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] Its core functionality includes a rich set of S3 generic (grouped, 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 powerful data manipulation functions—such as vectorized and verbose hash-joins or fast (aggregation, recast) pivots—functions and classes for indexed (time-aware) computations on time series and panel data, recursive tools to deal with nested data, and advanced descriptive statistical tools. This functionality is supported by efficient algorithms for grouping, ordering, deduplication, and matching callable at both R and C levels. Lastly, it includes efficient object converters, functions for memory efficient R programming, such as (grouped) transformation and math by reference, and helper functions to handle variable labels, attributes, and missing data. **collapse** is class-agnostic, providing statistical operations for 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 also globally and interactively configurable, allowing changing defaults of key function arguments, such as the `na.rm` arguments to statistical functions or `sort` arguments to grouping algorithms (default `TRUE`), and modifying the namespace to mask equivalent but less performant base R/**tidyverse** functions.[2]

---

[1] Website: https://sebkrantz.github.io/collapse/. Linecount (v2.1.2): R: 13,785, C: 19,013, C++: 9,844. Exported namespace: 392 objects, of which 238 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; and (3) create a new foundation package for statistics and data transformation that implements successful ideas in the R ecosystem and other programming environments in a stable, high performance, and broadly compatible manner.[3]

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 2025) provides an enhanced high-performance data frame with parsimonious 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 2025) and **Rfast** (Papadakis *et al.* 2023) offer fast statistical calculations along the rows and columns of matrices as well as faster statistical procedures. **DescTools** (Signorell 2023) provides a wide variety of descriptive statistics, including weighted versions. **survey** (Lumley 2004) offers computations on complex surveys. **labelled** (Larmarange 2023) provides tools to deal with labelled data. Packages like **tidyr** (Wickham *et al.* 2024), **purrr** (Wickham and Henry 2023) and **rrapply** (Chau 2022) provide functions to deal with nested or messy data.

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 specialized statistics.

In summary, **collapse** is a new foundation package for advanced statistical computing and data transformation in R that integrates seamlessly with the ecosystem and offers outstanding speed and memory efficiency. Thus, many core tasks can be done with **collapse**, and easily extended by specialized packages, yielding more lightweight, faster, and shorter R programs.

---

[3]Such ideas include **tidyverse** syntax, vectorized aggregations (**data.table**), data transformation by reference (Python, **pandas**), vectorized and verbose joins (**polars**, STATA: StataCorp LLC. (2023)), indexed time series and panel data (**xts**, **plm**), summary statistics for panel data (STATA), variable labels (STATA), recast pivots (**reshape2**), etc...

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** (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,[4] 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 useful 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.[5] 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, 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")
```

This article cannot 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, including the vignettes/cheatsheet/blog/talk + slides.

---

[4] Such as weighted statistics, including several (weighted) quantile and mode estimators, support for fully time-aware computations on irregular series/panels, scaling and centering, advanced (grouped, weighted, panel-decomposed) descriptive statistics etc., all supporting missing values and vectors/matrices/data frames.

[5] 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 very "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.

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.[6] Specifically, operations are vectorized across columns and groups, and may also involve weights or transformations of the input data. Their 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 with `fmean()` demonstrate their basic usage on the familiar `iris` dataset recording 50 measurements of 4 variables for 3 species of iris flowers.[7] 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(num_vars(iris))
```

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

```
R> identical(fmean(num_vars(iris)), fmean(as.matrix(num_vars(iris))))
```

```
[1] TRUE
```

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

```
    setosa versicolor  virginica
     5.006      5.936      6.588
```

```
R> fmean(num_vars(iris), g = iris$Species, nthreads = 4)
```

	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

---

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

[7]`num_vars()` returns the numeric variables in a data frame-like object; `cat_vars()` the categorical ones.

[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.

2.1. Transformations

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.

