Biostat 203B Homework 2

Due Feb 7, 2025 @ 11:59PM

Sakshi Oza , 606542442

Display machine information for reproducibility:

```
sessionInfo()
```

```
R version 4.4.1 (2024-06-14)
Platform: aarch64-apple-darwin20
Running under: macOS Sonoma 14.4

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;  l

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_4.4.1    fastmap_1.2.0     cli_3.6.3         tools_4.4.1
 [5] htmltools_0.5.8.1 rstudioapi_0.17.0 yaml_2.3.10       rmarkdown_2.29
 [9] knitr_1.48        jsonlite_1.8.9    xfun_0.49         digest_0.6.37
[13] 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 --
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 becor
```

Display memory information of your computer

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

```
Totalram:    8.000 GiB
Freeram:   299.000 MiB
```

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-rw-r--@ 1 sakshihiteshoza  staff    19928140 Jun 24  2024 admissions.csv.gz
```

```
-rw-rw-r--@ 1 sakshihiteshoza  staff      427554 Apr 12  2024 d_hcpcs.csv.gz
-rw-rw-r--@ 1 sakshihiteshoza  staff      876360 Apr 12  2024 d_icd_diagnoses.csv.gz
-rw-rw-r--@ 1 sakshihiteshoza  staff      589186 Apr 12  2024 d_icd_procedures.csv.gz
-rw-rw-r--@ 1 sakshihiteshoza  staff       13169 Oct  3 06:07 d_labitems.csv.gz
-rw-rw-r--@ 1 sakshihiteshoza  staff    33564802 Oct  3 06:07 diagnoses_icd.csv.gz
-rw-rw-r--@ 1 sakshihiteshoza  staff     9743908 Oct  3 06:07 drgcodes.csv.gz
-rw-rw-r--@ 1 sakshihiteshoza  staff   811305629 Apr 12  2024 emar.csv.gz
-rw-rw-r--@ 1 sakshihiteshoza  staff   748158322 Apr 12  2024 emar_detail.csv.gz
-rw-rw-r--@ 1 sakshihiteshoza  staff     2162335 Apr 12  2024 hcpcsevents.csv.gz
-rw-rw-r--@ 1 sakshihiteshoza  staff        2907 Dec 28 18:04 index.html
-rw-r--r--@ 1 sakshihiteshoza  staff  2592909134 Jan 24 15:14 labevents.csv.gz
-rw-rw-r--@ 1 sakshihiteshoza  staff   117644075 Oct  3 06:08 microbiologyevents.csv.gz
-rw-rw-r--@ 1 sakshihiteshoza  staff    44069351 Oct  3 06:08 omr.csv.gz
-rw-rw-r--@ 1 sakshihiteshoza  staff     2835586 Apr 12  2024 patients.csv.gz
-rw-rw-r--@ 1 sakshihiteshoza  staff   525708076 Apr 12  2024 pharmacy.csv.gz
-rw-rw-r--@ 1 sakshihiteshoza  staff   666594177 Apr 12  2024 poe.csv.gz
-rw-rw-r--@ 1 sakshihiteshoza  staff    55267894 Apr 12  2024 poe_detail.csv.gz
-rw-rw-r--@ 1 sakshihiteshoza  staff   606298611 Apr 12  2024 prescriptions.csv.gz
-rw-rw-r--@ 1 sakshihiteshoza  staff     7777324 Apr 12  2024 procedures_icd.csv.gz
-rw-rw-r--@ 1 sakshihiteshoza  staff      127330 Apr 12  2024 provider.csv.gz
-rw-rw-r--@ 1 sakshihiteshoza  staff     8569241 Apr 12  2024 services.csv.gz
-rw-rw-r--@ 1 sakshihiteshoza  staff    46185771 Oct  3 06:08 transfers.csv.gz
```

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

