Encounters with (Big) Data

Chris Park

30 July, 2017

Objectives

The aim of today's workshop is to get hands-on experience in dealing with *reasonably large* data sets using R. By the end of the workshop, you should be able to:

- Read and write simple programs in the R language,
- Manipulate tabular data sets in R using the `dplyr` package,
- Visualize tabular data sets in R using the `ggplot2` package,
- Understand R's limitations when it comes to large data sets, and
- Manipulate *reasonably large* tabular data sets in R using the `sparklyr` package.

Getting Started

We will be using R Notebooks to cover the course material, so that code is interleaved with explanations (which hopefully makes it easier to understand code). These are like Zeppelin Notebooks, but tailor-made for R. Blocks of code within an R Notebook is called a *chunk*, and is visually marked by a grey rectangle. To run the code inside a chunk in RStudio, you can either:

- Click the *Run* button (small green "play" button), or
- Place your cursor inside the chunk and press *Ctrl + Shift + Enter*.

```r
print("Welcome to the 2017 Essex Big Data and Analytics Summer School!")
```

It is also possible to insert a new code chunk. To do this, you can either:

- Click the *Insert Chunk* button, or
- Place your cursor inside the chunk and press *Ctrl + Alt + I*.

Introduction to Programming in R

A *computer program* is simply a sequence of instructions that describe how we want to perform computations, e.g. add numbers, display messages, and plot graphs. There are many *programming languages* we can choose from to express our computations to the computer. Today, we use the R programming language to express our computations, as it is one of the most widely used languages when it comes to computing with tabular data sets. Using a language to describe our data processing tasks can be very useful:

- More flexibility for handling edge cases e.g. inconsistent formatting,
- Easier to share and reproduce work,
- Automate repetitive tasks instead of copying and pasting multiple times,
- Access to cutting-edge tools developed by the software community,
- etc.

Unlike humans however, computers can't handle ambiguity, e.g. as high as a kite. They are harsh taskmasters, and don't tolerate inaccuracies or inconsistencies in instructions. This means that the programs we write must be 100% accurate, and in the *form* that the computer expects. This domain has a big vocabulary so we're going to have to learn a few words:

Term	Definition
Variable	Containers for values that can change, i.e. are *variable.*
Expression	Combination of values, variables, and operators.
Assignment	A statement that *assigns* a *value* to a *variable.*
Comment	Notes for humans to read and understand a program.
Function	A *named* sequence of statements that performs a computation.

For example, in the **expression** below, we say that we are **assigning** the numeric value `1` to the **variable** `a`, rather than `a` *equals* `1`:

```
a = 1
```

Did anything happen? We can check by typing in `a` to see if `R` returns the value `1`.

```
a # Click "Run" or the green triangle or hit "Ctrl + Enter"
```

Now, since `a` is a *variable*, we can replace its value by assigning it a new value.

```
a = "Boo!"
a # 'a' no longer returns the value 1.
```

Note: the sentence `'a' no longer returns the value 1.` in the code above what is called a *comment.* It is just a helpful message for human readers, and ignored by the computer. Any text that follows `#` becomes a *comment* in R. Different programming languages have different symbols that mark the beginning of a comment, e.g. `//` in C and C++.

Containers and Values

R was written by statisticians, for statisticians. Statisticians often work with *collections* of values rather than individual values, e.g. a survey of annual income from a region rather than a single salary from an individual. For this reason, R was designed to be *collection-oriented*, and the basic data type in R is a `vector`, which is a kind of container that holds collection of values that are all of the *same type.* Broadly speaking, there are four basic *families* of *values* in R:

- Logical: `TRUE`, `FALSE`
- Numeric: `1`, `2`, `2.0`, `3.0`, ...
- Character: `"Alice"`, `'Bob'`, `"Cat!"`
- Missing: `NA` (Stands for `Not Available`)

The four basic types of *containers* for values are:

- Vectors: 1-dimensional collection of itmes which are of the *same* type.
- Matrices: 2-dimensional collections of items which are of the *same* type.
- Lists: tree-like collection of items where each *item* can be *different.*
- Data Frames: spreadsheet-like collection of items where each *column* can have *different* types of values.

Aside: Recall that `R` is a *collection-oriented* language designed for statistical computations on *collections* of values, and the basic data type is a `vector`. This means that even a single numeric value like `1` is in fact a `vector` of length 1, rather than a scalar. That is, `1` is identical to `c(1)` in R. This can have some performance implications in complex computations, as `R` needs to extract the value in and out of a container (i.e. the `vector`) rather than use it directly. This is not the case in most other programming languages.

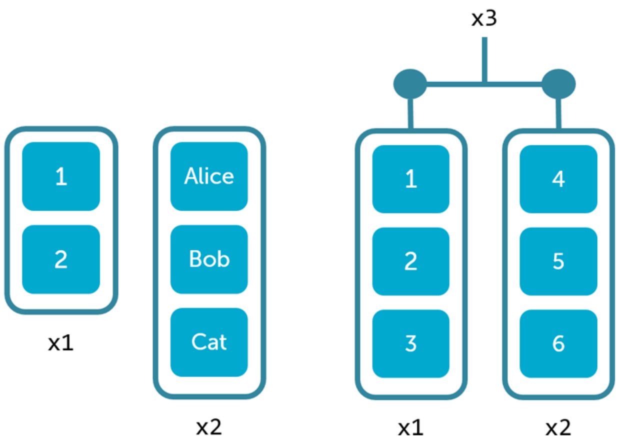

Figure 1: A vector (left) and matrix (right)

Vectors

To create a vector, we can combine or concatenate values as follows:

```
## Numeric Vector
c(1, 2)
1:2

## Character Vector
c("Alice", "Bob", "Cat")
c("A", "B", "C")
LETTERS[c(1, 2, 3)]
LETTERS[1:3]
letters[1:3]

## Logical Vector
c(TRUE, FALSE)
c(1, 2)[c(TRUE, FALSE)]

## In fact, even the single numeric value 1 is a vector.
1
is.vector(1)

## Exercise: Explain what is going on in the code below:
c(1, "A")
c(1, 2, 3, 4)[c(TRUE, FALSE)]
```

Matrices

```
## Numeric Matrix
i = 1:6
matrix(i, nrow = 3)
matrix(i, nrow = 3, ncol = 2)
matrix(i, ncol = 2)
cbind(1:3, 4:6)                 # column bind
rbind(c(1, 4), c(2, 5), c(3, 6)) # row bind
```

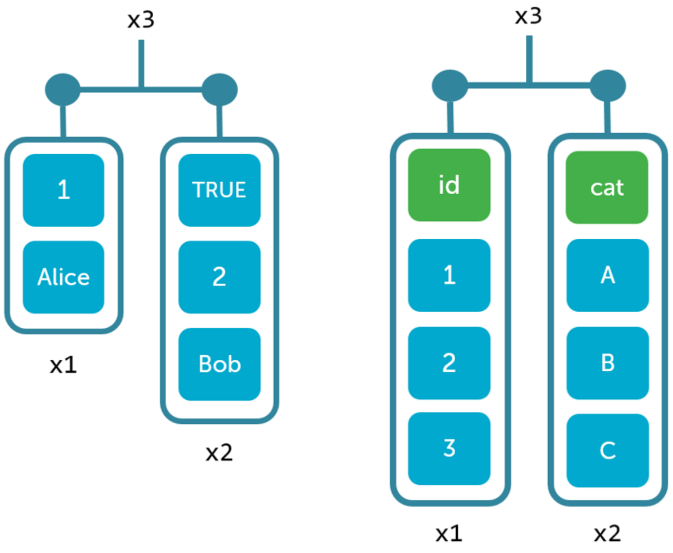

Figure 2: A list (left) and data frame (right)

```
## Character Matrix
matrix(LETTERS[i], nrow = 3)
```

Lists

```
x1 = list(1, "Alice")
x2 = list(TRUE, 2, "Bob")
x3 = list(x1, x2)
x3
```

