Business 4720

Introduction to Data Management with R

Joerg Evermann

Unless otherwise indicated, the copyright in this material is owned by Joerg Evermann. This material is licensed to you under the Creative Commons by-attribution non-commercial license (CC BY-NC 4.0)

1 Introduction

R is a highly acclaimed statistical software and programming language known for its robust capabilities in data analysis, visualization, and statistical computing. It was conceived in the early 1990s by Ross Ihaka and Robert Gentleman at the University of Auckland, New Zealand. Drawing inspiration from the S language developed at Bell Laboratories, R was designed to be a powerful and flexible tool for data analysis and statistical modeling.

One of the key advantages of R is its open-source nature, making it freely available to users worldwide. This accessibility has fostered a vibrant community of users and developers, continuously enhancing its functionality through comprehensive packages and extensions. The Comprehensive R Archive Network (CRAN), a repository of these packages, is a testament to R's extensible architecture, offering tools for a myriad of data analysis tasks.

R's popularity stems not only from its wide range of statistical techniques, including linear and nonlinear modeling, time-series analysis, classification, clustering, and others, but also from its exceptional capabilities in data visualization. The software provides an integrated suite of tools for data manipulation, calculation, and graphical display, making it an invaluable asset for statisticians, researchers, and data scientists.

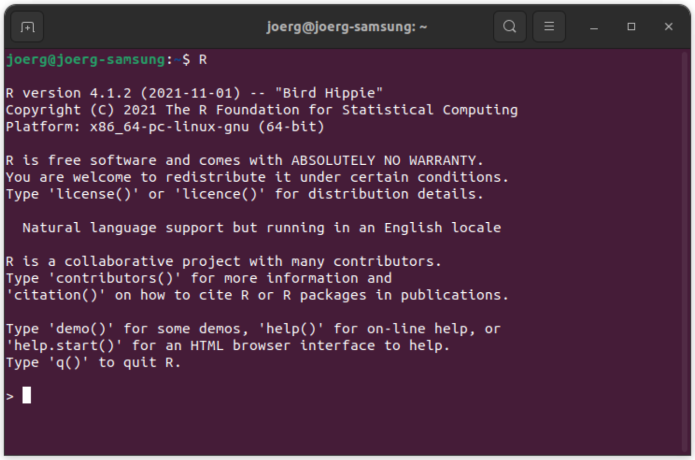
Moreover, R's programming language aspect allows for automation and customization in data analysis, which is highly beneficial for complex and repetitive tasks. Its compatibility with various data formats and integration with other programming languages and tools further enhances its versatility.

2 Using R

R is a command-line oriented software, that is, users type commands to perform calculations or call functions of R packages. A sequence of R commands can be assembled in a *script file*, so that they may be re-run when necessary. The advantage of this type of software over one with a graphical user interface is in the repeatability and replicability of the work. Ideally, data analysts will assemble an R script file for their entire data analysis, from raw data sets to finished statistical analyses and visualizations, so that all details of the analysis are available for replication and evaluation.

The R system can be launched simply by invoking the `R` command from the terminal window, as shown in Figure 1. R will display its version information and prompt for command entry with a > prompt.

To install R on Microsoft Windows or on MacOS, download the installation files from CRAN (Comprehensive R Archive Network) at `https://cran.r-project.org` and follow the instructions. R on Microsoft Windows and R on MacOS will show their command prompts inside a window but otherwise function similarly to R on Ubuntu that is installed in the course virtual machine.



```
joerg@joerg-samsung:~$ R

R version 4.1.2 (2021-11-01) -- "Bird Hippie"
Copyright (C) 2021 The R Foundation for Statistical Computing
Platform: x86_64-pc-linux-gnu (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

> 
```

Figure 1: The R command line interface

Tip: A good, easy, and comprehensive introduction to R can be found here: https://cran.r-project.org/doc/manuals/r-release/R-intro.pdf

Tips for working efficiently with R: To make using R more efficient, consider doing the following:

- Use the up-arrow key to retrieve earlier commands.
- The `history()` function shows your command history.
- Use a notepad app to assemble your commands, then copy/paste to R.
- Use a notepad app for your results, copy/paste from R.
- The Ubuntu terminal window uses SHIFT-CTRL-X , SHIFT-CTRL-C , SHIFT-CTRL-V for cut/copy/paste.
- Use multiple terminal and R windows (e.g. one for executing commands, one for reading help documentation or for listing files).
- Don't update packages in the middle of a project.
- Ensure you have a *repeatable, automatable script* for your entire data analysis at the end of a project.