For example, option `TRA = "fill"` replaces elements with the corresponding statistics:

```
R> fmean(iris$Sepal.Length, g = iris$Species, TRA = "fill")[c(1:5, 51:55)]

 [1] 5.006 5.006 5.006 5.006 5.006 5.936 5.936 5.936 5.936 5.936
```

Additionally, a `set` argument can be passed to *Fast Statistical Functions* to toggle transformation by reference. For example `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. It also avoids the need to convert atomic objects to data frames for grouped aggregations or transformations. The *Fast Statistical Functions* are complemented by a smaller set of *Data Transformation Functions*, including `TRA()`, infix functions such as `%r+%`, `%c+%`, `%+=%` and `setop()` for row/column-wise arithmetic operations (by reference), `dapply()` to apply functions across rows or columns of matrices/data frames, `BY()` for general split-apply-combine computing, and specialized functions such as `fscale()` or `f[hd]within()` for (grouped, weighted) scaling and centering:

```
R> BY(num_vars(iris), g = iris$Species, FUN = mad) |> head(1)

       Sepal.Length Sepal.Width Petal.Length Petal.Width
setosa       0.2965      0.3706       0.1483           0

R> fscale(iris$Sepal.Length, g = iris$Species) |> fsd(iris$Species)

    setosa versicolor  virginica
         1          1          1
```

---

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

3. Integration with data manipulation functions

collapse also provides a broad set of Fast Data Manipulation Functions, including `fselect()`, `fsubset()`, `fslice()`, `fgroup_by()`, `fsummarise()`, `fmutate()`, `frename()`, `fcount()`, etc. as optimized analogues to base R/**tidyverse** functions. These are integrated with the *Fast Statistical and Transformation Functions* to enable vectorized operations in a familiar data frame oriented and **tidyverse**-like workflow. I illustrate this using the included World Development Dataset (`wlddev`) recording five key indicators for 216 economies and years 1960-2020.

```
R> fndistinct(wlddev)
```

```
country    iso3c     date     year  decade  region  income    OECD    PCGDP
    216      216       61       61       7       7       4       2     9470
LIFEEX     GINI      ODA      POP
 10548      368     7832    12877
```

Below, I track changes in Life Expectancy at the country level, computing the range, average change per year, and correlation with GDP per Capita in log-differences[10] while also saving the latest measurements.[11] The `collap()` function called at the end to aggregate the country-level statistics across income groups is a convenience function for mixed-type aggregation.[12] By default, it uses `fmean()` for numeric data and `fmode()` for categorical data, here also applying population weights such that the most populous country in each group is kept.

```
R> wlddev |> fgroup_by(country, income) |>
+    fsummarise(n = fnobs(LIFEEX),
+               diff = fmax(LIFEEX) %-=% fmin(LIFEEX),
+               mean_diff = fmean(fdiff(LIFEEX)),
+               cor_PCGDP = pwcor(Dlog(LIFEEX), Dlog(PCGDP)),
+               across(c(LIFEEX, PCGDP, POP), flast)) |>
+    collap( ~ income, w = ~ POP, keep.w = FALSE) |> print(digits = 2)
```

```
        country              income  n diff mean_diff cor_PCGDP LIFEEX PCGDP
1 United States         High income 60   13      0.23  -0.00828     81 44599
2      Ethiopia          Low income 60   26      0.43   0.27153     64   691
3         India Lower middle income 60   24      0.41  -0.15366     69  2324
4         China Upper middle income 60   27      0.45   0.00077     76  8749
```

With the exception of `pwcor(Dlog(LIFEEX), Dlog(PCGDP))`, which is evaluated for every country, all other expressions in `fsummarise()` and also the final `collap()` call are fully vectorized, i.e., evaluated only once with grouping information passed directly to the g arguments of *Fast Statistical and Transformation Functions*. Using subtraction by reference in the range is thus memory efficient as the executed expression is equivalent to `fmax(LIFFEX, country)`

---

[10]`Dlog()` is a shorthand for `fdiff(log(x))` in **collapse**, `pwcor()` wraps `cor(..., use = "pairwise.complete.obs")`.

[11]Note that `fsummarise()` never re-groups data, making a call to `fungroup()` redundant.

[12]See also this blog post on aggregating survey data using **collapse** which showcases more aspects of the `collap()` function using real census data.

