8

Introduction to Relational Databases

All data processing we did so far in this book was file based. That is, our data was stored in files, and these files were read by our data management software (e.g., R). Using this software, we processed the data in various ways, and output the resulting datasets again to files. In this workflow, files are the basic containers of our data, which we use to store it persistently and to share and disseminate it. Due to its simplicity, flexibility, and versatility, file-based data storage is used for the vast majority of social science research projects. In this chapter, we go one step further. Rather than keeping our data in simple files, we use a kind of software specifically designed for storing and processing data: a database management system (DBMS).

There exist many different types of DBMS. Our focus here will be on what is probably the most common one: a *relational* DBMS. These systems are built on the idea that all data should be contained in tables that are linked to each other. This is an idea that should be straightforward to us, since we have dealt with tables from the beginning of this book. The concept of a relational database goes back several decades. In computing, this is a long time. Still, these databases continue to be around, in different forms and flavors, which attests to the power and flexibility of the concept. So, how can these database systems improve upon the standard file-based data management workflow?

- Organizing your data in a single file is simple, but quickly becomes difficult if your data is spread out across different files. If you follow the advice on a "good" table design in Chapter 3, you will probably require several tables to store data without redundancies; for example,

https://doi.org/10.1017/9781108990424.012 Published online by Cambridge University Press

if you use data with annual estimates of the GDP for different countries, you will use one table for the variables at the country level that remain constant over time (such as the year of independence), and one table for those variables that change annually (the GDP estimates). Using a single table is not a good idea, since you would have to repeat the constant country-level variables for every annual observation. This would mean that part of your data is redundant, and it is something we should avoid. With file-based data processing, however, each table requires a new file, so your entire "database" consists of many files and becomes difficult to maintain. Relational database systems, in contrast, are designed to manage many different tables simultaneously. Each database contains all tables for a project, keeping them together in one place.

- Not only are data spread out across many files difficult to handle, but they can also become internally inconsistent. Imagine in your table with GDP estimates, you have an entry that refers to a particular country in the country table, for example, Switzerland. You would like to join the two tables to create a dataset for analysis. However, what if Switzerland is somehow missing from the countries table? With file-based data storage, there is no mechanism to ensure that tables that refer to each other are *consistent* – that is, that links from one table to another are indeed valid and point to actual data. Relational databases have different mechanisms to maintain this *relational integrity*, that is, the consistency of data across different tables as you add, delete, or update data.
- When your tables become large, the performance of data operations becomes an issue when relying on file-based data storage. Loading a small file and filtering particular observations from it is easy and fast, but takes more and more time the larger your table becomes. Relational databases are designed for fast and efficient processing of your data. Operations such as searching and updating your data are tuned for optimal performance, even if your data is so big that it cannot all be kept in the computer's memory at the same time. All this complexity is safely hidden from you as the user of a DBMS – you tell the system what you want to do with your data, and the system internally uses whatever machinery is necessary to carry out these tasks.
- Finally, collaboration between different researchers is difficult in a file-based workflow. Different people would have to exchange different versions of files, while making sure that the content of these files remains consistent (see the second point above). Imagine two

researchers trying to update data in different columns of the same table: This is almost impossible in a file-based workflow, since both would have to work on a single copy of the same file (and possibly destroy the other's changes when saving it). Database management systems are centralized in a way so that many users can access the data at the same time. The different tables in a database can be modified by different users, according to the permissions they have. Each table exists only once in a database, rather than in different copies of a file.

While these advantages of relational DBMS will become clearer in this and the next chapters, using such a system in lieu of simple data files entails a technical overhead. Ultimately, it is up to you to decide what system or workflow you use for your project. If your project is small and narrow in scope, it will be perfectly fine to keep your data in files only. However, if your project involves several interlinked tables, some of which are large, or if more than one researcher works on the data, then you should consider using a database management system. In the following section, we introduce the general setup of such a system.

8.1 DATABASE SERVERS AND CLIENTS

Many database management systems are set up in client-server architecture. That is, the DBMS runs on a computer somewhere on the Internet (which is called the "server"), and so-called "clients" connect via the network to this server, send data processing instructions, and fetch data. Figure 8.1 illustrates this graphically.

