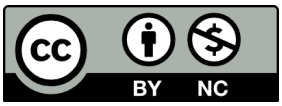
Business 4720

Introduction to Data Management with R

Joerg Evermann



Learning Goals

After reading this chapter, you should be able to:

- Create and manipulate basic data structures in R, including arrays, matrices, and data frames.
- Create summary information from R data frames and other data structures.
- Use the Tidyverse packages to retrieve information from R data frames, including filtering, grouping, and aggregation of information.
- Use SQL to operate on R data frames to retrieve information, including filtering, grouping, and aggregation of information.

Sources and Further Reading

The material in this chapter is based on the following sources.

Resources

Venables, W.N., Smith, D.M. and the R Core Team (2024) An Introduction to R.

https://cran.r-project.org/doc/manuals/r-release/R-intro.pdf

This tutorial is a good, easy, and comprehensive introduction to R. It covers the most important aspects of R in 100 pages. It is continuously updated by the R team for new versions of R and new features. Chapters 1 through 7 cover the initial material in this module.

Resources

Wickham, H., Cetinkaya-Rundel, M., and Grolemund, G. (2023) R for Data Science. O'Reilly Media, Inc.

https://r4ds.hadley.nz/

This is a free, open-source book directly from the authors of the popular and widely-used Tidyverse libraries. With many examples, and easy-to-understand descriptions, the book provides a solid foundation for business analytics with R. It provides sections on data import, data transformation, data visualization, and data communication, each with multiple chapters. This is a highly relevant resource for anyone doing data science in R.

Resources

Posit (2024) Cheatsheet.

https://posit.co/resources/cheatsheets/

Posit is the maker of the popular RStudio programming environment for R, the shiny library for building web applications and graphical dashboards with R, and a number of other open-source and proprietary tools around the R system. Posit provides a number of very helpful ”cheatsheets” that are one-page summaries of various R libraries. The ”cheatsheets” useful for this module is the one on the dplyr package for data transformation.

1 Introduction

R is a statistical software system and programming language known for its robust capabilities in data analysis, visualization, and statistical computing. It was originally developed 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 created a vibrant community of users and developers, continuously enhancing its functionality through comprehensive packages and extensions. The Comprehensive R Archive Network (CRAN) is a repository of these packages, 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

Figure 1: The R command line interface

of the work. Ideally, data analysts will assemble an R script file for their entire data analysis, from reading the raw data sets to the 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.

Tip

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 and edit your commands easily, then copy/paste to R for execution.
- Use a notepad app for your results, copy/paste from R.
- The Ubuntu terminal 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:

R

```
> 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 explicitly use the `print` function.

R

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

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

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

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

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

> **Tip**
>
> The boolean constants `TRUE` and `FALSE` can be abbreviated by `T` and `F`

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.

```R
> 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 type of the value and to coerce, that is, to change, the type of a value.

```R
# Check whether it is numeric
is.numeric(2)
# Check whether it is an integer number
is.integer(3.14)
# Make it an integer
as.integer(3.14)
# Make it a character string
as.character(3.14)
# Make it a character string
as.character(TRUE)
# Make it a number
as.numeric('3.1415')
```

4 The R Environment

The collection of variables, functions, and libraries that exists in R at any one time is called the R *workspace*. The functions `ls()` and `rm()`, named after their Unix bash shell equivalents, list and remove objects from the R workspace.

R
```
# Show objects in workspace
> ls()
[1] "a"           "b"
# Remove one object
> rm(b)
> ls()
[1] "a"
```

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()`.

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

R has a current 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 (`getwd()`), to set (change) it (`setwd()`, and to list the files in the working directory (`list.files()`):

R
```
# Get the working directory
getwd()
# Set the working directory
setwd('DataSets')
getwd()
# List files in working directory
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()` function with the name (or a vector of names) of packages to install from CRAN. Once installed, libraries can be attached to the R workspace with the `library()` function, which makes its functions and any data sets it contains available for use.

R
```r
# Install the tidyverse library
install.packages('tidyverse')
# Load/attach the tidyverse library
library(tidyverse)
# List all installed packages
installed.packages()
```

It is sometimes useful to assemble a series of related R commands for a particular data analysis task in a script file. Script files are useful to ensure that the analysis can be repeated. 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.

R
```r
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.

R
```r
quit()
```

5 Vectors

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.

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

In the above example, 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 repetition function `rep()` which repeats its first argument the number of times specified by its second argument.

