Package 'data.table'

December 24, 2025

Version 1.18.0

Title Extension of `data.frame`

Depends R (>= 3.4.0)

Imports methods

Suggests bit64 (>= 4.0.0), bit (>= 4.0.4), R.utils (>= 2.13.0), xts, zoo (>= 1.8-1), yaml, knitr, markdown

Description Fast aggregation of large data (e.g. 100GB in RAM), fast ordered joins, fast add/modify/delete of columns by group using no copies at all, list columns, friendly and fast character-separated-value read/write. Offers a natural and flexible syntax, for faster development.

License MPL-2.0 | file LICENSE

URL https://r-datatable.com, https://Rdatatable.gitlab.io/data.table, https://github.com/Rdatatable/data.table

BugReports https://github.com/Rdatatable/data.table/issues

VignetteBuilder knitr

Encoding UTF-8

ByteCompile TRUE

NeedsCompilation yes

Author Tyson Barrett [aut, cre] (ORCID: <https://orcid.org/0000-0002-2137-1391>),
Matt Dowle [aut],
Arun Srinivasan [aut],
Jan Gorecki [aut],
Michael Chirico [aut] (ORCID: <https://orcid.org/0000-0003-0787-087X>),
Toby Hocking [aut] (ORCID: <https://orcid.org/0000-0002-3146-0865>),
Benjamin Schwendinger [aut] (ORCID: <https://orcid.org/0000-0003-3315-8114>),
Ivan Krylov [aut] (ORCID: <https://orcid.org/0000-0002-0172-3812>),
Pasha Stetsenko [ctb],
Tom Short [ctb],
Steve Lianoglou [ctb],

Eduard Antonyan [ctb],
Markus Bonsch [ctb],
Hugh Parsonage [ctb],
Scott Ritchie [ctb],
Kun Ren [ctb],
Xianying Tan [ctb],
Rick Saporta [ctb],
Otto Seiskari [ctb],
Xianghui Dong [ctb],
Michel Lang [ctb],
Watal Iwasaki [ctb],
Seth Wenchel [ctb],
Karl Broman [ctb],
Tobias Schmidt [ctb],
David Arenburg [ctb],
Ethan Smith [ctb],
Francois Cocquemas [ctb],
Matthieu Gomez [ctb],
Philippe Chataignon [ctb],
Nello Blaser [ctb],
Dmitry Selivanov [ctb],
Andrey Riabushenko [ctb],
Cheng Lee [ctb],
Declan Groves [ctb],
Daniel Possenriede [ctb],
Felipe Parages [ctb],
Denes Toth [ctb],
Mus Yaramaz-David [ctb],
Ayappan Perumal [ctb],
James Sams [ctb],
Martin Morgan [ctb],
Michael Quinn [ctb],
@javrucebo [ctb] (GitHub user),
Marc Halperin [ctb],
Roy Storey [ctb],
Manish Saraswat [ctb],
Morgan Jacob [ctb],
Michael Schubmehl [ctb],
Davis Vaughan [ctb],
Leonardo Silvestri [ctb],
Jim Hester [ctb],
Anthony Damico [ctb],
Sebastian Freundt [ctb],
David Simons [ctb],
Elliott Sales de Andrade [ctb],
Cole Miller [ctb],
Jens Peder Meldgaard [ctb],
Vaclav Tlapak [ctb],

Kevin Ushey [ctb],
Dirk Eddelbuettel [ctb],
Tony Fischetti [ctb],
Ofek Shilon [ctb],
Vadim Khotilovich [ctb],
Hadley Wickham [ctb],
Bennet Becker [ctb],
Kyle Haynes [ctb],
Boniface Christian Kamgang [ctb],
Olivier Delmarcell [ctb],
Josh O'Brien [ctb],
Dereck de Mezquita [ctb],
Michael Czekanski [ctb],
Dmitry Shemetov [ctb],
Nitish Jha [ctb],
Joshua Wu [ctb],
Iago Giné-Vázquez [ctb],
Anirban Chetia [ctb],
Doris Amoakohene [ctb],
Angel Feliz [ctb],
Michael Young [ctb],
Mark Seeto [ctb],
Philippe Grosjean [ctb],
Vincent Runge [ctb],
Christian Wia [ctb],
Elise Maigné [ctb],
Vincent Rocher [ctb],
Vijay Lulla [ctb],
Aljaž Sluga [ctb],
Bill Evans [ctb],
Reino Bruner [ctb],
@badasahog [ctb] (GitHub user),
Vinit Thakur [ctb],
Mukul Kumar [ctb],
Ildikó Czeller [ctb]

Maintainer Tyson Barrett <t.barrett88@gmail.com>

Repository CRAN

Date/Publication 2025-12-24 12:05:11 UTC

Contents

`data.table-package` *Enhanced data.frame*

Description

`data.table` *inherits* from `data.frame`. It offers fast and memory efficient: file reader and writer, aggregations, updates, equi, non-equi, rolling, range and interval joins, in a short and flexible syntax, for faster development.

It is inspired by `A[B]` syntax in R where `A` is a matrix and `B` is a 2-column matrix. Since a `data.table` *is* a `data.frame`, it is compatible with R functions and packages that accept *only* `data.frames`.

Type `vignette(package="data.table")` to get started. The Introduction to data.table vignette introduces `data.table`'s `x[i, j, by]` syntax and is a good place to start. If you have read the vignettes and the help page below, please read the data.table support guide.

Please check the homepage for up to the minute live NEWS.

Tip: one of the *quickest* ways to learn the features is to type `example(data.table)` and study the output at the prompt.

Usage

```
data.table(..., keep.rownames=FALSE, check.names=FALSE, key=NULL, stringsAsFactors=FALSE)

## S3 method for class 'data.table'
x[i, j, by, keyby, with = TRUE,
  nomatch = NA,
  mult = "all",
  roll = FALSE,
  rollends = if (roll=="nearest") c(TRUE,TRUE)
             else if (roll>=0) c(FALSE,TRUE)
             else c(TRUE,FALSE),
  which = FALSE,
  .SDcols,
  verbose = getOption("datatable.verbose"),                    # default: FALSE
  allow.cartesian = getOption("datatable.allow.cartesian"),    # default: FALSE
  drop = NULL, on = NULL, env = NULL,
  showProgress = getOption("datatable.showProgress", interactive())]
```

Arguments

`...`	Just as `...` in `data.frame`. Usual recycling rules are applied to vectors of different lengths to create a list of equal length vectors.
`keep.rownames`	If `...` is a `matrix` or `data.frame`, `TRUE` will retain the rownames of that object in a column named `rn`.
`check.names`	Just as `check.names` in `data.frame`.
`key`	Character vector of one or more column names which is passed to `setkey`.
`stringsAsFactors`	Logical (default is `FALSE`). Convert all `character` columns to `factors`?
`x`	A `data.table`.
`i`	Integer, logical or character vector, single column numeric `matrix`, expression of column names, `list`, `data.frame` or `data.table`. `integer` and `logical` vectors work the same way they do in `[.data.frame` except logical NAs are treated as FALSE. `expression` is evaluated within the frame of the `data.table` (i.e. it sees column names as if they are variables) and can evaluate to any of the other types. `character`, `list` and `data.frame` input to `i` is converted into a `data.table` internally using `as.data.table`. If `i` is a `data.table`, the columns in `i` to be matched against `x` can be specified using one of these ways: • `on` argument (see below). It allows for both `equi-` and the newly implemented `non-equi` joins. • If not, `x` *must be keyed*. Key can be set using `setkey`. If `i` is also keyed, then first *key* column of `i` is matched against first *key* column of `x`, second against second, etc.. If `i` is not keyed, then first column of `i` is matched against first *key* column of `x`, second column of `i` against second *key* column of `x`, etc...

This is summarised in code as `min(length(key(x)), if (haskey(i)) length(key(i)) else ncol(i))`.

Using `on=` is recommended (even during keyed joins) as it helps understand the code better and also allows for *non-equi* joins.

When the binary operator `==` alone is used, an *equi* join is performed. In SQL terms, `x[i]` then performs a *right join* by default. `i` prefixed with `!` signals a *not-join* or *not-select*.

Support for *non-equi* join was recently implemented, which allows for other binary operators `>=`, `>`, `<=` and `<`.

See `vignette("datatable-keys-fast-subset")` and `vignette("datatable-secondary-indices-a`

Advanced: When `i` is a single variable name, it is not considered an expression of column names and is instead evaluated in calling scope.

`j` When `with=TRUE` (default), `j` is evaluated within the frame of the data.table; i.e., it sees column names as if they are variables. This allows to not just *select* columns in `j`, but also `compute` on them e.g., `x[, a]` and `x[, sum(a)]` returns `x$a` and `sum(x$a)` as a vector respectively. `x[, .(a, b)]` and `x[, .(sa=sum(a), sb=sum(b))]` returns a two column data.table each, the first simply *selecting* columns `a, b` and the second *computing* their sums.

As long as `j` returns a `list`, each element of the list becomes a column in the resulting `data.table`. When the output of `j` is not a `list`, the output is returned as-is (e.g. `x[ , a]` returns the column vector `a`), unless `by` is used, in which case it is implicitly wrapped in `list` for convenience (e.g. `x[ , sum(a), by=b]` will create a column named `V1` with value `sum(a)` for each group).

The expression `.()` is a *shorthand* alias to `list()`; they both mean the same. (An exception is made for the use of `.()` within a call to `bquote`, where `.()` is left unchanged.)

When `j` is a vector of column names or positions to select (as in `data.frame`), there is no need to use `with=FALSE`. Note that `with=FALSE` is still necessary when using a logical vector with length `ncol(x)` to include/exclude columns. Note: if a logical vector with length `k < ncol(x)` is passed, it will be filled to length `ncol(x)` with `FALSE`, which is different from `data.frame`, where the vector is recycled.

Advanced: `j` also allows the use of special *read-only* symbols: `.SD`, `.N`, `.I`, `.GRP`, `.BY`. See `special-symbols` and the Examples below for more.

Advanced: When `i` is a `data.table`, the columns of `i` can be referred to in `j` by using the prefix `i.`, e.g., `X[Y, .(val, i.val)]`. Here `val` refers to `X`'s column and `i.val` `Y`'s.

Advanced: Columns of `x` can now be referred to using the prefix `x.` and is particularly useful during joining to refer to `x`'s *join* columns as they are otherwise masked by `i`'s. For example, `X[Y, .(x.a-i.a, b), on="a"]`.

See `vignette("datatable-intro")` and `example(data.table)`.

`by` Column names are seen as if they are variables (as in `j` when `with=TRUE`). The `data.table` is then grouped by the `by` and `j` is evaluated within each group. The order of the rows within each group is preserved, as is the order of the groups. `by` accepts:

- A single unquoted column name: e.g., `DT[, .(sa=sum(a)), by=x]`

- a `list()` of expressions of column names: e.g., `DT[, .(sa=sum(a)), by=.(x=x>0, y)]`
- a single character string containing comma separated column names (where spaces are significant since column names may contain spaces even at the start or end): e.g., `DT[, sum(a), by="x,y,z"]`
- a character vector of column names: e.g., `DT[, sum(a), by=c("x", "y")]`
- or of the form `startcol:endcol`: e.g., `DT[, sum(a), by=x:z]`

Advanced: When `i` is a `list` (or `data.frame` or `data.table`), `DT[i, j, by=.EACHI]` evaluates `j` for the groups in `DT` that each row in `i` joins to. That is, you can join (in `i`) and aggregate (in `j`) simultaneously. We call this *grouping by each i*. See this StackOverflow answer for a more detailed explanation until we roll out vignettes.

Advanced: In the `X[Y, j]` form of grouping, the `j` expression sees variables in `X` first, then `Y`. We call this *join inherited scope*. If the variable is not in `X` or `Y` then the calling frame is searched, its calling frame, and so on in the usual way up to and including the global environment.

`keyby` Same as `by`, but with an additional `setkey()` run on the `by` columns of the result, for convenience. It is common practice to use `keyby=` routinely when you wish the result to be sorted. May also be `TRUE` or `FALSE` when `by` is provided as an alternative way to accomplish the same operation.

`with` By default `with=TRUE` and `j` is evaluated within the frame of `x`; column names can be used as variables. In the case of overlapping variable names inside `x` and in parent scope, you can use the double dot prefix `..cols` to explicitly refer to the `cols` variable in parent scope and not from `x`.

When `j` is a character vector of column names, a numeric vector of column positions to select, or of the form `startcol:endcol`, the value returned is always a `data.table`.

New code should rarely use this argument, which was originally needed for similarity to data.frame. For example, to select columns from a character vector `cols`, in data.frame we do `x[, cols]`, which has several equivalents in data.table: `x[, .SD, .SDcols=cols]`, `x[, ..cols]`, `x[, cols, env = list(cols = I(cols))]`, or `x[, cols, with=FALSE]`.

`nomatch` When a row in `i` has no match to `x`, `nomatch=NA` (default) means `NA` is returned. `NULL` (or `0` for backward compatibility) means no rows will be returned for that row of `i`.

`mult` When `i` is a `list` (or `data.frame` or `data.table`) and *multiple* rows in `x` match to the row in `i`, `mult` controls which are returned: `"all"` (default), `"first"` or `"last"`.

`roll` When `i` is a `data.table` and its row matches to all but the last `x` join column, and its value in the last `i` join column falls in a gap (including after the last observation in `x` for that group), then:

- `+Inf` (or `TRUE`) rolls the *prevailing* value in `x` forward. It is also known as last observation carried forward (LOCF).
- `-Inf` rolls backwards instead; i.e., next observation carried backward (NOCB).
- finite positive or negative number limits how far values are carried forward or backward.

- "nearest" rolls the nearest value instead.

Rolling joins apply to the last join column, generally a date but can be any variable. It is particularly fast using a modified binary search.

A common idiom is to select a contemporaneous regular time series (`dts`) across a set of identifiers (`ids`): `DT[CJ(ids,dts),roll=TRUE]` where `DT` has a 2-column key (id,date) and `CJ` stands for *cross join*.

`rollends` A logical vector length 2 (a single logical is recycled) indicating whether values falling before the first value or after the last value for a group should be rolled as well.

- If `rollends[2]=TRUE`, it will roll the last value forward. `TRUE` by default for LOCF and `FALSE` for NOCB rolls.
- If `rollends[1]=TRUE`, it will roll the first value backward. `TRUE` by default for NOCB and `FALSE` for LOCF rolls.

When `roll` is a finite number, that limit is also applied when rolling the ends.

`which` `TRUE` returns the row numbers of `x` that `i` matches to. If `NA`, returns the row numbers of `i` that have no match in `x`. By default `FALSE` and the rows in `x` that match are returned.

`.SDcols` Specifies the columns of `x` to be included in the special symbol `.SD` which stands for `Subset of data.table`. May be character column names, numeric positions, logical, a function name such as `is.numeric`, or a function call such as `patterns()`. `.SDcols` is particularly useful for speed when applying a function through a subset of (possible very many) columns by group; e.g., `DT[, lapply(.SD, sum), by="x,y", .SDcols=301:350]`.

For convenient interactive use, the form `startcol:endcol` is also allowed (as in by), e.g., `DT[, lapply(.SD, sum), by=x:y, .SDcols=a:f]`.

Inversion (column dropping instead of keeping) can be accomplished be prepending the argument with `!` or `-` (there's no difference between these), e.g. `.SDcols = !c('x', 'y')`.

Finally, you can filter columns to include in `.SD` based on their *names* according to regular expressions via `.SDcols=patterns(regex1, regex2, ...)`. The included columns will be the *intersection* of the columns identified by each pattern; pattern unions can easily be specified with `|` in a regex. You can filter columns on values by passing a function, e.g. `.SDcols=is.numeric`. You can also invert a pattern as usual with `.SDcols=!patterns(...)` or `.SDcols=!is.numeric`.

`verbose` `TRUE` turns on status and information messages to the console. Turn this on by default using `options(datatable.verbose=TRUE)`. The quantity and types of verbosity may be expanded in future.

`allow.cartesian`

`FALSE` prevents joins that would result in more than `nrow(x)+nrow(i)` rows. This is usually caused by duplicate values in `i`'s join columns, each of which join to the same group in `x` over and over again: a *misspecified* join. Usually this was not intended and the join needs to be changed. The word 'cartesian' is used loosely in this context. The traditional cartesian join is (deliberately) difficult to achieve in `data.table`: where every row in `i` joins to every row in `x` (a `nrow(x)*nrow(i)` row result). 'cartesian' is just meant in a 'large multiplicative' sense, so FALSE does not always prevent a traditional cartesian join.

`drop` Never used by `data.table`. Do not use. It needs to be here because `data.table` inherits from `data.frame`. See `vignette("datatable-faq")`.

`on` Indicate which columns in `x` should be joined with which columns in `i` along with the type of binary operator to join with (see non-equi joins below on this). When specified, this overrides the keys set on `x` and `i`. When `.NATURAL` keyword provided then *natural join* is made (join on common columns). There are multiple ways of specifying the `on` argument:

- As an unnamed character vector, e.g., `X[Y, on=c("a", "b")]`, used when columns `a` and `b` are common to both `X` and `Y`.
- *Foreign key joins*: As a *named* character vector when the join columns have different names in `X` and `Y`. For example, `X[Y, on=c(x1="y1", x2="y2")]` joins `X` and `Y` by matching columns `x1` and `x2` in `X` with columns `y1` and `y2` in `Y`, respectively.
  From v1.9.8, you can also express foreign key joins using the binary operator `==`, e.g. `X[Y, on=c("x1==y1", "x2==y2")]`.
  NB: shorthand like `X[Y, on=c("a", V2="b")]` is also possible if, e.g., column `"a"` is common between the two tables.
- For convenience during interactive scenarios, it is also possible to use `.()` syntax as `X[Y, on=.(a, b)]`.
- From v1.9.8, (non-equi) joins using binary operators `>=`, `>`, `<=`, `<` are also possible, e.g., `X[Y, on=c("x>=a", "y<=b")]`, or for interactive use as `X[Y, on=.(x>=a, y<=b)]`.

Note that providing `on` is *required* for `X[Y]` joins when `X` is unkeyed. See examples as well as `vignette("datatable-secondary-indices-and-auto-indexing")`.

`env` List or an environment, passed to `substitute2` for substitution of parameters in `i`, `j` and `by` (or `keyby`). Use `verbose` to preview constructed expressions. For more details see `vignette("datatable-programming")`.

