Biostat 203B Homework 2

Due Feb 7, 2025 @ 11:59PM

AUTHOR
Julie Lee 806409381

Display machine information for reproducibility:

```
sessionInfo()
```

```
R version 4.4.2 (2024-10-31)
Platform: x86_64-pc-linux-gnu
Running under: Ubuntu 24.04.1 LTS

Matrix products: default
BLAS:   /usr/lib/x86_64-linux-gnu/blas/libblas.so.3.12.0
LAPACK: /usr/lib/x86_64-linux-gnu/lapack/liblapack.so.3.12.0

locale:
 [1] LC_CTYPE=C.UTF-8       LC_NUMERIC=C           LC_TIME=C.UTF-8
 [4] LC_COLLATE=C.UTF-8     LC_MONETARY=C.UTF-8    LC_MESSAGES=C.UTF-8
 [7] LC_PAPER=C.UTF-8       LC_NAME=C              LC_ADDRESS=C
[10] LC_TELEPHONE=C         LC_MEASUREMENT=C.UTF-8 LC_IDENTIFICATION=C

time zone: Etc/UTC
tzcode source: system (glibc)

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

loaded via a namespace (and not attached):
 [1] htmlwidgets_1.6.4 compiler_4.4.2    fastmap_1.2.0     cli_3.6.3
 [5] tools_4.4.2       htmltools_0.5.8.1 rstudioapi_0.17.1 yaml_2.3.10
 [9] rmarkdown_2.29    knitr_1.49        jsonlite_1.8.9    xfun_0.50
[13] digest_0.6.37     rlang_1.1.4       evaluate_1.0.3
```

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

```
library(arrow)
```

```
Attaching package: 'arrow'

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

    timestamp
```

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

```
Loading required package: DBI
```

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

```

Attaching package: 'pryr'

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

    address
```

```
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
```

```
library(tidyverse)
```

```
── Attaching core tidyverse packages ──────────────────────── tidyverse 2.0.0 ──
✔ dplyr     1.1.4     ✔ readr     2.1.5
✔ forcats   1.0.0     ✔ stringr   1.5.1
✔ ggplot2   3.5.1     ✔ tibble    3.2.1
✔ lubridate 1.9.4     ✔ tidyr     1.3.1
✔ purrr     1.0.2
── Conflicts ────────────────────────────────────────── tidyverse_conflicts() ──
✖ dplyr::between()      masks data.table::between()
✖ purrr::compose()      masks pryr::compose()
✖ lubridate::duration() masks arrow::duration()
✖ tidyr::extract()      masks R.utils::extract()
✖ dplyr::filter()       masks stats::filter()
✖ dplyr::first()        masks data.table::first()
✖ lubridate::hour()     masks data.table::hour()
✖ lubridate::isoweek()  masks data.table::isoweek()
✖ dplyr::lag()          masks stats::lag()
✖ dplyr::last()         masks data.table::last()
✖ lubridate::mday()     masks data.table::mday()
✖ lubridate::minute()   masks data.table::minute()
✖ lubridate::month()    masks data.table::month()
✖ purrr::partial()      masks pryr::partial()
✖ lubridate::quarter()  masks data.table::quarter()
✖ lubridate::second()   masks data.table::second()
✖ purrr::transpose()    masks data.table::transpose()
✖ lubridate::wday()     masks data.table::wday()
✖ lubridate::week()     masks data.table::week()
✖ dplyr::where()        masks pryr::where()
✖ lubridate::yday()     masks data.table::yday()
✖ lubridate::year()     masks data.table::year()
ℹ Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts to
become errors
```

```
library(microbenchmark)
```

