Biostat 203B Homework 2

Due Feb 7, 2025 @ 11:59PM

Amaan Jogia-Sattar, 206324648

Display machine information for reproducibility:

```r
sessionInfo()
```

```
R version 4.4.2 (2024-10-31)
Platform: aarch64-apple-darwin20
Running under: macOS Sequoia 15.3

Matrix products: default
BLAS:   /Library/Frameworks/R.framework/Versions/4.4-arm64/Resources/lib/libRblas.0.dylib
LAPACK: /Library/Frameworks/R.framework/Versions/4.4-arm64/Resources/lib/libRlapack.dylib;

locale:
[1] en_US.UTF-8/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8

time zone: America/Los_Angeles
tzcode source: internal

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

loaded via a namespace (and not attached):
 [1] compiler_4.4.2    fastmap_1.2.0     cli_3.6.3         tools_4.4.2
 [5] htmltools_0.5.8.1 rstudioapi_0.17.1 yaml_2.3.10       rmarkdown_2.28
 [9] knitr_1.48        jsonlite_1.8.9    xfun_0.48         digest_0.6.37
[13] rlang_1.1.4       evaluate_1.0.1
```

Load necessary libraries (you can add more as needed).

```r
library(arrow)
```

```


Attaching package: 'arrow'


The following object is masked from 'package:utils':

    timestamp
```

```r
library(data.table)
library(duckdb)
```

```
Loading required package: DBI
```

```r
library(memuse)
library(pryr)
```

```


Attaching package: 'pryr'


The following object is masked from 'package:data.table':

    address
```

```r
library(R.utils)
```

```
Loading required package: R.oo

Loading required package: R.methodsS3

R.methodsS3 v1.8.2 (2022-06-13 22:00:14 UTC) successfully loaded. See ?R.methodsS3 for help.

R.oo v1.27.0 (2024-11-01 18:00:02 UTC) successfully loaded. See ?R.oo for help.


Attaching package: 'R.oo'
```

```
The following object is masked from 'package:R.methodsS3':

    throw


The following objects are masked from 'package:methods':

    getClasses, getMethods


The following objects are masked from 'package:base':

    attach, detach, load, save


R.utils v2.12.3 (2023-11-18 01:00:02 UTC) successfully loaded. See ?R.utils for help.


Attaching package: 'R.utils'


The following object is masked from 'package:arrow':

    timestamp


The following object is masked from 'package:utils':

    timestamp


The following objects are masked from 'package:base':

    cat, commandArgs, getOption, isOpen, nullfile, parse, use, warnings
```

```r
library(tidyverse)
```

```
-- Attaching core tidyverse packages ------------------------ tidyverse 2.0.0 --
v dplyr     1.1.4     v readr     2.1.5
v forcats   1.0.0     v stringr   1.5.1
v ggplot2   3.5.1     v tibble    3.2.1
v lubridate 1.9.3     v tidyr     1.3.1
v purrr     1.0.2
```

```
-- Conflicts ------------------------------------------ tidyverse_conflicts() --
x dplyr::between()      masks data.table::between()
x purrr::compose()      masks pryr::compose()
x lubridate::duration() masks arrow::duration()
x tidyr::extract()      masks R.utils::extract()
x dplyr::filter()       masks stats::filter()
x dplyr::first()        masks data.table::first()
x lubridate::hour()     masks data.table::hour()
x lubridate::isoweek()  masks data.table::isoweek()
x dplyr::lag()          masks stats::lag()
x dplyr::last()         masks data.table::last()
x lubridate::mday()     masks data.table::mday()
x lubridate::minute()   masks data.table::minute()
x lubridate::month()    masks data.table::month()
x purrr::partial()      masks pryr::partial()
x lubridate::quarter()  masks data.table::quarter()
x lubridate::second()   masks data.table::second()
x purrr::transpose()    masks data.table::transpose()
x lubridate::wday()     masks data.table::wday()
x lubridate::week()     masks data.table::week()
x dplyr::where()        masks pryr::where()
x lubridate::yday()     masks data.table::yday()
x lubridate::year()     masks data.table::year()
i Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts to becon
```

Display memory information of your computer

```
memuse::Sys.meminfo()
```

```
Totalram:  36.000 GiB
Freeram:   14.443 GiB
```

In this exercise, we explore various tools for ingesting the MIMIC-IV data introduced in homework 1.