`%-=% fmin(LIFFEX, country)`. `across()` also directly invokes `flast.data.frame()` on the subset of columns. *Fast Statistical Functions* also have a method for grouped data, hence `fsummarise()` is not always needed. Below, I average the data with population weights.[13]

```
R> wlddev |> fsubset(year >= 2015, income, PCGDP:POP) |>
+    fgroup_by(income) |> fmean(POP)


              income   sum.POP PCGDP LIFEEX  GINI         ODA
1         High income 5.901e+09 43340  80.70 36.14    81398948
2          Low income 3.296e+09   663  63.05 39.13  2466732035
3 Lower middle income 1.490e+10  2177  68.31 36.48  2234540565
4 Upper middle income 1.318e+10  8168  75.51 41.68   -86630117
```

Using the *Fast Statistical Functions* directly often results in more efficient code. For example, computing population shares for each year as `fmutate(wlddev, POP_share = fsum(POP, year, TRA = "/"))` is considerably more efficient than `wlddev |> fgroup_by(year) |> fmutate(POP_share = fsum(POP, TRA = "/")) |> fungroup()`, which in turn is more efficient than `POP / fsum(POP)` (extra allocation) or `proportions(POP)` (split-apply-combine).

collapse thus offers several ways to reach the same outcome. This may be confusing, but all options can be distinguished in efficiency terms. It is thus useful to be mindful of computer resources and attempt to craft minimalistic solutions. For example, I recently combined multiple spatial datasets on points of interest (POIs) and needed to deduplicate them. 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 ended up being a single line of the form `fsubset(data, 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. This is very efficient because it avoids materializing intermediate datasets and relegates all computations to `fmode()`.

3.1. Vectorizations for advanced tasks

`fsummarise()` and `fmutate()` follow an eager vectorization approach, implying that using any *Fast Statistical Function* in an expression causes the whole expression to be vectorized. This only applies to visible expressions, **collapse** cannot read the contents of custom functions, thus such functions are always evaluated split-apply-combine. The eager vectorization approach enables efficient grouped calculation of more complex statistics. Below, I forecast the population of each region via linear regression (`POP ~ year`) in a fully vectorized way.

```
R> wlddev |> collap(POP ~ region + year, FUN = fsum) |>
+    fmutate(POP = POP / 1e6) |> fgroup_by(region) |>
+    fmutate(dmy = fmean(year, TRA = "-")) |>
+    fsummarise(beta = fsum(POP, dmy) %/=% fsum(dmy, dmy),
+               POP20 = flast(POP)) |>
+    fmutate(POP21 = POP20 + beta, POP22 = POP21 + beta,
+            POP23 = POP22 + beta, POP24 = POP23 + beta)
```

---

[13]Functions like `fmean()`, when called on a grouped data frame with weights, by default (`keep.w = TRUE`) sum the weights column and retain it after the grouping columns to permit further weighted operations.

```
                      region   beta  POP20   POP21   POP22   POP23   POP24
1          East Asia & Pacific 18.731 2291.4 2310.2 2328.9 2347.6 2366.4
2      Europe & Central Asia  2.463  921.2  923.7  926.1  928.6  931.0
3  Latin America & Caribbean  6.460  646.4  652.9  659.4  665.8  672.3
4 Middle East & North Africa  5.409  456.7  462.1  467.5  472.9  478.3
5              North America  2.301  365.9  368.2  370.5  372.8  375.1
6                 South Asia 19.640 1835.8 1855.4 1875.1 1894.7 1914.3
7         Sub-Saharan Africa 12.852 1103.5 1116.3 1129.2 1142.0 1154.9
```

When `fsummarise()` evaluates an expression involving *Fast Statistical Functions*, it sets their g argument with a grouping ('GRP') object that is directly handed to C/C++, and also sets `use.g.names = FALSE`. Hence, weights (`w`) becomes the second positional argument. Similarly, `fmutate()` sets g and `TRA = "fill"`, which can be overwritten by the user (here with `TRA = "-"`). The expression `fsum(x, dmy) %/=% fsum(dmy, dmy)`, amounting to `cov(x, y)/var(y)`, is thus 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. In a 2023 blog post, I forecasted high-resolution population estimates for South Africa like this. Using 1 km^2 WorldPop data available for years 2014-2020, running 1.6 million cell-regressions and obtaining 2 forecasts for 2021 and 2022 took only 0.3 seconds on my M1 Mac. 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 300,000 groups for a simulation study, yielding a 70x performance increase over **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).[14]

```
R> wlddev |> fsubset(is.finite(POP)) |> fgroup_by(region) |>
+   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),
+             Q3 = fnth(LIFEEX, 0.75, POP, o = o, ties = "q8"),
+             max = fmax(LIFEEX))
```

```
                      region   min    Q1  mean median    Q3   max
1          East Asia & Pacific 18.91 65.28 68.45  69.67 73.86 85.08
2      Europe & Central Asia 45.37 68.68 72.30  71.58 76.67 85.42
3  Latin America & Caribbean 41.76 65.17 69.16  70.87 74.48 82.19
4 Middle East & North Africa 29.92 61.96 66.65  69.12 72.64 82.80
5              North America 68.90 73.57 75.54  75.62 78.38 82.05
6                 South Asia 32.45 55.08 60.19  62.00 66.67 78.92
7         Sub-Saharan Africa 26.17 46.51 52.53  52.23 58.32 74.51
```

Weighted quantiles have a sub-column parallel implementation,[15] and, as shown above, can

---

[14]**collapse** computes weighted quantiles in a theoretically consistent way, see fquantile for details.

