9

Relational Databases and Multiple Tables

In the previous chapter of this book, I gave an introduction to a relational database system and the SQL language that we use to interact with it. We defined a new table, populated it with data, and extracted and aggregated the information contained in it. For illustration purposes, this introduction used a single table only; however, as I emphasized repeatedly, the power of relational databases lies in their ability to manage many different, interlinked tables simultaneously. This is why in this chapter, we are adding more tables to our database.

At this point, let us quickly go through the motivation again for distributing data across multiple tables. In Chapter 3, we discussed good and bad designs: Ideally, you should set up your tables such that they avoid data redundancy – each piece of information should be stored only *once* in the database. In our example about *elections* and the *parties* participating in these elections, how could redundancy possibly occur? Imagine for a moment that we were to store elections and parties in one table:

country_name	election_date	vote_share	party_name_short	family_name
Austria	1919-02-16	40.75	SPÖ	Social dem.
Austria	1920-10-17	35.99	SPÖ	Social dem.
Austria	1923-10-21	39.60	SPÖ	Social dem.

The first three columns in this table contain information about election results: The country they are held in, the date, and the vote share of the given party. The remaining two columns contain the party information: The short name, as well as the party family. This short example shows that we have redundant data: The short name and the party family are

https://doi.org/10.1017/9781108990424.013 Published online by Cambridge University Press

repeated every time a party – in our case, the Austrian Social Democrats (SPÖ) – participates in an election. This is why the ParlGov project splits up their entire database into multiple tables. By separating data on election results from the data on political parties, we can reduce redundancy in the database. This is what our above example looks like in the actual ParlGov database: We have one table on election results (which is the one we used in the previous chapter):

country_name	election_date	vote_share	party_id
Austria	1919-02-16	40.75	973
Austria	1920-10-17	35.99	973
Austria	1923-10-21	39.60	973

and a second one on political parties (all shortened for presentational purposes):

party_id	party_name_short	family_name
973	SPÖ	Social dem.

By storing the party information in a separate table, we end up with *one* record for each party, rather than repeating this information for every election the party participates in. If we want to add new variables for parties (e.g., whether they have been coded as populist), we can do this by updating *one* row for each party. This facilitates the management of your data significantly and reduces errors. The above example also shows how we can link entries across tables: The parties results table has a `party_id` column, which we use in the elections table to identify the party that the given result belongs to. The use of these references is crucial, since we deal with different tables whose entries are linked to each other. In the world of relational databases, we often use integer numbers for this purpose. A unique identifier for a record in a table – such as `party_id` in the parties table – is called a *primary key*. A reference in a table that points to a record in a different table – such as `party_id` in the elections table – is called a *foreign key*. Much of the work we do below deals with these keys.

9.1 APPLICATION: THE RISE OF POPULISM IN EUROPE

In this chapter, we continue our work with election results, but extend it in a new direction. Over the recent decade, the Western world – and Europe in particular – has seen a strong rise in populism. Cas Mudde

https://doi.org/10.1017/9781108990424.013 Published online by Cambridge University Press

defines populism as a political discourse or even an ideology based on the “relationship between the people (good) and the elite (bad)” (Mudde, 2004). In this chapter, we want to track the rise of populism over time. Specifically, we do this by measuring the electoral success of political parties that have been defined as “populist.” In this example, we do not differentiate between different types of populism, as for example, right- and left-wing populism – readers that are interested in only one or the other can easily modify the example.

For this exercise, we need two tables in addition to the election results we used in the previous chapter. So far, we only used data on elections from *ParlGov* to compute a Gallagher index of disproportionality. In the elections table, however, parties are only referenced with an internal identifier (the `party_id`), which is why we need to bring in a separate table on political *parties* to obtain the names of the parties as well as other information about them. Since our goal is to measure the success of populism by the vote share of populist parties, we need to know whether a party is considered a populist party or not. For this, we rely on the PopuList database, a list of populist parties in Europe (Rooduijn et al., 2019). As of Version 2.0, the PopuList dataset can easily be linked to parties from ParlGov: Each party in the PopuList has a `parlgov_id`, which corresponds to the `party_id` in ParlGov. Combining data from ParlGov and the PopuList ultimately allows us to track the success of populist parties over time in parliamentary elections.

