Appendix A — Appendix

Bigger Data, Easier Workflows

AUTHOR
Nic Crane, Jonathan Keane, and Neal Richardson

A.1 Package Versions

Many R packages are under active development and occasionally updates can cause changes in compatibility. We've included the output of `sessionInfo()` so you can see the exact versions of the packages which were used to create the examples in this book.

```
sessionInfo()
```

```
R version 4.4.1 (2024-06-14)
Platform: x86_64-pc-linux-gnu
Running under: Ubuntu 22.04.4 LTS

Matrix products: default
BLAS:   /usr/lib/x86_64-linux-gnu/blas/libblas.so.3.10.0
LAPACK: /usr/lib/x86_64-linux-gnu/lapack/liblapack.so.3.10.0

locale:
 [1] LC_CTYPE=en_GB.UTF-8       LC_NUMERIC=C
 [3] LC_TIME=en_GB.UTF-8        LC_COLLATE=en_GB.UTF-8
 [5] LC_MONETARY=en_GB.UTF-8    LC_MESSAGES=en_GB.UTF-8
 [7] LC_PAPER=en_GB.UTF-8       LC_NAME=C
 [9] LC_ADDRESS=C               LC_TELEPHONE=C
[11] LC_MEASUREMENT=en_GB.UTF-8 LC_IDENTIFICATION=C

time zone: Europe/London
tzcode source: system (glibc)

attached base packages:
[1] stats     graphics  grDevices utils     datasets  methods   base

other attached packages:
[1] dplyr_1.1.4            arrow_17.0.0.100000267

loaded via a namespace (and not attached):
 [1] vctrs_0.6.5       cli_3.6.3         knitr_1.47        rlang_1.1.4
 [5] xfun_0.45         purrr_1.0.2       generics_0.1.3    assertthat_0.2.1
 [9] jsonlite_1.8.8    glue_1.7.0        bit_4.0.5         htmltools_0.5.8.1
[13] fansi_1.0.6       rmarkdown_2.27    tibble_3.2.1      evaluate_0.24.0
[17] fastmap_1.2.0     yaml_2.3.8        lifecycle_1.0.4   compiler_4.4.1
```

```
[21] pkgconfig_2.0.3   htmlwidgets_1.6.4 rstudioapi_0.16.0 digest_0.6.36
[25] R6_2.5.1          utf8_1.2.4        tidyselect_1.2.1  pillar_1.9.0
[29] magrittr_2.0.3    tools_4.4.1       bit64_4.0.5
```

You can find even more detailed information about your arrow build by calling the function `arrow_info()`, which prints out information about which version of the Arrow R package and Arrow C++ library you have installed.

It also provides information about which features the Arrow C++ library has enabled when built, and so if you're using a custom Arrow build, it can help you check you've got everything you need.

The output below shows information about the version of Arrow used to build this book.

```
arrow_info()
```

```
Arrow package version: 17.0.0.100000267

Capabilities:

acero      TRUE
dataset    TRUE
substrait FALSE
parquet    TRUE
json       TRUE
s3         TRUE
gcs        TRUE
utf8proc   TRUE
re2        TRUE
snappy     TRUE
gzip       TRUE
brotli     TRUE
zstd       TRUE
lz4        TRUE
lz4_frame  TRUE
lzo       FALSE
bz2        TRUE
jemalloc   TRUE
mimalloc   TRUE

Memory:

Allocator  mimalloc
Current   512 bytes
Max           64 Kb

Runtime:

SIMD Level          avx2
Detected SIMD Level avx2

Build:
```

```
C++ Library Version  18.0.0-SNAPSHOT
C++ Compiler                    GNU
C++ Compiler Version          11.4.0
```

A.2 Getting Started

A.2.1 PUMS dataset overview

One of the datasets we use throughout this book is the United States of America's Census Public Access Microdata dataset. This is a dataset that comes from a detailed survey that is sent out to a subset of US residents every year. The dataset is release for public use by the Census Bureau in a raw CSV form. We have cleaned it up and converted it to a Parquet-based dataset for use with Arrow for demonstration purposes in this book.

We chose this data because it is open access, somewhat familiar, but also large and diverse in scope. Most analyses using PUMS will filter to a single year, a single state, or specific variables to be able to run analyses in memory. And then if you want to run the same analysis on a different year or different state, you would run the same code again on a different subset and then compare together. With the power of the arrow R package and datasets, we can analyze the full dataset with all of the available years and states.

A.2.1.1 Getting the data

We offer a few different ways that you can get the data that we use in this book. There are tradeoffs to each, but they each should get you enough data to run the examples, even if it's not the entire full dataset.