Data Frames

```
x1 = 1:3
x2 = LETTERS[x1]
x3 = data.frame(id = x1, cat = x2)
data.frame(x3, x3)
data.frame(x3, x3, x3)
data.frame(list(a = x1, b = x2))
```

Checking Types

```
is.numeric(1:10)
is.numeric(c("A", "B"))
is.data.frame(matrix(1:10, nrow = 2))
is.matrix(data.frame(1:10, 2:11))
is.list(matrix(1:10, ncol = 2))

## Exercise: is a data.frame a list?

## Exercise: is a list a data.frame?
```

Functions

Functions allow us to avoid writing the same chunk of code over and over again. We can use them to package up a group of statements into a single bundle, and give it a name of our choice. It is good practice to give it a name that is related to what it was written for. For example, the `print()` function prints a message on screen. Suppose we wanted to create a kind of greeting system to `print()` the following greeting message to students.

```
## Print a greeting message for summer school students.
print("Hello, Peter! Welcome to the 2017 Essex BDAS!")
print("Hello, Chris! Welcome to the 2017 Essex BDAS!")
print("Hello, Sarah! Welcome to the 2017 Essex BDAS!")
```

Suppose we wanted to `print()` the message to 200 students. We will then have to repeat ourselves an awful lot, as the only portion of the code that needs changing is the name of each student. It would be very useful to have a way of simply slotting in a student's name into the message. This is exactly what a `function` allows us to do:

```
## We define a function called "greet_student" that prints a friendly
## welcome message to 2017 Essex BDAS students. It takes a single argument
## called "name", which should be a character vector.
greet_student = function(name) {
  print(paste0("Hello, ",
               name,
               "! ",
               "Welcome to the 2017 Essex BDAS!"))
}

## "sapply()" applies a function to each element in a vector.
student_names = c("Simon", "Jack", "Helen", "Alex")
sapply(student_names, greet_student)
```

Quirks

Flexibility vs Speed

R is a very *flexible* language, and the value of almost anything can be modified on the fly. This means that R has to look up the meaning of a value every time it performs a computation. Imagine you're reading a book, and the words on the page are constantly changing into a different langauge; take the word "zero" for example: zero (English), nulla (Hungarian), null (Norwegian), sifir (Turkish), ling (Chinese), etc. This is going to slow down your reading speed as you would have to look up the meaning of every word as you read. This is kind of what R is doing in the background.

In general, there is a speed-flexibility trade-off when it comes to programmming languages: inflexible languages tend to be faster, while flexible languages tend to be slower. For example, in an inflexible language like C or C++, you have to specify the types of every value you use in a program. For example, suppose we wanted to compute the column sums of a matrix.

```
library(Rcpp)

## Observe that we declare the variable and function types.
cppFunction('NumericVector colSums_C(NumericMatrix x) {
  int nrow = x.nrow(), ncol = x.ncol();
  NumericVector out(ncol);
  for (int j = 0; j < ncol; j++) {
    double total = 0;
    for (int i = 0; i < nrow; i++) {
      total += x(i, j);
    }
    out[j] = total;
  }
  return out;
}')

colSums_R = function(x) {
  n = ncol(x)
  out = numeric(n)
  for (j in 1:n) {
    out[j] = sum(x[, j])
  }
  out
}

## Simulate a 10 million by 2 matrix of random normals.
A = cbind(rnorm(1e8), rnorm(1e8))

## Compare times.
system.time(colSums_R(A))
# user  system elapsed
# 1.90    0.14    2.05

system.time(colSums_C(A))
# user  system elapsed
# 0.32    0.00    0.33
```

As we can see above, we don't need to declare types at all with R – we can simply define things on the fly; but this flexibility comes at the cost of speed because R is constantly looking things up in its dictionary, so it's doing a lot of work in the background. This kind of behaviour can also lead to obscure inconsistencies in your code and make it difficult to debug programs (this becomes particularly problematic with larger datasets).

```
c(1, 2, 3, 4)

c = sum
c(1, 2, 3, 4)

rm(c)
```

We can even do crazy things like:

```
2 * 2
2 / 2

`*` = `/`
2 * 2
2 / 2

rm(`*`)
```

Things can get even more interesting/confusing when you take into account *how* R looks up the meaning of things: R looks up the meaning of words from the inner-most dictionary first. This is what happens in the code below - defining a new *function* results in a new *inner dictionary* from which to look up the meaning of objects from.

Figure 3: A book within a book to depict R's scoping rules

```
f = function(n) {
  `*` = `/`
  n * n
}
## Exercise: What is happening in the code below?
f(2)
```

```
## [1] 1
```

```
2 * 2
```

```
## [1] 4
```

Copying

R also makes a lot of copies of objects. This seems to be historical baggage from the early days of statistical software, when software was developed on machines with very limited RAM (e.g. 32KB – compare that to 32GB of RAM on my machine, which is 32 million KB!). This meant that data couldn't fit into memory so small chunks of the data had to be *copied* into memory at each stage of the analysis. Unfortunately, this can lead to lots of unnecessary copying, which can be particularly problematic with big data sets, especially if they are stored in inefficient data structures like data frames. Data frames can cause problems as even a simple update leads to multiple copies of the data frame being made.

```
## We create a data frame consisting of 2 million random integers.
## Imagine that these are counts the number of steps people have
## taken across two days.
day1 = sample(9000:20000, 1e6, replace = TRUE)
```

```
day2 = sample(9000:20000, 1e6, replace = TRUE)
steps = data.frame(day1, day2)
head(steps)

## Suppose that there was a fault in the collection device and we need
## to increment the first 1000 counts for both days.
m = 1000

## Data frame way.
system.time({
  for (i in 1:m) {
    steps[i, ] = steps[i, ] + 1
  }
})
# user  system elapsed
# 10.35    3.78   14.22

## Vector way. This offloads the update process down to C code, so
## the copying is under control.
system.time({
  for (i in 1:m) {
    day1[i] = day1[i] + 1
    day2[i] = day2[i] + 1
  }
  steps2 = data.frame(day1, day2)
})
# user  system elapsed
# 0.02    0.00    0.01
#
# 14.22 / 0.01 ~ 1400

## Just like that, we achieved almost a 1500-fold increase in speed. While
## this may seem like a trivial exercise, its implications are quite
## profound: having an understanding of how R works under the hood can save
## you a lot of heartache (and money) down the line.
# identical(steps, steps2)
```

Big R

R has a physical limit to the size of data sets it can handle. This is because it requires that the data set of interest fits into RAM all at once (note: A 32-bit OS can't address more than 4GB of memory, and only a fraction of this is made available to R. With 64-bit machines, it all depends on how much money you're willing to spend on buying additional RAM). We can view our memory limit, in MBs, as follows:

```
memory.limit()
```

Let's try and hit our memory limit by simulating a large matrix.

```
## Simulate a large matrix.
zero_mat = matrix(0, 1e8, ncol = 10)
dim(zero_mat)
print(object.size(zero_mat), unit = "Gb")
```

This time, we simulate a large matrix of *integer* values.

```r
## Simulate a large integer matrix.
int_mat = matrix(as.integer(0), 1e+8, ncol = 10)
dim(int_mat)
print(object.size(int_mat), unit = "Gb")
```

Why is the `integer` matrix so much smaller? This is because `numeric` values take up 8 bytes by default, whereas `integer` values only take up 4. This means that if we know we only need to handle `integer` values, specifying this at the beginning when we create a data structure to hold these, can save a lot of memory.