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

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.

3.2. Grouping objects and lower-level API

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` for efficient factor generation) or 'GRP' objects as inputs. The latter contain all information **collapse**'s statistical functions may require to operate across groups and are thus passed to internal C/C++ code without checks. 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 two columns in the 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. I can employ the object with the *Fast Statistical Functions* and some utilities[16] 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),
+   get_vars(wlddev, "POP") |> fsum(g, use = FALSE))
```

---

[16]`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.

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

The above is equivalent to `collap(wlddev, country + PCGDP + LIFEEX ~ income + OECD, w = ~ POP)`, which internally toggles many of the same function calls.

Similarly, data can be transformed, here using `fwithin()` to level average differences in economic status, adding back the overall mean after subtracting out group means:[17]

```
R> add_vars(wlddev) <- get_vars(wlddev, c("PCGDP", "LIFEEX")) |>
+       fwithin(g, mean = "overall.mean") |> add_stub("center_")
```

The low-level API is useful for package development and standard-evaluation programming, as further elucidated in the vignette on developing with **collapse**. Users should note that **collapse** does not provide metaprogramming capabilities in its non-standard evaluation functions—such as quosures or indirection familiar to **tidyverse** users. Instead, there exist standard-evaluation equivalents to some of these functions which typically end with a `v` (for 'variables'), such as `collapv()`, `fslicev()`, `ftransformv()`, etc. In other cases, such as those typically handled by `fsummarise()` or `fmutate()`, users may need to use the low-level API for programming, or resort to `substitute()` and friends. The main reason for not providing higher-level metaprogramming capabilities is to keep all functions as simple as possible. It also compels users to think deeper about their programs and devise more efficient solutions.

These metaprogramming restrictions are also vindicated by internally complex data manipulation functions producing greater overheads that can noticeably add up in larger scripts:[18]

```
R>  bmark(collapse = collapse::fsummarise(mtcars, sum = sum(mpg)),
+          dplyr = dplyr::summarise(mtcars, sum = sum(mpg)))
```

```
  expression       min    median mem_alloc n_itr n_gc total_time
1   collapse   3.61µs    4.26µs   14.33KB  9999    1    43.81ms
2      dplyr 211.27µs 240.26µs    3.92MB  7651   52       1.86s
```

Grouped programming using 'GRP' objects and *Fast Statistical Functions* is also particularly powerful with vectors and matrices. 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()`.[19]

---

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

[18]`bmark()` is a slim wrapper around `bench::mark()`. See the Computational details section.

[19]Another 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.

4. Time series and panel series

collapse also provides a flexible high-performance architecture to perform time aware computations on time series and panel series. In particular, users can either apply time series and panel data transformations by passing individual and/or time identifiers to the respective functions in an ad-hoc fashion, or by using '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, 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.[20] But first, a basic example:

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

```
Time Series:
Start = 1937
End = 1960
Frequency = 1
```

---

[20]This is premised on the observation that the most common form of temporal identifier is a plain numeric vector representing 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'.

```
 [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
```

To illustrate the full capabilities of these time series functions, I generate a sector-level trade dataset of export values (v) by country (c), sector (s), and year (y). Like many detailed trade datasets, it is unbalanced—not all sectors/products are exported by country c in all years.