`showProgress` `TRUE` shows progress indicator with estimated time to completion for lengthy "by" operations.

Details

`data.table` builds on base R functionality to reduce 2 types of time:

1. programming time (easier to write, read, debug and maintain), and
2. compute time (fast and memory efficient).

The general form of data.table syntax is:

```
DT[ i,  j,  by ] # + extra arguments
    |   |   |
    |   |    -------> grouped by what?
    |    -------> what to do?
     ---> on which rows?
```

The way to read this out loud is: "Take DT, subset rows by i, *then* compute j grouped by by. Here are some basic usage examples expanding on this definition. See the vignette (and examples) for working examples.

```
X[, a]               # return col 'a' from X as vector. If not found, search in parent frame.
 X[, .(a)]                   # same as above, but return as a data.table.
X[, sum(a)]           # return sum(a) as a vector (with same scoping rules as above)
 X[, .(sum(a)), by=c]        # get sum(a) grouped by 'c'.
X[, sum(a), by=c]       # same as above, .() can be omitted in j and by on single expression for conve
X[, sum(a), by=c:f]      # get sum(a) grouped by all columns in between 'c' and 'f' (both inclusive)

X[, sum(a), keyby=b]     # get sum(a) grouped by 'b', and sort that result by the grouping column 'b'
 X[, sum(a), by=b, keyby=TRUE] # same order as above, but using sorting flag
X[, sum(a), by=b][order(b)] # same order as above, but by chaining compound expressions
X[c>1, sum(a), by=c]     # get rows where c>1 is TRUE, and on those rows, get sum(a) grouped by 'c'
X[Y, .(a, b), on="c"]    # get rows where Y$c == X$c, and select columns 'X$a' and 'X$b' for those row
X[Y, .(a, i.a), on="c"]   # get rows where Y$c == X$c, and then select 'X$a' and 'Y$a' (=i.a)
X[Y, sum(a*i.a), on="c", by=.EACHI] # for *each* 'Y$c', get sum(a*i.a) on matching rows in 'X$c'

X[, plot(a, b), by=c]    # j accepts any expression, generates plot for each group and returns no dat
# see ?assign to add/update/delete columns by reference using the same consistent interface
```

A data.table query may be invoked on a data.frame using functional form DT(...), see examples. The class of the input is retained.

A data.table is a list of vectors, just like a data.frame. However :

1. it never has or uses rownames. Rownames based indexing can be done by setting a *key* of one or more columns or done *ad-hoc* using the on argument (now preferred).
2. it has enhanced functionality in [.data.table for fast joins of keyed tables, fast aggregation, fast last observation carried forward (LOCF) and fast add/modify/delete of columns by reference with no copy at all.

See the see also section for the several other *methods* that are available for operating on data.tables efficiently.

Note

If keep.rownames or check.names are supplied they must be written in full because R does not allow partial argument names after .... For example, data.table(DF, keep=TRUE) will create a column called keep containing TRUE and this is correct behaviour; data.table(DF, keep.rownames=TRUE) was intended.

POSIXlt is not supported as a column type because it uses 40 bytes to store a single datetime. They are implicitly converted to POSIXct type with *warning*. You may also be interested in IDateTime instead; it has methods to convert to and from POSIXlt.

References

https://r-datatable.com (data.table homepage)
https://en.wikipedia.org/wiki/Binary_search

See Also

special-symbols, data.frame, [.data.frame, as.data.table, setkey, setorder, setDT, setDF, J, SJ, CJ, merge.data.table, tables, test.data.table, IDateTime, unique.data.table, copy, :=, setalloccol, truelength, rbindlist, setNumericRounding, datatable-optimize, fsetdiff, funion, fintersect, fsetequal, anyDuplicated, uniqueN, rowid, rleid, na.omit, frank, rowwiseDT

Examples

```
## Not run:
example(data.table)  # to run these examples yourself

## End(Not run)
DF = data.frame(x=rep(c("b","a","c"),each=3), y=c(1,3,6), v=1:9)
DT = data.table(x=rep(c("b","a","c"),each=3), y=c(1,3,6), v=1:9)
DF
DT
identical(dim(DT), dim(DF))    # TRUE
identical(DF$a, DT$a)          # TRUE
is.list(DF)                    # TRUE
is.list(DT)                    # TRUE

is.data.frame(DT)              # TRUE

tables()

# basic row subset operations
DT[2]                          # 2nd row
DT[3:2]                        # 3rd and 2nd row
DT[order(x)]                   # no need for order(DT$x)
DT[order(x), ]                 # same as above. The ',' is optional
DT[y>2]                        # all rows where DT$y > 2
DT[y>2 & v>5]                  # compound logical expressions
DT[!2:4]                       # all rows other than 2:4
DT[-(2:4)]                     # same

# select|compute columns data.table way
DT[, v]                        # v column (as vector)
DT[, list(v)]                  # v column (as data.table)
DT[, .(v)]                     # same as above, .() is a shorthand alias to list()
DT[, sum(v)]                   # sum of column v, returned as vector
DT[, .(sum(v))]                # same, but return data.table (column autonamed V1)
DT[, .(sv=sum(v))]             # same, but column named "sv"
DT[, .(v, v*2)]                # return two column data.table, v and v*2

# subset rows and select|compute data.table way
DT[2:3, sum(v)]                # sum(v) over rows 2 and 3, return vector
DT[2:3, .(sum(v))]             # same, but return data.table with column V1
DT[2:3, .(sv=sum(v))]          # same, but return data.table with column sv
DT[2:5, cat(v, "\n")]          # just for j's side effect

# select columns the data.frame way
```

```
DT[, 2]                        # 2nd column, returns a data.table always
colNum = 2
DT[, ..colNum]                 # same, .. prefix conveys one-level-up in calling scope
DT[["v"]]                      # same as DT[, v] but faster if called in a loop

# grouping operations - j and by
DT[, sum(v), by=x]             # ad hoc by, order of groups preserved in result
DT[, sum(v), keyby=x]          # same, but order the result on by cols
DT[, sum(v), by=x, keyby=TRUE] # same, but using sorting flag
DT[, sum(v), by=x][order(x)]   # same but by chaining expressions together

# fast ad hoc row subsets (subsets as joins)
DT["a", on="x"]                # same as x == "a" but uses binary search (fast)
                               #   NB: requires DT to be keyed!
DT["a", on=.(x)]               # same, for convenience, no need to quote every column
                               #   NB: works regardless of whether or not DT is keyed!
DT[.("a"), on="x"]             # same
DT[x=="a"]              # same, single "==" internally optimised to use binary search (fast)
DT[x!="b" | y!=3]              # not yet optimized, currently vector scan subset
DT[.("b", 3), on=c("x", "y")]  # join on columns x,y of DT; uses binary search (fast)
DT[.("b", 3), on=.(x, y)]      # same, but using on=.()
DT[.("b", 1:2), on=c("x", "y")]             # no match returns NA
DT[.("b", 1:2), on=.(x, y), nomatch=NULL]   # no match row is not returned
DT[.("b", 1:2), on=c("x", "y"), roll=Inf]  # locf, nomatch row gets rolled by previous row
DT[.("b", 1:2), on=.(x, y), roll=-Inf]      # nocb, nomatch row gets rolled by next row
DT["b", sum(v*y), on="x"]                  # on rows where DT$x=="b", calculate sum(v*y)

# all together now
DT[x!="a", sum(v), by=x]                    # get sum(v) by "x" for each i != "a"
DT[!"a", sum(v), by=.EACHI, on="x"]         # same, but using subsets-as-joins
DT[c("b","c"), sum(v), by=.EACHI, on="x"]   # same
DT[c("b","c"), sum(v), by=.EACHI, on=.(x)]  # same, using on=.()

# joins as subsets
X = data.table(x=c("c","b"), v=8:7, foo=c(4,2))
X

DT[X, on="x"]                          # right join
X[DT, on="x"]                          # left join
DT[X, on="x", nomatch=NULL]            # inner join
DT[!X, on="x"]                         # not join
DT[X, on=c(y="v")]                     # join using column "y" of DT with column "v" of X
DT[X, on="y==v"]                       # same as above (v1.9.8+)

DT[X, on=.(y<=foo)]                    # NEW non-equi join (v1.9.8+)
DT[X, on="y<=foo"]                     # same as above
DT[X, on=c("y<=foo")]                  # same as above
DT[X, on=.(y>=foo)]                    # NEW non-equi join (v1.9.8+)
DT[X, on=.(x, y<=foo)]                 # NEW non-equi join (v1.9.8+)
DT[X, .(x,y,x.y,v), on=.(x, y>=foo)]  # Select x's join columns as well

DT[X, on="x", mult="first"]            # first row of each group
DT[X, on="x", mult="last"]             # last row of each group
```

```
DT[X, sum(v), by=.EACHI, on="x"]      # join and eval j for each row in i
DT[X, sum(v)*foo, by=.EACHI, on="x"]  # join inherited scope
DT[X, sum(v)*i.v, by=.EACHI, on="x"]  # 'i,v' refers to X's v column
DT[X, on=.(x, v>=v), sum(y)*foo, by=.EACHI] # NEW non-equi join with by=.EACHI (v1.9.8+)

# setting keys
kDT = copy(DT)                        # (deep) copy DT to kDT to work with it.
setkey(kDT,x)                         # set a 1-column key. No quotes, for convenience.
setkeyv(kDT,"x")                      # same (v in setkeyv stands for vector)
v="x"
setkeyv(kDT,v)                        # same
haskey(kDT)                           # TRUE
key(kDT)                              # "x"

# fast *keyed* subsets
kDT["a"]                              # subset-as-join on *key* column 'x'
kDT["a", on="x"]                      # same, being explicit using 'on=' (preferred)

# all together
kDT[!"a", sum(v), by=.EACHI]          # get sum(v) for each i != "a"

# multi-column key
setkey(kDT,x,y)                       # 2-column key
setkeyv(kDT,c("x","y"))               # same

# fast *keyed* subsets on multi-column key
kDT["a"]                              # join to 1st column of key
kDT["a", on="x"]                      # on= is optional, but is preferred
kDT[.("a")]                           # same, .() is an alias for list()
kDT[list("a")]                        # same
kDT[.("a", 3)]                        # join to 2 columns
kDT[.("a", 3:6)]                      # join 4 rows (2 missing)
kDT[.("a", 3:6), nomatch=NULL]        # remove missing
kDT[.("a", 3:6), roll=TRUE]           # locf rolling join
kDT[.("a", 3:6), roll=Inf]            # same as above
kDT[.("a", 3:6), roll=-Inf]           # nocb rolling join
kDT[!.("a")]                          # not join
kDT[!"a"]                             # same

# more on special symbols, see also ?"special-symbols"
DT[.N]                                  # last row
DT[, .N]                                # total number of rows in DT
DT[, .N, by=x]                          # number of rows in each group
DT[, .SD, .SDcols=x:y]                  # select columns 'x' through 'y'
DT[ , .SD, .SDcols = !x:y]              # drop columns 'x' through 'y'
DT[ , .SD, .SDcols = patterns('^[xv]')] # select columns matching '^x' or '^v'
DT[, .SD[1]]                            # first row of all columns
DT[, .SD[1], by=x]                      # first row of 'y' and 'v' for each group in 'x'
DT[, c(.N, lapply(.SD, sum)), by=x]     # get rows *and* sum columns 'v' and 'y' by group
DT[, .I[1], by=x]                       # row number in DT corresponding to each group
DT[, grp := .GRP, by=x]                 # add a group counter column
DT[ , dput(.BY), by=.(x,y)]             # .BY is a list of singletons for each group
X[, DT[.BY, y, on="x"], by=x]           # join within each group
```

```
DT[, {
  # write each group to a different file
  fwrite(.SD, file.path(tempdir(), paste0('x=', .BY$x, '.csv')))
}, by=x]
dir(tempdir())

# add/update/delete by reference (see ?assign)
print(DT[, z:=42L])                   # add new column by reference
print(DT[, z:=NULL])                  # remove column by reference
print(DT["a", v:=42L, on="x"])        # subassign to existing v column by reference
print(DT["b", v2:=84L, on="x"])       # subassign to new column by reference (NA padded)

DT[, m:=mean(v), by=x][]              # add new column by reference by group
                                      # NB: postfix [] is shortcut to print()
# advanced usage
DT = data.table(x=rep(c("b","a","c"),each=3), v=c(1,1,1,2,2,1,1,2,2), y=c(1,3,6), a=1:9, b=9:1)

DT[, sum(v), by=.(y%%2)]              # expressions in by
DT[, sum(v), by=.(bool = y%%2)]      # same, using a named list to change by column name
DT[, .SD[2], by=x]                    # get 2nd row of each group
DT[, tail(.SD,2), by=x]               # last 2 rows of each group
DT[, lapply(.SD, sum), by=x]          # sum of all (other) columns for each group
DT[, .SD[which.min(v)], by=x]         # nested query by group

DT[, list(MySum=sum(v),
          MyMin=min(v),
          MyMax=max(v)),
    by=.(x, y%%2)]                    # by 2 expressions

# programmatic query with env=
DT[, .(funvar = fun(var)), by=grp_var,
   env = list(fun="sum", var="a", funvar="sum_a_by_x", grp_var="x")]

DT[, .(a = .(a), b = .(b)), by=x]     # list columns
DT[, .(seq = min(a):max(b)), by=x]    # j is not limited to just aggregations
DT[, sum(v), by=x][V1<20]             # compound query
DT[, sum(v), by=x][order(-V1)]        # ordering results
DT[, c(.N, lapply(.SD,sum)), by=x]    # get number of observations and sum per group
DT[, {tmp <- mean(y);
      .(a = a-tmp, b = b-tmp)
      }, by=x]                        # anonymous lambda in 'j', j accepts any valid
                                      # expression. TO REMEMBER: every element of
                                      # the list becomes a column in result.
pdf("new.pdf")
DT[, plot(a,b), by=x]                 # can also plot in 'j'
dev.off()


# using rleid, get max(y) and min of all cols in .SDcols for each consecutive run of 'v'
DT[, c(.(y=max(y)), lapply(.SD, min)), by=rleid(v), .SDcols=v:b]

# Support guide and links:
# https://github.com/Rdatatable/data.table/wiki/Support
```

```
## Not run:
if (interactive()) {
  vignette(package="data.table")  # 9 vignettes

  test.data.table()               # 6,000 tests

  # keep up to date with latest stable version on CRAN
  update.packages()

  # get the latest devel version that has passed all tests
  update_dev_pkg()
  # read more at:
  # https://github.com/Rdatatable/data.table/wiki/Installation
}

## End(Not run)
```

`.Last.updated` *Number of rows affected by last update*

Description

Returns number of rows affected by last `:=` or `set()`.

Usage

```
.Last.updated
```

Details

Be aware that in the case of duplicate indices, multiple updates occur (duplicates are overwritten); `.Last.updated` will include *all* of the updates performed, including duplicated ones. See examples.

Value

Integer.

See Also

`:=`

Examples

```
d = data.table(a=1:4, b=2:5)
d[2:3, z:=5L]
.Last.updated

# updated count takes duplicates into account #2837
DT = data.table(a = 1L)
```

```
DT[c(1L, 1L), a := 2:3]
.Last.updated
```

`.selfref.ok` *Tests self reference of a data.table*

Description

In rare situations, as presented in examples below, a data.table object may lose its internal attribute that holds a self-reference. This function tests just that.

It is not expected that many end users will have need for this highly technical function about data.table internals.

Usage

```
.selfref.ok(x)
```

Arguments

`x`	A data.table.

Value

`TRUE` if self reference attribute is properly set, `FALSE` otherwise.

Examples

```
d1 = structure(list(V1=1L), class=c("data.table","data.frame"))
.selfref.ok(d1)
setDT(d1)
.selfref.ok(d1)

saveRDS(d1, f<-tempfile())
d2 = readRDS(f)
.selfref.ok(d2)
invisible(file.remove(f))
setDT(d2)
.selfref.ok(d2)

d3 = unserialize(serialize(d2, NULL))
.selfref.ok(d3)
setDT(d3)
.selfref.ok(d3)
```

:= *Assignment by reference*

Description

Fast add, remove and update subsets of columns, by reference. := operator can be used in two ways: LHS := RHS form, and Functional form. See Usage.

Note that when using ':=' inside a {...} block in j, the ':=' call must be the only statement. For assigning to multiple columns, use the functional form: DT[, `:=`(col1=val1, col2=val2)].

set is a low-overhead loop-able version of :=. It is particularly useful for repetitively updating rows of certain columns by reference (using a for-loop). See Examples. It can not perform grouping operations.

let is an alias for the functional form and behaves exactly like `:=`.

Usage

```
# 1. LHS := RHS form
# DT[i, LHS := RHS, by = ...]
# DT[i, c("LHS1", "LHS2") := list(RHS1, RHS2), by = ...]

# 2a. Functional form with `:=`
# DT[i, `:=`(LHS1 = RHS1,
#            LHS2 = RHS2,
#            ...), by = ...]

# 2b. Functional form with let
# DT[i, let(LHS1 = RHS1,
#            LHS2 = RHS2,
#            ...), by = ...]

# 3. Multiple columns in place
# DT[i, names(.SD) := lapply(.SD, fx), by = ..., .SDcols = ...]

set(x, i = NULL, j, value)
```

Arguments

LHS	A character vector of column names (or numeric positions) or a variable that evaluates as such. If the column doesn't exist, it is added, *by reference*.
RHS	A list of replacement values. It is recycled in the usual way to fill the number of rows satisfying i, if any. To remove a column use NULL.
x	A data.table. Or, set() accepts data.frame, too.
i	Optional. Indicates the rows on which the values must be updated. If not NULL, implies *all rows*. Missing or zero values are ignored. The := form is more

powerful as it allows adding/updating columns by reference based on *subsets* and `joins`. See `Details`.

In `set`, only integer type is allowed in `i` indicating which rows `value` should be assigned to. `NULL` represents all rows more efficiently than creating a vector such as `1:nrow(x)`.

`j` — Column name(s) (character) or number(s) (integer) to be assigned `value` when column(s) already exist, and only column name(s) if they are to be created.

`value` — A list of replacement values to assign by reference to `x[i, j]`.

