disk.frame - larger-than-RAM data manipulatoin

Yihui Xie

2019-08-17

Contents

Chapter 1

The story of how `disk.frame` came to be

I was working at a one of Australia's largest banks and their shiny new SAS server was experiencing huge instability issues. As a result, we had to SAS on our laptop to perform huge amounts of data manipulation. A simple SQL query can take up wards of 20 minutes.

I had enough.

That's why I created `disk.frame` - a larger-than-RAM data manipulation framework for R.

Chapter 2

Introduction

2.1 The story of how `disk.frame` came to be

I was working at a one of Australia's largest banks and their shiny new SAS server was experiencing huge instability issues. As a result, we had to SAS on our laptop to perform huge amounts of data manipulation. A simple SQL query can take up wards of 20 minutes.

I had enough.

That's why I created `disk.frame` - a larger-than-RAM data manipulation framework for R.

Chapter 3

Quick Start - replicating dplyr's tutorial on nycflight13

The `disk.frame` package aims to be the answer to the question: how do I manipulate structured tabular data that doesn't fit into Random Access Memory (RAM)?

In a nutshell, `disk.frame` makes use of two simple ideas

1) split up a larger-than-RAM dataset into chunks and store each chunk in a separate file inside a folder and
2) provide a convenient API to manipulate these chunks

`disk.frame` performs a similar role to distributed systems such as Apache Spark, Python's Dask, and Julia's JuliaDB.jl for *medium data* which are datasets that are too large for RAM but not quite large enough to qualify as *big data*.

In this tutorial, we introduce `disk.frame`, address some common questions, and replicate the sparklyr data manipulation tutorial using `disk.frame` constructs.

3.1 Installation

Simply run

```r
install.packages("disk.frame") # when CRAN ready
```

or

```
devtools::install_github("xiaodaigh/disk.frame")
```

3.2 Set-up disk.frame

disk.frame works best if it can process multiple data chunks in parallel. The best way to set-up disk.frame so that each CPU core runs a background worker is by using

```
setup_disk.frame()
```

The setup_disk.frame() sets up background workers equal to the number of CPU cores; please note that, by default, hyper-threaded cores are counted as one not two.

Alternatively, one may specify the number of workers using setup_disk.frame(workers = n).

3.3 Basic Data Operations with disk.frame

The disk.frame package provides convenient functions to convert data.frames and CSVs to disk.frames.

3.3.1 Creating a disk.frame from data.frame

We convert a data.frame to disk.frame using the as.data.frame function.

```
library(nycflights13)
library(dplyr)
library(disk.frame)
library(data.table)

# convert the flights data to a disk.frame and store the disk.frame in the folder
# "tmp_flights" and overwrite any content if needed
flights.df <- as.disk.frame(
  flights,
  outdir = file.path(tempdir(), "tmp_flights.df"),
  overwrite = TRUE)
#> Warning in expand_(path, Sys.getenv("R_FS_HOME") != "" || is_windows()):
#> '.Random.seed' is not an integer vector but of type 'NULL', so ignored
```

```
flights.df
#> path: "C:\Users\RTX2080\AppData\Local\Temp\RtmpAVR4h1/tmp_flights.df"
#> nchunks: 6
#> nrow: 336776
#> ncol: 19
```

You should now see a folder called `tmp_flights` with some files in it, namely `1.fst`, `2.fst`…. where each `fst` files is one chunk of the `disk.frame`.

3.3.2 Creating a disk.frame from CSV

```
library(nycflights13)
# write a csv
csv_path = file.path(tempdir(), "tmp_flights.csv")
data.table::fwrite(flights, csv_path)

# load the csv into a disk.frame
df_path = file.path(tempdir(), "tmp_flights.df")
flights.df <- csv_to_disk.frame(
  csv_path,
  outdir = df_path,
  overwrite = T)

flights.df
#> path: "C:\Users\RTX2080\AppData\Local\Temp\RtmpAVR4h1/tmp_flights.df"
#> nchunks: 6
#> nrow: 336776
#> ncol: 19
```

If the CSV is too large to read in, then we can also use the `in_chunk_size` option to control how many rows to read in at once. For example to read in the data 100,000 rows at a time.

```
library(nycflights13)
library(disk.frame)

# write a csv
csv_path = file.path(tempdir(), "tmp_flights.csv")

data.table::fwrite(flights, csv_path)

df_path = file.path(tempdir(), "tmp_flights.df")
```

```
flights.df <- csv_to_disk.frame(
  csv_path,
  outdir = df_path,
  in_chunk_size = 100000)
#> read 336776 rows from C:\Users\RTX2080\AppData\Local\Temp\RtmpAVR4h1/tmp_flights.cs

flights.df
#> path: "C:\Users\RTX2080\AppData\Local\Temp\RtmpAVR4h1/tmp_flights.df"
#> nchunks: 4
#> nrow: 336776
#> ncol: 19
```

`disk.frame` also has a function `zip_to_disk.frame` that can convert every CSV in a zip file to `disk.frame`s.