9.2 ADDING THE TABLES

Let us now do some practical work to see tables and the references between them in action. Do not forget to create a new database, following the instructions in Chapter 2. We use the `dbadvanced` database for this chapter and connect to it:

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

We first add the elections table from the previous chapter, using the corresponding function from R’s `DBI` interface: `dbWriteTable()`. This function simplifies the import, since it automatically creates the table structure for us. This is convenient, but you have to make sure that the column types in the initial R data frame have the correct types, as they will be used to

https://doi.org/10.1017/9781108990424.013 Published online by Cambridge University Press

specify the columns in the database table (see the previous chapter). We also add the `year` again as a separate column:

```r
elections <- read.csv(file.path("ch09", "elections.csv"))
elections$election_date <- as.Date(elections$election_date)
dbWriteTable(db, "elections", elections)
dbExecute(db,
   "ALTER TABLE elections ADD COLUMN year integer")
dbExecute(db,
   "UPDATE elections SET year = extract(year from election_date)")
```

Our next step is to add the ParlGov table with political parties to our database, using again the functionality provided by R's `DBI` extension:

```r
parties <- read.csv(file.path("ch09", "parties.csv"))
dbWriteTable(db, "parties", parties)
```

Since ParlGov does not provide information about whether a party is considered populist or not, we rely on the PopuList data described above. Before we can later merge this data to our `parties` table, we need to also import it as a table, using the file `populist.csv` in the repository for this chapter:

```r
populist <- read.csv(file.path("ch09", "populist.csv"))
dbWriteTable(db, "populist", populist)
```

You should now have three tables in your database: the `elections` table from the previous chapter, and the `parties` and `populist` tables that we just created. Let us check if this is the case:

```r
dbListTables(db)

[1] "elections" "parties"   "populist"
```

The structure of the `parties` table should be obvious. Most importantly, as already mentioned above, each party has a `party_id`, which corresponds to the `party_id` in the `elections` table and helps us link each election result to the party it belongs to. This is similar for the PopuList table (or rather, the reduced version I have prepared for this chapter), where each party has a `parlgov_id`, along with information on whether it qualifies as a "populist" party according to the PopuList dataset, and whether it is considered to be a party on the far left or the far right:

https://doi.org/10.1017/9781108990424.013 Published online by Cambridge University Press

```
dbGetQuery(db, "SELECT * FROM populist LIMIT 3")

  parlgov_id populist farleft farright
1       1536        1       0        1
2         50        1       0        1
3        669        1       0        0
```

9.3 JOINING THE TABLES

Before we work with all three tables, let me demonstrate the linking of tables using the two tables from ParlGov only. To briefly repeat, we have a table with data on election results (`elections`), and a `parties` table with data on parties. Each entry in `elections` refers to a party from `parties` by means of a party identifier, called `party_id` in both tables. In the database world, this is called a "one-to-many" relationship between the two tables, since each party belongs to several election results – it usually participated in several elections. The combination of two tables that contain corresponding data is called a "join." Joining two tables is a temporary operation – in contrast to a merge operation, we do not end up with a new, persistent table that contains the linked records. Rather, a join creates a *temporary* dataset with the corresponding records, which we can use for further data operations, or export for later analysis. The *storage* of our data, however, is still done in separate tables, which helps us avoid redundant data in our database.