3 R Basics

The most basic way to use R is to simply use it as a calculator, as shown in the following R code example. Type "`1+1`" at the ">" prompt, then press the RETURN key to execute the statement. R will respond on the following line with the result:

```
> 1+1
[1] 2
```

A *variable* in R is a named storage space for numbers, characters, strings, and other data elements. Traditionally, values are assigned to variables using the <− operator, but one may also use the more ”normal” assignment operator =. Using the <− assignment operator helps to clearly distinguish assignment from equality testing, which uses ==.

The following R code example introduces the R function called `print()` that does as its name suggests. Most data types and data structures that can be assigned to variables have a useful print function associated to them, so that on the interactive R command line you can simply type their name to get their value. In interactive mode, R calls the `print()` function automatically, in an R script that you execute from file, you will have to exiplictly use the `print` function.

```
> a <- 3
> b <- 2
> print(a * b)
[1] 6
> a
[1] 3
```

A common structured data type in R is a *vector*. A vector in R contains elements of the same data type and is ordered. When assigning elements of different datatypes to a vector, R will coerce the types of all elements to a common datatype.

```
> v <- c(1, 'a', TRUE)
> v
[1] "1"    "a"    "TRUE"
> v <- c(1, 2, 3, 4)
> v*3
[1]  3  6  9 12
```

Note that R automatically determined that multiplication with a scalar is an element-wise operation and applies it to each element of the vector.

Useful functions to create vectors are the sequence function `seq()`, which accepts the lower and upper limit and a step size as parameter, and the repetion function `rep()` which repeats its first argument the number of times specified by its second argument.

```
> s <- seq(0, 6, by=.5)
> print(s)
> r <- rep(3.5, 5)
> print(r)
```

R provides useful functions for numerical vectors, to find their length, their maximum and minimum value, the square root of their values, as well as the variance and standard deviation of the elements. Note that R automatically determines whether functions are applied to the whole vector, like `var()` or `sd()`, or whether functions are applied element-wise to each element, like `sqrt()`. Vector concatenation, using the `c()` function, automatically ”flattens” the vectors.

```
> length(v)
> max(v)
> min(v)
> sqrt(vv)
> var(v)
> sd(v)
> vv <- c(v, c(7, 8, 9), v)
> print(vv)
```

The most common way to select elements from vectors is by *indexing* with a boolean vector. In the following example, the expression `vv < 5` yields a vector of boolean values. Indexing the variable `vv` with that vector determines which elements of `vv` to select.

```
> vv < 5
> vv[vv < 5]
> vv[vv < 5] <- vv[vv < 5] + 5
```

Vectors can also be indexed numerically, selecting elements by their position. R allows you to specify a sequence using the `:` operator and exclusion of elements using `-`. The first line in the following example selects elements at positions 3 through 7, the second line selects elemtns *except* those at positions 3 through 7.

```
> vv[3:7]
> vv[-(3:7)]
```

Important:

- R begins indexing positions with 1, while other programming languages begin at 0.

Tip:

- The boolean constants `TRUE` and `FALSE` can be abbreviated by `T` and `F`

R also has special symbols to denote infinity (`inf`) and results that are not a number (`NaN`):

```
> 2 / 0
[1] Inf
> 0 / 0
[1] NaN
```

Importantly, `NaN` is *not* the same as a missing value, which is denoted by `NA`, as in the following R code example. The `is.na()` function can be used to identify and index `NA` and then filter them. Any `NA` typically yields an `NA` when an aggregate function is applied. Many functions offer an option to remove `NA` values prior to applying them, as shown for the `sum()` function in the following R code block.

```
> v[3] <- NA
> v*3
[1]  3  6 NA 12
> is.na(v)
[1] FALSE FALSE  TRUE FALSE
> sum(v)
[1] NA
> sum(v, na.rm=TRUE)
[1] 7
```

The boolean logical *and* and *or* are represented by the operators `&` and `|` shown in the R code block below.

```
> TRUE & FALSE
FALSE
> TRUE | FALSE
TRUE
```

Character strings in R are enclosed in single or double quotes (but not mixed quotes!). Two useful functions are `paste()` which pastes its arguments together with an optional separator between them and returns a characters string, and the `strsplit()` function which accepts a string (or vector of strings) to split, and a separator character that identifies where to split the string. It returns a list of vectors of strings.

```
> label1 = 'I Love R'
> label2 = 'and BUSI 4760'
> paste(label1, label2, sep=' ')
> strsplit('Hello World! My first string', ' ')
```

Because you can assign arbitrary values to variables in R, R provides functions to test the value type and to change or coerce the value type. A *factor* data type in R represents categorical data, encoded as different character strings or different numbers.

