BIO8068 Data visualisation and management

Relational data

Roy Sanderson

## 1. Introduction

You will often have to deal with data that comes from multiple sources but that is linked in some way. For example, tables of species, plots, environmental data, survey sites, where there are links between certain common fields within each table. Another common example is that of GPS data, where you might have tables of animal positions, gps tags, time-stamps, animal characteristics (gender, age) etc. and you need to be able to organise them. The classic method if organising such data is in the form of a *relational database* in which the features of each table are strictly-enforced, so that no invalid data can accidentally be entered, and the tables all remain properly synchronised with each other. Commercial packages include Microsoft Access (either installed locally or on a server), and Oracle. Open source methods include MySQL and PostgreSQL, the latter being particularly useful for ecologists as it has an add-on PostGIS which integrates with QGIS to make the database spatially-aware. This can be useful for GPS data. All four types of relational database can be operated via a graphical user interface, to show the tables and their relationships, or more usually via Structured Query Languange (SQL). SQL has its own unique syntax, and there is not time to cover its use in-depth in this module. Fortunately, main of the relationships between tables can be undertaken directly within R, although as “referential integrity” is not enforced you need to be careful not to make incorrect queries of the data. The main aims of this practical are to:

* Introduce you to different types of relationships between tables in R, using a widely used non-ecological dataset (airline flights dataset)
* Demonstrate how to connect to a PostgreSQL server from within R using basic SQL commands, and retrieve data (GPS tracking of deer)
* Use dplyr functions to access and manipulate data from the GPS deer PostgreSQL server without the need for SQL syntax

## 2. Relationships between tables

You are already familiar with the concept of ‘tidy’ data, typically from a single table. With relational database you have multiple tables, and the relations are always defined in terms of the relationships between any two tables. Three or more tables are always a property of each pair of tables within the sets of tables as a whole.

To help you understand the concepts we are going to use dplyr, and data from the nycflights13 package. This is airline flight delays: obviously a non-ecological example, but you will see it in many textbooks, and it illustrates the key points. We will cover three broad types of ‘verbs’ to work with relational data:

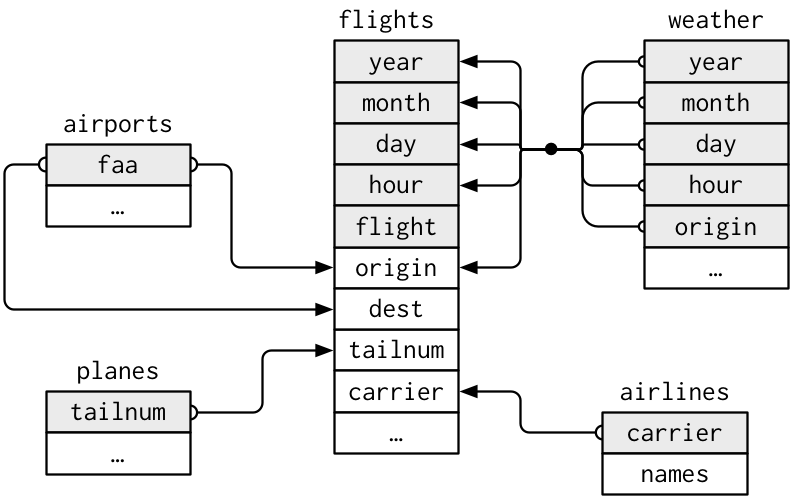
* **Mutating joins** add new variables to one data frame by matching observations in the other
* **Filtering joins** filter out observations from one data frame depending on whether they match an observation in the other data frame
* **Set operations** use set commands *and*, *both*, *not* etc. when comparing observations in tables

## 3. The nycflights13 package

Install the package in the usual way; it contains data on over 300,000 flights that departed New York County airports (including JFK airport) in 2013, so it is a big dataset. The data are in the form of tibbles, so are easy to examine:

library(dplyr)  
library(nycflights13)  
flights  
airlines  
airports  
planes  
weather

Hopefully you can see that there are common columns in all 5 tables. How do they link together?