The big circle represents the database management system. In the figure, this DBMS manages just one database, but in reality, there can

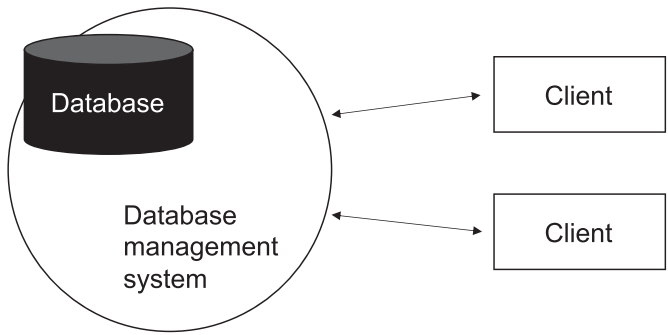

FIGURE 8.1. Interacting with a database management system.

https://doi.org/10.1017/9781108990424.012 Published online by Cambridge University Press

be many of them. The clients, depicted as rectangles, interact with the DBMS: They connect to a particular database, can upload and retrieve data from it, or send instructions for data processing. When you connect to a database server from R (as we will do in this and the following chapters), your R instance is one of those clients. R, however, is not the only client software that can communicate with a database – there are many others. For example, most DBMS come with simple text-based clients, which you can run on your local machine to send commands to the server. Also, all the major programming languages and most statistical software packages have extensions that allow you to connect to a database server.

In many cases, these clients are based on computers other than the database server, and the communication between clients and the server is done over the network. While it may seem unnecessarily complicated, this separation is actually very useful. For once, it allows the database server to be operating on powerful hardware, which is necessary in particular if you deal with large datasets and/or complex calculations. Running these operations on your local workstation or laptop would be much slower and, in many cases, even impossible. Also, the shared client-server setup is well designed for multiple users accessing a single database, which is very useful for collaborative projects.

The communication with a (relational) database server is done with a language designed for this purpose, the Structured Query Language (SQL). Some pronounce it as "Sequel," others prefer "Ess-Queue-Ell." SQL, as the idea of a "relational" database in general, has a long history, and there are many different dialects of the language. In this book, we rely on the PostgreSQL database system as well as the SQL dialect it uses. PostgreSQL is free and open source, well-known, and many other programming languages and tools can interface to it. While other relational databases such as MySQL, Oracle or Microsoft SQL Server differ in the features they offer (and therefore also the SQL dialect they understand), this book introduces some general concepts of the relational approach and SQL that apply regardless of what system you work with.

To set up a client-server structure for the purpose of this book, we have installed the PostgreSQL server on your system in Chapter 2. This server should now be running. If not, you need to go back to Chapter 2 and the online installation instructions on the book's companion website. Now is also a good time to follow the instructions in Chapter 2 to create a new database specifically for this chapter, if you have not done so already.

We use the name `dbintro` for this database, but you are of course free to choose any name you prefer.

R has a generic interface to communicate with relational databases called `DBI`, the "R Database Interface." The `RPostgres` package we use is built on this interface. There are many different types of database servers, and using this standardized interface means that you connect to any of them in the same way. For these connections, you typically need to specify at least the type of server (PostgreSQL, MySQL, etc), the name of the database, and your username and password (remember again to change the username and password to match your setup). Here is how to do this for our server and the `dbintro` database:

```
library(RPostgres)
db <- dbConnect(Postgres(),
   dbname = "dbintro",
   user = "postgres",
   password = "pgpasswd")
```

If you get an error that says something like "could not connect to server," the PostgreSQL server may not have been installed properly, or that you forgot to start it. In this case, I recommend that you go back to Chapter 2 and complete again the steps described there and in the online instructions.