So, how do we join tables in SQL? Again, we use a `SELECT` statement for this. All we need to change is the `FROM` part of the statement, such that it does not select from a single table, but from a set of two joined tables. This is indicated by the `JOIN` keyword:

```
dbGetQuery(db,
   "SELECT
      elections.country_name, election_date, party_name_short, family_name
   FROM elections JOIN parties ON elections.party_id = parties.party_id
   LIMIT 3")

  country_name election_date party_name_short         family_name
1      Denmark    1915-05-07               RV             Liberal
2      Denmark    1953-09-22             GrFa           no family
3       Greece    1977-11-20              EDA Communist/Socialist
```

It is not difficult to understand what this statement does: `elections` should be joined to `parties`, by linking entries where the `party_id` in `elections` (which is a foreign key) corresponds to the `party_id` in `parties` (which is a primary key). It is not a requirement that the join attributes in the two tables have the same name, but we often follow this convention to make the relationship more obvious. The variables we select – the country name, the election date, etc. – are specified in the first part of the `SELECT` statement. Since `country_name` appears both in the elections and the parties table, we need to tell SQL which one we want, by specifying the name of the table before the name of the field (`elections.country_name`).

The type of join that is carried out with the simple `JOIN` keyword is called an *inner* join – in fact, you could write `INNER JOIN` instead and get the exact same result. An inner join links all pairs of entries from the two tables that have the same value in the join attribute. That is exactly what we want in the vast majority of cases. Although much less frequently used, there are other types of joins that retain *all* records from one of the tables, but only the matching records from the other (the `LEFT JOIN` and the `RIGHT JOIN`). Even though the join of the two tables is only temporary, we can use it in the `SELECT` statement as if it were a new, big table. For example, we can count the number of records:

```
dbGetQuery(db,
   "SELECT count(*)
   FROM elections JOIN parties ON elections.party_id = parties.party_id")

  count
1  5247
```

Alternatively, we can run aggregations on it. Here is an example that makes use of the `party_family` variable contained in ParlGov: We compute the average vote share of social democratic parties per year, to see the ups and downs in their electoral success:

```
dbGetQuery(db,
   "SELECT year, avg(vote_share)
   FROM elections JOIN parties ON elections.party_id = parties.party_id
   WHERE family_name = 'Social democracy'
   GROUP BY year
   ORDER BY year
   LIMIT 3")

  year     avg
1 1900 12.7500
2 1901 17.0600
3 1902  9.4025
```

In this statement, you recognize all the different parts of a data aggregation, as introduced in the previous chapter: the grouping variable `year`

https://doi.org/10.1017/9781108990424.013 Published online by Cambridge University Press

(computed by extracting the year from the election date), and the aggregation function (the average over the `vote_share` values for a given year). Importantly, we filter out the social democratic parties with the `WHERE` keyword, since these are the parties we are interested in. Finally, we order the result by year, and truncate it for display purposes using the `LIMIT` keyword – if you would like to see the entire time series, just remove this last part of the statement.

9.4 MERGING DATA FROM THE POPULIST

In the previous section, we joined the elections and the parties tables. Joining means that the two tables are dynamically combined within a query, while the original data remains in separate tables. Is this what we should also do when linking parties from ParlGov with data on populist parties from the PopuList? We could do a simple join on the party identifier:

```
test <- dbGetQuery(db,
   "SELECT *
   FROM parties JOIN populist ON parties.party_id = populist.parlgov_id")
nrow(test)

[1] 199
```

As per the logic of an inner join, we only get the matching records from both tables – this is why the result of the join contains only 199 entries, which is a small subset of the almost 1,300 parties from ParlGov. It is easy to see why: Unlike ParlGov, which goes back more than a century, the PopuList covers only recent years. Also, it identifies only populist and eurosceptic parties, which is why it contains only a subset of recent parties.

Our parties table and the data from the PopuList are coded at exactly the same level – both contain information about political parties as unit of observation. In other words, the relationship between the two is a *one-to-one* relationship rather than the *one-to-many* relationship we have for parties and elections. While it is technically possible to use SQL joins whenever we want to combine information from two tables, in this case it may be more useful to merge the variables from the PopuList to our `parties` table. Again, merging means that we amend the `parties` table, such that it *persistently* stores the additional variables from the PopuList. We can then simply access the information about whether a party is considered as populist in the `parties` table, rather than having to join it with `populist` every time.

https://doi.org/10.1017/9781108990424.013 Published online by Cambridge University Press

To merge the PopuList coding to the existing `parties` table, we first add a new column:

```
dbExecute(db, "ALTER TABLE parties ADD COLUMN populist integer")
```

