collapse: Advanced and Fast Statistical Computing and Data Transformation in R

Sebastian Krantz
Kiel Institute for the World Economy

Abstract

collapse is a large C/C++-based infrastructure package facilitating complex statistical computing, data transformation, and exploration tasks in R—at outstanding levels of performance and memory efficiency. It also implements a class-agnostic approach to R programming, supporting vector, matrix and data frame-like objects and their popular extensions (‘`units`’, ‘`integer64`’, ‘`xts`’, ‘`tibble`’, ‘`data.table`’, ‘`sf`’, ‘`pdata.frame`’), enabling its seamless integration with large parts of the R ecosystem. This article introduces the package’s key components and design principles in a structured way, supported by a rich set of examples. A small benchmark demonstrates its computational performance.

Keywords: statistical computing, vectorization, data manipulation and transformation, class-agnostic programming, summary statistics, C/C++, R.

1. Introduction

collapse is a large C/C++-based R package that provides an integrated suite of statistical and data manipulation functions.[1] Core functionality includes a rich set of S3 generic (grouped and weighted) statistical functions for vectors, matrices, and data frames, which provide efficient low-level vectorizations, OpenMP multithreading, and skip missing values by default (`na.rm = TRUE`). It also provides functions and classes for fully indexed (time-aware) computations on time series and panel data, advanced descriptive statistical tools, recursive tools to deal with nested data, and powerful data manipulation functions—such as vectorized and verbose hash-joins or fast aggregation and recast pivots. This functionality is supported by efficient algorithms for grouping, ordering, deduplication, and matching callable at R and C levels. The package also provides efficient data object conversions, functions for memory efficient R programming, such as (grouped) transformation and math by reference, and helpers to effectively deal with variable labels, attributes, and missing data. **collapse** is class-agnostic, providing statistical operations on vectors, matrices, and data frames/lists, and seamlessly supporting extensions to these objects popular in the R ecosystem—notably, ‘`units`’, ‘`integer64`’, ‘`xts`’, ‘`tibble`’, ‘`data.table`’, ‘`sf`’, and ‘`pdata.frame`’. It is globally and interactively configurable, which includes setting different defaults for key function arguments, such as `na.rm` arguments to statistical functions or `sort` arguments to grouping algorithms (default `TRUE`), and modifying the package namespace to mask equivalent but slower base R or **tidyverse** functions.[2]

[1]Website: https://sebkrantz.github.io/collapse/. Linecount (v2.0.19): R: 13,646, C: 18,594, C++: 9,844. Exported namespace: 391 objects, of which 237 functions (excl. methods and shorthands), and 2 datasets.

[2]**collapse**’s namespace is fully compatible with base R and the **tidyverse** (Wickham *et al.* 2019), but can be interactively modified to mask/overwrite key functions with the much faster **collapse** equivalents. See Section 8.

Why combine all of these features in a package? The short answer is to make computations in R as flexible and powerful as possible. The more elaborate answer is to (1) facilitate complex data transformation, exploration, and computing tasks in R; (2) increase the performance and memory efficiency of R programs;[3] and (3) to create a new foundation package for statistics and data transformation in R that implements many successful ideas developed in the R ecosystem and other programming environments such as Python or STATA (StataCorp LLC. 2023), in a stable, high performance, and broadly compatible manner.[4]

R already has a large and tested data manipulation and statistical computing ecosystem. Notably, the **tidyverse** (Wickham *et al.* 2019) provides a consistent toolkit for data manipulation in R, centered around the '`tibble`' (Müller and Wickham 2023) object and tidy data principles (Wickham 2014). **data.table** (Dowle and Srinivasan 2023) provides an enhanced high-performance data frame with parsimonious data manipulation syntax. **sf** (Pebesma 2018) provides a data frame for spatial data and supporting functionality. **tsibble** (Wang *et al.* 2020) and **xts** (Ryan and Ulrich 2023) provide classes and operations for time series data, the former via an enhanced '`tibble`', the latter through an efficient matrix-based class. Econometric packages like **plm** (Croissant and Millo 2008) and **fixest** (Bergé 2018) also provide solutions to deal with panel data and irregularity in the time dimension. Packages like **matrixStats** (Bengtsson 2023) and **Rfast** (Papadakis *et al.* 2023) offer fast statistical calculations along the rows and columns of matrices as well as faster basic statistical procedures. **DescTools** (Signorell 2023) provides a wide variety of descriptive statistics, including weighted versions. **survey** (Lumley 2004) allows statistical computations on complex survey data. **labelled** (Larmarange 2023) provides tools to deal with labelled data. Packages like **tidyr** (Wickham *et al.* 2023b), **purrr** (Wickham and Henry 2023) and **rrapply** (Chau 2022) provide some functions to deal with nested data and messy structures.

collapse relates to and integrates key elements from these projects. It offers **tidyverse**-like data manipulation at the speed and stability of **data.table** for any data frame-like object. It can turn any vector/matrix/data frame into a time-aware indexed series or frame and perform operations such as lagging, differencing, scaling or centering, encompassing and enhancing core manipulation functionality of **plm**, **fixest**, and **xts**. It also performs fast (grouped, weighted) statistical computations along the columns of matrix-like objects, complementing and enhancing **matrixStats** and **Rfast**. Its low-level vectorizations and workhorse algorithms are accessible at the R and C-levels, unlike **data.table**, where most vectorizations and algorithms are internal. It also supports variable labels and intelligently preserves attributes of all objects, complementing **labelled**. It provides novel recursive tools to deal with nested data, enhancing **tidyr**, **purrr**, and **rrapply**. Finally, it provides a small but consistent and powerful set of descriptive statistical tools, yielding sufficient detail for most data exploration purposes, requiring users to invoke packages like **DescTools** or **survey** only for specific statistics.

[3]Principally by avoiding R-level repetition such as applying R functions across columns/groups using a split-apply-combine logic, but also by avoiding object conversions and the need for certain classes to do certain things, such as converting matrices to '`data.frame`' or '`data.table`' just to compute statistics by groups.

[4]Such ideas include **tidyverse** syntax, vectorized aggregations (**data.table**), data transformation by reference (Python, **pandas**), vectorized and verbose joins (**polars**, STATA), indexed time series and panel data (**xts**, **plm**), summary statistics for panel data (STATA), variable labels (STATA), recast pivots (**reshape(2)**), etc...

In summary, **collapse** is a foundation package for statistical computing and data transformation in R that enhances and integrates seamlessly with the R ecosystem while being outstandingly computationally efficient. A significant benefit is that, rather than piecing together a fragmented ecosystem oriented at different classes and tasks, many core computational tasks can be done with **collapse**, and easily extended by more specialized packages. This tends to result in R scripts that are shorter, more efficient, and more lightweight in dependencies.

Other programming environments such as Python and Julia now also offer computationally very powerful libraries for tabular data such as **DataFrames.jl** (Bouchet-Valat and Kamiński 2023), **Polars** (Vink *et al.* 2023), and **Pandas** (Wes McKinney 2010; **pandas** Development Team 2023), and supporting numerical libraries such as **Numpy** (Harris *et al.* 2020), or **StatsBase.jl** (JuliaStats 2023). In comparison with these, **collapse** offers a class-agnostic approach bridging the divide between data frames and atomic structures, has more advanced statistical capabilities,[5] supports recast pivots and recursive operations on lists, variable labels, verbosity for critical operations such as joins, and is extensively globally configurable. In short, it is very utile for complex statistical workflows, rich datasets (e.g., surveys), and for integrating with different parts of the R ecosystem. On the other hand, **collapse**, for the most part, does not offer a sub-column-level parallel architecture and is thus not highly competitive with top frameworks, including **data.table**, on aggregating billion-row datasets with few columns.[6] Its vectorization capabilities are also limited to the statistical functions it provides and not, like **DataFrames.jl**, to any Julia function. However, as demonstrated in Section 3.1, vectorized statistical functions can be combined to calculate more complex statistics in a vectorized way.

The package has a built-in structured documentation facilitating its use. This documentation includes a central overview page linking to all other documentation pages and supplementary topic pages which briefly describe related functionality. The names of these extra pages are collected in a global macro `.COLLAPSE_TOPICS` and can be called directly with `help()`:

```
R> .COLLAPSE_TOPICS
```

```
 [1] "collapse-documentation"     "fast-statistical-functions"
 [3] "fast-grouping-ordering"     "fast-data-manipulation"
 [5] "quick-conversion"           "advanced-aggregation"
 [7] "data-transformations"       "time-series-panel-series"
 [9] "list-processing"            "summary-statistics"
[11] "recode-replace"             "efficient-programming"
[13] "small-helpers"              "collapse-options"
```

```
R> help("collapse-documentation")
```

[5]Such as weighted statistics, including various quantile and mode estimators, support for fully time-aware computations on irregular series/panels, higher order centering, advanced (grouped, weighted, panel-decomposed) descriptive statistics etc., all supporting missing values.

[6]As can be seen in the DuckDB Benchmarks: **collapse** is highly competitive on the 10-100 million observations datasets, but deteriorates in performance at larger data sizes. There may be performance improvements for "long data" in the future, but, at present, the treatment of columns as fundamental units of computation (in most cases) is a tradeoff for the highly flexible class-agnostic architecture.

This article does not fully present **collapse**, but the following sections introduce its key components, starting with (2) the *Fast Statistical Functions* and their (3) integration with data manipulation functions; (4) architecture for time series and panel data; (5) table joins and pivots; (6) list processing functions; (7) descriptive tools; and (8) global options. Section 9 provides a small benchmark, Section 10 concludes. For deeper engagement with **collapse**, consult the documentation and resources (vignettes/cheatsheet/blog/slides/talk).

2. Fast statistical functions

The *Fast Statistical Functions*, comprising `fsum()`, `fprod()`, `fmean()`, `fmedian()`, `fmode()`, `fvar()`, `fsd()`, `fmin()`, `fmax()`, `fnth()`, `ffirst()`, `flast()`, `fnobs()`, and `fndistinct()`, are a consistent set of S3-generic statistical functions providing fully vectorized statistical operations in R.[7] Specifically, operations are vectorized across columns and groups, and may also involve weights or transformations of the input data. The functions basic syntax is

```
FUN(x, g = NULL, w = NULL, TRA = NULL, na.rm = TRUE, use.g.names = TRUE, ...)
```

with arguments `x` - data (vector, matrix or data frame-like), `g` - groups (atomic vector, list of vectors, or '`GRP`' object), `w` - sampling weights (only some functions), and `TRA` - transformation of `x`. The following examples using `fmean()` demonstrate their basic usage on the familiar `iris` dataset, recording 50 measurements of 4 variables for 3 species of iris flowers. All examples support weights (`w`), and `fmean()` can also be multithreaded across columns (`nthreads`).[8]

```
R> fmean(iris$Sepal.Length)

[1] 5.843

R> fmean(iris[1:4])

Sepal.Length  Sepal.Width Petal.Length  Petal.Width
       5.843        3.057        3.758        1.199

R> identical(fmean(iris[1:4]), fmean(as.matrix(iris[1:4])))

[1] TRUE

R> fmean(iris$Sepal.Length, g = iris$Species)

    setosa versicolor  virginica
     5.006      5.936      6.588

R> fmean(iris[1:4], g = iris$Species, nthreads = 4)
```

[7]'Vectorization' in R means that these operations are implemented using compiled C/C++ code.