```
R> set.seed(101)
R> exports <- expand.grid(y = 2001:2010, c = paste0("c", 1:10),
+    s = paste0("s", 1:10)) |> fmutate(v = abs(rnorm(1e3))) |>
+    colorder(c, s) |> fsubset(-sample.int(1e3, 500))
```

The following extracts one country-sector series from the `exports` dataset. It is irregular, missing years 2003 and 2006.[21] Indexation using the `t` argument then allows for correct (time-aware) computations in this context without the need to 'expand' the data/fill gaps.

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

```
[1] 2001 2002 2004 2005 2007 2008 2009 2010
```

```
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 associated 'operators' `L()`/`D()`/`G()` to 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. Below, I demonstrate two ways to compute a sequence of lagged growth rates using either `G()` or `fgrowth()` and `tfm()`—a shorthand for `ftransform()`.[22]

```
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 2002 -18.15 0.5525    NA     NA
2 c1 s1 2003 214.87 0.6749 22.17     NA
3 c1 s1 2004 -31.02 0.2144 -68.24  -61.2
```

---

[21]`%=%` is an infix operator for the `massign()` function in **collapse** which is a multivariate version of `assign()`.
[22]Several 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, ensuring that the 'list' method applies.

```
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 2002 0.5525 -18.15     NA     NA
2 c1 s1 2003 0.6749 214.87  22.17     NA
3 c1 s1 2004 0.2144 -31.02 -68.24  -61.2
```

These functions and operators are also integrated with `fgroup_by()` and `fmutate()` for vectorized computations. But, as mentioned earlier, 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 2002 0.5525     NA
2 c1 s1 2003 0.6749  22.17
3 c1 s1 2004 0.2144 -68.24
4 c1 s1 2005 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 behave 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)
```

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.[23] 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 the time-based computations.

An example using the exports data follows. Note that data need not be sorted for indexing.

```
R> exportsi <- exports |> findex_by(c, s, y)
R> exportsi |> G() |> head(5)


   c  s    y    G1.v
1 c1 s1 2002      NA
2 c1 s1 2003  22.17
3 c1 s1 2004 -68.24
4 c1 s1 2005  44.98
5 c1 s1 2006 277.76

Indexed by:  c.s [1] | y [5 (10)]

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


      c.s    y
1   c1.s1 2002
2   c1.s1 2003
---
499 c10.s10 2007
500 c10.s10 2009

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 1 ..
   ..$ y  : Ord.factor w/ 10 levels "2001"<"2002"<..: 2 3 4 5 6 7 8 9..
   ..- attr(*, "nam")= chr [1:3] "c" "s" "y"

R> is_irregular(vi)

[1] TRUE
```

---

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

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

```
      2001  2002  2003  2004  2005 2006  2007    2008  2009  2010
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)
```

```
      2001 2002   2003   2004   2005  2006    2007   2008  2009   2010
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
```

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)]
```

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 (StataCorp LLC. 2023), the implementation of joins in most open-source software 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 many 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, verbose = 1,
  drop.dup.cols = FALSE, require = NULL, column = NULL, attr = NULL, ...)
```

It defaults to left join and only takes first matches from y (`multiple = FALSE`), thus, 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 with 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' produces an error if the key is not unique.

```
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
```

Similarly, the `require` argument allows users to demand a minimum matching success rate.

```
R> join(teacher, course, on = "id", require = list(x = 0.8)) |>
+    tryCatch(error = function(e) substr(e, 102, 200) |> cat())

Matched 75.0% of records in table teacher (x), but 80.0% is required
```

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:

```
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.[24] The join relationship is indicated inside the `<>` as the number of records joined from each table divided by the number of unique matches.

---

[24]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 remove duplicate columns from either 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 hashed jointly.

`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 functions.

`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, FUN = "last", ...)
```

The demonstration below employs the included Groningen Growth and Development Centre 10-Sector Database (`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.[25]

```
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 to 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.

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.[26] Note below

---

[25]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.

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

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`, `drop`, `sort`, and `transpose` can be used to control the casting process/output.

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

```
  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 needs 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 = "recast"))
```

```
  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
```

```
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 representation. 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 = "recast", 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
```

The documentation examples demonstrate more features of `pivot()`. 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, for example, agricultural employment in Argentina can be accessed as:[27]

---

[27]If a nested structure is not needed, `flatten = TRUE` lets `rsplit()` operate like a faster version of `split()`.

```
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 and 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.[28] However, there are limits to such 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
```

---

[28]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 keep 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, `unlist2d()` provides 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. Further useful functions include `has_elem()` to check for the existence of

elements, `ldepth()` to return the maximum level of recursion, `is_unlistable()` to check whether a list has atomic elements in all final nodes, `[ir]reg_elem()` to recursively extract the (non-)atomic elements, and `rapply2d()` to apply functions to nested lists of data objects.

