13

Accessing Databases

This chapter introduces relational databases as a way to structure and organize complex data sets. After introducing the purpose and format of relational databases, it describes the syntax for interacting with them using R. By the end of the chapter you will be able to wrangle data from a database.

13.1 An Overview of Relational Databases

Simple data sets can be stored and loaded from `.csv` files, and are readily represented in the computer's memory as a data frame. This structure works great for data that is structured just as a set of observations made up of features. However, as data sets become more complex, you run against some limitations.

In particular, your data may not be structured in a way that it can easily and efficiently be represented as a single data frame. For example, imagine you were trying to organize information about music playlists (e.g., on a service such as *Spotify*). If your playlist is the unit of analysis you are interested in, each playlist would be an observation (row) and would have different features (columns) included. One such feature you could be interested in is the songs that appear on the playlist (implying that one of your columns should be `songs`). However, playlists may have *lots* of different songs, and you may also be tracking further information about each song (e.g., the artist, the genre, the length of the song). Thus you could not easily represent each song as a simple data type such as a number or string. Moreover, because the same song may appear in multiple playlists, such a data set would include a lot of duplicate information (e.g., the title and artist of the song).

To solve this problem, you could use multiple data frames (perhaps loaded from multiple `.csv` files), joining those data frames together as described in Chapter 11 to ask questions of the data. However, that solution would require you to manage multiple different `.csv` files, as well as to determine an effective and consistent way of joining them together. Since organizing, tracking, and updating multiple `.csv` files can be difficult, many large data sets are instead stored in **databases**. Metonymically, a database is a specialized application (called a database management system) used to save, organize, and access information—similar to what `git` does for versions of code, but in this case for the kind of data that might be found in multiple `.csv` files. Because many organizations store their data in a database of some kind, you will need to be able to access that data to analyze it.

Moreover, accessing data directly from a database makes it possible to process data sets that are too large to fit into your computer's memory (RAM) at once. The computer will not be required to hold a reference to all the data at once, but instead will be able to apply your data manipulation (e.g., selecting and filtering the data) to the data stored on a computer's hard drive.

13.1.1 What Is a Relational Database?

The most commonly used type of database is a **relational database**. A relational database organizes data into **tables** similar in concept and structure to a data frame. In a table, each **row** (also called a **record**) represents a single "item" or observation, while each **column** (also called a **field**) represents an individual data property of that item. In this way, a database table mirrors an R data frame; you can think of them as somewhat equivalent. However, a relational database may be made up of dozens, if not hundreds or even thousands, of different tables—each one representing a different facet of the data. For example, one table may store information about which music playlists are in the database, another may store information about the individual songs, another may store information about the artists, and so on.

What makes relational databases special is how they specify the *relationships* between these tables. In particular, each record (row) in a table is given a field (column) called the **primary key**. The primary key is a unique value for each row in the table, so it lets you refer to a particular record. Thus even if there were two songs with the same name and artist, you could still distinguish between them by referencing them through their primary key. Primary keys can be any unique identifier, but they are almost always numbers and are frequently automatically generated and assigned by the database. Note that databases can't just use the "row number" as the primary key, because records may be added, removed, or reordered—which means a record won't always be at the same index!

Moreover, each record in one table may be *associated* with a record in another—for example, each record in a `songs` table might have an associated record in the `artists` table indicating which artist performed the song. Because each record in the `artists` table has a unique key, the `songs` table is able to establish this association by including a field (column) for each record that contains the corresponding key from `artists` (see Figure 13.1). This is known as a **foreign key** (it is the key from a "foreign" or other table). Foreign keys allow you to join tables together, similar to how you would with `dplyr`. You can think of foreign keys as a formalized way of defining a consistent column for the `join()` function's `by` argument.

Databases can use tables with foreign keys to organize data into complex structures; indeed, a database may have a table that *just* contains foreign keys to link together other tables! For example, if a database needs to represent data such that each playlist can have multiple songs, and songs can be on many playlists (a "many-to-many" relationship), you could introduce a new "bridge table" (e.g., `playlists_songs`) whose records represent the associations between the two other tables (see Figure 13.2). You can think of this as a "table of lines to draw between the other tables." The database could then join *all three* of the tables to access the information about all of the songs for a particular playlist.

table: artists JOIN songs ON artists.id = songs.artist_id

artists.id	artists.name	songs.id	songs.title	songs.artist_id
10	David Bowie	83	Starman	10
11	Queen	80	Bohemian Rhapsody	11
11	Queen	81	Don't Stop Me Now	11
12	Prince	82	Purple Rain	12

Figure 13.1 An example pair of database tables (top). Each table has a *primary key* column `id`. The `songs` table (top right) also has an `artist_id` *foreign key* used to associate it with the `artists` table (top left). The bottom table illustrates how the foreign key can be used when joining the tables.

> **Going Further**: Database design, development, and use is actually its own (very rich) problem domain. The broader question of making databases reliable and efficient is beyond the scope of this book.

13.1.2 Setting Up a Relational Database

To use a relational database on your own computer (e.g., for experimenting or testing your analysis), you will need to install a separate software program to manage that database. This program is called a **relational database management system** (**RDMS**). There are a couple of different popular RDMS systems; each of them provides roughly the same syntax (called SQL) for manipulating the tables in the database, though each may support additional specialized features. The most popular RDMSs are described here. You are not required to install any of these RDMSs to work with a database through R; see Section 13.3, below. However, brief installation notes are provided for your reference.

- **SQLite**[1] is the simplest SQL database system, and so is most commonly used for testing and development (though rarely in real-world "production" systems). SQLite databases have the advantage of being highly self-contained: each SQLite database is a single file (with the

[1] **SQLite:** https://www.sqlite.org/index.html

table: playlists

id	name
100	Awesome Mix
101	Sweet Tunes

table: songs

id	title
80	Bohemian Rhapsody
81	Don't Stop Me Now
82	Purple Rain
83	Starman

table: playlists_songs

playlist_id	songs_id
100	81
100	82
100	83
101	80
101	82

table: playlist JOIN playlists_songs ON playlist.id = playlist_id JOIN songs ON songs.id = songs_id

playlist_id	playlists.name	songs.id	songs.title
100	Awesome Mix	81	Don't Stop Me Now
100	Awesome Mix	82	Purple Rain
100	Awesome Mix	83	Starman
101	Sweet Tunes	80	Bohemian Rhapsody
101	Sweet Tunes	82	Purple Rain

Figure 13.2 An example "bridge table" (top right) used to associate many playlists with many songs. The bottom table illustrates how these three tables might be joined.

`.sqlite` extension) that is formatted to enable the SQLite RDMS to access and manipulate its data. You can almost think of these files as advanced, efficient versions of `.csv` files that can hold multiple tables! Because the database is stored in a single file, this makes it easy to share databases with others or even place one under version control.

To work with an SQLite database you can download and install a command line application[2] for manipulating the data. Alternatively, you can use an application such as *DB Browser for SQLite*,[3] which provides a graphical interface for interacting with the data. This is particularly useful for testing and verifying your SQL and R code.

- **PostgreSQL**[4] (often shortened to "Postgres") is a free open source RDMS, providing a more robust system and set of features (e.g., for speeding up data access and ensuring data integrity) and functions than SQLite. It is often used in real-world production systems, and is the recommended system to use if you need a "full database." Unlike with SQLite, a Postgres database is not isolated to a single file that can easily be shared, though there are ways to export a database.

[2] **SQLite download page:** https://www.sqlite.org/download.html; look for "Precompiled Binaries" for your system.
[3] **DB Browser for SQLite:** http://sqlitebrowser.org
[4] **PostgreSQL:** https://www.postgresql.org

You can download and install the Postgres RDMS from its website;[5] follow the instructions in the installation wizard to set up the database system. This application will install the manager on your machine, as well as provide you with a graphical application (*pgAdmin*) to administer your databases. You can also use the provided `psql` command line application if you add it to your PATH; alternatively, the *SQL Shell* application will open the command line interface directly.

- **MySQL**[6] is a free (but closed source) RDMS, providing a similar level of features and structure as Postgres. MySQL is a more popular system than Postgres, so its use is more common, but can be somewhat more difficult to install and set up.

  If you wish to set up and use a MySQL database, we recommend that you install the Community Server Edition from the MySQL website.[7] Note that you do *not* need to sign up for an account (click the smaller "No thanks, just start my download" link instead).

We suggest you use SQLite when you're just experimenting with a database (as it requires the least amount of setup), and recommend Postgres if you need something more full-featured.

13.2 A Taste of SQL

The reason all of the RDMSs described in Section 13.1.2 have "SQL" in their names is because they all use the same syntax—SQL—for manipulating the data stored in the database. **SQL** (**S**tructured **Q**uery **L**anguage) is a programming language used specifically for managing data in a relational database—a language that is *structured* for *querying* (accessing) that information. SQL provides a relatively small set of commands (referred to as **statements**), each of which is used to interact with a database (similar to the operations described in the *Grammar of Data Manipulation* used by `dplyr`).

This section introduces the most basic of SQL statements: the `SELECT` statement used to access data. Note that it is absolutely possible to access and manipulate a database through `R` without using SQL; see Section 13.3. However, it is often useful to understand the underlying commands that `R` is issuing. Moreover, if you eventually need to discuss database manipulations with someone else, this language will provide some common ground.

Caution: Most RDMSs support SQL, though systems often use slightly different "flavors" of SQL. For example, data types may be named differently, or different RDMSs may support additional functions or features.

Tip: For a more thorough introduction to SQL, *w3schools*[a] offers a very newbie-friendly tutorial on SQL syntax and usage. You can also find more information in Forta, *Sams Teach Yourself SQL in 10 Minutes, Fourth Edition* (Sams, 2013), and van der Lans, *Introduction to SQL, Fourth Edition* (Addison-Wesley, 2007).

[a] https://www.w3schools.com/sql/default.asp

[5] **PostgreSQL download page:** https://www.postgresql.org/download
[6] **MySQL:** https://www.mysql.com
[7] **MySQL download Page:** https://dev.mysql.com/downloads/mysql

The most commonly used SQL statement is the **SELECT** statement. The `SELECT` statement is used to access and extract data from a database (without modifying that data)—this makes it a **query** statement. It performs the same work as the `select()` function in `dplyr`. In its simplest form, the `SELECT` statement has the following format:

```
/* A generic SELECT query statement for accessing data */
SELECT column FROM table
```

(In SQL, comments are written on their own line surrounded by `/* */`.)