Observe that we've ended up repeating ourselves a lot. If we wanted to experiment with different types of simulated matrices, it would be convenient to have a function that does this for us. Writing functions can be a bit tricky in the beginning, but it is important to get used to this way of thinking. Writing *pseudocode* can help you get started.

```
sim_mat =
  function(type, val, nrow, ncol, unit) {
    <<coerce values to specified type>>
    <<simulate large matrix with specified dims>>
    <<print the memory footprint in specified unit>>
  }
```

It is the conceptual deconstruction reflected in the pseudocode above that is most important. Translating this into a programming language (say, `R`) is largely a trivial task.

```r
## Write the "sim_mat" function. Notice that we provide default values for
## all the arguments, for convenience.
sim_mat =
  function(type = "numeric", val = 0, nrow = 1e+8, ncol = 10, unit = "Gb") {
    type_val = as(val, type)
    mat = matrix(type_val, nrow, ncol)
    print(object.size(mat), unit = unit)
  }

## We call the sim_mat function with different types of values.
sim_mat("numeric")
sim_mat("integer")
```

Now, let's really try and reach out memory limit.

```r
sim_mat(ncol = 1000)
```

Parallel Processing

Now, recall that we talked a bit about *distributed* processing and Map Reduce earlier (in our slides. The basic idea is to break down a big task into small chunks (a bit like programming itself), perform operations on each chunk in parallel (e.g. `Map()`), then combine the result (e.g. `Reduce()`). We can simulate this behaviour within `R` as follows (note: the example below is over-simplified and strictly for demonstration purposes only. In practice, it would make no sense to do this kind of splitting and combining with such a small vector, and for an operation as simple as adding up a bunch of numeric values).

```r
## Split the numeric vector `x` into 4 chunks.
x = 1:100
x_split = split(x, 1:4)
x_split
```

```r
## Compute the sum of each chunk, by "mapping" the function "sum" to each
## chunk in "x_split".
```

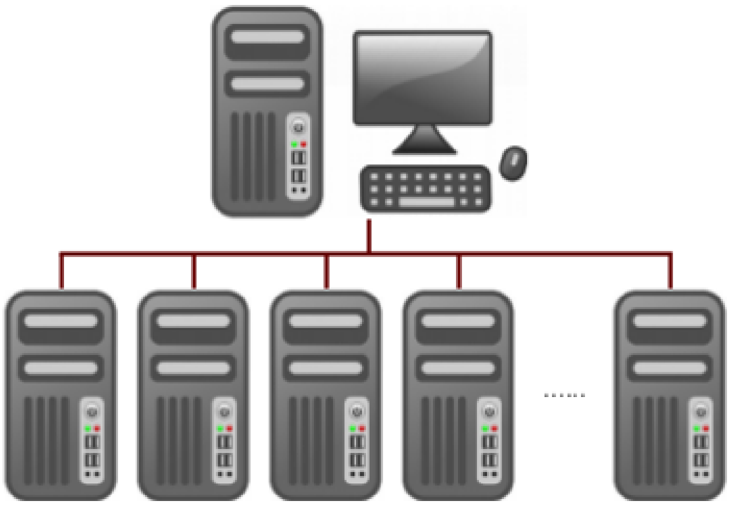

Figure 4: Parallel Processing

```
comp = Map(sum, x_split)
comp

## Combine results, i.e. "Reduce".
Reduce(sum, comp)

## Identical to:
sum(x)
```

Note that we have used multiple variables to store *intermediate results* (e.g. `x_split`, `comp`). It can sometimes be useful to be able to *chain together* expressions, using the *pipe operator* %>%. This operator *pipes* the result of the current expression into another expression. This can make code easier to follow and create less variable clutter. We will be making heavy use of this operator in our workshop today.

```
## Version 1: Intermediate variables
x_split = split(x, 1:4)
comp = Map(sum, x_split)
Reduce(sum, comp)
```

```
## Version 2: Nested calls
Reduce(sum, Map(sum, split(x, 1:4)))
```

```
## Version 3: Pipe operator; arguably more compact and easier to follow.
library(dplyr)

x %>%
  split(., 1:4) %>%  # "." is a placeholder for "x".
  Map(sum, .) %>%
  Reduce(sum, .)

## Another benefit is that other mainstream programming languages used in
## data science read similarly. For example, you just have to replace
## `%>%` with say a `.` operator in Python.
```

We can also use the pipe operator within functions. Suppose we wanted to generalize this Map Reduce example, so that we can split the data into more chunks.

```r
sum_MR = function(x = 1:100, n = 4) {
  x %>%
    split(., 1:n) %>% {
      ## Curly braces can be used to group 1+ statements.
      ## Here we print the split dataset for confirmation.
      print(.)
      Map(sum, .)
    } %>%
    Reduce(sum, .)
}

sum_MR()
```

```r
sum_MR(n = 5)
```

The above was *simulating* parallel processing to help visualize the process. R also parallel computing capabilities.

```r
library(parallel)

## How many CPU cores are avalable?
nc = detectCores() - 1

## Create a local cluster of processing nodes.
cl = makeCluster(nc)

## Split data into partitions (~ same number in each)
pt = clusterSplit(cl, 1:1e8)

## Parallel sapply.
system.time(parSapply(cl, pt, function(x) x^2))
# user  system elapsed
# 1.47    1.17    3.23

## Serial sapply
system.time(sapply(pt, function(x) x^2))
# user  system elapsed
# 0.30    0.03    0.33

## Return resources to the OS.
stopCluster(cl)
```

Note that running in parallel doesn't always lead to faster execution times. Sometimes, the CPU is multitasking on other processes or has to wait for system resources like network conenctions to become available. More importantly:

- Communication: data needs to be transferred back and forth between processes.
- Collision: processes can get in each other's way when we try and access the same data at the same time.
- Load balancing: work needs to be divided fairly across the processes. Otherwise, some processes may remain idle and unproductive when there is still lots of work to be done.

All of this adds overhead. Suppose you were tasked with mowing a large lawn. You could get a few of your friends/kids to help you out, but it wouldn't make much sense to ask for help if the lawn is small enough for you to mow yourself. In this case, it might be better to invest in a better lawn mower. Alternatively, having 2 or 3 friends help for a reasonably large lawn might make sense, but having 10 won't - you might keep running into each other, it will be hard to communicate who is doing which section of the lawn and keep track of each other's progress, etc.

Data Manipulation with `dplyr`

Disclaimer: figures and exercises in this section are adapted from Modern Data Science with R by Baumer et al. and R for Data Science by Wickham & Grolemund.

Now that we're somewhat familiar with the R environment, we wil move on to to handling some actual data. In this section, we use real data sets from the `nycflights13` package, which contains information about flight arrival and departures for all 336,776 commercial flights in the following airports in New York City in 2013.

Table	Description
`airlines`	Airline carrier codes and names.
`airports`	Airport names and locations.
`flights`	Airline on-time data for all flights departing NYC in 2013.
`planes`	Construction information about each plane.
`weather`	hourly weather data for each airport.

The data comes from the US Bureau of Transportation Statistics, and detailed documentation can be found in `?nycflights13`.

We can use this dataset to answer questions such as:

- When is the best time to travel?
- Does bad weather cause flight delays?
- Do single delays cause multiple delays?
- Do older planes suffer from more delays?

We will be using the `dplyr` package to explore the data. This package provides 5 verbs that can be used for simple data manipulation tasks. These provide a rich means of slicing-and-dicing a single table of data when used in conjunction with each other.

Verb	Description
`select`	`select` a subset of columns from a data frame.
`filter`	`filter` (i.e. select a subset) rows of a data frame.
`arrange`	`arrange` (i.e. reorder) rows in ascending or descending order.
`mutate`	`mutate` (i.e. transform) existing columns to derive new columns.
`summarize`	`summarize` (i.e. collapse) the rows down to a single row.

The verbs can also be used in conjunction with `group_by`to partition a table into specified *groups* of rows and applying a summary operation by group, e.g. the range of values contained within each group.