Display memory information of your computer

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

```
Totalram:  62.792 GiB
Freeram:   43.417 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 24124692
-rw-r--r-- 1 root    root       19928140 Jun 24  2024 admissions.csv.gz
-rw-r--r-- 1 root    root         427554 Apr 13  2024 d_hcpcs.csv.gz
-rw-r--r-- 1 root    root         876360 Apr 13  2024 d_icd_diagnoses.csv.gz
-rw-r--r-- 1 root    root         589186 Apr 13  2024 d_icd_procedures.csv.gz
-rw-r--r-- 1 root    root          13169 Oct  3 13:07 d_labitems.csv.gz
-rw-r--r-- 1 root    root       33564802 Oct  3 13:07 diagnoses_icd.csv.gz
-rw-r--r-- 1 root    root        9743908 Oct  3 13:07 drgcodes.csv.gz
-rw-r--r-- 1 root    root      811305629 Apr 13  2024 emar.csv.gz
-rw-r--r-- 1 root    root      748158322 Apr 13  2024 emar_detail.csv.gz
-rw-r--r-- 1 root    root        2162335 Apr 13  2024 hcpcsevents.csv.gz
-rw-r--r-- 1 root    root           2907 Dec 29 02:04 index.html
-rw-r--r-- 1 huazhou huazhou 18402851720 Jan 29 05:16 labevents.csv
-rw-r--r-- 1 root    root     2592909134 Oct  3 13:08 labevents.csv.gz
-rw-r--r-- 1 root    root      117644075 Oct  3 13:08 microbiologyevents.csv.gz
-rw-r--r-- 1 root    root       44069351 Oct  3 13:08 omr.csv.gz
-rw-r--r-- 1 root    root        2835586 Apr 13  2024 patients.csv.gz
-rw-r--r-- 1 root    root      525708076 Apr 13  2024 pharmacy.csv.gz
-rw-r--r-- 1 root    root      666594177 Apr 13  2024 poe.csv.gz
-rw-r--r-- 1 root    root       55267894 Apr 13  2024 poe_detail.csv.gz
-rw-r--r-- 1 root    root      606298611 Apr 13  2024 prescriptions.csv.gz
-rw-r--r-- 1 root    root        7777324 Apr 13  2024 procedures_icd.csv.gz
-rw-r--r-- 1 root    root         127330 Apr 13  2024 provider.csv.gz
-rw-r--r-- 1 root    root        8569241 Apr 13  2024 services.csv.gz
-rw-r--r-- 1 root    root       46185771 Oct  3 13:08 transfers.csv.gz
```

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

```
total 45206348
-rw-r--r-- 1 root    root          41566 Apr 13  2024 caregiver.csv.gz
-rw-r--r-- 1 huazhou huazhou 41935806083 Jan 29 05:38 chartevents.csv
-rw-r--r-- 1 root    root     3502392765 Apr 13  2024 chartevents.csv.gz
-rw-r--r-- 1 root    root          58741 Apr 13  2024 d_items.csv.gz
-rw-r--r-- 1 root    root       63481196 Apr 13  2024 datetimeevents.csv.gz
-rw-r--r-- 1 root    root        3342355 Oct  3 11:36 icustays.csv.gz
-rw-r--r-- 1 root    root           1336 Dec 29 02:04 index.html
-rw-r--r-- 1 root    root      311642048 Apr 13  2024 ingredientevents.csv.gz
-rw-r--r-- 1 root    root      401088206 Apr 13  2024 inputevents.csv.gz
-rw-r--r-- 1 root    root       49307639 Apr 13  2024 outputevents.csv.gz
-rw-r--r-- 1 root    root       24096834 Apr 13  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.

```
file_path <- "~/mimic/hosp/admissions.csv.gz"

# Function 1: Using read.csv (Base R)
time_base <- system.time(
  df_base <- read.csv(file_path, stringsAsFactors = TRUE)
)
size_base <- object_size(df_base)

# Function 2: Using read_csv (tidyverse)
time_tidy <- system.time(
  df_tidy <- read_csv(file_path)
)
```

```
Rows: 546028 Columns: 16
── Column specification ────────────────────────────────────────────────────────
Delimiter: ","
chr  (8): admission_type, admit_provider_id, admission_location, discharge_l...
dbl  (3): subject_id, hadm_id, hospital_expire_flag
dttm (5): admittime, dischtime, deathtime, edregtime, edouttime

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.
```

```
size_tidy <- object_size(df_tidy)

# Function 3: Using fread (data.table)
time_dt <- system.time(
  df_dt <- fread(file_path)
)
size_dt <- object_size(df_dt)

results <- data.frame(
  Method = c("Base R (read.csv)", "Tidyverse (read_csv)",
             "Data.table (fread)"),
  UserTime = c(time_base["user.self"], time_tidy["user.self"],
               time_dt["user.self"]),
  SystemTime = c(time_base["sys.self"], time_tidy["sys.self"],
                 time_dt["sys.self"]),
  ElapsedTime = c(time_base["elapsed"], time_tidy["elapsed"],
```

