 1 again, but also run the `psql` client in the second one window (terminal) and log in to your account to view in a new one transaction started there, the contents of the `Account_X` tables and `Account_Y` after each instructions for modifying account balances in the first transaction. Observe the action default insulation level.
3. Perform your own deadlock example from the 9.2 table by creating two sessions of the same user and check how PostgreSQL deals with this problem.
4. Start a new transaction. Increase the price of the book with ID `ISBN = 1` by PLN `500`. Create a security point `S1`. Increase the price of the ID book `ISBN = 15` by PLN `300`. Create a security point `S2`. Delete book with identifier `ISBN = 4`. Roll back the transaction to `S1` and see contents of table `t_book`. Try rolling back the transaction to `S2`. Reverse the entire transaction.
5. Start a new transaction. Remove author named `Mickiewicz`, create security point `S1`, then change the type of the `title` attribute in `t_book` to `VARCHAR(60)`. Try to roll back the transaction to `S1`. What have you noticed?
6. Start two sessions of the same user and start transactions in them. In the first session, execute the command that increases the price of the book by `ISBN = 2` by `10%`, in the second session for the same book, increase its price by `20%`. What did you observe in the second session? Confirm changes in the first session and the price of the book o `ISBN = 2`. In the second session, read the price of the book with `ISBN = 2` and then commit changes. Read the price of the book again with `ISBN = 2` in the first one session.
7. Start two sessions of the same user and start transactions in them. In the first session, set the transaction isolation level to `serializable`:
   `SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;`
   and read information about the 5 initial books arranged in ascending order by `ISBN`. In second session, double the price of the book by `ISBN = 1`. Commit the change. Check again what relation image `t_book` is seen by the first one launched session. Try to increase the price of the book by `ISBN = 1` this session o `50%`. What Have you noticed?

Chapter 10

Creating Database Applications in Java

From the outset, the creators of the Java language aimed to develop mechanisms that would enable programmers to create Java applications utilizing databases.

As a result, Sun Microsystems developed the JDBC (Java Database Connectivity) interface – an Application Programming Interface (API) designed for this purpose.

In this chapter, we will introduce the elements of the Java language that are essential for building a database-driven application.

10.1. Setting Up the Work Environment

To create database applications in Java, three key components are required:

- A relational database management system – PostgreSQL in our case. The installation of the database management system has already been described in subsection 1.2.
- JDK (*Java Development Kit*) – a set of tools and libraries necessary for creating, debugging, and running Java applications.
- A development environment – an Integrated Development Environment (IDE) that facilitates the creation and subsequent development of applications. For the presentation of examples, two environments will be used: IntelliJ IDEA and Visual Studio Code (VSC).

If you choose Visual Studio Code as your development environment, it is recommended to install two additional tools:

- A distributed version control system, Git – a useful tool for tracking changes in source code during software development. Git allows development teams to col-

laborate on shared projects, including tracking change history, resolving conflicts, managing versions, and working on multiple branches of a project simultaneously. Storing projects in repositories also provides an additional layer of security in case project files are damaged.

- Apache Maven – a tool for managing and automating the build process of projects, primarily in Java. Maven offers a range of features, including compilation, testing, and deployment.

. For creating a database application, we will use JDK version 22 (the latest available at the time of writing this script). The installation files are available on the website: `https://www.oracle.com/java/technologies/downloads/#java22`. Simply select your operating system platform and the preferred installation file, e.g., the Windows x64 MSI Installer.

Java JDK includes all the necessary tools for creating (compiling), debugging, and running applications, along with the Java Runtime Environment (JRE). The runtime environment comprises the JVM (*Java Virtual Machine*), library classes, and other auxiliary files needed to run Java applications.

Installing the JDK on Windows is performed using a simple wizard, where you can optionally specify the installation location on the disk. By default, this is the "Program Files" folder on the C drive. Unless specific reasons, it is recommended not to change this default location.

It is beneficial to install the JDK before setting up the development environment, as the IDE installer will automatically detect the JDK's location on the system and configure it properly.

The IntelliJ IDEA development environment can be downloaded from the website: `https://www.jetbrains.com/idea/download/?section=windows`. We will use the free IntelliJ IDEA Community Edition.

Installing IntelliJ IDEA on Windows is also carried out via a wizard, allowing you to specify the installation directory on the disk and configure installation options, such as: creating a desktop shortcut, adding the path to the bin subdirectory to the PATH environment variable, enabling the option to open a folder as a project in IntelliJ from the context menu, or associating selected file extensions with the program (see Fig. 10.1).

The installation file for Visual Studio Code is available on the website: `https://code.visualstudio.com/download`. You should choose the version suitable for your system and hardware platform. For Windows and PC computers, the best choice is the System Installer x64. The installation process is done via a wizard, which essentially involves confirming the subsequent installation steps.

The Apache Maven application is available at `https://maven.apache.org/download.cgi` in the form of an archive file (both binary and source versions). For the binary version,

extract the archive and copy the program folder to the "C:\Program Files" subdirectory. It is also recommended to add the Maven "bin" subdirectory to the "PATH" environment variable.

The Windows installation version of Git can be found at `https://www.git-scm.com/download/win`. You can choose between the installed version or the portable version, available for both 32-bit and 64-bit computers. Download the 64-bit "Git for Windows Setup" file and run the installer. The installer consists of several steps and offers a range of options. We will choose the default settings, which are sufficient for our needs.

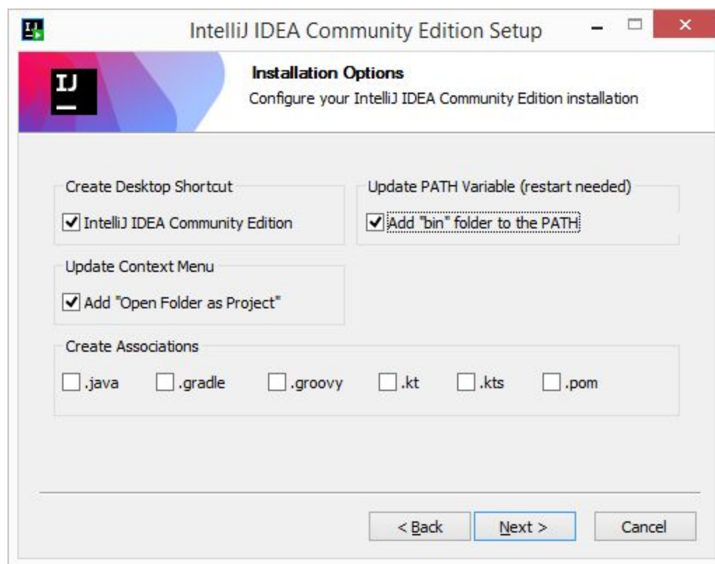

Figure 10.1: Option Selection Window during IntelliJ IDEA Installation

10.2. Java JDBC

The creators of Java decided to use a solution similar to the proven ODBC (Open Database Connectivity) access to databases, developed by Microsoft. The idea is that Java programs communicate with the management program to utilize a previously registered database access driver. This approach allows the creation of database applications in Java that can use databases from different vendors. All that is needed is for the vendor to provide a driver for their database. The JDBC architecture diagram is shown in the figure 10.2.

1. A Java application communicates with the database using the JDBC API, utilizing both the driver manager and the drivers themselves.
2. The JDBC API provides the necessary classes and interfaces to enable communication between a Java application and the database.

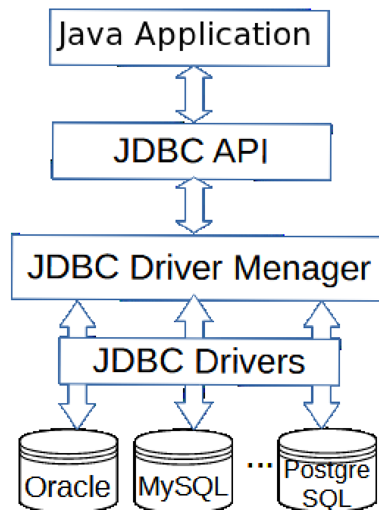

Figure 10.2: JDBC Architecture

3. The JDBC driver manager allows for managing drivers from different vendors. This enables a Java application to use multiple databases from different providers during its execution.
4. A driver acts as an intermediary between the application and the database. It translates commands from the Java application into commands that the database system understands. It also performs the reverse conversion – preparing the results returned by the database in a form that is „understandable" for the Java application.

10.3. Types of JDBC Drivers

The JDBC specification defines four types of drivers:

- Type 1 translates JDBC to ODBC and uses ODBC drivers to communicate with the database. This type is not recommended.
- Type 2 is partially written in Java and partially in native code for a specific platform. It communicates with the database using the client interface of the given database system. It also requires the installation of platform-specific software components.
- Type 3 is written in Java and uses a database-independent communication protocol. Communication occurs with a server-side component that translates the driver's requests into the specific protocol of the given database system. The client-side software is independent of the database system.
- Type 4 is written in Java and translates JDBC requests into the specific protocol of the given database system.

Most database system vendors provide Type 3 or Type 4 drivers.

10.4. Structure of a Database Application in Java

In order for a Java application to use a database, several steps must be followed:

1. Establish a connection to the database.
2. Create a query.
3. Execute the query.
4. Process the query results.
5. Close the connection to the database.

During the application's execution, various queries can be created, and the results can be processed in different ways. Steps 2 – 4 may be repeated multiple times.

