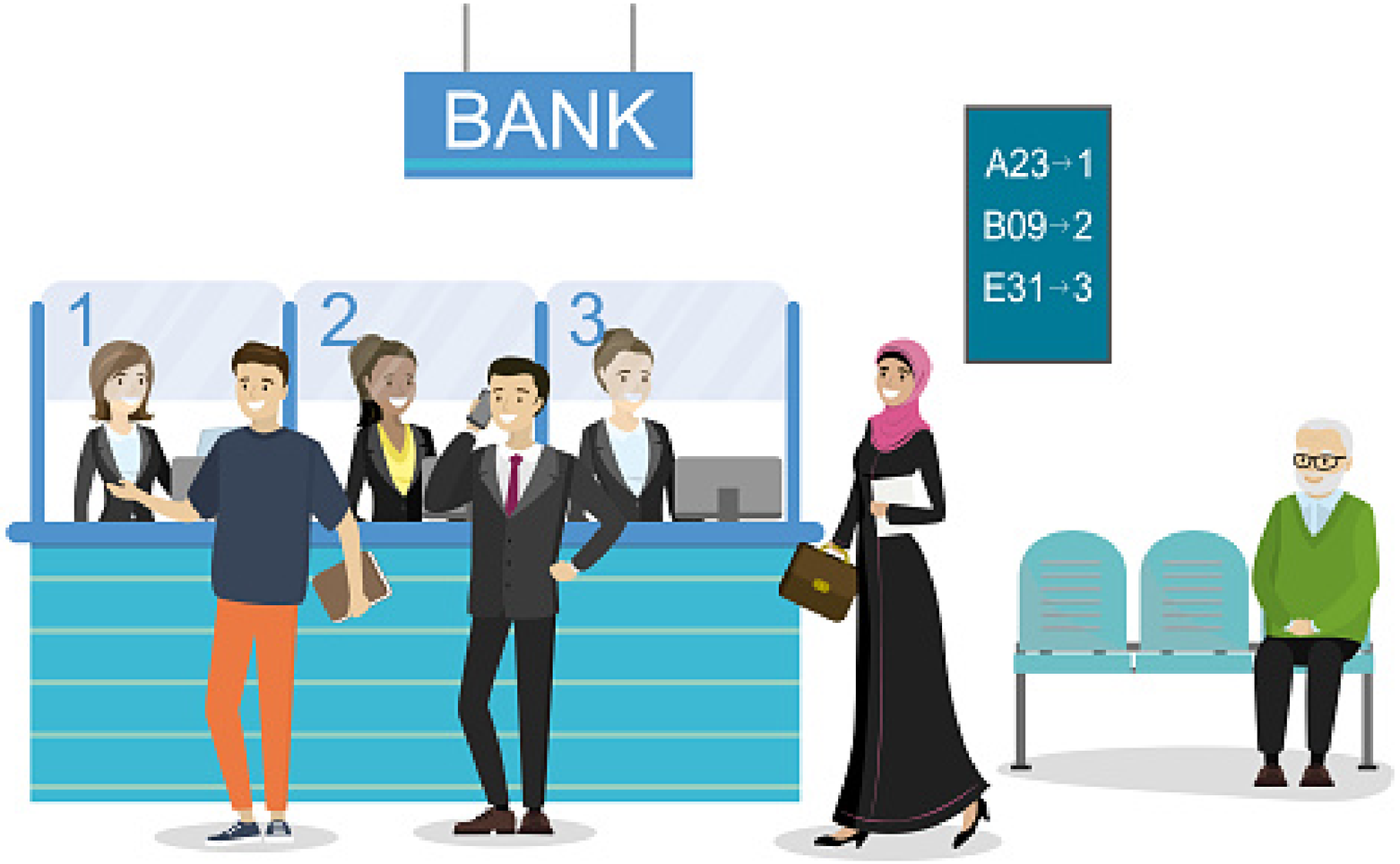
Monitoring and managing memory

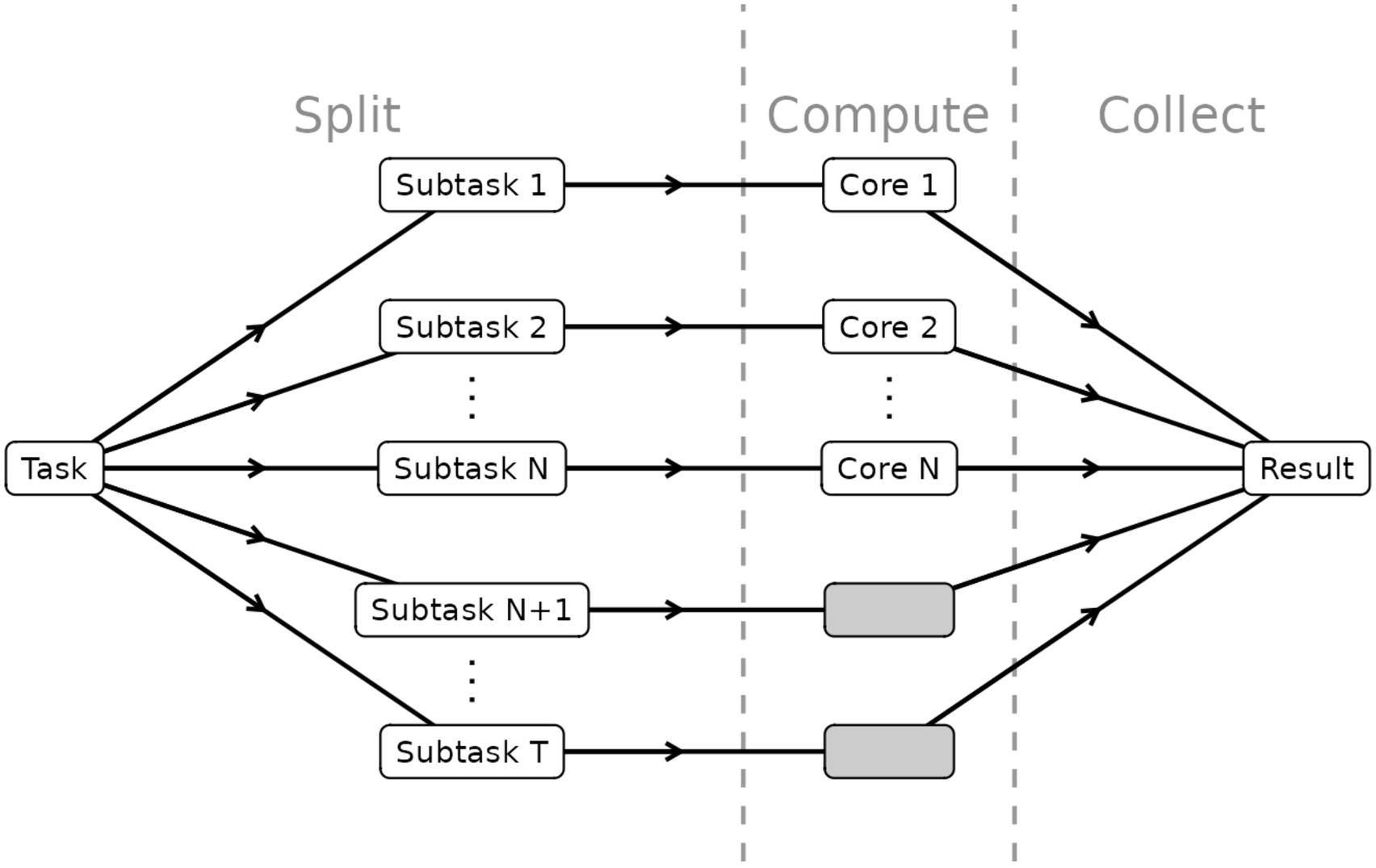
PARALLEL PROGRAMMING IN R

Nabeel Imam
Data Scientist

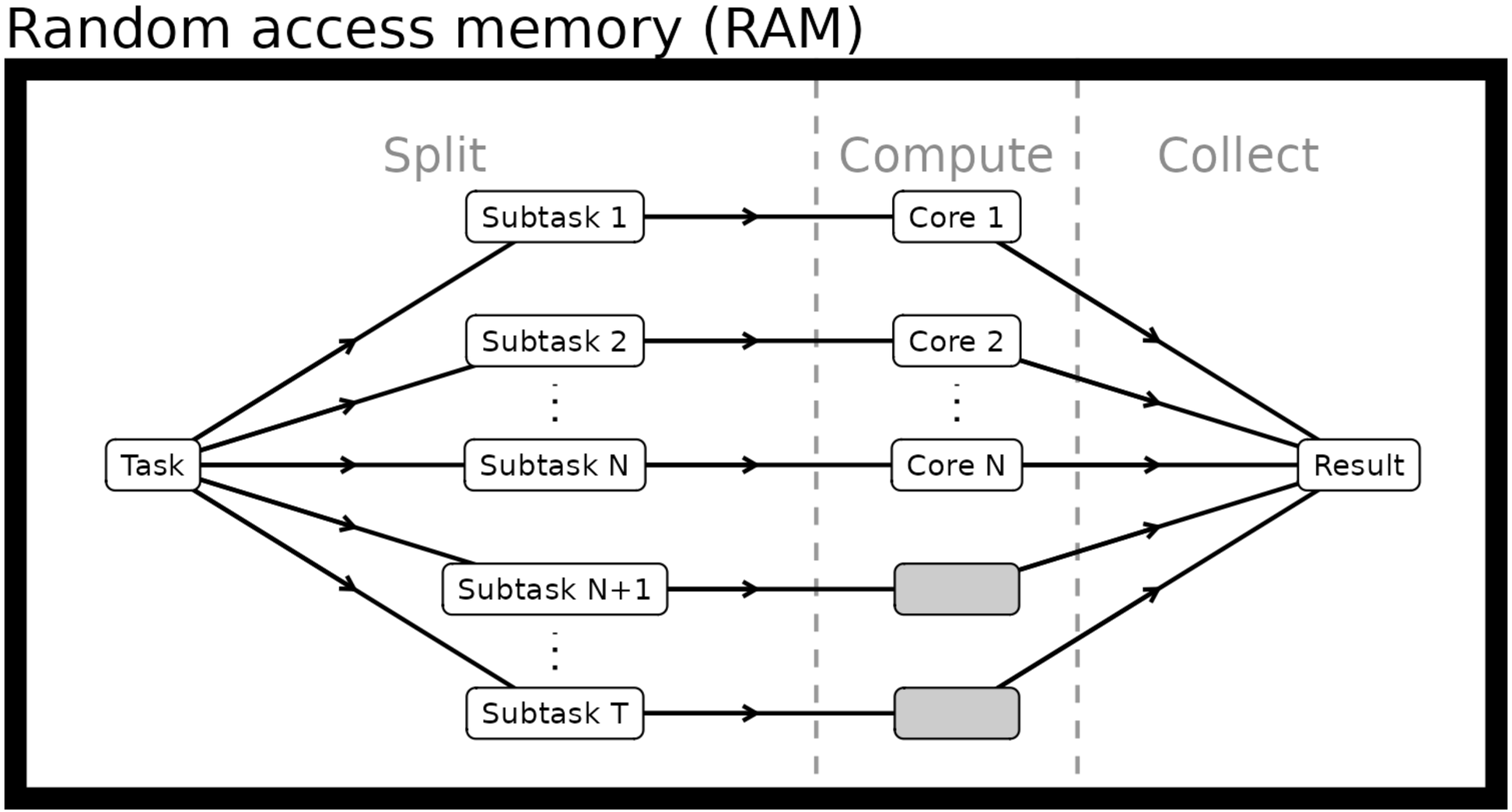
The queue and the space



The parallel flow



The parallel flow



The births data

```
print(ls_files)
```

```
 [1] "./births/AK.csv"
 [2] "./births/AL.csv"
 [3] "./births/AR.csv"
 [4] "./births/AZ.csv"
 [5] "./births/CA.csv"
 [6] "./births/CO.csv"