[8]Not all functions are multithreaded, and parallelism is implemented differently for different functions, as detailed in the respective function documentation. The default use of single instruction multiple data (SIMD) parallelism also implies limited gains from multithreading for simple (non-grouped) operations.

```
           Sepal.Length Sepal.Width Petal.Length Petal.Width
setosa            5.006       3.428        1.462       0.246
versicolor        5.936       2.770        4.260       1.326
virginica         6.588       2.974        5.552       2.026
```

```
R> fmean(iris$Sepal.Length, g = iris$Species, TRA = "fill")[1:10]
```

```
 [1] 5.006 5.006 5.006 5.006 5.006 5.006 5.006 5.006 5.006 5.006
```

2.1. Transformations

The final example expands the mean vector to full length—like `ave(x, g)` but much faster. The `TRA` argument toggles (grouped) replacing and sweeping operations (by reference), generalizing `sweep(x, 2, STATS = fmean(x))`.[9] Table 1 lists the 11 possible `TRA` operations.

String	*Description*
`"replace_na"/"na"`	replace missing values in `x` by `STATS`
`"replace_fill"/"fill"`	replace data and missing values in `x` by `STATS`
`"replace"`	replace data by `STATS` but preserve missing values in `x`
`"-"`	subtract `STATS` (center)
`"-+"`	subtract `STATS` and add overall average statistic
`"/"`	divide by `STATS` (scale)
`"%"`	compute percentages (divide and multiply by 100)
`"+"`	add `STATS`
`"*"`	multiply by `STATS`
`"%%"`	modulus (remainder from division by `STATS`)
`"-%%"`	subtract modulus (make data divisible by `STATS`)

Table 1: Available `TRA` argument choices.

Additionally, a `set` argument can be passed to *Fast Statistical Functions* to toggle transformation by reference, e.g., `fmean(iris$Sepal.Length, g = iris$Species, TRA = "fill", set = TRUE)` would modify `Sepal.Length` in-place and return the result invisibly.

Having grouping and data transformation functionality directly built into generic statistical functions facilitates and speeds up many common operations. Take for example this generated sector-level trade dataset of export values (v) by country (c), sector (s), and year (y).[10]

```
R> set.seed(101)
R> exports <- expand.grid(y=1:10, c=paste0("c",1:10), s=paste0("s",1:10)) |>
+    tfm(v = abs(rnorm(1e3))) |> colorder(c, s) |> ss(-sample.int(1e3, 500))
```

Like any real trade dataset, it is unbalanced—`ss(-sample.int(1e3, 500))` randomly removes 500 rows. Suppose we wanted to extract the latest trade within the last two years.

[9]The `TRA` argument internally calls `TRA()`: `TRA(x, STATS, FUN = "-", g = NULL, set = FALSE, ...)`.

[10]`tfm()` abbreviates `ftransform()`—a faster equivalent of `transform()`. `ss()` replaces `[.data.frame`.

```
R> latest <- fsubset(exports, y >= 8 & y == fmax(y, list(c, s), "fill"), -y)
```

Below, I compute how many products different countries have exported in the last two years

```
R> with(latest, fndistinct(s, c))

 c1  c2  c3  c4  c5  c6  c7  c8  c9 c10 
  9   9   9  10   9   9   8  10   9  10 
```

and this computes Balassa (1965)'s index of Revealed Comparative Advantage, defined as the share of a sector in country exports divided by the share of the sector in world exports.

```
R> with(latest, fsum(v, c, TRA = "/") / fsum(v, s, TRA = "/"))[1:10]

 [1] 0.6674 0.5673 0.6403 0.8162 0.7989 0.9122 0.8561 0.5611 0.7847 1.1557
```

More complex use cases are frequent in my work. For example, I recently combined multiple spatial datasets on points of interest (POIs). In the face of significant duplicates and problems matching POIs directly across datasets, I decided to keep the richest source for each location and POI type. After creating POI confidence, location, and type indicators comparable across sources (datasets), my deduplication expression was `fsubset(data_long, source == fmode(source, list(location, type), confidence, "fill"))`—which retains POIs from the confidence-weighted most frequent (i.e., richest) source by location and type.

2.2. Grouping objects and optimization

Whereas the `g` argument supports ad-hoc grouping with vectors and lists/data frames, for repeated operations the cost of grouping can be minimized by using factors (see `?qF`) or '`GRP`' objects as inputs. The latter contain all information **collapse**'s statistical functions may require to operate across groups. They can be created with `GRP()`. Its basic syntax is

```
GRP(X, by = NULL, sort = TRUE, return.groups = TRUE, method = "auto", ...)
```

Below, I create a '`GRP`' object from the included World Development Dataset (`wlddev`). The `by` argument also supports column names/indices, and `X` could also be an atomic vector.

```
R> str(g <- GRP(wlddev, ~ income + OECD))

Class 'GRP'  hidden list of 9
 $ N.groups   : int 6
 $ group.id   : int [1:13176] 3 3 3 3 3 3 3 3 3 3 ...
 $ group.sizes : int [1:6] 2745 2074 1830 2867 3538 122
 $ groups     :'data.frame':	6 obs. of  2 variables:
  ..$ income: Factor w/ 4 levels "High income",..: 1 1 2 3 4 4
  .. ..- attr(*, "label")= chr "Income Level"
  ..$ OECD  : logi [1:6] FALSE TRUE FALSE FALSE FALSE TRUE
  .. ..- attr(*, "label")= chr "Is OECD Member Country?"
 $ group.vars  : chr [1:2] "income" "OECD"
 $ ordered     : Named logi [1:2] TRUE FALSE
  ..- attr(*, "names")= chr [1:2] "ordered" "sorted"
 $ order       : int [1:13176] 245 246 247 248 249 250 251 252 253 254 ...
  ..- attr(*, "starts")= int [1:6] 1 2746 4820 6650 9517 13055
  ..- attr(*, "maxgrpn")= int 3538
  ..- attr(*, "sorted")= logi FALSE
 $ group.starts: int [1:6] 245 611 1 306 62 7687
 $ call        : language GRP.default(X = wlddev, by = ~income + OECD)
```

'GRP' objects make grouped statistical computations in **collapse** fully programmable, e.g., I can employ the object with the *Fast Statistical Functions* and some utilities[11] to efficiently aggregate GDP per capita, life expectancy, and country name, with population weights.

```
R> add_vars(g$groups,
+   get_vars(wlddev, "country") |> fmode(g, wlddev$POP, use = FALSE),
+   get_vars(wlddev, c("PCGDP", "LIFEEX")) |> fmean(g, wlddev$POP, use = F))

               income  OECD       country   PCGDP LIFEEX
1         High income FALSE  Saudi Arabia 22426.7  73.00
2         High income  TRUE United States 31749.6  75.84
3          Low income FALSE      Ethiopia   557.1  53.51
4 Lower middle income FALSE         India  1238.8  60.59
5 Upper middle income FALSE         China  3820.6  68.21
6 Upper middle income  TRUE        Mexico  8311.2  69.06
```

For advanced data aggregation, **collapse** also provides a convenience function, `collap()`, which, by default, uses `fmean()` for numeric, `fmode()` for categorical, and `fsum()` for weight columns, and preserves their order. The equivalent expression using this function would be

```
R> collap(wlddev, country + PCGDP + LIFEEX ~ income + OECD, w = ~ POP)

        country              income  OECD   PCGDP LIFEEX       POP
1  Saudi Arabia         High income FALSE 22426.7  73.00 3.114e+09
2 United States         High income  TRUE 31749.6  75.84 5.573e+10
3      Ethiopia          Low income FALSE   557.1  53.51 2.095e+10
4         India Lower middle income FALSE  1238.8  60.59 1.138e+11
5         China Upper middle income FALSE  3820.6  68.21 1.114e+11
6        Mexico Upper middle income  TRUE  8311.2  69.06 8.162e+09
```

Similarly, data can be transformed, here using `fmean()` to center the data by country to level differences in average economic status, adding back the overall mean across countries.[12]

```
R> add_vars(wlddev) <- get_vars(wlddev, c("PCGDP", "LIFEEX")) |>
+      fmean(wlddev$iso3c, TRA = "-+") |> add_stub("center_")
```

[11] `add_vars()` is a fast `cbind.data.frame()` which also has an assignment method, and `get_vars()` enables fast and secure extraction of data frame columns.

[12] `add_stub()` adds a prefix (or suffix if `pre = FALSE`) to columns (→ `center_PCGDP` and `center_LIFEEX`).

For (higher-dimensional) centering, **collapse** also has specialized function(s) `f[hd]within()` with additional options, and `fscale()` supports various scaling and centering operations.

Exempting `collap()`, these examples may seem bulky for quick analysis, but a robust low-level API is very useful for package development, as further elucidated in the vignette on developing with **collapse**. I also wrote a blog post on aggregating survey data using **collapse**, which showcases more aspects of the `collap()` function using real census data. Grouped programming using 'GRP' objects and *Fast Statistical Functions* is also particularly powerful with vectors and matrices. For example, in the useR 2022 presentation I aggregate global input-output tables stored as matrices (`x`) from the country to the region level using a single grouping object and expressions of the form `x |> fsum(g) |> t() |> fsum(g) |> t()`.[13]

3. Integration with data manipulation functions

collapse also provides a broad set of fast data manipulation functions akin to base R and **tidyverse** functions, including `fselect()`, `fsubset()`, `fgroup_by()`, `fsummarise()`, `ftransform()`, `fmutate()`, `across()`, `frename()`, `fcount()`, etc. These are integrated with the *Fast Statistical Functions* to enable vectorized statistical operations in a familiar data frame oriented and **tidyverse**-like workflow. For example, the following code aggregates the `wlddev` data by income group for years post 2015 (to smooth volatility), with population weights.

```
R> wlddev |> fsubset(year >= 2015) |> fgroup_by(income) |>
+    fsummarise(country = fmode(country, POP),
+               across(c(PCGDP, LIFEEX, GINI), fmean, POP))

               income       country PCGDP LIFEEX  GINI
1         High income United States 43340  80.70 36.14
2          Low income      Ethiopia   663  63.05 39.13
3 Lower middle income         India  2177  68.31 36.48
4 Upper middle income         China  8168  75.51 41.68
```

This code is very fast because data does not need to be split by groups. Under the hood it is principally a syntax translation to the low-level API introduced above.[14] *Fast Statistical Functions* also have a method for grouped data, so `fsummarise()` is not always needed.

```
R> wlddev |> fsubset(year >= 2015, income, PCGDP:GINI, POP) |>
+    fgroup_by(income) |> fmean(POP, keep.w = FALSE)

               income PCGDP LIFEEX  GINI
1         High income 43340  80.70 36.14
2          Low income   663  63.05 39.13
3 Lower middle income  2177  68.31 36.48
4 Upper middle income  8168  75.51 41.68
```

[13]A recent application with vectors involved numerically optimizing a parameter a in an equation of the form $\sum_i x_{ij}^a \; \forall j \in J$ so as to minimize the deviation from a target y_j where there are J groups (1 million in my case) - see the first example in this blog post for an illustration.

[14]`fgroup_by()` creates a 'GRP' object from the `income` column, attaches it as an attribute, and `fsummarise()`/`across()` fetches it and passes it to the `g` arguments of the *Fast Statistical Functions* set as a keyword argument (and sets `use.g.names = FALSE`). Thus, `w` becomes the second positional argument. Since `fmean()` is S3 generic, `across()` directly invokes `fmean.data.frame()` on the subset of columns.