This query will return the data from the specified column in the specified table (keywords like `SELECT` in SQL are usually written in all-capital letters—though they are not case-sensitive—while column and table names are often lowercase). For example, the following statement would return all of the data from the `title` column of the `songs` table (as shown in Figure 13.3):

```
/* Access the `title` column from the `songs` table */
SELECT title FROM songs
```

This would be equivalent to `select(songs, title)` when using `dplyr`.

You can select multiple columns by separating the names with commas (`,`). For example, to select both the `id` and `title` columns from the `songs` table, you would use the following query:

```
/* Access the `id` and `title` columns from the `songs` table */
SELECT id, title FROM songs
```

If you wish to select *all* the columns, you can use the special `*` symbol to represent "everything"—the same wildcard symbol you use on the command line! The following query will return all columns from the `songs` table:

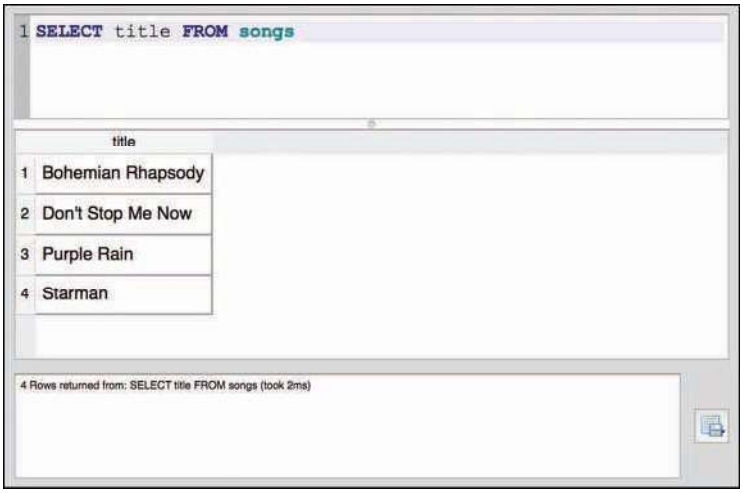


Figure 13.3 A `SELECT` statement and results shown in the SQLite Browser.

```
/* Access all columns from the `songs` table */
SELECT * FROM songs
```

Using the * wildcard to select data is common practice when you just want to load the entire table from the database.

You can also optionally give the resulting column a new name (similar to a **mutate** manipulation) by using the **AS** keyword. This keyword is placed immediately after the name of the column to be aliased, followed by the new column name. It doesn't actually change the table, just the label of the resulting "subtable" returned by the query.

```
/* Access the `id` column (calling it `song_id`) from the `songs` table */
SELECT id AS song_id FROM songs
```

The SELECT statement performs a select data manipulation. To perform a filter manipulation, you add a **WHERE** clause at the end of the SELECT statement. This clause includes the keyword WHERE followed by a *condition*, similar to the boolean expression you would use with dplyr. For example, to select the title column from the songs table with an artist_id value of 11, you would use the following query (also shown in Figure 13.4):

```
/* Access the `title` column from the `songs` table if `artist_id` is 11 */
SELECT title FROM songs WHERE artist_id = 11
```

This would be the equivalent to the following dplyr statement:

```
# Filter for the rows with a particular `artist_id`, and then select
# the `title` column
filter(songs, artist_id == 11)  %>%
  select(title)
```

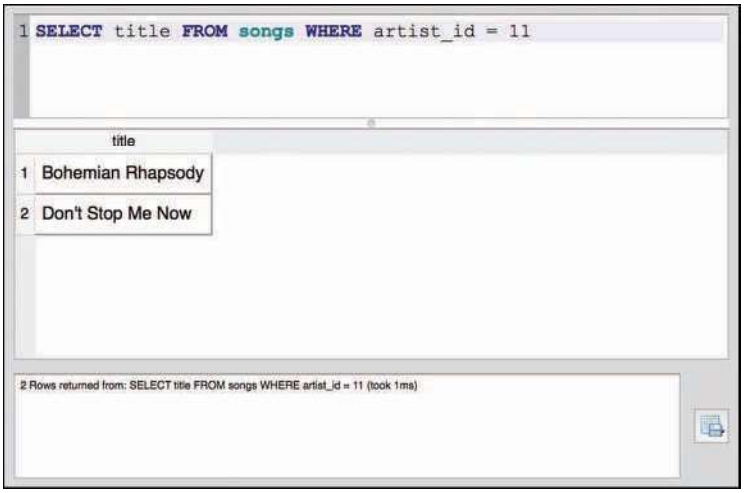


Figure 13.4 A WHERE clause and results shown in the SQLite Browser.

The filter condition is applied to the whole table, not just the selected columns. In SQL, the filtering occurs before the selection.

Note that a `WHERE` condition uses **`=`** (not `==`) as the "is equal" operator. Conditions can also use other relational operators (e.g., `>`, `<=`), as well as some special keywords such as `LIKE`, which will check whether the column value is *inside* a string. (String values in SQL must be specified in quotation marks—it's most common to use single quotes.)

You can combine multiple `WHERE` conditions by using the **`AND`**, **`OR`**, and **`NOT`** keywords as boolean operators:

```
/* Access all columns from `songs` where EITHER condition applies */
SELECT * FROM songs WHERE artist_id = 12 OR title = 'Starman'
```

The statement `SELECT columns FROM table WHERE conditions` is the most common form of SQL query. But you can also include other keyword clauses to perform further data manipulations. For example, you can include an **`ORDER_BY`** clause to perform an **arrange** manipulation (by a specified column), or a **`GROUP_BY`** clause to perform aggregation (typically used with SQL-specific aggregation functions such as `MAX()` or `MIN()`). See the official documentation for your database system (e.g., for Postgres[8]) for further details on the many options available when specifying `SELECT` queries.

The `SELECT` statements described so far all access data in a single table. However, the entire point of using a database is to be able to store and query data across *multiple* tables. To do this, you use a **join** manipulation similar to that used in `dplyr`. In SQL, a join is specified by including a **`JOIN`** clause, which has the following format:

```
/* A generic JOIN between two tables */
SELECT columns FROM table1 JOIN table2
```

As with `dplyr`, an SQL join will by default "match" columns if they have the same value in the same column. However, tables in databases often don't have the same column names, or the shared column name doesn't refer to the same value—for example, the `id` column in `artists` is for the artist ID, while the `id` column in `songs` is for the song ID. Thus you will almost always include an **`ON`** clause to specify which columns should be matched to perform the join (writing the names of the columns separated by an **`=`** operator):

```
/* Access artists, song titles, and ID values from two JOINed tables */
SELECT artists.id, artists.name, songs.id, songs.title FROM artists
    JOIN songs ON songs.artist_id = artists.id
```

This query (shown in Figure 13.5) will select the IDs, names, and titles from the `artists` and `songs` tables by matching to the *foreign key* (`artist_id`); the `JOIN` clause appears on its own line just for readability. To distinguish between columns with the same name from different tables, you specify each column first by its table name, followed by a period (`.`), followed by the column name. (The dot can be read like "apostrophe s" in English, so `artists.id` would be "the `artists` table's `id`.")

[8] **PostgreSQL: SELECT:** https://www.postgresql.org/docs/current/static/sql-select.html

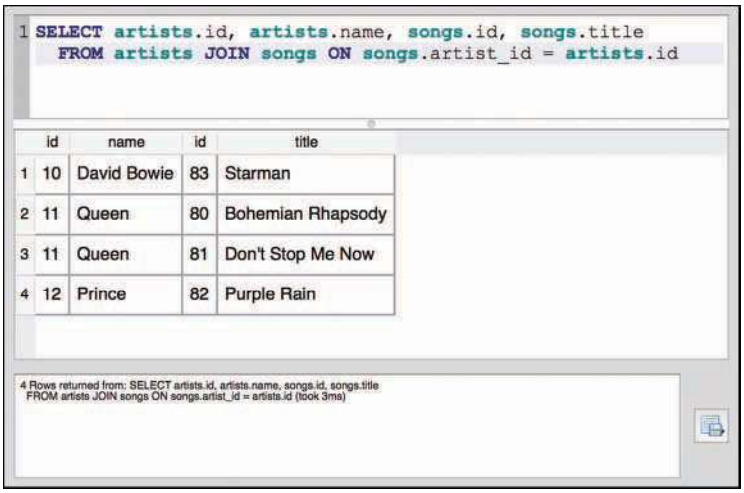


Figure 13.5 A `JOIN` statement and results shown in the SQLite Browser.

You can join on multiple conditions by combining them with `AND` clauses, as with multiple `WHERE` conditions.

Like `dplyr`, SQL supports four kinds of joins (see Chapter 11 to review them). By default, the `JOIN` statement will perform an *inner join*—meaning that only rows that contain matches in both tables will be returned (e.g., the joined table will not have rows that don't match). You can also make this explicit by specifying the join clause with the keywords **`INNER JOIN`**. Alternatively, you can specify that you want to perform a **`LEFT JOIN`**, **`RIGHT JOIN`**, or **`OUTER JOIN`** (i.e., a *full join*). For example, to perform a *left join* you would use a query such as the following:

```
/* Access artists and song titles, including artists without any songs */
SELECT artists.id, artists.name, songs.id, songs.title FROM artists
    LEFT JOIN songs ON songs.artist_id = artists.id
```

Notice that the statement is written the same way as before, except with an extra word to clarify the type of join.

As with `dplyr`, deciding on the type of join to use requires that you carefully consider which observations (rows) must be included, and which features (columns) must *not* be missing in the table you produce. Most commonly you are interested in an inner join, which is why that is the default!

13.3 Accessing a Database from R

SQL will allow you to query data from a database; however, you would have to execute such commands through the RDMS itself (which provides an interpreter able to understand the syntax). Luckily, you can instead use `R` packages to connect to and query a database directly, allowing you to

use the same, familiar R syntax and data structures (i.e., data frames) to work with databases. The simplest way to access a database through R is to use the **dbplyr**[9] package, which was developed as part of the `tidyverse` collection. This package allows you to query a relational database using `dplyr` functions, avoiding the need to use an external application!