7. Summary statistics

collapse's summary statistics functions offer a parsimonious yet 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 dataset (`wlddev`). 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 `fbetween()`/ `fwithin()` supporting 'indexed_series' (see 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 compute this decomposition, considering the group-means instead of the between transformation[29] and adding the overall 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 and with sampling weights. `qsu()` also has a data frame method, and by default computes simple statistics:[30]

---

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

[30]The `pid` argument to `qsu()` can also be used to pass identifiers for panel-decomposition, e.g., `pid = iso3c`. With indexed data, it is automatically set to the first column in the index (`effect = 1`).

```
R> qsu(wlddev, by = LIFEEX ~ income, w = ~ POP)
```

	N	WeightSum	Mean	SD	Min	Max
High income	3828	5.87959699e+10	75.6926	4.5346	42.672	85.4171
Low income	1792	2.09491614e+10	53.5061	8.8691	26.172	74.43
Lower middle income	2790	1.13837685e+11	60.5865	8.3607	18.907	76.699
Upper middle income	3249	1.19295269e+11	68.2698	7.1889	36.535	80.279

For greater detail, `descr()` provides a rich (grouped, weighted) statistical description. It does not support panel-variance decompositions like `qsu()`, but also computes detailed (grouped, weighted) frequency tables for categorical data and is thus very useful with mixed-type data.

```
R> wlda15 <- wlddev |> fsubset(year >= 2015) |> fgroup_by(iso3c) |> flast()
R> wlda15 |> descr(by = 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

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
--------------------------------------------------------------------------------
```

It also has a `stepwise` argument to describe one variable at a time, allowing users to naturally 'click-through' the variables in a large dataset while spreading the computational burden. The

documentation provides more details and examples. Both `qsu()` and `descr()` come with an `as.data.frame()` method for efficient tidying and easy 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 values 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.* 2023): 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_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.

Such global powers confer responsibilities upon package developers, as further elucidated in the vignette on developing with **collapse**. As a general rule, options `mask` and `remove` should be off-limits inside packages, and other options need to be reset immediately using `on.exit()`.

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 a 2024 Apple MacBook Pro with 48GB M4 Pro chip. 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 41.49ms   43.2ms   166.2MB    47   47      2.04s
2 collapse_int  5.47ms  9.28ms    38.2MB   220   44      2.01s

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  68.7ms   70ms   166.2MB    28   28      2.01s
2 collapse_char 13.8ms 15.7ms    38.2MB   123   21      2.01s

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 15.13ms 15.58ms    76.3MB    38   67   608.03ms
2 collapse_int  5.47ms  7.74ms    38.2MB   196   49      1.43s
```

```
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 60.8ms 62.4ms   114.5MB    32   32      2.07s
2 data.table_char 31.9ms   34ms    38.1MB    58   18      2.02s
3   collapse_char 19.6ms 21.8ms    38.1MB    90   18      2.01s
```

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    5.67ms   6.21ms   7.86KB    324    0         2s
2 collapse 280.93µs 321.64µs   7.86KB   6143    1         2s


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


  expression      min   median mem_alloc n_itr n_gc total_time
1        R    6.01ms   6.21ms  32.09KB    324    0         2s
2 collapse 278.92µs 322.14µs   7.86KB   6164    0         2s


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


  expression    min median mem_alloc n_itr n_gc total_time
1        MS 78.7ms 79.7ms   238.9KB    26    0      2.07s
2   collapse 19.4ms 19.6ms    27.2KB   101    0      2.02s
```

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    5.36ms 5.59ms   7.87MB    319   20      1.79s
2   collapse 923.24µs 1.14ms   7.71MB   1279   60      1.58s
```

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    1255      14    4840   26207

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	437.44ms	471.82ms	50.5MB	5	50	2.35s
2	data.table	7.56ms	8.59ms	20.8MB	208	18	2s
3	collapse	3ms	3.75ms	9.2MB	467	21	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 becomes even more costly.

	expression	min	median	mem_alloc	n_itr	n_gc	total_time
1	dplyr_mean	1.17s	1.18s	48.57MB	2	42	2.36s
2	data.table_mean	7.64ms	8.79ms	18.93MB	190	18	2s
3	collapse_mean	3.28ms	3.77ms	9.11MB	470	17	2.01s

	expression	min	median	mem_alloc	n_itr	n_gc	total_time
1	dplyr_median	4.05s	4.05s	52.8MB	1	86	4.05s
2	data.table_median	21.45ms	22.51ms	18.9MB	86	8	2.01s
3	collapse_median	9.7ms	10.34ms	11.1MB	171	11	2s

So far, **data.table**, by virtue of it's internal vectorizations (also via dedicated grouped C implementations of simple functions), is competitive.[31] Below, I compute the range of one column (x) using max(x) - min(x). As elucidated in Section 3, this expression is also vectorized in **collapse**, where it amounts to fmax(x, g) - fmin(x, g), but not in **data.table**.