 [7] "./births/CT.csv"
 [8] "./births/DC.csv"
 [9] "./births/DE.csv"
 [10] "./births/FL.csv"
...
```

Mapping with futures

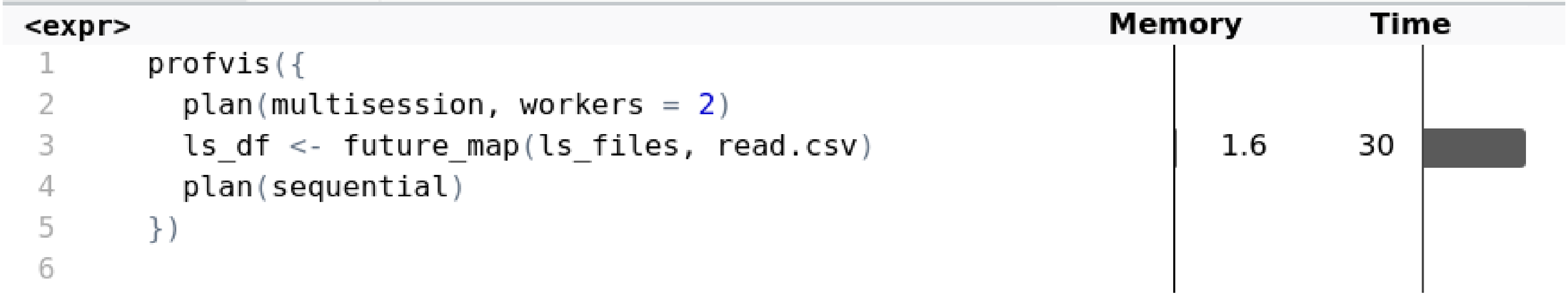
```
plan(multisession, workers = 2)
ls_df <- future_map(ls_files, read.csv)
plan(sequential)

print(ls_df)
```

```
[[1]]
   state month plurality weight_gain_pounds mother_age
      AK     1         1                 30         43
   ...
[[2]]
   state month plurality weight_gain_pounds mother_age