3.3.3 Simple `dplyr` verbs and lazy evaluation

```
flights.df1 <- select(flights.df, year:day, arr_delay, dep_delay)
flights.df1
#> path: "C:\Users\RTX2080\AppData\Local\Temp\RtmpAVR4h1/tmp_flights.df"
#> nchunks: 4
#> nrow: 336776
#> ncol: 19
```

```
class(flights.df1)
#> [1] "disk.frame"        "disk.frame.folder"
```

The class of `flights.df1` is also a `disk.frame` after the `dplyr::select` transformation. Also, `disk.frame` operations are by default (and where possible) **lazy**, meaning it doesn't perform the operations right away. Instead, it waits until you call `collect`. Exceptions to this rule are the `*_join` operations which evaluated *eagerly* under certain conditions see **Joins for disk.frame in-depth** for details.

For lazily constructed `disk.frame`s (e.g. `flights.df1`). The function `collect` can be used to bring the results from disk into R, e.g.

```
collect(flights.df1) %>% head
#>    year month day arr_delay dep_delay
#> 1: 2013     1   1        11         2
#> 2: 2013     1   1        20         4
#> 3: 2013     1   1        33         2
#> 4: 2013     1   1       -18        -1
#> 5: 2013     1   1       -25        -6
#> 6: 2013     1   1        12        -4
```

Of course, for larger-than-RAM datasets, one wouldn’t call `collect` on the whole `disk.frame` (because why would you need `disk.frame` otherwise). More likely, one would call `collect` on a `filter`ed dataset or one summarized with `group_by`.

Some examples of other dplyr verbs applied:

```
filter(flights.df, dep_delay > 1000) %>% collect %>% head
#>   year month day dep_time sched_dep_time dep_delay arr_time sched_arr_time
#> 1 2013     1   9      641            900      1301     1242           1530
#> 2 2013     1  10     1121           1635      1126     1239           1810
#> 3 2013     6  15     1432           1935      1137     1607           2120
#> 4 2013     7  22      845           1600      1005     1044           1815
#> 5 2013     9  20     1139           1845      1014     1457           2210
#>   arr_delay carrier flight tailnum origin dest air_time distance hour
#> 1      1272      HA     51  N384HA    JFK  HNL      640     4983    9
#> 2      1109      MQ   3695  N517MQ    EWR  ORD      111      719   16
#> 3      1127      MQ   3535  N504MQ    JFK  CMH       74      483   19
#> 4       989      MQ   3075  N665MQ    JFK  CVG       96      589   16
#> 5      1007      AA    177  N338AA    JFK  SFO      354     2586   18
#>   minute            time_hour
#> 1      0 2013-01-09T14:00:00Z
#> 2     35 2013-01-10T21:00:00Z
#> 3     35 2013-06-15T23:00:00Z
#> 4      0 2013-07-22T20:00:00Z
#> 5     45 2013-09-20T22:00:00Z
```

```
mutate(flights.df, speed = distance / air_time * 60) %>% collect %>% head
#>   year month day dep_time sched_dep_time dep_delay arr_time sched_arr_time
#> 1 2013     1   1      517            515         2      830            819
#> 2 2013     1   1      533            529         4      850            830
#> 3 2013     1   1      542            540         2      923            850
#> 4 2013     1   1      544            545        -1     1004           1022
#> 5 2013     1   1      554            600        -6      812            837
#> 6 2013     1   1      554            558        -4      740            728
#>   arr_delay carrier flight tailnum origin dest air_time distance hour
#> 1        11      UA   1545  N14228    EWR  IAH      227     1400    5
#> 2        20      UA   1714  N24211    LGA  IAH      227     1416    5
#> 3        33      AA   1141  N619AA    JFK  MIA      160     1089    5
#> 4       -18      B6    725  N804JB    JFK  BQN      183     1576    5
#> 5       -25      DL    461  N668DN    LGA  ATL      116      762    6
#> 6        12      UA   1696  N39463    EWR  ORD      150      719    5
#>   minute            time_hour    speed
#> 1     15 2013-01-01T10:00:00Z 370.0441
#> 2     29 2013-01-01T10:00:00Z 374.2731
#> 3     40 2013-01-01T10:00:00Z 408.3750
#> 4     45 2013-01-01T10:00:00Z 516.7213
#> 5      0 2013-01-01T11:00:00Z 394.1379
#> 6     58 2013-01-01T10:00:00Z 287.6000
```