> **Going Further**: RStudio also provides an interface and documentation for connecting to a database through the IDE; see the *Databases Using R* portal.[a]
>
> [a] https://db.rstudio.com

Because `dbplyr` is another external package (like `dplyr` and `tidyr`), you will need to install it before you can use it. However, because `dbplyr` is actually a "backend" for `dplyr` (it provides the behind-the-scenes code that `dplyr` uses to work with a database), you actually need to use functions from `dplyr` and so load in the `dplyr` package instead. However, you will also need to load the `DBI` package, which is installed along with `dbplyr` and allows you to connect to the database:

```
install.packages("dbplyr") # once per machine
library("DBI")             # in each relevant script
library("dplyr")           # need dplyr to use its functions on the database!
```

You will also need to install an additional package depending on which kind of database you wish to access. These packages provide a common *interface* (set of functions) across multiple database formats—they will allow you to access an SQLite database and a Postgres database using the same R functions.

```
# To access an SQLite database
install.packages("RSQLite") # once per machine
library("RSQLite")          # in each relevant script

# To access a Postgres database
install.packages("RPostgreSQL") # once per machine
library("RPostgreSQL")          # in each relevant script
```

Remember that databases are managed and accessed through an RDMS, which is a separate program from the R interpreter. Thus, to access databases through R, you will need to "connect" to that external RDMS program and use R to issue statements *through* it. You can connect to an external database using the **`dbConnect()`** function provided by the `DBI` package:

```
# Create a "connection" to the RDMS
db_connection <- dbConnect(SQLite(), dbname = "path/to/database.sqlite")

# When finished using the database, be sure to disconnect as well!
dbDisconnect(db_connection)
```

The `dbConnect()` function takes as a first argument a "connection" interface provided by the relevant database connection package (e.g., `RSQLite`). The remaining arguments specify the

[9] **dbplyr** repository page: https://github.com/tidyverse/dbplyr

location of the database, and are dependent on where that database is located and what kind of database it is. For example, you use a `dbname` argument to specify the path to a local SQLite database file, while you use `host`, `user`, and `password` to specify the connection to a database on a remote machine.

> **Caution**: Never include your database password directly in your R script—saving it in plain text will allow others to easily steal it! Instead, `dbplyr` recommends that you prompt users for the password through RStudio by using the `askForPassword()`[a] function from the `rstudioapi` package (which will cause a pop-up window to appear for users to type in their password). See the *dbplyr introduction vignette*[b] for an example.
>
> [a] https://www.rdocumentation.org/packages/rstudioapi/versions/0.7/topics/askForPassword
> [b] https://cran.r-project.org/web/packages/dbplyr/vignettes/dbplyr.html

Once you have a connection to the database, you can use the **`dbListTables()`** function to get a vector of all the table names. This is useful for checking that you've connected to the database (as well as seeing what data is available to you!).

Since all SQL queries access data `FROM` a particular table, you will need to start by creating a *reference to* that table in the form of a variable. You can do this by using the **`tbl()`** function provided by `dplyr` (*not* `dbplyr`!). This function takes as arguments the connection to the database and the name of the table you want to reference. For example, to query a `songs` table as in Figure 13.1, you would use the following:

```
# Create a reference to the "songs" table in the database
songs_table <- tbl(db_connection, "songs")
```

If you print this variable out, you will notice that it looks *mostly* like a normal data frame (specifically a tibble), except that the variable refers to a *remote* source (since the table is in the database, not in R!); see Figure 13.6.

Once you have a reference to the table, you can use the same `dplyr` functions discussed in Chapter 11 (e.g., `select()`, `filter()`). Just use the table in place of the data frame to manipulate!

```
# Construct a query from the `songs_table` for songs by Queen (artist ID 11)
queen_songs_query <- songs_table %>%
  filter(artist_id == 11) %>%
  select(title)
```

```
> print(songs_table)
# Source:   table<songs> [?? x 3]
# Database: sqlite 3.22.0 [/Users/mikefree/Documents/music_db.sqlite]
     id title            artist_id
  <int> <chr>                <int>
1    80 Bohemian Rhapsody       11
2    81 Don't Stop Me Now       11
3    82 Purple Rain             12
4    83 Starman                 10
>
```

Figure 13.6 A database `tbl`, printed in RStudio.This is only a *preview* of the data that will be returned by the data base.

The `dbplyr` package will automatically convert a sequence of `dplyr` functions into an equivalent SQL statement, without the need for you to write any SQL! You can see the SQL statement it is generating by using the **`show_query()`** function:

```
# Display the SQL syntax stored in the query `queen_songs_query`
show_query(queen_songs_query)
    # <SQL>
    # SELECT `title`
    # FROM `songs`
    # WHERE (`artist_id` = 11.0)
```

Importantly, using `dplyr` methods on a table does not return a data frame (or even a tibble). In fact, it displays just a small preview of the requested data! Actually querying the data from a database is relatively slow in comparison to accessing data in a data frame, particularly when the database is on a remote computer. Thus `dbplyr` uses **lazy evaluation**—it actually executes the query on the database only when you explicitly tell it to do so. What is shown when you print the `queen_songs_query` is just a subset of the data; the results will not include all of the rows returned if there are a large number of them! RStudio very subtly indicates that the data is just a preview of what has been requested—note in Figure 13.6 that the dimensions of the `songs_table` are unknown (i.e., `table<songs> [?? X 3]`). Lazy evaluation keeps you from accidentally making a large number of queries and downloading a huge amount of data as you are designing and testing your data manipulation statements (i.e., writing your `select()` and `filter()` calls).

To actually query the database and load the results into memory as a `R` value you can manipulate, use the **`collect()`** function. You can often add this function call as a last step in your pipe of `dplyr` calls.

```
# Execute the `queen_songs_query` request, returning the *actual data*
# from the database
queen_songs_data <- collect(queen_songs_query) # returns a tibble
```

This tibble is exactly like those described in earlier chapters; you can use `as.data.frame()` to convert it into a data frame. Thus, anytime you want to query data from a database in `R`, you will need to perform the following steps:

```
# 1. Create a connection to an RDMS, such as a SQLite database
db_connection <- dbConnect(SQLite(), dbname = "path/to/database.sqlite")

# 2. Access a specific table within your database
some_table <- tbl(db_connection, "TABLE_NAME")

# 3. Construct a query of the table using `dplyr` syntax
db_query <- some_table %>%
  filter(some_column == some_value)

# 4. Execute your query to return data from the database
results <- collect(db_query)
```

```
# 5. Disconnect from the database when you're finished
dbDisconnect(db_connection)
```

And with that, you have accessed and queried a database using R! You can now write R code to use the same `dplyr` functions for either a local data frame or a remote database, allowing you to test and then expand your data analysis.

> **Tip**: For more information on using `dbplyr`, check out the introduction vignette.[a]
>
> [a] https://cran.r-project.org/web/packages/dbplyr/vignettes/dbplyr.html

For practice working with databases, see the set of accompanying book exercises.[10]

[10] **Database exercises:** https://github.com/programming-for-data-science/chapter-13-exercises

This page intentionally left blank

14

Accessing Web APIs

Previous chapters have described how to access data from local `.csv` files, as well as from local databases. While working with local data is common for many analyses, more complex shared data systems leverage **web services** for data access. Rather than store data on each analyst's computer, data is stored on a *remote server* (i.e., a central computer somewhere on the internet) and accessed similarly to how you access information on the web (via a URL). This allows scripts to always work with the latest data available when performing analysis of data that may be changing rapidly, such as social media data.

In this chapter, you will learn how to use `R` to programmatically interact with data stored by web services. From an `R` script, you can read, write, and delete data stored by these services (though this book focuses on the skill of reading data). Web services may make their data accessible to computer programs like `R` scripts by offering an application programming interface (API). A web service's API specifies *where* and *how* particular data may be accessed, and many web services follow a particular style known as *REpresentational State Transfer (REST)*.[1] This chapter covers how to access and work with data from these RESTful APIs.

14.1 What Is a Web API?

An **interface** is the point at which two different systems meet and *communicate*, exchanging information and instructions. An **application programming interface (API)** thus represents a way of communicating with a computer application by writing a computer program (a set of formal instructions understandable by a machine). APIs commonly take the form of **functions** that can be called to give instructions to programs. For example, the set of functions provided by a package like `dplyr` make up the API for that package.

While some APIs provide an interface for leveraging some *functionality*, other APIs provide an interface for accessing *data*. One of the most common sources of these data APIs are web services—that is, websites that offer an interface for accessing their data.

With web services, the interface (the set of "functions" you can call to access the data) takes the form of **HTTP requests**—that is, requests for data sent following the ***H**yper**T**ext **T**ransfer **P**rotocol*.

[1] Fielding, R. T. (2000). *Architectural styles and the design of network-based software architectures.* University of California, Irvine, doctoral dissertation. https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm. Note that this is the original specification and is very technical.

This is the same protocol (way of communicating) used by your browser to view a webpage! An HTTP request represents a message that your computer sends to a **web server**: another computer on the internet that "serves," or provides, information. That server, upon receiving the request, will determine what data to include in the **response** it sends back to the requesting computer. With a web browser, the response data takes the form of HTML files that the browser can *render* as webpages. With data APIs, the response data will be structured data that you can convert into R data types such as lists or data frames.

In short, loading data from a web API involves sending an HTTP request to a server for a particular piece of data, and then receiving and parsing the response to that request.

Learning how to use web APIs will greatly expand the available data sets you may want to use for analysis. Companies and services with large amounts of data, such as Twitter,[2] iTunes,[3] or Reddit,[4] make (some of) their data publicly accessible through an API. This chapter will use the GitHub API[5] to demonstrate how to work with data stored in a web service.

14.2 RESTful Requests

There are two parts to a request sent to a web API: the name of the resource (data) that you wish to access, and a verb indicating what you want to do with that resource. In a way, the verb is the function you want to call on the API, and the resource is an argument to that function.

14.2.1 URIs

Which resource you want to access is specified with a **Uniform Resource Identifier (URI)**.[6] A URI is a generalization of a URL (Uniform Resource Locator)—what you commonly think of as a "web address." A URI acts a lot like the address on a postal letter sent within a large organization such as a university: you indicate the business address as well as the department and the person to receive the letter, and will get a different response (and different data) from Alice in Accounting than from Sally in Sales.