```
total 8506792
-rw-rw-r--@ 1 sakshihiteshoza  staff       41566 Apr 12  2024 caregiver.csv.gz
-rw-rw-r--@ 1 sakshihiteshoza  staff  3502392765 Apr 12  2024 chartevents.csv.gz
-rw-r--r--  1 sakshihiteshoza  staff           0 Feb  7 14:53 chartevents_filtered.csv.gz
-rw-rw-r--@ 1 sakshihiteshoza  staff       58741 Apr 12  2024 d_items.csv.gz
-rw-rw-r--@ 1 sakshihiteshoza  staff    63481196 Apr 12  2024 datetimeevents.csv.gz
-rw-rw-r--@ 1 sakshihiteshoza  staff     3342355 Oct  3 04:36 icustays.csv.gz
-rw-rw-r--@ 1 sakshihiteshoza  staff        1336 Dec 28 18:04 index.html
-rw-rw-r--@ 1 sakshihiteshoza  staff   311642048 Apr 12  2024 ingredientevents.csv.gz
-rw-rw-r--@ 1 sakshihiteshoza  staff   401088206 Apr 12  2024 inputevents.csv.gz
-rw-rw-r--@ 1 sakshihiteshoza  staff    49307639 Apr 12  2024 outputevents.csv.gz
-rw-rw-r--@ 1 sakshihiteshoza  staff    24096834 Apr 12  2024 procedureevents.csv.gz
```

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

Solution 1.1: 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.)

```
# Measure time and memory for read.csv (base R)
time_read_csv_base <- system.time({
  data_base <- read.csv("~/mimic/hosp/admissions.csv.gz")
})
memory_read_csv_base <- object_size(data_base)

# Measure time and memory for read_csv (tidyverse)
time_read_csv_tidyverse <- system.time({
  data_tidyverse <- read_csv("~/mimic/hosp/admissions.csv.gz",
                             show_col_types = FALSE)

})
memory_read_csv_tidyverse <- object_size(data_tidyverse)

# Measure time and memory for fread (data.table)
time_read_fread <- system.time({
  data_fread <- fread("~/mimic/hosp/admissions.csv.gz")
})
memory_read_fread <- object_size(data_fread)
```

```
# Output the results for runtime
cat("Runtime (in seconds):\n")
```

```
Runtime (in seconds):
```

```
cat("Base R read.csv: ", time_read_csv_base["elapsed"], " seconds\n")
```

```
Base R read.csv:  5.556  seconds
```

```r
cat("Tidyverse read_csv: ", time_read_csv_tidyverse["elapsed"], " seconds\n")
```

```
Tidyverse read_csv:  0.616  seconds
```

```r
cat("Data.table fread: ", time_read_fread["elapsed"], " seconds\n\n")
```

```
Data.table fread:  0.319  seconds
```

```r
# Output the memory usage
cat("Memory usage (in bytes):\n")
```

```
Memory usage (in bytes):
```

```r
cat("Base R read.csv: ", memory_read_csv_base, " bytes\n")
```

```
Base R read.csv:  200098832  bytes
```

```r
cat("Tidyverse read_csv: ", memory_read_csv_tidyverse, " bytes\n")
```

```
Tidyverse read_csv:  70022592  bytes
```

```r
cat("Data.table fread: ", memory_read_fread, " bytes\n")
```

```
Data.table fread:  63465008  bytes
```

1. Fastest Function:

- `fread` from the `data.table` package is the fastest, with a runtime of approx **0.286 seconds**.

2. Difference in Parsed Data Types:

- `read.csv` (base R) converts character columns into factors by default.
- `read_csv` (tidyverse) and `fread` (data.table) do **not** convert characters to factors by default, and they infer column types automatically.

3. Memory Usage:

- `fread` (data.table) is the most memory-efficient, using approx **63,465,008 bytes** (~63 MB).
- `read_csv` (tidyverse) is the second most efficient, using approx **70,022,592 bytes** (~70 MB).
- `read.csv` (base R) uses the most memory at approx **200,098,832 bytes** (~200 MB)

Solution 1.2: 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`.)