```
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)))
```

---

[31] 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.

```
  expression     min  median mem_alloc n_itr n_gc total_time
1      dplyr 90.25ms 99.01ms   20.18MB    20   44      2.09s
2 data.table 59.54ms 63.14ms    5.77MB    32   27      2.04s
3   collapse  5.32ms  5.88ms     6.8MB   303   11        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] 434526/435352 (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] 435352/435352 (100%) <3689.42:1st> airports[faa] 118/1255
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**:[32]

```
        expression     min  median mem_alloc n_itr n_gc total_time
1      dplyr_joins 140.85ms 173.7ms    559.4MB    12   28      2.12s
2 data.table_joins  99.45ms 132.9ms      491MB    15   36      2.11s
3   collapse_joins   9.15ms  11.2ms     89.7MB   142   40      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.[33]

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

```
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 58.2ms 58.2ms      255MB     1   27     58.2ms
2 data.table 41.9ms 43.9ms      209MB    14   22    613.6ms
3   collapse   13ms 15.3ms      209MB    34   57    512.9ms
```

---

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

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

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 353.82ms 383.22ms     143MB     5   37         2s
2     data.table 222.83ms 230.79ms    21.7MB     9   40      2.08s
3  collapse_fsum   5.04ms   7.01ms    39.1MB   233   23      2.07s
4  collapse_itnl   4.12ms   4.51ms    12.4MB   392    8      2.01s
```

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 show 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 personal computers.

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.[34] 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 → 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 3.5.0 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.

Last but not least, **collapse** users are encouraged to also familiarize themselves with the **fastverse**,[35] a suite of complementary high-performance packages for statistical computing and data manipulation in R which offer more advanced tools in several statistical computing domains. The **fastverse** metapackage also provides a lightweight framework to jointly load and manage these packages, as well as to build customized and fully separate package verses.

---

[34]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.

[35]Website: https://fastverse.github.io/fastverse/

Computational details

The results in this paper were obtained using R (R Core Team 2025) 4.4.3 with **collapse** 2.1.2, **data.table** 1.17.0, **dplyr** 1.1.4, **tidyr** 1.3.1, **matrixStats** 1.5.0, **fastverse** 0.3.4, **nycflights23** 0.2.0 (Ismay, Couch, and Wickham 2024), and **bench** 1.1.4 (Hester and Vaughan 2025). 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 MacBook Pro (2024) with a 48GB M4 Pro processor (single core speed ∼4.4 GHz). Packages were compiled from source using Apple Clang version 17.0.0 with OpenMP enabled and the -O2 flag.

The `bmark()` function used for benchmarking is defined as follows:

```
bmark <- function(...) {
  bench::mark(..., min_time = 2, check = FALSE) |>
    janitor::clean_names() |>
    fselect(expression, min, median, mem_alloc, n_itr, n_gc, total_time) |>
    fmutate(expression = names(expression)) |>
    dapply(as.character) |> qDF()
}
```

Acknowledgments

References

Bengtsson H (2025). **matrixStats**: *Functions that Apply to Rows and Columns of Matrices (and to Vectors)*. R package version 1.5.0, 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 (2025). **data.table**: *Extension of 'data.frame'*. R package version 1.17.0, 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 (2025). **bench**: *High Precision Timing of R Expressions*. R package version 1.1.4, URL https://CRAN.R-project.org/package=bench.

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

Ismay C, Couch SP, Wickham H (2024). **nycflights23**: *Flights and Other Useful Metadata for NYC Outbound Flights in 2023*. R package version 0.2.0, URL https://CRAN.R-project.org/package=nycflights23.

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.

McKinney W (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.

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 (2025). 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`.

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.* (2023). ***dplyr****: A Grammar of Data Manipulation*. R package version 1.1.4, URL `https://CRAN.R-project.org/package=dplyr`.

Wickham H, *et al.* (2024). ***tidyr****: Tidy Messy Data*. R package version 1.3.1, 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