      AL    10         1                 60         33
   ...
...
```

Profiling with two workers

```
profvis({
  plan(multisession, workers = 2)
  ls_df <- future_map(ls_files, read.csv)
  plan(sequential)
})
```

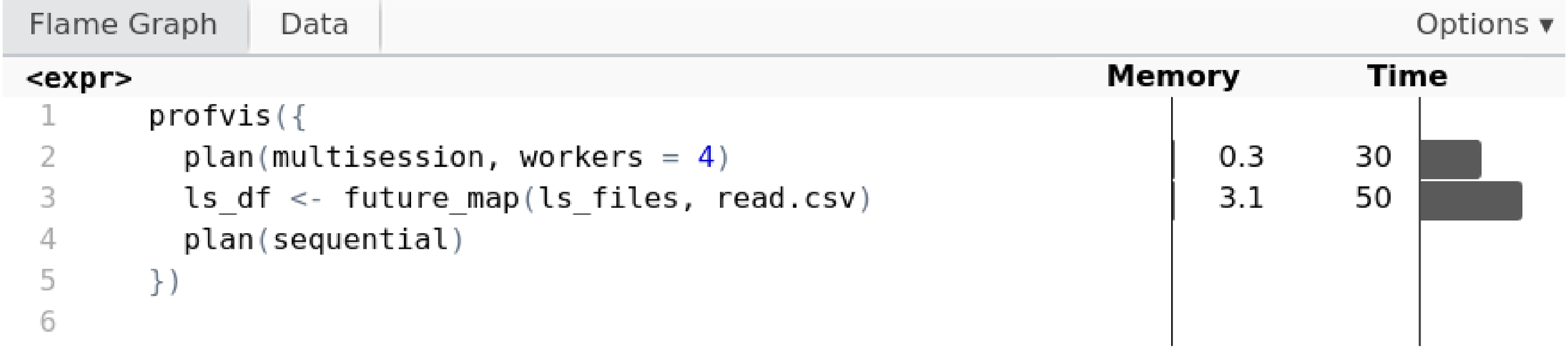
Flame Graph | Data | Options ▾



Profiling with four workers

```
profvis({
  plan(multisession, workers = 4)
  ls_df <- future_map(ls_files, read.csv)
  plan(sequential)
})
```