```
# Measure time and memory usage for read_csv without col_types
time_read_csv_without_coltypes <- system.time({
  data_tidyverse <- read_csv("~/mimic/hosp/admissions.csv.gz",
                             # Silent the Column specification
                             show_col_types = FALSE)
})
memory_read_csv_without_coltypes <- object_size(data_tidyverse)
```

```
# Specifying column types based on data structure
col_types <- cols(
  subject_id = col_integer(),
  hadm_id = col_integer(),
  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_logical()
)

# Measure time and memory usage for read_csv with col_types
time_read_csv_coltypes <- system.time({
  data_tidyverse_coltypes <- read_csv("~/mimic/hosp/admissions.csv.gz",
                                      col_types = col_types)
})
memory_read_csv_coltypes <- object_size(data_tidyverse_coltypes)
```

```
# Compare the results
cat("Runtime with col_types specified: ", time_read_csv_coltypes["elapsed"],
    " seconds\n")
```

```
Runtime with col_types specified:  0.475  seconds
```

```
cat("Memory usage with col_types specified: ", memory_read_csv_coltypes,
    " bytes\n")
```

```
Memory usage with col_types specified:  63470560  bytes
```

```
cat("Runtime without col_types: ", time_read_csv_without_coltypes["elapsed"],
    " seconds\n")
```

```
Runtime without col_types:  0.533  seconds
```

```
cat("Memory usage without col_types: ", memory_read_csv_without_coltypes,
    " bytes\n")
```

```
Memory usage without col_types:  70022592  bytes
```

- Runtime change: Yes, the runtime with the `col_types` specified is **0.568 seconds**, which is a bit slower than the runtime without `col_types` (**0.549 seconds**). This indicates that specifying column types decreases the loading speed by a minor difference.
- Memory usage: The memory usage is **63,470,560 bytes** with the `col_types` specified and **70,022,592 bytes** with `read_csv`. The difference in memory usage is huge (**6552032 bytes**), meaning that specifying column types will optimize the memory instead of using default types where double takes more memory in case we have integer types.

Q2. Ingest big data files

Solution 2.1 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.

```
library(readr)
system.time({
  labevents_data <- read_csv("~/mimic/hosp/labevents.csv.gz")
})
```

- It took more than 3 minutes.

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

```
-rw-r--r--@ 1 sakshihiteshoza  staff  2592909134 Jan 24 15:14 /Users/sakshihiteshoza/mimic/h
```

```
cat("Memory of labevents.csv.gz: ", 2592909134*9.5367e-7, " MiB\n")
```

```
Memory of labevents.csv.gz:  2472.78  MiB
```

`read_csv` cannot ingest `labevents.csv.gz` because the available ram is ~284 MiB where as the compressed file `labevents.csv.gz` requires about 2473 MiB.

Solution 2.2 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`.)

```
labevents_data <- read_csv("~/mimic/hosp/labevents.csv.gz",
                      col_select = c(subject_id, itemid, charttime, valuenum))

# Check the data
head(labevents_data)
```

- It took more than 3 minutes

```
zcat < ~/mimic/hosp/labevents.csv.gz | wc -l
```

```
 158374765
```

```
subject_id_one_row = 4 # integer is 4 bytes
itemid_one_row = 4 # integer is 4 bytes
valuenum_one_row = 8 # double is 8 bytes
charttime_one_row = 4 # timestamp is 4 bytes
one_row_total = subject_id_one_row + itemid_one_row + valuenum_one_row
one_row_total = one_row_total + charttime_one_row

# Total rows from above cell's output
total_rows = 158374765
cat("Memory required with colselect labevents.csv.gz: ",
    one_row_total * total_rows * 9.5367e-7, " MiB\n")
```