A.2.1.1.1 Get a subset dataset

When writing the book we found it useful to have a small version of the dataset to test our code against. We have this dataset hosted in the GitHub repository under the releases: https://github.com/arrowrbook/book/releases

This subset only includes the person-level data for years 2005, 2018, 2021 and only for states Alaska, Alabama, Arkansas, Arizona, California, Washington, Wisconsin, West Virginia, and Wyoming.

Simply download it and unzip it into a directory called `data` in your working directory and you can run the examples in this book.

A.2.1.1.2 Download a full version from AWS S3

We also host a full version of the dataset in AWS S3. However, we have set this bucket to have the person who requests the download to pay for the transfer cost. This means that you cannot download the data without first creating an AWS account, configuring it, and you will be billed a very small amount for the cost of the data transfer. The way to configure this in AWS might change, but the AWS documentation have instructions for how to do this.

Once you have setup your AWS account and CLI, download the data into a `data` directory to use:

```
aws s3 cp --request-payer requester --recursive \
  s3://scaling-arrow-pums/ ./data/
```

This is the full dataset the book was built with, but does require that you setup an AWS account, configure it correctly, and pay the small transfer fee.

A.2.1.1.3 Download the raw data from the Census Bureau and record it yourself

We also have scripts that will download the raw data from the Census Bureau and do the recoding we started. Follow the instructions in the `README.md` file under `pums_dataset` in the github repository. There are also scripts for downloading the shape files PUMA_shp_to_parquet.py.

There are a few variables you should set, and you can control the amount of parallelism for downloading, unzipping, etc.

This is the full dataset the book was built with, but does require computational time to finish.

A.2.1.2 Dataset recoding

This dataset is a re-coding and enriching of the Public Use Microdata Sample (PUMS) collected and provided by the United States Census. It covers years 2005–2022 using the 1-year estimates (though 2020 is missing since that year's was only released in 5-year estimates due to COVID).

The raw data was retrieved from the Census's FTP site and the values to recode categorical and string data was retrieved from the Census's API (via the **censusapi** R package).

The data was recoded with the following general principles:

- If there were string values and there were less than or equal to 10 unique values, we converted these to factors.
- If there were string values and there were more than 10 unique values, we converted these to strings.
- We used integer or floats for values that were numeric in nature, and recoded special values (*eg* variable `RETP` "Retirement income past 12 months" where a value of -1 means "N/A (Less than 15 years old)") that are missing-like as `NA`. Note: there are also a number of values that are top and bottom coded—these are also converted to numerics (*eg* so a maximum value in those columns actually represents that value or larger; variable `WKHP` or "Usual hours worked per week past 12 months" which has a value of 99 marked as "99 Or More Usual Hours").
- If there were codes that broadly corresponded to `TRUE` and `FALSE` (e.g. "yes" and "no"), these were converted into booleans

The book Analyzing US Census Data: Methods, Maps, and Models in R has chapters dedicated to analyzing this kind of microdata with **tidycensus** package. Though the tidycensus package and approach will have slight differences from analyzing this data with arrow, the concepts and analytic approach will be the same.

Though we have not purposefully altered this data, this data should not be relied on to be a perfect or even possibly accurate representation of the official PUMS dataset.

A.2.1.3 Datasets and partitioning

There are two datasets, one at `s3://scaling-arrow-pums/person/` which has person-level data and another at `s3://scaling-arrow-pums/household/` which has household-level data.

Each of these datasets is subsequently partitioned by year and then by state/territory with prefixes like `year=2019/location=il` with Parquet files below that.

A.2.1.4 Using the PUMS dataset

A detailed description of how to analyze PUMS or other survey data is beyond the scope of this book, though if you're interested in learning more details, the book Analyzing US Census Data: Methods, Maps, and Models in R has chapters dedicated to analyzing this kind of microdata. But it's helpful to explore some examples.

The PUMS dataset comes from surveying around 1% of the US population. It also asks a number of sensitive questions, so the Census Bureau is careful to avoid accidentally identifying specific people in the dataset. For these two reasons, the dataset is actually not the raw responses—where each row is one respondent—but rather each row has a specific weight applied to it. This weight could be thought of as something along the lines of "this number of respondents responded with this set of answers" though it is more complicated than that. Because of this, in order to make estimates about populations, we need to use the weighting columns from the dataset which tell us how many people are represented in each row to get an accurate measure in our final calculations which is different from a typical tidy workflow where each row is a single individual and you can use simple aggregations across rows.