10.4.1. Example – Establishing a Connection to the Database

Currently, various development environments are used to create applications, which make the work of developers easier. In this subsection, we will show how to create a project in two popular, freely licensed development environments: IntelliJ IDEA and Visual Studio Code. Additionally, these applications are available for the most popular operating systems: Windows, Linux, and macOS.

Project in IntelliJ IDEA

We begin by creating a new project in IntelliJ IDEA. In the wizard window (Fig. 10.3), we choose the project name, the location of the project files, check the option to create a Git repository, select the installed JDK version, Maven Archetype, and choose the archetype (a project template that includes a predefined directory structure, configuration files, and dependencies necessary for building the selected type of project) `https://maven.apache.org/archetypes/maven-archetype-quickstart/` in its latest version. After creating the project, a window may appear with a message from Microsoft Defender Configuration. In this case, click on Automatically to add our application to the Defender exceptions list.

Next, we will add a dependency to the project, which represents the driver for the PostgreSQL database (Fig. 10.4). In a Maven project, dependencies are added to the „pom.xml" file (Listing 10.1).

Listing 10.1: *PostgreSQL Dependency in the Project*

```
<dependency>
    <groupId>org.postgresql</groupId>
    <artifactId>postgresql</artifactId>
    <version>42.7.3</version>
</dependency>
```

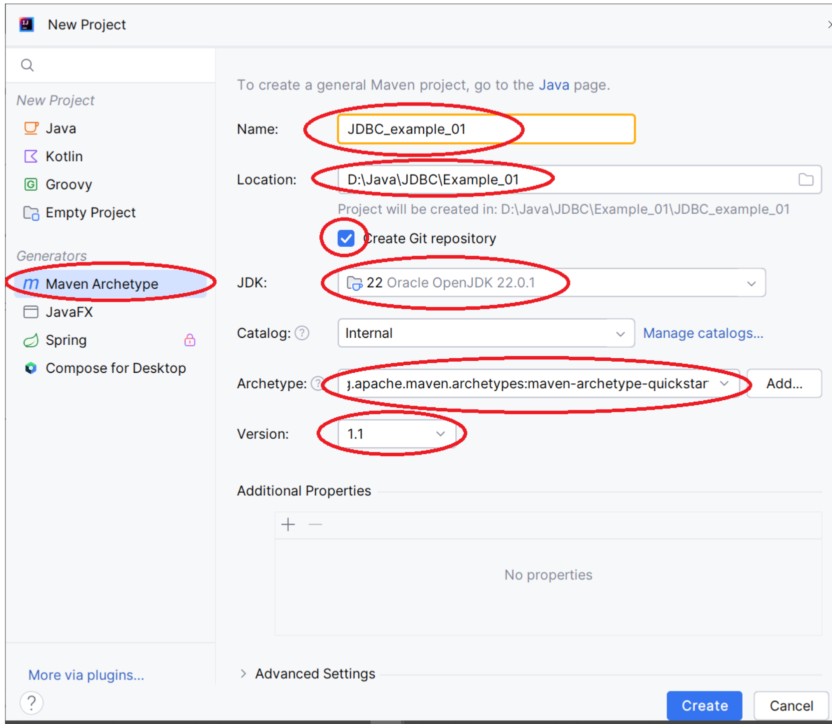

Figure 10.3: New Project Wizard Window in IntelliJ IDEA

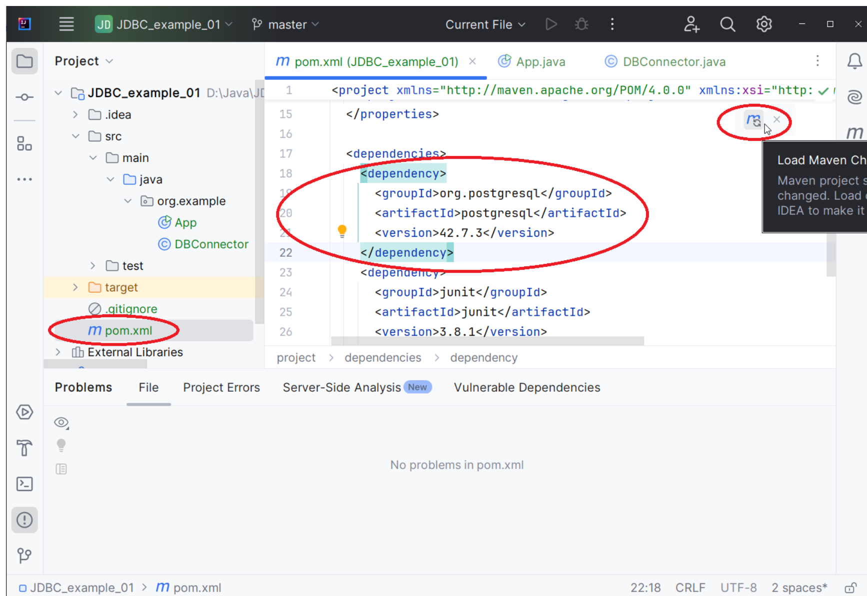

Figure 10.4: Code for the PostgreSQL Database Driver Dependency

Next, the changes should be committed by clicking the Load Maven Changes button (Figure 10.4), as without this step, the changes won't take effect. However, this does not load the database driver yet; this task is handled by the driver manager. You can check the available driver versions on the website: `https://jdbc.postgresql.org`.

Let's define a class whose task will be to establish and close the database connection (Listing 10.2).

The class `Connection` is used to establish a connection with the database. It represents an open connection to a data source and is used to execute SQL statements and manage transactions. An object of the `Connection` class is returned by the `getConnection` method of the `DriverManager` class, which manages the database drivers. The connection is made to the database whose location is specified by the URL address. Login is performed for the given database user and password. It is important to make sure that before running the Java program, you:

- Start the database server (PostgreSQL);
- Create a database with the given name (in this case, `test_00`) and make it available on port number 5432 on the local machine;
- Create a user with the given login (postgres) and password (12345).

LISTING 10.2: *Class responsible for establishing and closing the database connection*

```
1  package org.example;
2  
3  import java.sql.Connection;
4  import java.sql.DriverManager;
5  import java.sql.SQLException;
6  public class DBConnector {
7    private static final String URL = "jdbc:postgresql://localhost
       :5432/test_00";
8    private static final String user = "postgres";
9    private static final String password = "12345";
10   public static Connection connect() throws SQLException {
11      Connection connection =
12         DriverManager.getConnection(URL, user, password);
13      System.out.println("Connected");
14      return connection;
15   }
16   public static void close(Connection connection) throws
       SQLException {
17       connection.close();
18   }
19 }
```

Once the connection is established, the message „Connected” will appear on the screen, which is intended for demonstration purposes and is unnecessary in a regular program. If the connection fails, an exception of the class `SQLException` will be thrown.

To test the connection to the database, we will write a simple program – listing 10.3. In the main method, the static connect method from the `DBConnector` class is called to establish the connection with the database. The code responsible for interacting with the database is placed in the `try-catch-finally` block because exceptional situations may occur during the program execution, which need to be handled. These include, among others: issues with accessing the database (e.g., opening or closing it) or errors related to SQL queries, such as syntax errors or transaction errors.

In line 15, an object `stat` of type `Statement` is created, which allows the execution of SQL commands on the connected database. The execution of SQL commands is handled by the `executeUpdate` and `executeQuery` methods of this class. In the example program, four SQL commands are executed: the first creates a table `Greetings` with a `Message` column, the second inserts the text „Hello World!” into the table, the third command – a query – retrieves the entire contents of the `Greetings` table into an object of type `ResultSet`, and the fourth deletes the created table from the database.

The `ResultSet` interface provides access to the table containing the query results. There can be multiple rows in the result table, and access to the next row is through the `next` method, which moves the cursor to the next row or returns `null` when there are no more rows. We can read individual values from the table using the access methods from the `ResultSet` interface, which handle all primitive types and strings, such as `getInt`, `getBoolean`, `getString`. Values can be retrieved by passing the column number or name as a parameter to the access method. Columns are numbered starting from 1. To ensure maximum portability, columns in the result set should be read from left to right in each row, and each column should be read only once.

The data processing is handled by the `while` loop in line 21, where the `next` method is responsible for iterating through the result rows, and the `getString` method reads the value from the first column of the result set.

Listing 10.3: *Public class with code for checking database connections*

```
1 package org.example;
2
3 import java.sql.Connection;
4 import java.sql.ResultSet;
5 import java.sql.SQLException;
6 import java.sql.Statement;
7
8 public class Example_01 {
```

```
9   public static void main( String[] args ) {
10    Connection connection = null;
11    ResultSet result = null;
12    try {
13      connection = DBConnector.connect();
14      Statement statement = connection.createStatement();
15      statement.executeUpdate("CREATE TABLE Greetings (Message
    CHAR(20))");
16      statement.executeUpdate("INSERT INTO Greetings VALUES ('
    Hello World!')");
17      result = statement.executeQuery("SELECT * FROM Greetings");
18      while (result.next())
19        System.out.println(result.getString(1));
20      statement.executeUpdate("DROP TABLE Greetings");
21    }
22    catch (SQLException e) {
23      System.err.println("SQL Status: " + e.getSQLState());
24      System.err.println("Error code: " + e.getErrorCode());
25      System.err.println(e.getMessage());
26      Throwable t = e.getCause();
27      while (t != null) {
28         System.out.println("Exception reason: " + t);
29         t = t.getCause();
30      }
31    }
32    finally {
33      try {
34         if (result != null) {
35           result.close();                 // release of resources
36         }
37         if (connection != null) {
38            DBConnector.close(connection); // closing the
    connection
39         }
40      }
41      catch (SQLException e) {
42        e.printStackTrace();
43      }
44    }
45  }
46 }
```

Project in Visual Studio Code

To start, Visual Studio Code (VSC) needs to be set up for efficient Java application development. One of VSC's strengths is its ability to use extensions, which significantly enhance the environment's capabilities. When developing Java applications, we need to install the Java Extension Pack, and it is also recommended to install the "Language Support for Java (TM) by Red Hat" extension (which includes features like code completion and support for Maven and Gradle). The "Maven for Java" extension will be required if we create projects using Maven.