Display the contents of MIMIC `hosp` and `icu` data folders:

```
ls -l ~/mimic/hosp/
```

```
total 12306256
-rw-r--r--  1 amaanjsattar  staff    19928140 Jun 24  2024 admissions.csv.gz
```

```
-rw-r--r--  1 amaanjsattar  staff      427554 Apr 12  2024 d_hcpcs.csv.gz
-rw-r--r--  1 amaanjsattar  staff      876360 Apr 12  2024 d_icd_diagnoses.csv.gz
-rw-r--r--  1 amaanjsattar  staff      589186 Apr 12  2024 d_icd_procedures.csv.gz
-rw-r--r--  1 amaanjsattar  staff       13169 Oct  3 06:07 d_labitems.csv.gz
-rw-r--r--  1 amaanjsattar  staff    33564802 Oct  3 06:07 diagnoses_icd.csv.gz
-rw-r--r--  1 amaanjsattar  staff     9743908 Oct  3 06:07 drgcodes.csv.gz
-rw-r--r--  1 amaanjsattar  staff   811305629 Apr 12  2024 emar.csv.gz
-rw-r--r--  1 amaanjsattar  staff   748158322 Apr 12  2024 emar_detail.csv.gz
-rw-r--r--  1 amaanjsattar  staff     2162335 Apr 12  2024 hcpcsevents.csv.gz
-rw-r--r--  1 amaanjsattar  staff        2907 Jan 14 12:16 index.html
-rw-r--r--  1 amaanjsattar  staff  2592909134 Oct  3 06:08 labevents.csv.gz
-rw-r--r--  1 amaanjsattar  staff   117644075 Oct  3 06:08 microbiologyevents.csv.gz
-rw-r--r--  1 amaanjsattar  staff    44069351 Oct  3 06:08 omr.csv.gz
-rw-r--r--  1 amaanjsattar  staff     2835586 Apr 12  2024 patients.csv.gz
-rw-r--r--  1 amaanjsattar  staff   525708076 Apr 12  2024 pharmacy.csv.gz
-rw-r--r--  1 amaanjsattar  staff   666594177 Apr 12  2024 poe.csv.gz
-rw-r--r--  1 amaanjsattar  staff    55267894 Apr 12  2024 poe_detail.csv.gz
-rw-r--r--  1 amaanjsattar  staff   606298611 Apr 12  2024 prescriptions.csv.gz
-rw-r--r--  1 amaanjsattar  staff     7777324 Apr 12  2024 procedures_icd.csv.gz
-rw-r--r--  1 amaanjsattar  staff      127330 Apr 12  2024 provider.csv.gz
-rw-r--r--  1 amaanjsattar  staff     8569241 Apr 12  2024 services.csv.gz
-rw-r--r--  1 amaanjsattar  staff    46185771 Oct  3 06:08 transfers.csv.gz
```

```
ls -l ~/mimic/icu/
```

```
total 8506792
-rw-r--r--  1 amaanjsattar  staff       41566 Apr 12  2024 caregiver.csv.gz
-rw-r--r--  1 amaanjsattar  staff  3502392765 Apr 12  2024 chartevents.csv.gz
-rw-r--r--  1 amaanjsattar  staff       58741 Apr 12  2024 d_items.csv.gz
-rw-r--r--  1 amaanjsattar  staff    63481196 Apr 12  2024 datetimeevents.csv.gz
-rw-r--r--  1 amaanjsattar  staff     3342355 Oct  3 04:36 icustays.csv.gz
-rw-r--r--  1 amaanjsattar  staff        1336 Jan 14 12:16 index.html
-rw-r--r--  1 amaanjsattar  staff   311642048 Apr 12  2024 ingredientevents.csv.gz
-rw-r--r--  1 amaanjsattar  staff   401088206 Apr 12  2024 inputevents.csv.gz
-rw-r--r--  1 amaanjsattar  staff    49307639 Apr 12  2024 outputevents.csv.gz
-rw-r--r--  1 amaanjsattar  staff    24096834 Apr 12  2024 procedureevents.csv.gz
```

Q1. `read.csv` (base R) vs `read_csv` (tidyverse) vs `fread` (data.table)

Q1.1 Speed, memory, and data types

There are quite a few utilities in R for reading plain text data files. Let us test the speed of reading a moderate sized compressed csv file, `admissions.csv.gz`, by three functions: `read.csv` in base R, `read_csv` in tidyverse, and `fread` in the data.table package.

Which function is fastest? Is there difference in the (default) parsed data types? How much memory does each resultant dataframe or tibble use? (Hint: `system.time` measures run times; `pryr::object_size` measures memory usage; all these readers can take gz file as input without explicit decompression.)

Solution Comparing function speeds:

We will compute the runtimes and memory usage of each function. Note that the runtime utilized corresponds to the `elapsed` runtime as displayed by `system.time`.

```
# read.csv (base r)
# speed
runtime_base <- system.time(
  {admissions_base <- read.csv('~/mimic/hosp/admissions.csv.gz')}
  )[['elapsed']]
# memory usage
memuse_base <- format(object_size(admissions_base), units = 'auto')
```

```
# Measure runtime
runtime_tidy <- system.time(
  {admissions_tidy <- read.csv('~/mimic/hosp/admissions.csv.gz')}
)[['elapsed']]

# memory usage
memuse_tidy <- format(object_size(admissions_tidy), units = 'auto')
```

```
# fread (data.table package) speed
runtime_dt <- system.time(
  {admissions_dt <- fread('~/mimic/hosp/admissions.csv.gz')}
)[['elapsed']]
# memory usage
memuse_dt <- format(object_size(admissions_dt), units = 'auto')
```

```
# Check the structure of each dataframe

cat('\nBase R (read.csv):\n')
```

```
Base R (read.csv):
```

```r
str(admissions_base)
```

```
'data.frame':   546028 obs. of  16 variables:
 $ subject_id          : int  10000032 10000032 10000032 10000032 10000068 10000084 10000084
 $ hadm_id             : int  22595853 22841357 25742920 29079034 25022803 23052089 29888819
 $ admittime           : chr  "2180-05-06 22:23:00" "2180-06-26 18:27:00" "2180-08-05 23:44:0
 $ dischtime           : chr  "2180-05-07 17:15:00" "2180-06-27 18:49:00" "2180-08-07 17:50:0
 $ deathtime           : chr  "" "" "" "" ...
 $ admission_type      : chr  "URGENT" "EW EMER." "EW EMER." "EW EMER." ...
 $ admit_provider_id   : chr  "P49AFC" "P784FA" "P19UTS" "P060TX" ...
 $ admission_location  : chr  "TRANSFER FROM HOSPITAL" "EMERGENCY ROOM" "EMERGENCY ROOM" "EMI
 $ discharge_location  : chr  "HOME" "HOME" "HOSPICE" "HOME" ...
 $ insurance           : chr  "Medicaid" "Medicaid" "Medicaid" "Medicaid" ...
 $ language            : chr  "English" "English" "English" "English" ...
 $ marital_status      : chr  "WIDOWED" "WIDOWED" "WIDOWED" "WIDOWED" ...
 $ race                : chr  "WHITE" "WHITE" "WHITE" "WHITE" ...
 $ edregtime           : chr  "2180-05-06 19:17:00" "2180-06-26 15:54:00" "2180-08-05 20:58:0
 $ edouttime           : chr  "2180-05-06 23:30:00" "2180-06-26 21:31:00" "2180-08-06 01:44:0
 $ hospital_expire_flag: int  0 0 0 0 0 0 0 0 0 0 ...
```

```r
cat('\nTidyverse (read_csv):\n')
```

```

Tidyverse (read_csv):
```

```r
str(admissions_tidy)
```

```
'data.frame':   546028 obs. of  16 variables:
 $ subject_id          : int  10000032 10000032 10000032 10000032 10000068 10000084 10000084
 $ hadm_id             : int  22595853 22841357 25742920 29079034 25022803 23052089 29888819
 $ admittime           : chr  "2180-05-06 22:23:00" "2180-06-26 18:27:00" "2180-08-05 23:44:0
 $ dischtime           : chr  "2180-05-07 17:15:00" "2180-06-27 18:49:00" "2180-08-07 17:50:0
 $ deathtime           : chr  "" "" "" "" ...
 $ admission_type      : chr  "URGENT" "EW EMER." "EW EMER." "EW EMER." ...
 $ admit_provider_id   : chr  "P49AFC" "P784FA" "P19UTS" "P060TX" ...
 $ admission_location  : chr  "TRANSFER FROM HOSPITAL" "EMERGENCY ROOM" "EMERGENCY ROOM" "EMI
 $ discharge_location  : chr  "HOME" "HOME" "HOSPICE" "HOME" ...
 $ insurance           : chr  "Medicaid" "Medicaid" "Medicaid" "Medicaid" ...
 $ language            : chr  "English" "English" "English" "English" ...
 $ marital_status      : chr  "WIDOWED" "WIDOWED" "WIDOWED" "WIDOWED" ...
 $ race                : chr  "WHITE" "WHITE" "WHITE" "WHITE" ...
 $ edregtime           : chr  "2180-05-06 19:17:00" "2180-06-26 15:54:00" "2180-08-05 20:58:
 $ edouttime           : chr  "2180-05-06 23:30:00" "2180-06-26 21:31:00" "2180-08-06 01:44:
 $ hospital_expire_flag: int  0 0 0 0 0 0 0 0 0 0 ...
```

```
cat('\nData.table (fread):\n')
```

```
Data.table (fread):
```

```
str(admissions_dt)
```

```
Classes 'data.table' and 'data.frame':  546028 obs. of  16 variables:
 $ subject_id          : int  10000032 10000032 10000032 10000032 10000068 10000084 10000084
 $ hadm_id             : int  22595853 22841357 25742920 29079034 25022803 23052089 29888819
 $ admittime           : POSIXct, format: "2180-05-06 22:23:00" "2180-06-26 18:27:00" ...
 $ dischtime           : POSIXct, format: "2180-05-07 17:15:00" "2180-06-27 18:49:00" ...
 $ deathtime           : POSIXct, format: NA NA ...
 $ admission_type      : chr  "URGENT" "EW EMER." "EW EMER." "EW EMER." ...
 $ admit_provider_id   : chr  "P49AFC" "P784FA" "P19UTS" "P06OTX" ...
 $ admission_location  : chr  "TRANSFER FROM HOSPITAL" "EMERGENCY ROOM" "EMERGENCY ROOM" "EM
 $ discharge_location  : chr  "HOME" "HOME" "HOSPICE" "HOME" ...
 $ insurance           : chr  "Medicaid" "Medicaid" "Medicaid" "Medicaid" ...
 $ language            : chr  "English" "English" "English" "English" ...
 $ marital_status      : chr  "WIDOWED" "WIDOWED" "WIDOWED" "WIDOWED" ...
 $ race                : chr  "WHITE" "WHITE" "WHITE" "WHITE" ...
 $ edregtime           : POSIXct, format: "2180-05-06 19:17:00" "2180-06-26 15:54:00" ...
 $ edouttime           : POSIXct, format: "2180-05-06 23:30:00" "2180-06-26 21:31:00" ...
 $ hospital_expire_flag: int  0 0 0 0 0 0 0 0 0 0 ...
 - attr(*, ".internal.selfref")=<externalptr>
```