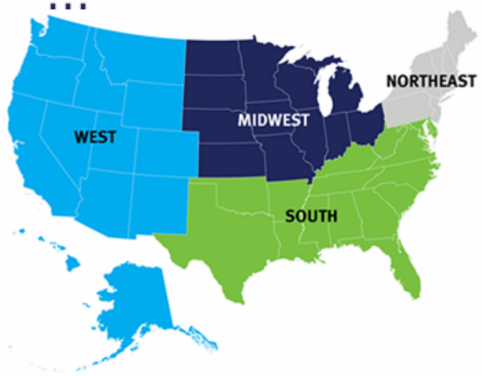
Flame Graph | Data | Options ▾



Behind the scenes

[1] "./births/ND.csv"
[2] "./births/SD.csv"
[3] "./births/NE.csv"
[4] "./births/MN.csv"
[5] "./births/MO.csv"
...

[1] "./births/WA.csv"
[2] "./births/OR.csv"
[3] "./births/CA.csv"
[4] "./births/NV.csv"
[5] "./births/AK.csv"
...



[1] "./births/NY.csv"
[2] "./births/ME.csv"
[3] "./births/PA.csv"
[4] "./births/MA.csv"
[5] "./births/NJ.csv"
...

[1] "./births/TX.csv"
[2] "./births/GA.csv"
[3] "./births/FL.csv"
[4] "./births/AL.csv"
[5] "./births/MS.csv"
...

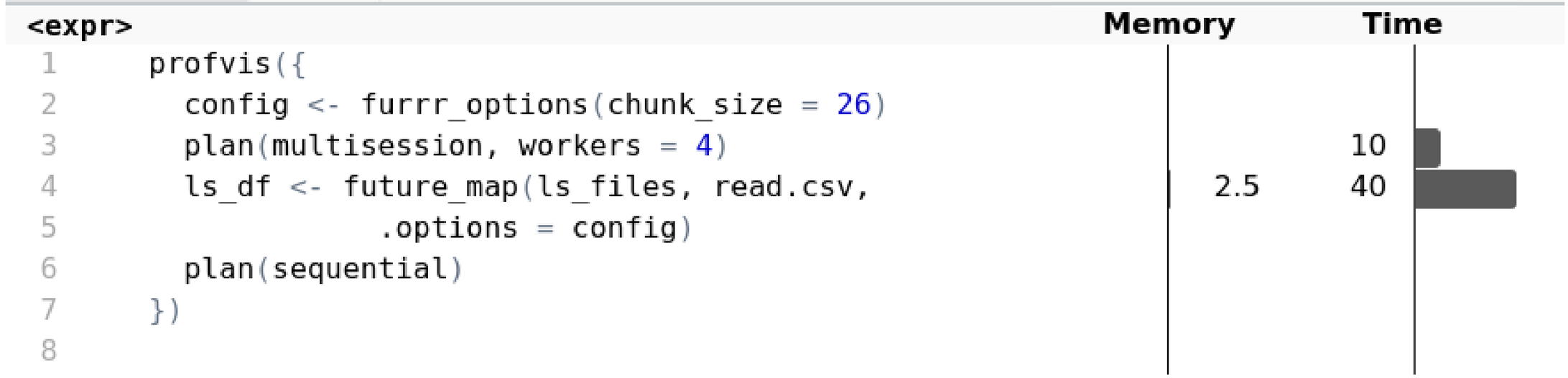
Managing memory by chunking

```
config <- furrr_options(chunk_size = 26)
plan(multisession, workers = 4)
ls_df <- future_map(ls_files, read.csv,
                    .options = config)
plan(sequential)
```

Managing memory by chunking

```
profvis({
  config <- furrr_options(chunk_size = 26)
  plan(multisession, workers = 4)
  ls_df <- future_map(ls_files, read.csv,
              .options = config)
  plan(sequential)
})
```

Flame Graph | Data | Options ▾

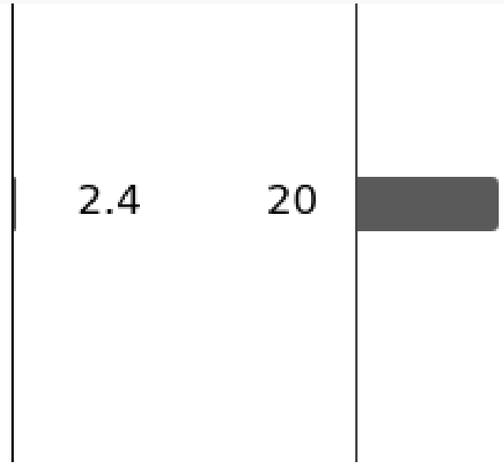


Chunking with parallel

```
cl <- makeCluster(4)


ls_df <- parLapply(cl, ls_files, read.csv)


stopCluster(cl)
```

Flame Graph | Data | Options ▾

<expr>	Memory	Time
1 profvis({		
2 cl <- makeCluster(4)		
3		
4 ls_df <- parLapply(cl, ls_files, read.csv)	2.4	20
5		
6 stopCluster(cl)		
7 })		
8		



Chunking with parallel

```
cl <- makeCluster(4)
ls_df <- parLapply(cl, ls_files, read.csv,
                   chunk.size = 26)
stopCluster(cl)
```

Flame Graph | Data | Options ▾

<expr>	Memory	Time
1 profvis({		
2 cl <- makeCluster(4)		
3 ls_df <- parLapply(cl, ls_files, read.csv,	1.0	20
4 chunk.size = 26)		
5 stopCluster(cl)		
6 })		
7		

When to chunk?