Like postal letter addresses, URIs have a very specific format used to direct the request to the right resource, illustrated in Figure 14.1.

Figure 14.1 The format (schema) of a URI.

[2] **Twitter API**: https://developer.twitter.com/en/docs
[3] **iTunes search API**: https://affiliate.itunes.apple.com/resources/documentation/itunes-store-web-service-search-api/
[4] **Reddit API**: https://www.reddit.com/dev/api/
[5] **GitHub API**: https://developer.github.com/v3/
[6] **Uniform Resource Identifier (URI) Generic Syntax** (official technical specification): https://tools.ietf.org/html/rfc3986

Not all parts of the URI are required. For example, you don't necessarily need a `port`, `query`, or `fragment`. Important parts of the URI include:

- `scheme` (`protocol`): The "language" that the computer will use to communicate the request to the API. With web services this is normally `https` (secure HTTP).
- `domain`: The address of the web server to request information from.
- `path`: The identifier of the resource on that web server you wish to access. This may be the name of a file with an extension if you're trying to access a particular file, but with web services it often just looks like a folder path!
- `query`: Extra parameters (arguments) with further details about the resource to access.

The `domain` and `path` usually specify the location of the resource of interest. For example, `www.domain.com/users` might be an *identifier* for a *resource* that serves information about all the users. Web services can also have "subresources" that you can access by adding extra pieces to the path. For example, `www.domain.com/users/layla` might access to the specific resource ("layla") that you are interested in.

With web APIs, the URI is often viewed as being broken up into three parts, as shown in Figure 14.2:

- The **base URI** is the domain that is included on *all* resources. It acts as the "root" for any particular endpoint. For example, the GitHub API has a base URI of `https://api.github.com`. All requests to the GitHub API will have that base.
- An **endpoint** is the location that holds the specific information you want to access. Each API will have many different endpoints at which you can access specific data resources. The GitHub API, for example, has different endpoints for `/users` and `/orgs` so that you can access data about users or organizations, respectively.

  Note that many endpoints support accessing multiple subresources. For example, you can access information about a specific user at the endpoint `/users/:username`. The colon `:` indicates that the subresource name is a *variable*—you can replace that part of the endpoint with whatever string you want. Thus if you were interested in the GitHub user `nbremer`,[7] you would access the `/users/nbremer` endpoint.

  Subresources may have further subresources (which may or may not have variable names). The endpoint `/orgs/:org/repos` refers to the list of repositories belonging to an organization. Variable names in endpoints might alternatively be written inside of curly braces `{}`—for example, `/orgs/{org}/repos`. Neither the colon nor the braces are

Figure 14.2 The anatomy of a web API request URI.

[7] **Nadieh Bremer, freelance data visualization designer:** https://www.visualcinnamon.com

programming language syntax; instead, they are common conventions used to communicate how to specify endpoints.

- **Query parameters** allow you to specify additional information about which exact information you want from the endpoint, or how you want it to be organized (see Section 14.2.1.1 for more details).

Remember: One of the biggest challenges in accessing a web API is understanding what resources (data) the web service makes available and which endpoints (URIs) can request those resources. Read the web service's documentation carefully—popular services often include examples of URIs and the data returned from them.

A query is constructed by appending the endpoint and any query parameters to the base URI. For example, so you could access a GitHub user by combining the base URI (`https://api.github.com`) and endpoint (`/users/nbremer`) into a single string: `https://api.github.com/users/nbremer`. Sending a request to that URI will return data about the user—you can send this request from an `R` program or by visiting that URI in a web browser, as shown in Figure 14.3. In short, you can access a particular data *resource* by sending a request to a particular *endpoint*.

Indeed, one of the easiest ways to make a request to a web API is by navigating to the URI using your web browser. Viewing the information in your browser is a great way to explore the resulting data, and make sure you are requesting information from the proper URI (i.e., that you haven't made a typo in the URI).

Tip: The JSON format (see Section 14.4) of data returned from web APIs can be quite messy when viewed in a web browser. Installing a browser extension such as *JSONView*[a] will format the data in a somewhat more readable way. Figure 14.3 shows data formatted with this extension.

[a] https://chrome.google.com/webstore/detail/jsonview/chklaanhfefbnpoihckbnefhakgolnmc

14.2.1.1 Query Parameters

Web URIs can optionally include **query parameters**, which are used to request a more specific subset of data. You can think of them as additional optional arguments that are given to the request function—for example, a keyword to search for or criteria to order results by.

The query parameters are listed at the end of a URI, following a question mark (**?**) and are formed as *key–value* pairs similar to how you named items in lists. The **key** (parameter name) is listed first, followed by an equals sign (**=**), followed by the **value** (parameter value), with no spaces between anything. You can include multiple query parameters by putting an ampersand (**&**) between each key–value pair. You can see an example of this syntax by looking at the URL bar in a web browser when you use a search engine such as Google or Yahoo, as shown in Figure 14.4. Search engines produce URLs with a lot of query parameters, not all of which are obvious or understandable.

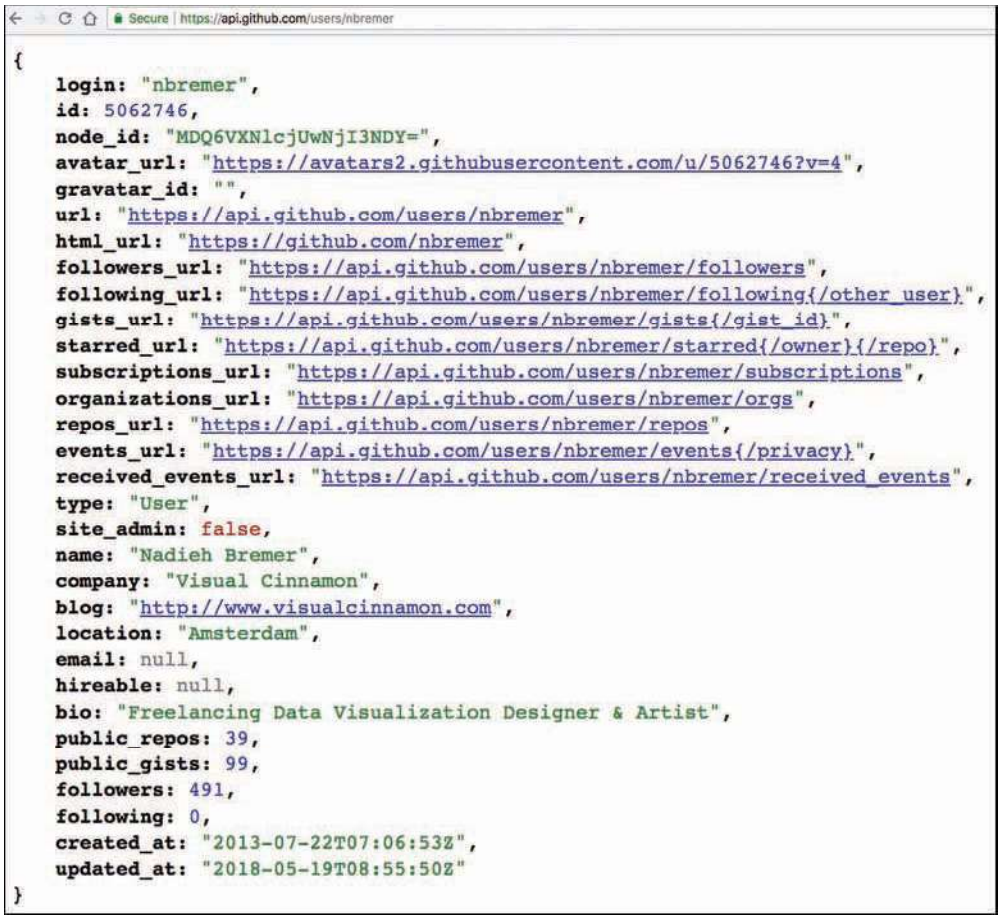


Figure 14.3 GitHub API response returned by the URI `https://api.github.com/users/nbremer`, as displayed in a web browser.

Notice that the exact query parameter name used differs depending on the web service. Google uses a `q` parameter (likely for "query") to store the search term, while Yahoo uses a `p` parameter.

Similar to arguments for functions, API endpoints may either require query parameters (e.g., you *must* provide a search term) or optionally allow them (e.g., you *may* provide a sorting order). For example, the GitHub API has a `/search/repositories` endpoint that allows users to *search* for a specific repository: you are required to provide a `q` parameter for the query, and can optionally provide a `sort` parameter for how to sort the results:

```
# A GitHub API URI with query parameters: search term `q` and sort
# order `sort`
https://api.github.com/search/repositories?q=dplyr&sort=forks
```

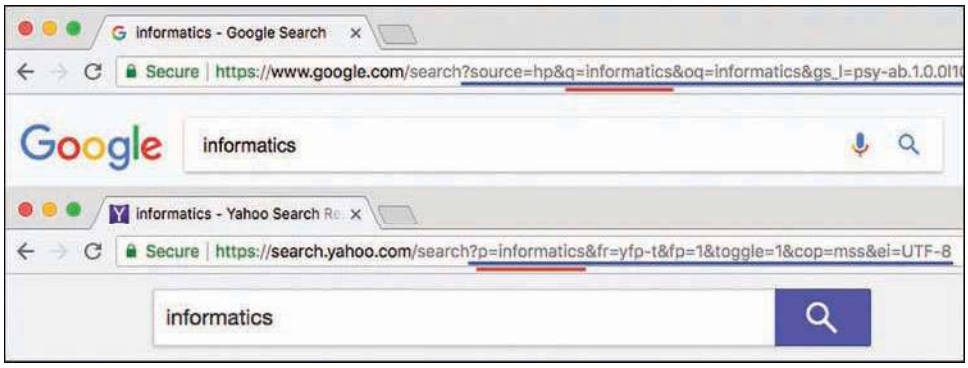


Figure 14.4 Search engine URLs for Google (top) and Yahoo (bottom) with query parameters (underlined in blue). The "search term" parameter for each web service is underlined in red.

Results from this request are shown in Figure 14.5.

> **Caution**: Many special characters (e.g., punctuation) cannot be included in a URL. This group includes characters such as spaces! Browsers and many HTTP request packages will automatically *encode* these special characters into a usable format (for example, converting a space into a `%20`), but sometimes you may need to do this conversion yourself.