We observe that utilizing the `base r` function `read.csv` was the most time-intensive, taking approximately 9.89 seconds to complete. Using `read_csv` from the `tidyverse` was notably faster, with an execution time of approximately 5.533 seconds. The fastest function won by a considerable margin, being the `fread` function from the `data.table` package. This function took 0.446 seconds to execute.. In terms of memory usage, `fread` was the least memory-intensive, utilizing 63.47 MB. Comparatively, `read.csv` and `read_csv` utilize equal memory (200.10 MB = 200.10 MB. We also recognize crucial differences in parsed data types for each function. It appears that in all cases, string handling was identical and these columns were handled as `character (chr)` types. The most notable difference was in handling columns with date values. Both `read.csv` and `read_csv` stored these columns as `chr` type, while `fread` converted them to `POSIXct`. This is both memory-efficient and timesaving, as it bypasses any manual date conversion we would have to do when utilizing the other two functions. Ultimately, it appears that `fread` is the most memory-efficient and fastest function for reading plain text data files, while `read_csv` may be more ideal for `tidyverse-based workflows`. The base R function `read.csv` does not appear to be very efficient in speed or memory, nor does it appear to properly parse datetime data types.

Q1.2 User-supplied data types

Re-ingest `admissions.csv.gz` by indicating appropriate column data types in `read_csv`. Does the run time change? How much memory does the result tibble use? (Hint: `col_types` argument in `read_csv`.)

Solution Re-ingesting with User-Supplied Data Types: We will now specify column types within the `read_csv` function to see if this affects runtime and/or memory usage.

```r
admissions_coltypes <- cols(
  subject_id = col_integer(),
  hadm_id = col_integer(),
  admittime = col_datetime(),
  dischtime = col_datetime(),
  deathtime = col_datetime(),
  admission_type = col_character(),
  admit_provider_id = col_character(),
  admission_location = col_character(),
  discharge_location = col_character(),
  insurance = col_character(),
  language = col_character(),
  marital_status = col_character(),
  race = col_character(),
  edregtime = col_datetime(),
  edouttime = col_datetime(),
  hospital_expire_flag = col_integer()
)
# check runtime
runtime_tidy_spec <- system.time({
  admissions_tidy_spec <- read_csv('~/mimic/hosp/admissions.csv.gz',
                                   col_types = admissions_coltypes)}
  )[['elapsed']]
```

```r
# check memory usage
memuse_tidy_spec <- format(object_size(admissions_tidy_spec), units = 'auto')
```

We observe that after specifying the column types ourselves and reading in the data, the process is both faster and more memory-efficient. The runtime is 0.674 seconds, compared to 5.533 seconds when we didn't specify column types. Similarly, we observe that memory usage is 63.47 MB, compared to 200.10 MB when types were not pre-specified. Ultimately, by avoiding unnecessary character-types, this user specification is able to greatly enhance the speed and storage efficiency of our ingesting process.

Q2. Ingest big data files

Let us focus on a bigger file, `labevents.csv.gz`, which is about 130x bigger than `admissions.csv.gz`.

```bash
ls -l ~/mimic/hosp/labevents.csv.gz
```

```
-rw-r--r--  1 amaanjsattar  staff  2592909134 Oct  3 06:08 /Users/amaanjsattar/mimic/hosp/lal
```

Display the first 10 lines of this file.

```bash
zcat < ~/mimic/hosp/labevents.csv.gz | head -10
```

```
labevent_id,subject_id,hadm_id,specimen_id,itemid,order_provider_id,charttime,storetime,value
1,10000032,,2704548,50931,P69FQC,2180-03-23 11:51:00,2180-03-23 15:56:00,___,95,mg/dL,70,100
2,10000032,,36092842,51071,P69FQC,2180-03-23 11:51:00,2180-03-23 16:00:00,NEG,,,,,,ROUTINE,
3,10000032,,36092842,51074,P69FQC,2180-03-23 11:51:00,2180-03-23 16:00:00,NEG,,,,,,ROUTINE,
4,10000032,,36092842,51075,P69FQC,2180-03-23 11:51:00,2180-03-23 16:00:00,NEG,,,,,,ROUTINE,"I
5,10000032,,36092842,51079,P69FQC,2180-03-23 11:51:00,2180-03-23 16:00:00,NEG,,,,,,ROUTINE,
6,10000032,,36092842,51087,P69FQC,2180-03-23 11:51:00,,,,,,,,ROUTINE,RANDOM.
7,10000032,,36092842,51089,P69FQC,2180-03-23 11:51:00,2180-03-23 16:15:00,,,,,,,ROUTINE,PRESU
8,10000032,,36092842,51090,P69FQC,2180-03-23 11:51:00,2180-03-23 16:00:00,NEG,,,,,,ROUTINE,MI
9,10000032,,36092842,51092,P69FQC,2180-03-23 11:51:00,2180-03-23 16:00:00,NEG,,,,,,ROUTINE,"(
```

Q2.1 Ingest `labevents.csv.gz` by `read_csv`

Try to ingest `labevents.csv.gz` using `read_csv`. What happens? If it takes more than 3 minutes on your computer, then abort the program and report your findings.