```
                  time_dt["elapsed"]),
  MemoryUsage = c(size_base, size_tidy, size_dt)
)

print(results)
```

```
               Method UserTime SystemTime ElapsedTime MemoryUsage
1    Base R (read.csv)   25.671      0.278      25.950   186.29 MB
2 Tidyverse (read_csv)    1.997      0.327       1.158    70.02 MB
3   Data.table (fread)    2.089      0.126       0.803    63.47 MB
```

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.)

(1.1 Solution) The fastest function in R for reading a plain text data file, specifically the compressed CSV file admissions.csv.gz, is the fread function from the data.table package. This conclusion is supported by its low values for SystemTime, ElapsedTime, and Usertime, highlighting its superior speed compared to other methods. Furthermore, fread is the most memory-efficient option, with the lowest Memory Usage. Based on a trial run, the memory usage for each method is as follows: Base R (read.csv) uses 200.10 MB, Tidyverse (read_csv) uses 70.02 MB, and data.table (fread) uses only 63.47 MB. These results clearly demonstrate that fread is the most efficient method for handling moderate to large CSV files in R.

We investigate the difference in the (default) parsed data types:

The read.csv (Base R) by default, converts character columns into factors. We notice that the variables subject_id, hadm_id are set to integers. All of the other variables are set to factors.

```
cat("Base R (read.csv):\n")
```

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

```
str(df_base)
```

```
'data.frame':   546028 obs. of  16 variables:
 $ subject_id          : int  10000032 10000032 10000032 10000032 10000068 10000084
10000084 10000108 10000117 10000117 ...
 $ hadm_id             : int  22595853 22841357 25742920 29079034 25022803 23052089
29888819 27250926 22927623 27988844 ...
 $ admittime           : Factor w/ 534919 levels "2105-10-04 17:26:00",..: 435477 436427
437166 436920 301301 306085 306741 325043 445952 457945 ...
 $ dischtime           : Factor w/ 528871 levels "2105-10-12 11:11:00",..: 430229 431159
431914 431687 297620 302449 303014 321091 440561 452461 ...
 $ deathtime           : Factor w/ 11789 levels "","2110-01-25 09:40:00",..: 1 1 1 1 1 1
1 1 1 1 ...
 $ admission_type      : Factor w/ 9 levels "AMBULATORY OBSERVATION",..: 9 6 6 6 5 6 5 5
5 7 ...
 $ admit_provider_id   : Factor w/ 2046 levels "","P00230","P004G6",..: 1008 1589 373 118
813 870 728 832 973 248 ...
```

```
 $ admission_location  : Factor w/ 12 levels "","AMBULATORY SURGERY TRANSFER",..: 10 4 4
4 4 12 8 4 4 12 ...
 $ discharge_location  : Factor w/ 14 levels "","ACUTE HOSPITAL",..: 8 8 10 8 1 9 1 1 1 9
...
 $ insurance           : Factor w/ 6 levels "","Medicaid",..: 2 2 2 2 1 3 3 1 2 2 ...
 $ language            : Factor w/ 26 levels "","American Sign Language",..: 8 8 8 8 8 8
8 8 8 8 ...
 $ marital_status      : Factor w/ 5 levels "","DIVORCED",..: 5 5 5 5 4 3 3 4 2 2 ...
 $ race                : Factor w/ 33 levels "AMERICAN INDIAN/ALASKA NATIVE",..: 29 29 29
29 29 29 29 29 29 29 ...
 $ edregtime           : Factor w/ 372693 levels "","2106-02-06 15:47:00",..: 301539
302202 302719 302538 208011 211326 211789 224593 308885 317302 ...
 $ edouttime           : Factor w/ 372756 levels "","2106-02-07 09:31:00",..: 301592
302258 302774 302601 208035 211346 211823 224632 308945 317374 ...
 $ hospital_expire_flag: int  0 0 0 0 0 0 0 0 0 0 ...
```

The read_csv Tidyverse function does not convert character columns to factors by default. We notice that the variables subject_id, hadm_id, and hospital_expire_flag are set to numbers. The variables admittime, dischtime, deathtime, edregtime, and edouttime are set to POSIXct, and all other variables are set to characters.