- Chunking is performed optimally by default
- With large data objects and running low on memory
  - Try using fewer cores if feasible
  - Experiment with a few chunk sizes to get to optimum

Let's practice!

PARALLEL PROGRAMMING IN R

Reproducibility in parallel

PARALLEL PROGRAMMING IN R

Nabeel Imam
Data Scientist

What is reproducibility?

Same input produces the same results every time we run the code

- Code can be tested
- Results can be replicated by others

The customer lucky draw

```
print(customer_ids)
```

```
$USA
   [1] 465500 612953 106420 279492 376941 163474 164493 801983 898941 406844 829157 ...
$Canada
   [1] 140521 398164 817703 715385 771801 656814 721270 719120 425819 774558 111418 ...
$Mexico
   [1] 714842 486725 706765 858020 790364 390760 198667 419197 352989 202494 756636 ...
$UK
   [1] 886285 151731 274940 779966 375535 431644 880434 649074 765423 449147 408041 ...
```

The customer lucky draw

```
lucky_draw <- function (ids) {
  sample(ids, 1)
}

cl <- makeCluster(4)

set.seed(1234)
parLapply(cl, customer_ids, lucky_draw)
stopCluster(cl)
```

```
$USA
[1] 673576

$Canada
[1] 164613

$Mexico
[1] 769658

$UK
[1] 683102
```

The reproducibility problem

Winners from first run

```
$USA
[1] 673576

$Canada
[1] 164613

$Mexico
[1] 769658

$UK
[1] 683102
```

Winners from second run

```
$USA
[1] 638051

$Canada
[1] 133431

$Mexico
[1] 522137

$UK
[1] 856141
```

Solution

```
cl <- makeCluster(4)

# A seed for all worker processes in cluster
clusterSetRNGStream(cl, 1234)

parLapply(cl, customer_ids, lucky_draw)
stopCluster(cl)
```

Multiple runs with same results

Winners from first run

```
$USA
[1] 421408

$Canada
[1] 877562

$Mexico
[1] 460786

$UK
[1] 658513
```

Winners from second run

```
$USA
[1] 421408

$Canada
[1] 877562

$Mexico
[1] 460786

$UK
[1] 658513
```

Multiple runs with same results

First run

```
cl <- makeCluster(4)

clusterSetRNGStream(cl, 1234)

run1 <- parLapply(cl, customer_ids, lucky_draw)
stopCluster(cl)
```

Second run

```
cl <- makeCluster(4)

clusterSetRNGStream(cl, 1234)

run2 <- parLapply(cl, customer_ids, lucky_draw)
stopCluster(cl)

identical(run1, run2)
```

```
[1] TRUE
```

Reproducible results with furrr

First run

```
config <- furrr_options(seed = 1234)

plan(multisession, workers = 4)

run1 <- future_map(customer_ids, lucky_draw,
                   .options = config)
plan(sequential)
```

Second run

```
plan(multisession, workers = 4)

run2 <- future_map(customer_ids, lucky_draw,
                   # Using the same configuration
                   .options = config)
plan(sequential)

identical(run1, run2)
```

```
[1] TRUE
```

Reproducible results with foreach

First run

```
install.packages("doRNG")
library(doRNG)

cl <- makeCluster(4)
registerDoParallel(cl)
registerDoRNG(1234)

run1 <- foreach(i = customer_ids) %dopar% {
  lucky_draw(i)
}
stopCluster(cl)
```

Second run

```
cl <- makeCluster(4)
registerDoParallel(cl)
registerDoRNG(1234) # Same seed

run2 <- foreach(i = customer_ids) %dopar% {
  lucky_draw(i)
}
stopCluster(cl)

identical(run1, run2)
```

```
[1] TRUE
```

When to think about reproducibility

- Direct call to random number generators
  - `rnorm` , `rbinom` , etc
- Sampling randomly
  - Bootstraps
  - `sample_n()` from `dplyr`

Let's practice!