Let us take a closer look at the database connection. The function to connect to the database server is `dbConnect()`. It returns a connection object, which we call `db`. We will use this object later to send all sorts of commands to the server. When we are done, we should close the connection properly (see the end of this chapter). To connect, we need to provide a few connection parameters to the function. First, this is the name of database we want to connect to, the `dbname`. Since a given server can host many different databases with different users and for different purposes, we need to specify which one we want to work with. In this chapter, this is the `dbintro` database, but we will use other databases in the following chapters. Second, you need to provide your username and password, so that the server knows who it is communicating with. This user-based authentication also allows you to later define different permissions for different users, for example, by giving some users read-only access to the database, while allowing others to modify the data. This is something we return to in Chapter 10. Note that in the above example, we omit several other connection parameters, for example, the name of the computer running the server. This is because you are running

https://doi.org/10.1017/9781108990424.012 Published online by Cambridge University Press

PostgreSQL on your local machine, using default settings. It is easy to extend the above code to connect to servers on other computers, if the need arises.

8.2 SQL BASICS

Before we use the connection to our database to send commands and work with actual data, let us cover some basics of the SQL language. As we will see below, much of it is actually close to human (English) language, so it is not too difficult to understand. Unlike many other programming languages, SQL is case-insensitive, so it does not matter if you write `SELECT * from myTable`, or `select * FrOM MYTABLE`. However, I strongly recommend that you follow the convention to spell SQL keywords in upper case, and names of tables, columns, and functions in lower case. The above statement then becomes `SELECT * FROM mytable`. I will follow this convention throughout the book.

In a relational database, all data is contained in tables, and the SQL language is designed around these tables. As social scientists, we refer to the data in our tables as *observations* or *cases*, each of which consists of different *variables*. In the database world, we prefer the terms *rows* (or *records*) and *columns* (or *fields*) of a table. Tables in relational databases have *typed* columns. This means that we need to define whether we want to stick text or numbers (or something else) into a given column, and the database system then ensures that only allowed data of the given type is stored in that column. This is similar to R's data frames (although R adjusts types dynamically, while a database does not), but very different from the non-standardized tables you can find, for example, in spreadsheets. Different database systems (and therefore, the different SQL dialects they use) vary in the column types they offer. In our discussion, we will not go into the details regarding these differences, but rather try to introduce SQL that also works beyond the PostgreSQL system we use for our exercises.

Relational databases employ a strict separation of data structure and the data itself. This is why in SQL, there are several dedicated commands to define and modify the structure of your tables: You can introduce new tables, define which columns they should consist of and what the types of these columns should be, and you can also delete columns or entire tables. This category of statements is called *data definition*. The second category of commands is for the updating of the data contained in the tables of a database: You can insert new rows or delete existing ones, or

https://doi.org/10.1017/9781108990424.012 Published online by Cambridge University Press

update the information contained in particular fields of a table. These are examples of *data manipulation* statements. Finally, we ultimately want to extract data from the tables in our database, which is why we need *data extraction* commands.

In our example below, we use R as a client to connect to the database and to send SQL statements to it. While the R code and the functions we use for this are of course specific to R, the SQL code is not – you could send the same statements via a different client, and the database would do exactly the same. However, we will also be using some convenience functions for R that facilitate, for example, the loading of data into the database. These are features offered by R (or rather, its database interface DBI) and not by SQL.

8.3 Application: Electoral Disproportionality by Country

In democracies, the main way by which institutions aggregate the preferences of citizens is through elections. In an election, citizens cast their votes, and the result of election – for example, the composition of the national parliament – is supposed to reflect the distribution of voters. However, electoral systems vary tremendously in the way they translate the votes cast in an election into a particular distribution of seats, which has long been the focus of much research in comparative politics (see e.g. Grofman and Lijphart, 1986). One outcome that is associated with the voting system is the "disproportionality" of an electoral result. Disproportionality refers to the difference between the *share of votes* that a party achieves in an election, and the *share of seats* it ultimately gets in parliament. In a majoritarian system with its winner-takes-all logic (as in the UK, for example), disproportionality between vote and seat shares will be highest, since the votes cast for candidates that do *not* win a precinct ultimately do not count. Disproportionality is typically measured using Gallagher's least squares index, which is defined as the square root of half the sum of squared differences between seat shares (S_i) and vote shares (V_i):