R
```r
# Generate a sequence
s <- seq(0, 6, by=.5)
print(s)
# Repeat a value
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. 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.

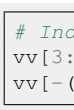
R
```r
length(v)
max(v)
min(v)
sqrt(v)
var(v)
sd(v)
# Vectors get flattened
vv <- c(v, c(7, 8, 9), v)
print(vv)
```

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 (true or false). Indexing the variable `vv` with that vector determines which elements of `vv` to select (those elements for which the corresponding entry is true).

R
```r
vv < 5
vv[vv < 5]
vv[vv < 5] <- vv[vv < 5] + 5
```

Vectors can also be indexed numerically, selecting elements by their position. This is also called "Slicing". R allows you select a sequence of values with the `:` operator, and the exclusion of elements, or sequences of elements, using $-$. The first line in the following example selects elements at positions 3 through 7, the second line selects elemnts *except* those at positions 3 through 7.

```R
# Indexing is inclusive
vv[3:7]
vv[-(3:7)]
```



Important

- R begins indexing positions with 1, while other programming languages begin at 0.
- Indexing in R is inclusive, that is, both the start and end index are included.

Missing values are designated by the special symbol `NA`. The `is.na()` function can be used to identify and select `NA` and then filter them. When an aggregation function encounters a `NA`, it will normally return a `NA`, so it is important to remove or replace `NA` values before analyzing the data. 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.

```R
# Introduce a missing value
v[3] <- NA
# Missing value arithmetic
v*3
# Testing for missing values
is.na(v)
# Missing values in aggregate functions
sum(v)
sum(v, na.ram=TRUE)
```

Important string functions are `grep()` and `agrep()`. The `grep()` function checks whether each string in a vector of strings contains a substring that matches a regular expression, and returns the indices of strings in the vector that match the pattern. 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.

```R
# Match/find North American phone numbers
> grep('^([0-9]{3})[ -]?[0-9]{3}[ -]?[0-9]{4}$',
    c('709 864 5000', 'abc def 9999', '709-865-5000'))
[1] 1 3
# Match/find Canadian postal codes
> grep('[A-V][0-9][A-V] [0-9][A-V][0-9]',
    c('A0P 1L0', '0AB L2K', 'A0X 1Z0'))
[1] 1
```

The `agrep()` function calculates the Levenshtein distance between a regular expression and a vector of strings and returns the indices of those strings that are within a

certain distance. The Levenshtein distance is defined as the sum of insertions, deletions, and substitutions of characters to transform one string into another.

R
```
# Match/find strings up to Levenshtein distance of 3
> agrep('apple',
    c('apricot', 'banana', 'grape', 'pineapple'),
    max.distance=3)
[1] 1 3 4
```

6 Arrays, Matrices, 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 ("Slicing") generalizes from vectors to matrices and arrays simply by indexing each dimension with the same syntax as used for vectors. The `array()` function 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:

- The first dimension is the row, the second is the column
- Initially, the array is created from a range of numbers between 1 and 20, and the `dim` argument specifies the dimensionality.
- R fills arrays by column, unless otherwise specified
- A dimension need not be slided or indexed, as in `a[,2]` or `a[,2:4]` which do not subset the first dimension (rows). The result is that all rows are returned in these examples.
- Reversing the index reverses the result that is returned, as in `a[3:1,2:4]` which reverses the indexing of the first dimension (rows).

R
```
# Default is to fill by column
a <- array(1:20, dim=c(4,5))
a
# Result:
#      [,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
# Indexing is inclusive and starts at 1
a[,2]
a[,2:4]
a[3,2:4]
a[3:1,2:4]
```

Constructing a *matrix* with the `matrix()` function is similar to constructing an array, but instead of providing the dimensionality with `dim`, one must provide either the number of rows or columns (`nrow` or `ncol`) and how to fill the matrix from the elements provided using the `byrow` argument; by default, R fills arrays and matrices by column.

The `t()` function returns the transpose of a matrix, that is, it reverses rows and columns. Binding (combining) two matrices together by columns with `cbind()` or by rows with `rbind()` requires compatible dimensions, that is, same number of rows for `cbind()` or same number of columns for `rbind()`.

R

```r
b <- matrix(20:1, nrow=5, byrow=T)
b
# Result:
#      [,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
# Test if it is a matrix
is.matrix(b)
is.matrix(a)
# Transpose
t(b)
# Bind (combine) by columns
cbind(a, t(b))
# Bind (combine) by rows
rbind(t(a), b)
```

A *data frame* is the most widely used data structure for data analytics and statistics in 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 then adds a normally distributed random variable. The two are then combined into a data frame with the `data.frame()` function. The `colnames()` function retrieves the column names, but can also 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.