3.1. Vectorizations for advanced tasks

`fsummarise()` and `fmutate()` can also evaluate arbitrary functions in the classical (split-apply-combine) way and handle more complex expressions involving multiple columns and/or functions. However, using any *Fast Statistical Function* causes the whole expression to be vectorized, i.e., evaluated only once and not for every group. This eager vectorization approach enables efficient grouped calculation of more complex statistics. The example below forecasts the exports for each country-sector via linear regression (`v ~ y`) in a vectorized way.

```
R> exports |> fgroup_by(c, s) |> fmutate(dmy = fwithin(y)) |>
+    fsummarise(v_10 = flast(v), beta = fsum(v, dmy) %/=% fsum(dmy, dmy)) |>
+    fmutate(v_11 = v_10 + beta, v_12 = v_11 + beta, beta = NULL) |> head(4)

   c  s   v_10   v_11     v_12
1 c1 s1 0.2233 0.2152  0.20707
2 c1 s2 1.6860 1.7235  1.76112
3 c1 s3 0.2971 0.1364 -0.02433
4 c1 s4 1.0617 1.3544  1.64707
```

The expression `fsum(v, dmy) %/=% fsum(dmy, dmy)` amounts to `cov(v, y)/var(y)`, but is both vectorized across groups and memory efficient—leveraging the weights (`w`) argument to `fsum()` to compute products (`v * dmy` and `dmy * dmy`) on the fly and division by reference (`%/=%`) to avoid an additional allocation for the final result. I do not recommend forecasting trade in this way, rather, this example is inspired by a 2023 bog post where I forecasted high-resolution (1km2) population estimates for South Africa. The data, taken from WorldPop, was available for the years 2014-2020, and I needed estimates for 2021-2022. Linear regression was sensible, and using the above expression I was able to run 1.6 million regressions and obtain 2 forecasts in 0.26 seconds on a laptop. Another neat example from the community, shared by Andrew Ghazi in a blog post, vectorizes an expression to compute the p value, `2 * pt(abs(fmean(x) * sqrt(6) / fsd(x)), 5, lower.tail = FALSE)`, across 300k groups for a simulation study, yielding a 70x performance increase over **dplyr**. The eager vectorization approach of **collapse** here replaces `fmean(x)` and `fsd(x)` by their grouped versions and evaluates the entire expression once rather than 300k times as in **dplyr**.

collapse also vectorizes advanced statistics. The following calculates a weighted set of summary statistics by groups, with weighted quantiles type 8 following Hyndman and Fan (1996).[15]

```
R> wlddev |> fsubset(is.finite(POP)) |> fgroup_by(income, OECD) |>
+    fmutate(o = radixorder(GRPid(), LIFEEX)) |>
+    fsummarise(min = fmin(LIFEEX),
+               Q1 = fnth(LIFEEX, 0.25, POP, o = o, ties = "q8"),
+               mean = fmean(LIFEEX, POP),
+               median = fmedian(LIFEEX, POP, o = o, ties = "q8"),
+               Q3 = fnth(LIFEEX, 0.75, POP, o = o, ties = "q8"),
+               max = fmax(LIFEEX))
```

[15]**collapse** calculates weighted quantiles in a theoretically consistent way by applying the probability measure to the sum of weights to create a target sum and cumulatively summing weights to find order statistics which are then combined following Hyndman and Fan (1996). See fquantile for details.

```
                income  OECD   min    Q1  mean median    Q3   max
1          High income FALSE 42.67 70.58 73.00  73.80 76.52 85.42
2          High income  TRUE 55.42 72.61 75.84  76.24 78.81 84.36
3           Low income FALSE 26.17 46.95 53.51  53.01 60.37 74.43
4 Lower middle income FALSE 18.91 54.69 60.59  62.31 67.55 76.70
5 Upper middle income FALSE 36.53 65.86 68.21  69.50 73.27 80.28
6 Upper middle income  TRUE 45.37 64.56 69.06  71.96 74.96 77.69
```

Weighted quantiles have a sub-column parallel implementation,[16] and, as shown above, can also harness an (optional) optimization utilizing an overall ordering vector (combining groups and the data column) to avoid repeated partial sorting of the same elements within groups.

4. Time series and panel data

collapse also provides a flexible high-performance architecture to perform (time aware) computations on time series and panel series. In particular, the user can either apply time series and panel data transformations without any classes by passing individual and/or time identifiers to the respective functions in an ad-hoc fashion, or use '`indexed_frame`' and '`indexes_series`' classes, which implement full and deep indexation for worry-free application in many contexts. Table 2 compactly summarizes **collapse**'s architecture for time series and panel data.

Classes, constructors and utilities
`findex_by()`, `findex()`, `unindex()`, `reindex()`, `timeid()`, `is_irregular()`,
`to_plm()` + S3 methods for '`indexed_frame`', '`indexed_series`' and '`index_df`'

Core time-based functions
`flag()`, `fdiff()`, `fgrowth()`, `fcumsum()`, `psmat()`
`psacf()`, `pspacf()`, `psccf()`

Data transformation functions with supporting methods
`fscale()`, `f[hd]between()`, `f[hd]within()`

Data manipulation functions with supporting methods
`fsubset()`, `funique()`, `roworder[v]()` (internal), `na_omit()` (internal)

Summary functions with supporting methods
`varying()`, `qsu()`

Table 2: Time series and panel data architecture.

4.1. Ad-hoc computations

Time series functions such as `fgrowth()` (to compute growth rates) are S3 generic and can be applied to most time series classes. In addition to a `g` argument for grouped computation,

[16]Use `set_collapse(nthreads = #)` or the `nthreads` arguments to `fnth()`/`fmedian()`/`fmode()` (default 1).

these functions also have a `t` argument for indexation. If `t` is a plain numeric vector or a factor, it is coerced to integer and interpreted as time steps.[17] But first, a basic example:

```
R> fgrowth(airmiles) |> round(2)

Time Series:
Start = 1937
End = 1960
Frequency = 1
 [1]    NA 16.50 42.29 54.03 31.65  2.38 15.23 33.29 54.36 76.92  2.71 -2.10
[13] 12.91 18.51 32.03 18.57 17.82 13.61 18.19 12.83 13.32  0.01 15.49  4.25
```

The following extracts one sector from the `exports` dataset generated above, creating an irregular time series missing the 3rd and 6th period.[18] Indexation using the `t` argument allows for correct (time-aware) computations on this context without 'expanding' the data.

```
R> .c(y, v) %=% fsubset(exports, c == "c1" & s == "s7", -c, -s)
R> print(y)

[1]  1  2  4  5  7  8  9 10

R> fgrowth(v, t = y) |> round(2)

[1]     NA 175.52     NA -22.37     NA 624.27 -79.01 534.56

R> fgrowth(v, -1:3, t = y) |> head(4)

        FG1     --     G1   L2G1   L3G1
[1,] -63.71 0.3893     NA     NA     NA
[2,]     NA 1.0726 175.52     NA     NA
[3,]  28.82 0.8450     NA -21.22 117.05
[4,]     NA 0.6559 -22.37     NA -38.85
```

Functions `flag()/fdiff()/fgrowth()` also have shorthands `L()/D()/G()` which both facilitate their use inside formulas and provide an enhanced data frame interface for convenient ad-hoc computations. With panel data, `t` can be omitted, but this requires sorted data with consecutive groups.[19] Below, I demonstrate two ways to compute a sequence of lagged growth rates using both the `G()` operator and the `tfm()` function - a shorthand for `ftransform()`.[20]

[17]This is premised on the observation that the most common form of temporal identifier is a numeric variable denoting calendar years. Users need to manually call `timeid()` on plain numeric vectors with decimals to yield an appropriate integer representation. If `t` is a numeric time object (e.g., '`Date`', '`POSIXct`', etc.), then it is internally passed through `timeid()` which computes the greatest common divisor (GCD) and generates an integer time-id. For the GCD approach to work, `t` must have an appropriate class, e.g., for monthly/quarterly data, `zoo::yearmon()/zoo::yearqtr()` should be used instead of '`Date`' or '`POSIXct`'.

[18]`%=%` is an infix operator for the `massign()` function in **collapse** which is a multivariate version of `assign()`.

[19]This is because a group-lag is computed in a single pass, requiring all group elements to be consecutive.

[20]See also Footnote 10. A number of key functions in **collapse** have syntactic shorthands. The `list(v = v)` is needed here to prevent `fgrowth()` from creating a matrix with the growth rates—the '`list`' method applies.

```
R> G(exports, -1:2, by = v ~ c + s, t = ~ y) |> head(3)

   c  s y  FG1.v      v   G1.v L2G1.v
1 c1 s1 2 -18.15 0.5525     NA     NA
2 c1 s1 3 214.87 0.6749  22.17     NA
3 c1 s1 4 -31.02 0.2144 -68.24  -61.2

R> tfm(exports, fgrowth(list(v = v), -1:2, g = list(c, s), t = y)) |> head(3)

   c  s y      v  FG1.v   G1.v L2G1.v
1 c1 s1 2 0.5525 -18.15     NA     NA
2 c1 s1 3 0.6749 214.87  22.17     NA
3 c1 s1 4 0.2144 -31.02 -68.24  -61.2
```

These functions and operators are also integrated with `fgroup_by()` and `fmutate()` for vectorized computations. However, using ad-hoc grouping is always more efficient.

```
R> A <- exports |> fgroup_by(c, s) |> fmutate(gv = G(v, t = y)) |> fungroup()
R> head(B <- exports |> fmutate(gv = G(v, g = list(c, s), t = y)), 4)

   c  s y      v     gv
1 c1 s1 2 0.5525     NA
2 c1 s1 3 0.6749  22.17
3 c1 s1 4 0.2144 -68.24
4 c1 s1 5 0.3108  44.98

R> identical(A, B)

[1] TRUE
```

4.2. Indexed series and frames

For more complex use cases, indexation is convenient. **collapse** supports **plm**'s '`pseries`' and '`pdata.frame`' classes through dedicated methods. Flexibility and performance considerations lead to the creation of new classes '`indexes_series`' and '`indexed_frame`' which inherit from the former. Any data frame-like object can become an '`indexed_frame`' and function as usual for other operations. The technical implementation of these classes is described in the vignette on object handling and, in more detail, in the documentation. Their basic syntax is:

```
data_ix <- findex_by(data, id1, ..., time)
data_ix$indexed_series; with(data, indexed_series)
index_df <- findex(data_ix)
```

Data can be indexed using one or more indexing variables. Unlike '`pdata.frame`', an '`indexed_frame`' is a deeply indexed structure—every series inside the frame is already an '`indexes_series`'. A comprehensive set of methods for subsetting and manipulation, and applicable 'pseries' and 'pdata.frame' methods for time series and transformation functions like `flag()/L()`, ensure that these objects behave in a time-/panel-aware manner in any caller environment (`with()`, `lm()`, etc.). Indexation can be undone using `unindex()` and redone with `reindex()` and a suitable 'index_df'. 'indexes_series' can be atomic vectors or matrices (including objects such as 'ts' or 'xts') and can be created directly using `reindex()`.

```
data <- unindex(data_ix)
data_ix <- reindex(data, index = index_df)
indexed_series <- reindex(vec/mat, index = vec/index_df)
```

An example using the `exports` data follows:

```
R> exportsi <- exports |> findex_by(c, s, y)
R> exportsi |> G() |> print(max = 15)

   c  s y   G1.v
1 c1 s1 2     NA
2 c1 s1 3  22.17
3 c1 s1 4 -68.24
 [ reached 'max' / getOption("max.print") -- omitted 497 rows ]

Indexed by:  c.s [100] | y [10]

R> exportsi |> findex() |> print(2)

    c.s y
1 c1.s1 2
2 c1.s1 3
---
499 c10.s10 7
500 c10.s10 9

c.s [100] | y [10]
```