```
cat("\nTidyverse (read_csv):\n")
```

```
Tidyverse (read_csv):
```

```
str(df_tidy)
```

```
spc_tbl_ [546,028 × 16] (S3: spec_tbl_df/tbl_df/tbl/data.frame)
 $ subject_id          : num [1:546028] 1e+07 1e+07 1e+07 1e+07 1e+07 ...
 $ hadm_id             : num [1:546028] 22595853 22841357 25742920 29079034 25022803 ...
 $ admittime           : POSIXct[1:546028], format: "2180-05-06 22:23:00" "2180-06-26
18:27:00" ...
 $ dischtime           : POSIXct[1:546028], format: "2180-05-07 17:15:00" "2180-06-27
18:49:00" ...
 $ deathtime           : POSIXct[1:546028], format: NA NA ...
 $ admission_type      : chr [1:546028] "URGENT" "EW EMER." "EW EMER." "EW EMER." ...
 $ admit_provider_id   : chr [1:546028] "P49AFC" "P784FA" "P19UTS" "P06OTX" ...
 $ admission_location  : chr [1:546028] "TRANSFER FROM HOSPITAL" "EMERGENCY ROOM"
"EMERGENCY ROOM" "EMERGENCY ROOM" ...
 $ discharge_location  : chr [1:546028] "HOME" "HOME" "HOSPICE" "HOME" ...
 $ insurance           : chr [1:546028] "Medicaid" "Medicaid" "Medicaid" "Medicaid" ...
 $ language            : chr [1:546028] "English" "English" "English" "English" ...
 $ marital_status      : chr [1:546028] "WIDOWED" "WIDOWED" "WIDOWED" "WIDOWED" ...
 $ race                : chr [1:546028] "WHITE" "WHITE" "WHITE" "WHITE" ...
 $ edregtime           : POSIXct[1:546028], format: "2180-05-06 19:17:00" "2180-06-26
15:54:00" ...
 $ edouttime           : POSIXct[1:546028], format: "2180-05-06 23:30:00" "2180-06-26
21:31:00" ...
 $ hospital_expire_flag: num [1:546028] 0 0 0 0 0 0 0 0 0 0 ...
 - attr(*, "spec")=
```

```
  .. cols(
  ..   subject_id = col_double(),
  ..   hadm_id = col_double(),
  ..   admittime = col_datetime(format = ""),
  ..   dischtime = col_datetime(format = ""),
  ..   deathtime = col_datetime(format = ""),
  ..   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(format = ""),
  ..   edouttime = col_datetime(format = ""),
  ..   hospital_expire_flag = col_double()
  .. )
 - attr(*, "problems")=<externalptr>
```

The fread (Data.Table) function automatically detects the appropriate column types (numeric, character, etc). We notice that the variables subject_id, hadm_id, hospital_expire_flag are set to integers, the variables admittime, dischtime, deathtime, edregtime, and edouttime are set to POSIXct, and the rest of the variables are set to characters.

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

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

```
str(df_dt)
```

```
Classes 'data.table' and 'data.frame':  546028 obs. of  16 variables:
 $ subject_id          : int  10000032 10000032 10000032 10000032 10000068 10000084
10000084 10000108 10000117 10000117 ...
 $ hadm_id             : int  22595853 22841357 25742920 29079034 25022803 23052089
29888819 27250926 22927623 27988844 ...
 $ 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"
"EMERGENCY ROOM" ...
 $ 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>
```

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`.)

```
file_path <- "~/mimic/hosp/admissions.csv.gz"

time_tidy_default <- system.time(
  df_tidy_default <- read_csv(file_path)
)
```

```
Rows: 546028 Columns: 16
── Column specification ────────────────────────────────────────────────────────
Delimiter: ","
chr  (8): admission_type, admit_provider_id, admission_location, discharge_l...
dbl  (3): subject_id, hadm_id, hospital_expire_flag
dttm (5): admittime, dischtime, deathtime, edregtime, edouttime

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.
```

```
size_tidy_default <- object_size(df_tidy_default)

col_types <- cols(
  subject_id = col_double(),
  hadm_id = col_double(),
  admittime = col_datetime(format = ""),
  dischtime = col_datetime(format = ""),
  deathtime = col_datetime(format = ""),
  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(format = ""),
  edouttime = col_datetime(format = ""),
  hospital_expire_flag = col_double()
)

time_tidy_optimized <- system.time(
  df_tidy_optimized <- read_csv(file_path, col_types = col_types)
)
size_tidy_optimized <- object_size(df_tidy_optimized)
```

```
results <- data.frame(
  Method = c("read_csv (Default)",
             "read_csv (Appropriate Column Data Types)"),
  ElapsedTime = c(time_tidy_default["elapsed"],
                  time_tidy_optimized["elapsed"]),
  MemoryUsage = c(size_tidy_default, size_tidy_optimized)
)

print(results)
```

```
                                  Method ElapsedTime MemoryUsage
1                     read_csv (Default)       1.243    70.02 MB
2 read_csv (Appropriate Column Data Types)       1.036    70.02 MB
```

(1.2) Solution When we re-ingest admissions.csv.gz with specified column types, we observe that the memory usage remains unchanged at 70.02 MB, consistent with the default ingestion. However, specifying column types leads to a noticeable improvement in performance, reducing the elapsed time from 1.371 seconds to 1.010 seconds. This demonstrates that while explicitly defining column types does not impact memory consumption, it optimizes parsing speed, making the data loading process more efficient. Thus, for large datasets, pre-defining column types can be beneficial in enhancing performance.

Q2. Ingest big data files

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

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