$$\text{LSq} = \sqrt{\frac{1}{2}\sum_{i=1}^{n}(V_i - S_i)^2}$$

What is the disproportionality in actual elections? To find out, we use data from the *ParlGov* database, a comprehensive resource with

https://doi.org/10.1017/9781108990424.012 Published online by Cambridge University Press

information on election results, political parties, and the composition of governments in EU and OECD countries (Döring and Manow, 2018). Importantly for our purpose, ParlGov consists of a set of different tables, since it is already structured internally as a relational database. In this chapter, we are going to use only one of these tables, which is the one with data on elections; in the next chapter, we will extend our work with ParlGov and add another table from the database. These tables we use are not the original ones. They have been revised slightly for the purpose of our exercises: I dropped variables we do not need and elections with missing data, kept only parliamentary elections, and retained only European countries.

8.4 CREATING A TABLE WITH NATIONAL ELECTIONS

We have a PostgreSQL database and can connect to it, but so far we do not have any tables. As we have seen above, relational databases require us to specify the structure of the data (i.e., the tables) first, before we can add data. This sounds more difficult than it actually is – all we need to do is tell PostgreSQL what the name of the new table should be, what columns it should contain, and what the types of these columns are. First, let us take a look at the election data we have from ParlGov. If you open the file `elections.csv` in a text editor, this is what you should see:

```
election_id,country_name,election_date,party_id,vote_share,seats,seats_total
402,Austria,1945-11-25,1013,49.8,85,165
402,Austria,1945-11-25,973,44.6,76,165
402,Austria,1945-11-25,769,5.4,4,165
```

The structure of the table is not difficult to understand. Each line contains an election result for a political party. Each election has its `election_id` and is linked to a country. The data also contain the `election_date`. The next three columns store the party-specific information about the election result: the party (identified by the `party_id`), the `vote_share` it achieved, and the number of seats it obtained. Finally, the last column contains the total number of seats that were filled in the respective election.

Let us now create a table structure in SQL, so that we can import the ParlGov election data. There are a few things we need for this. First, recall that we created our database connection above, and can access it via the `db` connection object. We tell R to use this connection whenever we send an SQL command to the database. The `dbExecute()` function is what we need for this. It has two parameters: First, the connection (this is our `db`

https://doi.org/10.1017/9781108990424.012 Published online by Cambridge University Press

object); second, a string with the SQL command we want to send to the database. Creating a new table in SQL is done using the `CREATE TABLE` statement:

```
dbExecute(db,
   "CREATE TABLE elections (
      election_id integer,
      country_name varchar,
      election_date date,
      party_id integer,
      vote_share real,
      seats integer,
      seats_total integer)")
```

As mentioned above, we write the SQL keywords in upper case. What exactly happens in this statement? We provide the name of the new table (`elections`), followed by the list of columns and their types. The set of columns and types needs to be enclosed in parentheses and separated by commas. In the example, we use four different types of columns:

1. Integer numbers, given as `integer`.
2. Text, given as `varchar` (a set of characters with variable length).
3. Date values, given as `date`, which will later help us order elections by calendar date as well as conduct other date-based calculations.
4. Decimal values. Here, we use `real` numbers, which is one of PostgreSQL's decimal number types.

If you need more information about the column types PostgreSQL can handle, take a quick look at the documentation at https://www.postgresql.org/docs/current/datatype.html. To reiterate our point from above: The code we present here mixes SQL code (the long text starting with `CREATE TABLE`) with R code (`dbExecute()`) to send it to the database. If you were to use a client other than R to connect to the database, you would need the SQL part, but not the R function calls. To check whether the table was successfully created, R's DBI interface has a useful function that prints out a list of all tables in a database:

```
dbListTables(db)
```

```
[1] "elections"
```