This is quite complex, but shows that \* flights connects to planes via tailnum \* flights connects to airlines via carrier \* flights connects to airports via both origin and dest (destination) \* flights connects to weather via origin as well as date/time variables

**Question**: if you wanted to work out the route each plane takes from its origin to its destination, what tables and variables would be needed?

## 4. Relationship keys

A *key* concept to understand with relational data is that of the **key** . It is a variable that uniquely identifies and observation in a dataset. So each aeroplane is uniquely identified by tailnum. Sometimes multiple variables will be needed, **Question**: how many (and which) variables are needed to identify a unique observation in the weather data frame?

There are two types of key: \* **Primary key** uniquely identifies each observation (row) in its own table. Here planes$tailnum uniquely identifies each aeroplane in the planes data frame. \* **Foreign key** uniquely identifies an observation in another table. So flights$tailnam is foreign as it matches each flight to a unique plane, although that plane may fly multiple times.

To check that you’ve correctly identified your primary key, make sure that there is only one observation for each primary key:

# Check number of observations  
planes %>%   
 count(tailnum)

## # A tibble: 3,322 x 2  
## tailnum n  
## <chr> <int>  
## 1 N10156 1  
## 2 N102UW 1  
## 3 N103US 1  
## 4 N104UW 1  
## 5 N10575 1  
## 6 N105UW 1  
## 7 N107US 1  
## 8 N108UW 1  
## 9 N109UW 1  
## 10 N110UW 1  
## # … with 3,312 more rows

# Obviously too many to check manually, so make sure none greater than 1  
planes %>%  
 count(tailnum) %>%   
 filter(n > 1)

## # A tibble: 0 x 2  
## # … with 2 variables: tailnum <chr>, n <int>

Try and repeat the exercise for the airlines table. Some tables in this package do not have an explict unique primary key, for example the flights table, even with year, month, day, flight:

flights %>%   
 count(year, month, day, flight) %>%   
 filter(n > 1)

## # A tibble: 29,768 x 5  
## year month day flight n  
## <int> <int> <int> <int> <int>  
## 1 2013 1 1 1 2  
## 2 2013 1 1 3 2  
## 3 2013 1 1 4 2  
## 4 2013 1 1 11 3  
## 5 2013 1 1 15 2  
## 6 2013 1 1 21 2  
## 7 2013 1 1 27 4  
## 8 2013 1 1 31 2  
## 9 2013 1 1 32 2  
## 10 2013 1 1 35 2  
## # … with 29,758 more rows

Sometimes such issues can be resolved by artificially creating a unique id for each row, or “surrogate” key, by using mutate() with row\_number(). Have a go at creating a column for the flights table called surrogate\_key that simply contains the row numbers.

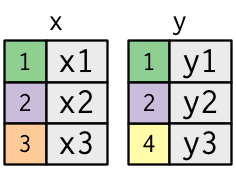
## 5. Mutating joins

This type of join combines variables from two tables, adding variables to the right. If you already have a lot of variables in a table, it might be easiest to use the View() function to display what’s going on in RStudio. To make it easier to see what’s happening, we’ll create a flights2 table with just a few variables, then add the full airline carrier name rather than just the code, via a mutating join:

## # A tibble: 336,776 x 8  
## year month day hour origin dest tailnum carrier  
## <int> <int> <int> <dbl> <chr> <chr> <chr> <chr>   
## 1 2013 1 1 5 EWR IAH N14228 UA   
## 2 2013 1 1 5 LGA IAH N24211 UA   
## 3 2013 1 1 5 JFK MIA N619AA AA   
## 4 2013 1 1 5 JFK BQN N804JB B6   
## 5 2013 1 1 6 LGA ATL N668DN DL   
## 6 2013 1 1 5 EWR ORD N39463 UA   
## 7 2013 1 1 6 EWR FLL N516JB B6   
## 8 2013 1 1 6 LGA IAD N829AS EV   
## 9 2013 1 1 6 JFK MCO N593JB B6   
## 10 2013 1 1 6 LGA ORD N3ALAA AA   
## # … with 336,766 more rows