Details

`:=` is defined for use in `j` only. It *adds* or *updates* or *removes* column(s) by reference. It makes no copies of any part of memory at all. Please read `vignette("datatable-reference-semantics")` and follow with examples. Some typical usages are:

```
DT[, col := val]                    # update (or add at the end if doesn't exist) a column called "col
DT[i, col := val]                    # same as above, but only for those rows specified in i and (for n
 DT[i, "col a" := val]                        # same. column is called "col a"
DT[i, (3:6) := val]                   # update existing columns 3:6 with value. Aside: parens are not
DT[i, colvector := val, with = FALSE]      # OLD syntax. The contents of "colvector" in calling scope
DT[i, (colvector) := val]                # same (NOW PREFERRED) shorthand syntax. The parens are enough
DT[i, colC := mean(colB), by = colA]       # update (or add) column called "colC" by reference by grou
  DT[,`:=`(new1 = sum(colB), new2 = sum(colC))] # Functional form
 DT[, let(new1 = sum(colB), new2 = sum(colC))] # New alias for functional form.
```

The `.Last.updated` variable contains the number of rows updated by the most recent `:=` or `set` calls, which may be useful, for example, in production settings for testing assumptions about the number of rows affected by a statement; see `.Last.updated` for details.

Note that for efficiency no check is performed for duplicate assignments, i.e. if multiple values are passed for assignment to the same index, assignment to this index will occur repeatedly and sequentially; for a given use case, consider whether it makes sense to create your own test for duplicates, e.g. in production code.

All of the following result in a friendly error (by design) :

```
  x := 1L
  DT[i, col] := val
  DT[i]$col := val
DT[, {col1 := 1L; col2 := 2L}]          # Using `{}` in `j` is reserved for single `:=` expressions.
                                  # For multiple updates, use the functional form `:=`() instead.
```

For additional resources, please read `vignette("datatable-faq")`. Also have a look at StackOverflow's data.table tag.

`:=` in `j` can be combined with all types of `i` (such as binary search), and all types of by. This a one reason why `:=` has been implemented in `j`. Please see `vignette("datatable-reference-semantics")` and also `FAQ 2.16` for analogies to SQL.

When `LHS` is a factor column and `RHS` is a character vector with items missing from the factor levels, the new level(s) are automatically added (by reference, efficiently), unlike base methods.

Unlike '<-' for data.frame, the (potentially large) LHS is not coerced to match the type of the (often small) RHS. Instead the RHS is coerced to match the type of the LHS, if necessary. Where this involves double precision values being coerced to an integer column, a warning is given when fractional data is truncated. It is best to get the column types correct up front and stick to them. Changing a column type is possible but deliberately harder: provide a whole column as the RHS. This RHS is then *plonked* into that column slot and we call this *plonk syntax*, or *replace column syntax* if you prefer. By needing to construct a full length vector of a new type, you as the user are more aware of what is happening and it is clearer to readers of your code that you really do intend to change the column type; e.g., DT[, colA:=as.integer(colA)]. A plonk occurs whenever you provide a RHS value to ':=' which is nrow long. When a column is *plonked*, the original column is not updated by reference because that would entail updating every single element of that column whereas the plonk is just one column pointer update.

data.tables are *not* copied-on-change by :=, setkey or any of the other set* functions. See copy.

Value

DT is modified by reference and returned invisibly. If you require a copy, take a copy first (using DT2 = copy(DT)).

Advanced (internals):

It is easy to see how *sub-assigning* to existing columns is done internally. Removing columns by reference is also straightforward by modifying the vector of column pointers only (using memmove in C). However adding (new) columns is more tricky as to how the data.table can be grown *by reference*: the list vector of column pointers is *over-allocated*, see truelength. By defining := in j we believe update syntax is natural, and scales, but it also bypasses [<- dispatch and allows := to update by reference with no copies of any part of memory at all.

Since [.data.table incurs overhead to check the existence and type of arguments (for example), set() provides direct (but less flexible) assignment by reference with low overhead, appropriate for use inside a for loop. See examples. := is more powerful and flexible than set() because := is intended to be combined with i and by in single queries on large datasets.

Note

DT[a > 4, b := c] is different from DT[a > 4][, b := c]. The first expression updates (or adds) column b with the value c on those rows where a > 4 evaluates to TRUE. X is updated *by reference*, therefore no assignment needed. Note that this does not apply when i is missing, i.e. DT[].

The second expression on the other hand updates a *new* data.table that's returned by the subset operation. Since the subsetted data.table is ephemeral (it is not assigned to a symbol), the result would be lost; unless the result is assigned, for example, as follows: ans <- DT[a > 4][, b := c].

See Also

data.table, copy, setalloccol, truelength, set, .Last.updated

Examples

```
DT = data.table(a = LETTERS[c(3L,1:3)], b = 4:7)
DT[, c := 8]                # add a numeric column, 8 for all rows
DT[, d := 9L]               # add an integer column, 9L for all rows
DT[, c := NULL]             # remove column c
DT[2, d := -8L]             # subassign by reference to d; 2nd row is -8L now
DT                          # DT changed by reference
DT[2, d := 10L][]           # shorthand for update and print

DT[b > 4, b := d * 2L]      # subassign to b with d*2L on those rows where b > 4 is TRUE
DT[b > 4][, b := d * 2L]    # different from above. [, := ] is performed on the subset
                            # which is an new (ephemeral) data.table. Result needs to be
                            # assigned to a variable (using `<-`).

DT[, e := mean(d), by = a]  # add new column by group by reference
DT["A", b := 0L, on = "a"]  # ad-hoc update of column b for group "A" using
    # joins-as-subsets with binary search and 'on='
# same as above but using keys
setkey(DT, a)
DT["A", b := 0L]            # binary search for group "A" and set column b using keys
DT["B", f := mean(d)]       # subassign to new column, NA initialized

# Adding multiple columns
## by name
DT[ , c('sin_d', 'log_e', 'cos_d') :=
   .(sin(d), log(e), cos(d))]
## by patterned name
DT[ , paste(c('sin', 'cos'), 'b', sep = '_') :=
   .(sin(b), cos(b))]
## using lapply & .SD
DT[ , paste0('tan_', c('b', 'd', 'e')) :=
   lapply(.SD, tan), .SDcols = c('b', 'd', 'e')]
## using forced evaluation to disambiguate a vector of names
##   and overwrite existing columns with their squares
sq_cols = c('b', 'd', 'e')
DT[ , (sq_cols) := lapply(.SD, `^`, 2L), .SDcols = sq_cols]
## by integer (NB: for robustness, it is not recommended
##   to use explicit integers to update/define columns)
DT[ , c(2L, 3L, 4L) := .(sqrt(b), sqrt(d), sqrt(e))]
## by implicit integer
DT[ , grep('a$', names(DT)) := tolower(a)]
## by implicit integer, using forced evaluation
sq_col_idx = grep('d$', names(DT))
DT[ , (sq_col_idx) := lapply(.SD, dnorm),
   .SDcols = sq_col_idx]

# Examples using `set` function
## Set value for single cell
set(DT, 1L, "b", 10L)
## Set values for multiple columns in a specific row
set(DT, 2L, c("b", "d"), list(20L, 30L))
## Set values by column indices
```

```
set(DT, 3L, c(2L, 4L), list(40L, 50L))
## Set value for an entire column without specifying rows
set(DT, j = "b", value = 100L)
set(DT, NULL, "b", 100L) # equivalent
## Set values for multiple columns without specifying rows
set(DT, j = c("b", "d"), value = list(200L, 300L))
## Set values for multiple columns with multiple specified rows.
set(DT, c(1L, 3L), c("b", "d"), value = list(500L, 800L))

## Not run:
# Speed example:

m = matrix(1, nrow = 2e6L, ncol = 100L)
DF = as.data.frame(m)
DT = as.data.table(m)

system.time(for (i in 1:1000) DF[i, 1] = i)
# 15.856 seconds
system.time(for (i in 1:1000) DT[i, V1 := i])
# 0.279 seconds  (57 times faster)
system.time(for (i in 1:1000) set(DT, i, 1L, i))
# 0.002 seconds  (7930 times faster, overhead of [.data.table is avoided)

# However, normally, we call [.data.table *once* on *large* data, not many times on small data.
# The above is to demonstrate overhead, not to recommend looping in this way. But the option
# of set() is there if you need it.

## End(Not run)
```

`address` *Address in RAM of a variable*

Description

Returns the pointer address of its argument.

Usage

```
address(x)
```

Arguments

x	Anything.

Details

Sometimes useful in determining whether a value has been copied or not, programmatically.

Value

A character vector length 1.

References

https://stackoverflow.com/a/10913296/403310 (but implemented in C without using `.Internal(inspect())`)

See Also

copy

Examples

```
x=1
address(x)
```

---

all.equal *Equality Test Between Two Data Tables*

Description

Convenient test of data equality between `data.table` objects. Performs some factor level *stripping*.

Usage

```
  ## S3 method for class 'data.table'
all.equal(target, current, trim.levels=TRUE, check.attributes=TRUE,
   ignore.col.order=FALSE, ignore.row.order=FALSE, tolerance=sqrt(.Machine$double.eps),
    ...)
```

Arguments

`target, current`	`data.tables` to compare. If `current` is not a `data.table`, but `check.attributes` is FALSE, it will be coerced to one via as.data.table.
`trim.levels`	A logical indicating whether or not to remove all unused levels in columns that are factors before running equality check. It effect only when `check.attributes` is TRUE and `ignore.row.order` is FALSE.
`check.attributes`	A logical indicating whether or not to check attributes. Note that this will apply not only to the data.tables, but also to attributes of the columns. `"row.names"` and any internal data.table attributes are always skipped.
`ignore.col.order`	A logical indicating whether or not to ignore columns order in `data.table`.
`ignore.row.order`	A logical indicating whether or not to ignore rows order in `data.table`. This option requires datasets to use data types on which join can be made, so no support for *list, complex, raw*, but still supports integer64.

tolerance	A numeric value used when comparing numeric columns, by default `sqrt(.Machine$double.eps)`. Unless non-default value provided it will be forced to `0` if used together with `ignore.row.order` and duplicate rows detected or factor columns present.
...	Passed down to internal call of `all.equal`.

Details

For efficiency data.table method will exit on detected non-equality issues, unlike most `all.equal` methods which process equality checks further. Besides that fact it also handles the most time consuming case of `ignore.row.order = TRUE` very efficiently.

Value

Either `TRUE` or a vector of mode `"character"` describing the differences between `target` and `current`.

See Also

`all.equal`

Examples

```
dt1 <- data.table(A = letters[1:10], X = 1:10, key = "A")
dt2 <- data.table(A = letters[5:14], Y = 1:10, key = "A")
isTRUE(all.equal(dt1, dt1))
is.character(all.equal(dt1, dt2))

# ignore.col.order
x <- copy(dt1)
y <- dt1[, .(X, A)]
all.equal(x, y)
all.equal(x, y, ignore.col.order = TRUE)

# ignore.row.order
x <- setkeyv(copy(dt1), NULL)
y <- dt1[sample(nrow(dt1))]
all.equal(x, y)
all.equal(x, y, ignore.row.order = TRUE)

# check.attributes
x = copy(dt1)
y = setkeyv(copy(dt1), NULL)
all.equal(x, y)
all.equal(x, y, check.attributes = FALSE)
x = data.table(1L)
y = 1L
all.equal(x, y)
all.equal(x, y, check.attributes = FALSE)

# trim.levels
x <- data.table(A = factor(letters[1:10])[1:4]) # 10 levels
y <- data.table(A = factor(letters[1:5])[1:4]) # 5 levels
```

```
all.equal(x, y, trim.levels = FALSE)
all.equal(x, y, trim.levels = FALSE, check.attributes = FALSE)
all.equal(x, y)
```

`as.data.table` *Coerce to data.table*

Description

Functions to check if an object is `data.table`, or coerce it if possible.

Usage

```
as.data.table(x, keep.rownames=FALSE, ...)

## S3 method for class 'data.table'
as.data.table(x, ..., key=NULL)

## S3 method for class 'array'
as.data.table(x, keep.rownames=FALSE, key=NULL, sorted=TRUE,
              value.name="value", na.rm=TRUE, ...)

is.data.table(x)
```

Arguments

`x`	An R object.
`keep.rownames`	Default is `FALSE`. If `TRUE`, adds the input object's names as a separate column named `"rn"`. `keep.rownames = "id"` names the column `"id"` instead. For lists and when calling `data.table()`, names from the first named vector are extracted and used as row names, similar to `data.frame()` behavior.
`key`	Character vector of one or more column names which is passed to `setkeyv`.
`sorted`	logical used in *array* method, default `TRUE` is overridden when `key` is provided.
`value.name`	character scalar used in *array* method, default `"value"`.
`na.rm`	logical used in *array* method, default `TRUE` will remove rows with `NA` values.
`...`	Additional arguments to be passed to or from other methods.

Details

`as.data.table` is a generic function with many methods, and other packages can supply further methods.

If a `list` is supplied, each element is converted to a column in the `data.table` with shorter elements recycled automatically. Similarly, each column of a `matrix` is converted separately.

`character` objects are *not* converted to `factor` types unlike `as.data.frame`.

If a data.frame is supplied, all classes preceding "data.frame" are stripped. Similarly, for data.table as input, all classes preceding "data.table" are stripped. as.data.table methods returns a *copy* of original data. To modify by reference see setDT and setDF.

keep.rownames argument can be used to preserve the (row)names attribute in the resulting data.table.

See Also

data.table, setDT, setDF, copy, setkey, J, SJ, CJ, merge.data.table, :=, setalloccol, truelength, rbindlist, setNumericRounding, datatable-optimize

Examples

```
nn = c(a=0.1, b=0.2, c=0.3, d=0.4)
as.data.table(nn)
as.data.table(nn, keep.rownames=TRUE)
as.data.table(nn, keep.rownames="rownames")

# char object not converted to factor
cc = c(X="a", Y="b", Z="c")
as.data.table(cc)
as.data.table(cc, keep.rownames=TRUE)
as.data.table(cc, keep.rownames="rownames")

mm = matrix(1:4, ncol=2, dimnames=list(c("r1", "r2"), c("c1", "c2")))
as.data.table(mm)
as.data.table(mm, keep.rownames=TRUE)
as.data.table(mm, keep.rownames="rownames")
as.data.table(mm, key="c1")

ll = list(a=1:2, b=3:4)
as.data.table(ll)
as.data.table(ll, keep.rownames=TRUE)
as.data.table(ll, keep.rownames="rownames")

DF = data.frame(x=rep(c("x","y","z"),each=2), y=c(1,3,6), row.names=LETTERS[1:6])
as.data.table(DF)
as.data.table(DF, keep.rownames=TRUE)
as.data.table(DF, keep.rownames="rownames")

DT = data.table(x=rep(c("x","y","z"),each=2), y=c(1:6))
as.data.table(DT)
as.data.table(DT, key='x')

ar = rnorm(27)
ar[sample(27, 15)] = NA
dim(ar) = c(3L,3L,3L)
as.data.table(ar)
```

as.data.table.xts *Efficient xts to as.data.table conversion*

Description

Efficient conversion xts to data.table.

Usage

```
## S3 method for class 'xts'
as.data.table(x, keep.rownames = TRUE, key=NULL, ...)
```

Arguments

x	xts to convert to data.table
keep.rownames	Default is TRUE. If TRUE, adds the xts input's index as a separate column named "index". keep.rownames = "id" names the index column "id" instead.
key	Character vector of one or more column names which is passed to setkeyv.
...	ignored, just for consistency with as.data.table

See Also

as.xts.data.table

Examples

```
if (requireNamespace("xts", quietly = TRUE)) {
  data(sample_matrix, package = "xts")
  sample.xts <- xts::as.xts(sample_matrix) # xts might not be attached on search path
  # print head of xts
  print(head(sample.xts))
  # print data.table
  print(as.data.table(sample.xts))
}
```

as.matrix *Convert a data.table to a matrix*

Description

Converts a data.table into a matrix, optionally using one of the columns in the data.table as the matrix rownames.

Usage

```
## S3 method for class 'data.table'
as.matrix(x, rownames=NULL, rownames.value=NULL, ...)
```

Arguments

x	a data.table
rownames	optional, a single column name or column number to use as the rownames in the returned matrix. If TRUE the key of the data.table will be used if it is a single column, otherwise the first column in the data.table will be used.
rownames.value	optional, a vector of values to be used as the rownames in the returned matrix. It must be the same length as nrow(x).
...	Required to be present because the generic as.matrix generic has it. Arguments here are not currently used or passed on by this method.

Details

as.matrix is a generic function in base R. It dispatches to as.matrix.data.table if its x argument is a data.table.

The method for data.tables will return a character matrix if there are only atomic columns and any non-(numeric/logical/complex) column, applying as.vector to factors and format to other non-character columns. Otherwise, the usual coercion hierarchy (logical < integer < double < complex) will be used, e.g., all-logical data frames will be coerced to a logical matrix, mixed logical-integer will give an integer matrix, etc.

Value

A new matrix containing the contents of x.

See Also

data.table, as.matrix, data.matrix array

Examples

```
DT <- data.table(A = letters[1:10], X = 1:10, Y = 11:20)
as.matrix(DT) # character matrix
as.matrix(DT, rownames = "A")
as.matrix(DT, rownames = 1)
as.matrix(DT, rownames = TRUE)

setkey(DT, A)
as.matrix(DT, rownames = TRUE)
```

---

as.xts.data.table *Efficient data.table to xts conversion*

Description

Efficient conversion of data.table to xts, data.table must have a time based type in first column. See ?xts::timeBased for supported types

Usage

```
as.xts.data.table(x, numeric.only = TRUE, ...)
```

Arguments

x	data.table to convert to xts, must have a time based first column. As xts objects are indexed matrixes, all columns must be of the same type. If columns of multiple types are selected, standard as.matrix rules are applied during the conversion.
numeric.only	If TRUE, only include numeric columns in the conversion and all non-numeric columns will be omitted with warning
...	ignored, just for consistency with generic method.

See Also

as.data.table.xts

Examples

```
if (requireNamespace("xts", quietly = TRUE)) {
 sample.dt <- data.table(date = as.Date((Sys.Date()-999):Sys.Date(),origin="1970-01-01"),
                         quantity = sample(10:50,1000,TRUE),
                         value = sample(100:1000,1000,TRUE))
  # print data.table
  print(sample.dt)
  # print head of xts
  print(head(as.xts.data.table(sample.dt))) # xts might not be attached on search path
}
```

---

between *Convenience functions for range subsets*

Description

Intended for use in i in [.data.table.

between is equivalent to lower<=x & x<=upper when incbounds=TRUE, or lower<x & y<upper when FALSE. With a caveat that NA in lower or upper are taken as unlimited bounds not NA. This can be changed by setting NAbounds to NA.

inrange checks whether each value in x is in between any of the intervals provided in lower,upper.