The index statistics are: `[N. ids] | [N. periods (total periods:  (max-min)/GCD)]`.

```
R> vi <- exportsi$v; str(vi, width = 70, strict = "cut")

 'indexed_series' num [1:500] 0.552 0.675 0.214 0.311 1.174 ...
 - attr(*, "index_df")=Classes 'index_df', 'pindex' and 'data.frame'..
  ..$ c.s: Factor w/ 100 levels "c1.s1","c2.s1",..: 1 1 1 1 1 1 1 1 ..
  ..$ y  : Ord.factor w/ 10 levels "1"<"2"<"3"<"4"<..: 2 3 4 5 6 7 8..
  ..- attr(*, "nam")= chr [1:3] "c" "s" "y"

R> is_irregular(vi)

[1] TRUE
```

```
R> vi |> psmat() |> head(3)

       1     2     3     4     5    6     7      8     9    10
c1.s1 NA 0.552 0.675 0.214 0.311 1.17 0.619 0.1127 0.917 0.223
c2.s1 NA 0.795    NA    NA 0.237   NA    NA 0.0585 0.818    NA
c3.s1 NA 0.709 0.268 1.464    NA   NA 0.467 0.1193 0.467    NA

R> fdiff(vi) |> psmat() |> head(3)

       1  2      3      4      5     6      7      8     9     10
c1.s1 NA NA  0.122 -0.461 0.0964 0.863 -0.555 -0.506 0.804 -0.694
c2.s1 NA NA     NA     NA     NA    NA     NA     NA 0.759     NA
c3.s1 NA NA -0.441  1.196     NA    NA     NA -0.348 0.348     NA
```

`psmat()`, for panel-series to matrix, generates a matrix/array from panel data. Thanks to deep indexation, indexed computations work in arbitrary data masking environments.

```
R> settransform(exportsi, v_ld = Dlog(v))
R> lm(v_ld ~ L(v_ld, 1:2), exportsi) |> summary() |> coef() |> round(3)

               Estimate Std. Error t value Pr(>|t|)
(Intercept)      -0.008      0.141  -0.058    0.954
L(v_ld, 1:2)L1   -0.349      0.115  -3.042    0.004
L(v_ld, 1:2)L2   -0.033      0.154  -0.215    0.831
```

It is worth highlighting that the flexibility of this architecture is new to the R ecosystem: A ‘`pdata.frame`’ or ‘`fixest_panel`’ only works inside **plm**/**fixest** estimation functions.[21] Time series classes like ‘`xts`’ and ‘`tsibble`’ also do not provide deeply indexed structures or native handling of irregularity in basic operations. ‘`indexed_series`’ and ‘`indexed_frame`’, on the other hand, work ’anywhere’, and can be superimposed on any suitable object, as long as **collapse**’s functions (`flag()`/`L()` etc.) are used to perform time-based computations.

Indexed series/frames also support transformation such as grouped scaling with `fscale()` or demeaning with `fwithin()`. Functions `psacf()`/`pspacf()`/`psccf()` provide panel-data autocorrelation functions, which are computed using group-scaled and suitably lagged panel-series. The ‘`index_df`’ attached to these objects can also be used with other general tools such as `collapse::BY()` to perform grouped computations using 3rd-party functions. An example of calculating a 5-year rolling average is given below (`ix()` abbreviates `findex()`).

```
R> BY(vi, ix(vi)$c.s, data.table::frollmean, 5) |> head(10)

 [1]     NA     NA     NA     NA 0.5853 0.5986 0.4861 0.6267 0.6092     NA

Indexed by:  c.s [2] | y [9 (10)]
```

[21] And, in the case of **fixest**, inside **data.table** due to dedicated methods.

5. Table joins and pivots

Among all data manipulation functions **collapse** provides, its implementations of table joins and pivots are particularly noteworthy since they offer several new features, including rich verbosity for table joins, pivots supporting variable labels, and 'recast' pivots. Both implementations provide outstanding computational performance, syntax, and memory efficiency.

5.1. Joins

Compared to commercial software such as STATA, the implementation of joins in most open-source software, including R, provides no information on how many records were joined from both tables. This often provokes manual efforts to validate the join operation. `collapse::join` provides a rich set of options to understand table join operations. Its syntax is:

```
join(x, y, on = NULL, how = "left", suffix = NULL, validate = "m:m",
  multiple = FALSE, sort = FALSE, keep.col.order = TRUE,
  drop.dup.cols = FALSE, verbose = 1, column = NULL, attr = NULL, ...)
```

It defaults to left join and only takes first matches from `y` (`multiple = FALSE`), i.e., it simply adds columns to `x`, which is efficient and sufficient/desired in many cases. By default (`verbose = 1`), it prints information about the join operation and number of records joined. To demonstrate `join()`, I generate a small database for a bachelor in economics curriculum. It has a `teacher` table of 4 teachers (`id`: PK) and a linked (`id`: FK) `course` table of 5 courses.

```
R> teacher <- data.frame(id = 1:4, names = c("John", "Jane", "Bob", "Carl"),
+    age = c(35, 32, 42, 67), subject = c("Math", "Econ", "Stats", "Trade"))
R> course <- data.frame(id = c(1, 2, 2, 3, 5), semester = c(1, 1, 2, 1, 2),
+    course = c("Math I", "Microecon", "Macroecon", "Stats I", "History"))
R> join(teacher, course, on = "id")

left join: teacher[id] 3/4 (75%) <1:1st> course[id] 3/5 (60%)
  id names age subject semester    course
1  1  John  35    Math        1    Math I
2  2  Jane  32    Econ        1 Microecon
3  3   Bob  42   Stats        1   Stats I
4  4  Carl  67   Trade       NA      <NA>
```

Users can request the generation of a `.join` column (`column = "name"/TRUE`), akin to STATA's `_merge` column, to indicate the origin of records in the joined table—useful on a full join.

```
R> join(teacher, course, how = "full", multiple = TRUE, column = TRUE)

full join: teacher[id] 3/4 (75%) <1:1.33> course[id] 4/5 (80%)
  id names age subject semester    course   .join
1  1  John  35    Math        1    Math I matched
2  2  Jane  32    Econ        1 Microecon matched
3  2  Jane  32    Econ        2 Macroecon matched
4  3   Bob  42   Stats        1   Stats I matched
5  4  Carl  67   Trade       NA      <NA> teacher
6  5  <NA>  NA    <NA>        2   History  course
```

An alternative is to request an attribute (`attr = "name"/TRUE`) that also summarizes the join operation, including the output of `fmatch()` (the workhorse of `join()` if `sort = FALSE`).

```
R> join(teacher, course, multiple = TRUE, attr = "jn") |> attr("jn") |> str()

left join: teacher[id] 3/4 (75%) <1:1.33> course[id] 4/5 (80%)
List of 3
 $ call   : language join(x = teacher, y = course, multiple = TRUE,"..
 $ on.cols:List of 2
  ..$ x: chr "id"
  ..$ y: chr "id"
 $ match  : 'qG' int [1:5] 1 2 3 4 NA
  ..- attr(*, "N.nomatch")= int 1
  ..- attr(*, "N.groups")= int 5
  ..- attr(*, "N.distinct")= int 4
```

Users can also invoke the `validate` argument to examine the uniqueness of the join keys in either table: passing a '1' for a non-unique key produces an error.

```
R> join(teacher, course, on = "id", validate = "1:1") |>
+    tryCatch(error = function(e) strwrap(e) |> cat(sep = "\n"))

Error in join(teacher, course, on = "id", validate = "1:1"): Join is
not 1:1: teacher (x) is unique on the join columns; course (y) is
not unique on the join columns
```

A few further particularities are worth highlighting. First, `join()` is class-agnostic and preserves the attributes of `x` (any list-based object). It supports 6 different join operations (`"left"`, `"right"`, `"inner"`, `"full"`, `"semi"`, or `"anti"` join). This demonstrates the latter two:

```
R> for (h in c("semi", "anti")) join(teacher, course, how = h) |> print()

semi join: teacher[id] 3/4 (75%) <1:1st> course[id] 3/5 (60%)
  id names age subject
1  1  John  35    Math
2  2  Jane  32    Econ
3  3   Bob  42   Stats
anti join: teacher[id] 3/4 (75%) <1:1st> course[id] 3/5 (60%)
  id names age subject
1  4  Carl  67   Trade
```

By default (`sort = FALSE`), the order of rows in `x` is preserved. Setting `sort = TRUE` sorts all records in the joined table by the keys.[22] The join relationship is indicated inside the `<>` as the number of records joined from each table divided by the number of unique matches.

[22]This is done using a separate sort-merge-join algorithm, so it is faster than performing a hash join (using `fmatch()`) followed by sorting, particularly if the data is already sorted on the keys.

```
R> course$names <- teacher$names[course$id]
R> join(teacher, course, on = "id", how = "inner", multiple = TRUE)

inner join: teacher[id] 3/4 (75%) <1:1.33> course[id] 4/5 (80%)
duplicate columns: names => renamed using suffix '_course' for y
  id names age subject semester    course names_course
1  1  John  35    Math        1    Math I         John
2  2  Jane  32    Econ        1 Microecon         Jane
3  2  Jane  32    Econ        2 Macroecon         Jane
4  3   Bob  42   Stats        1   Stats I          Bob
```

As shown above, `join()`'s handling of duplicate columns in both tables is rather special. By default (`suffix = NULL`), `join()` extracts the name of the `y` table and appends `y`-columns with it. `x`-columns are not renamed. This is congruent to the principle of adding columns to `x` and altering this table as little as possible. Alternatively, option `drop.dup.cols = "x"/"y"` can be used to simply remove duplicate columns from `x` or `y` before the join operation.

```
R> join(teacher, course, on = "id", multiple = TRUE, drop.dup.cols = "y")

left join: teacher[id] 3/4 (75%) <1:1.33> course[id] 4/5 (80%)
duplicate columns: names => dropped from y
  id names age subject semester    course
1  1  John  35    Math        1    Math I
2  2  Jane  32    Econ        1 Microecon
3  2  Jane  32    Econ        2 Macroecon
4  3   Bob  42   Stats        1   Stats I
5  4  Carl  67   Trade       NA      <NA>
```

A final noteworthy feature is that `fmatch()` has a built-in overidentification check, which warns if more key columns than necessary to identify the records are provided. This check only triggers with 3+ id columns as for efficiency reasons the first two ids are jointly hashed.
`join()` is thus a highly efficient, versatile, and verbose implementation of table joins for R.

5.2. Pivots

The reshaping/pivoting functionality of both commercial and open-source software is also presently unsatisfactory for complex datasets such as surveys or disaggregated production, trade, or financial sector data, where variable names resemble codes and variable labels are essential to making sense of the data. Such datasets can presently only be reshaped by losing these labels or additional manual efforts to retain them. Modern R packages also offer different reshaping functions, such as `data.table::melt()`/`tidyr::pivot_longer()` to combine columns and `data.table::dcast()`/`tidyr::pivot_wider()` to expand them, requiring users to learn both. Since the depreciation of **reshape(2)** (Wickham 2007), there is also no modern replacement for `reshape2::recast()`, requiring R users to consecutively call two reshaping functions, incurring a high cost in terms of both syntax and memory efficiency.

