10

Database Fine-Tuning

In Chapters 8 and 9, we covered the basics of relational databases – tables as the main containers of data, and how they can be created, populated, and joined with SQL. In these chapters, we dealt exclusively with conceptual questions about data and how it is stored in a relational database. In this final chapter on databases, we move on to two more operational questions. Recall that data structure (facilitating the use of multiple tables, avoiding redundancy) was only one of the reasons for storing data in a database. There are at least two more reasons that can make databases such as PostgreSQL a useful choice for social science projects. First, databases can handle large datasets much more efficiently than a file-based workflow, and second, databases permit data processing shared by multiple users, such that it is possible to give some users write access to certain parts of the data, while others can only read it.

Most database systems do not solve these issues automatically. Rather, they require some fine-tuning by the user, but fortunately, none of this is very complicated. To show how database systems such as PostgreSQL deal with these challenges, we cover two topics in this chapter. First, we discuss the use of so-called search *indexes* that allow the database to quickly look up particular entries in a table based on one or more of the fields. Indexes are not only used in relational databases; rather, they are data structures that you can find in many systems dealing with large amounts of data (although they are often hidden from the user). Second, we introduce PostgreSQL's multi-user capabilities, where you can add several users to a database and equip them with particular privileges for data access and data manipulation.

https://doi.org/10.1017/9781108990424.014 Published online by Cambridge University Press

Person No.	City
18665	3
16889	8
17443	9
14662	8
16881	8

FIGURE 10.1. A table without an index.

In this chapter, we are not going to work with another real-world example to demonstrate indexes and multi-user features. Rather, we will use an artificial dataset, which makes it possible for us to create large tables without the need to import them from a file.

10.1 SPEEDING UP DATA ACCESS WITH INDEXES

An index is an additional data structure added to your database that allows the database system to quickly locate records in a given table, based on one or more of the fields in the table. You can think of database index very much like the index of a book: A book's index is essentially a list of important keywords and topics covered in the book, and it provides you with the page number(s) that contain relevant information about the respective topic. Here is an example that briefly illustrates how an index works. In Figure 10.1, you can see a table with data on persons located in three cities (3, 8, and 9). Now imagine that you want to select all persons in a particular city, for example, those in city 8. Without a search index, the database system needs to go through all records in the table sequentially, test whether the city attribute is equal to 8, and retain those where this condition is satisfied.

This would of course be a very fast operation for our small table, due to the fact that it only has five rows. However, as the size of the table grows, so does the retrieval time: In computer science, this time is typically measured in relation to the number of rows in the table. Our simple, non-indexed table requires retrieval times that scale *linearly* with the number of records: If we double the number of records, the retrieval time doubles, too. This becomes a real issue when we deal with large tables and require many repeated lookups. Luckily, we can solve this issue with the help of indexes. A search index is created to speed up the retrieval of records based on a particular search attribute. Figure 10.2 shows our example again, but with a search index on the `city` field added.

https://doi.org/10.1017/9781108990424.014 Published online by Cambridge University Press

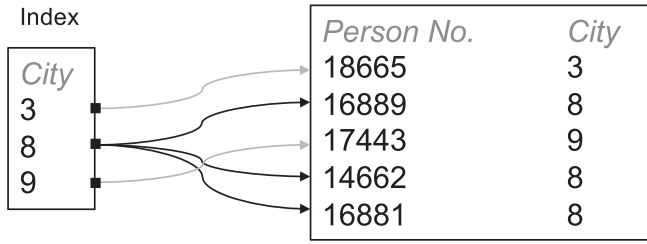

FIGURE 10.2. A table with an index.

The index stores pointers to the different values in the search attribute. So if we want to look up the persons living in city 8, all the database system needs to do is locate the value 8 in the index, and follow the pointers to the three records available for city 8. This is much faster compared to the simple traversal of the entire table as in our above example: The retrieval time using an index is usually *logarithmic* in the number of records in the table. This is a tremendous speedup: A table with 5 million entries would require 5 million steps for a naïve, sequential search, compared to less than 25 steps for a search using an index (binary logarithm).