Solution: Ingesting Large File When you try to ingest `labevents.csv.gz` using `read.csv`, the runtime is quite large due to the large file size, as well as the manner in which the function loads everything into memory at once. As such, if you only need a smaller subset of the data or you can process it in a more piecewise fashion, other data ingestion tools may be a much faster bet.

```r
system.time({
  labevents_tidy <- read_csv('~/mimic/hosp/labevents.csv.gz')
})
```

I allowed this cell to run for approximately ten minutes (in one particular instance), and the file was eventually read. However, this is a time- and memory-intensive operation to be run locally, so a workaround solution may be warranted. It appears that the large file size and default data parsing may both be contributing to the time-intensive nature of this procedure, so we may consider ingesting only a subset of the data and/or specifying data types beforehand.

Q2.2 Ingest selected columns of `labevents.csv.gz` by `read_csv`

Try to ingest only columns `subject_id`, `itemid`, `charttime`, and `valuenum` in `labevents.csv.gz` using `read_csv`. Does this solve the ingestion issue? (Hint: `col_select` argument in `read_csv`.)

Solution Ingesting Selected Columns:

```r
system.time({
  labevents_filtered <- read_csv('~/mimic/hosp/labevents.csv.gz',
                                 col_select = c(
                                   subject_id,
                                   itemid,
                                   charttime,
                                   valuenum
                                 ))
})
```

Selecting a smaller subset of columns did improve our ingestion issue, though it still took quite a bit of time to read in the data (just shy of three minutes, in one particular instance). This is still a marked improvement from the 10+ minutes elapsed while running the prior cell. Ultimately, only selecting the columns that will be required for subsequent analyses is an excellent strategy when dealing with large datasets, but if runtime issues persist, more memory-efficient ingestion processes should be considered. After all, the function is still loading all the data into memory, which is ideal for smaller csv files but may be too time and memory-intensive for particularly large files. As a result, partitioning files via `parquet` may be more optimal and a better use of time.

Q2.3 Ingest a subset of `labevents.csv.gz`

Our first strategy to handle this big data file is to make a subset of the `labevents` data. Read the MIMIC documentation for the content in data file `labevents.csv`.

In later exercises, we will only be interested in the following lab items: creatinine (50912), potassium (50971), sodium (50983), chloride (50902), bicarbonate (50882), hematocrit (51221), white blood cell count (51301), and glucose (50931) and the following columns: `subject_id`, `itemid`, `charttime`, `valuenum`. Write a Bash command to extract these columns and rows from `labevents.csv.gz` and save the result to a new file `labevents_filtered.csv.gz` in the current working directory. (Hint: Use `zcat <` to pipe the output of `labevents.csv.gz` to `awk` and then to `gzip` to compress the output. Do **not** put `labevents_filtered.csv.gz` in Git! To save render time, you can put `#| eval: false` at the beginning of this code chunk. TA will change it to `#| eval: true` before rendering your qmd file.)

Display the first 10 lines of the new file `labevents_filtered.csv.gz`. How many lines are in this new file, excluding the header? How long does it take `read_csv` to ingest `labevents_filtered.csv.gz`?

Solution: Subset Large File

```
zcat < ~/mimic/hosp/labevents.csv.gz |
awk -F, 'BEGIN {OFS=","; print "subject_id,itemid,charttime,valuenum"}
  NR > 1 && ($5 == 50912 || $5 == 50971 || $5 == 50983 || $5 == 50902 ||
             $5 == 50882 || $5 == 51221 || $5 == 51301 || $5 == 50931) {
    print $2, $5, $7, $10
}' | gzip > labevents_filtered.csv.gz
# Count the number of rows, excluding header
zcat labevents_filtered.csv.gz | tail -n +2 | wc -l
```

We observe that there are 32,679,896 lines, excluding our header. Now, we will display the first ten lines of our new file. Since we are including the header in this file, we can add an extra line in the `head` parameter:

```
zcat labevents_filtered.csv.gz | head -11
```

```
subject_id,itemid,charttime,valuenum
10000032,50931,2180-03-23 11:51:00,95
10000032,50882,2180-03-23 11:51:00,27
10000032,50902,2180-03-23 11:51:00,101
10000032,50912,2180-03-23 11:51:00,0.4
10000032,50971,2180-03-23 11:51:00,3.7
10000032,50983,2180-03-23 11:51:00,136
10000032,51221,2180-03-23 11:51:00,45.4
10000032,51301,2180-03-23 11:51:00,3
10000032,51221,2180-05-06 22:25:00,42.6
10000032,51301,2180-05-06 22:25:00,5
```

Lastly, we can measure how long it takes for `read_csv` to ingest `labevents_filtered.csv.gz`:

```r
runtime_filtered <- system.time({
  labevents_filtered <- read_csv('labevents_filtered.csv.gz')
})[['elapsed']]
```

```
Rows: 32679896 Columns: 4
-- Column specification --------------------------------------------------------
Delimiter: ","
dbl  (3): subject_id, itemid, valuenum
dttm (1): charttime

i Use `spec()` to retrieve the full column specification for this data.
i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

We observe that it takes 7.817 seconds to ingest `labevents_filtered.csv.gz` using `read_csv`. ### Q2.4 Ingest `labevents.csv` by Apache Arrow

Our second strategy is to use Apache Arrow for larger-than-memory data analytics. Unfortunately Arrow does not work with gz files directly. First decompress `labevents.csv.gz` to `labevents.csv` and put it in the current working directory (do not add it in git!). To save render time, put `#| eval: false` at the beginning of this code chunk. TA will change it to `#| eval: true` when rendering your qmd file.

Then use `arrow::open_dataset` to ingest `labevents.csv`, select columns, and filter `itemid` as in Q2.3. How long does the ingest+select+filter process take? Display the number of rows and the first 10 rows of the result tibble, and make sure they match those in Q2.3. (Hint: use `dplyr` verbs for selecting columns and filtering rows.)

Write a few sentences to explain what is Apache Arrow. Imagine you want to explain it to a layman in an elevator.

Solution Ingesting Using Apache Arrow: First, we will decompress `labevents.csv.gz` to `labevents.csv` and place it in our current working directory.