Usage

```
between(x, lower, upper, incbounds=TRUE, NAbounds=TRUE, check=FALSE, ignore_tzone=FALSE)
x %between% y

inrange(x, lower, upper, incbounds=TRUE)
x %inrange% y
```

Arguments

x	Any orderable vector, i.e., those with relevant methods for `<=`, such as numeric, character, Date, etc. in case of between and a numeric vector in case of inrange.
lower	Lower range bound. Either length 1 or same length as x.
upper	Upper range bound. Either length 1 or same length as x.
y	A length-2 vector or list, with y[[1]] interpreted as lower and y[[2]] as upper.
incbounds	TRUE means inclusive bounds, i.e., [lower,upper]. FALSE means exclusive bounds, i.e., (lower,upper). It is set to TRUE by default for infix notations.
NAbounds	If lower (upper) contains an NA what should lower<=x (x<=upper) return? By default TRUE so that a missing bound is interpreted as unlimited.
check	Produce error if any(lower>upper)? FALSE by default for efficiency, in particular type character.
ignore_tzone	TRUE means skip timezone checks among x, lower, and upper.

Details

non-equi joins were implemented in v1.9.8. They extend binary search based joins in data.table to other binary operators including >=, <=, >, <. inrange makes use of this new functionality and performs a range join.

Value

Logical vector the same length as x with value TRUE for those that lie within the specified range.

Note

Current implementation does not make use of ordered keys for %between%.

See Also

data.table, like, %chin%

Examples

```
X = data.table(a=1:5, b=6:10, c=c(5:1))
X[b %between% c(7,9)]
X[between(b, 7, 9)] # same as above
# NEW feature in v1.9.8, vectorised between
X[c %between% list(a,b)]
X[between(c, a, b)] # same as above
X[between(c, a, b, incbounds=FALSE)] # open interval

# inrange()
Y = data.table(a=c(8,3,10,7,-10), val=runif(5))
range = data.table(start = 1:5, end = 6:10)
Y[a %inrange% range]
```

```
Y[inrange(a, range$start, range$end)] # same as above
Y[inrange(a, range$start, range$end, incbounds=FALSE)] # open interval
```

`cbindlist` *Column bind multiple data.tables*

Description

Column bind multiple `data.tables`.

Usage

```
cbindlist(l)
setcbindlist(l)
```

Arguments

`l`	`list` of `data.tables` to merge.

Details

Column bind only stacks input elements. Works like `data.table`, but takes `list` type on input. Zero-column tables in `l` are omitted. Tables in `l` should have matching row count; recycling of length-1 rows is not yet implemented. Indices of the input tables are transferred to the resulting table, as well as the *key* of the first keyed table.

Value

A new `data.table` based on the stacked objects.

For `setcbindlist`, columns in the output will be shared with the input, i.e., *no copy is made*.

Note

No attempt is made to deduplicate resulting names. If the result has any duplicate names, keys and indices are removed.

See Also

`data.table`, `rbindlist`, `setDT`

Examples

```
d1 = data.table(x=1:3, v1=1L, key="x")
d2 = data.table(y=3:1, v2=2L, key="y")
d3 = data.table(z=2:4, v3=3L)
cbindlist(list(d1, d2, d3))
cbindlist(list(d1, d1))
d4 = setcbindlist(list(d1))
d4[, v1:=2L]
identical(d4, d1)
```

cdt *data.table exported C routines*

Description

Some of the internally used C routines are now exported. This interface should be considered experimental. List of exported C routines and their signatures are provided below in the usage section.

Usage

```
# SEXP DT_subsetDT(SEXP x, SEXP rows, SEXP cols);
# p_DT_subsetDT = R_GetCCallable("data.table", "DT_subsetDT");
```

Details

Details on how to use these can be found in the *Writing R Extensions* manual *Linking to native routines in other packages* section. An example use with Rcpp:

```
 dt = data.table::as.data.table(iris)
Rcpp::cppFunction("SEXP mysub2(SEXP x, SEXP rows, SEXP cols) { return DT_subsetDT(x,rows,cols); }",
  include="#include <datatableAPI.h>",
  depends="data.table")
mysub2(dt, 1:4, 1:4)
```

Note

Be aware C routines are likely to have less input validation than their corresponding R interface. For example one should not expect `DT[-5L]` will be equal to `.Call(DT_subsetDT, DT, -5L, seq_along(DT))` because translation of `i=-5L` to `seq_len(nrow(DT))[-5L]` might be happening on R level. Moreover checks that `i` argument is in range of `1:nrow(DT)`, missingness, etc. might be happening on R level too.

References

https://cran.r-project.org/doc/manuals/r-release/R-exts.html

chmatch *Faster match of character vectors*

Description

`chmatch` returns a vector of the positions of (first) matches of its first argument in its second. Both arguments must be character vectors.

`%chin%` is like `%in%`, but for character vectors.

Usage

```
chmatch(x, table, nomatch=NA_integer_)
x %chin% table
chorder(x)
chgroup(x)
```

Arguments

x	character vector: the values to be matched, or the values to be ordered or grouped
table	character vector: the values to be matched against.
nomatch	the value to be returned in the case when no match is found. Note that it is coerced to integer.

Details

Fast versions of match, %in% and order, optimised for character vectors. chgroup groups together duplicated values but retains the group order (according the first appearance order of each group), efficiently. They have been primarily developed for internal use by data.table, but have been exposed since that seemed appropriate.

Strings are already cached internally by R (CHARSXP) and that is utilised by these functions. No hash table is built or cached, so the first call is the same speed as subsequent calls. Essentially, a counting sort (similar to base::sort.list(x,method="radix"), see setkey) is implemented using the (almost) unused truelength of CHARSXP as the counter. *Where* R *has* used truelength of CHARSXP (where a character value is shared by a variable name), the non zero truelengths are stored first and reinstated afterwards. Each of the ch* functions implements a variation on this theme. Remember that internally in R, length of a CHARSXP is the nchar of the string and DATAPTR is the string itself.

Methods that do build and cache a hash table (such as the fastmatch package) are *much* faster on subsequent calls (almost instant) but a little slower on the first. Therefore chmatch may be particularly suitable for ephemeral vectors (such as local variables in functions) or tasks that are only done once. Much depends on the length of x and table, how many unique strings each contains, and whether the position of the first match is all that is required.

It may be possible to speed up fastmatch's hash table build time by using the technique in data.table, and we have suggested this to its author. If successful, fastmatch would then be fastest in all cases.

Value

As match and %in%. chorder and chgroup return an integer index vector.

Note

The name charmatch was taken by charmatch, hence chmatch.

See Also

match, %in%

Examples

```
# Please type 'example(chmatch)' to run this and see timings on your machine

N = 1e5
# N is set small here (1e5) to reduce runtime because every day CRAN runs and checks
# all documentation examples in addition to the package's test suite.
# The comments here apply when N has been changed to 1e8 and were run on 2018-05-13
# with R 3.5.0 and data.table 1.11.2.

u = as.character(as.hexmode(1:10000))
y = sample(u,N,replace=TRUE)
x = sample(u)
                                          #  With N=1e8 ...
system.time(a <- match(x,y))              #  4.6s
system.time(b <- chmatch(x,y))            #  1.8s
identical(a,b)

system.time(a <- x %in% y)              #  4.5s
system.time(b <- x %chin% y)            #  1.7s
identical(a,b)

# Different example with more unique strings ...
u = as.character(as.hexmode(1:(N/10)))
y = sample(u,N,replace=TRUE)
x = sample(u,N,replace=TRUE)
system.time(a <- match(x,y))              # 46s
system.time(b <- chmatch(x,y))            # 16s
identical(a,b)
```

---

copy *Copy an entire object*

Description

In data.table parlance, all set* functions change their input *by reference*. That is, no copy is made at all, other than temporary working memory, which is as large as one column. The only other data.table operator that modifies input by reference is :=. Check out the See Also section below for other set* function data.table provides.

copy() copies an entire object.

Usage

```
copy(x)
```

Arguments

x	A data.table.

Details

`data.table` provides functions that operate on objects *by reference* and minimise full object copies as much as possible. Still, it might be necessary in some situations to work on an object's copy which can be done using `DT.copy <- copy(DT)`. It may also be sometimes useful before `:=` (or `set`) is used to subassign to a column by reference.

A `copy()` may be required when doing `dt_names = names(DT)`. Due to R's *copy-on-modify*, `dt_names` still points to the same location in memory as `names(DT)`. Therefore modifying `DT` *by reference* now, say by adding a new column, `dt_names` will also get updated. To avoid this, one has to *explicitly* copy: `dt_names <- copy(names(DT))`.

Value

Returns a copy of the object.

Note

To confirm precisely whether an object is a copy of another, compare their exact memory address with `address`.

See Also

`data.table`, `address`, `setkey`, `setDT`, `setDF`, `set :=`, `setorder`, `setattr`, `setnames`

Examples

```
# Type 'example(copy)' to run these at prompt and browse output

DT = data.table(A=5:1,B=letters[5:1])
DT2 = copy(DT)        # explicit copy() needed to copy a data.table
setkey(DT2,B)         # now just changes DT2
identical(DT,DT2)     # FALSE. DT and DT2 are now different tables

DT = data.table(A=5:1, B=letters[5:1])
nm1 = names(DT)
nm2 = copy(names(DT))
DT[, C := 1L]
identical(nm1, names(DT)) # TRUE, nm1 is also changed by reference
identical(nm2, names(DT)) # FALSE, nm2 is a copy, different from names(DT)
```

---

data.table-class *S4 Definition for data.table*

Description

A `data.table` can be used in S4 class definitions as either a parent class (inside a `contains` argument of `setClass`), or as an element of an S4 slot.

Author(s)

Steve Lianoglou

See Also

data.table

Examples

```
## Used in inheritance.
setClass('SuperDataTable', contains='data.table')

## Used in a slot
setClass('Something', representation(x='character', dt='data.table'))
x <- new("Something", x='check', dt=data.table(a=1:10, b=11:20))
```

---

data.table-condition-classes

Condition Handling with Classed Conditions

Description

data.table provides specific condition classes for common operations, making it easier to handle conditions programmatically. This is particularly useful when writing robust code or packages that use data.table. Relying on the exact text of condition messages is fragile (it is not uncommon to change the wording slightly, or for the user's session not to be in English); prefer using the signal class where possible.

Details

Available Condition Classes: data.table provides the following specific condition classes:

Error Classes:

- dt_missing_column_error: When referencing columns that don't exist
- dt_invalid_input_error: When providing invalid input types or empty required arguments
- dt_unsortable_type_error: When trying to sort/key unsupported types
- dt_join_type_mismatch_error: When column types are incompatible in joins/set operations
- dt_invalid_let_error: When using assignment operators incorrectly

Warning Classes:

- dt_missing_fun_aggregate_warning: When aggregation function is missing in operations that require it

Backward Compatibility: All condition classes inherit from base R's condition system, so existing tryCatch(..., error = ...) code continues to work unchanged. The new classes simply provide more specific handling options when needed.

See Also

tryCatch, test, https://adv-r.hadley.nz/conditions.html

Examples

```
# Handle missing column errors specifically
DT <- data.table(a = 1:3, b = 4:6)
tryCatch({
  setkey(DT, nonexistent_col)
}, dt_missing_column_error = function(e) {
  cat("Missing column detected:", conditionMessage(e), "\n")
}, error = function(e) {
  cat("Other error:", conditionMessage(e), "\n")
})
```

---

data.table-options *Global Options for the data.table Package*

Description

The data.table package uses a number of global options to control its behavior. These are regular R options that can be set with options() and retrieved with getOption(). For example:

```
# Get the current value of an option
getOption("datatable.print.topn")

# Set a new value for an option
options(datatable.print.topn = 10)
```

This page provides a comprehensive, up-to-date list of all user-configurable options. NB: If you're reading this on the web, make sure the version numbers match with what you have installed.

Printing Options

See print.data.table for a full description of printing data.tables.

`datatable.print.topn` An integer, default `5L`. When a data.table is printed, only the first topn and last topn rows are displayed.

`datatable.print.nrows` An integer, default `100L`. The total number of rows to print before the topn logic is triggered.

`datatable.print.class` A logical, default `FALSE`. If `TRUE`, the class of each column is printed below its name.

`datatable.print.keys` A logical, default `FALSE`. If `TRUE`, the table's keys are printed above the data.

`datatable.show.indices` A logical, default `TRUE`. A synonym for `datatable.print.keys` for historical reasons.

`datatable.print.trunc.cols` A logical, default `FALSE`. If `TRUE` and a table has more columns than fit on the screen, it truncates the middle columns.

`datatable.prettyprint.char` An integer, default `100L`. The maximum number of characters to display in a character column cell before truncating.

`datatable.print.colnames` A logical, default `TRUE`. If `TRUE`, prints column names.

`datatable.print.rownames` A logical, default `TRUE`. If `TRUE`, prints row numbers.

File I/O Options (fread and fwrite)

See `fread` and `fwrite` for a full description of data.table I/O.

`datatable.fread.input.cmd.message` A logical, default `TRUE`. If `TRUE`, `fread` will print the shell command it is using when the input is a command (e.g., `fread("grep ...")`).

`datatable.fread.datatable` A logical, default `TRUE`. If `TRUE`, `fread` returns a `data.table`. If `FALSE`, it returns a `data.frame`.

`datatable.integer64` A character string, default `"integer64"`. Controls how `fread` handles 64-bit integers. Can be "integer64", "double", or "character".

`datatable.logical01` A logical, default `FALSE`. If `TRUE`, `fread` will interpret columns containing only 0 and 1 as logical.

`datatable.keepLeadingZeros` A logical, default `FALSE`. If `TRUE`, `fread` preserves leading zeros in character columns by reading them as strings; otherwise they may be coerced to numeric.

`datatable.logicalYN` A logical, default `FALSE`. If `TRUE`, `fread` will interpret "Y" and "N" as logical.

`datatable.na.strings` A character vector, default `"NA"`. Global default for strings that `fread` should interpret as `NA`.

`datatable.fwrite.sep` A character string, default `","`. The default separator used by `fwrite`.

`datatable.showProgress` An integer or logical, default `interactive()`. Controls whether long-running operations like `fread` display a progress bar.

Join and Subset Options

`datatable.allow.cartesian` A logical, default `FALSE`. Controls the default value of the `allow.cartesian` parameter; see `data.table`. If the value of this parameter is FALSE, an error is raised as a safeguard against an explosive Cartesian join.

`datatable.join.many` A logical. Stub description to be embellished later in PR #4370.

Performance and Indexing Options

`datatable.auto.index` A logical, default `TRUE`. If `TRUE`, `data.table` automatically creates a secondary index on-the-fly when a column is first used in a subset, speeding up all subsequent queries.

`datatable.use.index` A logical, default `TRUE`. A global switch to control whether existing secondary indices are used for subsetting.

`datatable.forder.auto.index` A logical, default `TRUE`. Similar to `datatable.auto.index`, but applies to ordering operations (`forder`).

`datatable.optimize` A numeric, default `Inf`. Controls the GForce query optimization engine. The default enables all possible optimizations. See `datatable.optimize`.

`datatable.alloccol` An integer, default `1024L`. Controls the number of column slots to pre-allocate, improving performance when adding many columns. See `alloc.col`.

`datatable.reuse.sorting` A logical, default `TRUE`. If `TRUE`, `data.table` can reuse the sorted order of a table in joins, improving performance.

Development and Verbosity Options

`datatable.quiet` A logical, default `FALSE`. The master switch to suppress all `data.table` status messages, including the startup message.

`datatable.verbose` A logical, default `FALSE`. If `TRUE`, `data.table` will print detailed diagnostic information as it processes a query.

`datatable.enlist` Experimental feature. Default is `NULL`. If set to a function (e.g., `list`), the `j` expression can return a `list`, which will then be "enlisted" into columns in the result.

Back-compatibility Options

`datatable.old.matrix.autoname` Logical, default `TRUE`. Governs how the output of expressions like `data.table(x=1, cbind(1))` will be named. When `TRUE`, it will be named `V1`, otherwise it will be named `V2`.

See Also

`options`, `getOption`, `data.table`

`datatable.optimize` *Optimisations in data.table*

Description

`data.table` internally optimises certain expressions in order to improve performance. This section briefly summarises those optimisations.

Note that there's no additional input needed from the user to take advantage of these optimisations. They happen automatically.

Run the code under the *example* section to get a feel for the performance benefits from these optimisations.

Note that for all optimizations involving efficient sorts, the caveat mentioned in `setorder` applies – whenever data.table does the sorting, it does so in "C-locale". This has some subtle implications; see Examples.

Details

`data.table` reads the global option `datatable.optimize` to figure out what level of optimisation is required. The default value `Inf` activates *all* available optimisations.

For `getOption("datatable.optimize") >= 1`, these are the optimisations:

- The base function `order` is internally replaced with `data.table`'s *fast ordering*. That is, `DT[order(...)]` gets internally optimised to `DT[forder(...)]`.
- The expression `DT[, lapply(.SD, fun), by=.]` gets optimised to `DT[, list(fun(a), fun(b), ...), by=.]` where `a,b, ...` are columns in `.SD`. This improves performance tremendously.
- Similarly, the expression `DT[, c(.N, lapply(.SD, fun)), by=.]` gets optimised to `DT[, list(.N, fun(a), fun(b), ...)]`. `.N` is just for example here.
- `base::mean` function is internally optimised to use `data.table`'s `fastmean` function. `mean()` from `base` is an S3 generic and gets slow with many groups.

For `getOption("datatable.optimize") >= 2`, additional optimisations are implemented on top of the optimisations already shown above.

- Expressions in `j` which contain only the functions `min, max, mean, median, var, sd, sum, prod, first, last, head, tail` (for example, `DT[, list(mean(x), median(x), min(y), max(y)), by=z]`), they are very effectively optimised using what we call *GForce*. These functions are automatically replaced with a corresponding GForce version with pattern `g*`, e.g., `prod` becomes `gprod`.