As the output shows, we currently have one table in the database, which is the `elections` table we created. Now, we have completed the definition of our table structure and can fill it with data. This is done

https://doi.org/10.1017/9781108990424.012 Published online by Cambridge University Press

using the SQL `INSERT INTO` command. To insert some data for the 1919 elections in Austria, you use the following statement:

```
dbExecute(db,
   "INSERT INTO elections
    VALUES (1030, 'Austria', '1919-02-16', 97, 40.75, 72, 170)")
```

Here we specify which table we want to add our data to, and provide in parentheses the values the respective columns should have. Note that we have to surround string values such as `Austria` with single quotation marks (`'`). The same holds for date values, which will automatically be recognized as a date if they are formatted in a standard way. The above example inserts data for *all* columns in the table, and the new values have to be provided in the exact same order in which the columns appear in the table (which is what we defined above). If we want to insert data for only specific columns, and possibly in a different order, we have to specify the columns we want to insert into – let us take an election from Belgium as an example:

```
dbExecute(db,
   "INSERT INTO elections
      (election_id, country_name, election_date, vote_share, party_id)
   VALUES (872, 'Belgium', '1908-05-24', 22.6, 2422)")
```

This omits the two fields with the party's seats and the total number of seats, and uses the party id as the last value. To check whether this worked, let us move from data manipulation to data extraction. We use another very important SQL command: `SELECT`. In the simplest form, `SELECT` can be used to retrieve all data from a table, without any filtering or transformations. Let us do this for all data we have in `elections`, which at this point are only two elections from Austria and Belgium:

```
dbGetQuery(db, "SELECT * FROM elections")
```

```
  election_id country_name election_date party_id vote_share seats seats_total
1        1030      Austria    1919-02-16       97      40.75    72         170
2         872      Belgium    1908-05-24     2422      22.60    NA          NA
```

Since this is an SQL statement, which, unlike `dbExecute()` above, *returns* data rather than just manipulating it, we need to use a different R function that fetches data from the database: `dbGetQuery()`. For simplicity, we simply output the result of this function – a data frame – to the console, but in a regular script, you would store it in a new R object for later use. The asterisk `*` in the SQL code stands for "all columns," and we need

https://doi.org/10.1017/9781108990424.012 Published online by Cambridge University Press

to provide the name of the table we want to extract from. You can see that the `seats` and `seats_total` values for Belgium are correctly stored as `NA`, since we chose not to provide them when we inserted the data for Belgium. Here, `NA` is R's convention to represent missing data. In relational databases, missing values are usually encoded as `NULL` (this is not a string, therefore no quotes), and the DBI functions take care of mapping R's NAs to NULL values in the database.

`SELECT` is probably the most powerful command in SQL, and we can cover only some variations of the above statement. Let us assume we want to see only a subset of the columns in a table. This is possible by specifying the columns names explicitly, rather than using the wildcard character `*`:

```
dbGetQuery(db, "SELECT country_name, election_date FROM elections")
```

```
  country_name election_date
1      Austria    1919-02-16
2      Belgium    1908-05-24
```

We can also extract just a subset of all data with the `WHERE` keyword:

```
dbGetQuery(db,
   "SELECT country_name, vote_share
   FROM elections
   WHERE vote_share > 40")
```

```
  country_name vote_share
1      Austria      40.75
```

It is also possible to dynamically compute new columns in a `SELECT` statement, for example, to output the vote share as a proportion rather than a percentage:

```
dbGetQuery(db,
   "SELECT country_name, vote_share / 100 AS vote_share_prop
   FROM elections")
```

```
  country_name vote_share_prop
1      Austria          0.4075
2      Belgium          0.2260
```

In this statement, we transform the vote share by dividing it by 100, and output the result in a new column called `vote_share_prop`. This new column appears only in the result – it does *not* change the original table in any way. There is much more we can do with `SELECT` statements, some of which is shown below after importing the entire ParlGov table into our database. But before we do so, we first need to remove the data we have added to our table with a `DELETE` statement:

https://doi.org/10.1017/9781108990424.012 Published online by Cambridge University Press

```
dbExecute(db, "DELETE FROM elections")
```