```
gunzip -c ~/mimic/hosp/labevents.csv.gz > labevents.csv
```

Next, we will use `arrow::open_dataset` to ingest `labevents.csv`, select columns, and filter `itemid` in accordance with the guidelines established in the previous exercise.

```
# Time the operation
# Use open_dataset to ingest file
# select columns
# filter itemid
# arrange by subject_id, charttime, itemid
runtime_arrow <- system.time({
  labevents_filtered <- arrow::open_dataset(
    'labevents.csv',
    format = 'csv') %>%
    select(subject_id, itemid, charttime, valuenum) %>%
    filter(itemid %in% c(
      50912,
      50971,
      50983,
      50902,
      50882,
      51221,
      51301,
      50931)) %>%
    collect() %>%
    arrange(subject_id, charttime, itemid)
})[['elapsed']]
print(nrow(labevents_filtered))
```

```
[1] 32679896
```

```
print(head(labevents_filtered, 10))
```

```
# A tibble: 10 x 4
   subject_id itemid charttime           valuenum
        <int>  <int> <dttm>                 <dbl>
 1   10000032  50882 2180-03-23 04:51:00     27
 2   10000032  50902 2180-03-23 04:51:00    101
 3   10000032  50912 2180-03-23 04:51:00      0.4
 4   10000032  50931 2180-03-23 04:51:00     95
 5   10000032  50971 2180-03-23 04:51:00      3.7
 6   10000032  50983 2180-03-23 04:51:00    136
 7   10000032  51221 2180-03-23 04:51:00     45.4
 8   10000032  51301 2180-03-23 04:51:00      3
 9   10000032  50882 2180-05-06 15:25:00     27
10   10000032  50902 2180-05-06 15:25:00    105
```

Note: Since the operation above has consistently taken < 1 minute to execute locally, I have decided to exclude the `eval = false` specification.

We observe that the operation took just 51.584 seconds, which is much faster than the operations we conducted that were reliant upon local memory. We also display the number of rows, as well as the first ten rows of the filtered dataset. There appear to be 32,679,896 rows. The first ten lines are displayed above.

Comparing this with the first ten rows we displayed in the previous exercise, we have obtained an identical result.

Here is an abridged explanation of Apache Arrow: Apache Arrow is a platform utilizing an in-memory, columnar format framework that reduces the memory and CPU toll associated with processing data. It is language-agnostic. It is optimized for handling large-scale data, with its columnar formatting minimizing waste of serialization and deserialization by standardizing the process outright. It also eases the process of data transfer between platforms, as well as moving data from one programming language to another. Technical details aside, it essentially makes data handling much more efficient and versatile.

Q2.5 Compress `labevents.csv` to Parquet format and ingest/select/filter

Re-write the csv file `labevents.csv` in the binary Parquet format (Hint: `arrow::write_dataset`.) How large is the Parquet file(s)? How long does the ingest+select+filter process of the Parquet file(s) take? Display the number of rows and the first 10 rows of the result tibble and make sure they match those in Q2.3. (Hint: use `dplyr` verbs for selecting columns and filtering rows.)

Write a few sentences to explain what is the Parquet format. Imagine you want to explain it to a layman in an elevator. **Solution**: Parquet Formatting

```
# Write the parquet file
  arrow::write_dataset(
  arrow::open_dataset('labevents.csv', format = 'csv'),
  path = 'part-0.parquet',
```

```r
  format = 'parquet'
)
# Measure runtime
# Ingestion, Selection, and Filtering step
runtime_parquet <- system.time({
  labevents_parquet <- arrow::open_dataset('part-0.parquet',
                        format = 'parquet') %>%
    select(subject_id, itemid, charttime, valuenum) %>%
    filter(itemid %in% c(
      50912,
      50971,
      50983,
      50902,
      50882,
      51221,
      51301,
      50931)) %>%
    arrange(subject_id, charttime, itemid) %>%
    collect()
})[['elapsed']]
print(nrow(labevents_parquet))
```