Categorical data is treated different from numerical or character string data in many statistical analyses.

```
> is.numeric(vv)
> is.integer(vv)
> mode(vv)
> as.character(vv)
> is.character(as.character(vv))
> as.factor(as.character(vv))
> levels(as.factor(as.character(vv)))
```

Important string functions are `grep()`, which checks whether strings contain a substring that matches a regular expression, and `agrep()`, which calculates the Levenshtein distance between a regular expression and a set of strings. The Levenshtein distance is defined as the sum of insertions, deletions, and substitutions of characters to transform one string into another. The first use of `grep()` in the following R code block matches a phone number, the second use of `grep()` matches a Canadian postal code, while the last two examples of `grep()` and `agrep()` exemplify the difference between exact matching with `grep()` and approximate matching with `agrep()`.

```
> grep('^([0-9]{3})[ -]?[0-9]{3}[ -]?[0-9]{4}$',
    c('709 864 5000', 'abc def 9999', '709-865-5000'))
[1] 1 3
> grep('[A-V][0-9][A-V] [0-9][A-V][0-9]',
    c('A0P 1L0', '0AB L2K', 'A0X 1Z0'))
[1] 1
> grep('apple', c('apricot', 'banana', 'grape', 'pineapple'))
[1] 4
> agrep('apple',
    c('apricot', 'banana', 'grape', 'pineapple'),
    max.distance=3)
[1] 1 3 4
```

4 The R Environment

The collection of variables, functions and libraries that exists in R at any one time is called the R *workspace*. R provides many functions to manipulate objects in its workspace, among them `ls()` and `rm()`, named after their Unix bash shell equivalents. The following R code illustrates the use of these functions. Results may vary depending on what variables have been created prior to these commands.

```
> ls()
[1] "a"          "b"          "v"
> rm(v)
> ls()
[1] "a"          "b"
```

R comes with a built-in user manual that one can access with the `help()` function or simply the `?` operator. Help is available on any function in R, as shown in the following example. For added convenience, R provides a web browser interface to its help pages that is started by `help.start()`.

```
> help()
> help(lm)
> ?lm
> ??lm
> help.start()
```

R has a working directory where it reads and writes files from and to. On Ubuntu Linux, this is the directory from which the `r` command was issued. R provides functions to get the working directory, to set (change) it, and to list the files in the working directory:

```
> getwd()
[1] "/home/busi4720"
> setwd('DataSets')
> getwd()
[1] "/home/busi4720/DataSets"
> list.files()
```

Tip: It is often more convenient to change the working directory in the terminal, prior to invoking `r`.

A collection of related functions is called a *library* in R. While some libraries come with the base R system, other packages will need to be downloaded and installed. The CRAN (comrehensive R archive network) provides libraries in convenient form. To install packages from CRAN, use the `install.packages()` which accepts the name (or a vector of names) of packages to install from CRAN. On some systems, R may prompt the user from which CRAN location to install packages. Normally, there is little difference other than download speed.

Installed libraries can be attached to the R workspace with the `library()` function. The `library()` function with an argument attaches the specified package and makes its functions and data sets available for use. The `library()` function without any arguments shows which libraries are installed. The `search()` function shows which packages are currently attached to the workspace. Finally, `installed.packages()` provides details of all installed packages.

```
> search()
> library(matrixcalc)
> search()
> library()
> install.packages('lavaan')
> library()
> installed.packages()
```

It is sometimes useful to assemble a set of related R commands in a script file. As noted earlier, script files are useful to improve the replicability of the data analysis. The `source()` function will read and execute a file containing R commands. As noted earlier, in a script file, you will need to use the `print()` function to print the values of variables.

```
> source('MyFirstScript.R')
```

Finally, the `quit()` function ends an R session. When using `quit()` without arguments, R will ask whether to save the workspace image. R stores its *workspace* in each directory in a file called ".RData" and will read it when restarted from that directory. R also stores its *command history* in each directory in a file called ".Rhsitory" and will read it when restarted from that directory.

```
> quit()
```

5 Arrays, Matrices, Lists, and DataFrames

R *arrays* are multi-dimensional objects that can hold any primitive data type, usually numerical. A *matrix* is simply a two-dimensional array. The following example shows how indexing generalizes from vectors to matrices and arrays simply by indexing each dimension with the same syntax as used for vectors. The `array()` creates multi-dimensional arrays from existing data, the `dim()` function returns the number of dimensions of an array.

A few important things to note in the following R code block example:

- Initially, the array is created from a range of numbers between 1 and 20, and the `dim` argument specifies the dimensionality.
- A dimension need not be subsetted or indexed, as in `a[,2]` or `a[,2:4]` which do not subset the first dimension
- Reversing the index reverses the result that is returned, as in `a[3:1,2:4]` which reverses the indexing of the first dimension.