To add an extension, click on the Extensions button (see figure 10.5), type in a relevant search term, such as "Java", and then select and install the appropriate extension from the list. Information about the selected extension will appear on the right side of the window—it's worth reviewing. This process can be repeated to install other extensions, like "Language Support for Java (TM) by Red Hat" (which is located further down the extension list) and "Maven for Java" (just type "Maven" into the search box).

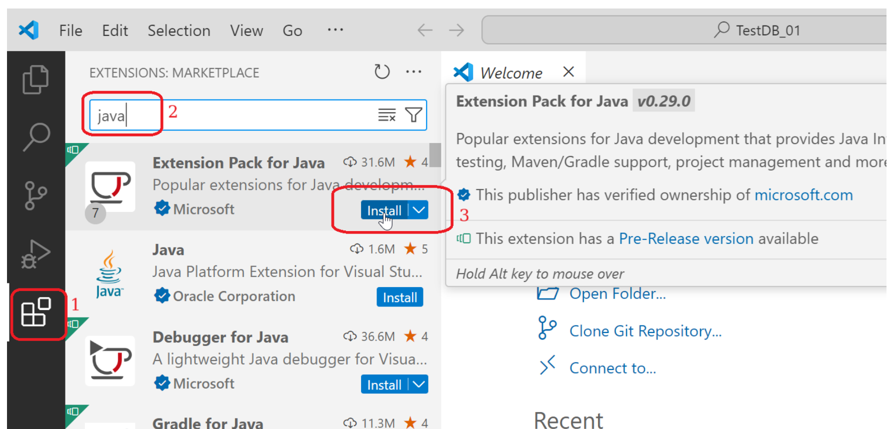

Figure 10.5: Visual Studio Code – installing extensions

To create a project in Visual Studio Code, follow these steps:

1. From the "View" menu, select "Command Palette" (`Ctrl+Shift+P`).
2. In the command palette, type „Maven" and select "Maven: New Project..." from the suggestions.
3. Choose an archetype, such as "maven-archetype-quickstart", similar to the process in IntelliJ.

Next, provide the following:

- The version number (e.g., "1.1").
- A group ID (e.g., "org.example").
- An artifact ID, which will also serve as the project name (e.g., "jdbc_example_01").
- The directory where the project will be saved.

In interactive mode within the terminal, you will also need to specify the version number. To use default values, simply press "Enter". Once the project creation is complete, press any key to close the terminal.

Maven will generate the folder structure and a "pom.xml" file (see Figure 10.6).

To access the project folder structure, click the "Explorer" button (`Ctrl+Shift+E`) on the left-hand toolbar (Figure 10.6 (1)). In the subdirectory *..\scr\java\org\example*, you can either: place the files from Listings 10.2 and 10.3, or create new Java files by clicking the button to add a new file (Figure 10.6 (2)) and entering the code.

Finally, add the dependency from Listing 10.1 to the "pom.xml" file. After making changes, a dialog box will appear prompting you to synchronize the configuration. Confirm the changes.

Your program is now ready to compile and run.

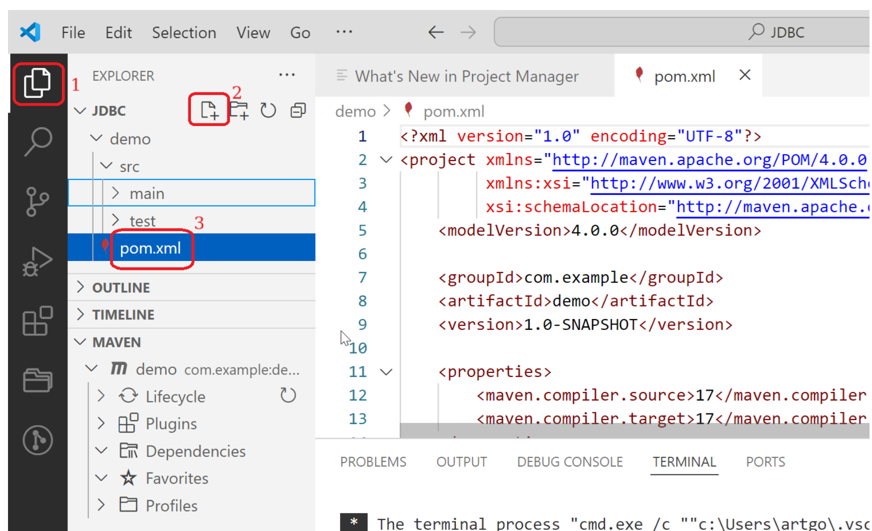

Figure 10.6: Visual Studio Code – project view

10.5. Working with SQL Commands

To execute SQL commands, an object of type `Statement` is required. It can be obtained using an object of type `Connection`, which in turn is returned by the `getConnection` method of the `DriverManager` class and represents a connection to a specific database.

```
Connection connection = DriverManager.getConnection(URL, user, password);
Statement statement = connection.createStatement();
```

Using an object of the `Statement` class, SQL commands can be executed. A command can be written as a string:

```
String command = "UPDATE Cars" + " SET Price = Price + 1000"
                       + " WHERE Brand NOT LIKE '%Fiat%' ";
```

and then used in the `executeUpdate` method of the `Statement` class:

```
int changedRecords = statement.executeUpdate(command);
```

The method returns the number of modified records or −1 if no modification occurred (a value −1 is also returned for commands that do not modify data). In this example, it would return the number of cars whose prices were increased.

The `executeUpdate` method is used for executing SQL commands such as `INSERT`, `UPDATE`, and `DELETE`, as well as data definition commands like `CREATE TABLE` and `DROP TABLE`.

To execute commands that return data, such as `SELECT`, the `executeQuery` method is used. The `execute` method can also be used, allowing the execution of any SQL command. It is typically used for commands entered by the user during the application's runtime.

The `executeQuery` method returns an object of type `ResultSet`, which is used to read individual records from the result of the command.

```
ResultSet results = statement.executeQuery("SELECT * FROM Cars");
```

Processing results is usually done in a loop:

```
while(results.next()) {
   // operation on a single row
}
```

Access to the first row of the result table is possible after executing the `next` method. This method returns `null` upon reaching the end of the table.

When browsing the contents of a table row, you can use various methods to access individual fields, such as:

```
String name = results.getString(1);
double price = rs.getDouble("Price");
```

Access methods corresponding to each Java data type, such as `getString` or `getDouble`, are available. There are two versions: one with a numeric parameter and another with a string parameter representing the name of the database table column. Using a numeric parameter refers to the column with the given number. For instance, the method `results.getString(1)` returns the value of the first column for a given row. (**Note**: Columns are numbered starting from 1, not 0.)

The `execute` method executes an SQL command and returns `true` if the result is a set of records. To retrieve results, the methods `getResultSet` or `getUpdateCount` should be used.

For an object of type `Connection`, multiple objects of type `Statement` can be created. The same `Statement` object can be reused multiple times to execute different SQL commands. It is important to note that a `Statement` object can manage at most one `ResultSet` object at a time. If the program executes multiple queries and the results need to be processed simultaneously, multiple `Statement` objects should be created. Some database drivers, such as Microsoft SQL Server, allow only one active `Statement` object. To obtain information about the maximum number of active `Statement` objects allowed by a given driver, you can use the `getMaxStatement` method of the `DatabaseMetaData` interface.

After finishing the use of objects like `ResultSet`, `Statement`, or `Connection`, the `close` method should be called. The `close` method of a `Statement` object closes the associated result set if it is open. Similarly, the `close` method of the `Connection` class closes all statement objects associated with that connection.

When using short-lived connections, ensure that the connection object does not remain open. In such cases, the `close` method should be called within a `finally` block – see listing 10.4.

Listing 10.4: *Code Skeleton for Short-Lived Database Connections*

```
try {
   Connection connection = ...;
   try {
     Statement statement = connection.createStatement();
     ResultSet result = statement.executeQuery(query);
     // processing the query result
   }
   finally {
     connection.close()
   }
}
catch {
  // exception handling
}
```

10.6. SQLException Exceptions

During the execution of a single SQL command, multiple errors may occur, generating an exception. For a `SQLException`, a chain of `SQLException` objects can be created, which can be retrieved using the `getNextException` method. Each exception in the chain corresponds to one error. Starting with Java 6, you can use the `Iterator<Throwable>` iterator to iterate through all exceptions using a single loop:

```
for (Throwable e : sqlException) {
  // operations on the object "e", np. e.printStackTrace();
}
```

It is important to note that the iterator, like the `getCause` method, provides access to `Throwable` objects, while the `getNextException` method is specific to `SQLException` objects.