3.3.4 Examples of NOT fully supported `dplyr` verbs

The `arrange` function arranges (sort) each chunk but not the whole dataset. So use with caution. Similarly `summarise` creates summary variables within each chunk and hence also needs to be used with caution. In the Group By section, we demonstrate how to use `summarise` in the `disk.frame` context correctly with `hard_group_by`s.

```
# this only sorts within each chunk
arrange(flights.df, dplyr::desc(dep_delay)) %>% collect %>% head
#> Warning in arrange.disk.frame(flights.df, dplyr::desc(dep_delay)):
#> disk.frame only sorts (arange) WITHIN each chunk
#>   year month day dep_time sched_dep_time dep_delay arr_time sched_arr_time
#> 1 2013     1   9      641            900      1301     1242           1530
#> 2 2013     1  10     1121           1635      1126     1239           1810
#> 3 2013    12   5      756           1700       896     1058           2020
#> 4 2013     1   1      848           1835       853     1001           1950
#> 5 2013    12  19      734           1725       849     1046           2039
#> 6 2013    12  17      705           1700       845     1026           2020
#>   arr_delay carrier flight tailnum origin dest air_time distance hour
#> 1      1272      HA     51  N384HA    JFK  HNL      640     4983    9
#> 2      1109      MQ   3695  N517MQ    EWR  ORD      111      719   16
#> 3       878      AA    172  N5DMAA    EWR  MIA      149     1085   17
#> 4       851      MQ   3944  N942MQ    JFK  BWI       41      184   18
#> 5       847      DL   1223  N375NC    EWR  SLC      290     1969   17
#> 6       846      AA    172  N5EMAA    EWR  MIA      145     1085   17
#>   minute            time_hour
#> 1      0 2013-01-09T14:00:00Z
#> 2     35 2013-01-10T21:00:00Z
#> 3      0 2013-12-05T22:00:00Z
#> 4     35 2013-01-01T23:00:00Z
#> 5     25 2013-12-19T22:00:00Z
#> 6      0 2013-12-17T22:00:00Z
```

```
summarize(flights.df, mean_dep_delay = mean(dep_delay, na.rm =T)) %>% collect
#>   mean_dep_delay
#> 1       8.771247
#> 2      12.742945
#> 3      18.157297
#> 4       8.028106
```

3.3.5 Piping

One can chain `dplyr` verbs together like with a `data.frame`

```
c4 <- flights %>%
  filter(month == 5, day == 17, carrier %in% c('UA', 'WN', 'AA', 'DL')) %>%
  select(carrier, dep_delay, air_time, distance) %>%
  mutate(air_time_hours = air_time / 60) %>%
  collect %>%
  arrange(carrier)# arrange should occur after `collect`

c4  %>% head
#>   carrier dep_delay air_time distance air_time_hours
#> 1      AA        -7      142     1089       2.366667
#> 2      AA        -9      186     1389       3.100000
#> 3      AA        -6      143     1096       2.383333
#> 4      AA        -4      114      733       1.900000
#> 5      AA        -2      146     1085       2.433333
#> 6      AA        -7      119      733       1.983333
```

3.3.6 List of supported `dplyr` verbs

```
select
rename
filter
arrange # within each chunk
group_by # within each chunk
summarise/summarize # within each chunk
mutate
transmute
left_join
inner_join
full_join # careful. Performance!
semi_join
anit_join
```

3.4 Sharding and distribution of chunks

Like other distributed data manipulation frameworks `disk.frame` utilizes the *sharding* concept to distribute the data into chunks. For example "to shard by `cust_id`" means that all rows with the same `cust_id` will be stored in the same chunk. This enables `group_by` by `cust_id` to produce the same results as non-chunked data.

The `by` variables that were used to shard the dataset are called the `shardkeys`. The *sharding* is performed by computing a deterministic hash on the shard keys (the `by` variables) for each row. The hash function produces an integer between `1` and `n`, where `n` is the number of chunks.

3.5 Grouping

The `disk.frame` implements the `group_by` operation with a significant caveat. In the `disk.frame` framework, group-by happen WITHIN each chunk and not ACROSS chunks. To achieve group by across chunk we need to put **all rows with the same group keys into the same file chunk**; this can be achieved with `hard_group_by`. However, the `hard_group_by` operation can be **VERY TIME CONSUMING** computationally and should be **avoided** if possible.

The `hard_group_by` operation is best illustrated with an example, suppose a `disk.frame` has three chunks

```
# chunk1 = 1.fst
#  id n
#1  a 1
#2  a 2
#3  b 3
#4  d 4

# chunk2 = 2.fst
#  id n
#1  a 4
#2  a 5
#3  b 6
#4  d 7

# chunk3 = 3.fst
#  id n
#1  a 4
#2  b 5
#3  c 6
```

and notice that the id column contains 3 distinct values "a","b", and "c". To perform hard_group_by(df, by = id) MAY give you the following disk.frame where all the ids with the same values end up in the same chunks.

```
# chunk1 = 1.fst
#  id n
#1  b 3
#2  b 6

# chunk2 = 2.fst
#  id n
#1  c 6
#2  d 4
#3  d 7

# chunk3 = 3.fst
#  id n
#1  a 1
#2  a 2
#3  a 4
#4  a 5
#5  a 4
```

Also, notice that there is no guaranteed order for the distribution of the ids to the chunks. The order is random, but each chunk is likely to have a similar number of rows, provided that id does not follow a skewed distribution i.e. where a few distinct values make up the majority of the rows.