As with any operation that removes data, you need to be very careful: This statement deletes *all your data in the table* but keeps the table structure. Now that we have an empty table, we can insert all of the data from our election data frame into the table. Writing separate `INSERT` statements for each row would be very cumbersome. Luckily, there are several ways in which you can easily load data into a PostgreSQL table. In the example below, we use a function from R's `DBI` interface, which takes an R data frame and sends it to a database. The file `elections.csv` in the data repository contains the elections data. We load the data into R and then use `dbAppendTable()` to append it to the empty `elections` table that we created above:

```
elections <- read.csv(file.path("ch08", "elections.csv"))
dbAppendTable(db, "elections", elections)
```

As all database functions, `dbAppendTable()` first takes the database connection to be used. Second is the name of the database table that the records should be appended to. Last, we specify the data frame that should be appended to the given table. If the import is successful, you can again use the `SELECT` command to browse some of the data (the `LIMIT` keyword restricts the output to a certain number of rows). Note that in a database table, the data has no fixed ordering; the following `SELECT` statement returns two rows, but on your system, these may be different from the ones you see in the example:

```
dbGetQuery(db, "SELECT * FROM elections LIMIT 2")
```

```
  election_id country_name election_date party_id vote_share seats seats_total
1         402      Austria    1945-11-25     1013       49.8    85         165
2         402      Austria    1945-11-25      973       44.6    76         165
```

The second, and slightly more convenient way to load data is through the DBI's `dbWriteTable()` function. This function takes a data frame and a table name, and sticks the data into a given database table. If the table does not exist, it can even generate a new table structure before inserting the data. To try this, we first delete the entire table:

```
dbExecute(db, "DROP TABLE elections")
```

Now, we want to add the entire elections table to the database in a single step. Before we can do this, we need to make sure that all the columns in the data frame have the correct type. This is not the case for

https://doi.org/10.1017/9781108990424.012 Published online by Cambridge University Press

the election date, which is still a string variable. Therefore, we first convert it to a date, and then upload the entire table in one step:

```
elections$election_date <- as.Date(elections$election_date)
dbWriteTable(db, "elections", elections)
```

Again, the `dbWriteTable()` function is convenient, since it creates the new table in the database according to the structure of the data frame, and then uploads the data contained in the data frame to it. You can check again with a `SELECT` statement that the import was done successfully.

After the successful import of the `election` table, we create a new field for the year of an election with `ALTER TABLE`, to make future operations with yearly aggregations easier. In PostgreSQL, you can extract some part of a date (e.g., the year, the day, or the month) with the `extract()` function:

```
dbExecute(db, "ALTER TABLE elections ADD year integer")
dbExecute(db,
   "UPDATE elections
   SET year = extract(year from election_date)")
```

8.5 Computing Electoral Disproportionality

With our table successfully imported into our relational database, we can now proceed to compute the Gallagher index of disproportionality in SQL. We do so for each election, using data on vote shares and seat shares. Our elections table from ParlGov already contains information about each party's vote share (in the `vote_share` variable, in percent). We need a separate field for the seat share, which we can simply compute as the fraction of the actual seats for the respective party (`seats`) and the total number of seats in parliament (`seats_total`). Following the convention of separating data definition from data manipulation, we first need to create an (empty) new field for the seat share:

```
dbExecute(db, "ALTER TABLE elections ADD seat_share real")
```

We again use a `real` type for this variable, since it will contain decimal numbers. The `ALTER TABLE` command can not only be used for adding new columns, it can also delete (`DROP COLUMN`) them or change their type (`SET DATA TYPE`). Now, we can fill the new column by computing the percentage of the total seats that the party received. If we were to do so by simply dividing `seats` by `seats_total` and multiply it by 100 (to obtain a percentage), we would get the wrong result: The result of dividing two

https://doi.org/10.1017/9781108990424.012 Published online by Cambridge University Press

integer numbers in PostgreSQL is again an integer, which is why the result would have the decimal places removed. To fix this, we multiply it with a decimal number (100.0) and not an integer number (100) – as a result, PostgreSQL will carry out the computation with decimal numbers, which is what we want:

```
dbExecute(db,
  "UPDATE elections SET seat_share = 100.0 * seats / seats_total")
```