```
Memory required with colselect labevents.csv.gz:  3020.745  MiB
```

`read_csv` cannot ingest `labevents.csv.gz` with selected columns too because the available ram is ~284 MiB where as memory required requires about 3021 MiB.

Solution 2.3 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 = ","}

{
    if ($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
```

```
time_taken <- system.time({
  labevents_filtered <- read_csv("./labevents_filtered.csv.gz",
    col_names = FALSE,
    col_types = cols(
      subject_id = col_integer(),
      itemid = col_integer(),
      charttime = col_datetime(),
      valuenum = col_double()
    ))
  colnames(labevents_filtered) <- c("subject_id", "itemid",
                                    "charttime", "valuenum")
})
```

```
Warning: The following named parsers don't match the column names: subject_id,
itemid, charttime, valuenum
```

```
# Printing the time taken
print(time_taken)
```

```
   user  system elapsed
 16.062   3.158   8.170
```

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

```
# A tibble: 10 x 4
   subject_id itemid charttime           valuenum
        <dbl>  <dbl> <dttm>                 <dbl>
 1   10000032  50931 2180-03-23 11:51:00     95
 2   10000032  50882 2180-03-23 11:51:00     27
 3   10000032  50902 2180-03-23 11:51:00    101
 4   10000032  50912 2180-03-23 11:51:00      0.4
 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  51221 2180-05-06 22:25:00     42.6
10   10000032  51301 2180-05-06 22:25:00      5
```

It took about 16 seconds overall to load the file, out of which 9.741 seconds were spent running your R code (user time), and about 3 seconds were used by the system for tasks like file I/O. The remaining time could be due to overhead like waiting for resources, parallel processing inefficiencies, or other factors.

```
print(nrow(labevents_filtered))
```

```
[1] 32679896
```

```
print(ncol(labevents_filtered))
```

```
[1] 4
```

Solution 2.4 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.

```
system("gunzip -k ./labevents_filtered.csv.gz")
```

```
library(arrow)
library(dplyr)

# Measure time taken for ingestion, selection, and filtering using Apache Arrow
time_taken_arrow <- system.time({
  # Load the dataset using Apache Arrow
```

```
  dataset <- open_dataset("./labevents_filtered.csv",format = "csv")

  # Select the relevant columns and filter based on the 'itemid' values
  # The itemid values correspond to the following lab items:
  # Creatinine (50912), Potassium (50971), Sodium (50983), Chloride (50902),
  # Bicarbonate (50882), Hematocrit (51221), White blood cell count (51301),
  # and Glucose (50931)
  result <- dataset %>%
    select(subject_id, itemid, charttime, valuenum) %>%
    filter(itemid %in% c(50912, 50971, 50983, 50902, 50882,
                         51221, 51301, 50931))


  # Collect the result as a tibble
  result_tibble <- result %>%
    collect()
})

# Print the time taken for the entire process
print(time_taken_arrow)

# Display the number of rows and the first 10 rows of the result tibble
print(nrow(result_tibble))
print(head(result_tibble, 10))
```

The process of ingesting, selecting columns, and filtering rows using Apache Arrow took approximately 3.9 seconds in total. This includes: - User time: 3.8 seconds for executing code. - System time: 0.67 seconds for system-level operations.

Apache Arrow is a framework that enhances data processing by providing an efficient in-memory columnar format for large datasets. It allows data to be processed quickly and shared across different programming languages without needing to copy data. This makes it highly suitable for modern data analytics pipelines, especially when working with datasets that are too large to fit into memory.

Solution 2.5 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.

```
library(arrow)
library(dplyr)

# Write the CSV file to Parquet format
arrow::write_dataset(
  open_dataset("./labevents_filtered.csv", format = "csv"),
  path = "./labevents_filtered.parque",
  format = "parquet"
)

# Check the size of the Parquet file
parquet_file_info <- file.info("./labevents_filtered.parquet")
print(parquet_file_info$size)
```

```
[1] 96
```

```
# Measure the time taken for ingestion, selection, and filtering using
# Parquet format
time_taken_parquet <- system.time({
  # Load the Parquet dataset
  dataset_parquet <- open_dataset("./labevents_filtered.parquet",
                                  format = "parquet")

  # Select relevant columns and filter based on itemid values
  result_parquet <- dataset_parquet %>%
    select(subject_id, itemid, charttime, valuenum) %>%
    filter(itemid %in% c(50912, 50971, 50983, 50902,
                         50882, 51221, 51301, 50931)) %>%
    collect()
})

# Print the time taken for the process
print(time_taken_parquet)
```

```
   user  system elapsed
  0.943   0.551   0.533
```

```r
# Display the number of rows and the first 10 rows of the result tibble
print(nrow(result_parquet))
```

```
[1] 32679896
```