Typically, group_by is performed WITHIN each chunk. This is not an issue if the chunks have already been sharded on the by variables beforehand; however, if this is not the case then one may need a second stage aggregation to obtain the correct result, see *Two-stage group by*.

By forcing the user to choose group_by (within each chunk) and hard_group_by (across all chunks), this ensures that the user is conscious of the choice they are making. In sparklyr the equivalent of a hard_group_by is performed, which we should avoid, where possible, as it is time-consuming and expensive. Hence, disk.frame has chosen to explain the theory and allow the user to make a conscious choice when performing (hard_)group_by.

```
flights.df %>%
  hard_group_by(carrier) %>% # notice that hard_group_by needs to be set
  summarize(count = n(), mean_dep_delay = mean(dep_delay, na.rm=T)) %>%  # mean follows normal R
  collect %>%
  arrange(carrier)
#> Appending disk.frames:
```

```
#> # A tibble: 16 x 3
#>    carrier count mean_dep_delay
#>    <chr>   <int>          <dbl>
#>  1 9E      18460          16.7
#>  2 AA      32729           8.59
#>  3 AS        714           5.80
#>  4 B6      54635          13.0
#>  5 DL      48110           9.26
#>  6 EV      54173          20.0
#>  7 F9        685          20.2
#>  8 FL       3260          18.7
#>  9 HA        342           4.90
#> 10 MQ      26397          10.6
#> 11 OO         32          12.6
#> 12 UA      58665          12.1
#> 13 US      20536           3.78
#> 14 VX       5162          12.9
#> 15 WN      12275          17.7
#> 16 YV        601          19.0
```

3.5.1 Two-stage group by

For most group-by tasks, the user can achieve the desired result WITHOUT using `hard = TRUE` by performing the group by in two stages. For example, suppose you aim to count the number of rows group by `carrier`, you can set `hard = F` to find the count within each chunk and then use a second group-by to summaries each chunk's results into the desired result. For example,

```
flights.df %>%
  group_by(carrier) %>% # `group_by` aggregates within each chunk
  summarize(count = n()) %>%  # mean follows normal R rules
  collect %>%  # collect each individul chunks results and row-bind into a data.table
  group_by(carrier) %>%
  summarize(count = sum(count)) %>%
  arrange(carrier)
#> # A tibble: 16 x 2
#>    carrier count
#>    <chr>   <int>
#>  1 9E      18460
#>  2 AA      32729
#>  3 AS        714
#>  4 B6      54635
#>  5 DL      48110
#>  6 EV      54173
#>  7 F9        685
#>  8 FL       3260
#>  9 HA        342
#> 10 MQ      26397
#> 11 OO         32
#> 12 UA      58665
#> 13 US      20536
#> 14 VX       5162
#> 15 WN      12275
#> 16 YV        601
```

Because this two-stage approach avoids the expensive `hard group_by` operation, it is often significantly faster. However, it can be tedious to write; and this is a con of the `disk.frame` chunking mechanism.

Note: this two-stage approach is similar to a map-reduce operation.

3.6 Restrict input columns for faster processing

One can restrict which input columns to load into memory for each chunk; this can significantly increase the speed of data processing. To restrict the input columns, use the `srckeep` function which only accepts column names as a string vector.

```
flights.df %>%
  srckeep(c("carrier","dep_delay")) %>%
  hard_group_by(carrier) %>%
  summarize(count = n(), mean_dep_delay = mean(dep_delay, na.rm=T)) %>%  # mean follows normal R
  collect
#> Appending disk.frames:
#> # A tibble: 16 x 3
#>    carrier count mean_dep_delay
#>    <chr>   <int>          <dbl>
#>  1 9E      18460          16.7
#>  2 MQ      26397          10.6
#>  3 UA      58665          12.1
#>  4 US      20536           3.78
#>  5 AA      32729           8.59
#>  6 F9        685          20.2
#>  7 HA        342           4.90
#>  8 VX       5162          12.9
#>  9 WN      12275          17.7
#> 10 B6      54635          13.0
#> 11 FL       3260          18.7
#> 12 OO         32          12.6
#> 13 AS        714           5.80
#> 14 DL      48110           9.26
#> 15 EV      54173          20.0
#> 16 YV        601          19.0
```

Input column restriction is one of the most critical efficiencies provided by `disk.frame`. Because the underlying format allows random access to columns (i.e. retrieve only the columns used for processing), hence one can drastically reduce the amount of data loaded into RAM for processing by keeping only those columns that are directly used to produce the results.

3.7 Joins

`disk.frame` supports many dplyr joins including:

```
left_join
inner_join
semi_join
inner_join
full_join # requires hard_group_by on both left and right
```

In all cases, the left dataset (`x`) must be a `disk.frame`, and the right dataset (`y`) can be either a `disk.frame` or a `data.frame`. If the right dataset is a `disk.frame` and the `shardkey`s are different between the two `disk.frame`s then two expensive `hard` `group_by` operations are performed *eagerly*, one on the left `disk.frame` and one on the right `disk.frame` to perform the joins correctly.