Detailed information about an exception can be obtained using the methods `getSQLState` and `getErrorCode`. The `getMessage` method returns a description of the exception. The `getSQLState` method provides information in the form of a five-character error or warning code. In this code, the first two characters represent the error class, while the remaining three provide the subclass, offering more specific details about the issue. For instance, a code starting with "80" indicates connection problems, with subclasses detailing specific issues:

```
- "08001": SQL-client unable to establish SQL-connection.
- "08003": Connection does not exist.
- "08006": Connection failure.
```

For example, when a database connection fails, you might receive the following details:

```
SQLState: 08001
Error Code: 0
Message: No suitable driver found for
              jdbc:postgresql://localhost:5432/mydatabase
```

Database drivers can also report warnings. This is done using the `SQLWarning` class, which is a subclass of `SQLException`. Information about the occurrence of warnings can be retrieved by calling the `getSQLState` and `getErrorCode` methods. Similar to SQL exceptions, warnings also form chains. To retrieve all warnings, you can use a loop:

```
SQLWarning w = stat.getWarning();
while (w != null) { // operations on the w object
  w = w.nextWarning();
}
```

To avoid duplicating code for handling database-related exceptions, we will write the `main` method (Listing 10.5) in a way that it comprehensively handles exceptions. Meanwhile, methods demonstrating various database usage examples will declare the possibility of throwing an `SQLException` – for example, the `createTable` method from Listing 10.5.

In subsequent examples, simply replace the `createTable` method with a new method or place the call to the new method on the next line after the `createTable` method. Naturally, the order of database operations is crucial. First, we should create the table, then populate it with data, and finally execute a query related to that table. If, for instance, we run a program that creates a table for the second time, an exception will likely be thrown, indicating that the table already exists. On the other hand, various queries can be executed multiple times on tables that have already been created and populated with data.

Listing 10.5: *The program code with the* `main` *method that handles SQLException exceptions*

```
1  public class Example_02 {
2      public static void main( String[] args ) {
3          Connection connection = null;
4          try {
5              createTable(connection);
6          }
7          catch (SQLException e) {
8              System.err.println("SQL State: " + e.getSQLState());
9              System.err.println("Error Code: " + e.getErrorCode());
10             System.err.println(e.getMessage());
11             Throwable t = e.getCause();
12             while (t != null) {
13                 System.out.println("Cause: " + t);
14                 t = t.getCause();
15             }
16         }
17         finally {
18             try {
19               if (connection != null) {
20                 DBConnector.close(connection);
21               }
22             } catch (SQLException e) {
23                 e.printStackTrace();
24             }
25         }
```

```
26     }
27     public static void createTable(Connection connection)
28                           throws SQLException { // ... }
29 }
```

10.7. Example – creating a table in the database

To create a table in the database, use the `executeUpdate` method of the `Statement` class, passing an SQL command as a parameter to create a specific table (Listing 10.6, line 7). The `executeUpdate` method returns "0" for SQL commands that do not return a value or the number of updated rows for SQL commands that make changes.

In line 5, there is an SQL command that, without any warnings, removes the table `t_client`, if it exists, regardless of whether the table is empty or filled with data.

Listing 10.6: *The* `createTable` *method of the* `Example_2` *class, which creates a table in the database*

```
1  public static void createTable(Connection connection)
2                              throws SQLException {
3    connection = DBConnector.connect();
4    Statement statement = connection.createStatement();
5    int res = statement.executeUpdate(
6      "DROP TABLE IF EXISTS t_client");
7      String query = "CREATE TABLE t_client(" +
8      "client_id int PRIMARY KEY," +
9      "name varchar(30)," +
10     "lsurname varchar(40)," +
11     "street varchar(30)," +
12     "city varchar(30)," +
13     "voivodeship varchar(30)," +
14     "zip_code char(6) CHECK(zip_code ~ ('^\\d{2}-\\d{3}$'))," +
15     "telephone char(11) CHECK(telephone ~ ('^\\d{3}-\\d{3}-\\d{3}$
       ')));";
16     res = statement.executeUpdate(query);
17     statement.close();
18   }
```

10.8. Example – inserting data into the table

It is rare to include data intended for populating a table directly in Java code, especially when there is a large amount of data. In the presented example, the data will be stored

in a text file (Listing 10.7), which contains SQL commands for inserting records into the t_client table.

Listing 10.7: *The* t_client.sql *file, containing SQL commands for populating the* t_client *table with sample data*

```
1 INSERT INTO t_client VALUES('1', 'Jan', 'Kowalski',
2  'Akacjowa', 'Warszawa', 'mazowieckie', '00-950', '502-501-501');
3 INSERT INTO t_client VALUES('2', 'Antoni', 'Kwiatkowski',
4  '1 Maja', 'Poznań', 'wielkopolskie', '60-002', '503-533-533');
5 INSERT INTO t_client VALUES('3', 'Maja', 'Smith',
6  'Szczytowa', 'Łódź', 'łódzkie', '70-445', '501-233-453');
7 INSERT INTO t_client VALUES('4', 'Hernyk', 'Maciąg', 'Leśna',
8  'Iława', 'warmińsko-mazurskie', '14-200', '607-113-783');
9 INSERT INTO t_client VALUES('5', 'Michał', 'Gołąb',
10  'Ikara', 'Siewierz', 'śląskie', '42-470', '606-552-983');
11 INSERT INTO t_client VALUES('6', 'Anna', 'Basińska',
12  'Lipowa', 'Sopot', 'pomorskie', '80-336', '505-236-903');
13 INSERT INTO t_client VALUES('7', 'Dariusz', 'Wałek',
14  '3 Maja', 'Poronin', 'małopolskie', '34-425', '602-003-677');
15 INSERT INTO t_client VALUES('8', 'Telimena', 'Walewska',
16  'Rocha', 'Gdańsk', 'pomorskie', '80-002', '603-535-517');
17 INSERT INTO t_client VALUES('9', 'Joanna', 'Kowal',
18  '1 Maja', 'Sosnowiec', 'śląskie', '34-112', '703-522-636');
19 INSERT INTO t_client VALUES('10', 'Grzegorz', 'Kwiatek',
20  'Ludowa', 'Ełk', 'warmińsko-mazurskie', '19-300', '701-583-983');
```

The insertDate method (Listing 10.8) essentially reads successive lines from a text file, which contain SQL commands, and executes them using the execute method.

Note that this method can easily be modified so that the filename containing the SQL commands is passed as a parameter. This would allow you to write a simple program where the filename with SQL commands is provided via the command line. In this way, you can quickly create multiple database tables and populate them with data.

Listing 10.8: *The* insertDate *method of the* Example_2 *class, which populates the* t_client *table with sample data*

```
1 public static void insertDate(Connection connection) throws
      SQLException {
2    connection = DBConnector.connect();
3    try {
4       BufferedReader file = new BufferedReader(new FileReader("
     t_client.sql"));
5       Statement statement = connection.createStatement();
```

```
6        String query;
7        while ((query = file.readLine()) != null) {
8           boolean result = statement.execute(query);
9        }
10       statement.close();
11       file.close();
12    } catch (FileNotFoundException e) {
13        System.err.println(e.getMessage());
14    } catch (IOException ex) {
15       System.err.println(ex.getMessage());
16    }}
```

10.9. Example – executing queries

When searching for the data we are interested in from the database, we can use a regular SQL query or a prepared statement.

A prepared statement allows us to create SQL queries with parameters, which helps protect against SQL injection attacks. Instead of allowing the user to create complete queries, we provide them the ability to input parameters for queries that we have prepared.

Another advantage of prepared statements is their increased efficiency, as the database engine can cache these queries.

To use prepared statements, we need to create an object of type `PreparedStatement`, to which we pass the SQL query, for example:

```
String sqlQuery = "SELECT publisher, price FROM t_book " +
                         "WHERE publisher = ?" + " AND price > ?";
PreparedStatement preparedStatement =
                          connection.prepareStatement(sqlQuery);
```

In a query, a placeholder for a parameter is indicated by a question mark ("?"). The value of the parameter is provided using setter methods, which allow you to pass parameters of a specific type, such as `setInt`, `setString`, etc. These methods take two parameters.

The first parameter of the method specifies the variable (the index of the "?") to which we want to assign a value. The value "1" refers to the first occurrence of the "?" in the SQL statement. The second parameter specifies the value to assign to the variable. In the example, the first occurrence of "?" represents the publisher's name, which is a `String`. The parameter value is set using the `preparedStatement` instruction, for example:

```
preparedStatement.setString(1, "Addison-Wesley");
```

The second occurrence of "?" represents the minimum price of the book, for example:

```
preparedStatement.setDouble(2, 50);
```

The `selectPublisher` method, shown in Listing 10.9, allows creating and executing a query that searches for books in the table that are published by the specified publisher and cost more than the given amount. The query results will display information about the books that meet the criteria defined by the user. To execute the query, we use the `executeQuery` method from the `PreparedStatement` class.

Listing 10.9: *The* `selectPublisher` *method of the* `Example_2` *class, which prints to the screen books from the specified publisher that cost more than the given amount*

```
1  public static void selectPublisher(Connection connection,
2              String name, double price) throws SQLException {
3    connection = DBConnector.connect();
4    Statement statement = connection.createStatement();
5    String sqlQuery = "SELECT publisher, price FROM t_book" +
6                      " WHERE publisher = ?" + " AND price > ?";
7    PreparedStatement preparedStatement =
8                    connection.prepareStatement(sqlQuery);
9    preparedStatement.setString(1, name);
10   preparedStatement.setDouble(2, price);
11   ResultSet results = preparedStatement.executeQuery();
12   while ( results.next() ) {
13     System.out.print(results.getString("publisher") + " - ");
14     System.out.println(results.getDouble("price") + ". ");
15   }
16   statement.close();
17 }
```

10.10. Example – creating an application with a GUI (Graphical User Interface)

In this example, we will demonstrate how to use a simple graphical user interface (GUI) to display data retrieved from a database.