- An array with two columns is interpreted as a set of indexes, as in `a[i] <- 0`

```
> a <- array(1:20, dim=c(4,5))
> a
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    5    9   13   17
[2,]    2    6   10   14   18
[3,]    3    7   11   15   19
[4,]    4    8   12   16   20
> dim(a)
[1] 4 5
> a[,2]
> a[,2:4]
> a[3,2:4]
> a[3:1,2:4]
> i <- array(c(1:3,3:1), dim=c(3,2))
> a[i] <- 0
> a
```

Constructing a *matrix* with the `matrix()` function is similar to constructing an array, but instead of providing the dimensionality with `dim`, one must provide the number of rows or colums (`nrow` or `ncol`) and how to fill the matrix from the elements provided using the `byrow` argument. The `t` function returns the transpose of a matrix, that is, it reverses rows and columns. Binding two matrices together by columns with `cbind()` or by rows with `rbind()` requires compatible dimensions.

```
> b <- matrix(20:1, nrow=5, byrow=T)
> b
     [,1] [,2] [,3] [,4]
[1,]   20   19   18   17
[2,]   16   15   14   13
[3,]   12   11   10    9
[4,]    8    7    6    5
[5,]    4    3    2    1
> is.matrix(b)
> is.matrix(a)
> t(b)
> cbind(a, t(b))
> rbind(t(a), b)
```

A *list* in R is an ordered collection of elements that, in contrast to vectors, *may be of different types*. Lists are created using the `lis()` function. Note the difference in selecting elements: The `[[]]` operator returns the element at that position in the list, whereas the `[]` operator contains a list that contains the element at that position in the list.

```
> l <- list('a', 3, 'b', 2, TRUE)
> l[[2]]
> l[2]
> is.list(l)
> is.list(l[[2]])
> is.list(l[2])
> as.list(vv)
```

A *data frames* are the most widely used data structure for data analytics and statistics in basic R. It is essentially a table with a set of columns whose elements are of the same type. Columns are named and columns can be selected using the `$` symbol. Useful functions on data frames are `summary()`, `head()` and `tail()`. The following R code block creates a variable `x` as a vector of 50 normally distributed random values using the `rnorm()` function. The variable `y` is created from vector `x` and additional normally distributed random variables. The two are then combined into a data frame. The `colnames()` function can retrieve the column names, but can also be used to change/update the column names. The `nrow()` and `ncol()` functions return the number of rows and columns, `head()` and `tail()` return the first few or last few rows, and `cov()` is an example of a statistical function that returns the covariance matrix of all columns in the data frame.

```
> x <- rnorm(50)
> y <- 2*x + rnorm(50)
> data <- data.frame(x, y)
> colnames(data)
> colnames(data) <- c('Pred', 'Crit')
> nrow(data)
> ncol(data)
> data$Pred
> summary(data)
> head(data)
> tail(data)
> cov(data)
```

Data frames may be written to CSV files and read from CSV files, as shown in the following R code block. Both functions, `write.csv()` and `read.csv()` have a range of options for reading/writing files with or without header lines, different separators, for skipping rows, different decimal points, whitespace stripping, etc. Consult the R built-in help system for details.

```
> write.csv(data, 'data.csv', row.names=FALSE)
> new.data <- read.csv('data.csv')
> colnames(new.data)
```

dplyr	Manipulate data
forcats	Work with categorical variables (factors)
ggplot2	Grammar of Graphics
lubridate	Date and time parsing and arithmetic
purrr	Functional programming
readr	Read files in various formats
stringr	Work with character strings
tibble	A tibble is better than a table
tidyr	Make data tidy

Table 1: Tidyverse packages for R

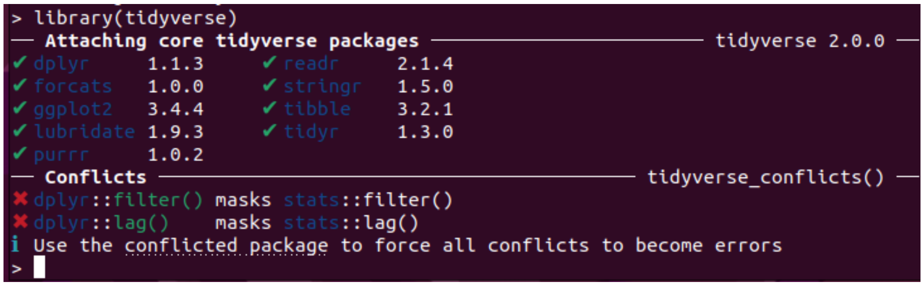
6 Tidyverse

The Tidyverse is a collection of R libraries designed for data science that share an underlying design philosophy, grammar, and data structures. Developed by Hadley Wickham and others, the Tidyverse libraries are built to work together seamlessly, making data science tasks more straightforward and intuitive. At the core of Tidyverse's philosophy is the concept of ”tidy data,” which arranges data in a structured way that simplifies analysis. This structure involves organizing data into rows and columns where each variable is a column, each observation is a row, and each type of observational unit forms a table.