## # A tibble: 336,776 x 7  
## year month day hour tailnum carrier name   
## <int> <int> <int> <dbl> <chr> <chr> <chr>   
## 1 2013 1 1 5 N14228 UA United Air Lines Inc.   
## 2 2013 1 1 5 N24211 UA United Air Lines Inc.   
## 3 2013 1 1 5 N619AA AA American Airlines Inc.   
## 4 2013 1 1 5 N804JB B6 JetBlue Airways   
## 5 2013 1 1 6 N668DN DL Delta Air Lines Inc.   
## 6 2013 1 1 5 N39463 UA United Air Lines Inc.   
## 7 2013 1 1 6 N516JB B6 JetBlue Airways   
## 8 2013 1 1 6 N829AS EV ExpressJet Airlines Inc.  
## 9 2013 1 1 6 N593JB B6 JetBlue Airways   
## 10 2013 1 1 6 N3ALAA AA American Airlines Inc.   
## # … with 336,766 more rows

You may have notice the function left\_join above. These very simple datasets show the concepts clearly. Imagine two small data frames, each with two columns, the first a key:



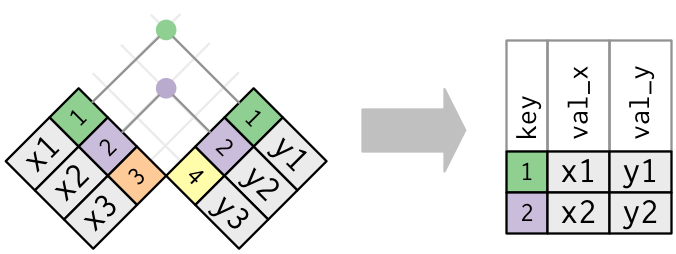
You can easily create these two data frames in R:

x <- tribble(  
 ~key, ~val\_x,  
 1, "x1",  
 2, "x2",  
 3, "x3"  
)  
y <- tribble(  
 ~key, ~val\_y,  
 1, "y1",  
 2, "y2",  
 4, "y3"  
)

The colours in the ‘key’ column show where the variables match up (1 and 2) and where they don’t match (3 and 4). We can show the matches with dots linking the related keys, and output table.

### 5.1 Inner join

This is a simple join; you’ll recall only keys 1 and 2 matched up in our example:



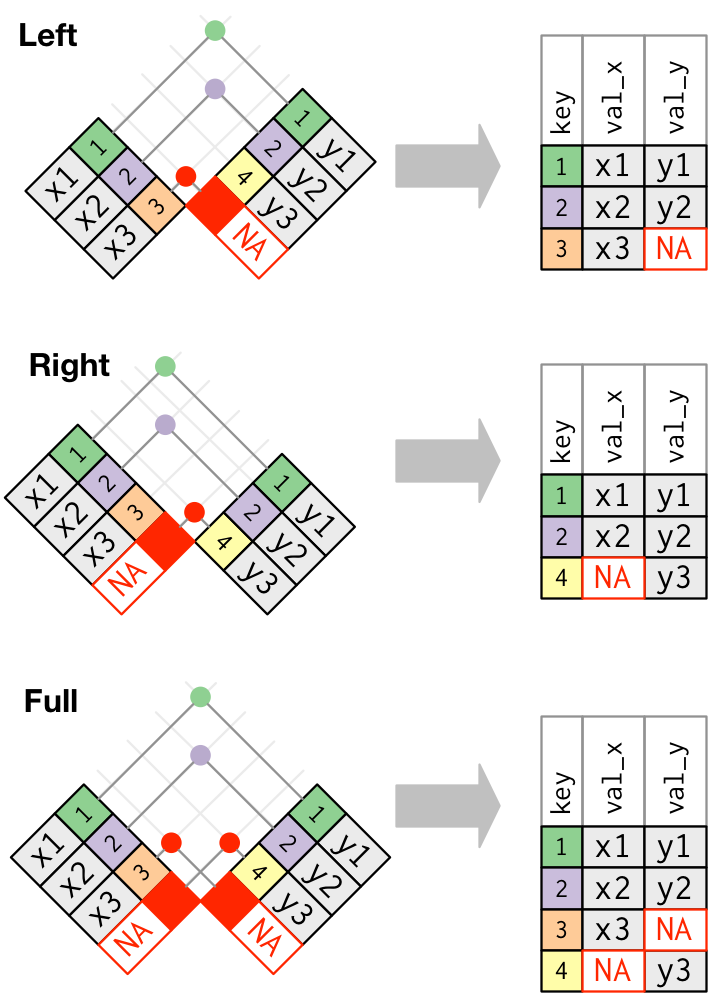
Only the matched rows are returned, no unmatched rows come back. In dplyr this is:

x %>%   
 inner\_join(y, by = "key")

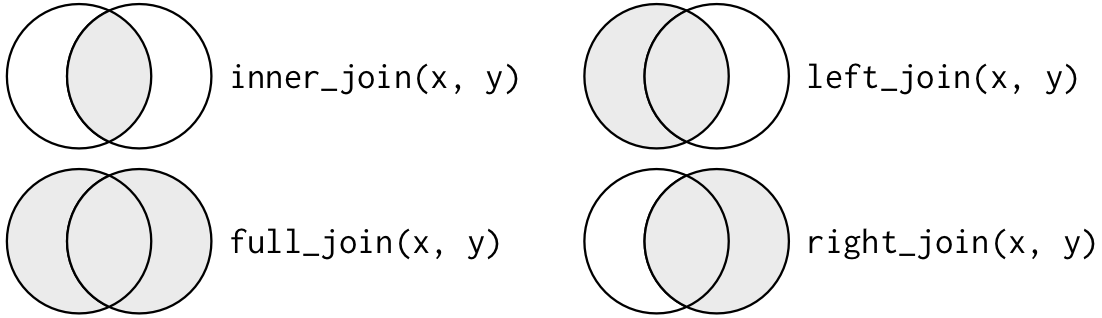
## # A tibble: 2 x 3  
## key val\_x val\_y  
## <dbl> <chr> <chr>  
## 1 1 x1 y1   
## 2 2 x2 y2

### 5.2 Outer joins

Outer joins keep all the observations from at least one of the tables, with the result that NA’s appear in the outpus. The left\_join used above is most common:



Another way of viewing the same types of join is through a Venn diagram:



**Question**: Try doing a full join on the example x and y data frames.

**Note**: The joining variable above is specified with the word by. If this is omitted by = NULL and the join will be made on all variables with the same name. It is possible to make joins on multiple variables.

### 5.3 Filtering joins

These match observations in a similar way to mutating joins, and are of two types:

* semi\_join(x, y) keeps all observations in x that have a match in y
* anti\_join(x, y) omits all observations in x that have a match in y

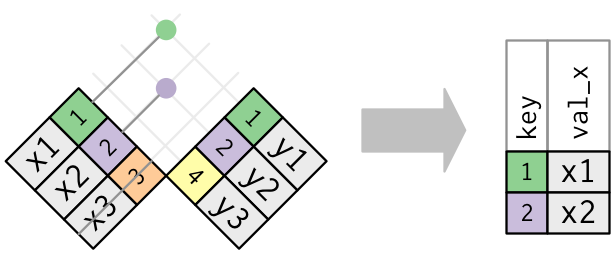
Semi-joins are handy when producing summaries of data from two tables. For example, if you want to know the dates and times just for the 10 most popular flight destinations, first create a table top\_dest, then do a semi-join on the flights table to extract all their dates:

## # A tibble: 10 x 2  
## dest n  
## <chr> <int>  
## 1 ORD 17283  
## 2 ATL 17215  
## 3 LAX 16174  
## 4 BOS 15508  
## 5 MCO 14082  
## 6 CLT 14064  
## 7 SFO 13331  
## 8 FLL 12055  
## 9 MIA 11728  
## 10 DCA 9705