The application will allow the user to browse data from different tables (Fig. 10.7). At the top of the window, there is a drop-down list from which the user can select any table from the connected database. After selecting a table, the column names and the values from the first record will be displayed on the panel. Using the "Previous" and "Next" buttons, the user can browse through all records in the selected table. When a new table is chosen, the column names from the new table will appear on the panel.

The key parts of the application are shown in Listings 10.10 to 10.14. The full code of the program can be found in "Appendix B".

To create the GUI panel, the following components will be needed:

```
private JButton previousButton;
private JButton nextButton;
private DBPanel dataPanel;
private JScrollPane scrollPane;
private final JComboBox<String> tableNames;
```

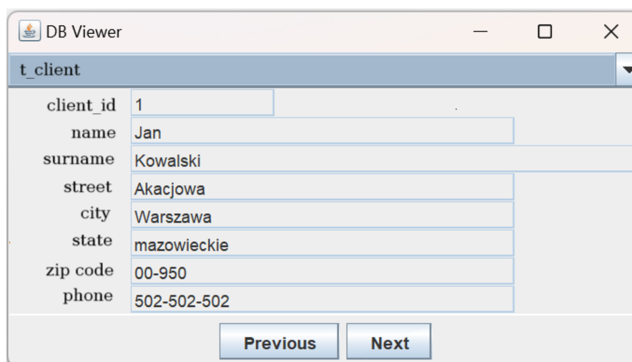

Figure 10.7: Application window

These are buttons (`JButton`) that allow navigating to the next or previous record, a drop-down list (`JComboBox`) for selecting the table, and a panel with scrolling capability (`JScrollPane`). It allows scrolling the content when it does not fit within the window. Inside this panel, there will be another panel (`DBPanel`) that displays records from the database.

The `DBPanel` class is a subclass of `JPanel` – a component where records from the selected table are displayed.

In Listing 10.10, the constructor definition of the `DBPanel` class is provided. The constructor receives a `RowSet` parameter that contains the result of an SQL query fetching all records from the table selected by the user.

The `RowSet` class allows storing a set of data, similar to the `ResultSet` class, but it is more flexible and offers more capabilities, such as working in disconnected mode from the database, easily scrolling data backward and forward, or updating data.

The layout manager `GridBagLayout` is used to arrange the components on the panel (line 3). It arranges components in a two-dimensional grid but also offers additional features, such as precise positioning of components (fields `gbc.gridx` and `gbc.gridy`), specifying the size (`gbc.gridwidth` and `gbc.gridheight`), and aligning the component's

content (`gbc.anchor`). The parameters for positioning components in this layout are defined using an object of the `GridBagConstraints` class.

The table content is displayed on the panel in two columns. In the first column, labels are used to display the column names of the table. The labels are right-aligned (line 12), while text fields are used to display the data, which are left-aligned (line 19). Thanks to using text fields, the user can select and copy the content with the mouse, but they cannot modify it (line 16).

In line 7, the table metadata is read, and line 8 accesses this data – specifically, it retrieves the number of columns in the table. During each iteration of the loop, a pair of components representing the column name (lines 13 and 17) and a text field, which will later hold the value of the first record in that column, is added to the panel. The column name is retrieved in line 10.

For the text fields, an empty text field is created in the constructor. Only the size of the data (e.g., the length of the string or the number of characters for a number) is retrieved, and the field's size is set to ensure that the entire content will be displayed on the screen (lines 14 and 15).

The `add` method adds a component to the panel, and the second parameter specifies a constraint. The `fields` array is a list of `JTextField` elements, where each text field stores the name of the columns from the selected table in the database. All the text fields created in the loop are also added to the `fields` array (line 17).

Listing 10.10: *The constructor of the* `DBPanel` *class*

```
1  public DBPanel(RowSet rs) throws SQLException {
2    fields = new ArrayList<>();
3    setLayout(new GridBagLayout());
4    GridBagConstraints gbc = new GridBagConstraints();
5    gbc.gridwidth = 1;
6    gbc.gridheight = 1;
7    ResultSetMetaData rsmd = rs.getMetaData();
8    for (int i = 1; i <= rsmd.getColumnCount(); i++) {
9      gbc.gridy = i - 1;
10     String columnName = rsmd.getColumnLabel(i);
11     gbc.gridx = 0;
12     gbc.anchor = GridBagConstraints.EAST;
13     add(new JLabel(columnName + "  "), gbc);
14     int columnWidth = rsmd.getColumnDisplaySize(i);
15     var tb = new JTextField(columnWidth);
16     tb.setEditable(false);
17     fields.add(tb);
18     gbc.gridx = 1;
```

```
19    gbc.anchor = GridBagConstraints.WEST;
20    add(tb, gbc);}
```

Since the application also displays information about table names and column names, it must obtain this information from somewhere. This type of data is called metadata, and in Java, it can be retrieved from the database using the DatabaseMetaData class. The DatabaseMetaData class is used to read information about the database, while the ResultSetMetaData class allows you to obtain information about a result set.

When executing a query and receiving a result set, you can also obtain information about the number of columns as well as the name, type, and size of each column. A loop to browse such metadata may look like this:

```
ResultSet rs = stat.executeQuery("SELECT * FROM " + tableName);
ResultSetMetaData metaData = rs.getMetaData();
for (int i = 1; i <= metaData.getColumnCount(); i++) {
  String columnName = metaData.getColumnLabel(i);
  int columnWidth = metaData.getColumnDisplaySize(i);
  ...
}
```

In listing 10.11, the method for displaying a single data record on the panel is shown. This method is used both to display the first record immediately after selecting a table, and also when navigating to the previous or next record.

The data record to be displayed on the panel is passed as a parameter to the method. Inside the method, within a loop that iterates over all the elements in the fields array list, the next text field is retrieved (line 6) and the text from the data record (represented by the variable rs) is inserted into it (line 5). The text is displayed in line 7.

Listing 10.11: *The* showRow *method of the* DBPanel *class displays a data record on the panel*

```
1  public void showRow(ResultSet rs) {
2    try {
3      if (rs == null) return;
4      for (int i = 1; i <= fields.size(); i++) {
5        String field = rs == null ? "" : rs.getString(i);
6         JTextField tb = fields.get(i - 1);
7         tb.setText(field);
8       }
9     }
10    catch (SQLException ex) { MySQLExceptionInfo.print(ex); }
11 }
```

In the constructor of the public class (listing 10.12), there are instructions responsible for:

- creating the components that will be displayed in the application window;
- creating and assigning appropriate listeners to selected components (e.g., buttons, drop-down list) in order to interact with the user;
- preparing data that will appear in the components visible when the application starts, such as tables in the drop-down list.

In line 2, an object of the `JComboBox` class is created, which should be filled with the names of tables from the database. In line 4, the application connects to the database, and lines 4 and 5 read the metadata about the tables. In the following instructions, inside the `while` loop, the names of tables are read and added to the drop-down list `tableNames`. The `getTable` method has four parameters: the first specifies the catalog where tables are searched, the second defines the schema pattern (group of tables), the third is the table name pattern (e.g., `UJD` to search for tables starting with `UJD`), and the fourth specifies an array of string values representing object types (e.g., `TABLE`, `SYSTEM TABLE`). The `null` values will cause the method to take the default value.

After using the `ResultSet` object, the `close` method should be called. This releases the database resources. The use of the `close` method is guaranteed by the `try-finally` construct.

In line 17, an action listener is added to the `tableNames` object, which will respond to the user's selection from the list. In our case, it will call the `showTable` method, passing the table name (the `tableName.getSelectedItem` method returns the selected item from the drop-down list) and the object representing the connection to the database. In line 22, the object representing the drop-down list is added to the application window.

The following instructions (lines 23-32) handle the event of closing the application window. Handling this event involves terminating the connection to the database. Then, an object representing the panel is created, where the buttons will be placed, along with the buttons themselves. Action listeners are added to the buttons. The "Previous" button handles calling the `showPreviousRow` method, while the "Next" button handles calling the `showNextRow` method. In the last instruction of the constructor, a check is made to see if there are any items in the drop-down list. If so, the first record from the first table is displayed in the application window.