In sum, this dataset uses survey weights, so the individual rows do not represent a single individual. Instead, we must use the weight columns if we are counting people or calculating many statistics (measures of central tendency especially).

Let's look at an example, if we are doing an age breakdown for the state of Alaska, we might think we could do simply:

```
pums_person |>
  filter(location == "ak") |>
  mutate(
    age_group = case_when(
      AGEP < 25 ~ "24 and under",
      AGEP < 35 ~ "25–34",
      AGEP < 45 ~ "35–44",
      AGEP < 55 ~ "45–54",
      AGEP < 65 ~ "55–64",
      TRUE ~ "65+"
    )
  ) |>
  group_by(year, age_group) |>
  summarize(num_people = n()) |>
  arrange(year, age_group) |>
  collect()
```

year	age_group	num_people
<int>	<chr>	<int>
2005	24 and under	2366
2005	25–34	676

year	age_group	num_people
<int>	<chr>	<int>
2005	35–44	915
2005	45–54	1041
2005	55–64	637
2005	65+	494
2006	24 and under	2433
2006	25–34	713
2006	35–44	891
2006	45–54	1084

1-10 of 102 rows Previous **1** 2 3 4 5 6 … 11 Next

Looking at the results here is off, the numbers look way too low. If we add up the total of all age groups for 2021, we get: 6,411 which is far under the estimates of the 2021 population of 732,673.

But if we instead sum the person weight column (`PWGTP`) we get very different results:

```
pums_person |>
  filter(location == "ak") |>
  mutate(
    age_group = case_when(
      AGEP < 25 ~ "24 and under",
      AGEP < 35 ~ "25–34",
      AGEP < 45 ~ "35–44",
      AGEP < 55 ~ "45–54",
      AGEP < 65 ~ "55–64",
      TRUE ~ "65+"
    )
  ) |>
  group_by(year, age_group) |>
  summarize(num_people = sum(PWGTP)) |>
  arrange(year, age_group) |>
  collect()
```

year	age_group	num_people
<int>	<chr>	<int>
2005	24 and under	265268
2005	25–34	76200
2005	35–44	101715
2005	45–54	108320
2005	55–64	65431
2005	65+	41068
2006	24 and under	254235
2006	25–34	89767
2006	35–44	98676

	year	age_group	num_people
	<int>	<chr>	<int>
	2006	45–54	109716

1-10 of 102 rows Previous 1 2 3 4 5 6 … 11 Next

And here, if we do our sum for 2021 again, we get a number that matches the overall population for Alaska in 2021: 732,673.

A.2.2 Arrow data types

In the introduction, we mentioned that Arrow is designed for interoperability between different systems, and provides a standard for how to represent tabular data. In order to achieve this interoperability, Arrow defines a set of data types which cover the main data types used in different data systems. These data types are similar to those used in R but are not identical. In R, you may have previously encountered:

- integers (e.g. `1L`)
- numeric (e.g. `1.1`)
- complex (e.g. `1 + 1i`)
- character (e.g. `"a"`)
- factors (e.g. `factor("a")`)
- logical (e.g. `TRUE`)
- other types relating to dates, times of day, and durations

Arrow data types are similar to these, but some are more precise and also include some data types which don't exist in R. The Arrow data types are:

- **integers**: Arrow has multiple integer types which vary on whether they are signed—if they can be both positive and negative, or just positive—and how much space in memory they take up
- **floating point** numbers: these map to numeric values, and vary on how much space in memory they take up
- **decimal** numbers: these use integers to represent non-integer data with exact precision, to allow for more precise arithmetic
- **utf8** and **binary**: similar to R's character vectors
- **dictionaries**: similar to R factors
- **boolean**: equivalent to R logical values
- **datetimes** and **dates**
- **durations**
- **time** of day

A.2.2.1 Bit-width sizes

Another aspect of these Arrow data types is that some of them can come in different sizes. For example, integers can be 8, 16, 32, or 64 bits. They can also be signed (can be positive of negative) or unsigned (only positive). The size of an integer refers to how much space it takes up in memory, and the practical impact of this is range of values it can hold. Eight bit values can take up 2^8 bits, which comes to 256. This means that an unsigned 8-bit integer can be any value between 0 and 255, and a signed 8-bit integer can be any value between -128 and 127.

There is a trade off between the number of bits and that size of numbers that can be represented. For example, if you have a column that never has values over 100, using an 8-bit integer would hold that data and be smaller than storing it in a column that is 16, 32, or 64 bits. However, if you have a have a column that frequently takes values up to ~10 billion, you're going to need to use a 64-bit integer.