```R
# Create a vector of 50 normally distributed random variables
x <- rnorm(50)
# Create another vector with random variables
y <- 2*x + rnorm(50)
# Create a data frame from the two vectors
data <- data.frame(x, y)
#  Get the column names
colnames(data)
# Update the column names
colnames(data) <- c('Pred', 'Crit')
# Get the number of rows and columns
nrow(data)
ncol(data)
# Get the "Pred" column of the data frame
data$Pred
# Print a summary
summary(data)
# Print first and last rows
head(data)
tail(data)
# Calculate the covariance matrix
cov(data)
```

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

```R
# Write CSV file into current working directory
# Omit the row names
write.csv(data, 'data.csv', row.names=FALSE)
# Read the data from the current working directory
new.data <- read.csv('data.csv')
```

Hands-On Exercise

These 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. Create an array with 3 columns and 50 rows of random numbers with mean of 2 and standard deviation of 4 (use the `rnorm()` function)
2. Create a dataframe from the array and name the columns as "A", "B", "C"
3. "Clip" the values so that all values lie between -3 and $+7$
4. Summarize the data
5. Print the pairwise covariance matrix of the three columns in the data frame
6. Find the square root of each of the diagonal entries of the covariance matrix, compare this to the standard deviation of 4. Tip: Use the `diag()` function.
7. Save the data frame in a CSV file using your first name as file name (file ending '.csv')

7 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.

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,

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

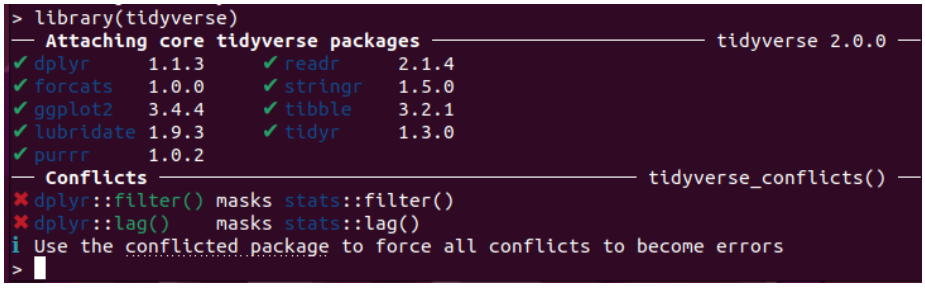


Figure 2: Attaching the tidyverse packages in R

and visualization, greatly enhancing the productivity and effectiveness of data analysis 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.

Tidyverse Basics

This subsection introduces basic ideas of data manipulation with the Tidyverse package, primarily using the dplyr package. A simple data set is used to illustrate basic functionality. The example dataset for this section is the Fuel Consumption Ratings for battery electric vehicles, provided the Government of Canada through its Open Government Portal[1]. The dataset contains the variables shown in Table 2.

Reading Data: 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 much more capable than a data frame. When using data frames with any dplyr function, they are automatically converted to tibbles.

[1] https://open.canada.ca/data/en/dataset/98f1a129-f628-4ce4-b24d-6f16bf24dd64

Column	Data Type	Definition
Make	Categorical (string)	Manufacturer
Model	Categorical (string)	Model name
Year	Numeric	Model year
Category	Categorical (string)	Small, Midsize, Large, Pickup, SUV, Station Wagon, etc.
City	Numeric	Consumption in l/100km equiv.
Hwy	Numeric	Consumption in l/100km equiv.
Comb	Numeric	Consumption in l/100km equiv.
Range	Numeric	Driving range in km

Table 2: Fuel efficiency data set variables

The following R code reads a CSV file using the `read_csv()` function from the Tidyverse readr library (not the `read.csv()` function from basic R!) and prints the first few lines and a summary.

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 much more capable than a data frame. When using data frames with any dplyr function, they are automatically converted to tibbles. The output looks slightly different than that for data frames, but accomplishes essentially the same things.

R

```r
# Read CSV into a Tibble
data <- read_csv('https://evermann.ca/busi4720/fuel.csv')
# Examine the data
# Dimensions (rows, columns)
dim(data)
# Column names
colnames(data)
# Summary
summary(data)
```

The functionality in the dplyr library is intended to mirror SQL queries from the earlier module on relational databases. The main dplyr "verbs" used in the examples are summarized in Table 3. The examples below will illustrate each of these "verbs" or functions and also show the equivalement SQL statements.

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.

Filtering: The example below starts with the tibble that was read earlier and pipes it into the `filter()` function. The filter function specifies two conditions, filtering

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

Table 3: Important dplyr functions

data by vehicle Make and vehicle model Year. The output of the `filter` function is piped into the `print()` function.

R