The default value of this new column is `NULL` (the SQL value for missing data). We then use an amended version of an `UPDATE` statement, which uses a second table to update the values in the given table. More precisely, it links the two tables similar to a join, and copies the values of the `populist` variable from the `populist_parties` table to the `parties` table:

```
dbExecute(db,
   "UPDATE parties
   SET populist = populist.populist
   FROM populist
   WHERE parties.party_id = populist.parlgov_id")
```

Again, the logic of this statement is not difficult to understand. We update the `parties` table and want to set the values of the `populist` field to the corresponding ones from the `populist_parties` table. In the `WHERE` clause, we need to specify – similar to the join above – what attributes the two tables should be linked on. Importantly, this updates the values only for the parties contained in the PopuList data, because these are the only ones that can be matched. For all other parties, the default values (missing, or in the database terminology: `NULL`) remain.

With the new variable `populist` now being part of our table with political parties, we can modify the above aggregation query such that it counts, for example, the number of populist parties per year that participated in elections:

```
dbGetQuery(db,
   "SELECT year, count(*) AS num_parties
   FROM elections JOIN parties ON elections.party_id = parties.party_id
   WHERE populist = 1 AND year >= 1998
   GROUP BY year
   ORDER BY year DESC
   LIMIT 5")

  year num_parties
1 2017          19
2 2016          18
3 2015          23
4 2014          13
5 2013          18
```

This statement is very similar to the one above, where we computed the average vote share of social democratic parties by year. We change the aggregation function to output the count of elections, join the two tables as above, and restrict the combined result to parties that are populist (`populist = 1`) and elections in 1998 and later, since this is the first year for which there is data from the PopuList.

9.5 MAINTAINING REFERENTIAL INTEGRITY

When introducing relational databases, we discussed some of their advantages for data management and processing. One of them was that databases can help us avoid data redundancy, but at the same time ensure that our data remains consistent. For example, by splitting up the data on election results and the parties participating in these elections, we can avoid that information on parties is repeatedly stored every time a party participates in an election. Splitting data into several tables may be useful for eliminating data redundancy, but at the same time creates other problems. As we have seen above, every row in the `elections` table has a pointer to the corresponding row in the parties table. This is implemented by means of an integer number – `party_id` in `elections` points to the corresponding `party_id` in `parties`. The latter is a primary key in the `parties` table – a field that uniquely identifies an entry. The former is a foreign key in the `elections` table – a field that references an entry in another table.

Problems can now arise if the pointer to the entry in the other table is invalid – in our example, this would mean that we have a row with `party_id = 1556` in `elections`, but no corresponding entry with `party_id = 1556` in the `parties` table. In other words, we would have an election result for a party that does not exist in our database, and our data would therefore be inconsistent. In database terminology, this is called a violation of *referential integrity*. Referential integrity applies if every reference between tables is valid, that is, if it points to an existing entry in the respective table. Of course, we want referential integrity at all times, since otherwise we would have major gaps in our data – in this case, an election result we cannot link to a party. How can we ensure that errors of this kind do not arise?

At the moment, the database does nothing to help us address this challenge. We could, for example, delete any party from the `parties` table, leaving a number of invalid foreign keys in the `elections` table. Why? The reason is that our database does not "know" yet that one field in one table references entries in another table. Let us proceed step by step to define this relationship in SQL. First, we need to introduce `party_id` as a primary

https://doi.org/10.1017/9781108990424.013 Published online by Cambridge University Press

key in the parties table. Again, a primary key is a field (or in some case, a combination of two or more fields) that uniquely identifies each line in the table. It is common practice to use positive integer values for this – luckily, we already have such a field in our table and only need to define it as primary key. We do this using the ALTER TABLE statement again, but this time without adding a new field:

```
dbExecute(db, "ALTER TABLE parties ADD PRIMARY KEY (party_id)")
```

When we define a primary key, the database does different things. Most importantly, it introduces logical checks, for example, by ensuring that no single value of the primary key occurs more than once. For example, try adding a new record with 1739 as the value for the primary key:

```
dbExecute(db,
   "INSERT INTO parties (party_id, party_name_short)
   VALUES (1739, 'New Party')")
```

This value already exists in the table, which is why PostgreSQL refuses to add the new entry. We get an error message telling us that the value 1739 already exists as a primary key.