```
-rw-r--r-- 1 root root 2592909134 Oct  3 13:08 /home/jlee0344/mimic/hosp/labevents.csv.gz
```

Display the first 10 lines of this file.

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

```
labevent_id,subject_id,hadm_id,specimen_id,itemid,order_provider_id,charttime,storetime,v
alue,valuenum,valueuom,ref_range_lower,ref_range_upper,flag,priority,comments
1,10000032,,2704548,50931,P69FQC,2180-03-23 11:51:00,2180-03-23
15:56:00,___,95,mg/dL,70,100,,ROUTINE,"IF FASTING, 70-100 NORMAL, >125 PROVISIONAL
DIABETES."
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,"BENZODIAZEPINE IMMUNOASSAY SCREEN DOES NOT DETECT SOME
DRUGS,;INCLUDING LORAZEPAM, CLONAZEPAM, AND FLUNITRAZEPAM."
```

```
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,PRESUMPTIVELY POSITIVE.
8,10000032,,36092842,51090,P69FQC,2180-03-23 11:51:00,2180-03-23
16:00:00,NEG,,,,,,ROUTINE,METHADONE ASSAY DETECTS ONLY METHADONE (NOT OTHER
OPIATES/OPIOIDS).
9,10000032,,36092842,51092,P69FQC,2180-03-23 11:51:00,2180-03-23
16:00:00,NEG,,,,,,ROUTINE,"OPIATE IMMUNOASSAY SCREEN DOES NOT DETECT SYNTHETIC
OPIOIDS;SUCH AS METHADONE, OXYCODONE, FENTANYL, BUPRENORPHINE, TRAMADOL,;NALOXONE,
MEPERIDINE.  SEE ONLINE LAB MANUAL FOR DETAILS."
```

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.

Using the read_csv function from the readr package.

```
file_path_labevents <- "~/mimic/hosp/labevents.csv.gz"
cat("\n=== Ingesting 'labevents.csv.gz' using read_csv() ===\n")

time_full <- system.time({
  df_full <- read_csv(file_path_labevents)
})

size_full <- object_size(df_full)

cat("\n=== Ingestion Performance Metrics ===\n")
cat("Elapsed Time (seconds):", time_full["elapsed"], "\n")
cat("Memory Usage (MB):", size_full / (1024^2), "\n")
print(head(df_full))
```

(2.1) Solution

First attempt using R Server from Laptop: Attempting to ingest ‘labevents.csv.gz’ using read_csv() (from the readr package) in R crashed due to my Mac’s 8GB RAM, which is insufficient for handling the 18.4GB decompressed file. Since R loads data entirely into memory, it requires approximately twice the file size (~36.8GB RAM), causing a memory overflow. To handle large files efficiently, alternatives include using chunked processing with read_csv(). This crash is expected because R primarily operates in-memory (RAM) rather than efficiently streaming large datasets. When using read_csv(), R attempts to load the entire dataset into RAM instead of processing it in chunks. If the available memory is insufficient, the system quickly runs out of resources, leading to an R session crash. This limitation makes handling large datasets challenging without proper memory management techniques.

The second attempt using the provided R Server successfully ingested ‘labevents.csv.gz’ using read_csv() from the readr package without crashing. The function was able to display the first six rows of the dataset. However, the ingestion process was slow, taking approximately 197.825 seconds (3.29 minutes) to fully load the file. Additionally,

the memory usage was 19,337.91 MB (≈19 GB). This suggests that while read_csv() can handle the ingestion, it may not be the most efficient option for large datasets due to high memory consumption and long processing times.

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`.)