You can find out more about the Arrow data types by reading the project documentation, though for many people working with Arrow, you don't need to have a thorough understanding of these data types, as Arrow automatically converts between Arrow and R data types. If you don't have a specific reason to deviate from the default conversion, there's usually little benefit to doing so Switching from a 32 bit integer to an 8 bit integer won't lead to significant performance gains for most datasets, and optimizing for the best partitioning structure and storage format is much more important.

See Section A.2.2.3 and Section A.2.2.4 for more details about these conversions.

A.2.2.2 Casting

If you want to convert from one Arrow data type to another, you can use casting in dplyr pipelines. For example, if we create a tibble with a column of integers, and convert it to an Arrow table, the default conversion creates a 32-bit integer.

```
tibble::tibble(x = 1:3) |>
  arrow_table()
```

```
Table
3 rows x 1 columns
$x <int32>
```

However, we can use `cast()` to convert it to a different bitwidth, in this example, a 64-bit integer.

```
tibble::tibble(x = 1:3) |>
  arrow_table() |>
  mutate(y = cast(x, int64()))
```

```
Table (query)
x: int32
y: int64 (cast(x, {to_type=int64, allow_int_overflow=false, allow_time_truncate=false,
allow_time_overflow=false, allow_decimal_truncate=false, allow_float_truncate=false,
allow_invalid_utf8=false}))

See $.data for the source Arrow object
```

A.2.2.3 Translations from R to Arrow

Table A.1 is slightly modified from the Arrow project documentation, but clearly marks the mappings between R types and Arrow types.

Table A.1: R data types and their equivalent Arrow data types

Original R type	Arrow type after translation
logical	boolean
integer	int32
double (“numeric”)	float64 [1]
character	utf8 [2]
factor	dictionary
raw	uint8
Date	date32
POSIXct	timestamp
POSIXlt	struct
data.frame	struct
list [3]	list
bit64::integer64	int64
hms::hms	time32
difftime	duration

[1]: The two types `float64` and `double` are the same in Arrow C++; however, only `float64()` is used in arrow since the function `double()` already exists in base R.

[2]: If the character vector is exceptionally large—over 2GB of strings—it will be converted to a `large_utf8` Arrow type.

[3]: Only lists where all elements are the same type are able to be translated to Arrow list type (which is a “list of” some type). Arrow has a heterogeneous list type, but that is not exposed in the arrow R package.

A.2.2.4 Converting from Arrow to R

Table A.2 shows Arrow types and the R types they are translated to.

Table A.2: Arrow data types and their equivalent R data types

Original Arrow type	R type after translation
boolean	logical
int8	integer

Original Arrow type	R type after translation
int16	integer
int32	integer
int64	integer [1]
uint8	integer
uint16	integer
uint32	integer [1]
uint64	integer [1]
float16	- [2]
float32	double
float64	double
utf8	character
large_utf8	character
binary	arrow_binary [3]
large_binary	arrow_large_binary [3]
fixed_size_binary	arrow_fixed_size_binary [3]
date32	Date
date64	POSIXct
time32	hms::hms
time64	hms::hms
timestamp	POSIXct
duration	difftime
decimal	double
dictionary	factor [4]
list	arrow_list [5]
large_list	arrow_large_list [5]

Original Arrow type	R type after translation
fixed_size_list	arrow_fixed_size_list [5]
struct	data.frame
null	vctrs::vctrs_unspecified
map	arrow_list [5]
union	- [2]

[1]: These integer types may contain values that exceed the range of R's `integer` type (32 bit signed integer). When they do, `uint32` and `uint64` are converted to `double` ("numeric") and `int64` is converted to `bit64::integer64`. This conversion can be disabled (so that `int64` always yields a `bit64::integer64` vector) by setting `options(arrow.int64_downcast = FALSE)`.

[2]: Some Arrow data types do not currently have an R equivalent and will raise an error if cast to or mapped to via a schema.

[3]: `arrow*_binary` classes are implemented as lists of raw vectors.

[4]: Due to the limitation of R factors, Arrow `dictionary` values are coerced to string when translated to R if they are not already strings.

[5]: `arrow*_list` classes are implemented as subclasses of `vctrs_list_of` with a `ptype` attribute set to what an empty Array of the value type converts to.

A.3 Cloud

A.3.1 Network data transfer monitoring with nethogs

If you have a Linux machine and want to test the amount of data transferred to your machine while running similar examples to the ones found in this book, after installing nethogs, you can run the following code.

```
sudo nethogs -v 3
```

This runs the `nethogs` utility as a root user.