Rather than using an existing field as primary key, you can also have the database create and maintain one for you. Simply add a new field of the type serial, and you will get an integer variable that automatically increments when new records are added to the table (you do not need to provide values for it). If you define this field as primary key, you never have to worry about duplicate key values anymore.

We now have a primary key for the parties table, and PostgreSQL ensures that the key does what it is supposed to do: uniquely identify parties in our database. The second step to have the database check and maintain referential integrity of our data is to define the party_id field in elections as foreign key. We again use an ALTER TABLE statement to do this:

```
dbExecute(db,
   "ALTER TABLE elections
   ADD FOREIGN KEY (party_id) REFERENCES parties (party_id)")
```

Using this statement, we tell the database that party_id in elections points to party_id in parties. This means that all party IDs used in

https://doi.org/10.1017/9781108990424.013 Published online by Cambridge University Press

the elections table must be present somewhere in the parties table. Since PostgreSQL created the foreign key without any error messages, we know that this is the case. However, once we attempt to delete a party from parties, the database blocks this operation if this party is referenced from elections. Try this statement:

```
dbExecute(db, "DELETE FROM parties WHERE party_id = 1739")
```

Now, the database refuses to delete party 1739, since this would leave some election results without a corresponding party. So in essence, by specifying in our database which attributes are primary keys and foreign keys, the database helps us maintain the consistency of our data and ensures that referential integrity is not violated. Using these mechanisms, distributing data over multiple tables becomes much more manageable.

9.6 RESULTS: THE RISE OF POPULISM IN EUROPE

We can finally put our data together and create a dataset for our analysis of the rise of populism in Europe over time. In the following code example, we again join the parties and elections tables, the latter now amended with the PopuList coding. We aggregate the joined tables by country and election date, which allows us to plot the success of populist parties per country, as measured by the vote share in the respective election:

```
populism_ds <- dbGetQuery(db,
   "SELECT
      elections.country_name,
      election_date,
      sum(vote_share) AS total_vote_share
   FROM elections JOIN parties USING (party_id)
   WHERE populist = 1 AND year >= 1998
   GROUP BY elections.country_name, election_date
   ORDER BY country_name, election_date")
```

The plot in Figure 9.1 shows that in particular in Eastern Europe, populist parties have been gaining ground in the recent decade. In several countries, they now achieve vote shares of up to 50% and more.

As a last step, we need to close the connection to our database:

```
dbDisconnect(db)
```