14.2.1.2 Access Tokens and API Keys

Many web services require you to register with them to send them requests. This allows the web service to limit access to the data, as well as to keep track of who is asking for which data (usually so that if someone starts "spamming" the service, that user can be blocked).

To facilitate this tracking, many services provide users with **access tokens** (also called **API keys**). These unique strings of letters and numbers identify a particular developer (like a secret password that works just for you). Furthermore, your API key can provide you with additional access to information based on which user you are. For example, when you get an access key for the GitHub API, that key will provide you with additional access and control over your repositories. This enables you to request information about private repos, and even programmatically interact with GitHub through the API (i.e., you can delete a repo[8]—so tread carefully!).

Web services will require you to include your access token in the request, usually as a query parameter; the exact name of the parameter varies, but it often looks like `access_token` or `api_key`. When exploring a web service, keep an eye out for whether it requires such tokens.

[8] **GitHub API, delete a repository** https://developer.github.com/v3/repos/#delete-a- repository

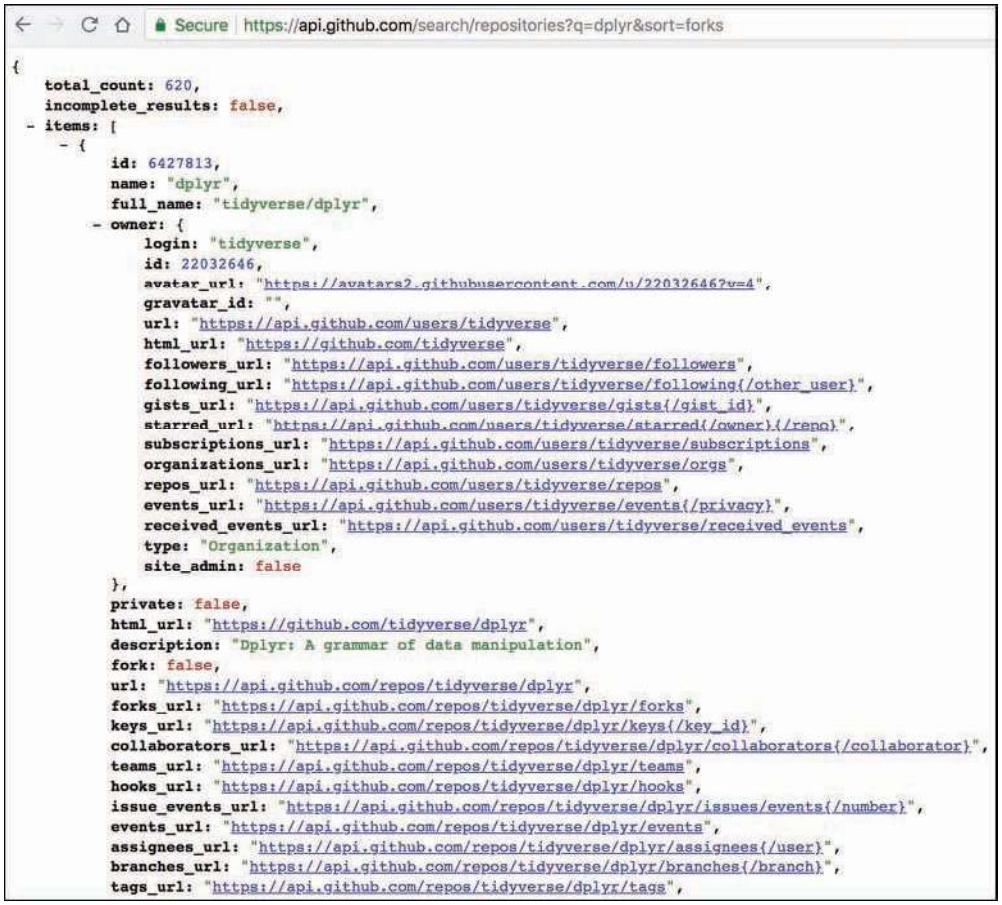


Figure 14.5 A subset of the GitHub API response returned by the URI `https://api.github.com/search/repositories?q=dplyr&sort=forks`, as displayed in a web browser.

Caution: Watch out for APIs that mention using an authentication service called **OAuth** when explaining required API keys. OAuth is a system for performing **authentication**—that is, having someone *prove* that they are who they say they are. OAuth is generally used to let someone log into a website from your application (like what a "Log in with Google" button does). OAuth systems require more than one access key, and these keys *must* be kept secret. Moreover, they usually require you to run a web server to use them correctly (which requires significant extra setup; see the full `httr` documentation[a] for details). You can do this in R, but may want to avoid this challenge while learning how to use APIs.

[a] https://cran.r-project.org/web/packages/httr/httr.pdf

Access tokens are a lot like passwords; you will want to keep them secret and not share them with others. This means that you should not include them in any files you commit to git and push to GitHub. The best way to ensure the secrecy of access tokens in R is to create a separate script file in your repo (e.g., `api_keys.R`) that includes exactly one line, assigning the key to a variable:

```
# Store your API key from a web service in a variable
# It should be in a separate file (e.g., `api_keys.R`)
api_key <- "123456789abcdefg"
```

To access this variable in your "main" script, you can use the **`source()`** function to load and run your `api_keys.R` script (similar to clicking the *Source* button to run a script). This function will execute all lines of code in the specified script file, as if you had "copy-and-pasted" its contents and run them all with `ctrl+enter`. When you use `source()` to execute the `api_keys.R` script, it will execute the code statement that defines the `api_key` variable, making it available in your environment for your use:

```
# In your "main" script, load your API key from another file

# (Make sure working directory is set before running the following code!)

source("api_keys.R") # load the script using a *relative path*
print(api_key) # the key is now available!
```

Anyone else who runs the script will need to provide an `api_key` variable to access the API using that user's own key. This practice keeps everyone's account separate.

You can keep your `api_keys.R` file from being committed by including the filename in the **`.gitignore`** file in your repo; that will keep it from even possibly being committed with your code! See Chapter 3 for details about working with the `.gitignore` file.

14.2.2 HTTP Verbs

When you send a request to a particular resource, you need to indicate what you want to *do* with that resource. This is achieved by specifying an **HTTP verb** in the request. The HTTP protocol supports the following verbs:

- `GET`: Return a representation of the current state of the resource.
- `POST`: Add a new subresource (e.g., insert a record).
- `PUT`: Update the resource to have a new state.
- `PATCH`: Update a portion of the resource's state.
- `DELETE`: Remove the resource.
- `OPTIONS`: Return the set of methods that can be performed on the resource.

By far the most commonly used verb is **GET**, which is used to "get" (download) data from a web service—this is the type of request that is sent when you enter a URL into a web browser. Thus you would send a `GET` request for the `/users/nbremer` endpoint to access that data resource.

Taken together, this structure of treating each datum on the web as a resource that you can interact with via HTTP requests is referred to as the **REST architecture** (*REpresentational State Transfer*). Thus, a web service that enables data access through named resources and responds to HTTP requests is known as a **RESTful** service, that has a RESTful API.

14.3 Accessing Web APIs from R

To access a web API, you just need to send an HTTP request to a particular URI. As mentioned earlier, you can easily do this with the browser: navigate to a particular address (base URI + endpoint), and that will cause the browser to send a `GET` request and display the resulting data. For example, you can send a request to the GitHub API to search for repositories that match the string "`dplyr`" (see the response in Figure 14.5):

```
# The URI for the `search/repositories` endpoint of the GitHub API: query
# for `dplyr`, sorting by `forks`
https://api.github.com/search/repositories?q=dplyr&sort=forks
```

This query accesses the `/search/repositories` endpoint, and also specifies two query parameters:

- `q`: The term(s) you are searching for
- `sort`: The attribute of each repository that you would like to use to sort the results (in this case, the number of forks of the repo)

(Note that the data you will get back is structured in JSON format. See Section 14.4 for details.)

While you can access this information using your browser, you will want to load it into R for analysis. In R, you can send `GET` requests using the **`httr`**[9] package. As with `dplyr`, you will need to install and load this package to use it:

```
install.packages("httr") # once per machine
library("httr")          # in each relevant script
```

This package provides a number of functions that reflect HTTP verbs. For example, the **`GET()`** function will send an HTTP `GET` request to the URI:

```
# Make a GET request to the GitHub API's "/search/repositories" endpoint
# Request repositories that match the search "dplyr", and sort the results
# by forks
url <- "https://api.github.com/search/repositories?q=dplyr&sort=forks"
response <- GET(url)
```

This code will make the same request as your web browser, and store the response in a variable called `response`. While it is possible to include query parameters in the URI string (as above), `httr`

[9] **Getting started with httr**: official quickstart guide for `httr`: https://cran.r-project.org/web/packages/httr/vignettes/quickstart.html

also allows you to include them as a list passed as a `query` argument. Furthermore, if you plan on accessing multiple different endpoints (which is common), you can structure your code a bit more modularly, as described in the following example; this structure makes it easy to set and change variables (instead of needing to do a complex `paste()` operation to produce the correct string):

```
# Restructure the previous request to make it easier to read and update. DO THIS.

# Make a GET request to the GitHub API's "search/repositories" endpoint
# Request repositories that match the search "dplyr", sorted by forks

# Construct your `resource_uri` from a reusable  `base_uri` and an `endpoint`
base_uri <- "https://api.github.com"
endpoint <- "/search/repositories"
resource_uri <- paste0(base_uri, endpoint)

# Store any query parameters you want to use in a list
query_params <- list(q = "dplyr", sort = "forks")

# Make your request, specifying the query parameters via the `query` argument
response <- GET(resource_uri, query = query_params)
```

If you try printing out the `response` variable that is returned by the `GET()` function, you will first see information about the response:

```
Response [https://api.github.com/search/repositories?q=dplyr&sort=forks]
  Date: 2018-03-14 06:43
  Status: 200
  Content-Type: application/json; charset=utf-8
  Size: 171 kB
```

This is called the **response header**. Each response has two parts: the **header** and the **body**. You can think of the response as an envelope: the header contains meta-data like the address and postage date, while the body contains the actual contents of the letter (the data).

> **Tip**: The URI shown when you print out the `response` variable is a good way to check exactly which URI you sent the request to: copy that into your browser to make sure it goes where you expected!