`collapse::pivot` provides a class-agnostic implementation of reshaping for R that addresses these shortcomings: it has a single intuitive syntax to perform 'longer', 'wider', and 'recast' pivots, and supports complex labelled data without loss of information. Its basic syntax is:

```
pivot(data, ids = NULL, values = NULL, names = NULL, labels = NULL,
  how = "longer", na.rm = FALSE, check.dups = FALSE, ...)
```

The demonstration below employs the included Groningen Growth and Development Centre 10-Sector Databas (GGDC10S), providing long-run internationally comparable data on sectoral productivity performance in Africa, Asia, and Latin America. While the database covers 10 sectors, for the demonstration I only retain Agriculture, Mining, and Manufacturing.[23]

```
R> data <- GGDC10S |>
+    fmutate(Label = ifelse(Variable == "VA", "Value Added", "Employment")) |>
+    fsubset(is.finite(AGR), Country, Variable, Label, Year, AGR:MAN)
R> namlab(data, N = TRUE, Ndistinct = TRUE, class = TRUE)
```

```
  Variable     Class    N Ndist         Label
1  Country character 4364    43       Country
2 Variable character 4364     2      Variable
3    Label character 4364     2          <NA>
4     Year   numeric 4364    67          Year
5      AGR   numeric 4364  4353   Agriculture
6      MIN   numeric 4355  4224        Mining
7      MAN   numeric 4355  4353 Manufacturing
```

To reshape this dataset into a longer format, it suffices to call `pivot(data, ids = 1:4)`. If `labels = "name"` is specified, variable labels (stored in `attr(column, "label")`) are saved in an additional column. In addition, `names = list(variable = "var_name", value = "val_name")` can be passed to set alternative names for the `variable` and `value` columns.

```
R> head(dl <- pivot(data, ids = 1:4, names = list("Sectorcode", "Value"),
+                   labels = "Sector", how = "longer"))
```

```
  Country Variable       Label Year Sectorcode      Sector Value
1     BWA       VA Value Added 1964        AGR Agriculture 16.30
2     BWA       VA Value Added 1965        AGR Agriculture 15.73
3     BWA       VA Value Added 1966        AGR Agriculture 17.68
4     BWA       VA Value Added 1967        AGR Agriculture 19.15
5     BWA       VA Value Added 1968        AGR Agriculture 21.10
6     BWA       VA Value Added 1969        AGR Agriculture 21.86
```

`pivot()` only requires essential information and intelligently guesses the rest. For example, the same result could have been obtained by specifying `values = c("AGR", "MIN", "MAN")` instead of `ids = 1:4`. An exact reverse operation can also be specified as `pivot(dl, 1:4, "Value", "Sectorcode", "Sector", "wider")`, with `dl` the long data.

[23]The `"Label"` column is added for demonstration purposes. `namlab()` provides a compact overview of variable names and labels stored in `attr(column, "label")`, with (optional) additional information/statistics.

The second option is a wider pivot with `how = "wider"`. Here, `names` and `labels` can be used to select columns containing the names of new columns and their labels.[24] Note below how the labels are combined with existing labels such that also this operation is without loss of information. It is, however, a destructive operation—with 2 or more columns selected through `values`, `pivot()` is not able to reverse it. Further arguments like `na.rm`, `fill`, `sort`, and `transpose` can be used to control the casting process.

```
R> head(dw <- pivot(data, c("Country", "Year"), names = "Variable",
+                   labels = "Label", how = "w"))

  Country Year AGR_VA AGR_EMP MIN_VA MIN_EMP MAN_VA MAN_EMP
1     BWA 1964  16.30   152.1  3.494  1.9400 0.7366   2.420
2     BWA 1965  15.73   153.3  2.496  1.3263 1.0182   2.330
3     BWA 1966  17.68   153.9  1.970  1.0022 0.8038   1.282
4     BWA 1967  19.15   155.1  2.299  1.1192 0.9378   1.042
5     BWA 1968  21.10   156.2  1.839  0.7855 0.7503   1.069
6     BWA 1969  21.86   157.4  5.245  2.0314 2.1396   2.124
```

```
R> namlab(dw)

  Variable                         Label
1  Country                       Country
2     Year                          Year
3   AGR_VA   Agriculture - Value Added
4  AGR_EMP    Agriculture - Employment
5   MIN_VA        Mining - Value Added
6  MIN_EMP         Mining - Employment
7   MAN_VA Manufacturing - Value Added
8  MAN_EMP  Manufacturing - Employment
```

For the recast pivot (`how = "recast"`), unless a column named `variable` exists in the data, the source and (optionally) destination of variable names need to be specified using a list passed to `names`, and similarly for `labels`. Again, taking along labels is entirely optional—omitting either the labels-list's `from` or `to` element will omit the respective operation.

```
R> head(dr <- pivot(data, c("Country", "Year"),
+               names = list(from = "Variable", to = "Sectorcode"),
+               labels = list(from = "Label", to = "Sector"), how = "r"))

  Country Year Sectorcode      Sector    VA   EMP
1     BWA 1964        AGR Agriculture 16.30 152.1
2     BWA 1965        AGR Agriculture 15.73 153.3
3     BWA 1966        AGR Agriculture 17.68 153.9
4     BWA 1967        AGR Agriculture 19.15 155.1
5     BWA 1968        AGR Agriculture 21.10 156.2
6     BWA 1969        AGR Agriculture 21.86 157.4
```

[24]If multiple columns are selected, they are combined using `"_"` for names and `" - "` for labels.

```
R> vlabels(dr)[3:6]

   Sectorcode         Sector             VA            EMP 
           NA             NA "Value Added"   "Employment" 
```

This (`dr`) is the tidy format (Wickham 2014) where each variable is a separate column. It is analytically more useful, e.g., to compute labor productivity as `settransform(dr, LP = VA / EMP)`, or to estimate a panel-regression with sector fixed-effects. The recast pivot is thus a natural operation to change data representations. As with the other pivots, it preserves all information and can be reversed by simply swapping the contents of the `from` and `to` keywords.

`pivot()` also supports fast aggregation pivots, the default being `FUN = "last"`, which simply overwrites values in appearance order if the combination of `ids` and `names` does not fully identify the data. The latter can be checked with `check.dups = TRUE`. There are a small number of internal functions: `"first"`, `"last"`, `"sum"`, `"mean"`, `"min"`, `"max"`, and `"count"`. These carry out computations 'on the fly' and are thus extremely fast. `pivot()` also supports *Fast Statistical Functions*, which will yield vectorized aggregations, but require a deep copy of the columns aggregated which is avoided using the internal functions. The following example performs aggregation across years with the internal mean function during a recast pivot.

```
R> head(dr_agg <- pivot(data, "Country", c("AGR", "MIN", "MAN"), how = "r",
+      names = list(from = "Variable", to = "Sectorcode"),
+      labels = list(from = "Label", to = "Sector"), FUN = "mean"))

  Country Sectorcode      Sector       VA      EMP
1     BWA        AGR Agriculture    462.2   188.06
2     ETH        AGR Agriculture  34389.9 17624.34
3     GHA        AGR Agriculture   1549.4  3016.04
4     KEN        AGR Agriculture 139705.9  5348.91
5     MWI        AGR Agriculture  28512.6  2762.62
6     MUS        AGR Agriculture   3819.6    59.34
```

More features of `pivot()` are demonstrated in the documentation examples. Notably, it can also perform longer and recast pivots without id variables, like `data.table::transpose()`.

6. List processing

Often in programming, nested structures are needed. A typical use case involves running statistical procedures for multiple configurations of variables and parameters and saving multiple objects (such as a model predictions and performance statistics) in a list. Nested data is also often the result of web scraping or web APIs. A typical use case in development involves serving different data according to user choices. Except for certain recursive functions found in packages such as **purr**, **tidyr**, or **rrapply**, R lacks a general recursive toolkit to create, query, and tidy nested data. **collapse**'s list processing functions attempt to provide a basic toolkit.

To create nested data, `rsplit()` generalizes `split()` and (recursively) splits up data frame-like objects into (nested) lists. For example, we can split the `GGDC10S` data by country and variable, such that, e.g., agricultural employment in Argentina can be accessed as:[25]

```
R> dl <- GGDC10S |> rsplit( ~ Country + Variable)
R> dl$ARG$EMP$AGR[1:12]
```

```
 [1] 1800 1835 1731 2030 1889 1843 1789 1724 1678 1725 1650 1553
```

This is a convenient data representation for *Shiny Apps* where we can let the user choose data (e.g., `dl[[input$country]][[input$variable]][[input$sector]]`) without expensive subsetting operations. As mentioned, such data representation can also be the result of an API call parsing JSON or a nested loop or `lapply()` call. Below, I write a nested loop running a regression of agriculture on mining and manuacturing output/employment:

```
R> result <- list()
R> for (country in c("ARG", "BRA", "CHL")) {
+    for (variable in c("EMP", "VA")) {
+      m <- lm(log(AGR+1) ~ log(MIN+1) + log(MAN+1) + Year,
+               data = dl[[country]][[variable]])
+      result[[country]][[variable]] <- list(model = m, BIC = BIC(m),
+                                             summary = summary(m))
+    }
+  }
```

This programming may not be ideal for this particular use case as I could have used data.frame-based tools and saved the result in a column.[26] However, there are limits to data.frame-based workflows. For example, I recently trained a complex ML model for different variables and parameters, which involved loading a different dataset at each iteration. Loops are useful in such cases, and lists a natural vehicle to structure complex outputs. The main issue with nested lists is that they are complex to query. What if we want to know just the R^2 of these 6 models? We would need to use, e.g., `result$ARG$EMP$summary$r.squared` for each model.

This nested list-access problem was the main reason for creating `get_elem()`: an efficient recursive list-filtering function which, by default, simplifies the list tree as much as possible.

```
R> str(r_sq_l <- result |> get_elem("r.squared"))
```

```
List of 3
 $ ARG:List of 2
  ..$ EMP: num 0.907
  ..$ VA : num 1
 $ BRA:List of 2
  ..$ EMP: num 0.789
  ..$ VA : num 0.999
 $ CHL:List of 2
  ..$ EMP: num 0.106
  ..$ VA : num 0.999
```

[25]If a nested structure is not needed, `flatten = TRUE` lets `rsplit()` operate like a faster version of `split()`.
[26]E.g., `GGDC10S |> fgroup_by(Country, Variable) |> fsummarise(result = my_fun(lm(log(AGR+1) log(MIN+1) + log(MAN+1) + Year)))` with `my_fun <- function(m) list(list(m, BIC(m), summary(m)))`.

```
R> rowbind(r_sq_l, idcol = "Country", return = "data.frame")

  Country    EMP     VA
1     ARG 0.9068 0.9996
2     BRA 0.7888 0.9988
3     CHL 0.1058 0.9991
```

Note how the `"summary"` branch was eliminated since it is common to all final nodes; `result |> get_elem("r.squared", keep.tree = TRUE)` could have been used to retain it. `rowbind()` then efficiently combines lists of lists. We can also apply `t_list()` to turn the list inside-out.

```
R> r_sq_l |> t_list() |> rowbind(idcol = "Variable", return = "data.frame")

  Variable    ARG    BRA    CHL
1      EMP 0.9068 0.7888 0.1058
2       VA 0.9996 0.9988 0.9991
```

`rowbind()` is limited if `get_elem()` returns a more nested or asymmetric list, potentially with vectors/arrays in the final nodes. Suppose we want to extract the coefficient matrices:

```
R> result$ARG$EMP$summary$coefficients

              Estimate Std. Error  t value  Pr(>|t|)
(Intercept)  26.583617  1.2832583  20.7157 1.747e-28
log(MIN + 1)  0.083168  0.0352493   2.3594 2.169e-02
log(MAN + 1) -0.064413  0.0767614  -0.8391 4.048e-01
Year         -0.009683  0.0005556 -17.4278 1.003e-24
```

For such cases, I created `unlist2d()` as a complete recursive generalization of `unlist()`. It creates a ‘`data.frame`’ (or ‘`data.table`’) representation of any nested list using recursive row-binding and coercion operations while generating (optional) id variables representing the list tree and (optionally) saving row names of matrices or data frames. In the present example

```
R> result |> get_elem("coefficients") |> get_elem(is.matrix) |>
+    unlist2d(idcols = c("Country", "Variable"),
+             row.names = "Covariate") |> head(3)

  Country Variable    Covariate Estimate Std. Error t value  Pr(>|t|)
1     ARG      EMP  (Intercept) 26.58362    1.28326 20.7157 1.747e-28
2     ARG      EMP log(MIN + 1)  0.08317    0.03525  2.3594 2.169e-02
3     ARG      EMP log(MAN + 1) -0.06441    0.07676 -0.8391 4.048e-01
```

where `get_elem(is.matrix)` is needed because the models also contain `"coefficients"`.