```
[1] 32679896
```

```r
print(head(labevents_parquet, 10))
```

```
# A tibble: 10 x 4
   subject_id itemid charttime           valuenum
        <int>  <int> <dttm>                 <dbl>
 1   10000032  50882 2180-03-23 04:51:00     27
 2   10000032  50902 2180-03-23 04:51:00    101
 3   10000032  50912 2180-03-23 04:51:00      0.4
 4   10000032  50931 2180-03-23 04:51:00     95
 5   10000032  50971 2180-03-23 04:51:00      3.7
 6   10000032  50983 2180-03-23 04:51:00    136
 7   10000032  51221 2180-03-23 04:51:00     45.4
 8   10000032  51301 2180-03-23 04:51:00      3
 9   10000032  50882 2180-05-06 15:25:00     27
10   10000032  50902 2180-05-06 15:25:00    105
```

Total runtime for the ingestion, selection, and filtering process was 6.213 seconds. Once again, we observe that there are 32,679,896 rows in the dataset. We compare the first ten lines with those in 2.4, verifying an identical result. We note that the first ten rows in 2.3 are not sorted in the same manner, and thus a mismatch is to be expected.

Here is an abridged explanation of the `parquet` file format: The `parquet` format is an open-source, `columnar` file format. In other words, it is column-oriented, as compared to the row-oriented structure of the `csv` file format. This means that every separate column is independently accessible, and data is intuitively organized within columns as opposed to rows. This makes the file format a fantastic candidate for column-wise parallel processing operations. Additionally, this columnar format can minimize file sizes when compared to standard `csv` files. If you are only working with a particular subset of columns, this file format may greatly speed up your workflow and reduce storage and memory tolls. For instance, if you wanted to perform operations on a column within a `csv` file format, you would need to read in the entire file. Conversely, you can access specific columns in isolation using the `parquet` format. Ultimately, this file format allows you to partition particularly sizable datasets into columns and perform operations independently, which maximizes efficiency for column-oriented data tasks and ultimately enables faster processing.

Q2.6 DuckDB

Ingest the Parquet file, convert it to a DuckDB table by `arrow::to_duckdb`, select columns, and filter rows as in Q2.5. How long does the ingest+convert+select+filter process take? Display the number of rows and the first 10 rows of the result tibble and make sure they match those in Q2.3. (Hint: use `dplyr` verbs for selecting columns and filtering rows.)

Write a few sentences to explain what is DuckDB. Imagine you want to explain it to a layman in an elevator. **Solution** DuckDB Formatting:

```r
runtime_duckdb <- system.time({
  labevents_duckdb <- arrow::to_duckdb(labevents_parquet) %>%
    select(subject_id, itemid, charttime, valuenum) %>%
    filter(itemid %in% c(
      50912,
      50971,
      50983,
      50902,
      50882,
      51221,
      51301,
      50931)) %>%
    arrange(subject_id, charttime, itemid) %>%
    collect()
})[['elapsed']]
print(nrow(labevents_duckdb))
```

```
[1] 32679896
```

```
print(head(labevents_duckdb, 10))
```

```
# A tibble: 10 x 4
   subject_id itemid charttime           valuenum
        <int>  <int> <dttm>                 <dbl>
 1   10000032  50882 2180-03-23 11:51:00     27  
 2   10000032  50902 2180-03-23 11:51:00    101  
 3   10000032  50912 2180-03-23 11:51:00      0.4
 4   10000032  50931 2180-03-23 11:51:00     95  
 5   10000032  50971 2180-03-23 11:51:00      3.7
 6   10000032  50983 2180-03-23 11:51:00    136  
 7   10000032  51221 2180-03-23 11:51:00     45.4
 8   10000032  51301 2180-03-23 11:51:00      3  
 9   10000032  50882 2180-05-06 22:25:00     27  
10   10000032  50902 2180-05-06 22:25:00    105  
```

We observe that the ingestion, conversion, selection, and filtering step took 2.181 seconds to execute. Once again, we observe that there are 32,679.896 rows in the dataset, and upon inspection, the ten-line preview of our resulting tibble is identical to previous results. Once again, we note that there was no sorting done in Q2.3, making mismatches in the `head` preview an expected result.

Here is an abridged explanation of DuckDB: DuckDB is an analytical database system that can efficiently process large data. It is unique in that it operates locally within your current application or notebook framework, making it simpler and easier to use. It utilizes a columnar file format, similar to `parquet`, and horizontally slices data into `row groups` within each column. This architecture makes it easy to query large datasets locally and with a single server. It integrates well with the tools that data professionals may use on `DataFrames`, like `Pandas` and `Polars`. Notably, it also allows you to run `SQL` queries and operations directly with no overhead requiremenmt. This interoperability makes it an extremely popular tool for data professionals with a variety of tech stacks. Ultimately, the tool bypasses the need for complicated setups and allows users access to powerful data processing tools at smaller scale than traditional distributed processing frameworks.
## Q3. Ingest and filter `chartevents.csv.gz`

`chartevents.csv.gz` contains all the charted data available for a patient. During their ICU stay, the primary repository of a patient's information is their electronic chart. The `itemid` variable indicates a single measurement type in the database. The `value` variable is the value measured for `itemid`. The first 10 lines of `chartevents.csv.gz` are

```
zcat < ~/mimic/icu/chartevents.csv.gz | head -10
```

```
subject_id,hadm_id,stay_id,caregiver_id,charttime,storetime,itemid,value,valuenum,valueuom,wa
10000032,29079034,39553978,18704,2180-07-23 12:36:00,2180-07-23 14:45:00,226512,39.4,39.4,kg
10000032,29079034,39553978,18704,2180-07-23 12:36:00,2180-07-23 14:45:00,226707,60,60,Inch,0
10000032,29079034,39553978,18704,2180-07-23 12:36:00,2180-07-23 14:45:00,226730,152,152,cm,0
10000032,29079034,39553978,18704,2180-07-23 14:00:00,2180-07-23 14:18:00,220048,SR (Sinus Rhy
10000032,29079034,39553978,18704,2180-07-23 14:00:00,2180-07-23 14:18:00,224642,Oral,,,0
10000032,29079034,39553978,18704,2180-07-23 14:00:00,2180-07-23 14:18:00,224650,None,,,0
10000032,29079034,39553978,18704,2180-07-23 14:00:00,2180-07-23 14:20:00,223761,98.7,98.7,°F
10000032,29079034,39553978,18704,2180-07-23 14:11:00,2180-07-23 14:17:00,220179,84,84,mmHg,0
10000032,29079034,39553978,18704,2180-07-23 14:11:00,2180-07-23 14:17:00,220180,48,48,mmHg,0
```

How many rows? 433 million.