  Normally, once the rows belonging to each group are identified, the values corresponding to the group are gathered and the `j`-expression is evaluated. This can be improved by computing the result directly without having to gather the values or evaluating the expression for each group (which can get costly with large number of groups) by implementing it specifically for a particular function. As a result, it is extremely fast.
- In addition to all the functions above, `.N` is also optimised to use GForce, when used separately or when combined with the functions mentioned above. Note further that GForce-optimized functions must be used separately, i.e., code like `DT[ , max(x) - min(x), by=z]` will *not* currently be optimized to use `gmax, gmin`.
- Expressions of the form `DT[i, j, by]` are also optimised when `i` is a *subset* operation and `j` is any/all of the functions discussed above.

For `getOption("datatable.optimize") >= 3`, additional optimisations for subsets in `i` are implemented on top of the optimisations already shown above. Subsetting operations are - if possible - translated into joins to make use of blazing fast binary search using indices and keys. The following queries are optimized:

- Supported operators: `==, %in%`. Non-equi operators(>, <, etc.) are not supported yet because non-equi joins are slower than vector based subsets.
- Queries on multiple columns are supported, if the connector is '`&`', e.g. `DT[x == 2 & y == 3]` is supported, but `DT[x == 2 | y == 3]` is not.
- Optimization will currently be turned off when doing subset when cross product of elements provided to filter on exceeds > 1e4. This most likely happens if multiple `%in%`, or `%chin%` queries are combined, e.g. `DT[x %in% 1:100 & y %in% 1:200]` will not be optimized since `100 * 200 = 2e4 > 1e4`.

- Queries with multiple criteria on one column are *not* supported, e.g. `DT[x == 2 & x %in% c(2,5)]` is not supported.
- Queries with non-missing j are supported, e.g. `DT[x == 3 & y == 5, .(new = x-y)]` or `DT[x == 3 & y == 5, new := x-y]` are supported. Also extends to queries using `with = FALSE`.
- "notjoin" queries, i.e. queries that start with `!`, are only supported if there are no `&` connections, e.g. `DT[!x==3]` is supported, but `DT[!x==3 & y == 4]` is not.

If in doubt, whether your query benefits from optimization, call it with the `verbose = TRUE` argument. You should see "Optimized subsetting...".

Auto indexing: In case a query is optimized, but no appropriate key or index is found, `data.table` automatically creates an *index* on the first run. Any successive subsets on the same column then reuse this index to *binary search* (instead of *vector scan*) and is therefore fast. Auto indexing can be switched off with the global option `options(datatable.auto.index = FALSE)`. To switch off using existing indices set global option `options(datatable.use.index = FALSE)`.

See Also

setNumericRounding, getNumericRounding

Examples

```
## Not run:
old = options(datatable.optimize = Inf)

# Generate a big data.table with a relatively many columns
set.seed(1L)
DT = lapply(1:20, function(x) sample(c(-100:100), 5e6L, TRUE))
setDT(DT)[, id := sample(1e5, 5e6, TRUE)]
print(object.size(DT), units="MiB") # 400MiB, not huge, but will do

# 'order' optimisation
options(datatable.optimize = 1L) # optimisation 'on'
system.time(ans1 <- DT[order(id)])
options(datatable.optimize = 0L) # optimisation 'off'
system.time(ans2 <- DT[order(id)])
identical(ans1, ans2)

# optimisation of 'lapply(.SD, fun)'
options(datatable.optimize = 1L) # optimisation 'on'
system.time(ans1 <- DT[, lapply(.SD, min), by=id])
options(datatable.optimize = 0L) # optimisation 'off'
system.time(ans2 <- DT[, lapply(.SD, min), by=id])
identical(ans1, ans2)

# optimisation of 'mean'
options(datatable.optimize = 1L) # optimisation 'on'
system.time(ans1 <- DT[, lapply(.SD, mean), by=id])
system.time(ans2 <- DT[, lapply(.SD, base::mean), by=id])
identical(ans1, ans2)

# optimisation of 'c(.N, lapply(.SD, ))'
```

```
options(datatable.optimize = 1L) # optimisation 'on'
system.time(ans1 <- DT[, c(.N, lapply(.SD, min)), by=id])
options(datatable.optimize = 0L) # optimisation 'off'
system.time(ans2 <- DT[, c(N=.N, lapply(.SD, min)), by=id])
identical(ans1, ans2)

# GForce
options(datatable.optimize = 2L) # optimisation 'on'
system.time(ans1 <- DT[, lapply(.SD, median), by=id])
system.time(ans2 <- DT[, lapply(.SD, function(x) as.numeric(stats::median(x))), by=id])
identical(ans1, ans2)

# optimized subsets
options(datatable.optimize = 2L)
system.time(ans1 <- DT[id == 100L]) # vector scan
system.time(ans2 <- DT[id == 100L]) # vector scan
system.time(DT[id %in% 100:500])    # vector scan

options(datatable.optimize = 3L)
system.time(ans1 <- DT[id == 100L]) # index + binary search subset
system.time(ans2 <- DT[id == 100L]) # only binary search subset
system.time(DT[id %in% 100:500])    # only binary search subset again

# sensitivity to collate order
old_lc_collate = Sys.getlocale("LC_COLLATE")

if (old_lc_collate == "C") {
  Sys.setlocale("LC_COLLATE", "")
}
DT = data.table(
  grp = rep(1:2, each = 4L),
  var = c("A", "a", "0", "1", "B", "b", "0", "1")
)
options(datatable.optimize = Inf)
DT[, .(max(var), min(var)), by=grp]
# GForce is deactivated because of the ad-hoc column 'tolower(var)',
#   through which the result for 'max(var)' may also change
DT[, .(max(var), min(tolower(var))), by=grp]

Sys.setlocale("LC_COLLATE", old_lc_collate)
options(old)

## End(Not run)
```

`dcast.data.table` *Fast dcast for data.table*

Description

`dcast.data.table` is `data.table`'s long-to-wide reshaping tool. In the spirit of `data.table`, it is very fast and memory efficient, making it well-suited to handling large data sets in RAM. More

importantly, it is capable of handling very large data quite efficiently in terms of memory usage. dcast.data.table can also cast multiple value.var columns and accepts multiple functions to fun.aggregate. See Examples for more.

Usage

```
## S3 method for class 'data.table'
dcast(data, formula, fun.aggregate = NULL, sep = "_",
    ..., margins = NULL, subset = NULL, fill = NULL,
    drop = TRUE, value.var = guess(data),
    verbose = getOption("datatable.verbose"),
    value.var.in.dots = FALSE, value.var.in.LHSdots = value.var.in.dots,
    value.var.in.RHSdots = value.var.in.dots)
```

Arguments

data	A data.table.
formula	A formula of the form LHS ~ RHS to cast, see Details.
fun.aggregate	Should the data be aggregated before casting? If the formula doesn't identify a single observation for each cell, then aggregation defaults to length with a warning of class 'dt_missing_fun_aggregate_warning'. To use multiple aggregation functions, pass a list; see Examples.
sep	Character vector of length 1, indicating the separating character in variable names generated during casting. Default is _ for backwards compatibility.
...	Any other arguments that may be passed to the aggregating function.
margins	Not implemented yet. Should take variable names to compute margins on. A value of TRUE would compute all margins.
subset	Specified if casting should be done on a subset of the data. Ex: subset = .(col1 <= 5) or subset = .(variable != "January").
fill	Value with which to fill missing cells. If fill=NULL and missing cells are present, then fun.aggregate is used on a 0-length vector to obtain a fill value.
drop	FALSE will cast by including all missing combinations. c(FALSE, TRUE) will only include all missing combinations of formula LHS; c(TRUE, FALSE) will only include all missing combinations of formula RHS. See Examples.
value.var	Name of the column whose values will be filled to cast. Function guess() tries to, well, guess this column automatically, if none is provided. Cast multiple value.var columns simultaneously by passing their names as a character vector. See Examples.
verbose	Not used yet. May be dropped in the future or used to provide informative messages through the console.
value.var.in.dots	logical; value.var.in.dots = TRUE is shorthand to save setting both value.var.in.LHSdots = TRUE and value.var.in.RHSdots = TRUE.

value.var.in.LHSdots
: logical; if TRUE, ... in LHS of the formula includes value.var variables. The default is FALSE, so that ... represents all variables not otherwise mentioned in formula or value.var (including default/guessed value.var).

value.var.in.RHSdots
: logical; analogous to value.var.in.LHSdots above, but with respect to RHS of the formula.

Details

The cast formula takes the form LHS ~ RHS, ex: var1 + var2 ~ var3. The order of entries in the formula is essential. There are two special variables: . represents no variable, while ... represents all variables not otherwise mentioned in formula, and value.var depending on value.var.in.LHSdots and value.var.in.RHSdots arguments; see Examples.

When not all combinations of LHS & RHS values are present in the data, some or all (in accordance with drop) missing combinations will replaced with the value specified by fill. Note that fill will be converted to the class of value.var; see Examples.

dcast also allows value.var columns of type list.

When variable combinations in formula don't identify a unique value, fun.aggregate will have to be specified, which defaults to length. For the formula var1 ~ var2, this means there are some (var1, var2) combinations in the data corresponding to multiple rows (i.e. x is not unique by (var1, var2).

The aggregating function should take a vector as input and return a single value (or a list of length one) as output. In cases where value.var is a list, the function should be able to handle a list input and provide a single value or list of length one as output.

If the formula's LHS contains the same column more than once, ex: dcast(DT, x+x~ y), then the answer will have duplicate names. In those cases, the duplicate names are renamed using make.unique so that key can be set without issues.

Names for columns that are being cast are generated in the same order (separated by an underscore, _) from the (unique) values in each column mentioned in the formula RHS.

From v1.9.4, dcast tries to preserve attributes wherever possible.

From v1.9.6, it is possible to cast multiple value.var columns and also cast by providing multiple fun.aggregate functions. Multiple fun.aggregate functions should be provided as a list, for e.g., list(mean, sum, function(x) paste(x, collapse="")). value.var can be either a character vector or list of length one, or a list of length equal to length(fun.aggregate). When value.var is a character vector or a list of length one, each function mentioned under fun.aggregate is applied to every column specified under value.var column. When value.var is a list of length equal to length(fun.aggregate) each element of fun.aggregate is applied to each element of value.var column.

Historical note: dcast.data.table was originally designed as an enhancement to reshape2::dcast in terms of computing and memory efficiency. reshape2 has since been superseded in favour of tidyr, and dcast has had a generic defined within data.table since v1.9.6 in 2015, at which point the dependency between the packages became more etymological than programmatic. We thank the reshape2 authors for the inspiration.

Value

A keyed `data.table` that has been cast. The key columns are equal to the variables in the `formula` LHS in the same order.

See Also

`melt.data.table`, `rowid`, https://cran.r-project.org/package=reshape

Examples

```
ChickWeight = as.data.table(ChickWeight)
setnames(ChickWeight, tolower(names(ChickWeight)))
DT <- melt(as.data.table(ChickWeight), id.vars=2:4) # calls melt.data.table

# dcast is an S3 method in data.table from v1.9.6
dcast(DT, time ~ variable, fun.aggregate=mean)
dcast(DT, diet ~ variable, fun.aggregate=mean)
dcast(DT, diet+chick ~ time, drop=FALSE)
dcast(DT, diet+chick ~ time, drop=FALSE, fill=0)

# using subset
dcast(DT, chick ~ time, fun.aggregate=mean, subset=.(time < 10 & chick < 20))

# drop argument, #1512
DT <- data.table(v1 = c(1.1, 1.1, 1.1, 2.2, 2.2, 2.2),
                 v2 = factor(c(1L, 1L, 1L, 3L, 3L, 3L), levels=1:3),
                 v3 = factor(c(2L, 3L, 5L, 1L, 2L, 6L), levels=1:6),
                 v4 = c(3L, 2L, 2L, 5L, 4L, 3L))
# drop=TRUE
dcast(DT, v1+v2~v3, value.var='v4')                      # default is drop=TRUE
dcast(DT, v1+v2~v3, value.var='v4', drop=FALSE)       # all missing combinations of LHS and RHS
dcast(DT, v1+v2~v3, value.var='v4', drop=c(FALSE, TRUE)) # all missing combinations of LHS only
dcast(DT, v1+v2~v3, value.var='v4', drop=c(TRUE, FALSE)) # all missing combinations of RHS only

# using . and ...
DT <- data.table(v1 = rep(1:2, each = 6),
                 v2 = rep(rep(1:3, 2), each = 2),
                 v3 = rep(1:2, 6),
                 v4 = rnorm(6))
dcast(DT, ... ~ v3, value.var="v4") # same as v1+v2 ~ v3, value.var="v4"
dcast(DT, ... ~ v3, value.var="v4", value.var.in.dots=TRUE) # same as v1+v2+v4~v3, value.var="v4"
dcast(DT, v1+v2+v3 ~ ., value.var="v4")

## for each combination of (v1, v2), add up all values of v4
dcast(DT, v1+v2 ~ ., value.var="v4", fun.aggregate=sum)

# fill and types
dcast(DT, v2~v3, value.var='v1', fun.aggregate=length, fill=0L)  #  0L --> 0
dcast(DT, v2~v3, value.var='v4', fun.aggregate=length, fill=1.1) # 1.1 --> 1L

# multiple value.var and multiple fun.aggregate
DT = data.table(x=sample(5,20,TRUE), y=sample(2,20,TRUE),
```

```
                z=sample(letters[1:2], 20,TRUE), d1=runif(20), d2=1L)
# multiple value.var
dcast(DT, x+y ~ z, fun.aggregate=sum, value.var=c("d1","d2"))
# multiple fun.aggregate
dcast(DT, x+y ~ z, fun.aggregate=list(sum, mean), value.var="d1")
# multiple fun.agg and value.var (all combinations)
dcast(DT, x+y ~ z, fun.aggregate=list(sum, mean), value.var=c("d1", "d2"))
# multiple fun.agg and value.var (one-to-one)
dcast(DT, x+y ~ z, fun.aggregate=list(sum, mean), value.var=list("d1", "d2"))
```

---

duplicated *Determine Duplicate Rows*

Description

duplicated returns a logical vector indicating which rows of a data.table are duplicates of a row with smaller subscripts.

unique returns a data.table with duplicated rows removed, by columns specified in by argument. When no by then duplicated rows by all columns are removed.

anyDuplicated returns the *index* i of the first duplicated entry if there is one, and 0 otherwise.

uniqueN is equivalent to length(unique(x)) when x is an atomic vector, and nrow(unique(x)) when x is a data.frame or data.table. The number of unique rows are computed directly without materialising the intermediate unique data.table and is therefore faster and memory efficient.

Usage

```
## S3 method for class 'data.table'
duplicated(x, incomparables=FALSE, fromLast=FALSE, by=seq_along(x), ...)

## S3 method for class 'data.table'
unique(x, incomparables=FALSE, fromLast=FALSE,
by=seq_along(x), cols=NULL, ...)

## S3 method for class 'data.table'
anyDuplicated(x, incomparables=FALSE, fromLast=FALSE, by=seq_along(x), ...)

uniqueN(x, by=if (is.list(x)) seq_along(x) else NULL, na.rm=FALSE)
```

Arguments

x	A data.table. uniqueN accepts atomic vectors and data.frames as well.
...	Not used at this time.
incomparables	Not used. Here for S3 method consistency.
fromLast	Logical indicating if duplication should be considered from the reverse side. For duplicated, this means the last (or rightmost) of identical elements will correspond to duplicated = FALSE. For unique, this means the last (or rightmost) of identical elements will be kept. See examples.

by	character or integer vector indicating which combinations of columns from x to use for uniqueness checks. By default all columns are being used. That was changed recently for consistency to data.frame methods. In version < 1.9.8 default was key(x).
cols	Columns (in addition to by) from x to include in the resulting data.table.
na.rm	Logical (default is FALSE). Should missing values (including NaN) be removed?

Details

Because data.tables are usually sorted by key, tests for duplication are especially quick when only the keyed columns are considered. Unlike unique.data.frame, paste is not used to ensure equality of floating point data. It is instead accomplished directly and is therefore quite fast. data.table provides setNumericRounding to handle cases where limitations in floating point representation is undesirable.

v1.9.4 introduces anyDuplicated method for data.tables and is similar to base in functionality. It also implements the logical argument fromLast for all three functions, with default value FALSE.

Note: When cols is specified, the resulting table will have columns c(by, cols), in that order.

Value

duplicated returns a logical vector of length nrow(x) indicating which rows are duplicates.

unique returns a data table with duplicated rows removed.

anyDuplicated returns a integer value with the index of first duplicate. If none exists, 0L is returned.

uniqueN returns the number of unique elements in the vector, data.frame or data.table.

See Also

setNumericRounding, data.table, duplicated, unique, all.equal, fsetdiff, funion, fintersect, fsetequal

Examples

```
DT <- data.table(A = rep(1:3, each=4), B = rep(1:4, each=3),
                 C = rep(1:2, 6), key = c("A", "B"))
duplicated(DT)
unique(DT)

duplicated(DT, by="B")
unique(DT, by="B")

duplicated(DT, by=c("A", "C"))
unique(DT, by=c("A", "C"))

DT = data.table(a=c(2L,1L,2L), b=c(1L,2L,1L))   # no key
unique(DT)                  # rows 1 and 2 (row 3 is a duplicate of row 1)

DT = data.table(a=c(3.142, 4.2, 4.2, 3.142, 1.223, 1.223), b=rep(1,6))
```

```
unique(DT)                  # rows 1,2 and 5

DT = data.table(a=tan(pi*(1/4 + 1:10)), b=rep(1,10))   # example from ?all.equal
length(unique(DT$a))        # 10 strictly unique floating point values
all.equal(DT$a,rep(1,10))   # TRUE, all within tolerance of 1.0
DT[,which.min(a)]           # row 10, the strictly smallest floating point value
identical(unique(DT),DT[1])  # TRUE, stable within tolerance
identical(unique(DT),DT[10]) # FALSE

# fromLast = TRUE vs. FALSE
DT <- data.table(A = c(1, 1, 2, 2, 3), B = c(1, 2, 1, 1, 2), C = c("a", "b", "a", "b", "a"))

duplicated(DT, by="B", fromLast=FALSE) # rows 3,4,5 are duplicates
unique(DT, by="B", fromLast=FALSE) # equivalent: DT[!duplicated(DT, by="B", fromLast=FALSE)]

duplicated(DT, by="B", fromLast=TRUE) # rows 1,2,3 are duplicates
unique(DT, by="B", fromLast=TRUE) # equivalent: DT[!duplicated(DT, by="B", fromLast=TRUE)]

# anyDuplicated
anyDuplicated(DT, by=c("A", "B"))    # 3L
any(duplicated(DT, by=c("A", "B")))  # TRUE

# uniqueN, unique rows on key columns
uniqueN(DT, by = key(DT))
# uniqueN, unique rows on all columns
uniqueN(DT)
# uniqueN while grouped by "A"
DT[, .(uN=uniqueN(.SD)), by=A]

# uniqueN's na.rm=TRUE
x = sample(c(NA, NaN, runif(3)), 10, TRUE)
uniqueN(x, na.rm = FALSE) # 5, default
uniqueN(x, na.rm=TRUE) # 3
```

`fcase` *fcase*

Description

`fcase` is a fast implementation of SQL `CASE WHEN` statement for R. Conceptually, `fcase` is a nested version of `fifelse` (with smarter implementation than manual nesting). It is comparable to `dplyr::case_when` and supports `bit64`'s `integer64` and `nanotime` classes.