This exemplifies the power of these tools to create, query, and combine nested data in very general ways, and with many applications. Further useful functions include `has_elem()` to check for the existence of elements, `ldepth()` to return the maximum level of recursion, and `is_unlistable()` to check whether a list has atomic elements in all final nodes.

7. Summary statistics

collapse's summary statistics functions offer a parsimonious and powerful toolkit to examine complex datasets. A particular focus has been on providing tools for examining longitudinal (panel) data. Recall the indexed World Development Panel (`wlddev`) from Section 2.2. The function `varying()` can be used to examine which of these variables are time-varying.

```
R> varying(wlddev, ~ iso3c)
```

```
      country          date          year        decade        region 
        FALSE          TRUE          TRUE          TRUE         FALSE 
       income          OECD         PCGDP        LIFEEX          GINI 
        FALSE         FALSE          TRUE          TRUE          TRUE 
          ODA           POP  center_PCGDP center_LIFEEX 
         TRUE          TRUE          TRUE          TRUE 
```

A related exercise is to decompose the variance of a panel series into variation between countries and variation within countries over time. Using the (de-)meaning functions supporting '`indexed_series`' from Table 2, this is easily demonstrated.

```
R> LIFEEXi <- reindex(wlddev$LIFEEX, wlddev$iso3c)
R> all.equal(fvar(LIFEEXi), fvar(fbetween(LIFEEXi)) + fvar(fwithin(LIFEEXi)))
```

```
[1] TRUE
```

The `qsu()` (quick-summary) function provides an efficient method to (approximately) compute this decomposition, considering the group-means instead of the between transformation[27] and adding the mean back to the within transformation to preserve the scale of the data.

```
R> qsu(LIFEEXi)
```

```
              N/T     Mean       SD      Min      Max
Overall     11670  64.2963  11.4764   18.907  85.4171
Between       207  64.9537   9.8936  40.9663  85.4171
Within    56.3768  64.2963   6.0842  32.9068  84.4198
```

The decomposition above implies more variation in life expectancy between countries than within countries over time. It can also be computed for different subgroups, such as OECD members and non-members, and with sampling weights, such as population.

```
R> qsu(LIFEEXi, g = wlddev$OECD, w = wlddev$POP) |> aperm()
```

[27] This is more efficient and equal to using the between transformation if the panel is balanced.

```
, , FALSE

             N/T        WeightSum     Mean      SD      Min      Max
Overall     9503  2.48998382e+11  63.5476  9.2368   18.907  85.4171
Between      171             171  63.5476  6.0788  43.0905  85.4171
Within   55.5731  1.45613089e+09  65.8807  6.9545  30.3388  82.8832

, , TRUE

             N/T        WeightSum     Mean      SD      Min      Max
Overall     2156  6.38797019e+10  74.9749  5.3627   45.369  84.3563
Between       36              36  74.9749  2.9256  66.2983  78.6733
Within   59.8889  1.77443616e+09  65.8807  4.4944  44.9513  77.2733
```

The output shows that the variation in life expectancy is significantly larger for non-OECD countries. For the latter the between- and within-country variation is approximately equal.[28] For greater detail, `descr()` provides a rich (grouped, weighted) statistical description.

```
R> wlda15 <- wlddev |> fsubset(year >= 2015) |> fgroup_by(iso3c) |> flast()
R> wlda15 |> descr(income + LIFEEX ~ OECD)

Dataset: wlda15, 2 Variables, N = 216
Grouped by: OECD [2]
         N   Perc
FALSE  180  83.33
TRUE    36  16.67
------------------------------------------------------------------------------
income (factor): Income Level
Statistics (N = 216)
         N   Perc  Ndist
FALSE  180  83.33      4
TRUE    36  16.67      2

Table (Freq Perc)
                      FALSE     TRUE    Total
High income          45 25.0  34 94.4  79 36.6
Upper middle income  58 32.2   2  5.6  60 27.8
Lower middle income  47 26.1   0  0.0  47 21.8
Low income           30 16.7   0  0.0  30 13.9
------------------------------------------------------------------------------
LIFEEX (numeric): Life expectancy at birth, total (years)
Statistics (N = 200, 7.41% NAs)
         N  Perc  Ndist   Mean    SD    Min    Max   Skew  Kurt
FALSE  164    82    164  71.25  7.06  53.28  85.08   -0.5  2.61
TRUE    36    18     36  80.83  2.55  75.05  84.36  -0.92  2.72
```

[28] `qsu()` also has a convenient formula interface to perform these transformations in an ad-hoc fashion, e.g., the above can be obtained using `qsu(wlddev, LIFEEX ~ OECD, ~ iso3c, ~ POP)`, without prior indexation.

```
Quantiles
          1%     5%    10%    25%    50%    75%    90%    95%    99%
FALSE   54.3  58.38  61.25  66.43  72.56  76.68  78.93  80.88  83.77
TRUE   75.12  75.83   76.8  79.04  81.77  82.63  83.21  83.54  84.13
-----------------------------------------------------------------------------
```

While `descr()` does not support panel-variance decompositions like `qsu()`, it also computes detailed (grouped, weighted) frequency tables for categorical data and is thus very utile with mixed-type data. A `stepwise` argument toggles describing one variable at a time, allowing users to naturally 'click-through' a large dataset rather than printing a massive output to the console. The documentation provides more details and examples. Both `qsu()` and `descr()` provide an `as.data.frame()` method for efficient tidying and further analysis.

A final noteworthy function from **collapse**'s descriptive statistics toolkit is `qtab()`, an enhanced drop-in replacement for `table()`. It is enhanced both in a statistical and a computational sense, providing a remarkable performance boost, an option (`sort = FALSE`) to preserve the first-appearance-order of vectors being cross-tabulated, support for frequency weights (`w`), and the ability to compute different statistics representing table entries using these weights—vectorized when using *Fast Statistical Functions*, as demonstrated below.

```
R> wlda15 |> with(qtab(OECD, income))

       income
OECD    High income Low income Lower middle income Upper middle income
  FALSE          45         30                  47                  58
  TRUE           34          0                   0                   2
```

This shows the total population (latest post-2015 estimates) in millions.

```
R> wlda15 |> with(qtab(OECD, income, w = POP) / 1e6)

       income
OECD    High income Low income Lower middle income Upper middle income
  FALSE       93.01     694.89             3063.54             2459.71
  TRUE      1098.75       0.00                0.00              211.01
```

This shows the average life expectancy in years. The use of `fmean()` toggles an efficient vectorized computation of the table entries (i.e., `fmean()` is only called once).

```
R> wlda15 |> with(qtab(OECD, income, w = LIFEEX, wFUN = fmean))

       income
OECD    High income Low income Lower middle income Upper middle income
  FALSE       78.75      62.81               68.30               73.81
  TRUE        81.09                                              76.37
```

Finally, this calculates a population-weighted average of life expectancy in each group.

```
R> wlda15 |> with(qtab(OECD, income, w = LIFEEX, wFUN = fmean,
+                        wFUN.args = list(w = POP)))
```

```
       income
OECD    High income Low income Lower middle income Upper middle income
  FALSE       77.91      63.81               68.76               75.93
  TRUE        81.13                                              76.10
```

'qtab' objects inherit the 'table' class, thus all 'table' methods apply. Apart from the above functions, **collapse** also provides functions `pwcor()`, `pwcov()`, and `pwnobs()` for convenient (pairwise, weighted) correlations, covariances, and observations counts, respectively.

8. Global options

collapse is globally configurable to an extent few packages are: the default value of key function arguments governing the behavior of its algorithms, and the exported namespace, can be adjusted interactively through the `set_collapse()` function. These options are saved in an internal environment called `.op`. Its contents can be accessed using `get_collapse()`.

The current set of options comprises the default behavior for missing values (`na.rm` arguments in all statistical functions and algorithms), sorted grouping (`sort`), multithreading and algorithmic optimizations (`nthreads`, `stable.algo`), presentational settings (`stub`, `digits`, `verbose`), and, surpassing all else, the package namespace itself (`mask`, `remove`).

As evident from previous sections, **collapse** provides performance-improved or otherwise enhanced versions of functionality already present in base R (like the *Fast Statistical Functions*, `funique()`, `fmatch()`, `fsubset()`, `ftransform()`, etc.) and other packages (esp. **dplyr** (Wickham *et al.* 2023a): `fselect()`, `fsummarise()`, `fmutate()`, `frename()`, etc.). The objective of being namespace compatible warrants such a naming convention, but this has a syntactical cost, particularly when **collapse** is the primary data manipulation package.

To reduce this cost, **collapse**'s `mask` option allows masking existing R functions with the faster **collapse** versions by creating additional functions in the namespace and instantly exporting them. All **collapse** functions starting with 'f' can be passed to the option (with or without the 'f'), e.g., `set_collapse(mask = c("subset", "transform"))` creates `subset <- fsubset` and `transform <- ftransform` and exports them. Special functions are `"n"`, `"table"`/`"qtab"`, and `"%in%"`, which create `n <- GRPN` (for use in `(f)summarise`/`(f)mutate`), `table <- qtab`, and replace `%in%` with a fast version using `fmatch()`, respectively. There are also several convenience keywords to mask related groups of functions. The most powerful of these is `"all"`, which masks all f-functions and special functions, as shown below.

```
set_collapse(mask = "all")
wlddev |> subset(year >= 1990 & is.finite(GINI)) |>
  group_by(year) |>
  summarise(n = n(), across(PCGDP:GINI, mean, w = POP))
with(mtcars, table(cyl, vs, am))
sum(mtcars)
diff(EuStockMarkets)
mean(num_vars(iris), g = iris$Species)
unique(wlddev, cols = c("iso3c", "year"))
range(wlddev$date)
wlddev |> index_by(iso3c, year) |>
  mutate(PCGDP_lag = lag(PCGDP),
         PCGDP_diff = PCGDP - PCGDP_lag,
         PCGDP_growth = growth(PCGDP)) |> unindex()
```

The above is now 100% **collapse** code. Similarly, using this option, all code in this article could have been written without f-prefixes. Thus, **collapse** is able to offer a fast and syntactically clean experience of R - without the need to even restart the session. Masking is completely and interactively reversible: calling `set_collapse(mask = NULL)` instantly removes the additional functions. Option `remove` can further be used to remove (un-export) any **collapse** function, allowing manual conflict management. Function `fastverse::fastverse_conflicts()` from the related **fastverse** project (Krantz 2024) can be used to display namespace conflicts with **collapse**. Invoking either `mask` or `remove` detaches **collapse** and re-attaches it at the top of the search path, letting its namespace to take precedence over other packages.