This speedup, however, has certain costs. An index is an additional data structure that needs to be stored somewhere, so the size of your database on disk will become larger (which is something that, in most cases, you can simply ignore since the gains for retrieval are significant). An issue that may be more relevant arises when we insert new records into the table (or when we update the indexed column for some of the records): With an index in place, simply adding the new record to the table is not enough; the database system also needs to update the index such that it contains a pointer to the new record. If you insert many records, or update the existing ones frequently, this will become slower in comparison to a non-indexed table. In a typical workflow in the social sciences, however, this is less likely to happen. Therefore, you can usually create an index *after* all the data has been inserted, which avoids this problem.

Thankfully, indexing functionality for different kinds of data is readily built into PostgreSQL (and many other database systems), so as a user you do not have to worry about any of the inner workings. To demonstrate the speedup we can gain from an index, let us perform a little experiment. We create a large table both in R and in a relational database, and measure how long it takes for a certain subset of the table to be retrieved. For this experiment, we slightly expand the above example. We create a table with persons that are located in cities, and are observed annually

https://doi.org/10.1017/9781108990424.014 Published online by Cambridge University Press

(e.g., in a longitudinal survey). The `expand.grid()` function is useful here, which creates a data.frame out of all possible combinations of the values in the given vectors. We add a randomly generated `result` variable to the data frame, which holds the value measured for the respective person in the respective year. Finally, we randomly shuffle the order of the entries in our table, to exclude any effects from our data having been inserted in a particular ordered sequence.

```
survey <- expand.grid(person = 1:100, city = 1:1000, year = 1970:2020)
survey$result <- runif(nrow(survey))
survey <- survey[sample(nrow(survey)), ]
```

We can now simulate a simple data retrieval operation from our table, where we extract all records for city 80 and the years 2000 and later. When we do this, we measure the time the system takes to carry out this task, by computing the difference between the system time immediately before the data retrieval and immediately after. In the following code chunk, we enclose the three lines of code in curly brackets, such that they are executed immediately after one another:

```
{start_time <- Sys.time()
nrow(subset(survey, city == 80 & year >= 2000))
extime_R <- Sys.time() - start_time}
```

The time used for extracting the relevant records is 52.09 milliseconds. On your system, this value will be different, since execution times depend on a multitude of factors; however, the precise value is not important since we are interested in relative differences between R and PostgreSQL. Now, let us do the same experiment in a relational database. As always, we create a new, blank database for this chapter (see Chapter 2 for instructions), and connect to it:

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

We again create our artificial dataset, this time using an SQL statement. The `generate_series()` function in PostgreSQL makes the generation of the numeric sequences easy. All possible combinations of these values are generated by referencing them in the `FROM` part of the statement. Again, we add a `result` value randomly:

https://doi.org/10.1017/9781108990424.014 Published online by Cambridge University Press

```
dbExecute(db,
   "CREATE TABLE survey AS
      (SELECT *, random() AS result FROM
         generate_series(1,100) AS person,
         generate_series(1,1000) AS city,
         generate_series(1970, 2020) AS year)")
```

This table has exactly the same content and size as the data frame we used above. At present, it is just a simple table in the database, without any indexes to facilitate data retrieval. This is how we perform the same query as above in SQL, again measuring the execution time (note again the curly brackets):

```
{start_time <- Sys.time()
dbGetQuery(db,
   "SELECT count(*) FROM survey
   WHERE city = 80 AND year >= 2000")
extime_pg1 <- Sys.time() - start_time}
```

Without an index, the search in our `survey` table takes longer than the one in the R data frame: 196.99 milliseconds. Clearly, without an index, R's basic data structures perform even better than a relational database. Does an index solve this problem? Adding an index to a table is easy. All you need is a `CREATE INDEX` statement, where you specify the table and the column that should be indexed. As a rule of thumb, it is advisable to index those columns that are typically used to retrieve records from the table, and to create a separate index for each of them. In our case, these are the `city` and `year` columns in the `survey` table. If you have a primary key in the table and it has been explicitly defined as such, there is no need to create an index, since PostgreSQL does this automatically:

```
dbExecute(db, "CREATE INDEX ON survey (city)")
dbExecute(db, "CREATE INDEX ON survey (year)")
```

If we now run our query again with the same statement as above:

```
{start_time <- Sys.time()
dbGetQuery(db,
   "SELECT count(*) FROM survey
   WHERE city = 80 AND year >= 2000")
extime_pg2 <- Sys.time() - start_time}
```

we see a considerable performance improvement. Now, the query only takes 1.79 milliseconds, which is much faster than the previous SQL query on the non-indexed table (by a factor of about 110), but also about

https://doi.org/10.1017/9781108990424.014 Published online by Cambridge University Press

29 times faster than R's data frame. So if your data processing includes large datasets with many repeated data retrievals, it is absolutely essential that you properly index your data. If you do not need the additional features of a database system and use a purely file-based data workflow, you can consider using alternatives to R's basic data frames. For example, the `data.table` package provides a tabular data structure that can be indexed, and has a much better retrieval performance, in particular for large tables.