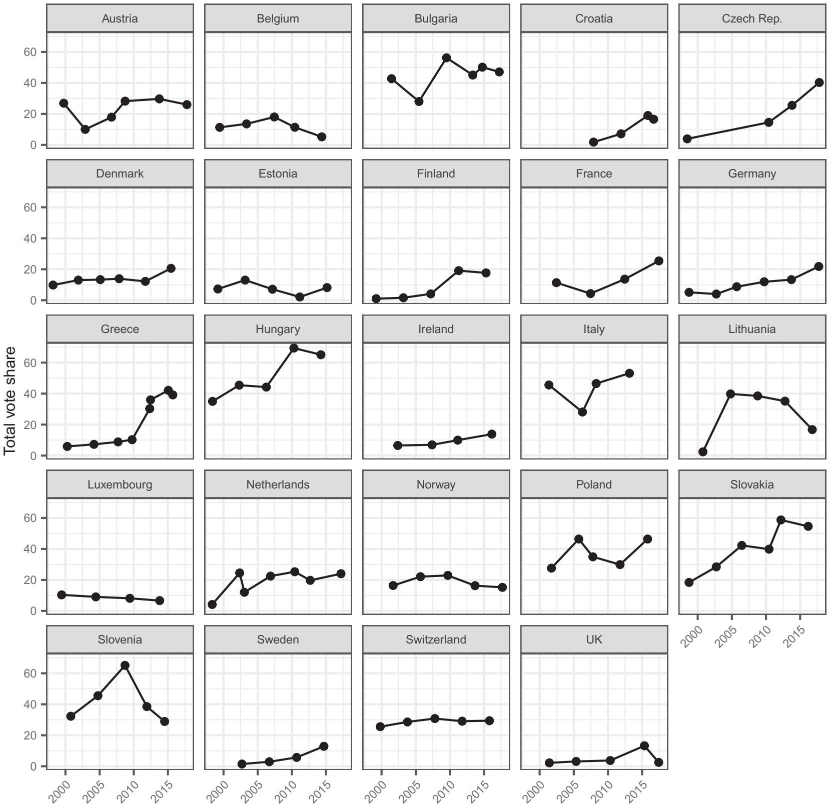

FIGURE 9.1. Vote shares of populist parties in different countries.

9.7 SUMMARY AND OUTLOOK

The art of working with relational databases necessarily involves multiple tables. In this chapter, we extended the single-table example from the previous chapter such that it uses two tables. More precisely, we added a second table with data about political parties to the existing elections table, such that we have more information about the parties themselves. We supplemented the latter table with data from the PopuList project, which identifies populist parties in Europe. Using our data, we were able to plot the electoral gains of populist parties in Europe over the recent years.

Spreading information out over several tables in a relational database involves different challenges. First, we need to think about the structure of our data: What tables do we need, and what variables are they supposed to contain? These are conceptual questions about our database, and they relate directly to what we discussed in earlier chapters of this book (e.g., designing a database such that it avoids storing redundant data). When we use existing datasets, we often do not have a choice and have to use the data in the way it is provided to us. However, when designing databases for our own projects, taking some time to think about the data structure is important. Database designers have even developed an entire modeling approach for this purpose, which is based on the definition of real-world *entities* and the *relationships* between them. These "Entity-Relationship" models can then be used to define the actual tables in a relational database. For most applications in the social sciences, however, this conceptual step is not required, as the complexity of the data is limited.

Also, there are technical challenges we need to overcome when working with multiple tables. The first we discussed is the dynamic combination of data from different tables. While stored across multiple tables, matching entries from them can be joined in SQL to perform various tasks such as aggregation, or can be exported for analysis. Importantly, joins are dynamic, and the original data are still kept in their original tables. The second challenge the database can solve for us is to keep our data consistent across different tables. For example, if a table has a *foreign key* that refers to a *primary key* in another table, the database can make sure that corresponding entries for the latter exist in the second table. This way, we can automatically ensure *referential integrity* of the database and prevent operations that would violate it.

While we now know a lot about databases already, we still need to explore two more features that can be really useful for our work: the ability for multiple contributors to jointly work on datasets, and to quickly search large amounts of data. The next chapter addresses these two questions, and wraps up the basic introduction of relational databases in this book. Before we proceed, here are some recommendations from this chapter:

- *Think about the structure of your data:* This came up repeatedly in the book, and here it is again. Choosing a good structure for your database first requires a good understanding of what is in your data: What real-world entities are described, and what are their features? How do these entities relate to each other? Once you have answered these questions, it becomes easier to design a structure for your data.
- *All tables need a primary key:* For a smooth operation of a relational database, it is absolutely necessary to have sensible primary keys for

https://doi.org/10.1017/9781108990424.013 Published online by Cambridge University Press

all your tables. A good choice is a single integer number. Some datasets already have a primary key, for others you can easily create one in your database with a `serial` field.

- *Make use of the integrity checks in a DB:* In the chapter, we saw how PostgreSQL can help you maintain referential integrity and make sure that the data is consistent across tables. I recommend using these features, in particular when your database becomes more complex. Without these checks, errors and missing data can occur without you noticing.
- *Merge only when you have to:* While joins are the standard operation to combine data from different tables in a relational database, it is also possible to merge tables by copying data from one to the other. This is something you should only do when it is really necessary, since it can violate the principle of avoiding redundant data.

https://doi.org/10.1017/9781108990424.013 Published online by Cambridge University Press