However, if the right dataset is a `data.frame` then `hard_group_by`s are only performed in the case of `full_join`.

Note `disk.frame` does not support `right_join` the user should use `left_join` instead.

The below joins are performed *lazily* because `airlines.dt` is a `data.table` not a `disk.frame`:

```
# make airlines a data.table
airlines.dt <- data.table(airlines)
# flights %>% left_join(airlines, by = "carrier") #
flights.df %>%
  left_join(airlines.dt, by ="carrier") %>%
  collect %>%
  head
```

```
#>    year month day dep_time sched_dep_time dep_delay arr_time
#> 1: 2013     1   1      517            515         2      830
#> 2: 2013     1   1      533            529         4      850
#> 3: 2013     1   1      542            540         2      923
#> 4: 2013     1   1      544            545        -1     1004
#> 5: 2013     1   1      554            600        -6      812
#> 6: 2013     1   1      554            558        -4      740
#>    sched_arr_time arr_delay carrier flight tailnum origin dest air_time
#> 1:            819        11      UA   1545  N14228    EWR  IAH      227
#> 2:            830        20      UA   1714  N24211    LGA  IAH      227
#> 3:            850        33      AA   1141  N619AA    JFK  MIA      160
#> 4:           1022       -18      B6    725  N804JB    JFK  BQN      183
#> 5:            837       -25      DL    461  N668DN    LGA  ATL      116
#> 6:            728        12      UA   1696  N39463    EWR  ORD      150
#>    distance hour minute            time_hour                   name
#> 1:     1400    5     15 2013-01-01T10:00:00Z  United Air Lines Inc.
#> 2:     1416    5     29 2013-01-01T10:00:00Z  United Air Lines Inc.
#> 3:     1089    5     40 2013-01-01T10:00:00Z American Airlines Inc.
#> 4:     1576    5     45 2013-01-01T10:00:00Z        JetBlue Airways
#> 5:      762    6      0 2013-01-01T11:00:00Z   Delta Air Lines Inc.
#> 6:      719    5     58 2013-01-01T10:00:00Z  United Air Lines Inc.
```

```
flights.df %>%
  left_join(airlines.dt, by = c("carrier", "carrier")) %>%
  collect %>%
  tail
#>    year month day dep_time sched_dep_time dep_delay arr_time
#> 1: 2013     9  30       NA           1842        NA       NA
#> 2: 2013     9  30       NA           1455        NA       NA
#> 3: 2013     9  30       NA           2200        NA       NA
#> 4: 2013     9  30       NA           1210        NA       NA
#> 5: 2013     9  30       NA           1159        NA       NA
#> 6: 2013     9  30       NA            840        NA       NA
#>    sched_arr_time arr_delay carrier flight tailnum origin dest air_time
#> 1:           2019        NA      EV   5274  N740EV    LGA  BNA       NA
#> 2:           1634        NA      9E   3393            JFK  DCA       NA
#> 3:           2312        NA      9E   3525            LGA  SYR       NA
#> 4:           1330        NA      MQ   3461  N535MQ    LGA  BNA       NA
#> 5:           1344        NA      MQ   3572  N511MQ    LGA  CLE       NA
#> 6:           1020        NA      MQ   3531  N839MQ    LGA  RDU       NA
#>    distance hour minute            time_hour                     name
#> 1:      764   18     42 2013-09-30T22:00:00Z ExpressJet Airlines Inc.
#> 2:      213   14     55 2013-09-30T18:00:00Z        Endeavor Air Inc.
#> 3:      198   22      0 2013-10-01T02:00:00Z        Endeavor Air Inc.
#> 4:      764   12     10 2013-09-30T16:00:00Z                Envoy Air
#> 5:       419   11      59 2013-09-30T15:00:00Z                Envoy Air
#> 6:       431    8      40 2013-09-30T12:00:00Z                Envoy Air
```