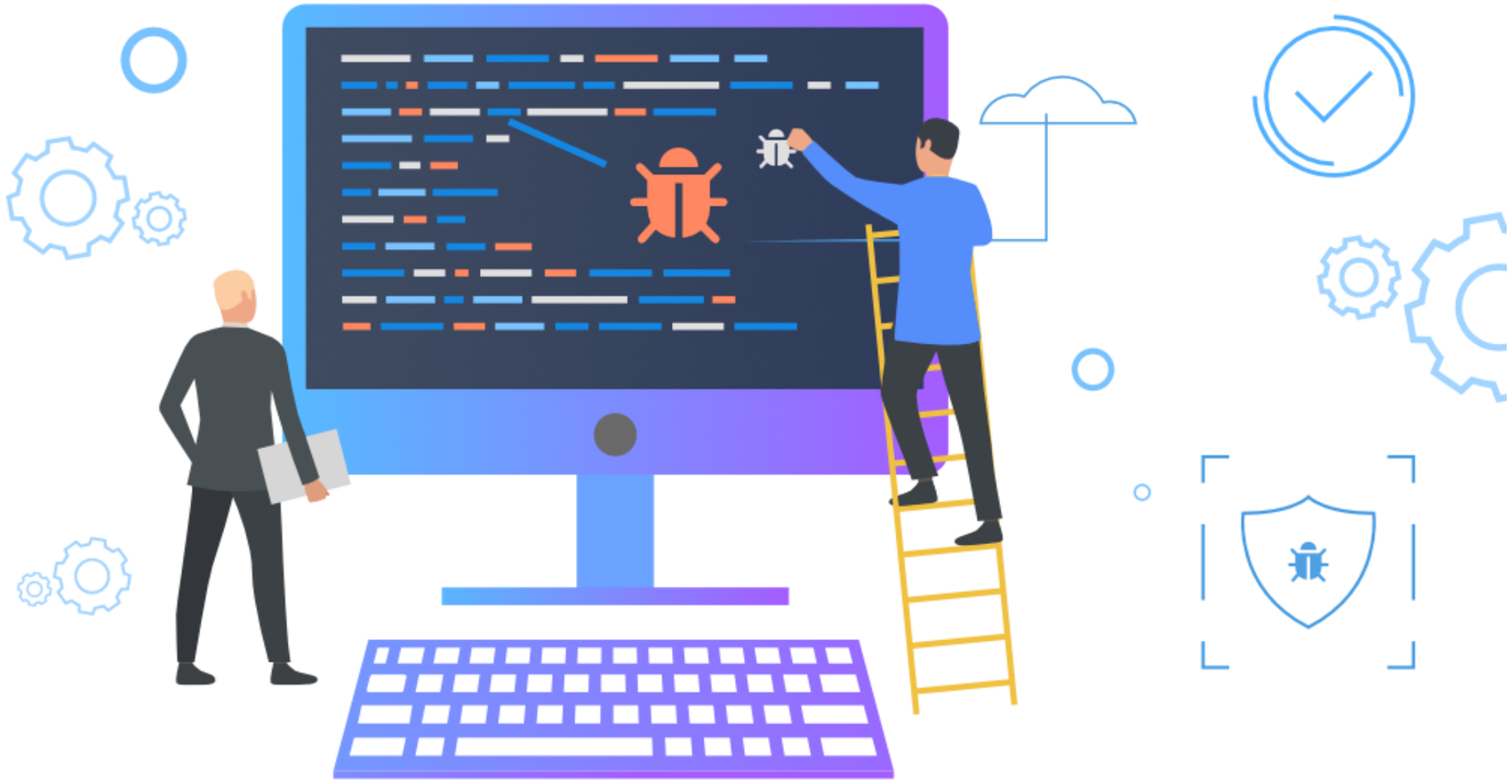
PARALLEL PROGRAMMING IN R

Debugging in parallel

PARALLEL PROGRAMMING IN R

Nabeel Imam
Data Scientist

What is debugging?



Reading files in parallel

```
print(file_list)
```

```
 [1] "./stocks/2011.csv"
 [2] "./stocks/2012.csv"
 [3] "./stocks/2013.csv"
 [4] "./stocks/2014.csv"
 [5] "./stocks/2015.csv"
 ...
```

The filtering function

```
filterCSV <- function (filepath) {

  # Read CSV
  df <- read.csv(filepath)

  # Filter data
  df <- df %>%
    dplyr::filter(Company == "Tesla")

  # Write to back to same path
  write.csv(df, filepath)
}
```

The parallel apply

```
cl <- makeCluster(4)
clusterEvalQ(cl, library(dplyr))

dummy <- parLapply(cl, file_list, filterCSV)
stopCluster(cl)
```

```
Error in checkForRemoteErrors(val) :
  one node produced an error: ? In argument: `Company == "Tesla"`.
Caused by error:
! object 'Company' not found
```

The sequential run

```
short_list <- file_list[1:5]

dummy <- lapply(short_list, filterCSV)

read.csv(short_list[1])
```

```
         Date  Open  High   Low Close Adj.Close   Volume Company Year
1  2011-01-03 5.368 5.400 5.180 5.324     5.324  6415000   Tesla 2011
2  2011-01-04 5.332 5.390 5.204 5.334     5.334  5937000   Tesla 2011
3  2011-01-05 5.296 5.380 5.238 5.366     5.366  7233500   Tesla 2011
...
```

Locate the error

Error message

```
Error in checkForRemoteErrors(val) :
  one node produced an error:
  In argument: `Company == "Tesla"`.
Caused by error:
! object 'Company' not found
```

Locate the error

```
filterCSV <- function (filepath) {

  # Read CSV
  df <- read.csv(filepath)

  # Filter data
  df <- df %>%
    dplyr::filter(Company == "Tesla")

  # Write to back to same path
  write.csv(df, filepath)
}
```

```
filterCSV_debug <- function (filepath) {

  df <- read.csv(filepath)

  print(
    # Paste file path and column names
    paste(filepath, ":",
          # Collapse column names into one string
          paste0(colnames(df), collapse = ","))
  )

  df <- df %>%
    dplyr::filter(Company == "Tesla")

  write.csv(df, filepath)
}
```

Locate the error

```
cl <- makeCluster(4)
clusterEvalQ(cl, library(dplyr))

dummy <- parLapply(cl, file_list, filterCSV_debug)
stopCluster(cl)
```

```
Error in checkForRemoteErrors(val) :
  one node produced an error: ? In argument: `Company == "Microsoft"`.
Caused by error:
! object 'Company' not found
```

Locate the error

```
cl <- makeCluster(4, outfile = "log.txt") # Log print messages into "log.txt"
clusterEvalQ(cl, library(dplyr))

parLapply(cl, file_list, filterCSV_debug)
stopCluster(cl)
```

```
Error in checkForRemoteErrors(val) :
  one node produced an error: ? In argument: `Company == "Tesla"`.
Caused by error:
! object 'Company' not found
```