10.2 COLLABORATIVE DATA MANAGEMENT WITH MULTIPLE USERS

One of the main benefits of managing data in a centralized, server-based setting is the possibility for many users to access the database. Imagine a situation in which a team of researchers collaboratively works on a new dataset (thereby actively modifying it), and another group of researchers prepare initial analyses on this dataset (with read-only access to the data). In a file-based workflow, the second team of researchers could be provided with regular snapshots of the data, shared as files. However, it is difficult to collaboratively edit and update data in a team of contributors if the data is stored solely in files. The reason is that changes by one person can easily be overwritten by another person, similar to what happens if several people edit a single text document at the same time.

Relational database systems have fine-grained mechanisms of access control, which makes it possible to fine-tune read and write privileges for many users of a database. These features can be useful to resolve issues arising in collaborative scenarios such as the one above, but also many others. In this section, I present a brief introduction to user privilege management in SQL. We continue to use our `survey` table created above, but will simulate access to this table by two users. The first one of them uses the connection we have initiated above, with the standard username and password. This user is a "super-user," owns the database `dbtuning` and can make any modification in it. The connection object we have created is `db`, which we will continue to use. For our exercise, however, we also need a second user. Therefore, our super-user first needs to create this second user in the database system:

```
dbExecute(db, "CREATE USER other WITH ENCRYPTED PASSWORD 'pgpasswd1'")
```

Note that we create the user through the `db` connection, which is the super-user connection with the privilege to add and modify users. The new

https://doi.org/10.1017/9781108990424.014 Published online by Cambridge University Press

user is called other, and we create this user with an encrypted password and not a plain text one (WITH ENCRYPTED PASSWORD). In PostgreSQL, users are defined at the level of the entire database server, not at the level of individual databases. Therefore, the new user is in principle available for all databases hosted on our server. At this point, however, there is not much this user can do, because no access privileges to databases and tables have been defined. Still, we can already connect to the server as the new user. We do so with a new connection object db1 that we will use for all operations that user other will perform later.

```
db1 <- dbConnect(Postgres(),
   dbname = "dbtuning",
   user = "other",
   password = "pgpasswd1")
```

We are connected to the dbtuning database, so everything we send over the db1 connection will be executed within this database. Let us try to select a few rows from the table:

```
dbGetQuery(db1,
   "SELECT avg(result) FROM survey
   WHERE city = 80 AND year = 2000")
```

As expected, this fails with an error message. The reason is simply that user other does not have any privileges for the database, which means that the user can neither read nor modify any data in it. Our super-user can enable this. The following statement (executed as user postgres through the db connection) allows user other to perform SELECT queries on the survey table:

```
dbExecute(db, "GRANT SELECT ON survey TO other")
```

Now, the user can successfully execute the above query:

```
dbGetQuery(db1,
   "SELECT avg(result) FROM survey
   WHERE city = 80 AND year = 2000")

        avg
1 0.5139814
```

In some cases, it may be necessary to let users update the data in a table. We can do this by granting update privileges for the entire table, but it is even possible to do this for individual columns only. Let us assume that the new user is supposed to update the survey results. We allow this by

https://doi.org/10.1017/9781108990424.014 Published online by Cambridge University Press

providing the user with the right to update the table, but only a specific column. In the case of our survey, we can use this functionality to let other users change the measured value in the `result` column, but not the basic structure of the survey with person IDs, cities, and years:

```
dbExecute(db, "GRANT UPDATE (result) ON survey TO other")
```