All `dplyr` verbs workin in a similar way:

- The first *argument* is a data frame,
- The subsequent arguments describe what to do with the data frame.
- The result is a new data frame.

```r
library(nycflights13)
library(dplyr)
```

```r
## Let's start by looking at the "flights" table.
head(flights)

## How big is it?
dim(flights)  # 336,776 rows by 19 columns, or 6,398,744 observations.
```

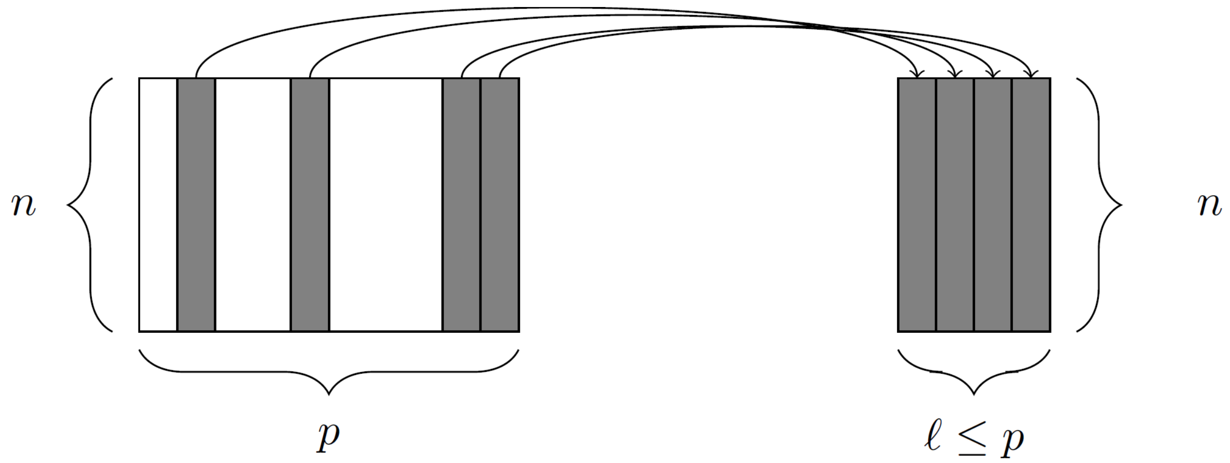

Figure 5: The `dplyr::select()` function

`select()`

The `flights` table has 19 columns. If we know in advance which columns we are interested in, it would make sense to take a subset of the table by `select`ing a few columns, e.g. departure and arrival times/delays.

```
## Take the table "flights", select the desired columns, and print the
## dimensions of the table.
flights %>%
  select(dep_delay, arr_delay, dep_time, arr_time) %>%
  dim()

## Exercise: Create a variable called "times_df" containing the table
##           above, and print the first few rows using "head()"
```

Sometimes it can be very useful to be able to extract columns by specifying a textual pattern:

```
flights %>%
  select(contains("delay"))

flights %>%
  select(starts_with("dep"))

flights %>%
  select(ends_with("delay"))

## Exercise: Write some R code to return a data frame which only contains
##           columns that begin with "arr" and end with "delay".
##           (HINT: try "select()"ing twice.)
```

`filter()`

We can also `filter` rows by a specified condition. For example, we could extract information about all flights that flow on Christmas day. Note the use of the `==` operator for checking equality. This is because `=` is used for variable assignment and function argument specification.

```
## Christmas flights
flights %>%
  filter(month == 12, day == 25) # Simply list conditions, separated by ","
```

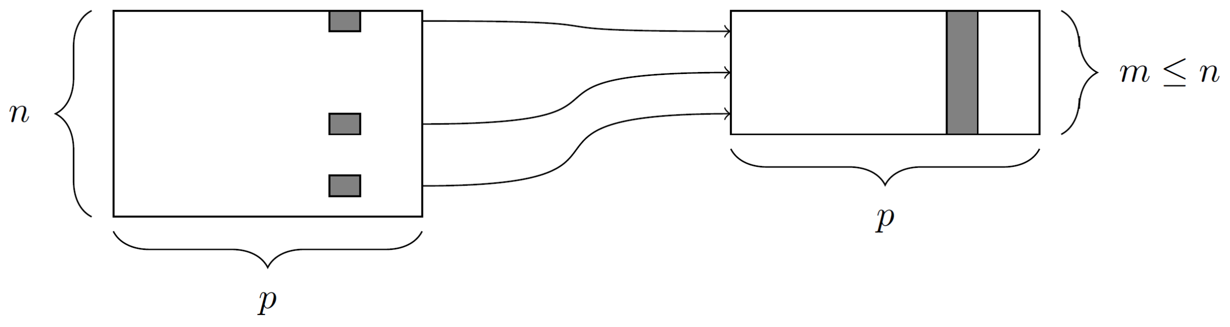

Figure 6: The `dplyr::filter()` function

```r
## Flights with a departure delay of more than 2 hours
flights %>%
  filter(dep_delay >= 120)

## Flights operated by JetBlue (B6) or ExpressJet (EV)
flights %>%
  filter(carrier %in% c("B6", "EV"))

## Winter flights
flights %>%
  filter(between(month, 1, 3))

## Cancelled flights
flights %>%
  filter(is.na(dep_time), is.na(arr_time))

## Exercises:
##
## Which of the flights:
##
## 1. Arrived more than three hours late but didn't leave late.
##
## 2. Flew in January, March, or May.
##
## 3. Were operated by Haiwaiian Airlines, Endevour Air, or
##    Frontier Airlines and ended up being cancelled.
##    Look at the 'airlines' table for carrier codes.
```

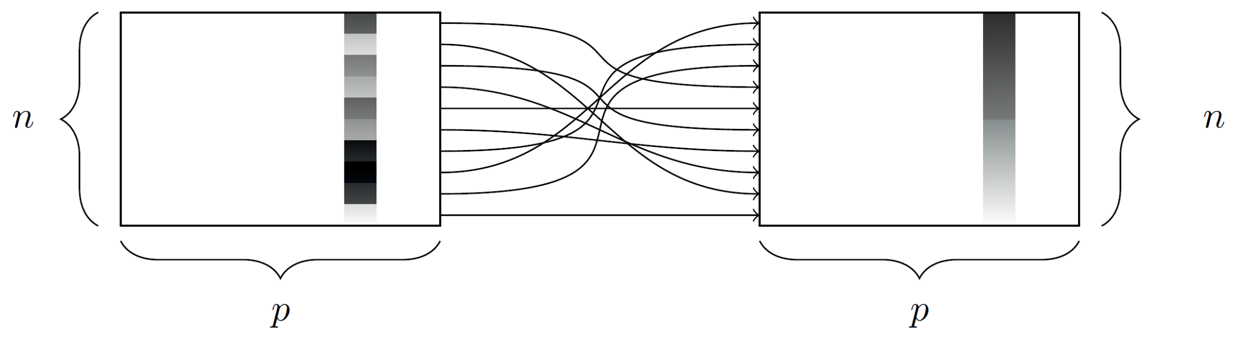

Figure 7: `The dplyr::arrange() function`

`arrange()`

The next verb is `arrange`: we can reorder the rows in ascending or descending order of specified columns. This would be very useful for finding out things like which of the flights had the longest or shortest delay.

```
## Flights with the longest departure delays.
flights %>%
  select(dep_delay) %>%
  arrange(dep_delay)

flights %>%
  select(dep_delay) %>%
  arrange(desc(dep_delay))

## Fastest flights
flights %>%
  select(air_time) %>%
  arrange(air_time)

## We can slice() a data frame to extract rows by index. For example,
## we can extract the 15th - 20th fastest flights:
flights %>%
  select(air_time) %>%
  arrange(air_time) %>%
  slice(15:20)

## Notice that there are a few duplicated values for 'air_time'.
## We can drop these as follows:
flights %>%
  select(air_time) %>%
  filter(!duplicated(air_time)) %>%
  arrange(air_time) %>%
  slice(15:20)

## Another way:
flights %>%
  select(air_time) %>%
  distinct() %>%
  arrange(air_time) %>%
  slice(15:20)
```