Listing 10.12: *Constructor of the public class* `MainFrame`

```
1 public MainFrame() {
2   tableNames = new JComboBox<String>();
3   try {
4     connection = DBConnector.connect();
5     DatabaseMetaData meta = connection.getMetaData();
```

```
6      ResultSet mrs = meta.getTables(null, null, null, new String[]{
     "TABLE"});
7      try {
8        while (mrs.next())
9          tableNames.addItem(mrs.getString(3));
10     } finally {
11       mrs.close();
12     }
13   }
14   catch (SQLException ex) {
15     MySQLExceptionInfo.print(ex);
16   }
17
18   tableNames.addActionListener( new ActionListener() {
19     @Override public void actionPerformed(ActionEvent e) {
20        showTable( (String) tableNames.getSelectedItem(),
     connection);
21    }} );
22
23    add(tableNames, BorderLayout.NORTH);
24
25    addWindowListener(new WindowAdapter()  {
26      public void windowClosing(WindowEvent event)  {
27        try {
28          if (connection != null) connection.close();
29        }
30        catch (SQLException ex) {
31          MySQLExceptionInfo.print(ex);
32        }
33      }
34    });
35
36    JPanel buttonPanel = new JPanel();
37    add(buttonPanel, BorderLayout.SOUTH);
38    previousButton = new JButton("Previous");
39    previousButton.addActionListener( new ActionListener() {
40      @Override public void actionPerformed(ActionEvent e) {
41        showPreviousRow();
42      }
43    });
44    buttonPanel.add(previousButton);
45    nextButton = new JButton("Next");
```

```
46    nextButton.addActionListener( new ActionListener() {
47      @Override public void actionPerformed(ActionEvent e) {
48        showNextRow();
49      }
50    });
51    buttonPanel.add(nextButton);
52    if (tableNames.getItemCount() > 0)
53      showTable(tableNames.getItemAt(0), connection);
54 }
```

In listing 10.13, there is the `showTable` method, which is used to display a single data record from the table whose name is passed as the first parameter. This method is called every time a new table is selected from the dropdown list. This involves removing the current panel from the application window and replacing it with a new one that is tailored to display data from the new table. At the beginning of the method, a connection to the database is made, and all data from the table specified by the user is retrieved.

In line 4, an object implementing the `RowSetFactory` interface is created, which is used to generate a `CachedRowSet` result set. This type allows for working in a disconnected mode from the database, storing results in RAM. The `crs` object is a field of the `MainFrame` class of type `CachedRowSet`. The `crs.populate(result);` instruction fills the data set with the results, and the `crs.setTableName(tableName)` instruction links the data set with the table name. Then, the panel displaying the previously selected table is removed (if it exists), and a new panel is created to display data from the newly selected table. It is easier to create a new panel than to modify the existing one. New panels are added to the application window, replacing the previous components (lines 9-11). The `pack` method adjusts the panel size to fit the content. Finally, the `showNextRow` method is called to display the first record.

Listing 10.13: *The* `showTable` *method of the* `MainFrame` *class that displays a data record on the panel*

```
1  public void showTable(String tableName, Connection conn) {
2    try (Statement stat = conn.createStatement();
3        ResultSet result = stat.executeQuery("SELECT * FROM " +
       tableName))  {
4      RowSetFactory factory = RowSetProvider.newFactory();
5      crs = factory.createCachedRowSet();
6      crs.setTableName(tableName);
7      crs.populate(result);
8      if (scrollPane != null) remove(scrollPane);
9      dataPanel = new DBPanel(crs);
10     scrollPane = new JScrollPane(dataPanel);
```

```
11        add(scrollPane, BorderLayout.CENTER);
12        pack();
13        showNextRow();
14      }
15      catch (SQLException ex) {
16        MySQLExceptionInfo.print(ex);
17      }
18  }
```

In listing 10.14, there is the `showNextRow` method of the `MainFrame` class, which displays the next data record on the panel. At the beginning, the method checks in a conditional statement whether the object representing the data set is "null" or if the last record is already displayed on the screen. If so, the method ends. Otherwise, the `next` method is called, which moves the cursor in the data set to the next record, followed by a call to the `showRow` method from the `DBPanel` class, passing the new data record to be displayed on the panel.

Listing 10.14: *The* `showNextRow` *method of the* `MainFrame` *class displays the next data record on the panel*

```
1  public void showNextRow() {
2          try {
3              if (crs == null || crs.isLast()) return;
4              crs.next();
5              dataPanel.showRow(crs);
6          }
7          catch (SQLException ex) { MySQLExceptionInfo.print(ex); }
8  }
```

10.11. Practice exercises

1. Write a program that executes SQL commands stored in a text file. The filename is provided by the user as a command-line argument.
2. Modify the method `createTable` from listing 10.6 so that it accepts two parameters. The second parameter should be an SQL command for creating a table. Use this method to create the table `t_book` to store book information and populate it with data from listing 10.15.
3. Write an application (extension of the application from subsection 10.10) that allows the user to display the previous record, as well as navigate to the first or last record in the result set, by clicking the "First" and "Last" buttons.

4. Write an application (extension of the application from subsection 10.10) that allows users to add, delete, and edit database records.

Listing 10.15: *The* `t_book.sql` *file, containing SQL commands for filling the* `t_book` *table with sample data*

```
INSERT INTO t_book VALUES ('1', 'C.J. Date',
  'A Guide to the SQL Standard', 'Addison-Wesley' , 6.07);
INSERT INTO t_book VALUES ('2', 'B. Schneier',
  'Applied Cryptography', 'Wiley', 46.66);
INSERT INTO t_book VALUES ('3', 'C. Stoll',
  'The Cuckoos Egg', 'Pocket books', 12.69);
INSERT INTO t_book VALUES ('4', 'E. Gamma et al.',
  'Design Patterns', 'Addison-Wesley', 50.68);
INSERT INTO t_book VALUES ('5', 'T. H. Cormen et al.',
  'Introduction to Algorithms', 'The MIT Press', 107.95);
INSERT INTO t_book VALUES ('6', 'D. Flanagan',
  'JavaScript: The Definitive Guide', 'OReilly Media', 57.95);
INSERT INTO t_book VALUES ('7', 'B. W. Kernighan',
  'The C Programming Language', 'Pearson', 55.99);
INSERT INTO t_book VALUES ('8', 'B. Stroustrup',
  'The C++ Programming Language', 'Addison-Wesley', 67.77);
INSERT INTO t_book VALUES ('9', 'E. S. Raymond',
  'The Cathedral and the Bazaar', 'OReilly Media', 61.54);
INSERT INTO t_book VALUES ('10', 'D. Kahn',
  'The Codebreakers', 'Scribner', 26.93);
INSERT INTO t_book VALUES ('11', 'F. Brooks Jr.',
  'The Mythical Man-Month', 'Addison-Wesley', 37.39);
INSERT INTO t_book VALUES ('12', 'T. Kidder ',
  'The Soul of a New Machine', 'Back Bay Books', 11.39);
```

Bibliography

[1] Ferrari L., Pirozzi E., *Learn PostgreSQL - Second Edition*, Packt Publishing, 2023.

[2] Luzanov P., Rogov E., Levshin I. (translated by Mantrova L.), *POSTGRES: The First Experience*, Packt Publishing, 2023.

[3] Molinaro B., de Graaf R., *SQL Cookbook: Query Solutions and Techniques for All SQL Users, 2nd Edition*, O'Reilly Media, Inc, USA, 2021.

[4] Schönig Hans-Jürgen, *Mastering PostgreSQL 15. Advanced techniques to build and manage scalable, reliable, and fault-tolerant database applications - Fifth Edition (ebook)*, Packt Publishing, 2023.

[5] Shan J., Goldwasser M., Malik U., Johnston B., *SQL for Data Analyticd: Harness the power of SQL to extract insights for data*, 3rd Edition, Packt Publishing, 2022.

[6] Viescas J. L., Hernandez M. J., *SQL Queries for Mere Mortals: A Hands-On Guide to Data Manipulation in SQL*, Third Edition, Pearson Education, Inc, publishing as Addison Wesley, 2014.

[7] Horstmann C. S., *Core Java, Volume II – Advanced Features, 11th Edition*, Pearson Education, Inc, publishing as Prentice Hall, 2019.