9. Benchmark

This section provides several simple benchmarks to show that **collapse** provides best-in-R performance for statistics and data manipulation on moderately sized datasets. They are executed on an Apple M1 MacBook Pro with 16 GB unified memory. It also discusses results from 3rd party benchmarks involving **collapse**. The first set of benchmarks show that **collapse** provides faster computationally intensive operations like unique values and matching on large integer and character vectors. It creates integer/character vectors of 10 million obs, with 1000 unique integers and 5776 unique strings, respectively, which are deduplicated/matched in the benchmark. These fast basic operations impact many critical components of the package.

```
R> set.seed(101)
R> int <- 1:1000; g_int <- sample.int(1000, 1e7, replace = TRUE)
R> char <- c(letters, LETTERS, month.abb, month.name)
R> g_char <- sample(char <- outer(char, char, paste0), 1e7, TRUE)
R> bmark(base_int = unique(g_int), collapse_int = funique(g_int))

    expression     min  median mem_alloc n_itr n_gc total_time
1     base_int 63.27ms 65.41ms    166.2MB    29   29      2.05s
2 collapse_int  8.36ms  8.62ms     38.2MB   217   44         2s

R> bmark(base_char = unique(g_char), collapse_char = funique(g_char))

     expression     min  median mem_alloc n_itr n_gc total_time
1     base_char 101.5ms 101.5ms    166.2MB     1   19    101.5ms
2 collapse_char  22.4ms  23.5ms     38.2MB    69   12       1.7s
```

```
R> bmark(base_int = match(g_int, int), collapse_int = fmatch(g_int, int))

    expression     min  median mem_alloc n_itr n_gc total_time
1     base_int  26.6ms 26.96ms    76.3MB    33   33   908.78ms
2 collapse_int  7.99ms  8.29ms    38.2MB   182   37      1.51s

R> bmark(base_char = match(g_char, char), data.table_char =
+        chmatch(g_char, char), collapse_char = fmatch(g_char, char))

       expression    min median mem_alloc n_itr n_gc total_time
1       base_char 57.1ms 57.9ms   114.5MB     9   22   520.29ms
2 data.table_char 40.5ms 41.5ms    38.1MB    40    8      1.66s
3   collapse_char 11.9ms 12.3ms    38.1MB   126   26      1.55s
```

The second set below shows that **collapse**'s statistical functions are very efficient on aggregating a numeric matrix with 10,000 rows and 1000 columns. They are faster than base R even without multithreading, but using 4 threads in this case induces a sizeable difference.

```
R> set_collapse(na.rm = FALSE, sort = FALSE, nthreads = 4)
R> m <- matrix(rnorm(1e7), ncol = 1000)
R> bmark(R = colSums(m), collapse = fsum(m))

  expression    min median mem_alloc n_itr n_gc total_time
1          R 9.19ms 9.25ms    7.86KB   215    0      2.01s
2   collapse  1.3ms 1.34ms    7.86KB  1480    0         2s

R> bmark(R = colMeans(m), collapse = fmean(m))

  expression    min median mem_alloc n_itr n_gc total_time
1          R  9.2ms 9.25ms    7.86KB   215    0         2s
2   collapse 1.29ms 1.34ms    7.86KB  1477    1         2s

R> bmark(MS = matrixStats::colMedians(m), collapse = fmedian(m))

  expression   min  median mem_alloc n_itr n_gc total_time
1         MS 106ms 106.3ms   86.04KB    19    0      2.04s
2   collapse  25ms  25.3ms    7.86KB    79    0      2.01s
```

Below I also show a grouped version summing the columns within 1000 random groups.

```
R> g <- sample.int(1e3, 1e4, TRUE)
R> bmark(R = rowsum(m, g), collapse = fsum(m, g))

  expression    min median mem_alloc n_itr n_gc total_time
1          R 7.06ms 7.39ms    7.85MB   245   12      1.85s
2   collapse 1.76ms    2ms    7.67MB   799   38      1.69s
```

I now turn to basic operations on a medium sized real-world database recording all flights from New York City (EWR, JFK, and LGA) in 2023—provided by the **nycflights23** package. The `flights` table has 435k flights, and grouping it by day and route yields 76k unique trips.

```
R> fastverse_extend(nycflights23, dplyr, data.table); setDTthreads(4)
R> list(flights, airports, airlines, planes, weather) |> sapply(nrow)

[1] 435352   1251     14   4840  26204

R> flights |> fselect(month, day, origin, dest) |> fnunique()

[1] 75899
```

In the following, I select 6 numeric variables and sum them across the 76k trips using **dplyr**, **data.table**, and **collapse**. Ostensibly, despite `sum()` being 'primitive' (implemented in C), there is a factor 100 between **dplyr**'s split-apply-combine and **collapse**'s fully vectorized execution.

```
R> vars <- .c(dep_delay, arr_delay, air_time, distance, hour, minute)
R> bmark(dplyr = flights |> group_by(month, day, origin, dest) |>
+                   summarise(across(all_of(vars), sum), .groups = "drop"),
+        data.table = qDT(flights)[, lapply(.SD, sum), .SDcols = vars,
+                                  by = .(month, day, origin, dest)],
+        collapse = flights |> fgroup_by(month, day, origin, dest) |>
+                      get_vars(vars) |> fsum())

  expression     min   median mem_alloc n_itr n_gc total_time
1      dplyr 464.4ms 543.81ms   51.46MB     4   32      2.19s
2 data.table  10.2ms   11.1ms   18.93MB   161   23      2.01s
3   collapse   4.6ms   5.05ms    9.11MB   355   20         2s
```

Below, I also benchmark the mean and median functions in the same way. It is evident that with non-primitive R functions the split-apply-combine logic is even more costly.

```
       expression     min   median mem_alloc n_itr n_gc total_time
1      dplyr_mean   1.48s    1.53s   51.46MB     2   42      3.07s
2 data.table_mean 10.41ms 11.46ms   18.93MB   152   20         2s
3   collapse_mean  4.77ms  5.12ms    9.11MB   349   20         2s

         expression     min   median mem_alloc n_itr n_gc total_time
1      dplyr_median   5.62s    5.62s    55.7MB     1   91      5.62s
2 data.table_median 29.19ms 30.27ms    18.9MB    65    5      2.02s
3   collapse_median  13.6ms  14.6ms    11.1MB   133   10      2.01s
```

So far, **data.table**, by virtue of it's internal vectorizations (also via dedicated grouped C implementations of simple functions), is competitive.[29] Below, I compute the range of one column (x) using `max(x) - min(x)`. As elucidated in Section 3.1, this expression is also vectorized in **collapse**, where it amounts to `fmax(x, g) - fmin(x, g)`, but not in **data.table**.

[29]Much longer data will likely also favor **data.table** over **collapse** due to its sub-column-level parallel grouping and implementation of simple functions like `sum()` and `mean()`, see, e.g., the DuckDB Benchmarks.

```
R> bmark(dplyr = flights |> group_by(month, day, origin, dest) |>
+      summarise(rng = max(arr_delay) - min(arr_delay), .groups = "drop"),
+    data.table = qDT(flights)[, .(rng = max(arr_delay) - min(arr_delay)),
+                              by = .(month, day, origin, dest)],
+    collapse = flights |> fgroup_by(month, day, origin, dest) |>
+      fsummarise(rng = fmax(arr_delay) - fmin(arr_delay)))
```

```
  expression      min   median mem_alloc n_itr n_gc total_time
1      dplyr 121.05ms 134.78ms   23.08MB    13   29      2.03s
2 data.table   77.8ms  82.68ms    5.77MB    24   15      2.01s
3   collapse   6.73ms   7.25ms     6.8MB   264    7         2s
```

I also benchmark table joins and pivots. The following demonstrates how all tables can be joined together using **collapse** and its default first-match left-join, which preserves `flights`.

```
R> flights |> join(weather, on = c("origin", "time_hour")) |>
+    join(planes, on = "tailnum") |> join(airports, on = c(dest = "faa")) |>
+    join(airlines, on = "carrier") |> dim()
```

```
left join: flights[origin, time_hour] 434462/435352 (99.8%) <21.94:1st> weat
duplicate columns: year, month, day, hour => renamed using suffix '_weather'
left join: x[tailnum] 424068/435352 (97.4%) <87.62:1st> planes[tailnum] 4840
duplicate columns: year => renamed using suffix '_planes' for y
left join: x[dest] 427868/435352 (98.3%) <3753.23:1st> airports[faa] 114/125
left join: x[carrier] 435352/435352 (100%) <31096.57:1st> airlines[carrier]
duplicate columns: name => renamed using suffix '_airlines' for y
[1] 435352     48
```

The verbosity of `join()` is essential to understanding what has happened here—how many records from each table were matched and which duplicate non-id columns were suffixed with the (default) y-table name. Normally, I would set `drop.dup.cols = "y"` as it seems not useful to preserve them here, but the other packages don't have this option. For the benchmark, I set `verbose = 0` in **collapse** and employ the fastest syntax for **dplyr** and **data.table**:[30]

```
         expression     min  median mem_alloc n_itr n_gc total_time
1       dplyr_joins 183.2ms 220.5ms   558.9MB     9   19      2.07s
2 data.table_joins 130.8ms 184.1ms   490.4MB    12   27       2.2s
3    collapse_joins  12.5ms  15.5ms    89.7MB   116   33      2.01s
```

Evidently, the vectorized hash join provided by **collapse** is 10x faster than **data.table** on this database, at a substantially lower memory footprint. It remains competitive on big data.[31]

Last but not least, I benchmark pivots, starting a with long pivot that simply melt the 6 columns aggregated beforehand into one column, duplicating all other columns 6 times:

[30] `left_join(..., multiple = "first")` for **dplyr** and `y[x, on = ids, mult = "first"]` for **data.table**.

[31] **data.table** joins utilize multithreaded radix-ordering—a very different logic more useful for big data.

```
R> bmark(tidyr = tidyr::pivot_longer(flights, cols = vars),
+        data.table = qDT(flights) |> melt(measure = vars),
+        collapse = pivot(flights, values = vars))

  expression    min median mem_alloc n_itr n_gc total_time
1      tidyr 82.6ms 82.7ms     251MB     5   23      416ms
2 data.table 50.9ms   52ms     209MB    11   18      579ms
3   collapse 17.2ms 18.5ms     209MB    24   43      456ms
```

Memory-wise, **collapse** and **data.table** are equally efficient, but **collapse** is faster, presumably due to more extensive use of `memset()` to copy values in C, or smaller R-level overheads.