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

```
# A tibble: 10 x 4
   subject_id itemid charttime           valuenum
        <int>  <int> <dttm>                 <dbl>
 1   10015785  51221 2150-12-07 22:40:00     38.6
 2   10015785  51301 2150-12-07 22:40:00      7
 3   10015785  50882 2150-12-07 22:40:00     29
 4   10015785  50902 2150-12-07 22:40:00    101
 5   10015785  50912 2150-12-07 22:40:00      0.6
 6   10015785  50931 2150-12-07 22:40:00    106
 7   10015785  50971 2150-12-07 22:40:00      3.4
 8   10015785  50983 2150-12-07 22:40:00    139
 9   10015834  51221 2156-12-04 23:28:00     41.2
10   10015834  51301 2156-12-04 23:28:00      6.6
```

The process of ingesting, selecting, and filtering the Parquet file took 0.611 seconds, with 32,679,896 rows in the filtered data. This shows Parquet's efficiency compared to CSV for handling large datasets.

Parquet is a special file format designed to store large amounts of data efficiently. Unlike regular text files like CSV, Parquet compresses the data, making it smaller and faster to work with. It also organizes data in a way that allows programs to quickly find and read only the parts they need, rather than loading the entire file. This saves time and space, especially when dealing with big datasets.

Solution 2.6 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.

```r
library(arrow)
library(duckdb)
library(dplyr)

# Measure time taken for the entire process
time_taken_duckdb <- system.time({
  # Ingest Parquet file and convert it to a DuckDB table
  duckdb_table <- to_duckdb(
    open_dataset("./labevents_filtered.parquet", format = "parquet")
  )

  # Select relevant columns and filter based on itemid values
  result_duckdb <- duckdb_table %>%
    select(subject_id, itemid, charttime, valuenum) %>%
    filter(itemid %in% c(50912, 50971, 50983, 50902,
                         50882, 51221, 51301, 50931)) %>%
    collect()
})

# Print the time taken for the entire process
print(time_taken_duckdb)
```

```
   user  system elapsed
  1.588   0.587   1.021
```

```r
# Display the number of rows and the first 10 rows of the result tibble
print(nrow(result_duckdb))
```

```
[1] 32679896
```

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

```
# A tibble: 10 x 4
   subject_id itemid charttime           valuenum
        <dbl>  <dbl> <dttm>                 <dbl>
 1   10033740  50983 2183-09-10 14:23:00    139
 2   10033760  51221 2145-04-22 00:45:00     33.9
 3   10033760  51301 2145-04-22 00:45:00      7.2
 4   10033760  50882 2145-04-22 00:45:00     26
 5   10033760  50902 2145-04-22 00:45:00    106
```

```
 6   10033760  50912 2145-04-22 00:45:00      0.6
 7   10033760  50931 2145-04-22 00:45:00    121
 8   10033760  50971 2145-04-22 00:45:00      3.4
 9   10033760  50983 2145-04-22 00:45:00    142
10   10033760  50882 2147-02-12 11:42:00     25
```

The process of ingesting the Parquet file, converting it to a DuckDB table, and selecting and filtering data took 1.11 seconds, with 32,679,896 rows in the filtered result.

DuckDB is like a mini, super-fast database that runs right on your computer without needing to set up any complex systems. It's designed to handle big datasets quickly and efficiently, much like big databases such as SQLite, but is specifically optimized for analytics and processing large files, including Parquet files.