[8] `https://jdbc.postgresql.org/`

[9] `https://www.postgresqltutorial.com/postgresql-jdbc/`

[10] `https://docs.oracle.com/javase/8/docs/technotes/guides/jdbc/`

Appendix A

```
DROP DATABASE IF EXISTS bookstore;
CREATE DATABASE bookstore;
\c bookstore

DROP TABLE IF EXISTS t_z_book CASCADE;
DROP TABLE IF EXISTS t_a_book CASCADE;
DROP TABLE IF EXISTS t_book CASCADE;
DROP TABLE IF EXISTS t_order CASCADE;
DROP TABLE IF EXISTS t_author CASCADE;
DROP TABLE IF EXISTS t_publisher CASCADE;
DROP TABLE IF EXISTS t_client CASCADE;
DROP TABLE IF EXISTS t_supplier CASCADE;

CREATE TABLE t_book(
  ISBN int2 PRIMARY KEY,
  title varchar(40) NOT NULL,
  publisher int2,
  year char(4) DEFAULT '2023' NOT NULL,
  binding char(9) CHECK(binding in ('paperback','hardcover')),
  supplier int2,
  price decimal(4,2),
  volume int2,
  CHECK(year ~ ('^\d{4}$')),
  CHECK(price >= 0.0),
  CHECK(volume >= 0)
);

CREATE TABLE t_author (
  author_id int2 PRIMARY KEY,
```

```
  name varchar(40),
  surname varchar(40)
);

CREATE TABLE t_a_book (
  author_id int2,
  ISBN int2
);

ALTER TABLE t_a_book
ADD CONSTRAINT PK_t_a_book PRIMARY KEY (author_id, ISBN);

ALTER TABLE t_a_book
ADD CONSTRAINT key_from_book FOREIGN KEY (ISBN)
REFERENCES t_book(ISBN) ON DELETE CASCADE;

ALTER TABLE t_a_book
ADD CONSTRAINT key_from_author FOREIGN KEY (author_id)
REFERENCES t_author(author_id) ON DELETE CASCADE ;

CREATE TABLE t_publisher (
  publisher_id int2 PRIMARY KEY,
  name varchar(10),
  address varchar(40)
);

ALTER TABLE t_book
ADD CONSTRAINT key_from_publisher FOREIGN KEY (publisher)
REFERENCES t_publisher(publisher_id) ON DELETE CASCADE ;

CREATE TABLE t_order (
  order_id int2 primary key,
  client_id int2,
  date_order DATE DEFAULT current_date,
  card int2,
  completed int2
);
```

```
CREATE TABLE t_z_book (
  order_id int2,
  isbn int2,
  shipping_date DATE DEFAULT current_date,
  volume int2
);



CREATE TABLE t_client(
  client_id int2 PRIMARY KEY,
  name varchar(30),
  surname varchar(40),
  street varchar(30),
  city varchar(30),
  voivodeship varchar(30),
  zip_code char(6) CHECK(zip_code ~ ('^\d{2}-\d{3}$')),
  telephone char(11) CHECK(telephone ~ ('^\d{3}-\d{3}-\d{3}$'))
);

ALTER TABLE t_order
ADD CONSTRAINT key_from_client FOREIGN KEY(client_id)
REFERENCES t_client(client_id) ON DELETE CASCADE;

ALTER TABLE t_z_book
ADD CONSTRAINT key_from_order FOREIGN KEY(order_id)
REFERENCES t_order(order_id) ON DELETE CASCADE;

CREATE TABLE t_supplier (
  supplier_id int2 PRIMARY KEY,
  name varchar(30),
  street varchar(30),
  city varchar(30),
  voivodeship varchar(40),
  zip_code char(6) CHECK(zip_code ~ ('^\d{2}-\d{3}$')),
  telephone char(11) CHECK(telephone ~ ('^\d{3}-\d{3}-\d{3}$'))
);

ALTER TABLE t_z_book
```

```
ADD CONSTRAINT key_z_book FOREIGN KEY(ISBN)
REFERENCES t_book(ISBN) ON DELETE CASCADE;

ALTER TABLE t_book
ADD CONSTRAINT key_from_supplier FOREIGN KEY(supplier)
REFERENCES t_supplier(suppler_id) ON DELETE CASCADE;

INSERT into t_author VALUES
(1, 'Joseph', 'Heller'), (2, 'Patrick', 'Suskind'),
(3, 'Ryszard', 'Kapusta'), (4, 'Milan', 'Kundera'),
(5, 'Piotr', 'Huelle'), (6, 'Maria', 'Heller'),
(7, 'Patrick', 'Nieznany'), (8, 'Ryszard', 'Mazowiecki'),
(9, 'Adam', 'Mickiewicz'), (10, 'Henryk', 'Sienkiewicz'),
(11, 'Jan', 'Wybicki'), (12, 'Cyprian', 'Norwid'),
(13, 'Wincenty', 'Witos'), (14, 'Milan', 'Kundera'),
(15, 'Piotr', 'Nowoczesny'), (16, 'Ryszard', 'Mazur'),
(17, 'Beata', 'Powstaniec'), (18, 'Dariusz', 'Port'),
(19, 'Wiktor', 'Porto'), (20, 'Patryk', 'Wellman'),
(21, 'Maria', 'Kuncewiczowa'), (22, 'Jan', 'Zamoyski'),
(23, 'Marian', 'Zamoyski'), (24, 'Adam', 'Zielony'),
(25, 'Henryk', 'Wolski'), (26, 'Juliusz', 'Cezar'),
(27, 'Maria', 'Konopnicka'), (28, 'Tadeusz', 'Konwicki'),
(29, 'Juliusz', 'Machulski'), (30, 'Piotr', 'Wierny');

INSERT into t_supplier VALUES
(1, 'Goniec', 'Wiejska', 'Lipsk', 'podlaskie',
'15-351', '857-444-555'),
(2, 'UPS', 'Sucha', 'Lublin', 'lubelskie',
'22-100', '325-443-685'),
(3, 'Konik', 'Miejska', 'Opole', 'opolskie',
'31-100', '428-319-726'),
(4, 'Pociag', 'Nowa', 'Gdynia', 'pomorskie',
'10-200', '648-276-394'),
(5, 'Poczta', 'Srebrna', 'Tarnobrzeg', 'podkarpackie',
'41-000', '358-256-244'),
(6, 'Stolica', 'Srebrna', 'Warszawa', 'mazowieckie',
'00-950', '328-665-813'),
(7, 'Kelner', 'Wysockiego', 'Krosno', 'podkarpackie',
```

```
'42-200', '927-367-883');

INSERT into t_publisher VALUES
(1, 'Czytelnik', '00-950 Warszawa, ul. Woronicza 17'),
(2, 'PIW', '00-134 Warszawa, ul. Matejki 13/162'),
(3, 'Znak', '00-098 Warszawa, ul Koralowa 14'),
(4, 'Helion', '81-547 Gdynia, ul. Folwarczna 6 m 238'),
(5, 'Robomatic', '50-26-606 Radom, ul. Wiejska 976'),
(6, 'Znak', '97-500 Radomsko, Szkolna 64/13');

INSERT into t_book VALUES
(1, 'Kontrabasista', 1, '1997', 'hardcover', 1, 20.4, 5),
(2, 'Mercedes Benc', 3, '2001', 'hardcover', 2, 30.0, 5),
(3, 'Tomik wierszy', 2, '1998', 'paperback', 3, 20.5, 4),
(4, 'Cesarz laleczka', 1, '1978', 'hardcover', 4, 25.5, 2),
(5, 'Lapidarium', 1, '1995', 'hardcover', 5, 35.1 , 2),
(6, 'Pan Tadeusz', 4, '1997', 'hardcover', 6, 20.0, 5),
(7, 'Potop', 5, '2001', 'hardcover', 7, 30, 5),
(8, 'Mazurek', 6, '1998', 'paperback', 1, 20.0, 4),
(9, 'Fortepian Chopina', 6, '1978', 'hardcover', 2, 25.3, 2),
(10, 'Pole dla wszystkich', 5, '1995', 'hardcover', 3, 35.0 , 2),
(11, 'Domek z kart', 4, '1997', 'hardcover', 4, 27.2, 5),
(12, 'Magiczne drzewo', 3, '2001', 'hardcover', 5, 31.3, 5),
(13, 'Zimna woda', 2, '1998', 'paperback', 6, 26.4, 4),
(14, 'Szepty nocne', 1, '1978', 'hardcover', 7, 25.7, 2),
(15, 'Nibelung', 6, '1995', 'hardcover', 1, 36.8 , 2),
(16, 'Pan Samochodzik i Vinci', 1, '2007',
'paperback', 1, 19.5, 5),
(17, 'Stare wilki', 3, '2001', 'hardcover', 2, 33.3, 5),
(18, 'Tosca', 2, '1998', 'paperback', 3, 22.2, 4),
(19, 'Winnetou na Marsie', 1, '1978', 'hardcover', 4, 27.7, 2),
(20, 'Stawka mniejsza od zera', 1, '1995', 'hardcover',
5, 34.99 , 2),
(21, 'Czterej pancerni bez psa', 4, '1997', 'hardcover', 6,
21.4, 5),
(22, 'Egzorcysta', 5, '2001', 'hardcover', 7, 30.0, 5),
(23, 'Himalaje', 6, '1998', 'paperback', 1, 29.99, 4),
(24, 'Nibylandia', 6, '1978', 'hardcover', 2, 24.9, 2),
```

```
(25, 'Ostatni Mohikanin', 5, '1995', 'hardcover', 3, 36.20 , 2),
(26, 'Ostatni most', 4, '1997', 'hardcover', 4, 28.6, 5),
(27, 'Rano', 3, '2001', 'hardcover', 5, 32.4, 5),
(28, 'Zwrot o 180 stopni', 2, '1998', 'paperback', 6, 23.6, 4),
(29, 'Samo życie', 1, '1978', 'hardcover', 7, 26.4, 2),
(30, 'Klan z Transylwanii', 6, '1995', 'hardcover', 1, 38.8 , 2),
(31, 'M jak masakra', 1, '1997', 'hardcover', 1, 25.5, 5),
(32, 'Bez szans', 3, '2001', 'hardcover', 2, 33.7, 5),
(33, 'Zakazany owoc', 2, '1998', 'paperback', 3, 15.50, 4),
(34, 'Byle do przodu', 1, '1978', 'hardcover', 4, 17.60, 2),
(35, 'Nic do stracenia', 1, '1995', 'hardcover', 5, 12.5 , 2),
(36, 'Anakonda', 4, '1997', 'hardcover', 6, 29.5, 5),
(37, 'Amok', 5, '2001', 'hardcover', 7, 36.0, 5),
(38, 'Rytualny taniec', 6, '1998', 'paperback', 1, 42.0, 4),
(39, 'Inwazja z Aldebrana', 6, '1978','hardcover', 2, 41.40, 2),
(40, 'K-11', 5, '1995', 'hardcover', 3, 33.0 , 2),
(41, 'Szklany szczyt', 4, '1997', 'hardcover', 4, 38.0, 5),
(42, 'Piknik pod Giewontem', 3, '2001', 'hardcover', 5, 27.7, 5),
(43, 'Killer', 2, '1998', 'paperback', 6, 22.0, 4),
(44, 'Madagaskar', 1, '1978', 'hardcover', 7, 28.8, 2),
(45, 'Niebo w ogniu', 6, '1995', 'hardcover', 1, 39.99 , 2);

INSERT into t_a_book VALUES
(2, 1), (5, 2), (4, 3), (3, 4), (2, 5), (9, 6), (10, 7), (11, 8),
(12, 9), (5, 10), (4, 11), (3, 12), (2, 13), (5, 14), (4, 15),
(2, 31), (5, 32), (4, 33), (15, 34), (16, 35), (17, 36), (19, 37),
(19, 38), (11, 39), (25, 40), (24, 41), (23, 42), (22, 43),
(25, 44), (24, 45), (22, 16), (25, 17), (14, 18), (13, 19),
(12, 20), (19, 21), (30, 22), (21, 23), (12, 24), (15, 25),
(14, 26), (23, 27), (22, 28), (15, 29), (14, 30);

INSERT into t_client VALUES
(1, 'Jan', 'Kowalski', 'Wiejska', 'Warszawa', 'mazowieckie',
'00-480', '624-451-526'),
(2, 'Tadeusz', 'Malinowski', 'Targowa', 'Warszawa',
'mazowieckie', '00-480', '624-421-332'),
(3, 'Krystyna', 'Torbicka', 'Krakowska', 'Warszawa',
'mazowieckie', '00-480', '624-212-111'),
```

```
(4, 'Anna', 'Marzec', 'Suraska', 'Lipsk', 'podlaskie',
'15-333', '744-314-314'),
(5, 'Adam', 'Koper', 'Lipowa', 'Lipsk', 'podlaskie',
'15-356', '756-334-373');

INSERT into t_order VALUES
(1, 1, '2007-01-10', 1,1), (2, 2, '2004-01-10', 1,0),
(3, 3, '2003-01-10', 0,0), (4, 2, '2002-01-11', 0,0),
(5, 4, '2001-01-11', 1,1), (6, 5, '2008-01-11', 1,1),
(7, 4, '2007-10-12', 1,1), (8, 4, '2010-01-12', 1,0),
(9, 3, '2011-01-12', 0,0), (10, 5, '2012-01-12', 0,0),
(11, 4, '2012-04-12', 1,1), (12, 1, '2012-05-12', 1,1);

INSERT into t_z_book VALUES
(1, 1, '2008-01-11', 1), (1, 2, '2008-01-11', 1),
(1, 3, '2008-01-11', 1), (2, 3, '2004-01-11', 1),
(2, 4, '2004-01-11', 2), (3, 1, '2003-01-11', 1),
(4, 2, '2002-01-12', 1), (4, 5, '2002-01-12', 2),
(5, 5, '2001-01-12', 1), (5, 4, '2001-01-12', 1),
(6, 15, '2008-01-12', 2), (7, 11, '2007-01-12', 1),
(8, 12, '2010-01-13', 1), (9, 13, '2011-01-13', 1),
(9, 3, '2011-01-13', 1), (9, 4, '2011-01-13', 2),
(10, 10, '2012-01-13', 1), (11, 7, '2012-04-13', 1),
(11, 5, '2012-04-13', 2), (12, 5, '2012-05-13', 1),
(12, 8, '2012-05-13', 1), (12, 9, '2012-05-13', 2);
```