Since you are almost always interested in working with the response body, you will need to extract that data from the response (e.g., open up the envelope and pull out the letter). You can do this with the `content()` function:

```
# Extract content from `response`, as a text string
response_text <- content(response, type = "text")
```

Note the second argument `type = "text"`; this is needed to keep `httr` from doing its own processing on the response data (you will use other methods to handle that processing).

14.4 Processing JSON Data

Now that you're able to load data into R from an API and extract the content as text, you will need to transform the information into a usable format. Most APIs will return data in **JavaScript Object Notation (JSON)** format. Like CSV, JSON is a format for writing down structured data—but, while `.csv` files organize data into rows and columns (like a data frame), JSON allows you to organize elements into key–value pairs similar to an R *list*! This allows the data to have much more complex structure, which is useful for web services, but can be challenging for data programming.

In JSON, lists of key–value pairs (called **objects**) are put inside braces (**`{ }`**), with the key and the value separated by a colon (**`:`**) and each pair separated by a comma (**`,`**). Key–value pairs are often written on separate lines for readability, but this isn't required. Note that keys need to be character strings (so, *"in quotes"*), while values can either be character strings, numbers, booleans (written in lowercase as `true` and `false`), or even other lists! For example:

```
{
  "first_name": "Ada",
  "job": "Programmer",
  "salary": 78000,
  "in_union": true,
  "favorites": {
    "music": "jazz",
    "food": "pizza",
  }
}
```

The above JSON object is equivalent to the following R list:

```
# Represent the sample JSON data (info about a person) as a list in R
list(
  first_name = "Ada",
  job = "Programmer",
  salary = 78000,
  in_union = TRUE,
  favorites = list(music = "jazz", food = "pizza") # nested list in the list!
)
```

Additionally, JSON supports **arrays** of data. Arrays are like *untagged lists* (or vectors with different types), and are written in square brackets (**`[ ]`**), with values separated by commas. For example:

```
["Aardvark", "Baboon", "Camel"]
```

which is equivalent to the R list:

```
list("Aardvark", "Baboon", "Camel")
```

Just as R allows you to have nested lists of lists, JSON can have any form of nested objects and arrays. This structure allows you to store arrays (think *vectors*) within objects (think *lists*), such as the following (more complex) set of data about Ada:

```
{
  "first_name": "Ada",
  "job": "Programmer",
  "pets": ["Magnet", "Mocha", "Anni", "Fifi"],
  "favorites": {
    "music": "jazz",
    "food": "pizza",
    "colors": ["green", "blue"]
  }
}
```

The JSON equivalent of a data frame is to store data as an *array of objects*. This is like having a list of lists. For example, the following is an array of objects of FIFA Men's World Cup data[10]:

```
[
  {"country": "Brazil", "titles": 5, "total_wins": 70, "total_losses": 17},
  {"country": "Italy", "titles": 4, "total_wins": 66, "total_losses": 20},
  {"country": "Germany", "titles": 4, "total_wins": 45, "total_losses": 17},
  {"country": "Argentina", "titles": 2, "total_wins": 42, "total_losses": 21},
  {"country": "Uruguay", "titles": 2, "total_wins": 20, "total_losses": 19}
]
```

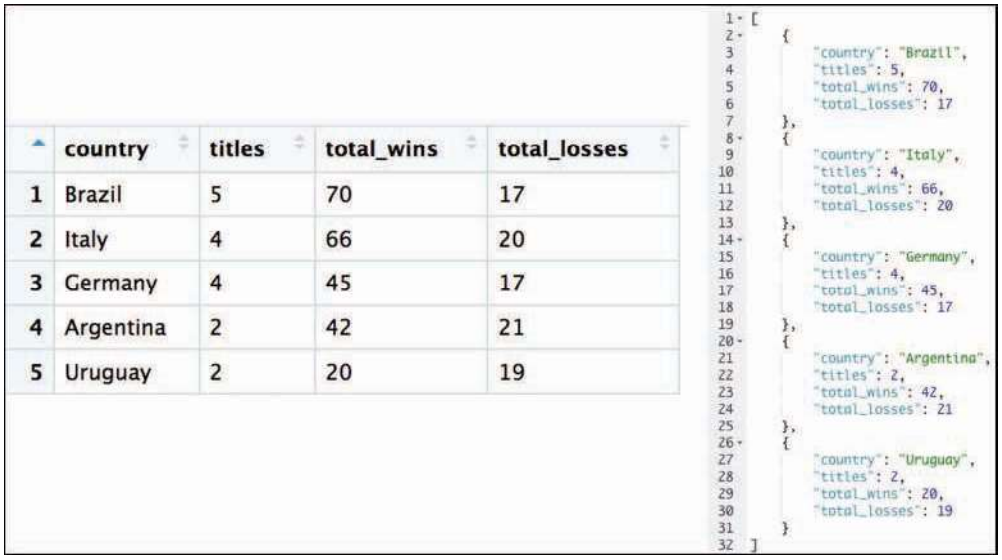
You could think of this information as a *list of lists* in R:

```
# Represent the sample JSON data (World Cup data) as a list of lists in R
list(
  list(country = "Brazil", titles = 5, total_wins = 70, total_losses = 17),
  list(country = "Italy", titles = 4, total_wins = 66, total_losses = 20),
  list(country = "Germany", titles = 4, total_wins = 45, total_losses = 17),
  list(country = "Argentina", titles = 2, total_wins = 42, total_losses = 21),
  list(country = "Uruguay", titles = 2, total_wins = 20, total_losses = 19)
)
```

This structure is incredibly common in web API data: as long as each object in the array has the same set of keys, then you can easily consider this structure to be a data frame where each object (list) represents an observation (row), and each key represents a feature (column) of that observation. A data frame representation of this data is shown in Figure 14.6.

> **Remember**: In JSON, tables are represented as lists of *rows*, instead of a data frame's list of *columns*.

[10] **FIFA World Cup data:** https://www.fifa.com/fifa-tournaments/statistics-and-records/worldcup/teams/index.html



	country	titles	total_wins	total_losses
1	Brazil	5	70	17
2	Italy	4	66	20
3	Germany	4	45	17
4	Argentina	2	42	21
5	Uruguay	2	20	19


```
[
    {
        "country": "Brazil",
        "titles": 5,
        "total_wins": 70,
        "total_losses": 17
    },
    {
        "country": "Italy",
        "titles": 4,
        "total_wins": 66,
        "total_losses": 20
    },
    {
        "country": "Germany",
        "titles": 4,
        "total_wins": 45,
        "total_losses": 17
    },
    {
        "country": "Argentina",
        "titles": 2,
        "total_wins": 42,
        "total_losses": 21
    },
    {
        "country": "Uruguay",
        "titles": 2,
        "total_wins": 20,
        "total_losses": 19
    }
]
```


Figure 14.6 A data frame representation of World Cup statistics (left), which can also be represented as JSON data (right).

14.4.1 Parsing JSON

When working with a web API, the usual goal is to take the JSON data contained in the response and convert it into an R data structure you can use, such as a list or data frame. This will allow you to interact with the data by using the data manipulation skills introduced in earlier chapters. While the `httr` package is able to parse the JSON body of a response into a list, it doesn't do a very clean job of it (particularly for complex data structures).

A more effective solution for transforming JSON data is to use the `jsonlite` package.[11] This package provides helpful methods to convert JSON data into R data, and is particularly well suited for converting content into data frames.

As always, you will need to install and load this package:

```
install.packages("jsonlite") # once per machine
library("jsonlite") # in each relevant script
```

The `jsonlite` package provides a function called **`fromJSON()`** that allows you to convert from a JSON string into a list—or even a data frame if the intended columns have the same lengths!

[11] Package **`jsonlite`**: full documentation for `jsonlite`: https://cran.r-project.org/web/packages/jsonlite/jsonlite.pdf

```
# Make a request to a given `uri` with a set of `query_params`
# Then extract and parse the results

# Make the request
response <- GET(uri, query = query_params)

# Extract the content of the response
response_text <- content(response, "text")

# Convert the JSON string to a list
response_data <- fromJSON(response_text)
```

Both the raw JSON data (`response_text`) and the parsed data structure (`response_data`) are shown in Figure 14.7. As you can see, the raw string (`response_text`) is indecipherable. However, once it is transformed using the `fromJSON()` function, it has a much more operable structure.

The `response_data` will contain a list built out of the JSON. Depending on the complexity of the JSON, this may already be a data frame you can `View()`—but more likely you will need to explore the list to locate the "main" data you are interested in. Good strategies for this include the following techniques:

- Use functions such as `is.data.frame()` to determine whether the data is already structured as a data frame.
- You can `print()` the data, but that is often hard to read (it requires a lot of scrolling).
- The `str()` function will return a list's structure, though it can still be hard to read.
- The `names()` function will return the keys of the list, which is helpful for delving into the data.

```
> response_text
[1] "{\n  \"total_count\": 707,\n  \"incomplete_results\":
false,\n  \"items\": [\n    {\n      \"id\": 6427813,\n
\"node_id\": \"MDEwOlJlcG9zaXRvcnk2NDI3ODEz\",\n      \"na
me\": \"dplyr\",\n      \"full_name\": \"tidyverse/dplyr\"
,\n      \"owner\": {\n        \"login\": \"tidyverse\",\n
\"id\": 22032646,\n        \"node_id\": \"MDEyOk9yZ2FuaXph
dGlvbjIyMDMyNjQ2\",\n        \"avatar_url\": \"https://ava
tars2.githubusercontent.com/u/22032646?v=4\",\n        \"g
ravatar_id\": \"\",\n        \"url\": \"https://api.github
.com/users/tidyverse\",\n        \"html_url\": \"https://g
ithub.com/tidyverse\",\n        \"followers_url\": \"https
://api.github.com/users/tidyverse/followers\",\n        \"
following_url\": \"https://api.github.com/users/tidyverse/
following{/other_user}\",\n        \"gists_url\": \"https:
//api.github.com/users/tidyverse/gists{/gist_id}\",\n
```

```
> fromJSON(response_text)
$total_count
[1] 707

$incomplete_results
[1] FALSE

$items
        id                                node_id
1  6427813     MDEwOlJlcG9zaXRvcnk2NDI3ODEz
2 67845042 MDEwOlJlcG9zaXRvcnk2Nzg0NTA0Mg==
3 59305491 MDEwOlJlcG9zaXRvcnk1OTMwNTQ5MQ==
4 24485567 MDEwOlJlcG9zaXRvcnkyNDQ4NTU2Nw==
5 126367748 MDEwOlJlcG9zaXRvcnkxMjYzNjc3NDg=
6 55175084 MDEwOlJlcG9zaXRvcnk1NTE3NTA4NA==
7 118410287 MDEwOlJlcG9zaXRvcnkxMTg0MTAyODc=
```

Figure 14.7 Parsing the text of an API response using `fromJSON()`. The untransformed text is shown on the left (`response_text`), which is transformed into a list (on the right) using the `fromJSON()` function.