```r
# Pipe a data frame or tibble into a filter and print results
data |>
  filter(Make=='Ford', Year==2023) |>
  print()

# Equivalent without pipe and boolean operator in filter
filter(data, Make=='Ford' & Year==2023)
```

The `filter()` function is similar to the `WHERE` clause in an SQL SELECT statement and the above R code is equivalent to the following SQL statement:

SQL

```sql
SELECT *
   FROM data
   WHERE Make=='Ford' AND Year==2023;
```

Selection: The following example adds the `select()` function to select specific columns from the data set:

```R
# Pipe a data frame or tibble into a filter,
# select specific columns and print results
data |>
  filter(Make=='Ford', Year==2023) |>
  select(Model, Category, Range) |>
  print()
```

There is no equivalent SQL clause for the `select()` function because the selection of result columns occurs in the main SELECT clause, as illustrated in the following equivalent SQL statement:

```SQL
SELECT Model, Category, Range
   FROM data
   WHERE Make=='Ford' AND Year==2023;
```

Column Creation: The next example shows the use the `mutate()` function to create a new column that expresses the range of a vehicle for highway driving. It involves a calculation using the Hwy fuel economy, the Comb combined fuel economy and the Range. The `mutate()` function is added into the above data manipulation pipeline:

```R
# Pipe a data frame or tibble into a filter,
# create a new calculated column,
# select specific columns and print results
data |>
  filter(Make=='Ford', Year==2023) |>
  mutate(HwyRange = Range * Comb / Hwy) |>
  select(Model, Category, Range, HwyRange) |>
  print()
```

An equivalent SQL statement constructs the new column in the main SELECT clause:

```SQL
SELECT Model, Category, Range, (Range*Comb)/Hwy AS HwyRange
   FROM data
   WHERE Make=='Ford' AND Year==2023;
```

Renaming: The example below shows renaming of columns using the `rename()` function. The Range column is renamed to be the CombRange column. While `mutate()` creates an additional columns, `rename()` does not and only renames an existing column. It is therefore more efficient for renaming.

```R
# Pipe a data frame or tibble into a filter,
# create new calculated columns,
# rename an existing column,
# select specific columns and print results
data |>
  filter(Make=='Ford', Year==2023) |>
  mutate(HwyRange = Range * Comb / Hwy) |>
  mutate(CityRange = Range * Comb / City) |>
  rename(CombRange = Range) |>
  select(Model, Category, CombRange, CityRange, HwyRange) |>
  print()
```

An equivalent SQL statement constructs the new column in the main SELECT clause using the AS keyword:

```SQL
SELECT Model, Category,
      Range AS CombRange,
      (Range*Comb)/Hwy AS HwyRange,
      (Range*Comb)/City As CityRange
   FROM data
   WHERE Make=='Ford' AND Year==2023;
```

Distinct Values: The `distinct()` function can be used to find the set of unique values of a column. For example, the following R code identifies the unique vehicle Makes and Models in the data set.

```R
# Pipe a data frame or tibble into a filter,
# select distinct value combinations and print results
data |>
  distinct(Make, Model) |>
  print()
```

SQL uses the DISTINCT keyword for this purpose as well, illustrated in the following example:

```SQL
SELECT DISTINCT Make, Model
  FROM data;
```

Ordering: The dplyr `arrange()` function is used to order data by values, in ascending or descending order. The following example illustrates this by adding the `arrange()` function to the data processing pipeline. The `arrange()` function orders the data first by vehicle Category in ascending order (the default ordering), then, for equal values of Model, by Range in descending order (using the `desc()` function:

```R
# Pipe a data frame or tibble into a filter,
# select specific columns,
# order the data and print results
data |>
  filter(Make=='Ford', Year==2023) |>
  select(Model, Category, Range) |>
  arrange(Category, desc(Range)) |>
  print()
```

An equivalent SQL statement uses the ORDER BY clause with the keywords ASC or DESC to indicate the direction of ordering:

```SQL
SELECT Model, Category, Range
   FROM data
   WHERE Make=='Ford' AND Year==2023
   ORDER BY Category ASC, Range DESC;
```

Relocating Columns: The function `relocate()` can be used to specify the order in which columns are returned. The following code example rearranges the column order before printing by adding the `relocate()` function to the data processing pipeline:

```R
# Pipe a data frame or tibble into a filter,
# select specific columns,
# order the data,
# move columns and print results
data |>
  filter(Make=='Ford', Year==2023) |>
  select(Model, Category, Range) |>
  arrange(Category, desc(Range)) |>
  relocate(Category, Range) |>
  print()
```

SQL does not require a function or keyword in the SELECT statements. There, columns are returned in the order in which they are specified in the SELECT clause:

```SQL
SELECT Category, Range, Model
   FROM data
   WHERE Make=='Ford' AND Year==2023
   ORDER BY Category ASC, Range DESC;
```

Grouping and Summarizing: The `group_by()` function can be used to group data in preparation for computing aggregate information. It is typically used with the `summarize()` function that calculates aggregates.

The following example shows how to compute mean fuel efficiency data, grouped by vehicle Make and vehicle Category. It adds both a `group_by()` and a `summarize()` function into the data processing pipeline. The summarized data is then filtered again, it is sorted by vehicle Category and mean combined fuel economy, and the columns are then reordered so that vehicle Category and mean combined economy are the first two columns.

R

```r
# Pipe a data frame or tibble into a filter,
# group the data
# summarize the data,
# filter the summary information,
# order the data,
# relocate columns and print results
data |>
  filter(Year==2023) |>
  group_by(Make, Category) |>
  summarize(meanCity = mean(City),
            meanHwy = mean(Hwy),
            meanComb = mean(Comb),
            maxRange = max(Range),
            nVehicle = n()) |>
  filter(nVehicle > 1) |>
  arrange(Category, meanComb) |>
  relocate(Category, meanComb) |>
  print()
```

The equivalent SQL statement also uses the GROUP BY clause. Summary information is defined in the main SELECT clause. The second `filter()` function in the R code refers to aggregate information, so that the equivalent clause in SQL is the HAVING clause:

SQL

```sql
SELECT Category,
       AVG(Comb) AS meanComb,
       Make,
       AVG(City) AS meanCity,
       AVG(Hwy) AS meanHwy,
       MAX(Range) AS maxRange,
       COUNT(*) AS nVehicle
   FROM data
   WHERE Year==2023
   GROUP BY Make, Category
   HAVING COUNT(*) > 1
   ORDER BY Category ASC, meanComb ASC;
```

8 Advanced Tidyverse for Data Analysis

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. The following R code block reads the data and coerces data types as required.

R

```r
rentals <- read_csv('http://evermann.ca/busi4720/rentals.csv')
rentals$rating <- as.factor(rentals$rating)
rentals$language <- as.factor(rentals$language)
rentals$customer_address <- as.integer(rentals$customer_address)
rentals$customer_store <- as.integer(rentals$customer_store)
rentals$rental_staff <- as.integer(rentals$rental_staff)
rentals$payment_staff <- as.integer(rentals$payment_staff)
rentals$rental_duration <- as.integer(rentals$rental_duration)

actors <-
  read_csv('https://evermann.ca/busi4720/actors.categories.csv')

addresses <- read_csv('https://evermann.ca/busi4720/addresses.csv')
addresses$phone <- as.character(addresses$phone)
```

The following paragraphs show further examples of data analysis with Tidyverse and introduce additional dplyr functions or additional ways to use the dplyr functions. These example data processing pipelines mirror the example in the previous module on SQL. Only new aspects of the examples are explained.

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

[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

```R
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 joins the rentals data with the actors data using a full join. The join is performed on the title column that is common to both data frames/tibbles. 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 `mutate()` function creates a new column named actor, which concatenates the actor's last name and first name, separated by a comma and a space.

The data is grouped by category, year, and title, using the `group_by()` function, and the `nest()` function is used to create a nested data frame, i.e. a data frame where the actors for each group are in a list-valued column.

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 rental and actor tibbles from the previous example and adds address information.

```R
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 address information into a tibble. An inner join is performed between the rental data and the address data, matching them on a specified key in each data frame, that is, the cusomter_address in the rentals data frame must be equal to the address_id in the addresses data frame.

This is followed by another inner join with the actors data frame. No join key/column is specified as the join is done on columns that have the same name in both data frames. This second join involves a many-to-many relationship and adds suffixes to to distinguish columns with the same name in the two data frames.

The data is grouped by city and actor and 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 their count value 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.

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.

R

```r
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)
```

In this example, there is no `summarize()` function after the `group_by()` function because `summarize()` omits all non-grouped columns, but the example requires phone numbers and cities 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 using the `mutate()` function instead. Also note the "negative" argument to the `select()` function, which is used to remove the "amount" column. The data processing 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.

```R
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.

```R
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 <= 5 | rank_bot <= 5) |>
  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 the `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 (using a boolean "or" operator), 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.

```R
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*.

9 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 Exercise

These 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