Appendix B

The Java application with a graphical user interface that allows viewing selected database tables.

Listing 10.16: *Listing of a program that displays database tables*

```
package org.example;

import java.awt.*;
import java.awt.event.*;
import java.sql.*;
import java.util.*;
import javax.sql.*;
import javax.sql.rowset.*;
import javax.swing.*;

public class App {
    public static void main(String[] args) {
        EventQueue.invokeLater(() ->
        {
            var frame = new MainFrame();
            frame.setTitle("DB Viewer");
            frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            frame.setVisible(true);
        });
    }
}

class MainFrame extends JFrame {
    private JButton previousButton;
    private JButton nextButton;
    private DBPanel dataPanel;
    private JScrollPane scrollPane;
    private final JComboBox<String> tableNames;
```

```
29    private Properties properties;
30    private CachedRowSet crs;
31    private Connection connection;
32
33    public MainFrame() {
34        tableNames = new JComboBox<String>();
35        try {
36            connection = DBConnector.connect();
37            DatabaseMetaData meta = connection.getMetaData();
38            ResultSet mrs = meta.getTables(null, null, null, new
    String[]{"TABLE"});
39            try {
40                while (mrs.next())
41                    tableNames.addItem(mrs.getString(3));
42            } finally {
43                mrs.close();
44            }
45        }
46        catch (SQLException ex) {
47            MySQLExceptionInfo.print(ex);
48        }
49        tableNames.addActionListener( new ActionListener() {
50                    @Override
51                    public void actionPerformed(ActionEvent e) {
52                        showTable( (String) tableNames.
    getSelectedItem(), connection);
53                    }
54                }
55                );
56        add(tableNames, BorderLayout.NORTH);
57        addWindowListener(new WindowAdapter() {
58            public void windowClosing(WindowEvent event)
59            {
60                try {
61                    if (connection != null) connection.close();
62                }
63                catch (SQLException ex) {
64                    MySQLExceptionInfo.print(ex);
65                }
66            }
67        });
68        JPanel buttonPanel = new JPanel();
```

```
69        add(buttonPanel, BorderLayout.SOUTH);
70        previousButton = new JButton("Poprzedni");
71        previousButton.addActionListener( new ActionListener() {
72                    @Override public void actionPerformed(
    ActionEvent e) {
73                        showPreviousRow();
74                    }
75                });
76        buttonPanel.add(previousButton);
77        nextButton = new JButton("Następny");
78        nextButton.addActionListener( new ActionListener() {
79                    @Override public void actionPerformed(
    ActionEvent e) {
80                        showNextRow();
81                    }
82                });
83        buttonPanel.add(nextButton);
84        if (tableNames.getItemCount() > 0)
85            showTable(tableNames.getItemAt(0), connection);
86    }
87
88    public void showTable(String tableName, Connection conn) {
89        try (Statement stat = conn.createStatement();
90             ResultSet result = stat.executeQuery("SELECT * FROM "
     + tableName))
91        {
92            RowSetFactory factory = RowSetProvider.newFactory();
93            crs = factory.createCachedRowSet();
94            crs.setTableName(tableName);
95            crs.populate(result);
96            if (scrollPane != null) remove(scrollPane);
97            dataPanel = new DBPanel(crs);
98            scrollPane = new JScrollPane(dataPanel);
99            add(scrollPane, BorderLayout.CENTER);
100           pack();
101           showNextRow();
102       }
103       catch (SQLException ex) {
104           MySQLExceptionInfo.print(ex);
105       }
106   }
107
```

```
108     public void showPreviousRow()
109     {
110         try {
111             if (crs == null || crs.isFirst()) return;
112             crs.previous();
113             dataPanel.showRow(crs);
114         }
115         catch (SQLException ex){
116             MySQLExceptionInfo.print(ex);
117         }
118     }
119 
120     public void showNextRow() {
121         try {
122             if (crs == null || crs.isLast()) return;
123             crs.next();
124             dataPanel.showRow(crs);
125         }
126         catch (SQLException ex) {
127             MySQLExceptionInfo.print(ex);
128         }
129     }
130 }
131 
132 class DBPanel extends JPanel {
133     private final java.util.List<JTextField> fields;
134 
135     public DBPanel(RowSet rs) throws SQLException {
136         fields = new ArrayList<>();
137         setLayout(new GridBagLayout());
138         GridBagConstraints gbc = new GridBagConstraints();
139         gbc.gridwidth = 1;
140         gbc.gridheight = 1;
141         ResultSetMetaData rsmd = rs.getMetaData();
142         for (int i = 1; i <= rsmd.getColumnCount(); i++) {
143             gbc.gridy = i - 1;
144             String columnName = rsmd.getColumnLabel(i);
145             gbc.gridx = 0;
146             gbc.anchor = GridBagConstraints.EAST;
147             add(new JLabel(columnName + "  "), gbc);
148             int columnWidth = rsmd.getColumnDisplaySize(i);
149             JTextField tb = new JTextField(columnWidth);
```

```
150             tb.setEditable(false);
151             fields.add(tb);
152             gbc.gridx = 1;
153             gbc.anchor = GridBagConstraints.WEST;
154             add(tb, gbc);
155         }
156     }
157 
158     public void showRow(ResultSet rs)
159     {
160         try {
161             if (rs == null) return;
162             for (int i = 1; i <= fields.size(); i++) {
163                 String field = rs == null ? "" : rs.getString(i);
164                 JTextField tb = fields.get(i - 1);
165                 tb.setText(field);
166             }
167         }
168         catch (SQLException ex) {
169             MySQLExceptionInfo.print(ex);
170         }
171     }
172 }
173 
174 class MySQLExceptionInfo {
175     public static void print(SQLException e) {
176         for(Throwable t :e)
177             t.printStackTrace();
178     }
179 }
180 
181 class DBConnector {
182     private static String URL = "jdbc:postgresql://localhost:5432/
        test_00";
183     private static String user = "postgres";
184     private static String password = "12345";
185 
186     public static Connection connect() throws SQLException {
187         Connection connection = DriverManager.getConnection(URL,
        user, password);
188         return connection;
189     }
```

```
190     public static void close(Connection connection) throws
        SQLException{
191         connection.close();
192     }
193 }
```

The Publishing House of Jan Dlugosz University in Czestochowa
42-200 Częstochowa, al. Armii Krajowej 36A
www.ujd.edu.pl
e-mail: wydawnictwo@ujd.edu.pl