As an example continuing the previous code:

```
# Use various methods to explore and extract information from API results

# Check: is it a data frame already?
is.data.frame(response_data) # FALSE

# Inspect the data!
str(response_data) # view as a formatted string
names(response_data) # "href" "items" "limit" "next" "offset" "previous" "total"

# Looking at the JSON data itself (e.g., in the browser),
# `items` is the key that contains the value you want

# Extract the (useful) data
items <- response_data$items # extract from the list
is.data.frame(items) # TRUE; you can work with that!
```

The set of responses—GitHub repositories that match the search term *"dplry"*—returned from the request and stored in the `response_data$items` key is shown in Figure 14.8.

14.4.2 Flattening Data

Because JSON supports—and in fact encourages—nested lists (lists within lists), parsing a JSON string is likely to produce a data frame whose columns *are themselves data frames*. As an example of what a nested data frame may look like, consider the following code:

```
# A demonstration of the structure of "nested" data frames

# Create a `people` data frame with a `names` column
people <- data.frame(names = c("Ed", "Jessica", "Keagan"))
```

	id	node_id	name	full_name
1	6427813	MDEwOlJlcG9zaXRvcnk2NDI3ODEz	dplyr	tidyverse/dplyr
2	67845042	MDEwOlJlcG9zaXRvcnk2Nzg0NTA0Mg==	m9-dplyr	INFO-201/m9-dplyr
3	59305491	MDEwOlJlcG9zaXRvcnk1OTMwNTQ5MQ==	sparklyr	rstudio/sparklyr
4	24485567	MDEwOlJlcG9zaXRvcnkyNDQ4NTU2Nw==	dplyr-tutorial	justmarkham/dplyr-tutorial
5	126367748	MDEwOlJlcG9zaXRvcnkxMjYzNjc3NDg=	ch10-dplyr	info201/ch10-dplyr
6	55175084	MDEwOlJlcG9zaXRvcnk1NTE3NTA4NA==	tidytext	juliasilge/tidytext
7	118410287	MDEwOlJlcG9zaXRvcnkxMTg0MTAyODc=	ch10-dplyr	info201a-w18/ch10-dplyr
8	50487685	MDEwOlJlcG9zaXRvcnk1MDQ4NzY4NQ==	lecture-8-exercises	INFO-498F/lecture-8-exercises
9	86504302	MDEwOlJlcG9zaXRvcnk4NjUwNDMwMg==	dbplyr	tidyverse/dbplyr
10	84520584	MDEwOlJlcG9zaXRvcnk4NDUyMDU4NA==	Data-Analysis-with-R	susanli2016/Data-Analysis-with-R

Figure 14.8 Data returned by the GitHub API: repositories that match the term *"dplyr"* (stored in the variable `response_data$items` in the code example).

```
# Create a data frame of favorites with two columns
favorites <- data.frame(
  food = c("Pizza", "Pasta", "Salad"),
  music = c("Bluegrass", "Indie", "Electronic")
)

# Store the second data frame as a column of the first -- A BAD IDEA
people$favorites <- favorites # the `favorites` column is a data frame!

# This prints nicely, but is misleading
print(people)
  #     names favorites.food favorites.music
  # 1      Ed          Pizza       Bluegrass
  # 2 Jessica          Pasta           Indie
  # 3  Keagan          Salad      Electronic

# Despite what RStudio prints, there is not actually a column `favorites.food`
people$favorites.food # NULL

# Access the `food` column of the data frame stored in `people$favorites`
people$favorites$food # [1] Pizza Pasta Salad
```

Nested data frames make it hard to work with the data using previously established techniques and syntax. Luckily, the `jsonlite` package provides a helpful function for addressing this issue, called **`flatten()`**. This function takes the columns of each nested data frame and converts them into appropriately named columns in the "outer" data frame, as shown in Figure 14.9:

```
# Use `flatten()` to format nested data frames
people <- flatten(people)
people$favorites.food # this just got created! Woo!
```

Note that `flatten()` works on only values that are already data frames. Thus you may need to find the appropriate element inside of the list—that is, the element that is the data frame you want to flatten.

In practice, you will almost always want to flatten the data returned from a web API. Thus, your algorithm for requesting and parsing data from an API is this:

1. Use `GET()` to *request the data* from an API, specifying the URI (and any query parameters).
2. Use `content()` to *extract the data* from your response as a JSON string (as "text").
3. Use `fromJSON()` to *convert the data* from a JSON string into a list.
4. Explore the returned information to *find your data* of interest.
5. Use `flatten()` to *flatten your data* into a properly structured data frame.
6. Programmatically analyze your data frame in R (e.g., with `dplyr`).

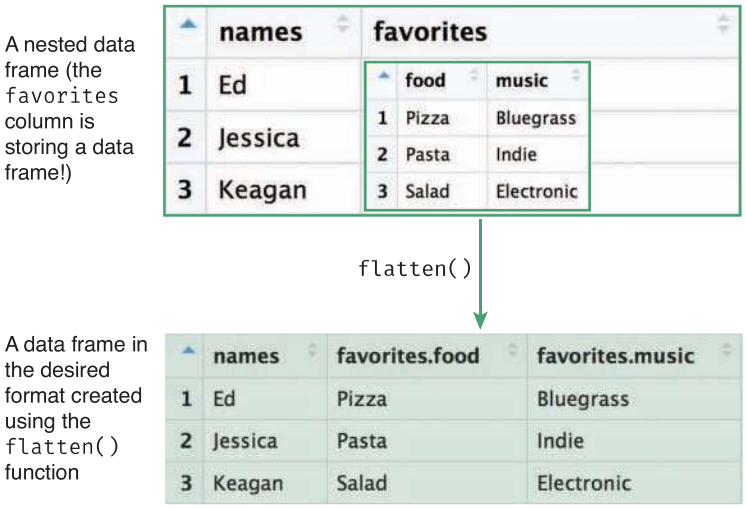


Figure 14.9 The `flatten()` function transforming a nested data frame (top) into a usable format (bottom).

14.5 APIs in Action: Finding Cuban Food in Seattle

This section uses the Yelp Fusion API[12] to answer the question:

> *"Where is the best Cuban food in Seattle?"*

Given the geographic nature of this question, this section builds a map of the best-rated Cuban restaurants in Seattle, as shown in Figure 14.12. The complete code for this analysis is also available online in the book's code repository.[13]

To send requests to the Yelp Fusion API, you will need to acquire an API key. You can do this by signing up for an account on the API's website, and registering an application (it is common for APIs to require you to register for access). As described earlier, you should store your API key in a separate file so that it can be kept secret:

```
# Store your API key in a variable: to be done in a separate file
# (i.e., "api_key.R")
yelp_key <- "abcdef123456"
```

This API requires you to use an alternative syntax for specifying your API key in the HTTP request—instead of passing your key as a query parameter, you'll need to add a header to the request that you make to the API. An **HTTP header** provides additional information to the server about *who is sending the request*—it's like extra information on the request's envelope. Specifically,

[12] **Yelp Fusion API documentation**: https://www.yelp.com/developers/documentation/v3
[13] **APIs in Action**: https://github.com/programming-for-data-science/in-action/tree/master/apis

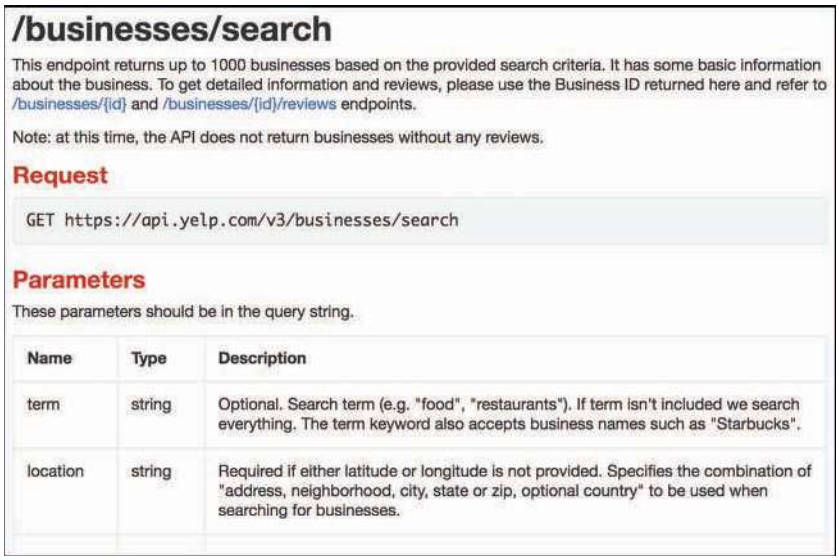
you will need to include an "Authorization" header containing your API key (in the format expected by the API) for the request to be accepted:

```
# Load your API key from a separate file so that you can access the API:
source("api_key.R") # the `yelp_key` variable is now available

# Make a GET request, including your API key as a header
response <- GET(
  uri,
  query = query_params,
  add_headers(Authorization = paste("bearer", yelp_key))
)
```

This code invokes the `add_headers()` method *inside* the `GET()` request. The header that it adds sets the value of the `Authorization` header to *"bearer yelp_key"*. This syntax indicates that the API should grant authorization to the bearer of the API key (you). This authentication process is used instead of setting the API key as a query parameter (a method of authentication that is not supported by the Yelp Fusion API).

As with any other API, you can determine the URI to send the request to by reading through the documentation. Given the prompt of *searching* for Cuban restaurants in Seattle, you should focus on the *Business Search* documentation,[14] a section of which is shown in Figure 14.10.



/businesses/search

This endpoint returns up to 1000 businesses based on the provided search criteria. It has some basic information about the business. To get detailed information and reviews, please use the Business ID returned here and refer to /businesses/{id} and /businesses/{id}/reviews endpoints.