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
```

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
```

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 3 : Apache arrow method

```
zcat < ~/mimic/icu/chartevents.csv.gz | \
awk -F, 'BEGIN {OFS = ","}

{
    if ($7 == 220045 || $7 == 220181 || $5 == 220179 ||
        $7 == 223761 || $7 == 220210)
        print $1, $5, $7, $9
}' | gzip > chartevents_filtered.csv.gz
```

```
system("gunzip -k ./chartevents_filtered.csv.gz")
```

```
library(arrow)
library(dplyr)

# Measure time taken for ingestion, selection, and filtering using Apache Arrow
time_taken_arrow <- system.time({
  # Load the dataset using Apache Arrow
  dataset <- open_dataset("./chartevents_filtered.csv", format = "csv")

  # Assuming the correct column names from the schema, rename accordingly
  # Update these names based on the actual schema
  renamed_dataset <- dataset %>%
    rename(
      # Replace with actual subject_id column name
      subject_id = `10000032`,
      # Replace with actual charttime column name
      charttime = `2180-07-23 14:00:00`,
      # Replace with actual item_id column name
      item_id = `223761`,
      # Replace with actual valuenum column name
      valuenum = `98.7`
    )

  # Select the relevant columns and filter based on the 'item_id' values
  result <- renamed_dataset %>%
    select(subject_id, charttime, item_id, valuenum) %>%
    filter(item_id %in% c(220045, 220181, 220179, 223761, 220210))

  # Collect the result as a tibble
```

```
  result_tibble <- result %>%
    collect()
})

# Print the time taken for the entire process
print(time_taken_arrow)
```

```
   user  system elapsed
  2.904   0.449   2.825
```

```
# Display the number of rows and the first 10 rows of the result tibble
print(nrow(result_tibble))
```

```
[1] 24816685
```

```
print(ncol(result_tibble))
```

```
[1] 4
```

```
print(head(result_tibble, 10))
```

```
# A tibble: 10 x 4
   subject_id charttime           item_id valuenum
        <int> <dttm>                <int>    <dbl>
 1   10000032 2180-07-23 07:11:00  220181       56
 2   10000032 2180-07-23 07:12:00  220045       91
 3   10000032 2180-07-23 07:12:00  220210       24
 4   10000032 2180-07-23 07:30:00  220045       93
 5   10000032 2180-07-23 07:30:00  220181       67
 6   10000032 2180-07-23 07:30:00  220210       21
 7   10000032 2180-07-23 08:00:00  220045       94
 8   10000032 2180-07-23 08:00:00  220181       64
 9   10000032 2180-07-23 08:00:00  220210       23
10   10000032 2180-07-23 09:00:00  220045      105
```

```
# Check number of rows in original compressed file including header
zcat < chartevents_filtered.csv.gz | tail -n +2 | wc -l
```

```
 24816685
```

```
# Check number of columns
zcat < chartevents_filtered.csv.gz | head -n 1 | awk -F',' '{print NF}'
```

```
4
```

The number of rows(24816685) and columns(4) match correctly in the compressed file and the loaded dataset.

- Ingests: Reads the CSV file using Apache Arrow's open_dataset(), which is designed to handle larger-than-memory datasets efficiently.
- Renames Columns: Changes the names of the columns from their raw schema format to more understandable ones (subject_id, charttime, etc.).
- Selects and Filters: Focuses on specific columns and filters the rows based on conditions applied to the item_id column.
- Collects the Data: The result is collected into a local tibble, allowing it to be printed and analyzed in memory.
- Timing: Measures the time taken for the entire process of ingestion, renaming, selection, filtering, and collection.

The process of loading the dataset using Apache Arrow, selecting relevant columns, and filtering based on item IDs took around 2.82 seconds and resulted in a dataset with 24.8 million rows. This demonstrates the efficiency of using Apache Arrow for handling large datasets.