```
file_path_labevents <- "~/mimic/hosp/labevents.csv.gz"
time_selected <- system.time(
  df_selected <- read_csv(
    file_path_labevents,
    col_select = c("subject_id", "itemid", "charttime", "valuenum")
  ) %>%
  arrange(subject_id, charttime, itemid)
)
```

```
Rows: 158374764 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.
```

```
size_selected <- object_size(df_selected)
cat("Elapsed Time (seconds):", time_selected["elapsed"], "\n")
```

```
Elapsed Time (seconds): 184.915
```

```
cat("Memory Usage (bytes):", size_selected, "\n")
```

```
Memory Usage (bytes): 5067997752
```

```
head(df_selected)
```

```
# A tibble: 6 × 4
  subject_id itemid charttime           valuenum
       <dbl>  <dbl> <dttm>                 <dbl>
1   10000032  50853 2180-03-23 11:51:00     15
2   10000032  50861 2180-03-23 11:51:00    102
3   10000032  50862 2180-03-23 11:51:00      3.3
4   10000032  50863 2180-03-23 11:51:00    109
5   10000032  50864 2180-03-23 11:51:00      8
6   10000032  50868 2180-03-23 11:51:00     12
```

(2.2) Solution By selecting only four columns (subject_id, itemid, charttime, and valuenum) when ingesting labevents.csv.gz using read_csv(), the process took 159.211 seconds (2.65 minutes) and used 5.07 GB (5,067,997,752 bytes) of memory. Comparing this to ingesting the entire file without column selection—where the memory usage was 19,337.91 MB (≈19 GB) and the ingestion time was 197.825 seconds (3.25 minutes)—we observe a notable improvement in both memory efficiency and processing time.This suggests that while read_csv() is capable of handling large dataset ingestion, it is not the most efficient option due to its high memory consumption and relatively slow processing speed. Selecting only the necessary columns significantly reduces memory usage and speeds up ingestion, making it a more practical approach for handling large datasets efficiently.

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`?

```
zcat ~/mimic/hosp/labevents.csv.gz |
  awk -F, 'BEGIN {OFS=","; print "subject_id,itemid,charttime,valuenum"}
  NR==1 {next}
  $5 ~ /^[0-9]+$/ && ($5 == 50912 || $5 == 50971 || $5 == 50983 ||
  $5 == 50902 || $5 == 50882 || $5 == 51221 || $5 == 51301 || $5 == 50931) &&
  $2 != "" && $7 != "" && $10 != "" {
      print $2,$5,$7,$10
  }'  | gzip > labevents_filtered.csv.gz
```

The first 10 lines (excluding the header) of the new file 'labevents_filtered.csv.gz' is presented below:

```
#!/bin/bash
file="labevents_filtered.csv.gz"
echo "First 10 lines of the file (including the header):"
header=$(zcat "$file" | head -n 1)
(
  echo "$header"
  zcat "$file" | tail -n +2 | sort -t, -k1,1n -k3,3 -k2,2n | head -n 10
) 2>/dev/null
```

```
First 10 lines of the file (including the header):
subject_id,itemid,charttime,valuenum
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
10000032,50931,2180-03-23 11:51:00,95
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
10000032,50882,2180-05-06 22:25:00,27
10000032,50902,2180-05-06 22:25:00,105
```

```
#Number of Lines in New File excluding the header
file="labevents_filtered.csv.gz"
line_count=$(zcat "$file" | tail -n +2 | wc -l)
echo "Number of lines in the file (excluding the header): $line_count"
```

```
Number of lines in the file (excluding the header): 32651024
```

```
file="labevents_filtered.csv.gz"
echo -e "\nMeasuring read time in R..."
Rscript -e "
start_time <- Sys.time()
df <- read.csv(gzfile('$file'))
end_time <- Sys.time()
time_taken <- round(difftime(end_time, start_time, units='secs'), 4)
print(paste('Time taken to ingest the file using read.csv():', time_taken,
            'seconds'))
"
```

```
Measuring read time in R...
[1] "Time taken to ingest the file using read.csv(): 51.0069 seconds"
```

(2.3) Solution After manually checking the specific columns that correspond to “subject_id”, “itemid”, “charttime”, and “valuenum” and filtering out rows that only have lab items creatinine (50912), potassium (50971), sodium (50983), chloride (50902), bicarbonate (50882), hematocrit (51221), white blood cell count (51301), and glucose (50931), we created the labevents_filtered.csv.gz. The number of lines in the new file (labevents_filtered.csv.gz) excluding the header is 32651024 lines. The time it takes to ingest the `labevents_filtered.csv.gz` file using the function read_csv is 51.0504 seconds.

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.