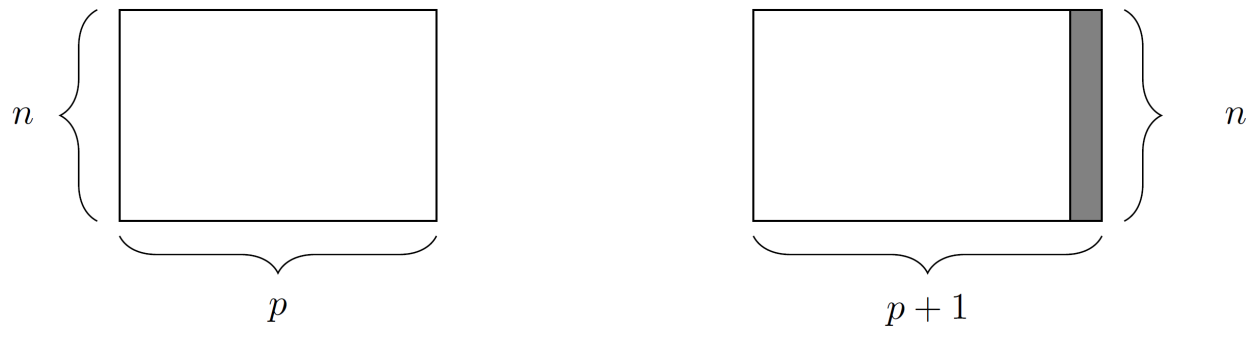

Figure 8: The `dplyr::mutate()` function

```
## We can start composing the verbs to write more complex queries:
## the following lists the carriers with the longest departure delays
## on christmas day.
flights %>%
  select(year, month, day, dep_delay, carrier) %>%
  filter(month == 12, day == 25) %>%
  select(carrier, dep_delay) %>%
  arrange(desc(dep_delay)) %>%
  slice(1:10)

## While it wouldn't be very efficient to store full names in the
## original table, since we've now reduced the table down to a
## much smaller sub-table, it would be useful to be able to
## see the full names of carriers. To do this, we can "join" the
## two tables. "inner_join()" finds matching entries in both tables
## and combines them into one.
flights %>%
  select(year, month, day, dep_delay, carrier) %>%
  filter(month == 12, day == 25) %>%
  select(carrier, dep_delay) %>%
  arrange(desc(dep_delay)) %>%
  slice(1:10) %>%
  inner_join(airlines, by = "carrier") %>%
  select(name, dep_delay)

## Exercises:
##
## 1. Which carriers had the shortest departure delays?
##
## 2. Which flights had the longest flight (i.e. greatest distance)?
```

`mutate()`

We can also derive new columns within a data frame by *mutating* existing columns with a call to `mutate()`. For example, it might be useful to have a column with information about how much time flights gained in the air, by computing the difference between the arrival and departure delays.

```r
## Add the "gain" column described above.
flights %>%
  filter(!is.na(arr_delay), !is.na(dep_delay)) %>%
  mutate(gain = arr_delay - dep_delay,
         gain_per_hour = gain / (air_time / 60)) %>%
  arrange(desc(gain), desc(gain_per_hour))

## It is also possible to discard all the old variables:
flights %>%
  filter(!is.na(arr_delay), !is.na(dep_delay)) %>%
  transmute(gain = arr_delay - dep_delay,
            gain_per_hour = gain / (air_time / 60)) %>%
  arrange(desc(gain), desc(gain_per_hour))

## Sort in descending order:
flights %>%
  filter(!is.na(arr_delay), !is.na(dep_delay)) %>%
  mutate(gain = arr_delay - dep_delay,
         gain_per_hour = gain / (air_time / 60)) %>%
  arrange(desc(gain), desc(gain_per_hour)) %>%
  inner_join(airlines, by = "carrier") %>%
  select(name, gain, gain_per_hour)

## All the "_time" columns in "flights" aren't in a very useful format.
## Let's convert these into "minutes since midnight" format.
flights %>%
  select(ends_with("_time"))

517 %/% 100      # Hour         (Integer division)
517 %/% 100 * 60 # Hour in mins (Integer division)
517 %% 100       # Minute       (Modulo remainder)

## Let's write a function that does this for us.
conv_time = function(time) time %/% 100 * 60 + time %% 100

## 5:17 am means 5 hrs and 17 mins, or 317 mins since midnight.
conv_time(517)

flights %>%
  mutate(dep_time_mins = conv_time(dep_time))

## Exercises:
##
## 1. Create a mutated version of all columns that end with "_time" and
##    arrange them in descending order of the new "dep_time".

## 2. Modify the "conv_time" function above so that it returns the
##    time since midnight in HOURS. Then repeat Q1 above.

## 3. Does air_time equal arr_time - dep_time? Why or why not?
##    (HINT: add a new column, and add another column computing
##           the difference between the new column and air_time.)
```

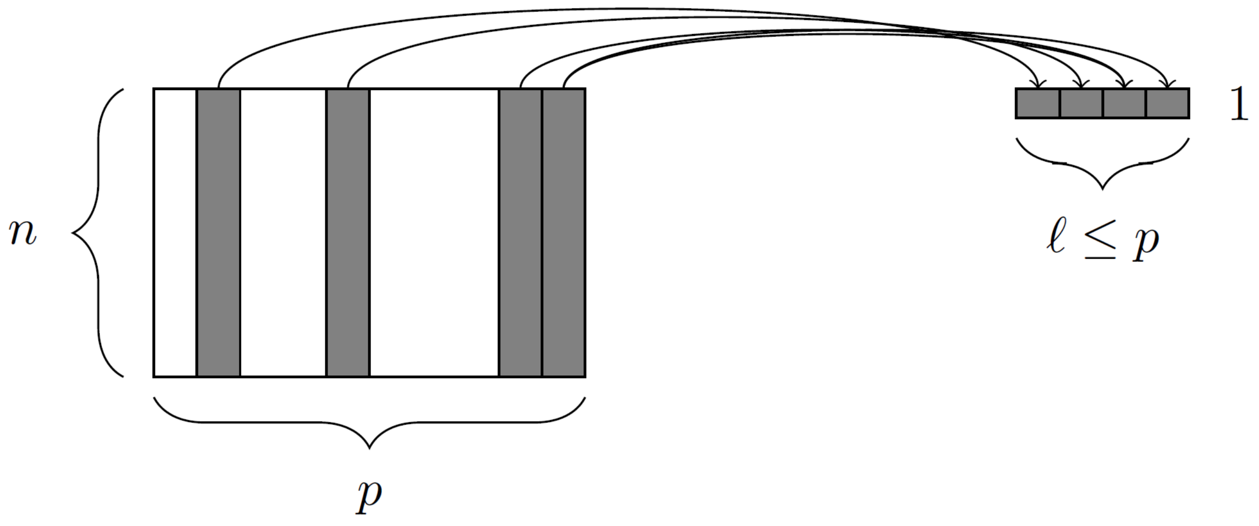

Figure 9: `The dplyr::summarise() function`

`summarise()`

The last of the five main `dplyr` verbs is `summarise`, which *collapses* a data frame down to a single row by applying a summary operation.

```r
## Compute the mean arrival and departure delays, and drop missing values.
flights %>%
  summarize(mean_arr_delay = mean(arr_delay, na.rm = TRUE),
            mean_dep_delay = mean(dep_delay, na.rm = TRUE))
```

`summarise` is particularly useful when applied to *groups* of observations by using it in conjunction with `group_by()`.

```r
## Mean arrival and departure delay times by carrier.
flights %>%
  filter(!is.na(arr_delay), !is.na(dep_delay)) %>%
  group_by(carrier) %>%
  summarize(mean_arr_delay = mean(arr_delay),
            mean_dep_delay = mean(dep_delay),
            count = n()) %>%
  inner_join(airlines, c("carrier" = "carrier")) %>%
  select(name, mean_arr_delay, mean_dep_delay, count)

## Exercises:
##
## 1. Which carrier had the lowest mean arrival delay?
##
## 2. Which carrier had the lowest mean departure delay on xmas day?
##
## 3. Which carrier had the highest maximum departure delay?
##
## 4. Which flights were the most delayed, based on arrival, on xmas day?
##
## 5. Which season had the highest proportion of cancelled flights?
##    Use the code below to get started.
library(lubridate)
```

```
## Function for adding seasonal information.
seas2013 = function(dates) {
  seasons = character(length(dates))
  seas = ymd(c("20130320", "20130621", "20130922", "20131221"))
  labs = c("Spring", "Summer", "Fall", "Winter")
  seasons[dates >= seas[4] | dates < seas[1]] = labs[4]
  for (i in 1:3) {
    seasons[dates >= seas[i] & dates < seas[i + 1]] = labs[i]
  }
  seasons
}

## Create a new version of the "flights" table with seasons.
flights2 = flights %>%
  filter(!is.na(year), !is.na(month), !is.na(day)) %>%
  mutate(date = make_date(year, month, day),
         seas = seas2013(date))

## The following gives the proportion of cancelled flights by month.
flights2 %>%
  filter(is.na(dep_time)) %>%
  group_by(month) %>%
  summarize(count = n()) %>%
  mutate(perc = round(count / sum(count) * 100, 2))

## So back to the question:
##
## Which month had the highest proportion of cancelled flights?

## Another one: Which season had the lowest proportion of cancelled flights?
```

Data Visualization with ggplot2

So far, we looked at how to program in R and learned how to manipulate data frames using the `dplyr` package. But all our work has been textual so far – we haven't produced any pretty pictures! In this section, we will learn how to visualize data using the `ggplot2` package.