3.8 Window functions and arbitrary functions

`disk.frame` supports all `data.frame` operations, unlike Spark which can only perform those operations that Spark has implemented. Hence windowing functions like `rank` are supported out of the box.

```
# Find the most and least delayed flight each day
bestworst <- flights.df %>%
   srckeep(c("year","month","day", "dep_delay")) %>%
   group_by(year, month, day) %>%
   select(dep_delay) %>%
   filter(dep_delay == min(dep_delay, na.rm = T) || dep_delay == max(dep_delay, na.rm
   collect
#> Adding missing grouping variables: `year`, `month`, `day`
#> Adding missing grouping variables: `year`, `month`, `day`
#> Adding missing grouping variables: `year`, `month`, `day`
#> Adding missing grouping variables: `year`, `month`, `day`


bestworst %>% head
#> # A tibble: 6 x 4
#> # Groups:   year, month, day [1]
#>    year month   day dep_delay
#>   <int> <int> <int>     <int>
#> 1  2013     1     1         2
#> 2  2013     1     1         4
#> 3  2013     1     1         2
#> 4  2013     1     1        -1
#> 5  2013     1     1        -6
#> 6  2013     1     1        -4
```

```
# Rank each flight within a daily
ranked <- flights.df %>%
  srckeep(c("year","month","day", "dep_delay")) %>%
  group_by(year, month, day) %>%
  select(dep_delay) %>%
  mutate(rank = rank(desc(dep_delay))) %>%
  collect
#> Adding missing grouping variables: `year`, `month`, `day`
#> Adding missing grouping variables: `year`, `month`, `day`
#> Adding missing grouping variables: `year`, `month`, `day`
#> Adding missing grouping variables: `year`, `month`, `day`

ranked %>% head
#> # A tibble: 6 x 5
#> # Groups:   year, month, day [1]
#>    year month   day dep_delay  rank
#>   <int> <int> <int>     <int> <dbl>
#> 1  2013     1     1         2   313
#> 2  2013     1     1         4   276
#> 3  2013     1     1         2   313
#> 4  2013     1     1        -1   440
#> 5  2013     1     1        -6   742
#> 6  2013     1     1        -4   633
```

3.9 Arbitrary by-chunk processing

One can apply arbitrary transformations to each chunk of the `disk.frame` by using the `delayed` function which evaluates lazily or the `map.disk.frame(lazy = F)` function which evaluates eagerly. For example to return the number of rows in each chunk

```
flights.df1 <- delayed(flights.df, ~nrow(.x))
collect_list(flights.df1) %>% head # returns number of rows for each data.frame in a list
#> [[1]]
#> [1] 100000
#>
#> [[2]]
#> [1] 100000
#>
#> [[3]]
#> [1] 100000
#>
#> [[4]]
#> [1] 36776
```

and to do the same with `map.disk.frame`

```
map(flights.df, ~nrow(.x), lazy = F) %>% head
#> [[1]]
#> [1] 100000
#>
#> [[2]]
#> [1] 100000
#> 
#> [[3]]
#> [1] 100000
#> 
#> [[4]]
#> [1] 36776
```

The `map` function can also output the results to another disk.frame folder, e.g.

```
# return the first 10 rows of each chunk
flights.df2 <- map(flights.df, ~.x[1:10,], lazy = F, outdir = file.path(tempdir(), "tm

flights.df2 %>% head
#>    year month day dep_time sched_dep_time dep_delay arr_time
#> 1: 2013     1   1      517            515         2      830
#> 2: 2013     1   1      533            529         4      850
#> 3: 2013     1   1      542            540         2      923
#> 4: 2013     1   1      544            545        -1     1004
#> 5: 2013     1   1      554            600        -6      812
#> 6: 2013     1   1      554            558        -4      740
#>    sched_arr_time arr_delay carrier flight tailnum origin dest air_time
#> 1:            819        11      UA   1545  N14228    EWR  IAH      227
#> 2:            830        20      UA   1714  N24211    LGA  IAH      227
#> 3:            850        33      AA   1141  N619AA    JFK  MIA      160
#> 4:           1022       -18      B6    725  N804JB    JFK  BQN      183
#> 5:            837       -25      DL    461  N668DN    LGA  ATL      116
#> 6:            728        12      UA   1696  N39463    EWR  ORD      150
#>    distance hour minute            time_hour
#> 1:     1400    5     15 2013-01-01T10:00:00Z
#> 2:     1416    5     29 2013-01-01T10:00:00Z
#> 3:     1089    5     40 2013-01-01T10:00:00Z
#> 4:     1576    5     45 2013-01-01T10:00:00Z
#> 5:      762    6      0 2013-01-01T11:00:00Z
#> 6:      719    5     58 2013-01-01T10:00:00Z
```

Notice `disk.frame` supports the `purrr` syntax for defining a function using `~`.

3.10 Sampling

In the `disk.frame` framework, sampling a proportion of rows within each chunk can be performed using `sample_frac`.

```
flights.df %>% sample_frac(0.01) %>% collect %>% head
#>   year month day dep_time sched_dep_time dep_delay arr_time sched_arr_time
#> 1 2013    12  13      859            900        -1     1226           1206
#> 2 2013    10  14     1910           1859        11     2123           2126
#> 3 2013    10   1     1719           1730       -11     1828           1847
#> 4 2013    12  17     2136           2129         7       34             10
#> 5 2013    12  10      736            737        -1      936            921
#> 6 2013    12   8     1326           1329        -3     1547           1538
#>   arr_delay carrier flight tailnum origin dest air_time distance hour
#> 1        20      B6   1701  N527JB    JFK  FLL      159     1069    9
#> 2        -3      FL    645  N956AT    LGA  ATL      106      762   18
#> 3       -19      B6   1516  N183JB    JFK  SYR       48      209   17
#> 4        24      B6    527  N593JB    EWR  MCO      142      937   21
#> 5        15      B6    885  N216JB    JFK  RDU       88      427    7
#> 6         9      EV   4687  N13566    EWR  CVG      108      569   13
#>   minute            time_hour
#> 1      0 2013-12-13T14:00:00Z
#> 2     59 2013-10-14T22:00:00Z
#> 3     30 2013-10-01T21:00:00Z
#> 4     29 2013-12-18T02:00:00Z
#> 5     37 2013-12-10T12:00:00Z
#> 6     29 2013-12-08T18:00:00Z
```