```
#l eval : false
gzip -d -c labevents_filtered.csv.gz > labevents.csv
```

```
start_time <- Sys.time()
file_path <- "labevents.csv"
dataset <- open_dataset(file_path, format = "csv")

# Filter out specific values in item_id
filtered_data_apache <- dataset %>%
  select(subject_id, itemid, charttime, valuenum) %>%
  arrange(subject_id, charttime, itemid) %>%
  filter(itemid %in% c(50912, 50971, 50983, 50902, 50882, 51221, 51301,
                       50931)) %>%
  collect()

end_time <- Sys.time()

time_taken <- end_time - start_time
cat("Time taken for ingest, select, and filter: ", round(time_taken, 4),
    "seconds\n")
```

```
Time taken for ingest, select, and filter:  7.9102 seconds
```

```
cat("Number of rows in the result: ", nrow(filtered_data_apache), "\n")
```

```
Number of rows in the result:  32651024
```

The first 10 rows of the resulting tibble are:

```
print(head(filtered_data_apache, 10))
```

```
# A tibble: 10 × 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
```

(2.4) Solution

The time that it takes to ingest, select, and filter out for specific item_id (creatinine (50912), potassium (50971), sodium (50983), chloride (50902), bicarbonate (50882), hematocrit (51221), white blood cell count (51301), and glucose (50931)) to create the resulting tibble takes 8.6242 seconds. The number of rows of the result tibble is 32651024 rows.

Apache Arrow is a technology that helps software developers create powerful applications to work with large amounts of data quickly and efficiently. It provides a special way to organize and store data in a format that works well across different programming languages. This format is designed to handle both simple data (like lists or tables) and more complex structures, making it easier to analyze data on modern hardware, such as CPUs and GPUs. The way Apache Arrow structures data (utilizes a standardized columnar format) allows for much faster processing and analysis because it takes advantage of how modern hardware works. This means tasks like running queries or performing analytics can be done more quickly and with better use of the computer's resources.

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.

```
csv_file <- "labevents.csv"
parquet_dir <- "labevents_parquet"

start_time <- Sys.time()
write_dataset(
  read_csv_arrow(csv_file), parquet_dir, format = "parquet"
)
end_time_conversion <- Sys.time()

cat(
  "Time taken to convert CSV to Parquet:",
  round(difftime(end_time_conversion, start_time, units = "secs"), 4),
  "seconds\n"
)
```

```
Time taken to convert CSV to Parquet: 3.7919 seconds
```

```
parquet_size <- sum(
  file.info(list.files(parquet_dir, recursive = TRUE, full.names = TRUE))$size
) / (1024^2)  # Convert to MB
cat("Total size of Parquet file(s):", round(parquet_size, 2), "MB\n")
```

```
Total size of Parquet file(s): 127.15 MB
```

```
start_time <- Sys.time()
dataset <- open_dataset(parquet_dir, format = "parquet")

filtered_data <- dataset %>%
  select(subject_id, itemid, charttime, valuenum) %>%
  filter(itemid %in% c(
    50912, 50971, 50983, 50902, 50882, 51221, 51301, 50931
  )) %>%
  arrange(subject_id, charttime, itemid) %>%
  collect()
end_time <- Sys.time()

cat(
  "Time taken to ingest, select, and filter Parquet file(s):",
  round(difftime(end_time, start_time, units = "secs"), 4),
  "seconds\n"
)
```

```
Time taken to ingest, select, and filter Parquet file(s): 4.1882 seconds
```

```
cat("Number of rows in the filtered data:", nrow(filtered_data), "\n")
```

```
Number of rows in the filtered data: 32651024
```

```
print(head(filtered_data, 10))  # Explicitly print the first 10 rows
```

```
# A tibble: 10 × 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
```

(2.5 Solution) After running the code above, we were able to obtain the following information: The time it takes to convert the CSV file to Parquet is 16.6681 seconds. The total size of Parquet files is 127.15 MB. The time it takes to

ingest, select, and filter Parquet files is 0.3625 seconds. The number of rows in the filtered dataset is 32651024 rows. The Parquet format is an open-source file format used to store data in columns instead of rows, making it especially useful for big data and analytics. Storing data in columns helps compress the files better, so they take up less space compared to formats like CSV. It also makes data queries faster because only the relevant columns are read, reducing the amount of data scanned and speeding up processing. Side Note: The resulting tibble matches the one in question 2.4.

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.