`ggplot2` allows us to build statistical graphics incrementally by providing a kind of *grammar* for data visualization, much like how `dplyr` provided a *grammar* for data manipulation. Not surprisingly, both packages were created by the same person – Chief Scientist of RStudio and Adjunct Professor of Statistics at the Univesity of Auckland, Stanford University, and Rice University, Dr. Hadley Wickham. Whilst packages like `ggplot2` and `dplyr` do *not* add much in the way of *new* functionality to R, it does provide a uniform, user-friendly interface for interacting with data.

There are three key components in a plot produced using `ggplot2`:

- A set of `data` to plot,
- A set of `aes`thetic mappings, and
- A visualization layer.

An aesthetic is mapping between a variable and visual cues that can represent variable values, e.g. glyphs. A *glyph* is a basic graphical element that represents a single observation, e.g. mark, symbol. For example, in a scatterplot, the *positions* of a glyph on the plot act as visual cues that help us understand how two variables are related.

```r
library(ggplot2)

## The "women" data set contains the heights and weights of 15 American
## women in ordinary indoor clothing and with shoes on. We can study
## the relationship between the variables "height" and "weight" by
## mapping the height to the x-axis and weight to the y-axis. The
## `aes()` function *maps* variables from the data to graphical
## attributes. We can assign this graphic to the variable "p".
p = ggplot(data = women, aes(x = height, y = weight))

## At this point, "p" is just an empty plot, as we haven't added a
## visualization layer. We can add features to "p" using the `+`
## operator. In this case, we add points.
p + geom_point()
p + geom_point(size = 3)

## We can also plot using pipe operators. Here, we convert heights and
## weights to sensible units and derive a new categorical variable
## called "weight_class".
women2 = women %>%
  mutate(height_cm = height * 2.54,
         weight_kg = weight * 0.453,
         weight_class = cut(weight, 3,
                            labels = c("Low", "Medium", "High")))

## Map the colour of each dot to "weight_class". Observe that we
## can omit specifying the x and y arguments in ggplot().
women2 %>%
  ggplot(aes(height, weight)) + # x-axis, y-axis
  geom_point(aes(color = weight_class))

## Map the size of each dot to "weight_class".
```

```r
women2 %>%
  ggplot(aes(height, weight)) +
  geom_point(aes(size = weight_class))

## The two example above define an additional aesthetic (colour and size)
## to display the additional variable "weight_class". An alternative way
## is to use multiple side-by-side graphs, called "facets".
p = women2 %>%
  ggplot(aes(height, weight))

p + geom_point() +
  facet_wrap(~ weight_class)

## Exercise:
##
## 1. Derive a new variable called "height_class", with labels "Short",
##    "Average", and "Tall"; then produce a plot as above, but map the
##    color of each dot to the variable "height_class".
##
## 2. Use facets instead.
```

Other plots include:

```r
## Line plot
women2 %>%
  ggplot(aes(height_cm, weight_kg)) +
  geom_line()

## Box plot
women2 %>%
  ggplot(aes(weight_class, height_cm)) +
  geom_boxplot(fill = "darkseagreen")
```

Now that we've covered the basics, let's visualize the `flights` data. Before we proceed any further though, we create an enriched version of the table to make it easier to visualize it over time.

```r
library(lubridate)

## Add seasons.
seas2013 = function(dates) {
  seasons = character(length(dates))
  seas = ymd(c("20130320", "20130621", "20130922", "20131221"))
  labs = c("Spring", "Summer", "Fall", "Winter")
  seasons[dates >= seas[4] | dates < seas[1]] = labs[4]
  for (i in 1:3) {
    seasons[dates >= seas[i] & dates < seas[i + 1]] = labs[i]
  }
  seasons
}

## Function for converting time format.
conv_time = function(time, min = TRUE) {
  if (min) {
    time %/% 100 * 60 + time %% 100
  } else {
    round(time %/% 100 + time %% 100 / 60, 2)
  }
}

## Create new version of "flights".
flights2 = flights %>%
  filter(!is.na(year), !is.na(month), !is.na(day)) %>%
  mutate(cancelled = is.na(dep_time),
         dep_delayed = dep_delay > 0,
         arr_delayed = arr_delay > 0,
         delayed = dep_delay > 0 | arr_delay > 0,
         date = make_date(year, month, day),
         seas = seas2013(date),
         wday = wday(date, label = TRUE),
         week = week(date),
         hour = hour(time_hour),
         minute = minute(time_hour),
         month_lab = month(month, lab = TRUE),
         id = row_number(),
         dep_min = conv_time(dep_time),
         dep_hr = conv_time(dep_time, min = FALSE),
         sched_dep_min = conv_time(sched_dep_time),
         sched_dep_hr = conv_time(sched_dep_time, min = FALSE),
         arr_min = conv_time(arr_time),
         arr_hr = conv_time(arr_time, min = FALSE),
         sched_arr_min = conv_time(sched_arr_time),
         sched_arr_hr = conv_time(sched_arr_time, min = FALSE),
         air_time_min = conv_time(air_time),
         air_time_hr = conv_time(air_time, min = FALSE),
         dep_datetime = make_datetime(year, month, day,
                                      dep_time %/% 100,
                                      dep_time %% 100),
         arr_datetime = make_datetime(year, month, day,
                                      arr_time %/% 100,
                                      arr_time %% 100),
         sched_dep_datetime = make_datetime(year, month, day,
                                            sched_dep_time %/% 100,
                                            sched_dep_time %% 100),
         sched_arr_datetime = make_datetime(year, month, day,
                                            sched_arr_time %/% 100,
                                            sched_arr_time %% 100))

## Look at some of the new variables.
flights2 %>%
  select(wday, week, month_lab, dep_time, dep_datetime, dep_hr, dep_min)

## Save
saveRDS(flights2, "flights2.Rds")
flights2 = readRDS("flights2.Rds")
```

We are now ready to start visualizing the `flights` data. Let's start by plotting a histogram of the departure time. Recall that histograms divide up the x-axis into *bins* and count the number of observations that fall into each bin; it then displays the count with bars. We can adjust the `binwidth` to cover different ranges to reveal different characteristics of the data. It is also possible to specify the number of `bins`.

```r
## Base plot for "dep_min".
p = flights2 %>% ggplot(aes(dep_min))

## Histogram of departure times in minutes.
p + geom_histogram(bins = 100)
p + geom_histogram(binwidth = 100)

## Lines instead of bars.
p + geom_freqpoly(binwidth = 100)

## Distribution of xmas day flights.
flights2 %>%
  filter(month == 12, day == 25) %>%
  ggplot(aes(dep_datetime)) +
  geom_freqpoly(bins = 100)

## Number of flights by day of week.
flights2 %>%
  ggplot(aes(wday)) +
  geom_bar(fill = "lightblue")

## Fewer flights on weekends.
flights2 %>%
  group_by(date) %>%
  summarise(count = n()) %>%
  mutate(wday = wday(date, lab = TRUE)) %>%
  ggplot(aes(wday, count)) +
  geom_boxplot()

## Exercise: Plot the distribution of flights in December.
```