3.11 Writing Data

One can output a `disk.frame` by using the `write_disk.frame` function. E.g.

```
write_disk.frame(flights.df, outdir="out")
```

this will output a disk.frame to the folder "out"

```
fs::dir_delete(file.path(tempdir(), "tmp_flights.df"))
fs::dir_delete(file.path(tempdir(), "tmp2"))
fs::file_delete(file.path(tempdir(), "tmp_flights.csv"))
```

There are a number of concepts and terminologies that are useful to understand in order to use `disk.frame` effectively.

3.12 What is a `disk.frame` and what are chunks?

A `disk.frame` is nothing more a folder and in that folder there should be `fst` files named "1.fst", "2.fst", "3.fst" etc. Each of the ".fst" file is called a *chunk*.

3.13 Workers and parallelism

Parallelism in `disk.frame` is achieved using the `future` package. When performing many tasks, `disk.frame` uses multiple workers, where each *worker* is an R session, to perform the tasks in parallel. For example, suppose we wish to compute the number of rows for each chunk, we can clearly perform this simultaneously in parallel. The code to do that is

```
# use only one column is fastest
df[,.N, keep = "first_col"]
```

Say there are `n` chunks in `df`, and there are `m` workers. Then the first `m` chunks will run `chunk[,.N]` simultaneously.

To see how many workers are at work, use

```
# see how many workers are available for work
future::nbrOfWorkers()
```

Let's set-up `disk.frame`

```
library(disk.frame)

# set up multiple
setup_disk.frame()
```

One of the most important tasks to perform before using the `disk.frame` package is to make some `disk.frame`s! There are a few functions to help you do that.

3.14 Convert a `data.frame` to `disk.frame`

Firstly there is `as.disk.frame()` which allows you to make a `disk.frame` from a `data.frame`, e.g.

```
flights.df = as.disk.frame(nycflights13::flights)
```

will convert the `nycflights13::flights` `data.frame` to a `disk.frame` somewhere in `tempdir()`. To find out the location of the `disk.frame` use:

```
attr(flights.df, "path")
```

You can also specify a location to output the `disk.frame` to using `outdir`

```
flights.df = as.disk.frame(nycflights13::flights, outdir = "some/path.df")
```

it is recommended that you use `.df` as the extension for a `disk.frame`, however this is not an enforced requirement.

However, one of the reasons for `disk.frame` to exist is to handle larger-than-RAM files, hence `as.disk.frame` is not all that useful because it can only convert data that can fit into RAM. `disk.frame` comes with a couple more ways to create `disk.frame`.

3.15 Creating `disk.frame` from CSVs

The function `csv_to_disk.frame` can convert CSV files to `disk.frame`. The most basic usage is

```
some.df = csv_to_disk.frame("some/path.csv", outdir = "some.df")
```

this will convert the CSV file `"some/path.csv"` to a `disk.frame`.

3.16 Multiple CSV files

However, sometimes we have multiple CSV files that you want to read in and row-bind into one large `disk.frame`. You can do so by supplying a vector of file paths e.g. from the result of `list.files`

```
some.df = csv_to_disk.frame(c("some/path/file1.csv", "some/path/file2.csv"))

# or
some.df = csv_to_disk.frame(list.files("some/path"))
```

3.17 Inputing CSV files chunk-wise

The `csv_to_disk.frame(path, ...)` function reads the file located at `path` in full into RAM but sometimes the CSV file may be too large to read in one go, as that would require loading the whole file into RAM. In that case, you can read the files chunk-by-chunk by using the `in_chunk_size` argument which controls how many rows you read in per chunk

```r
# to read in 1 million (=1e6) rows per chunk
csv_to_disk.frame(path, in_chunk_size = 1e6)
```

3.18 Sharding

One of the most important aspects of `disk.frame` is sharding. One can shard a `disk.frame` at read time by using the `shardby`

```r
csv_to_disk.frame(path, shardby = "id")
```

In the above case, all rows with the same `id` values will end up in the same chunk.