The next step is to calculate the difference between the vote share and seat share for each party in each election, square it, and then compute the sum over all these squared differences for a given election. Let us start with the first part, the squared differences between vote shares and the seat shares. We can simply include it as an additional field in a `SELECT` statement, something we have already introduced above. When performing this operation, we need to make sure to convert the ParlGov vote share from a percentage to a proportion, to make it comparable to the seat share. The `power()` function in SQL performs the exponentiation; alternatively, you could use the ^ operator for this:

```
dbGetQuery(db,
   "SELECT power(vote_share - seat_share, 2) AS squared_diffs
   FROM elections LIMIT 2")

  squared_diffs
1      2.941746
2      2.133375
```

Note that we are computing these squared differences only for illustration purposes; they are simply displayed, but not stored for later use. Next, we amend our SQL statement to compute the sum of these squared differences across all parties in an election. You recognize that what we need is simply an aggregation operation with grouping, similar to what we have done in previous chapters: We combine all squared differences with the same `election_id` and aggregate them by summing them up. In SQL, aggregation is yet another thing you can do with a `SELECT` statement: All you need to do is specify (one or more) aggregation functions, as well as the grouping levels with the `GROUP BY` keyword. We also divide the sum of squared differences by two, and take the square root:

```
dbGetQuery(db,
   "SELECT
      election_id,
      sqrt(0.5 * sum(power(vote_share - seat_share, 2))) AS lsq_index
   FROM elections
```

https://doi.org/10.1017/9781108990424.012 Published online by Cambridge University Press

```
   GROUP BY election_id LIMIT 2")

  election_id lsq_index
1         828  2.621663
2         938  2.389561
```

In the first part of the SELECT statement, we define what we would like to get out: First, this is the grouping variable election_id itself, so that we know which election a result refers to. Second, this is our above computation of the squared differences, but wrapped in the sum() function. This is the aggregation function that the database applies to each group as defined by the grouping variable. This sum is then multiplied with 0.5, and the square root function is applied to it. As above, we use the FROM keyword to tell the database which table to use for this calculation. Finally, we need to define the grouping variable using the GROUP BY keyword (the LIMIT keyword again limits the output to two rows, which is simply for presentation purposes).

8.6 Results: Electoral Disproportionality by Country

We are almost ready to create a graph with the index values by country. To do this, we make two adjustments to our previous SQL statement. First, we include the country name in the grouping, so that we know which country an election occurred in. This does not change our result, since a particular election is always linked to exactly one country. Second, we use only those elections from our table that were held after World War II. This is done by specifying a filter condition with the WHERE keyword we have used above:

```
dbGetQuery(db,
   "SELECT
       election_id, country_name,
       sqrt(0.5 * sum(power(vote_share - seat_share, 2))) AS lsq_index
   FROM elections
   WHERE year >= 1946
   GROUP BY election_id, country_name LIMIT 2")

  election_id country_name lsq_index
1         429       Norway  4.062591
2         466       Greece  6.958688
```

This is the data that we need to generate our plot. For each country and each election, Figure 8.2 shows the disproportionality scores that were computed above. You can clearly see considerable differences

https://doi.org/10.1017/9781108990424.012 Published online by Cambridge University Press

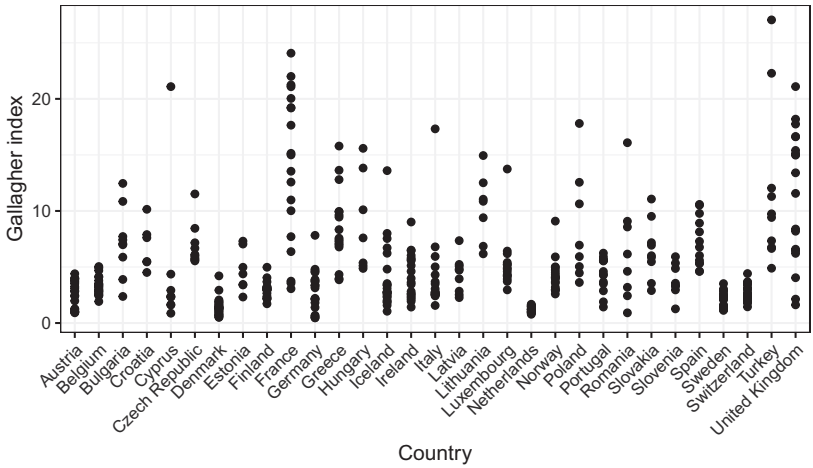

FIGURE 8.2. Gallagher index values for elections in different countries.