```
zcat < ~/mimic/icu/chartevents.csv.gz | tail -n +2 | wc -l
```

`d_items.csv.gz` is the dictionary for the `itemid` in `chartevents.csv.gz`.

```
zcat < ~/mimic/icu/d_items.csv.gz | head -10
```

```
itemid,label,abbreviation,linksto,category,unitname,param_type,lownormalvalue,highnormalvalu
220001,Problem List,Problem List,chartevents,General,,Text,,
220003,ICU Admission date,ICU Admission date,datetimeevents,ADT,,Date and time,,
220045,Heart Rate,HR,chartevents,Routine Vital Signs,bpm,Numeric,,
220046,Heart rate Alarm - High,HR Alarm - High,chartevents,Alarms,bpm,Numeric,,
220047,Heart Rate Alarm - Low,HR Alarm - Low,chartevents,Alarms,bpm,Numeric,,
220048,Heart Rhythm,Heart Rhythm,chartevents,Routine Vital Signs,,Text,,
220050,Arterial Blood Pressure systolic,ABPs,chartevents,Routine Vital Signs,mmHg,Numeric,90
220051,Arterial Blood Pressure diastolic,ABPd,chartevents,Routine Vital Signs,mmHg,Numeric,6
220052,Arterial Blood Pressure mean,ABPm,chartevents,Routine Vital Signs,mmHg,Numeric,,
```

In later exercises, we are interested in the vitals for ICU patients: heart rate (220045), mean non-invasive blood pressure (220181), systolic non-invasive blood pressure (220179), body temperature in Fahrenheit (223761), and respiratory rate (220210). Retrieve a subset of `chartevents.csv.gz` only containing these items, using the favorite method you learnt in Q2.

Document the steps and show code. Display the number of rows and the first 10 rows of the result tibble. **Solution**: Ingest and Filter ChartEvents

We will utilize `Apache Arrow` to efficiently write this dataset as a `parquet` file. Next, we will select the four key columns we specified in previous exercise. In this step, we will also be filtering for only the relevant vital measurements as specified above. In detail, here is the process that occurs below: - We utilize `arrow::write_dataset` to create our `parquet` file format from our `csv.gz` file. We specify the file path and will utilize this `parquet` file folder in subsequent steps. - We create `chartevents_filtered`, first referencing our `chartevents.parquet` file path we established in the previous segment of code. We filter this dataset using `dplyr` methods, the first of which involves using `select` to specify columns of interest. These are the same four columns we have been using across these exercises. We proceed to `filter` the rows for the particular vital measurements specified above. We then sort using `arrange`, by three of our four columns, such that the first ten lines of our resulting dataset will be identical to those in previous exercises. Lastly, we aggregate all of this information into our new, filtered dataset using `collect`. - We obtain the number of rows in our `chartevents_filtered` dataset using `nrow`. We display the first ten rows of the dataset using `head`.

```
# Convert the compressed chartevents file to parquet format
arrow::write_dataset(
  open_dataset('~/mimic/icu/chartevents.csv.gz',
               format = 'csv'),
  path = 'chartevents.parquet',
  format = 'parquet'
)
```

```
# Ingestion, Selection, and Filtering Step
  chartevents_filtered <- arrow::open_dataset('chartevents.parquet',
                        format = 'parquet') %>%
    select(subject_id, itemid, charttime, valuenum) %>%
    filter(itemid %in% c(
      220045,
      220181,
      220179,
      223761,
      220210)) %>%
    arrange(subject_id, charttime, itemid) %>%
    collect()
# Obtain number of rows
print(nrow(chartevents_filtered))
```

```
[1] 30195426
```

```
# Display the first ten rows
print(head(chartevents_filtered, 10))
```

```
# A tibble: 10 x 4
   subject_id itemid charttime           valuenum
        <int>  <int> <dttm>                 <dbl>
 1   10000032 223761 2180-07-23 07:00:00     98.7
 2   10000032 220179 2180-07-23 07:11:00     84
 3   10000032 220181 2180-07-23 07:11:00     56
 4   10000032 220045 2180-07-23 07:12:00     91
 5   10000032 220210 2180-07-23 07:12:00     24
 6   10000032 220045 2180-07-23 07:30:00     93
 7   10000032 220179 2180-07-23 07:30:00     95
 8   10000032 220181 2180-07-23 07:30:00     67
 9   10000032 220210 2180-07-23 07:30:00     21
10   10000032 220045 2180-07-23 08:00:00     94
```

We observe that there are 30,195,426 rows in this filtered dataset. The first ten lines are displayed above.