```
parquet_file <- "labevents_parquet"
start_time <- Sys.time()
arrow_data <- open_dataset(parquet_file, format = "parquet")
duckdb_conn <- dbConnect(duckdb::duckdb())

invisible(to_duckdb(
  .data = arrow_data,
  con = duckdb_conn,
  table_name = "labevents",
  auto_disconnect = FALSE
))

result <- tbl(duckdb_conn, "labevents") %>%
  select(subject_id, itemid, charttime, valuenum) %>%
  filter(itemid %in% c(50912, 50971, 50983, 50902,
                       50882, 51221, 51301, 50931)) %>%
  arrange(subject_id, charttime, itemid,) %>%
  collect()

end_time <- Sys.time()

elapsed_time <- end_time - start_time
cat("Ingest, convert, select, and filter time:",
    round(elapsed_time, 4), "seconds\n")
```

```
Ingest, convert, select, and filter time: 5.1738 seconds
```

```
cat("Number of rows in the result:", nrow(result), "\n")
```

```
Number of rows in the result: 32651024
```

```
print(head(result, 10))
```

```
# A tibble: 10 × 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
```

```
dbDisconnect(duckdb_conn)
```

(Solution 2.6) After running the code above, we were able to obtain the following information regarding the Parquet file that was converted to a DuckDB tab: The process to ingest, select, and filter is 4.0037 seconds. The number of rows in the filtered dataset is 32651024 rows. DuckDB is an open-source, column-oriented database designed for fast and efficient data analysis. It handles complex queries on large datasets, such as combining tables with hundreds of columns and billions of rows, all within an embedded setup. DuckDB is efficient for tasks such as analyzing log files, personal data on edge devices, and preparing data for machine learning, especially when privacy is important. It can read files directly (like Parquet or CSV), uses SQL to quickly find insights, and is lightweight and fast. Side Note: The tibble matches the ones from questions 2.4 and 2.5.

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,valueuo
m,warning
10000032,29079034,39553978,18704,2180-07-23 12:36:00,2180-07-23
14:45:00,226512,39.4,39.4,kg,0
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
Rhythm),,,0
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,0
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 millions.

```
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,highnormalv
alue
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,90,140
220051,Arterial Blood Pressure diastolic,ABPd,chartevents,Routine Vital
Signs,mmHg,Numeric,60,90
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.

```
zcat ~/mimic/icu/chartevents.csv.gz |
awk -F, '
BEGIN {
    OFS = ",";
    print "subject_id", "itemid", "charttime", "valuenum"
}
NR == 1 {next}
$7 ~ /^[0-9]+$/ &&
($7 == 220045 || $7 == 220181 || $7 == 220179 ||
 $7 == 223761 || $7 == 220210) {
    print $1, $7, $5, $9
}' | gzip > chartevents_filtered.csv.gz
```

Displaying the number of rows (excluding the header row)

```
zcat chartevents_filtered.csv.gz | tail -n +2 | wc -l
```

```
30195426
```

Displaying the First 10 rows of the result tibble:

```
#!/bin/bash
file="chartevents_filtered.csv.gz"
echo "First 10 lines of the file (including the header):"
(
  zcat "$file" | head -n 1
  zcat "$file" | tail -n +2 | sort -t, -k1,1n -k3,3 -k2,2n  | head -n 10
) 2>/dev/null
```

```
First 10 lines of the file (including the header):
subject_id,itemid,charttime,valuenum
10000032,223761,2180-07-23 14:00:00,98.7
10000032,220179,2180-07-23 14:11:00,84
10000032,220181,2180-07-23 14:11:00,56
10000032,220045,2180-07-23 14:12:00,91
10000032,220210,2180-07-23 14:12:00,24
10000032,220045,2180-07-23 14:30:00,93
10000032,220179,2180-07-23 14:30:00,95
10000032,220181,2180-07-23 14:30:00,67
10000032,220210,2180-07-23 14:30:00,21
10000032,220045,2180-07-23 15:00:00,94
```

(3.1) Solution We selected the same four columns from the previous problem: subject_id, itemid, charttime, and valuenum, which are essential for analyzing ICU patient data. To focus on specific vitals, we filtered rows corresponding to key itemid values: heart rate (220045), mean non-invasive blood pressure (220181), systolic non-invasive blood pressure (220179), body temperature in Fahrenheit (223761), and respiratory rate (220210). To handle the large file located at ~/mimic/icu/chartevents.csv.gz, we used the first strategy (from question 2.2) to create a filtered and compressed subset of the data. This involved using the bash command to first decompress the file with zcat, stream its content into awk for filtering and extracting the relevant columns, and finally compress the output with gzip, resulting in a new file named chartevents_filtered.csv.gz in the current working directory. This method efficiently processed the large dataset, reducing its size while retaining only the relevant rows and columns, producing a streamlined and compressed dataset ready for further analysis.