between countries when it comes to the disproportionality in the different elections. For example, some countries such as Sweden consistently have highly proportional outcomes, which means that the composition of the national parliament matches the distribution of votes across the different parties very well. This is not the case for other countries, however. The United Kingdom, for example, has several elections with high values of the Gallagher index. This is not surprising, given the electoral system in the UK: The country has a majority voting system, where in each district, only the candidate who gets most of the votes in that district wins. As a result, the votes for other parties are not reflected in the composition of the UK parliament.

Finally, once we are done working with our database, we need to close our database connection with:

```
dbDisconnect(db)
```

8.7 SUMMARY AND OUTLOOK

Database systems offer an alternative to standard, file-based data storage that is predominant in the social sciences. Databases are systems designed not just for data storage, but also for easy and efficient data manipulation and retrieval, possibly by several users in collaboration. They are often set up in a client-server fashion, where a central server keeps the data that

https://doi.org/10.1017/9781108990424.012 Published online by Cambridge University Press

can be accessed from different clients over the network. In this chapter, we introduced a relational database system, which is the most frequently used type of database. At the core of a relational database is the table as the main structure to store data. The interaction with a relational database happens via the Structured Query Language, SQL.

We covered three different types of SQL commands. *Data definition* statements such as `CREATE TABLE` help us set up the structure of our database by defining the tables and their columns. *Data manipulation* statements allow us to populate our tables with data, and to change and delete it, and *data extraction* commands retrieve data from our database for further use (e.g., in R). In this chapter, we started our exploration into the world of database using a single table only. This is obviously too limited, which is why we will add more tables in the next chapter.

Besides the focus on SQL, there are several lessons you can take away from this chapter.

- *Client-server setups are useful for many applications:* As explained in the chapter, the client-server setup we use with PostgreSQL allows you to outsource particular tasks to other machines, and R simply interacts with these servers as a client. This is useful for many other applications beyond PostgreSQL, for example, other types of database servers, or servers executing large computing tasks. It is now possible to obtain access to these servers, for example, through universities, which means that you do not have to manage such a system yourself.
- *Recognize the difference between SQL and R:* Obviously, R and SQL are designed for different tasks, but there is an important difference in the philosophy underlying these languages. In R, you give the R engine a precise set of instructions on what it should do. This is called *procedural* programming. In SQL, you say what you want as a result, but not how to get there. This is called *declarative* programming. Declarative programming is convenient for us, since we do not have to worry about handling large amounts of data on a disk – the database system does this for us.
- *Distinguish between R's built-in database functions and SQL:* We have seen that R offers a number of convenient database functions that make your work easier, such as `dbWriteTable()`. These functions internally generate SQL code, which is then executed by the server. Of course, while this may be convenient for you, it also gives you less control over these operations. For example, you can bypass the explicit step of creating a table, and let `dbWriteTable()` do all the work. For this, it

https://doi.org/10.1017/9781108990424.012 Published online by Cambridge University Press

is important to check the result in the database, for example, whether your automatically created table has the correct structure.

- *Different functions for sending commands and getting data:* It is important to remember the difference between data extraction (with SELECT) and the other operations that can be performed on the server. A SELECT statement returns data, the other statements do not. This is why there are two different functions in R for these different types of statements. `dbGetQuery()` is used only for data extraction, and requires an SQL statement that returns data (typically, SELECT). All other kinds of operation are done with `dbExecute()`.

https://doi.org/10.1017/9781108990424.012 Published online by Cambridge University Press