Note: at this time, the API does not return businesses without any reviews.

Request

```
GET https://api.yelp.com/v3/businesses/search
```

Parameters

These parameters should be in the query string.

Name	Type	Description
term	string	Optional. Search term (e.g. "food", "restaurants"). If term isn't included we search everything. The term keyword also accepts business names such as "Starbucks".
location	string	Required if either latitude or longitude is not provided. Specifies the combination of "address, neighborhood, city, state or zip, optional country" to be used when searching for businesses.

Figure 14.10 A subset of the Yelp Fusion API *Business Search* documentation.

[14] **Yelp Fusion API Business Search** endpoint documentation: https://www.yelp.com/developers/documentation/v3/business_search

As you read through the documentation, it is important to identify the query parameters that you need to specify in your request. In doing so, you are mapping from your question of interest to the specific R code you will need to write. For this question ("Where is the best Cuban food in Seattle?"), you need to figure out how to make the following specifications:

- **Food**: Rather than search all businesses, you need to search for only restaurants. The API makes this available through the `term` parameter.
- **Cuban**: The restaurants you are interested in must be of a certain type. To support this, you can specify the `category` of your search (making sure to specify a supported category, as described elsewhere in the documentation[15]).
- **Seattle**: The restaurant you are looking for must be in Seattle. There are a few ways of specifying a location, the most general of which is to use the `location` parameter. You can further limit your results using the `radius` parameter.
- **Best**: To find the best food, you can control how the results are sorted with the `sort_by` parameter. You'll want to sort the results before you receive them (that is, by using an API parameter and not `dplyr`) to save you some effort and to make sure the API sends only the data you care about.

Often the most time-consuming part of using an API is figuring out how to hone in on your data of interest using the parameters of the API. Once you understand how to control which resource (data) is returned, you can then construct and send an HTTP request to the API:

```
# Construct a search query for the Yelp Fusion API's Business Search endpoint
base_uri <- "https://api.yelp.com/v3"
endpoint <- "/businesses/search"
search_uri <- paste0(base_uri, endpoint)

# Store a list of query parameters for Cuban restaurants around Seattle
query_params <- list(
  term = "restaurant",
  categories = "cuban",
  location = "Seattle, WA",
  sort_by = "rating",
  radius = 8000 # measured in meters, as detailed in the documentation
)

# Make a GET request, including the API key (as a header) and the list of
# query parameters
response <- GET(
  search_uri,
  query = query_params,
  add_headers(Authorization = paste("bearer", yelp_key))
)
```

[15] **Yelp Fusion API Category List:** https://www.yelp.com/developers/documentation/v3/all_category_list

As with any other API response, you will need to use the `content()` method to extract the content from the response, and then format the result using the `fromJSON()` method. You will then need to find the data frame of interest in your response. A great way to start is to use the `names()` function on your result to see what data is available (in this case, you should notice that the `businesses` key stores the desired information). You can `flatten()` this item into a data frame for easy access.

```
# Parse results and isolate data of interest
response_text <- content(response, type = "text")
response_data <- fromJSON(response_text)

# Inspect the response data
names(response_data) # [1] "businesses" "total" "region"

# Flatten the data frame stored in the `businesses` key of the response
restaurants <- flatten(response_data$businesses)
```

The data frame returned by the API is shown in Figure 14.11.

Because the data was requested in sorted format, you can *mutate* the data frame to include a column with the rank number, as well as add a column with a string representation of the name and rank:

```
# Modify the data frame for analysis and presentation
# Generate a rank of each restaurant based on row number
restaurants <- restaurants %>%
  mutate(rank = row_number()) %>%
  mutate(name_and_rank = paste0(rank, ". ", name))
```

The final step is to create a map of the results. The following code uses two different visualization packages (namely, `ggmap` and `ggplot2`), both of which are explained in more detail in Chapter 16.

	id	alias	name	image_url	is_closed	url
1	Wk9f5Zpnu4T6Vzf6CF5iuA	paseo-caribbean-food-fremont-seattle-2	Paseo Caribbean Food - Fremont	https://s3-media3....	FALSE	https://www.yelp.com/biz...
2	Gn5erxCRML47GgbGYdxzFA	bongos-seattle	Bongos	https://s3-media2....	FALSE	https://www.yelp.com/biz...
3	sjq3-lLJ-QYoHNejt62mYw	geos-cuban-and-creole-cafe-seattle	Geo's Cuban & Creole Cafe	https://s3-media4....	FALSE	https://www.yelp.com/biz...
4	C4j9EqCHRg2TdQVD3wE8EA	el-diablo-coffee-seattle-2	El Diablo Coffee	https://s3-media1....	FALSE	https://www.yelp.com/biz...
5	QJlYzcWkdrHhDyzwJi3blQ	mojito-seattle	Mojito	https://s3-media2....	FALSE	https://www.yelp.com/biz...
6	XXO8vKCSqB0cz0rVTg1Bjg	un-bien-seattle-seattle	Un Bien - Seattle	https://s3-media2....	FALSE	https://www.yelp.com/biz...
7	o2BJ-GAetKKTJ8yqz4Yl_Q	snout-and-co-seattle-2	Snout & Co.	https://s3-media3....	FALSE	https://www.yelp.com/biz...
8	ZHErhyY2p1xd7vcuTXvbwA	cafe-con-leche-seattle	Cafe Con Leche	https://s3-media2....	FALSE	https://www.yelp.com/biz...
9	rGWsX_7SDtgPYXF6subJ3w	paseo-caribbean-food-seattle-8	Paseo Caribbean Food	https://s3-media1....	FALSE	https://www.yelp.com/biz...

Figure 14.11 A subset of the data returned by a request to the Yelp Fusion API for Cuban food in Seattle.

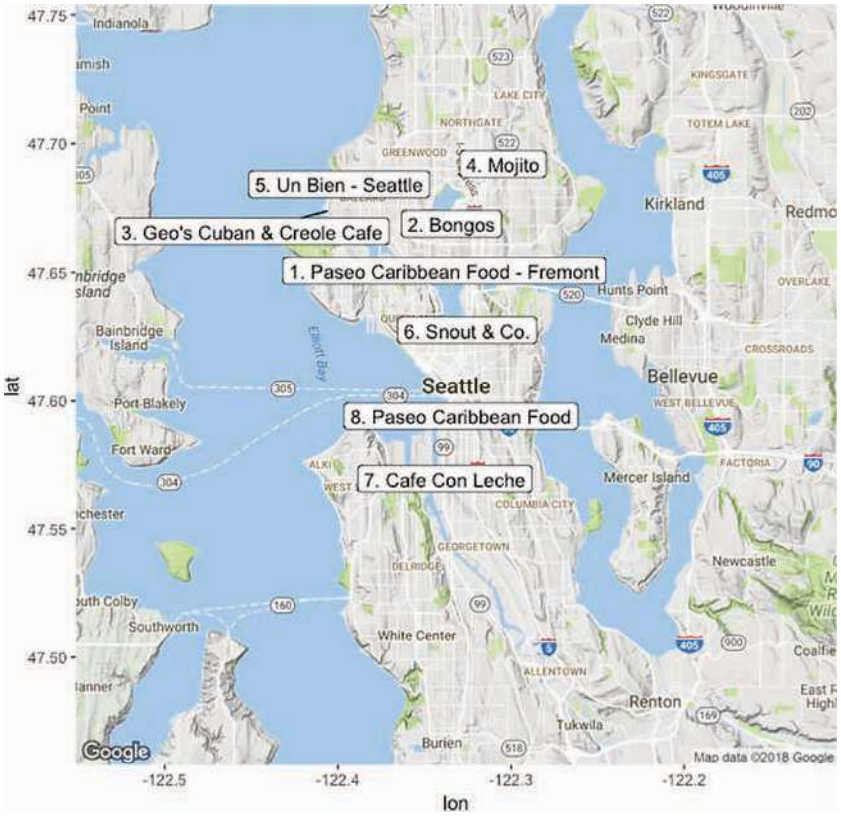


Figure 14.12 A map of the best Cuban restaurants in Seattle, according to the Yelp Fusion API.

```
# Create a base layer for the map (Google Maps image of Seattle)
base_map <- ggmap(get_map(location = "Seattle, WA", zoom =   ))   11

# Add labels to the map based on the coordinates in the data
base_map +
  geom_label_repel(
    data = response_data,
    aes(x = coordinates.longitude, y = coordinates.latitude, label = name_and_rank)
  )
```

Below is the full script that runs the analysis and creates the map—only 52 lines of clearly commented code to figure out where to go to dinner!

```
# Yelp API: Where is the best Cuban food in Seattle?
library("httr")
library("jsonlite")
library("dplyr")
library("ggrepel")
library("ggmap")

# Load API key (stored in another file)
source("api_key.R")

# Construct your search query
base_uri <- "https://api.yelp.com/v3/"
endpoint <- "businesses/search"
uri <- paste0(base_uri, endpoint)

# Store a list of query parameters
query_params <- list(
  term = "restaurant",
  categories = "cuban",
  location = "Seattle, WA",
  sort_by = "rating",
  radius = 8000
)

# Make a GET request, including your API key as a header
response <- GET(
  uri,
  query = query_params,
  add_headers(Authorization = paste("bearer", yelp_key))
)

# Parse results and isolate data of interest
response_text <- content(response, type = "text")
response_data <- fromJSON(response_text)

# Save the data frame of interest
restaurants <- flatten(response_data$businesses)

# Modify the data frame for analysis and presentation
restaurants <- restaurants %>%
  mutate(rank = row_number()) %>%
  mutate(name_and_rank = paste0(rank, ". ", name))

# Create a base layer for the map (Google Maps image of Seattle)
base_map <- ggmap(get_map(location = "Seattle, WA", zoom =   ))   11
```

```
# Add labels to the map based on the coordinates in the data
base_map +
  geom_label_repel(
    data = restaurants,
    aes(x = coordinates.longitude, y = coordinates.latitude, label = name_and_rank)
  )
```

Using this approach, you can use R to load and format data from web APIs, enabling you to analyze and work with a wider variety of data. For practice working with APIs, see the set of accompanying book exercises.[16]

[16] **API exercises:** https://github.com/programming-for-data-science/chapter-14-exercises