Examining logs

```
[1] "./stocks/2011.csv : Date,Open,High,Low,Close,Adj.Close,Volume,Company,Year"
[1] "./stocks/2014.csv : Date,Open,High,Low,Close,Adj.Close,Volume,Company,Year"
[1] "./stocks/2017.csv : Date,Open,High,Low,Close,Adj.Close,Volume,Year"
[1] "./stocks/2019.csv : Date,Open,High,Low,Close,Adj.Close,Volume,Company,Year"
[1] "./stocks/2012.csv : Date,Open,High,Low,Close,Adj.Close,Volume,Company,Year"
[1] "./stocks/2013.csv : Date,Open,High,Low,Close,Adj.Close,Volume,Company,Year"
[1] "./stocks/2015.csv : Date,Open,High,Low,Close,Adj.Close,Volume,Company,Year"
[1] "./stocks/2016.csv : Date,Open,High,Low,Close,Adj.Close,Volume,Company,Year"
[1] "./stocks/2020.csv : Date,Open,High,Low,Close,Adj.Close,Volume,Company,Year"
[1] "./stocks/2021.csv : Date,Open,High,Low,Close,Adj.Close,Volume,Company,Year"
```

Debugging with foreach

```
cl <- makeCluster(4,
                  # Supply a text file name to log print messages
                  outfile = "log.txt")

registerDoParallel(cl)

foreach(f = file_list,
        .packages = "dplyr") %dopar% {
  filterCSV_debug(f)
}

stopCluster(cl)
```

The good thing about furrr

```
plan(multisession, workers = 4)
future_map(file_list, filterCSV_debug)
plan(sequential)
```

The good thing about furrr

```
[1] "./stocks/2011.csv : Date,Open,High,Low,Close,Adj.Close,Volume,Company,Year"
[1] "./stocks/2012.csv : Date,Open,High,Low,Close,Adj.Close,Volume,Company,Year"
[1] "./stocks/2013.csv : Date,Open,High,Low,Close,Adj.Close,Volume,Company,Year"
[1] "./stocks/2014.csv : Date,Open,High,Low,Close,Adj.Close,Volume,Company,Year"
[1] "./stocks/2015.csv : Date,Open,High,Low,Close,Adj.Close,Volume,Company,Year"
[1] "./stocks/2016.csv : Date,Open,High,Low,Close,Adj.Close,Volume,Company,Year"
[1] "./stocks/2017.csv : Date,Open,High,Low,Close,Adj.Close,Volume,Year"
Error in (function (.x, .f, ..., .progress = FALSE)  :
  ? In index: 1.
Caused by error in `dplyr::filter()`:
? In argument: `Company == "Tesla"`.
Caused by error:
! object 'Company' not found
```

The steps

For errors in parallel

- Do a sequential run on a subset of the input
- Examine the error message and print appropriate messages
- Locate the error by printing or logging messages
- Fix the error

Let's practice!

PARALLEL PROGRAMMING IN R

Advanced debugging

PARALLEL PROGRAMMING IN R

Nabeel Imam
Data Scientist

The default error behavior

Toy example

```
var_list <- list(1:10,
                 1:10,
                 c("a", "b", "c"))

lapply(var_list, sqrt)
```

```
Error in FUN(X[[i]], ...) : non-numeric
argument to mathematical function
```