Now, the user can change the results, for example to correct a wrong entry

```
dbExecute(db1,
  "UPDATE survey SET result = 0.789
  WHERE person = 3 AND city = 80 AND year = 2000")
```

but the user cannot update data in the other columns of the table, which is exactly what we want. The following command fails with an error message:

```
dbExecute(db1, "UPDATE survey SET year = year + 1 WHERE year = 2000")
```

We only granted the user the privilege to update some data, but other manipulations (such as dropping some records) are not possible. If we want our new user to update the table or delete records from it, we need to expand the user's privileges. You can either specify these new privileges explicitly (e.g., `GRANT UPDATE`), or simply give the user all privileges on the `survey` table.

```
dbExecute(db, "GRANT ALL PRIVILEGES ON survey TO other")
```

Finally, we also show how to remove certain privileges after they have been granted. This is done with a `REVOKE` statement, which works similar to the `GRANT` statement used above. Again, you can specify the privileges you would like to drop explicitly, or simply revoke them all:

```
dbExecute(db, "REVOKE ALL PRIVILEGES ON survey FROM other")
```

As always, at the end of the chapter, we close our database connections:

```
dbDisconnect(db)
dbDisconnect(db1)
```

The examples above gave you an idea of how to fine-tune user access to the tables in your database. Access control works at the level of tables. By default, users (that do not own the table) have no access to a table. You can change this by granting read-only access to the table, or allow

https://doi.org/10.1017/9781108990424.014 Published online by Cambridge University Press

users to make changes to the data in the table (or only certain columns). These features allow for databases to be used in a collaborative setting, where multiple researchers jointly access a single database remotely.

10.3 SUMMARY AND OUTLOOK

Most of our discussion about databases in the previous chapters centered around questions of data structure and content. In this chapter, we looked into operational questions arising in the work with relational database systems. Databases shield a lot of technical complexity from users; all you need to do is define your tables and populate them with data, and the database system takes care of saving this data in a physical storage. While PostgreSQL and other systems have a lot of internal mechanisms to process the data as efficiently as possible, some fine-tuning may be necessary, depending on the context in which you use the database. I showed above how the retrieval of data in large tables can be sped up by several orders of magnitude through the use of indexes. Since these indexes also have certain costs (e.g., they make data insertions slower), the database system does not create them automatically, which is why you need to do this yourself using the respective SQL statements.

A second topic we covered in this chapter is the multi-user features of PostgreSQL. Due to the client-server setup of most database systems, it becomes possible for your data to be accessed by different people and from different places, something that is difficult and error-prone if your data is stored in files. However, collaborative access means that you need to think about who should get access to the data in the first place, and what the other users can do with the data: Are they supposed to only have read access, or can they even make modifications to it? The user privileges in PostgreSQL allow you to define this. You can create new users, and grant them permission to select data from a particular table, or modify (change and delete) it. These features make databases a powerful and convenient tool for storing and processing research data. Overall, there is a number of lessons learned in this chapter:

- *Index your large datasets:* When your datasets grow large, it is essential to work with indexes to speed up search and retrieval. As mentioned above, indexing features are not just available in relational databases; instead, this is a general technique to handle large amounts of data efficiently. You can also equip individual tables in R with an index using the `data.table` package, but of course without the other advantages offered by relational databases.

https://doi.org/10.1017/9781108990424.014 Published online by Cambridge University Press

- *User privileges allow for transparent data access:* Larger projects in the social sciences can easily involve several collaborators accessing a database. The user privilege system provides a way to regulate access to your data, with different access levels for different people. In particular when it comes to sensitive data, it is essential to define who can see, update, or delete what kinds of data in your project.
- *Tracking changes remains difficult:* In many projects (e.g., those that involve human coding), it is often desirable to track changes to the data made by users. This is difficult, regardless of whether your data is stored in files or in a database. One way to do this is to keep regular copies (snapshots) of your data. Alternatively, in PostgreSQL, you can use the `pgaudit` extension, which logs all database operations to a logfile.
- *Direct access to the DB:* Sometimes, it is useful to quickly browse a database, for example, to check whether the data was imported correctly. For this purpose, you can use a graphical database client, such as the free *pgAdmin* tool (https://www.pgadmin.org), or the commercial, but highly recommended *Postico* software (https://eggerapps.at/postico/). Using these tools, you can browse a database, look at some records from a table, or even make small updates manually.

https://doi.org/10.1017/9781108990424.014 Published online by Cambridge University Press