Others

```r
## Flights by hour
flights2 %>%
  ggplot(aes(hour)) +
  geom_bar() +
  scale_x_continuous(breaks = 0:24)

## Average departure delay among non-cancelled flights.
## Observe that variation decreases with the number of flights.
flights2 %>%
  filter(!is.na(dep_delay)) %>%
  group_by(tailnum) %>%
  summarize(mean_delay = mean(dep_delay, na.rm = TRUE),
            count = n()) %>%
  ggplot(aes(count, mean_delay)) +
  geom_point(alpha = 0.1) +
  geom_smooth() # add model fit.

## Number of flights per day.
p2 = flights2 %>%
  group_by(date) %>%
  summarize(count = n()) %>%
  ggplot(aes(date, count)) +
  geom_line() +
  xlab("Date") +
  ylab("Number of Flights Scheduled")

p2
p2 + geom_smooth()
p2 + geom_smooth(method = "lm")
```

Cancelled flights

```r
## Compare scheduled departure times for cancelled and non-cancelled.
flights2 %>%
  ggplot(aes(sched_dep_hr)) +
  geom_freqpoly(aes(colour = cancelled), binwidth = 0.5) +
  xlab("Hour") +
  ylab("Number of Flights Scheduled")

flights2 %>%
  ggplot(aes(cancelled)) +
  geom_bar(aes(fill = cancelled))
```

Delayed flights

```r
## Departure delay
flights2 %>%
  filter(!is.na(dep_delayed)) %>%
  ggplot(aes(dep_delayed)) +
  geom_bar(aes(fill = dep_delayed))

## Arrival delay
flights2 %>%
  filter(!is.na(arr_delayed)) %>%
  ggplot(aes(arr_delayed)) +
  geom_bar(aes(fill = arr_delayed))

## Departure delay by airport of origin
flights2 %>%
  group_by(origin) %>%
  summarise(mean_dep_delay = mean(dep_delay, na.rm = TRUE)) %>%
  ggplot(aes(origin, mean_dep_delay)) +
  geom_col() # geom_bar() uses stat = count() by default.
```

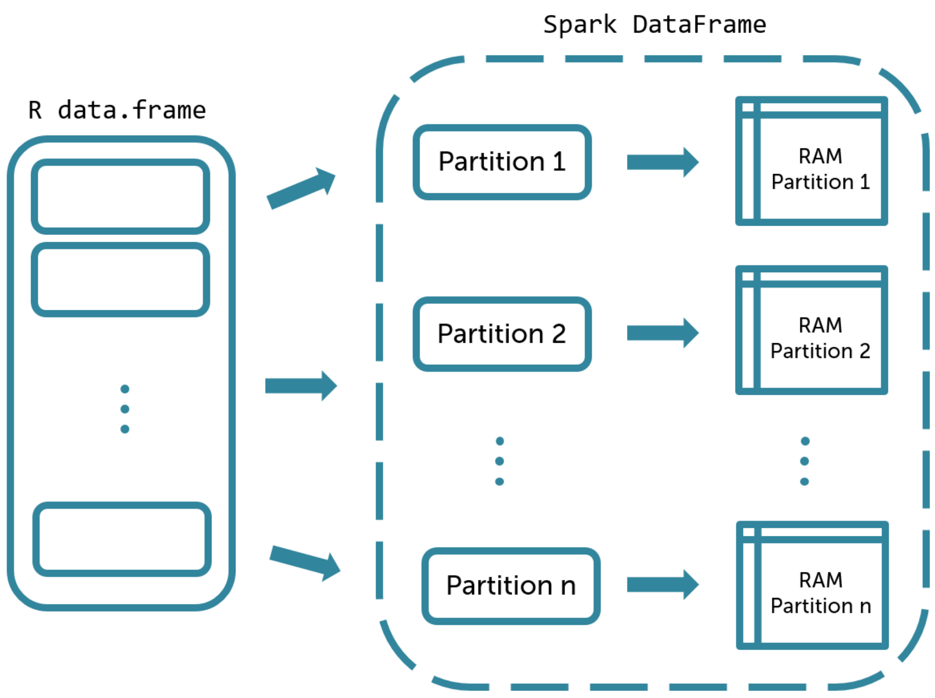

Figure 10: Distributing a local R `data.frame` to a Spark `DataFrame`

Sparklyr

Setup:

```
## Load required packages.
library(dplyr)
library(sparklyr)
library(nycflights13)
library(ggplot2)
library(lubridate)

## Set config parameters for training laptops.
config = spark_config()
config$spark.executor.cores = 4      # parallel::detectCores()
config$spark.executor.memory = "4G"  # approx. .5 * memory.size()
config$spark.driver.memory = "4G"

## Connect to Spark cluster
sc = spark_connect("local", config = config)
# spark_disconnect(sc)

## Create Spark DataFrame
flights_rdf = readRDS("flights2.Rds")

## Sparklyr doesn't seem to handle dates naturally.
date_cols = flights_rdf %>%
  select(matches("date"), "time_hour") %>%
  names()
```

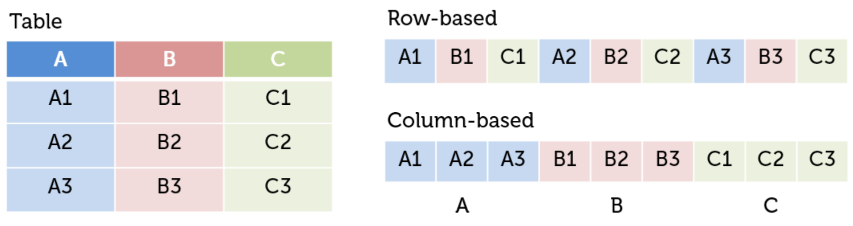

Figure 11: Row-based vs. column-based Storage

```r
date_cols
x = vector("list", length(date_cols))
for (i in seq_along(date_cols)) {
  x[[i]] = as.character(flights_rdf[[date_cols[i]]])
}
names(x) = date_cols
y = do.call("cbind.data.frame", x)
head(y)

flights2_rdf = flights_rdf %>%
  select(-one_of(date_cols)) %>%
  cbind(y)

# saveRDS(flights2_rdf, "fights2_rdf")
```

Efficient storage

We store the table as a Parquet file, which is an efficient column-based format for storing data. Each column is stored with other columns of the same type, and data is spread across different blocks on disk.

Why is this a good thing? suppose we want to `filter` the rows of `flights` based on the condition `dep_delay > 120`, i.e. departure delays of more than 2 hours. If the data is stored in a record-based format like .csv, the query needs to:

- Scan and read in the row,
- Parse the row into different columns, e.g. `year`, `month`, `day`, `dep_delay`, etc.,
- Obtain the relevant column (`dep_delay`), and
- Filter the row based on the condition `dep_delay > 120`.

for *every* row. If we use a column-based format like Parquet instead, the query only reads in the column of interest (`dep_delay`) - it ignores the other columns as it knows where that column resides and what type it is, all in advance. This results in a lot less work.

```r
## Create a Spark DataFrame out of the local R data.frame.
flights_sdf = copy_to(sc, flights2_rdf, "flights", overwrite = TRUE)

## Save as Parquet file. Parquet is a columnar file format, cf.
## csv which is record or row-based. When querying the latter,
## the query needs to scan every row of the data set: read it in,
## parse  it into fields, etc. With columnar formats, the query
## is only concerned with the columns required for the query,
## and doesn't bother with the others - so we can avoid doing
## a lot of work.
```

```r
# spark_write_parquet(flights_sdf, "flights_sdf.parquet")

## Read flights table.
flights_sdf = spark_read_parquet(sc,  "flights_sdf", "flights_sdf.parquet")

## Have an initial look at the table.
head(sdf_schema(flights_sdf))

## How many partitions?
sdf_num_partitions(flights_sdf)
```

Caching

In general, storing data in RAM is faster than disk (e.g. Hard Drive) because RAM sits very close to the CPU (the *brain* of the computer) so that data has to travel less. Moreover, physical hard disks have mechanical parts that move around and find a place to store data (think of a needle on an old record player finding a song of your choice), which takes time. RAM stands for Random Access Memory, because computers are said to be able to access any region of memory equally quickly. Disk space is about 100 times cheaper per byte than RAM, but between 5-100,000 times slower depending on the type of workload.