To complete the picture, I also also perform a wide pivot where the 6 columns are summed (for efficiency) across the 3 origin airports and expanded to create 18 airport-value columns.

```
R> bmark(tidyr = tidyr::pivot_wider(flights, id_cols = .c(month, day, dest),
+            names_from = "origin", values_from = vars, values_fn = sum),
+        data.table = dcast(qDT(flights), month + day + dest ~ origin,
+                           value.var = vars, fun = sum),
+        collapse_fsum = pivot(flights, .c(month, day, dest), vars,
+                              "origin", how = "wider", FUN = fsum),
+        collapse_itnl = pivot(flights, .c(month, day, dest), vars,
+                              "origin", how = "wider", FUN = "sum"))

     expression      min   median mem_alloc n_itr n_gc total_time
1         tidyr 452.49ms 496.93ms   142.4MB     4   25      2.03s
2    data.table 299.13ms 304.01ms      21MB     7   25      2.19s
3 collapse_fsum   8.65ms  10.23ms    39.1MB   169   12         2s
4 collapse_itnl   7.06ms   7.69ms    12.4MB   249    5         2s
```

Again, **collapse** is fastest, as it offers full vectorization, either via `fsum()`, which translates to `fsum(x, g, TRA = "fill")` before pivoting and thus entails a full deep copy of the `vars` columns, or via an optimized internal sum function which sums values 'on the fly' during the reshaping process. **data.table** is not vectorized here but at least memory efficient.

In summary, these benchmarks shows that **collapse** provides outstanding performance and memory efficiency on a typical medium-sized real-world database popular in the R community.

9.1. Other benchmarks

The DuckDB Benchmarks compare many software packages for database-like operations using large datasets (big data) on a linux server. The January 2025 run distinguishes 6 packages that consistently achieve outstanding performances: **DuckDB**, **Polars**, **ClickHouse**, Apache **Datafusion**, **data.table**, and **collapse**. Of these, **DuckDB**, **ClickHouse**, and **Datafusion** are vectorized database (SQL) engines, and **Polars** is a Python/Rust DataFrame library and SQL engine. These four are supported by (semi-)commercial entities, leaving **data.table** as the only fully community-led project, and **collapse** as the only project that is single-authored and without financial support. The benchmarks show that **collapse** achieves the highest relative performance on 'smaller' datasets (10-100 million rows) and performing advanced operations.

Since June 2024, there is also an independent database-like operations benchmark by Adrian Antico using a windows server and executing scripts inside IDEs (VScode, Rstudio), on which **collapse** achieved the overall fastest runtimes. I also very recently started a user-contributed benchmark Wiki as part of the fastverse project promoting high-performance software for R, where users can freely contribute benchmarks involving, but not limited to, **fastverse** packages. These benchmarks align in showing that **collapse** offers a computationally outstanding experience, particularly for medium-sized datasets, complex tasks, and on windows systems.[32]

9.2. Limitations and outlook

collapse maximizes three principal objectives: being class-agnostic/fully compatible with the R ecosystem (supporting statistical operations on vector, matrix and data.frame-like objects), being statistically advanced, and being fast. This warranted some design choices away from maximum performance for large data manipulation.[33] Its limited use of multithreading and SIMD instructions, partly by design constraints and by R's C API, and the use of standard types for internal indexing, imposes hard-limits—the maximum integer in R is 2,147,483,647 $\rightarrow$ the maximum vector length **collapse** supports. It is and will remain an in-memory tool.

Despite these constraints, **collapse** provides very respectable performance even on very large datasets by virtue of its algorithmic and memory efficiency. It is, together with the popular **data.table** package offering more sub-column-level parallel architecture for basic operations, well-positioned to remain a premier tool for in-memory statistics and data manipulation.

10. Conclusion

collapse was first released to CRAN in March 2020, and has grown and matured considerably over the course of 5 years. It has become a new foundation package for statistical computing and data transformation in R—one that is statistically advanced, class-agnostic, flexible, fast, lightweight, stable, and able to manipulate complex scientific data with ease. As such, it opens up new possibilities for statistics, data manipulation, and package development in R.

This article provided a quick guide to the package, articulating its key ideas and design principles and demonstrating all core features. At this point the API is stable—it has changed very little over the 5 years and no further changes are planned. Compatibility with R version 3.4.0 will be maintained for as long as possible. Minor new features are currently planned.

For deeper engagement with **collapse**, visit its website or start with the vignette summarizing all available documentation and resources. Users can also follow **collapse** on Twitter/X and Bluesky to be notified about major updates and participate in community discussions.

[32] Reasons for the particularly strong performance of **collapse** on Windows may be that it is largely written in C and has limited multithreading in favor or highly efficient serial algorithms—there appear to be persistent obstacles to efficient (low-overhead) multithreading on Windows, implying that multithreaded query engines do not develop their full thrust on medium-sized ($\leq$100 million row) datasets.

[33] Which nowadays would demand creating a multithreaded, vectorized query engine with optimized memory buffers/vector types to take full advantage of SIMD processing as in **DuckDB** or **Polars**. Such an architecture is very difficult to square with R vectors and R's 30-year old C API.

Computational details

The results in this paper were obtained using R (R Core Team 2023) 4.3.0 with **collapse** 2.0.19, **data.table** 1.16.4, **dplyr** 1.1.4, **tidyr** 1.3.1, **matrixStats** 1.0.0, **fastverse** 0.3.4, and **bench** 1.1.3 (Hester and Vaughan 2023). All packages used are available from the Comprehensive R Archive Network (CRAN) at https://CRAN.R-project.org/. The benchmark was run on an Apple M1 MacBook Pro (2020) with 16GB unified memory. Packages were compiled from source using Homebrew Clang version 16.0.4 with OpenMP enabled and the -O2 flag.

Acknowledgments

The source code of **collapse** has been heavily inspired by (and partly copied from) **data.table** (Matt Dowle and Arun Srinivasan), R's source code (R Core Team and contributors worldwide), the **kit** package (Morgan Jacob), and **Rcpp** (Dirk Eddelbuettel). Packages **plm** (Yves Croissant, Giovanni Millo, and Kevin Tappe) and **fixest** (Laurent Berge) have also provided a lot of inspiration (and a port to its demeaning algorithm in the case of **fixest**). I also thank many people from diverse fields for helpful answers on Stackoverflow and many other people for encouragement, feature requests, and helpful issues and suggestions.

References

Balassa B (1965). "Tariff protection in industrial countries: an evaluation." *Journal of Political Economy*, **73**(6), 573–594.

Bengtsson H (2023). ***matrixStats**: Functions that Apply to Rows and Columns of Matrices (and to Vectors).* R package version 1.0.0, URL https://CRAN.R-project.org/package=matrixStats.

Bergé L (2018). "Efficient Estimation of Maximum Likelihood Models with Multiple Fixed-Effects: the R Package **FENmlm**." *CREA Discussion Papers*, (13).

Bouchet-Valat M, Kamiński B (2023). "**DataFrames.jl**: Flexible and Fast Tabular Data in Julia." *Journal of Statistical Software*, **107**(4), 1–32. doi:10.18637/jss.v107.i04. URL https://www.jstatsoft.org/index.php/jss/article/view/v107i04.

Chau J (2022). ***rrapply**: Revisiting Base Rapply.* R package version 1.2.6, URL https://CRAN.R-project.org/package=rrapply.

Croissant Y, Millo G (2008). "Panel Data Econometrics in R: The **plm** Package." *Journal of Statistical Software*, **27**(2), 1–43. doi:10.18637/jss.v027.i02.

Dowle M, Srinivasan A (2023). ***data.table**: Extension of 'data.frame'.* R package version 1.14.8, URL https://CRAN.R-project.org/package=data.table.

Harris CR, *et al.* (2020). "Array programming with **NumPy**." *Nature*, **585**, 357–362. doi:10.1038/s41586-020-2649-2.

Hester J, Vaughan D (2023). ***bench**: High Precision Timing of R Expressions.* R package version 1.1.3, URL https://CRAN.R-project.org/package=bench.

Hyndman RJ, Fan Y (1996). "Sample Quantiles in Statistical Packages." *American Statistician*, pp. 361–365.

JuliaStats (2023). "**StatsBase.jl**: Julia Package for Basic Statistics." URL https://github.com/JuliaStats/StatsBase.jl.

Krantz S (2024). ***fastverse**: A Suite of High-Performance Packages for Statistics and Data Manipulation.* R package version 0.3.4, URL https://fastverse.github.io/fastverse/.

Larmarange J (2023). ***labelled**: Manipulating Labelled Data.* R package version 2.12.0, URL https://CRAN.R-project.org/package=labelled.

Lumley T (2004). "Analysis of Complex Survey Samples." *Journal of Statistical Software*, **9**(1), 1–19. R package verson 2.2.

Müller K, Wickham H (2023). ***tibble**: Simple Data Frames.* R package version 3.2.1, URL https://CRAN.R-project.org/package=tibble.

pandas Development Team (2023). "pandas-dev/pandas: **pandas**." doi:10.5281/zenodo.10426137. URL https://doi.org/10.5281/zenodo.10426137.

Papadakis M, *et al.* (2023). ***Rfast**: A Collection of Efficient and Extremely Fast R Functions.* R package version 2.1.0, URL https://CRAN.R-project.org/package=Rfast.

Pebesma E (2018). "Simple Features for R: Standardized Support for Spatial Vector Data." *The R Journal*, **10**(1), 439–446. doi:10.32614/RJ-2018-009. URL https://doi.org/10.32614/RJ-2018-009.

R Core Team (2023). *R: A Language and Environment for Statistical Computing.* R Foundation for Statistical Computing, Vienna, Austria. URL https://www.R-project.org/.

Ryan JA, Ulrich JM (2023). ***xts**: eXtensible Time Series.* R package version 0.13.1, URL https://CRAN.R-project.org/package=xts.

Signorell A (2023). ***DescTools**: Tools for Descriptive Statistics.* R package version 0.99.52, URL https://CRAN.R-project.org/package=DescTools.

StataCorp LLC (2023). *STATA Statistical Software: Release 18.* College Station, TX. URL https://www.stata.com.

Vink R, *et al.* (2023). "pola-rs/polars: Python **polars** 0.20.2." doi:10.5281/zenodo.10413093. URL https://doi.org/10.5281/zenodo.10413093.

Wang E, *et al.* (2020). "A New Tidy Data Structure to Support Exploration and Modeling of Temporal Data." *Journal of Computational and Graphical Statistics*, **29**(3), 466–478. doi:10.1080/10618600.2019.1695624. URL https://doi.org/10.1080/10618600.2019.1695624.

Wes McKinney (2010). "Data Structures for Statistical Computing in Python." In Stéfan van der Walt, Jarrod Millman (eds.), *Proceedings of the 9th Python in Science Conference*, pp. 56 – 61. doi:10.25080/Majora-92bf1922-00a.

Wickham H (2007). "Reshaping Data with the **reshape** Package." *Journal of Statistical Software*, **21**(12), 1–20. URL http://www.jstatsoft.org/v21/i12/.

Wickham H (2014). "Tidy Data." *Journal of Statistical Software*, **59**(10), 1–23. doi:10.18637/jss.v059.i10. URL https://www.jstatsoft.org/index.php/jss/article/view/v059i10.

Wickham H, Henry L (2023). ***purrr**: Functional Programming Tools.* R package version 1.0.1, URL https://CRAN.R-project.org/package=purrr.

Wickham H, *et al.* (2019). "Welcome to the **tidyverse**." *Journal of Open Source Software*, **4**(43), 1686. doi:10.21105/joss.01686.

Wickham H, *et al.* (2023a). ***dplyr**: A Grammar of Data Manipulation.* R package version 1.1.2, URL https://CRAN.R-project.org/package=dplyr.

Wickham H, *et al.* (2023b). ***tidyr**: Tidy Messy Data.* R package version 1.3.0, URL https://CRAN.R-project.org/package=tidyr.

Affiliation:

Sebastian Krantz
Kiel Institute for the World Economy
Haus Welt-Club
Düsternbrooker Weg 148
24105 Kiel, Germany
E-mail: `sebastian.krantz@ifw-kiel.de`