Key libraries in the Tidyverse include *ggplot2* for data visualization, *dplyr* for data manipulation, *tidyr* for data tidying, *readr* for reading data, *purrr* for functional programming, and *tibble* for providing a better version of a table data structure. In particular, ggplot2 allows for complex and aesthetically pleasing visualizations using a layered grammar of graphics (hence the name), while dplyr provides a set of tools for efficiently manipulating datasets, such as filtering rows, selecting columns, and aggregating data. tidyr helps in transforming messy data into a tidy format, making it easier to analyze and visualize. Table 1 contains a summary of the libraries.

The Tidyverse also emphasizes readability and expressiveness in code, which not only makes data analysis easier to write but also easier to read and understand. It has become a popular choice among data scientists and statisticians for its ease of use, efficiency, and the cohesive way it handles data analysis tasks. The integration of these packages under the Tidyverse umbrella simplifies the process of data manipulation, exploration, and visualization, greatly enhancing the productivity and effectiveness of data analysis in R.

Tip: An introduction to data science with the Tidyverse packages, directly from their authors, can be found here: `https://r4ds.hadley.nz/`



```
> library(tidyverse)
── Attaching core tidyverse packages ──────────────────────── tidyverse 2.0.0 ──
✔ dplyr     1.1.3     ✔ readr     2.1.4
✔ forcats   1.0.0     ✔ stringr   1.5.0
✔ ggplot2   3.4.4     ✔ tibble    3.2.1
✔ lubridate 1.9.3     ✔ tidyr     1.3.0
✔ purrr     1.0.2
── Conflicts ────────────────────────────────────────── tidyverse_conflicts() ──
✖ dplyr::filter() masks stats::filter()
✖ dplyr::lag()    masks stats::lag()
ℹ Use the conflicted package to force all conflicts to become errors
> 
```

Figure 2: Attaching the tidyverse packages in R

Loading and attaching the `tidyverse` library in R, using the `library(tidyverse)` function, loads all the associated core packages, as shown in Figure 2.

This section can give only a very brief outline of the capabilities of the tidyverse packages. The extensive documentation and various ”cheat sheets[1] provide additional details. This section focuses on the use of dplyr to analyze data from a set of CSV files representing the data of the Pagila database. The Pagila database[2] is a demonstration database originally developed for teaching and development of the MySQL RDBMS under the name Sakila[3]. Pagila is designed as a sample database to illustrate database concepts and is based on a fictional DVD rental store. It originally consists of several tables organized into categories like film and actor information, customer data, store inventory, and rental transactions. For this section, the Pagila data was summarized in a few related CSV files.

When reading CSV files with readr, the data is stored in a *tibble*, not a data frame. A tibble provides a number of extensions and convenience operations that make it significantly more capable than a data frame. When using data frames with dplyr, they are automatically converted to tibbles.

The following R code reads a CSV file using the `read_csv()` function and prints the first few lines and a summary. The output looks slightly different than that for data frames, but accomplishes essentially the same things.

```
rentals <- read_csv('rentals.csv')
head(rentals)
summary(rentals)
```

Attaching a tibble or data frame with `attach()` means that its columns become variables in the R workspace and need not be selected from the tibble (or data frame) using the $ operator. The following R code block transforms the data read from the CSV file

[1] https://posit.co/resources/cheatsheets/

[2] https://github.com/devrimgunduz/pagila, https://github.com/devrimgunduz/pagila/blob/master/LICENSE.txt

[3] https://dev.mysql.com/doc/sakila/en/, https://dev.mysql.com/doc/sakila/en/sakila-license.html

into the rentals tibble into appropriate data types, then detaches the tibble and prints a summary.

```r
# Fix the column datatypes:
attach(rentals)
rating <- as.factor(rating)
language <- as.factor(language)
customer_address <- as.integer(customer_address)
customer_store <- as.integer(customer_store)
rental_staff <- as.integer(rental_staff)
payment_staff <- as.integer(payment_staff)
rental_duration <- as.integer(rental_duration)
detach(rentals)
summary(rentals)
```

The tidyverse libraries make extensive use of the *pipe* operator in R. The pipe operator allows chaining of function calls and plugs the result of one function as the first argument into the next function. Originally, tidyverse used the `%>%` pipe operator from the `magrittr` library, but can now also be used with the new (since R version 4.1), native R pipe operator `|>`. For simple usage, the two behave identically and can be intermixed.