Spark allows us to pull data sets into a cluster-wide in-memory cache. Cache is memory that is much closer to the CPU than RAM. This means tranfers to and from cache takes much less time than RAM (roughly 10 times faster).

```r
## Register as Spark DataFrame and give it a table name for the Spark SQL
## context.
microbenchmark::microbenchmark(
  flights_sdf %>%
    sdf_sample(frac = 1000, replace = TRUE) %>%
    filter(!is.na(dep_min), !is.na(arr_min)) %>%
    group_by(wday) %>%
    summarise(mean_dep_delay = mean(dep_min),
              mean_arr_delay = mean(arr_min))
)
#      min       lq     mean   median      uq      max  neval
# 46.21666 52.87225 68.25759 57.39077 75.1553 155.1505    100

## Cache table.
tbl_cache(sc, "flights")
head(flights)

microbenchmark::microbenchmark(
  flights_sdf %>%
    filter(!is.na(dep_min), !is.na(arr_min)) %>%
    group_by(wday) %>%
    summarise(mean_dep_delay = mean(dep_min),
              mean_arr_delay = mean(arr_min))
)
#      min      lq     mean   median       uq      max  neval
# 4.380441 4.86123 6.506899 5.272292 5.665379 37.34042    100
```

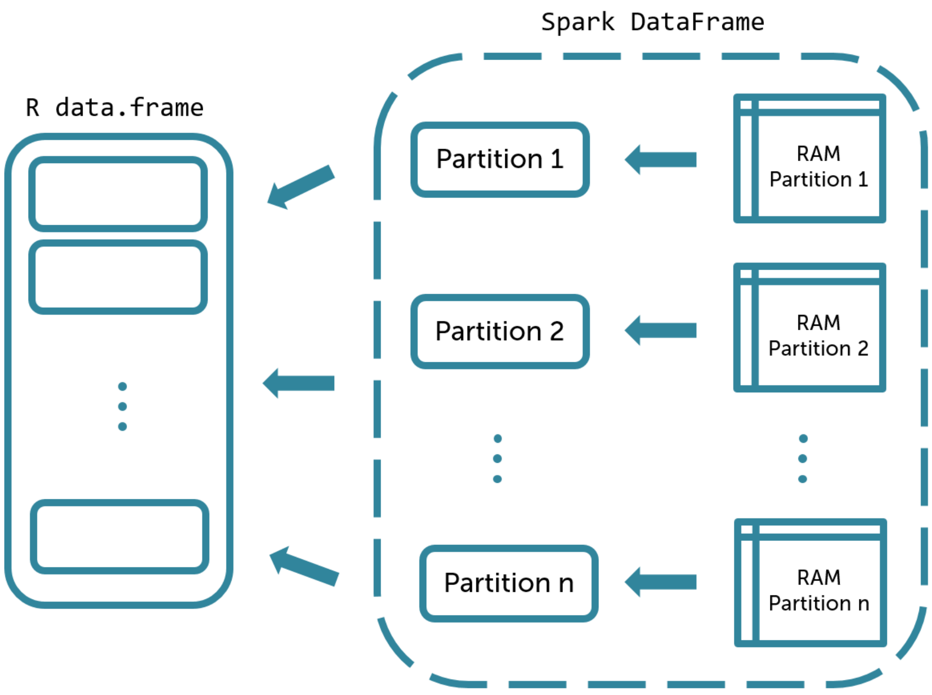

Figure 12: Collecting a Spark `DataFrame` to a local R `data.frame`

Plots

```
## Run some example code. Observe that only difference is
## `collect()`ing the distributed Spark DataFrame into a
## local R data frame prior to plotting.
flights_sdf %>%
  select(year, month, day, dep_delay, arr_delay) %>%
  filter(!is.na(arr_delay)) %>%
  mutate(gain = dep_delay - arr_delay) %>%
  filter(gain > 60) %>%
  collect() %>% # Collect distributed structure into memory.
  mutate(date = make_date(year, month, day)) %>%
  ggplot(aes(date, gain)) +
  geom_point(alpha = 0.5, col = "darkseagreen", cex = 2)

flights_sdf %>%
  select(sched_dep_hr, cancelled) %>%
  collect() %>%
  ggplot(aes(sched_dep_hr)) +
  geom_freqpoly(aes(colour = cancelled), binwidth = 0.5) +
  xlab("Hour") +
  ylab("Number of Flights Scheduled")
```

SQL

It is also possible to write queries in SQL against Spark tables.

```
library(DBI)
dbGetQuery(sc, "
SELECT dep_time FROM flights_sdf
WHERE dep_time > 100 AND NOT(dep_time) IS NULL
LIMIT 5
")

## It is in fact possible to translate "sparklyr" code directly to SQL.
q = flights_sdf %>%
  select(dep_time) %>%
  filter(dep_time > 100, !is.na(dep_time)) %>%
  head(5)
q

show_query(q)
```

Models

```
## Create test and training sets.
flights_part = flights_sdf %>%
  filter(dep_delay > 0, arr_delay > 0,
         !is.na(dep_delay), !is.na(arr_delay)) %>%
  select(dep_delay, arr_delay) %>%
  sdf_partition(training = 0.7, test = 0.3, seed = 1234)

## Plot
flights_part$training %>%
  collect() %>%
  ggplot(aes(dep_delay, arr_delay)) +
  geom_line() +
  geom_smooth(method = "lm")

flights_part$training %>%
  collect() %>%
  ggplot(aes(sample = log10(arr_delay))) +
  stat_qq()


## Fit a linear model.
flights_lm = flights_part$training %>%
  ml_linear_regression(response = "dep_delay",
                       features = "arr_delay")

summary(flights_lm)
```

Review

The aim of today's workshop was to get hands-on experience in dealing with *reasonably large* data sets using R. You should now be able to:

- Read and write simple programs in the R language,
- Manipulate tabular data sets in R using the `dplyr` package,
- Visualize tabular data sets in R using the `ggplot2` package,
- Understand R's limitations when it comes to large data sets, and
- Manipulate *reasonably large* tabular data sets in R using the `sparklyr` package.

Appendix

Project Ideas

- `sparklyr` Web Applications to combine `nycflights13` with `leaflet`.
- NYC Flights 2014 Data for comparing with `nycflights13`.
- R Data Sets

Assignment Operator

Some of you may be more familiar with `<-` for variable assignment. In this course we use the C-style `=` operator instead for greater compatibility with mainstream programming languages (it also saves us from having to type an extra keystroke every time we assign a variable).

The main difference between the two operators is that `=` is only allowed in two places: at the top level, or within braces/extra pair of parens. For example, to assign a variable during a function call, one would have to wrap the expression within an extra pair of parens:

```
# mean(y = 1:10)          # illegal
# mean((y = 1:10))        # legal
# mean(y <- 1:10)         # legal
```

Another example might be in control structures:

```
# if (x = 0) 1 else x     # illegal
# if ((x = 0)) 1 else x   # legal
# if (x <- 0) 1 else x    # legal
```

However, variable assignment during a function call or within control structures can cause confusion, especially when used named arguments. At the end of the day, the choice comes down to preference, but the course uses to `=` to avoid using constructs that are unique to domain-specific programming languages like R. If you're interested, you can read more about assignment operators on the R Developer website.

R References

- Book: Modern Data Science with R
- Book: Parallel Computing for Data Science
- Book: R for Data Science
- Tutorial: Hexagonal Binning
- Tutorial: Introduction to `dplyr`
- Tutorial: Introduction to `ggplot2`
- Tutorial: `rcpp`
- Tutorial: Spatial Visualizations in R
- Website: Leaflet for R
- Website: `sparklyr`
- Website: `SparkR`