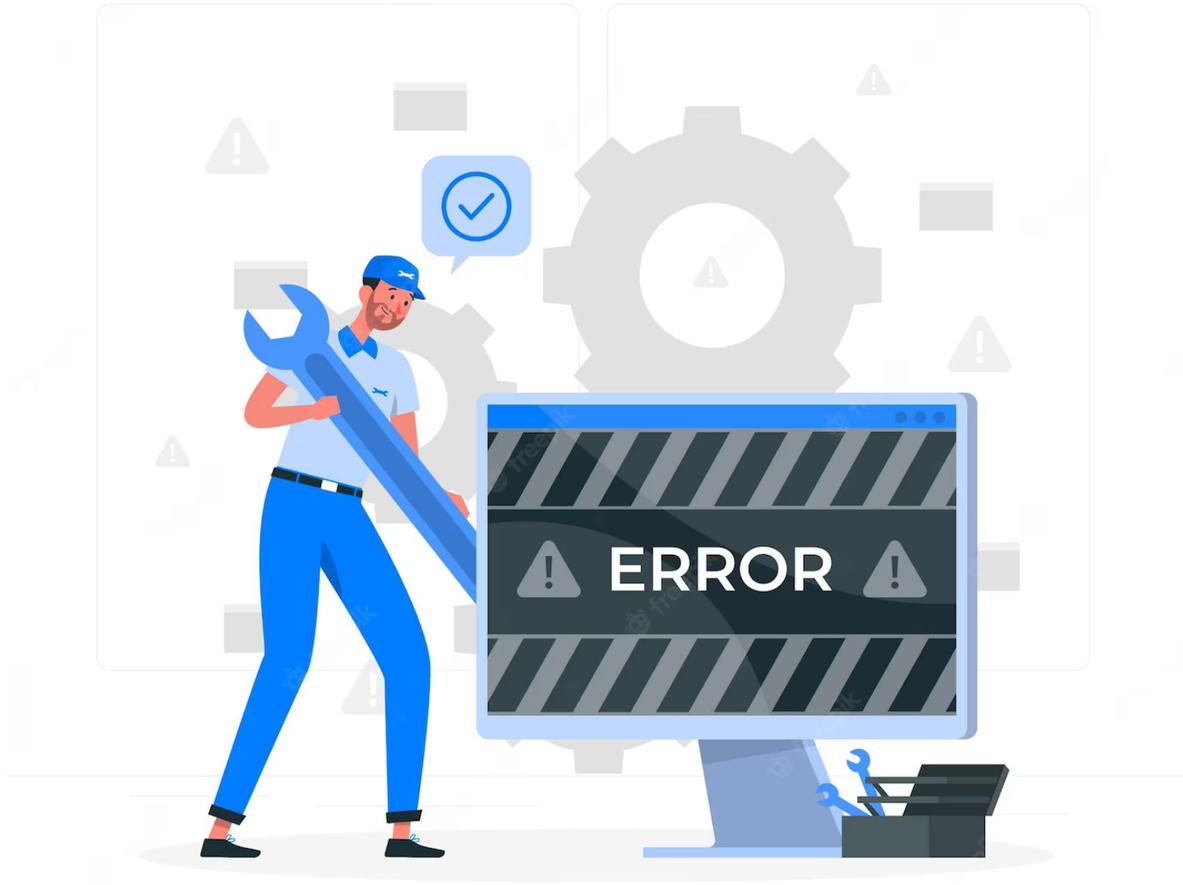
- Numeric
- Numeric
- Character

Ambiguity

[1] Designed by Freepik

Catching the error

```
sqrt_custom <- function(var) {
  tryCatch(
    # Expression to evaluate
    sqrt(var),
    # What to do with an error
    error = function (e) return(e)
    )
}
```



[1] Designed by Freepik

Catching the error

```
lapply(var_list, sqrt_custom)
```

```
[[1]]
 [1] 1.000000 1.414214 1.732051 2.000000 2.236068 2.449490 2.645751 2.828427 3.000000
[10] 3.162278

[[2]]
 [1] 1.000000 1.414214 1.732051 2.000000 2.236068 2.449490 2.645751 2.828427 3.000000
[10] 3.162278

[[3]]
<simpleError in sqrt(var): non-numeric argument to mathematical function>
```

Catching errors in parallel

```
cl <- makeCluster(3)
parLapply(cl, var_list, sqrt_custom)
stopCluster(cl)
```

```
[[1]]
 [1] 1.000000 1.414214 1.732051 2.000000 2.236068 2.449490 2.645751 2.828427 3.000000
[10] 3.162278

[[2]]
 [1] 1.000000 1.414214 1.732051 2.000000 2.236068 2.449490 2.645751 2.828427 3.000000
[10] 3.162278

[[3]]
<simpleError in sqrt(var): non-numeric argument to mathematical function>
```

The births data

```
print(ls_births)
```

```
$AK
 month  plurality
     1           1
     2           1
 ...
$AL
   month  plurality
      10           1
       9           1
 ...
...
```

The summarizing function

```
summarise_births <- function(df) {

  tryCatch({
    df %>%
      group_by(month) %>%
      summarise(total = sum(plurality))
  },
    error = function (e) "Error! Check data")

}
```

Parallel apply

```
cl <- makeCluster(4)
clusterEvalQ(cl, library(dplyr))
parLapply(cl, ls_births, summarise_births)
stopCluster(cl)
```

```
$AK
    month total
  1     1     7
  2     2    13
  ...

$AL
[1] "Error! Check data"

$AR
    month total
  1     1    28
  2     2    28
  ...
...
```

Examine the source

```
head(ls_births[["AL"]], n = 10)
```

```
      month plurality
263      10          1
335       9          1
473      12          1
474       6          1
475       9          1
839       9          1
1291     11      Twins
1369      4          1
1609      5          1
1610      5   Triplets
```

Future map

```
plan(multisession, workers = 4)
config <- furrr_options(packages = "dplyr")
future_map(ls_births, summarise_births,
           .options = config)
plan(sequential)
```

```
$AK
    month total
  1     1     7
  2     2    13
  ...

$AL
[1] "Error! Check data"

$AR
    month total
  1     1    28
  2     2    28
  ...
...
```

The foreach case

```
cl <- makeCluster(4)
registerDoParallel(cl)

foreach(df = ls_births,
        .packages = "dplyr"
       ) %dopar% {
  summarise_births(df)
}

stopCluster(cl)
```

```
$AK
    month total
  1     1     7
  2     2    13
  ...

$AL
[1] "Error! Check data"

$AR
    month total
  1     1    28
  2     2    28
  ...
...
```

Let's practice!

PARALLEL PROGRAMMING IN R

That's a wrap!

PARALLEL PROGRAMMING IN R

Nabeel Imam
Data Scientist

datacamp

What you learned

Parallel programming basics

- Clusters, cores, or workers
- Profiling with `profvis()`
- Benchmarking with `microbenchmark()`

What you learned

Functional

- `parallel`
  - `parLapply()` and family
  - `clusterMap()` for multiple inputs
- `furrr`
  - `future_map()` and other variants
  - `future_pmap()` for multiple inputs

Loops

- `foreach`
  - Parallel loops with `%dopar%`
  - Iterators

What you learned

Troubleshooting

- Memory management
- Reproducibility
- Debugging

Further learning

- **Parallel R** by Q. Ethan McCallum and Stephen Weston
- **Mastering Parallel Programming with R** by Simon R. Chapple et al.

Congratulations!

PARALLEL PROGRAMMING IN R