The following R code block demonstrates a simple sequence of data manipulation operations using functions from the dplyr library. It begins with the data tibble which is piped into the first function. The outputs of each function are piped into the following function, ending with `print()`. Note that print output can also be piped into other functions, allowing printing of intermediate results.

```r
rentals |>
  filter(if_any(everything(), is.na)) |>
  select(last_name, rental_date, return_date, title, amount) |>
  print(n=Inf, width=Inf)
```

- The `filter(if_any(everything(), is.na))` function is the first in the pipeline. It filters rows in the rentals data frame based on the presence of NA (missing) values. The `if_any()` function checks each column (indicated by `everything()`) for NA values. The filter function then retains only those rows that have at least one NA value in any column.
- Next, the `select()` function specifies the columns to retain in the resulting data frame. It narrows down the data frame to include only the last_name, rental_date, return_date, title, and amount columns. This step reduces the dataset to focus on these key variables, for easier analysis and reporting.
- Finally, the `print(n=Inf, width=Inf)` function displays the output. The `n=Inf` argument tells R to print all rows of the data frame, instead of just the first few rows as is the default behavior. Similarly, `width=Inf` ensures that all

`full_join`	Joins tibbles (also `outer join`, `left_join`, `inner_join`, `right_join`)
`filter`	filters by row
`select`	selects columns to retain
`mutate`	creates new columns
`rename`	renames columns
`distinct`	finds unique values
`group_by`	groups data
`nest`	nests data, tibbles in tibbles
`arrange`	sorts data rows
`relocate`	moves data columns
`print`	prints a tibble

Table 2: Important dplyr functions

columns are printed without any being truncated, which is useful for wide tibbles or data frames.

In summary, this R code example is used to filter a rentals table for rows containing missing values in any column, and then to select and print specific columns of interest. This kind of operation is typical in data cleaning and exploratory data analysis processes. The result shows that some films have not been rented (i.e. there is no rental date for them), and some rentals have not been returned (i.e. there is no return date for them).

The following paragraphs show examples of further data analysis with Tidyverse, introducing additional dplyr functions and their use. Dplyr functions are intended to mirror the SQL queries from the earlier chapter on relational databases. The main dplyr ”verbs” used in the examples are summarized in Table 2.

Example: Find all films and the actors that appeared in them, ordered by film category and year, for those films that are rated PG.

```r
actors <- read_csv('actors.categories.csv')

rentals |>
  full_join(actors, by='title',
    suffix=c('_customer', '_actor'),
    relationship='many-to-many') |>
  filter(rating == 'PG') |>
  mutate(actor =
    paste(last_name_actor, ', ', first_name_actor, sep='')) |>
  rename(year=release_year) |>
  select(actor, title, category, year) |>
  distinct(actor, title, category, year) |>
  group_by(category, year, title) |>
  nest() |>
  arrange(category, year, title) |>
  relocate(category, year, title) |>
  print(n=Inf, width=Inf)
```

This R code processes the rentals tibble and the actors tibbles through a sequence of functions in a pipeline.

- The `read_csv()` function is used to read data from a CSV file named ”actors.categories.csv” into an R data frame called ”actors”.
- The ”rentals()” data frame is combined with the ”actors” data frame using a full join. The join is performed on the ”title” column common to both data frames. The `suffix` argument adds distinct suffixes to column names from each data frame to avoid name clashes. The `relationship=’many-to-many’` indicates the nature of the join.
- The `filter()` on the combined data retains only rows where the ”rating” column has the value ”PG”.
- The `mutate()` function is used to create a new column named ”actor”, which concatenates the actor’s last name and first name, separated by a comma and a space.
- The ”release_year’ column is renamed to ’year’ using the `rename` function.
- The `select()` function is used to narrow down the data frame to only the columns ”actor”, ”title”, ”category”, and ”year”.
- Following this, the `distinct()` function ensures that only unique rows are retained, removing any duplicates.
- The data is grouped by ”category”, ”year”, and ”title”, and then `nest` is used to create a nested data frame, i.e. a dataframe where the actors for each group are in a list-valued columns.
- the `arrange()` sorts the data frame by ”category”, ”year”, and ”title”, while `relocate` moves these columns to the front of the data frame for easier viewing.

- Finally, the entire data frame is printed with all rows (`n=Inf`) and without truncating any columns (`width=Inf`).

Example: Find the most popular actors in the rentals in each city.