3.19 Just-in-time transformation

Sometimes, one may wish to perform some transformation on the CSV before writing out to disk. One can use the `inmapfn` argument to do that. The `inmapfn` name comes from INput MAPping FuNction. The general usage pattern is as follows:

```r
csv_to_disk.frame(file.path(tempdir(), "df.csv"), inmapfn = function(chunk) {
  some_transformation(chunk)
})
```

As a contrived example, suppose you wish to convert a string into date at read time:

```r
df = data.frame(date_str = c("2019-01-02", "2019-01-02"))

# write the data.frame
write.csv(df, file.path(tempdir(), "df.csv"))


# this would show that date_str is a string
str(collect(csv_to_disk.frame(file.path(tempdir(), "df.csv")))$date_str)
## chr [1:2] "2019-01-02" "2019-01-02"

# this would show that date_str is a string
df = csv_to_disk.frame(file.path(tempdir(), "df.csv"), inmapfn = function(chunk) {
  # convert to date_str to date format and store as "date"
  chunk[, date := as.Date(date_str, "%Y-%m-%d")]
  chunk[, date_str:=NULL]
})

str(collect(df)$date)
## Date[1:2], format: "2019-01-02" "2019-01-02"
```

3.20 Reading CSVs from zip files

Often, CSV comes zipped in a zip files. You can use the `zip_to_disk.frame` to convert all CSVs within a zip file

```
zip_to_disk.frame(path_to_zip_file)
```

The arguments for `zip_to_disk.frame` are the same as `csv_to_disk.frame`'s.

3.21 Using `add_chunk`

What if the method of converting to a `disk.frame` isn't implemented in `disk.frame` yet? One can use some lower level constructs provided by `disk.frame` to create `disk.frame`s. For example, the `add_chunk` function can be used to add more chunks to a `disk.frame`, e.g.

```
a.df = disk.frame() # create an empty disk.frame
add_chunk(a.df, cars) # adds cars as chunk 1
add_chunk(a.df, cars) # adds cars as chunk 2
```

Another example of using `add_chunk` is via `readr`'s chunked read functions to create a delimited file reader

```
delimited_to_disk.frame <- function(file, outdir, ...) {
  res.df = disk.frame(outdir, ...)
  readr::read_delim_chunked(file, callback = function(chunk) {
    add_chunk(res.df, chunk)
  }, ...)

  res.df
}

delimited_to_disk.frame(path, outdir = "some.df")
```

The above code uses `readr`'s `read_delim_chunked` function to read `file` and call `add_chunk`. The problem with this approach is that is it sequential in nature and hence is not able to take advantage of parallelism.

3.22 Exploiting the structure of a disk.frame

Of course, a `disk.frame` is just a folder with many `fst` files named as `1.fst`, `2.fst` etc. So one can simply create these `fst` files and ensure they have the same variable names and put them in a folder.

3.23 `disk.frame` supports `data.table` syntax

```
library(disk.frame)

# set-up disk.frame to use multiple workers
if(interactive()) {
  setup_disk.frame()
} else {
  setup_disk.frame(2)
}
#> The number of workers available for disk.frame is 2


library(nycflights13)

# create a disk.frame
flights.df = as.disk.frame(nycflights13::flights, outdir = file.path(tempdir(),"flights
```

In the following example, I will use the `.N` from the `data.table` package to count the unique combinations `year` and `month` within each chunk.

```
library(disk.frame)

flights.df = disk.frame(file.path(tempdir(),"flights13"))

names(flights.df)
#>  [1] "year"           "month"          "day"            "dep_time"
#>  [5] "sched_dep_time" "dep_delay"      "arr_time"       "sched_arr_time"
#>  [9] "arr_delay"      "carrier"        "flight"         "tailnum"
#> [13] "origin"         "dest"           "air_time"       "distance"
#> [17] "hour"           "minute"         "time_hour"

flights.df[,.N, .(year, month), keep = c("year", "month")]
#>     year month     N
#>  1: 2013     1 27004
#>  2: 2013    10 28889
#>  3: 2013    11   237
#>  4: 2013    11 27031
#>  5: 2013    12 28135
#>  6: 2013     2   964
#>  7: 2013     2 23987
#>  8: 2013     3 28834
#>  9: 2013     4  3309
#> 10: 2013     4 25021
#> 11: 2013     5 28796
#> 12: 2013     6  2313
#> 13: 2013     6 25930
#> 14: 2013     7 29425
#> 15: 2013     8   775
#> 16: 2013     8 28552
#> 17: 2013     9 27574
```

All `data.table` syntax are supported. However, `disk.frame` adds the ability to load only those columns required for the analysis using the `keep =` option. In the above analysis, only the `year` and `month` variables are required and hence `keep = c("year", "month")` was used.

Alternatively, we can use the `srckeep` function to achieve the same, e.g.

```
srckeep(flights.df, c("year", "month"))[,.N, .(year, month)]
```

3.23.1 External variables are captured

`disk.frame` sends the computaion to background workers which are essentially distinct and separate R sessions. Typically, the variables that you have available in your current R session aren't visible in the other R sessions, but `disk.frame` uses the `future` package's variable detection abilities to figure out which variables are in use and then send them to the background workers so they have access to the variables as well. E.g.

```
y = 42
some_fn <- function(x) x


flights.df[,some_fn(y)]
#> [1] 42 42 42 42 42 42
```

In the above example, neither `some_fn` nor `y` are defined in the background workers' environments, but `disk.frame` still manages to evaluate this code `flights.df[,some_fn(y)]`.