Usage

```
fcase(..., default=NA)
```

Arguments

...	A sequence consisting of logical condition (when)-resulting value (value) *pairs* in the following order when1, value1, when2, value2, ..., whenN, valueN. Logical conditions when1, when2, ..., whenN must all have the same length, type and attributes. Each value may either share length with when or be length 1. Please see Examples section for further details.
default	Default return value, NA by default, for when all of the logical conditions when1, when2, ..., whenN are FALSE or missing for some entries.

Details

fcase evaluates each when-value pair in order, until it finds a when that is TRUE. It then returns the corresponding value. During evaluation, value will be evaluated regardless of whether the corresponding when is TRUE or not, which means recursive calls should be placed in the last when-value pair, see Examples.

default is always evaluated, regardless of whether it is returned or not.

Value

Vector with the same length as the logical conditions (when) in ..., filled with the corresponding values (value) from ..., or eventually default. Attributes of output values value1, value2, ...valueN in ... are preserved.

See Also

fifelse

Examples

```
x = 1:10
fcase(
x < 5L, 1L,
x > 5L, 3L
)

fcase(
x < 5L, 1L:10L,
x > 5L, 3L:12L
)

# Lazy evaluation example
fcase(
x < 5L, 1L,
x >= 5L, 3L,
x == 5L, stop("provided value is an unexpected one!")
)

# fcase preserves attributes, example with dates
fcase(
x < 5L, as.Date("2019-10-11"),
```

```
x > 5L, as.Date("2019-10-14")
)

# fcase example with factor; note the matching levels
fcase(
x < 5L, factor("a", levels=letters[1:3]),
x > 5L, factor("b", levels=letters[1:3])
)

# Example of using the 'default' argument
fcase(
x < 5L, 1L,
x > 5L, 3L,
default = 5L
)

# fcase can be used for recursion, unlike fifelse
# Recursive function to calculate the Greatest Common Divisor
gcd_dt = function(x,y) {
  r = x%%y
  fcase(!r, y, r, gcd_dt(x, y)) # Recursive call must be in the last when-value pair
}
gcd_dt(10L, 1L)
```

`fcoalesce` *Coalescing missing values*

Description

Fill in missing values in a vector by successively pulling from candidate vectors in order. As per the ANSI SQL function COALESCE, `dplyr::coalesce` and `hutils::coalesce`. Unlike `BBmisc::coalesce` which just returns the first non-NULL vector. Written in C, and multithreaded for numeric and factor types.

Usage

```
fcoalesce(..., nan=NA)
```

Arguments

`...`	A set of same-class vectors. These vectors can be supplied as separate arguments or as a single plain list, data.table or data.frame, see examples.
`nan`	Either `NaN` or `NA`; if `NaN`, then `NaN` is treated as distinct from `NA`, otherwise they are treated the same during replacement (double columns only).

Details

Factor type is supported only when the factor levels of each item are equal.

`NaN` is considered missing (note `is.na(NaN)` and `all.equal(NA_real_, NaN)` are both `TRUE`).

Value

Atomic vector of the same type and length as the first vector, having NA values replaced by corresponding non-NA values from the other vectors. If the first item is NULL, the result is NULL.

See Also

fifelse, nafill

Examples

```
x = c(11L, NA, 13L, NA, 15L, NA)
y = c(NA, 12L, 5L, NA, NA, NA)
z = c(11L, NA, 1L, 14L, NA, NA)
fcoalesce(x, y, z)
fcoalesce(list(x,y,z))   # same
fcoalesce(x, list(y,z))  # same
x_num = c(NaN, NA_real_, 3.0)
fcoalesce(x_num, 1)           # default: NaN treated as missing -> c(1, 1, 3)
fcoalesce(x_num, 1, nan=NaN)  # preserve NaN -> c(NaN, 1, 3)
```

---

fctr *Create a factor retaining original ordering*

Description

Creates a factor.

By default, the output will have its levels in the original order, i.e., levels = unique(x), as opposed to factor's default where levels = sort(unique(x)).

Usage

```
fctr(x, levels=unique(x), ..., sort=FALSE, rev=FALSE)
```

Arguments

x	Object to be turned into a factor.
levels	Levels for the new factor; unique(x) by default.
...	Other arguments passed to factor.
sort	Logical, default FALSE. Should levels be sorted?
rev	Logical, default FALSE. Should levels be reversed? Applied *after* sort.

Value

Factor vector having levels ordered according to the order of elements in input and arguments sort, rev.

Examples

```
levels(factor(c("b","a","c")))
levels(fctr(c("b","a","c")))
levels(fctr(c("b","a","c"), rev=TRUE))
levels(fctr(c("b","a","c"), sort=TRUE))
levels(fctr(c("b","a","c"), sort=TRUE, rev=TRUE))
```

`fdroplevels` *Fast droplevels*

Description

Similar to `base::droplevels` but *much faster*.

Usage

```
fdroplevels(x, exclude = if (anyNA(levels(x))) NULL else NA, ...)
setdroplevels(x, except = NULL, exclude = NULL)

## S3 method for class 'data.table'
droplevels(x, except = NULL, exclude, ...)
```

Arguments

`x`	`factor` or `data.table` where unused levels should be dropped.
`exclude`	A `character` vector of factor levels which are dropped no matter of presented or not.
`except`	An `integer` vector of indices of data.table columns which are not modified by dropping levels.
`...`	further arguments passed to methods

Value

`fdroplevels` returns a `factor`.

`droplevels` returns a `data.table` where levels are dropped at factor columns.

See Also

`data.table`, `duplicated`, `unique`

Examples

```
# on vectors
x = factor(letters[1:10])
fdroplevels(x[1:5])
# exclude levels from drop
fdroplevels(x[1:5], exclude = c("a", "c"))

# on data.table
DT = data.table(a = factor(1:10), b = factor(letters[1:10]))
droplevels(head(DT))[["b"]]
# exclude levels
droplevels(head(DT), exclude = c("b", "c"))[["b"]]
# except columns from drop
droplevels(head(DT), except = 2)[["b"]]
droplevels(head(DT), except = 1)[["b"]]
```

---

fifelse *Fast ifelse*

Description

`fifelse` is a faster and more robust replacement of `ifelse`. It is comparable to `dplyr::if_else` and `hutils::if_else`. It returns a value with the same length as `test` filled with corresponding values from `yes`, `no` or eventually `na`, depending on `test`. Supports `bit64`'s `integer64` and `nanotime` classes.

Usage

```
fifelse(test, yes, no, na=NA)
```

Arguments

`test`	A logical vector.
`yes, no`	Values to return depending on `TRUE`/`FALSE` element of `test`. They must be the same type and be either length `1` or the same length of `test`.
`na`	Value to return if an element of `test` is `NA`. It must be the same type as `yes` and `no` and its length must be either `1` or the same length of `test`. Default value `NA`. `NULL` is treated as `NA`.

Details

In contrast to `ifelse` attributes are copied from the first non-`NA` argument to the output. This is useful when returning `Date`, `factor` or other classes.

Unlike `ifelse`, `fifelse` evaluates both `yes` and `no` arguments for type checking regardless of the result of `test`. This means that neither `yes` nor `no` should be recursive function calls. For recursion, use `fcase` instead.

Value

A vector of the same length as `test` and attributes as `yes`. Data values are taken from the values of `yes` and `no`, eventually `na`.

See Also

`fcoalesce`

`fcase`

Examples

```
x = c(1:4, 3:2, 1:4)
fifelse(x > 2L, x, x - 1L)

# unlike ifelse, fifelse preserves attributes, taken from the 'yes' argument
dates = as.Date(c("2011-01-01","2011-01-02","2011-01-03","2011-01-04","2011-01-05"))
ifelse(dates == "2011-01-01", dates - 1, dates)
fifelse(dates == "2011-01-01", dates - 1, dates)
yes = factor(c("a","b","c"))
no = yes[1L]
ifelse(c(TRUE,FALSE,TRUE), yes, no)
fifelse(c(TRUE,FALSE,TRUE), yes, no)

# Example of using the 'na' argument
fifelse(test = c(-5L:5L < 0L, NA), yes = 1L, no = 0L, na = 2L)

# Example showing both 'yes' and 'no' arguments are evaluated, unlike ifelse
fifelse(1 == 1, print("yes"), print("no"))
ifelse(1 == 1, print("yes"), print("no"))
```

---

foverlaps *Fast overlap joins*

Description

A *fast* binary-search based *overlap join* of two `data.tables`. This is very much inspired by `findOverlaps` function from the Bioconductor package `IRanges` (see link below under `See Also`).

Usually, `x` is a very large data.table with small interval ranges, and `y` is much smaller *keyed* `data.table` with relatively larger interval spans. For a usage in `genomics`, see the examples section.

NOTE: This is still under development, meaning it is stable, but some features are yet to be implemented. Also, some arguments and/or the function name itself could be changed.

Usage

```
foverlaps(x, y, by.x = key(x) %||% key(y),
    by.y = key(y), maxgap = 0L, minoverlap = 1L,
    type = c("any", "within", "start", "end", "equal"),
```

```
    mult = c("all", "first", "last"),
    nomatch = NA,
    which = FALSE, verbose = getOption("datatable.verbose"))
```

Arguments

x, y	data.tables. y needs to be keyed, but not necessarily x. See examples.
by.x, by.y	A vector of column names (or numbers) to compute the overlap joins. The last two columns in both by.x and by.y should each correspond to the start and end interval columns in x and y respectively. We should always have start <= end. If x is keyed, by.x is equal to key(x), else key(y). by.y defaults to key(y).
maxgap	Non-negative integer, i.e., maxgap >= 0. Default is 0 (no gap). For intervals [a,b] and [c,d], where a<=b and c<=d, when c > b or d < a, the two intervals don't overlap. If the gap between these two intervals is <= maxgap, these two intervals are considered as overlapping. Note: This is not yet implemented.
minoverlap	Positive integer, i.e., minoverlap > 0. Default is 1. For intervals [a,b] and [c,d], where a<=b and c<=d, when c<=b and d>=a, the two intervals overlap. If the length of overlap between these two intervals is >= minoverlap, then these two intervals are considered to be overlapping. Note: This is not yet implemented.
type	Default value is any. Allowed values are any, within, start, end and equal. The types shown here are identical in functionality to the function findOverlaps in the bioconductor package IRanges. Let [a,b] and [c,d] be intervals in x and y with a<=b and c<=d. For type="start", the intervals overlap iff a == c. For type="end", the intervals overlap iff b == d. For type="within", the intervals overlap iff a>=c and b<=d. For type="equal", the intervals overlap iff a==c and b==d. For type="any", as long as c<=b and d>=a, they overlap. In addition to these requirements, they also have to satisfy the minoverlap argument as explained above. NB: maxgap argument, when > 0, is to be interpreted according to the type of the overlap. This will be updated once maxgap is implemented.
mult	When multiple rows in y match to the row in x, mult=. controls which values are returned - "all" (default), "first" or "last".
nomatch	When a row (with interval say, [a,b]) in x has no match in y, nomatch=NA (default) means NA is returned for y's non-by.y columns for that row of x. nomatch=NULL (or 0 for backward compatibility) means no rows will be returned for that row of x.
which	When TRUE, if mult="all" returns a two column data.table with the first column corresponding to x's row number and the second corresponding to y's. When nomatch=NA, no matches return NA for y, and if nomatch=NULL, those rows where no match is found will be skipped; if mult="first" or "last", a vector of length equal to the number of rows in x is returned, with no-match entries filled with NA or 0 corresponding to the nomatch argument. Default is FALSE, which returns a join with the rows in y.

verbose — TRUE turns on status and information messages to the console. Turn this on by default using options(datatable.verbose=TRUE). The quantity and types of verbosity may be expanded in future.

Details

Very briefly, foverlaps() collapses the two-column interval in y to one-column of *unique* values to generate a lookup table, and then performs the join depending on the type of overlap, using the already available binary search feature of data.table. The time (and space) required to generate the lookup is therefore proportional to the number of unique values present in the interval columns of y when combined together.

Overlap joins takes advantage of the fact that y is sorted to speed-up finding overlaps. Therefore y has to be keyed (see ?setkey) prior to running foverlaps(). A key on x is not necessary, although it *might* speed things further. The columns in by.x argument should correspond to the columns specified in by.y. The last two columns should be the *interval* columns in both by.x and by.y. The first interval column in by.x should always be <= the second interval column in by.x, and likewise for by.y. The storage.mode of the interval columns must be either double or integer. It therefore works with bit64::integer64 type as well.

The lookup generation step could be quite time consuming if the number of unique values in y are too large (ex: in the order of tens of millions). There might be improvements possible by constructing lookup using RLE, which is a pending feature request. However most scenarios will not have too many unique values for y.

Value

A new data.table by joining over the interval columns (along with other additional identifier columns) specified in by.x and by.y.

NB: When which=TRUE: a) mult="first" or "last" returns a vector of matching row numbers in y, and b) when mult="all" returns a data.table with two columns with the first containing row numbers of x and the second column with corresponding row numbers of y.

nomatch=NA|NULL also influences whether non-matching rows are returned or not, as explained above.

See Also

data.table, https://www.bioconductor.org/packages/release/bioc/html/IRanges.html, setNumericRounding

Examples

```
require(data.table)
## simple example:
x = data.table(start=c(5,31,22,16), end=c(8,50,25,18), val2 = 7:10)
y = data.table(start=c(10, 20, 30), end=c(15, 35, 45), val1 = 1:3)
setkey(y, start, end)
foverlaps(x, y, type="any", which=TRUE) ## return overlap indices
foverlaps(x, y, type="any") ## return overlap join
foverlaps(x, y, type="any", mult="first") ## returns only first match
foverlaps(x, y, type="within") ## matches iff 'x' is within 'y'
```

```
## with extra identifiers (ex: in genomics)
x = data.table(chr=c("Chr1", "Chr1", "Chr2", "Chr2", "Chr2"),
               start=c(5,10, 1, 25, 50), end=c(11,20,4,52,60))
y = data.table(chr=c("Chr1", "Chr1", "Chr2"), start=c(1, 15,1),
               end=c(4, 18, 55), geneid=letters[1:3])
setkey(y, chr, start, end)
foverlaps(x, y, type="any", which=TRUE)
foverlaps(x, y, type="any")
foverlaps(x, y, type="any", nomatch=NULL)
foverlaps(x, y, type="within", which=TRUE)
foverlaps(x, y, type="within")
foverlaps(x, y, type="start")

## x and y have different column names - specify by.x
x = data.table(seq=c("Chr1", "Chr1", "Chr2", "Chr2", "Chr2"),
               start=c(5,10, 1, 25, 50), end=c(11,20,4,52,60))
y = data.table(chr=c("Chr1", "Chr1", "Chr2"), start=c(1, 15,1),
               end=c(4, 18, 55), geneid=letters[1:3])
setkey(y, chr, start, end)
foverlaps(x, y, by.x=c("seq", "start", "end"),
            type="any", which=TRUE)
```

`frank` *Fast rank*

Description

Similar to `base::rank` but *much faster*. And it accepts vectors, lists, `data.frames` or `data.tables` as input. In addition to the `ties.method` possibilities provided by `base::rank`, it also provides `ties.method="dense"`.

Like `forder`, sorting is done in "C-locale"; in particular, this may affect how capital/lowercase letters are ranked. See Details on `forder` for more.

`bit64::integer64` type is also supported.

Usage

```
frank(x, ..., na.last=TRUE, ties.method=c("average",
  "first", "last", "random", "max", "min", "dense"))

frankv(x, cols=seq_along(x), order=1L, na.last=TRUE,
      ties.method=c("average", "first", "last", "random",
        "max", "min", "dense"))
```

Arguments

x	A vector, or list with all its elements identical in length or data.frame or data.table.
...	Only for lists, data.frames and data.tables. The columns to calculate ranks based on. Do not quote column names. If ... is missing, all columns are considered by default. To sort by a column in descending order prefix "-", e.g., frank(x, a, -b, c). -b works when b is of type character as well.
cols	A character vector of column names (or numbers) of x, for which to obtain ranks.
order	An integer vector with only possible values of 1 and -1, corresponding to ascending and descending order. The length of order must be either 1 or equal to that of cols. If length(order) == 1, it is recycled to length(cols).
na.last	Control treatment of NAs. If TRUE, missing values in the data are put last; if FALSE, they are put first; if NA, they are removed; if "keep" they are kept with rank NA.
ties.method	A character string specifying how ties are treated, see Details.

Details

To be consistent with other data.table operations, NAs are considered identical to other NAs (and NaNs to other NaNs), unlike base::rank. Therefore, for na.last=TRUE and na.last=FALSE, NAs (and NaNs) are given identical ranks, unlike rank.

frank is not limited to vectors. It accepts data.tables (and lists and data.frames) as well. It accepts unquoted column names (with names preceded with a - sign for descending order, even on character vectors), for e.g., frank(DT, a, -b, c, ties.method="first") where a,b,c are columns in DT. The equivalent in frankv is the order argument.

In addition to the ties.method values possible using base's rank, it also provides another additional argument "dense" which returns the ranks without any gaps in the ranking. See examples.

Value

A numeric vector of length equal to NROW(x) (unless na.last = NA, when missing values are removed). The vector is of integer type unless ties.method = "average" when it is of double type (irrespective of ties).