This R code block below involves combining multiple data frames and then manipulating and summarizing the data. It builds on the reantal and actor tibbles from the previous example and includes address information.

```r
addresses <- read_csv('addresses.csv')
addresses$phone <- as.character(addresses$phone)

full_data <-
  rentals |>
    inner_join(addresses, by=c('customer_address'='address_id')) |>
    inner_join(actors, by='title',
      suffix=c('_customer', '_actor'),
      relationship='many-to-many')

full_data |>
  mutate(actor =
    paste(last_name_actor, ', ', first_name_actor, sep='')) |>
  group_by(city, actor) |>
  summarize(count=n()) |>
  mutate(ranking = min_rank(desc(count))) |>
  filter(ranking < 4) |>
  arrange(city, ranking, actor) |>
  print(n=25)
```

- The analysis starts by reading a CSV file containing addresses into a data frame.
- An inner join is first performed between these two data frames, matching them on a specified key.
- This is followed by another inner join with an ‘actors‘ data frame. This second join involves a many-to-many relationship and adds suffixes to overlapping column names to distinguish them.
- With the full data set, a new column is created by concatenating the first and last names of actors, forming complete names.
- The data is then grouped by city and actor.
- A new summary column is created that counts the number of occurrences (records) for each group.
- To create rankings, a new column is added that ranks the groups based on the count in descending order. The `min_rank()` function allows ties in the ranking, use `rank()` to break ties with gaps in ranking or `dense_rank()` to break ties with no gaps in ranking.

- The data is then filtered to include only those records with a ranking less than 4, focusing on the top three ranks for each group.
- Finally, the data is sorted by city, ranking, and actor and then printed.

Example: Find the customers who spend the most on rentals, with their phone numbers and cities, and the number of rentals with the higest total rental payments for each category grouped by rental duration.

```
full_data |>
   mutate(customer= paste(first_name_customer, last_name_customer)) |>
   select(customer, amount, rental_duration, category, phone, city) |>
   group_by(category, rental_duration, customer ) |>
   mutate(payments=sum(amount), num_rentals=n()) |>
   select(-amount) |>
   group_by(category, rental_duration) |>
   mutate(ranking = min_rank(desc(payments))) |>
   slice(which.min(ranking)) |>
   print(n=Inf, width=Inf)
```

By now, it should be clear what the functions in the analysis pipelines accomplish. However, a few interesting things to note. First, there is no `summarize()` function because `summarize()` omits all non-grouped columns, but the example requires phone numbers and citites of customers. Either these would need to be included somehow in the `summarize()` function, or as is done in this R code, summary columns are created with `mutate()`. Second, note the ”negative” argument to the `select()` function, which is used to remove the ”amount” column. Third, the pipeline uses multiple `group_by()` statements with different aggregate functions (`sum()`, `n()`, `min_rank()`) for the different groups. Finally, the R code uses `slice()` to select the rows with the smallest ranks.

Example: Get the total rental revenue, number of rentals, and the mean and standard deviation of the rental amounts for each country.

```
full_data |>
  group_by(country) |>
  summarize(revenue=sum(amount),
            numrentals=n(),
            mean_amount=mean(amount),
            sd_amount=sd(amount)) |>
  arrange(desc(mean_amount),
          desc(revenue)) |>
  print(n=Inf, width=Inf)
```

The R code for this query demonstrates a number different aggregate summary functions, `sum()`, `n()`, `mean()` and `sd()` (standard deviation). It also shows how to use the `desc()` function to arrange or sort data in decreasing order.

Example: Get the top 5 and the bottom 5 grossing customers for each quarter.

```
full_data |>
  mutate(customer=paste(first_name_customer,last_name_customer)) |>
  mutate(q=as.character(quarter(rental_date, with_year=T))) |>
  select(customer, q, amount, rental_date) |>
  group_by(q, customer) |>
  mutate(payments=sum(amount)) |>
  select(-amount) |>
  distinct(customer, q, payments) |>
  group_by(q) |>
  mutate(rank_top = min_rank(desc(payments))) |>
  mutate(rank_bot = min_rank(payments)) |>
  filter(rank_top < 6 | rank_bot < 6) |>
  arrange(q, desc(payments)) |>
  relocate(q, customer, payments, rank_top, rank_bot) |>
  print(n=Inf, width=Inf)
```

The code for this query again does not use a `summarize()` function. It also shows the use of the `quarter()` function from the ”lubridate” library. The lubridate library contains a large range of date and time related functions. Two ranking columns are created using the `mutate()` and `min_rank()` functions, once in descending order to get the top ranks, and again in ascending order to get the bottom ronks. The code uses `filter()` instead of `slice()` to select the top and bottom 5 ranks, uses `arrange()` to sort the data, and then uses `relocate()` to re-arrange the order of columns prior to printing.