## Joining, by = "dest"

## # A tibble: 141,145 x 19  
## year month day dep\_time sched\_dep\_time dep\_delay arr\_time  
## <int> <int> <int> <int> <int> <dbl> <int>  
## 1 2013 1 1 542 540 2 923  
## 2 2013 1 1 554 600 -6 812  
## 3 2013 1 1 554 558 -4 740  
## 4 2013 1 1 555 600 -5 913  
## 5 2013 1 1 557 600 -3 838  
## 6 2013 1 1 558 600 -2 753  
## 7 2013 1 1 558 600 -2 924  
## 8 2013 1 1 558 600 -2 923  
## 9 2013 1 1 559 559 0 702  
## 10 2013 1 1 600 600 0 851  
## # … with 141,135 more rows, and 12 more variables: sched\_arr\_time <int>,  
## # arr\_delay <dbl>, carrier <chr>, flight <int>, tailnum <chr>,  
## # origin <chr>, dest <chr>, air\_time <dbl>, distance <dbl>, hour <dbl>,  
## # minute <dbl>, time\_hour <dttm>

Graphically, you can view it as:



Anti-joins are less widely used, but can be useful to identify errors in the data.

## 6. Structured Query Language (SQL)

Many of the commands we have covered already have direct equivalents in SQL. For example, for the mutating joins:

|  |  |
| --- | --- |
| dplyr | SQL |
| inner\_join(x, y, by = “z”) | SELECT \* FROM x INNER JOIN y USING (z) |
| left\_join(x, y, by = “z”) | SELECT \* FROM x LEFT OUTER JOIN y USING (z) |
| right\_join(x, y, by = “z”) | SELECT \* FROM x RIGHT OUTER JOIN y USING (z) |
| full\_join(x, y, by = “z”) | SELECT \* FROM x FULL OUTER JOIN y USING (z) |

Obviously, handling SQL is more complicated that standard dplyr calls, but sometimes you may have no choice but to use them. I have setup a small PostgreSQL database gps\_tracking\_db on our Linux server mach-252.ncl.ac.uk which you can access with the username basic\_user and password basic\_user. Admittedly, this is not a very secure username or password, but I’m trusting you not to hack, and you should have read-only access. The structure of the full database is actually very complicated, as it has to account for the risk that GPS tags might become detached from a deer, the same animal might therefore be tracked by different GPS receivers at different times etc. The example is taken from Chapters 2 and 3 of *Spatial Database for GPS Wildlife Tracking data: a practical guide to creating a data management system with PostgreSQl/PostGIS and R* by Urbano and Cagnacci (2014). For simplicity, we will just focus on the tables main.animals with the names etc. of individual deer, their lu\_tables.lu\_species table, and lu\_tables.lu\_age\_class plus the tables of main.gps\_data and main.gps\_sensors.

## 6.1 SQL from within RStudio

First, install and load the DBI and `RPostgreSQL’ packages. You then need to open a connection to the server, with your username and password:

con <- dbConnect(drv = dbDriver("PostgreSQL"),  
 dbname = 'gps\_tracking\_db',   
 host = 'mach-252.ncl.ac.uk', # Name of PostgreSQL server  
 port = 5432, # default for PostgreSQL DBA  
 user = 'basic\_user',  
 password = 'basic\_user')

Now that you have created your connection con, first list the tables available:

dbListTables(con)

## [1] "lu\_age\_class" "lu\_species" "gps\_data" "gps\_sensors"   
## [5] "animals"

you can issue an SQL SELECT command, and then fetch the results. SQL commands are always in upper-case, and are typically of the form

SELECT <some column(s)> FROM <database(s)> WHERE <various constraints>;

**Note** There is a ; at the end of the SQL command. First, let’s select everything from the main.animals table and look at it; the SELECT \* option means “all columns in the table”

res <- dbSendQuery(con, "SELECT \* FROM main.animals;")  
animals <- fetch(res)  
animals