See Also

data.table, setkey, setorder

Examples

```
# on vectors
x = c(4, 1, 4, NA, 1, NA, 4)
# NAs are considered identical (unlike base R)
# default is average
frankv(x) # na.last=TRUE
frankv(x, na.last=FALSE)
```

```
# ties.method = min
frankv(x, ties.method="min")
# ties.method = dense
frankv(x, ties.method="dense")

# on data.table
DT = data.table(x, y=c(1, 1, 1, 0, NA, 0, 2))
frankv(DT, cols="x") # same as frankv(x) from before
frankv(DT, cols="x", na.last="keep")
frankv(DT, cols="x", ties.method="dense", na.last=NA)
frank(DT, x, ties.method="dense", na.last=NA) # equivalent of above using frank
# on both columns
frankv(DT, ties.method="first", na.last="keep")
frank(DT, ties.method="first", na.last="keep") # equivalent of above using frank

# order argument
frank(DT, x, -y, ties.method="first")
# equivalent of above using frankv
frankv(DT, order=c(1L, -1L), ties.method="first")
```

`fread` *Fast and friendly file finagler*

Description

Similar to `read.csv()` and `read.delim()` but faster and more convenient. All controls such as `sep`, `colClasses` and `nrows` are automatically detected.

`bit64::integer64`, `IDate`, and `POSIXct` types are also detected and read directly without needing to read as character before converting.

`fread` is for *regular* delimited files; i.e., where every row has the same number of columns. In future, secondary separator (`sep2`) may be specified *within* each column. Such columns will be read as type `list` where each cell is itself a vector.

Usage

```
fread(input, file, text, cmd, sep="auto", sep2="auto", dec="auto", quote="\"",
nrows=Inf, header="auto",
na.strings=getOption("datatable.na.strings","NA"),  # due to change to ""; see NEWS
stringsAsFactors=FALSE, verbose=getOption("datatable.verbose", FALSE),
skip="__auto__", select=NULL, drop=NULL, colClasses=NULL,
integer64=getOption("datatable.integer64", "integer64"),
col.names,
check.names=FALSE, encoding="unknown",
strip.white=TRUE, fill=FALSE, blank.lines.skip=FALSE, comment.char="",
key=NULL, index=NULL,
showProgress=getOption("datatable.showProgress", interactive()),
data.table=getOption("datatable.fread.datatable", TRUE),
```

```
nThread=getDTthreads(verbose),
logical01=getOption("datatable.logical01", FALSE),
logicalYN=getOption("datatable.logicalYN", FALSE),
keepLeadingZeros = getOption("datatable.keepLeadingZeros", FALSE),
yaml=FALSE, tmpdir=tempdir(), tz="UTC"
)
```

Arguments

input	A single character string. The value is inspected and deferred to either file= (if no \n present), text= (if at least one \n is present) or cmd= (if no \n is present, at least one space is present, and it isn't a file name). Exactly one of input=, file=, text=, or cmd= should be used in the same call.
file	File name in working directory, path to file (passed through path.expand for convenience), or a URL starting http://, file://, etc. Compressed files with extension '.gz' and '.bz2' are supported if the R.utils package is installed.
text	The input data itself as a character vector of one or more lines, for example as returned by readLines().
cmd	A shell command that pre-processes the file; e.g. fread(cmd=paste("grep",word,"filename")). See Details.
sep	The separator between columns. Defaults to the character in the set [,\t |;:] that separates the sample of rows into the most number of lines with the same number of fields. Use NULL or "" to specify no separator; i.e. each line a single character column like base::readLines does.
sep2	The separator *within* columns. A list column will be returned where each cell is a vector of values. This is much faster using less working memory than strsplit afterwards or similar techniques. For each column sep2 can be different and is the first character in the same set above [,\t |;], other than sep, that exists inside each field outside quoted regions in the sample. NB: sep2 is not yet implemented.
nrows	The maximum number of rows to read. Unlike read.table, you do not need to set this to an estimate of the number of rows in the file for better speed because that is already automatically determined by fread almost instantly using the large sample of lines. nrows=0 returns the column names and typed empty columns determined by the large sample; useful for a dry run of a large file or to quickly check format consistency of a set of files before starting to read any of them.
header	Does the first data line contain column names? Defaults according to whether every non-empty field on the first data line is type character. If so, or TRUE is supplied, any empty column names are given a default name.
na.strings	A character vector of strings which are to be interpreted as NA values. By default, ",," for columns of all types, including type character is read as NA for consistency. ,"", is unambiguous and read as an empty string. To read ,NA, as NA, set na.strings="NA". To read ,, as blank string "", set na.strings=NULL. When they occur in the file, the strings in na.strings should not appear quoted since that is how the string literal ,"NA", is distinguished from ,NA,, for example, when na.strings="NA".

stringsAsFactors
: Convert all or some character columns to factors? Acceptable inputs are TRUE, FALSE, or a decimal value between 0.0 and 1.0. For stringsAsFactors = FALSE, all string columns are stored as character vs. all stored as factor when TRUE. When stringsAsFactors = p for 0 <= p <= 1, string columns col are stored as factor if uniqueN(col)/nrow < p.

verbose
: Be chatty and report timings?

skip
: If 0 (default) start on the first line and from there finds the first row with a consistent number of columns. This automatically avoids irregular header information before the column names row. skip>0 means ignore the first skip rows manually. skip="string" searches for "string" in the file (e.g. a substring of the column names row) and starts on that line (inspired by read.xls in package gdata).

select
: A vector of column names or numbers to keep, drop the rest. select may specify types too in the same way as colClasses; i.e., a vector of colname=type pairs, or a list of type=col(s) pairs. In all forms of select, the order that the columns are specified determines the order of the columns in the result.

drop
: Vector of column names or numbers to drop, keep the rest.

colClasses
: As in utils::read.csv; i.e., an unnamed vector of types corresponding to the columns in the file, or a named vector specifying types for a subset of the columns by name. The default, NULL means types are inferred from the data in the file. Further, data.table supports a named list of vectors of column names *or numbers* where the list names are the class names; see examples. The list form makes it easier to set a batch of columns to be a particular class. When column numbers are used in the list form, they refer to the column number in the file not the column number after select or drop has been applied. If type coercion results in an error, introduces NAs, or would result in loss of accuracy, the coercion attempt is aborted for that column with warning and the column's type is left unchanged. If you really desire data loss (e.g. reading 3.14 as integer) you have to truncate such columns afterwards yourself explicitly so that this is clear to future readers of your code.

integer64
: "integer64" (default) reads columns detected as containing integers larger than 2^31 as type bit64::integer64. Alternatively, "double"|"numeric" reads as utils::read.csv does; i.e., possibly with loss of precision and if so silently. Or, "character".

dec
: The decimal separator as in utils::read.csv. When "auto" (the default), an attempt is made to decide whether "." or "," is more suitable for this input. See details.

col.names
: A vector of optional names for the variables (columns). The default is to use the header column if present or detected, or if not "V" followed by the column number. This is applied after check.names and before key and index.

check.names
: default is FALSE. If TRUE then the names of the variables in the data.table are checked to ensure that they are syntactically valid variable names. If necessary they are adjusted (by make.names) so that they are, and also to ensure that there are no duplicates.

encoding — default is "unknown". Other possible options are "UTF-8" and "Latin-1". Note: it is not used to re-encode the input, rather enables handling of encoded strings in their native encoding.

quote — By default ("\""), if a field starts with a double quote, fread handles embedded quotes robustly as explained under Details. If it fails, then another attempt is made to read the field *as is*, i.e., as if quotes are disabled. By setting quote="", the field is always read as if quotes are disabled. It is not expected to ever need to pass anything other than \"\" to quote; i.e., to turn it off.

strip.white — Logical, default TRUE, in which case leading and trailing whitespace is stripped from unquoted "character" fields. "numeric" fields are always stripped of leading and trailing whitespace.

fill — logical or integer (default is FALSE). If TRUE then in case the rows have unequal length, number of columns is estimated and blank fields are implicitly filled. If an integer is provided it is used as an upper bound for the number of columns. If fill=Inf then the whole file is read for detecting the number of columns.

blank.lines.skip — logical, default is FALSE. If TRUE blank lines in the input are ignored.

comment.char — Character vector of length one containing a single character of an empty string. Any text after the comment character in a line is ignored, including skipping comment-only lines. Use "" to turn off the interpretation of comments altogether.

key — Character vector of one or more column names which is passed to setkey. Only valid when argument data.table=TRUE. Where applicable, this should refer to column names given in col.names.

index — Character vector or list of character vectors of one or more column names which is passed to setindexv. As with key, comma-separated notation like index="x,y,z" is accepted for convenience. Only valid when argument data.table=TRUE. Where applicable, this should refer to column names given in col.names.

showProgress — TRUE displays progress on the console if the ETA is greater than 3 seconds. It is produced in fread's C code where the very nice (but R level) txtProgressBar and tkProgressBar are not easily available.

data.table — TRUE returns a data.table. FALSE returns a data.frame. The default for this argument can be changed with options(datatable.fread.datatable=FALSE).

nThread — The number of threads to use. Experiment to see what works best for your data on your hardware.

logical01 — If TRUE a column containing only 0s and 1s will be read as logical, otherwise as integer.

logicalYN — If TRUE a column containing only Ys and Ns will be read as logical, otherwise as character.

keepLeadingZeros — If TRUE a column containing numeric data with leading zeros will be read as character, otherwise leading zeros will be removed and converted to numeric.

yaml — If TRUE, fread will attempt to parse (using yaml.load) the top of the input as YAML, and further to glean parameters relevant to improving the performance of fread on the data itself. The entire YAML section is returned as parsed into a list in the yaml_metadata attribute. See Details.

tmpdir — Directory to use as the tmpdir argument for any tempfile calls, e.g. when the input is a URL or a shell command. The default is tempdir() which can be controlled by setting TMPDIR before starting the R session; see base::tempdir.

tz — Relevant to datetime values which have no Z or UTC-offset at the end, i.e. *unmarked* datetime, as written by utils::write.csv. The default tz="UTC" reads unmarked datetime as UTC POSIXct efficiently. tz="" reads unmarked datetime as type character (slowly) so that as.POSIXct can interpret (slowly) the character datetimes in local timezone; e.g. by using "POSIXct" in colClasses=. Note that fwrite() by default writes datetime in UTC including the final Z and therefore fwrite's output will be read by fread consistently and quickly without needing to use tz= or colClasses=. If the TZ environment variable is set to "UTC" (or "" on non-Windows where unset vs "" is significant) then the R session's timezone is already UTC and tz="" will result in unmarked datetimes being read as UTC POSIXct. For more information, please see the news items from v1.13.0 and v1.14.0.

Details

A sample of 10,000 rows is used for a very good estimate of column types. 100 contiguous rows are read from 100 equally spaced points throughout the file including the beginning, middle and the very end. This results in a better guess when a column changes type later in the file (e.g. blank at the beginning/only populated near the end, or 001 at the start but 0A0 later on). This very good type guess enables a single allocation of the correct type up front once for speed, memory efficiency and convenience of avoiding the need to set colClasses after an error. Even though the sample is large and jumping over the file, it is almost instant regardless of the size of the file because a lazy on-demand memory map is used. If a jump lands inside a quoted field containing newlines, each newline is tested until 5 lines are found following it with the expected number of fields. The lowest type for each column is chosen from the ordered list: logical, integer, integer64, double, character. Rarely, the file may contain data of a higher type in rows outside the sample (referred to as an out-of-sample type exception). In this event fread will *automatically* reread just those columns from the beginning so that you don't have the inconvenience of having to set colClasses yourself; particularly helpful if you have a lot of columns. Such columns must be read from the beginning to correctly distinguish "00" from "000" when those have both been interpreted as integer 0 due to the sample but 00A occurs out of sample. Set verbose=TRUE to see a detailed report of the logic deployed to read your file.

There is no line length limit, not even a very large one. Since we are encouraging list columns (i.e. sep2) this has the potential to encourage longer line lengths. So the approach of scanning each line into a buffer first and then rescanning that buffer is not used. There are no buffers used in fread's C code at all. The field width limit is limited by R itself: the maximum width of a character string (currently 2^31-1 bytes, 2GiB).

The filename extension (such as .csv) is irrelevant for "auto" sep and sep2. Separator detection is entirely driven by the file contents. This can be useful when loading a set of different files which may not be named consistently, or may not have the extension .csv despite being csv. Some datasets have been collected over many years, one file per day for example. Sometimes the file name format has changed at some point in the past or even the format of the file itself. So the idea is that you can loop fread through a set of files and as long as each file is regular and delimited, fread can read them all. Whether they all stack is another matter but at least each one is read quickly without you needing to vary colClasses in read.table or read.csv.

If an empty line is encountered then reading stops there with warning if any text exists after the empty line such as a footer. The first line of any text discarded is included in the warning message. Unless, it is single-column input. In that case blank lines are significant (even at the very end) and represent NA in the single column. So that `fread(fwrite(DT))==DT`. This default behaviour can be controlled using `blank.lines.skip=TRUE|FALSE`.

Line endings: All known line endings are detected automatically: `\n` (*NIX including Mac), `\r\n` (Windows CRLF), `\r` (old Mac) and `\n\r` (just in case). There is no need to convert input files first. `fread` running on any architecture will read a file from any architecture. Both `\r` and `\n` may be embedded in character strings (including column names) provided the field is quoted.

Decimal separator: `dec` is used to parse numeric fields as the separator between integral and fractional parts. When `dec='auto'`, during column type detection, when a field is a candidate for being numeric (i.e., parsing as lower types has already failed), `dec='.'` is tried, and, if it fails to create a numeric field, `dec=','` is tried. At the end of the sample lines, if more were successfully parsed with `dec=','`, `dec` is set to `','`; otherwise, `dec` is set to `'.'`.

Automatic detection of `sep` occurs *prior* to column type detection – as such, it is possible that `sep` has been inferred to be `','`, in which case `dec` is set to `'.'`.

Quotes:

When `quote` is a single character,

- Spaces and other whitespace (other than `sep` and `\n`) may appear in unquoted character fields, e.g., `...,2,Joe Bloggs,3.14,...`.
- When `character` columns are *quoted*, they must start and end with that quoting character immediately followed by `sep` or `\n`, e.g., `...,2,"Joe Bloggs",3.14,...`.

  In essence quoting character fields are *required* only if `sep` or `\n` appears in the string value. Quoting may be used to signify that numeric data should be read as text. Unescaped quotes may be present in a quoted field, e.g., `...,2,"Joe, "Bloggs"",3.14,...`, as well as escaped quotes, e.g., `...,2,"Joe \",Bloggs\"",3.14,...`.

  If an embedded quote is followed by the separator inside a quoted field, the embedded quotes up to that point in that field must be balanced; e.g. `...,2,"www.blah?x="one",y="two"",3.14,...`.

  On those fields that do not satisfy these conditions, e.g., fields with unbalanced quotes, `fread` re-attempts that field as if it isn't quoted. This is quite useful in reading files that contains fields with unbalanced quotes as well, automatically.

To read fields *as is* instead, use `quote = ""`.

CSVY Support:

Currently, the `yaml` setting is somewhat inflexible with respect to incorporating metadata to facilitate file reading. Information on column classes should be stored at the top level under the heading `schema` and subheading `fields`; those with both a `type` and a `name` sub-heading will be merged into `colClasses`. Other supported elements are as follows:

- `sep` (or alias `delimiter`)
- `header`
- `quote` (or aliases `quoteChar`, `quote_char`)
- `dec` (or alias `decimal`)
- `na.strings`

File Download:

When input begins with http://, https://, ftp://, ftps://, or file://, fread detects this and *downloads* the target to a temporary file (at tempfile()) before proceeding to read the file as usual. URLS (ftps:// and https:// as well as ftp:// and http://) paths are downloaded with download.file and method set to getOption("download.file.method"), defaulting to "auto"; and file:// is downloaded with download.file with method="internal". NB: this implies that for file://, even files found on the current machine will be "downloaded" (i.e., hard-copied) to a temporary file. See download.file for more details.

Automatic Decompression:

In many cases, fread can automatically detect and decompress files with common compression extensions directly, without needing an explicit connection object or shell commands. This works by checking the file extension.

- .gz and .bz2 are supported out of the box.
- .zip is also supported. If the archive contains a single data file, fread will read it. If the archive contains multiple files, fread will produce an error.

Shell commands:

fread accepts shell commands for convenience. The input command is run and its output written to a file in tmpdir (tempdir() by default) to which fread is applied "as normal". The details are platform dependent – system is used on UNIX environments, shell otherwise; see system.

Value

A data.table by default, otherwise a data.frame when argument data.table=FALSE.

References

Background :
https://cran.r-project.org/doc/manuals/R-data.html
https://stackoverflow.com/questions/1727772/quickly-reading-very-large-tables-as-dataframes-in-r
https://stackoverflow.com/questions/9061736/faster-than-scan-with-rcpp
https://stackoverflow.com/questions/415515/how-can-i-read-and-manipulate-csv-file-data-in-c
https://stackoverflow.com/questions/9352887/strategies-for-reading-in-csv-files-in-pieces
https://stackoverflow.com/questions/11782084/reading-in-large-text-files-in-r
https://stackoverflow.com/questions/45972/mmap-vs-reading-blocks
https://stackoverflow.com/questions/258091/when-should-i-use-mmap-for-file-access
https://stackoverflow.com/a/9818473/403310
https://stackoverflow.com/questions/9608950/reading-huge-files-using-memory-mapped-files

finagle = "to obtain (something) by indirect or involved means", https://www.merriam-webster.com/dictionary/finagler

On YAML, see https://yaml.org/.