Example: Find the set of film titles by rental customer and the total number rentals for each customer.

```
full_data |>
  mutate(customer=paste(first_name_customer,last_name_customer)) |>
  select(customer, title) |>
  nest(titles=title) |>
  rowwise() |>
  mutate(rentals=nrow(titles)) |>
  mutate(unique_titles=list(distinct(titles))) |>
  select(-titles) |>
  arrange(customer) |>
  print(n=Inf, width=Inf)
```

The code for this query works with nested data, that is, data with columns that contain lists, created using the `nest()` function. In this example, `nest(titles=title)` creates a columns called ”titles” that contains a list of all the elements of the ”title” column for each customer. The R code also demonstrates row-wise operations. Both `mutate()` functions after `rowwise()` function operate by row. Specifically, the first use of the `mutate()` function creates a new column ”rentals” which contains the number of rows in the titles column *for each row* (recall that the ”titles” column contains lists of film title). Similarly, the second use of the `mutate()` function creates a new column ”unique_title”that contains a list of distinct film titles from the ”titles” column *for each row*.

7 SQL and R

The ”sqldf” library in R allows users to perform SQL queries on R data frames. Essentially, it provides a bridge between SQL and R. This integration allows users who are familiar with SQL to leverage its powerful querying capabilities directly on R data structures, without the need to switch between different tools or environments.

One of the main advantages of ”sqldf” is its ability to handle large data frames more efficiently than some of R’s native functions. By utilizing SQL queries, users can perform complex data manipulations and aggregations with ease. The package supports various SQL commands including SELECT, JOIN, ORDER BY, and GROUP BY, among others, enabling a wide range of data operations that are familiar to SQL users.

Under the hood, ”sqldf” operates by temporarily converting data frames into databases, typically by creating an in-memory SQLite database, or, alternatively, using an existing database connection to any of a variety of RDBMS such as PostgreSQL. It then creates a table for each data frame, moves the data to the database tables, and executes SQL statements. It then moves the result set back into R as a data frame. This seamless process allows for a smooth integration of SQL’s data processing capabilities within the R environment.

”sqldf” is particularly useful for R users who are already comfortable with SQL syntax and for complex data manipulation tasks that might be more cumbersome or less intuitive in R’s native syntax. Its ability to handle data frames as if they were SQL tables makes it a highly valuable tool for data analysts and statisticians who work with large datasets and require the flexibility and power of SQL within the R programming environment.

The following R code block shows a very simple example. Note that the SQL `FROM` clause recognizes data frame names; any columns used in the SQL query must be named columns from those data frames.

```r
library(sqldf)
result_df <- sqldf('select distinct(title) from full_data')
```

When faced between the choice of data analytics using an SQL RDBMS or R/Tidyverse, there are a number of issues to consider:

- *Size of data*: R is limited by the amount of main memory of the computer. While large computers may offer 128GB or more, modern RDBMS can scale massively larger, in particular when distributing databases across a cluster of computers.

- *Access speed*: RDBMS have sophisticated indexing of tables and query planners that optimize complex queries for performance. While a dplyr analysis pipeline can also be optimized by carefully considering the order of function calls, the onus is on the data analyst to do this, while an RDBMS offer this ”out-of-the-box”.
- *Currency*: Using an RDBMS means that analytics can be performed on operational data, that is, the most current and up-to-date data. In contrast, the use of R involves first exporting data from the operational system and then analyzing it at a later time. However, while tempting, it is not generally recommended to perform complex analytics on an operational database, as it can significantly affect performance.
- *Transactions*: An RDBMS ensures consistent views of data across multi-user, concurrent updates. This means that, when using an operational database, the analysis sees consistent data, whereas an exported snapshot of the data may not necessarily be consistent, depending on the export mechanism.
- *Tools*: R has tools for statistical analysis and visualization, beyond mere reporting. So far, we have considered only simple descriptive analytics. However, when the data is to be used for sophisticated statistics or predictive analytics, it is no longer possible to do this on RDBMS.

These issues motivate the following recommendations:

- Do not ”hit” operational RDBMS for heavy-weight or frequent analytics. While it may be fine to do the occasional summary analytics on an operational database, this should not be normally done.
- Regularly export consistent data from RDBMS. If up-to-date data is needed, automate the export from the database to occur at regular intervals. However, note also that exporting data has a performance impact on operational databases.
- Sometimes, SQL may be the more intuitive language to specify the required analysis. In these cases, use separate in-memory or on-disk RDBMS for analytics (e.g. with `sqldf`) if desired/required.
- Finally, if the size of data is too large to handle in R, consider distributed tools such as Hadoop/Spark that are made for Big Data analytics.

Hands-On Exercises

The following hands-on exercises are designed to familiarize you with the Tidyverse packages, especially the dplyr package. Use these exercises with the Pagila CSV data set.

1. Find all films with a rating of ’PG’
2. List all customers who live in Canada (with their address)
3. Find the average *actual* rental duration for all films
    - This requires date arithmetic, use the `lubridate` package
4. Find the average overdue time for each customer
    - This requires date arithmetic, use the `lubridate` package
5. List all films that have never been rented
6. List the names of actors who have played in more than 15 films