## animals\_id animals\_code name sex age\_class\_code species\_code note  
## 1 1 F09 Daniela f 3 1 <NA>  
## 2 2 M03 Agostino m 3 1 <NA>  
## 3 3 M06 Sandro m 3 1 <NA>  
## 4 4 F10 Alessandra f 3 1 <NA>  
## 5 5 M10 Decimo m 3 1 <NA>  
## insert\_timestamp  
## 1 2019-03-09 21:13:26  
## 2 2019-03-09 21:13:26  
## 3 2019-03-09 21:13:26  
## 4 2019-03-09 21:13:26  
## 5 2019-03-09 21:13:26

Download the data from the lu\_tables.lu\_age\_class and lu\_tables.lu\_species in a similar way. Notice how the animals have codes for their species and ages, and it would be better to link them with the actual text. We can do this in SQL, which is analagous to an inner join but on multiple tables. This is hard to do in R which only handles pairs of tables.

# More comprehensive query linking tables  
res <- dbSendQuery(con,  
 "SELECT  
 animals.name,  
 animals.sex,  
 lu\_age\_class.age\_class\_description,  
 lu\_species.species\_description  
 FROM  
 lu\_tables.lu\_age\_class,  
 lu\_tables.lu\_species,  
 main.animals  
 WHERE  
 lu\_age\_class.age\_class\_code = animals.age\_class\_code  
 AND  
 lu\_species.species\_code = animals.species\_code;")  
animal\_names\_spp\_ages <- fetch(res)  
animal\_names\_spp\_ages

## name sex age\_class\_description species\_description  
## 1 Daniela f adult roe deer  
## 2 Agostino m adult roe deer  
## 3 Sandro m adult roe deer  
## 4 Alessandra f adult roe deer  
## 5 Decimo m adult roe deer

This is easier to read. We can take the SQL commands a step further, and rename some of the columns using the AS function as we read them from the database:

res <- dbSendQuery(con,  
 "SELECT  
 animals.animals\_id AS id,  
 animals.animals\_code AS code,  
 animals.name,  
 animals.sex,  
 lu\_age\_class.age\_class\_description AS age\_class,  
 lu\_species.species\_description AS species  
 FROM  
 lu\_tables.lu\_age\_class,  
 lu\_tables.lu\_species,  
 main.animals  
 WHERE  
 lu\_age\_class.age\_class\_code = animals.age\_class\_code  
 AND  
 lu\_species.species\_code = animals.species\_code;")  
animal\_names\_spp\_ages <- fetch(res)  
animal\_names\_spp\_ages

## id code name sex age\_class species  
## 1 1 F09 Daniela f adult roe deer  
## 2 2 M03 Agostino m adult roe deer  
## 3 3 M06 Sandro m adult roe deer  
## 4 4 F10 Alessandra f adult roe deer  
## 5 5 M10 Decimo m adult roe deer

Sketch out the relationships between the three tables animals, lu\_age\_class and lu\_species. What are the primary keys? Finally, once you have finished working with the database, remember to close the connection:

# Clear the last result  
dbClearResult(res)

## [1] TRUE

# Disconnect from the database  
dbDisconnect(con)

## [1] TRUE

## 6.2 Accessing PostgreSQL from dplyr

If you are going to do a lot of database work, it is worth learning how to use SQL. The dbplyr package is supposed to allow direct access to SQL databases, although I will admit having trouble configuring it. Sometimes you may find it simpler to use a general SELECT to gather all the data from a table, then manipulate it within R. The following commands (theoretically!) allow you to use dbplyr but they do not work with our PostgreSQL database:

library(dbplyr)  
species\_tbl <- tbl(con, "lu\_tables.lu\_species")  
species\_tbl

A word of caution. One weakness of R is that it has poor memory management, and SQL is generally better for huge databases. If you want to stick with R, and simply pull tables back from SQL using SELECT, you may want to learn about the data.table package. This is similar to some respects to the tidyverse packages, although not as well integrated. Its main advantage is that it copes with very large databases and tables rapidly.