See Also

read.csv, url, Sys.setlocale, setDTthreads, fwrite, bit64::integer64

Examples

```
# Reads text input directly :
fread("A,B\n1,2\n3,4")

# Reads pasted input directly :
fread("A,B
1,2
3,4
")

# Finds the first data line automatically :
fread("
This is perhaps a banner line or two or ten.
A,B
1,2
3,4
")

# Detects whether column names are present automatically :
fread("
1,2
3,4
")

# Numerical precision :

DT = fread("A\n1.0102030405060708090102030405060\n")
# TODO: add numerals=c("allow.loss", "warn.loss", "no.loss") from base::read.table, +"use.Rmpfr"
typeof(DT$A)=="double"   # currently "allow.loss" with no option

DT = fread("A\n1.46761e-313\n")   # read as 'numeric'
DT[,sprintf("%.15E",A)]  # beyond what double precision can store accurately to 15 digits
# For greater accuracy use colClasses to read as character, then package Rmpfr.

# colClasses
data = "A,B,C,D\n1,3,5,7\n2,4,6,8\n"
fread(data, colClasses=c(B="character",C="character",D="character"))  # as read.csv
fread(data, colClasses=list(character=c("B","C","D")))    # saves typing
fread(data, colClasses=list(character=2:4))     # same using column numbers

# drop
fread(data, colClasses=c("B"="NULL","C"="NULL"))   # as read.csv
fread(data, colClasses=list(NULL=c("B","C")))      #
fread(data, drop=c("B","C"))      # same but less typing, easier to read
fread(data, drop=2:3)             # same using column numbers

# select
# (in read.csv you need to work out which to drop)
fread(data, select=c("A","D"))    # less typing, easier to read
fread(data, select=c(1,4))        # same using column numbers

# select and types combined
```

```
fread(data, select=c(A="numeric", D="character"))
fread(data, select=list(numeric="A", character="D"))

# skip blank lines
fread("a,b\n1,a\n2,b\n\n\n3,c\n", blank.lines.skip=TRUE)
# fill
fread("a,b\n1,a\n2\n3,c\n", fill=TRUE)
fread("a,b\n\n1,a\n2\n\n3,c\n\n", fill=TRUE)

# fill with skip blank lines
fread("a,b\n\n1,a\n2\n\n3,c\n\n", fill=TRUE, blank.lines.skip=TRUE)

# check.names usage
fread("a b,a b\n1,2\n")
fread("a b,a b\n1,2\n", check.names=TRUE) # no duplicates + syntactically valid names

## Not run:
# Demo speed-up
n = 1e6
DT = data.table( a=sample(1:1000,n,replace=TRUE),
                 b=sample(1:1000,n,replace=TRUE),
                 c=rnorm(n),
                 d=sample(c("foo","bar","baz","qux","quux"),n,replace=TRUE),
                 e=rnorm(n),
                 f=sample(1:1000,n,replace=TRUE) )
DT[2,b:=NA_integer_]
DT[4,c:=NA_real_]
DT[3,d:=NA_character_]
DT[5,d:=""]
DT[2,e:=+Inf]
DT[3,e:=-Inf]

write.table(DT,"test.csv",sep=",",row.names=FALSE,quote=FALSE)
cat("File size (MiB):", round(file.info("test.csv")$size/1024^2),"\n")
# 50 MiB (1e6 rows x 6 columns)

system.time(DF1 <-read.csv("test.csv",stringsAsFactors=FALSE))
# 5.4 sec (first time in fresh R session)

system.time(DF1 <- read.csv("test.csv",stringsAsFactors=FALSE))
# 3.9 sec (immediate repeat is faster, varies)

system.time(DF2 <- read.table("test.csv",header=TRUE,sep=",",quote="",
    stringsAsFactors=FALSE,comment.char="",nrows=n,
    colClasses=c("integer","integer","numeric",
                 "character","numeric","integer")))
# 1.2 sec (consistently). All known tricks and known nrows, see references.

system.time(DT <- fread("test.csv"))
# 0.1 sec (faster and friendlier)

identical(DF1, DF2)
all.equal(as.data.table(DF1), DT)
```

```
# Scaling up ...
l = vector("list",10)
for (i in 1:10) l[[i]] = DT
DTbig = rbindlist(l)
tables()
write.table(DTbig, "testbig.csv", sep=",", row.names=FALSE, quote=FALSE)
# ~500MiB csv (10 million rows x 6 columns)

system.time({
  DF <- read.table("testbig.csv", header=TRUE, sep=",",
    quote="", stringsAsFactors=FALSE, comment.char="", nrows=1e7,
    colClasses=c("integer", "integer", "numeric",
                 "character", "numeric", "integer"))
})
# 17.0 sec (varies)

system.time(DT <- fread("testbig.csv"))
#  0.8 sec

all(mapply(all.equal, DF, DT))

# Reads URLs directly :
fread("https://www.stats.ox.ac.uk/pub/datasets/csb/ch11b.dat")

# Decompresses .gz and .bz2 automatically :
fread("https://github.com/Rdatatable/data.table/raw/1.14.0/inst/tests/ch11b.dat.bz2")

fread("https://github.com/Rdatatable/data.table/raw/1.14.0/inst/tests/issue_785_fread.txt.gz")


## End(Not run)
```

`frev` *Fast reverse*

Description

Similar to `rev` but *faster*.

Usage

```
frev(x)
```

Arguments

`x`	An atomic `vector` or `list`.

Details

Similar to `rev`, `frev` only retains three attributes: names, class, and factor levels.

Value

Returns the input reversed.

Examples

```
# on vectors
x = setNames(1:10, letters[1:10])
frev(x)

# list
frev(list(1, "a", TRUE))
```

---

froll *Rolling functions*

Description

Fast rolling functions to calculate aggregates on a sliding window. For a user-defined rolling function see frollapply. For "time-aware" (irregularly spaced time series) rolling function see frolladapt.

Usage

```
frollmean(x, n, fill=NA, algo=c("fast","exact"), align=c("right","left","center"),
  na.rm=FALSE, has.nf=NA, adaptive=FALSE, partial=FALSE, give.names=FALSE, hasNA)
frollsum(x, n, fill=NA, algo=c("fast","exact"), align=c("right","left","center"),
  na.rm=FALSE, has.nf=NA, adaptive=FALSE, partial=FALSE, give.names=FALSE, hasNA)
frollmax(x, n, fill=NA, algo=c("fast","exact"), align=c("right","left","center"),
  na.rm=FALSE, has.nf=NA, adaptive=FALSE, partial=FALSE, give.names=FALSE, hasNA)
frollmin(x, n, fill=NA, algo=c("fast","exact"), align=c("right","left","center"),
  na.rm=FALSE, has.nf=NA, adaptive=FALSE, partial=FALSE, give.names=FALSE, hasNA)
frollprod(x, n, fill=NA, algo=c("fast","exact"), align=c("right","left","center"),
  na.rm=FALSE, has.nf=NA, adaptive=FALSE, partial=FALSE, give.names=FALSE, hasNA)
frollmedian(x, n, fill=NA, algo=c("fast","exact"), align=c("right","left","center"),
  na.rm=FALSE, has.nf=NA, adaptive=FALSE, partial=FALSE, give.names=FALSE, hasNA)
frollvar(x, n, fill=NA, algo=c("fast","exact"), align=c("right","left","center"),
  na.rm=FALSE, has.nf=NA, adaptive=FALSE, partial=FALSE, give.names=FALSE, hasNA)
frollsd(x, n, fill=NA, algo=c("fast","exact"), align=c("right","left","center"),
  na.rm=FALSE, has.nf=NA, adaptive=FALSE, partial=FALSE, give.names=FALSE, hasNA)
```

Arguments

x	Integer, numeric or logical vector, coerced to numeric, on which sliding window calculates an aggregate function. It supports vectorized input, then it needs to be a data.table, data.frame or a list, in which case a rolling function is applied to each column/vector.

n	Integer, non-negative, non-NA, rolling window size. This is the *total* number of included values in aggregate function. In case of an adaptive rolling function, the window size has to be provided as a vector for each individual value of x. It supports vectorized input, then it needs to be a vector, or in case of an adaptive rolling a list of vectors.
fill	Numeric; value to pad by for an incomplete window iteration. Defaults to NA. When partial=TRUE this argument is ignored.
algo	Character, default "fast". When set to "exact", a slower (in some cases more accurate) algorithm is used. It will use multiple cores where available. See Details for more information.
align	Character, specifying the "alignment" of the rolling window, defaulting to "right". "right" covers preceding rows (the window *ends* on the current value); "left" covers following rows (the window *starts* on the current value); "center" is halfway in between (the window is *centered* on the current value, biased towards "left" when n is even).
na.rm	Logical, default FALSE. Should missing values be removed when calculating aggregate function on a window?
has.nf	Logical. If it is known whether x contains non-finite values (NA, NaN, Inf, -Inf), then setting this to TRUE or FALSE may speed up computation. Defaults to NA. See *has.nf argument* section below for details.
adaptive	Logical, default FALSE. Should the rolling function be calculated adaptively? See *Adaptive rolling functions* section below for details.
partial	Logical, default FALSE. Should the rolling window size(s) provided in n be computed also for leading incomplete running window? See partial *argument* section below for details.
give.names	Logical, default FALSE. When TRUE, names are automatically generated corresponding to names of x and names of n. If answer is an atomic vector, then the argument is ignored, see examples.
hasNA	Logical. Deprecated, use has.nf argument instead.

Details

froll* functions accept vector, list, data.frame or data.table. Functions operate on a single vector; when passing a non-atomic input, then the function is applied column-by-column, not to the complete set of columns at once.

Argument n allows multiple values to apply rolling function on multiple window sizes. If adaptive=TRUE, then n can be a list to specify multiple window sizes for adaptive rolling computation. See *Adaptive rolling functions* section below for details.

When multiple columns or multiple window widths are provided, then they are run in parallel. The exception is for algo="exact" or adaptive=TRUE, which runs in parallel even for single column and single window width. By default, data.table uses only half of available CPUs, see setDTthreads for details on how to tune CPU usage.

Setting options(datatable.verbose=TRUE) will display various information about how rolling function processed. It will not print information in real-time but only at the end of the processing.

Value

For a non *vectorized* input (`x` is not a list, and `n` specifies a single rolling window) a `vector` is returned, for convenience. Thus, rolling functions can be used conveniently within `data.table` syntax. For a *vectorized* input a list is returned.

`has.nf` argument

`has.nf` can be used to speed up processing in cases when it is known if `x` contains (or not) non-finite values (`NA`, `NaN`, `Inf`, `-Inf`).

- Default `has.nf=NA` uses faster implementation that does not support non-finite values, but when non-finite values are detected it will re-run non-finite aware implementation.
- `has.nf=TRUE` uses non-finite aware implementation straightaway.
- `has.nf=FALSE` uses faster implementation that does not support non-finite values. Then depending on the rolling function it will either:
  - (*mean, sum, prod, var, sd*) detect non-finite, re-run non-finite aware.
  - (*max, min, median*) does not detect non-finites and may silently produce an incorrect answer.

In general `has.nf=FALSE && any(!is.finite(x))` should be considered undefined behavior. Therefore `has.nf=FALSE` should be used with care.

Implementation

Most of the rolling functions have 4 different implementations. First factor that decides which implementation is used is the `adaptive` argument (either `TRUE` or `FALSE`), see section below for details. Then for each of those two algorithms there are usually two implementations depending on the `algo` argument.

- `algo="fast"` uses *"online"*, single pass, algorithm.
  - *max* and *min* rolling function will not do only a single pass but, on average, they will compute `length(x)/n` nested loops. The larger the window, the greater the advantage over the *exact* algorithm, which computes `length(x)` nested loops. Note that *exact* uses multiple CPUs so for a small window sizes and many CPUs it may actually be faster than *fast*. However, in such cases the elapsed timings will likely be far below a single second.
  - *median* will use a novel algorithm described by *Jukka Suomela* in his paper *Median Filtering is Equivalent to Sorting (2014)*. See references section for the link. Implementation here is extended to support arbitrary length of input and an even window size. Despite extensive validation of results this function should be considered experimental. When missing values are detected it will fall back to slower `algo="exact"` implementation.
  - *var* and *sd* will use numerically stable *Welford*'s online algorithm.
  - Not all functions have *fast* implementation available. As of now, adaptive *max*, *min*, *median*, *var* and *sd* do not have *fast* adaptive implementation, therefore it will automatically fall back to *exact* adaptive implementation. Similarly, non-adaptive fast implementations of *median*, *var* and *sd* will fall back to *exact* implementations if they detect any non-finite values in the input. `datatable.verbose` option can be used to check that.

- `algo="exact"` will make the rolling functions use a more computationally-intensive algorithm. For each observation in the input vector it will compute a function on a rolling window from scratch (complexity $O(n^2)$).
  - Depending on the function, this algorithm may suffer less from floating point rounding error (the same consideration applies to base `mean`).
  - In case of *mean*, it will additionally make an extra pass to perform floating point error correction. Error corrections might not be truly exact on some platforms (like Windows) when using multiple threads.

Adaptive rolling functions

Adaptive rolling functions are a special case where each observation has its own corresponding rolling window width. Therefore, values passed to `n` argument must be series corresponding to observations in `x`. If multiple windows are meant to be computed, then a list of integer vectors is expected; each list element must be an integer vector of window size corresponding to observations in `x`; see Examples. Due to the logic or implementation of adaptive rolling functions, the following restrictions apply:

- `align` does not support `"center"`.
- if a list of vectors is passed to `x`, then all vectors within it must have equal length due to the fact that length of adaptive window widths must match the length of vectors in `x`.

`partial` argument

`partial=TRUE` is used to calculate rolling moments *only* within the input itself. That is, at the boundaries (say, observation 2 for `n=4` and `align="right"`), we don't consider observations before the first as "missing", but instead shrink the window to be size `n=2`. In practice, this is the same as an *adaptive* window, and could be accomplished, albeit less concisely, with a well-chosen `n` and `adaptive=TRUE`. In fact, we implement `partial=TRUE` using the same algorithms as `adaptive=TRUE`. Therefore `partial=TRUE` inherits the limitations of adaptive rolling functions, see above. Adaptive functions use more complex algorithms; if performance is important, `partial=TRUE` should be avoided in favour of computing only missing observations separately after the rolling function; see examples.

`zoo` package users notice

Users coming from most popular package for rolling functions `zoo` might expect following differences in `data.table` implementation

- rolling function will always return result of the same length as input.
- `fill` defaults to `NA`.
- `fill` accepts only constant values. It does not support for *na.locf* or other functions.
- `align` defaults to `"right"`.
- `na.rm` is respected, and other functions are not needed when input contains `NA`.
- integers and logical are always coerced to numeric.
- when `adaptive=FALSE` (default), then `n` must be a numeric vector. List is not accepted.
- when `adaptive=TRUE`, then `n` must be vector of length equal to `nrow(x)`, or list of such vectors.

Note

Be aware that rolling functions operate on the physical order of input. If the intent is to roll values in a vector by a logical window, for example an hour, or a day, then one has to ensure that there are no gaps in the input, or use an adaptive rolling function to handle gaps, for which we provide helper function `frolladapt` to generate adaptive window size.

References

Round-off error, "Median Filtering is Equivalent to Sorting" by Jukka Suomela

See Also

`frollapply`, `frolladapt`, `shift`, `data.table`, `setDTthreads`

Examples

```
# single vector and single window
frollmean(1:6, 3)

d = as.data.table(list(1:6/2, 3:8/4))
# rollmean of single vector and single window
frollmean(d[, V1], 3)
# multiple columns at once
frollmean(d, 3)
# multiple windows at once
frollmean(d[, .(V1)], c(3, 4))
# multiple columns and multiple windows at once
frollmean(d, c(3, 4))
## three calls above will use multiple cores when available

# other functions
frollsum(d, 3:4)
frollmax(d, 3:4)
frollmin(d, 3:4)
frollprod(d, 3:4)
frollmedian(d, 3:4)
frollvar(d, 3:4)
frollsd(d, 3:4)

# partial=TRUE
x = 1:6/2
n = 3
ans1 = frollmean(x, n, partial=TRUE)
# same using adaptive=TRUE
an = function(n, len) c(seq.int(n), rep.int(n, len-n))
ans2 = frollmean(x, an(n, length(x)), adaptive=TRUE)
all.equal(ans1, ans2)
# speed up by using partial only for incomplete observations
ans3 = frollmean(x, n)
ans3[seq.int(n-1L)] = frollmean(x[seq.int(n-1L)], n, partial=TRUE)
all.equal(ans1, ans3)
```

```
# give.names
frollsum(list(x=1:5, y=5:1), c(tiny=2, big=4), give.names=TRUE)

# has.nf=FALSE should be used with care
frollmax(c(1,2,NA,4,5), 2)
frollmax(c(1,2,NA,4,5), 2, has.nf=FALSE)

# use verbose=TRUE for extra insight
.op = options(datatable.verbose = TRUE)
frollsd(c(1:5,NA,7:8), 4)
options(.op)

# performance vs exactness
set.seed(108)
x = sample(c(rnorm(1e3, 1e6, 5e5), 5e9, 5e-9))
n = 15
ma = function(x, n, na.rm=FALSE) {
  ans = rep(NA_real_, nx<-length(x))
  for (i in n:nx) ans[i] = mean(x[(i-n+1):i], na.rm=na.rm)
  ans
}
fastma = function(x, n, na.rm) {
  if (!missing(na.rm)) stop("NAs are unsupported, wrongly propagated by cumsum")
  cs = cumsum(x)
  scs = shift(cs, n)
  scs[n] = 0
  as.double((cs-scs)/n)
}
system.time(ans1<-ma(x, n))
system.time(ans2<-fastma(x, n))
system.time(ans3<-frollmean(x, n))
system.time(ans4<-frollmean(x, n, algo="exact"))
system.time(ans5<-frollapply(x, n, mean))
anserr = list(
  fastma = ans2-ans1,
  froll_fast = ans3-ans1,
  froll_exact = ans4-ans1,
  frollapply = ans5-ans1
)
errs = sapply(lapply(anserr, abs), sum, na.rm=TRUE)
sapply(errs, format, scientific=FALSE) # roundoff
```

`frolladapt` *Adapt rolling window to irregularly spaced time series*

Description

Helper function to generate adaptive window size based on the irregularly spaced time series index, to be passed as `n` argument to adaptive `froll` function (or `N` argument to adaptive `frollapply`). Experimental.

Usage

```
frolladapt(x, n, align="right", partial=FALSE, give.names=FALSE)
```

Arguments

x	Integer. Must be sorted with no duplicates or missing values. Other objects with numeric storage (including most commonly Date and POSIXct) will be coerced to integer, which, for example, in case of POSIXct means truncating to whole seconds. It does not support vectorized input.
n	Integer, positive, rolling window size. Up to n values nearest to each value of x, with distance in the units of x and according to the window implied by align, are included in each rolling aggregation window. Thus when x is a POSIXct, n are seconds, and when x is a Date, n are days. It supports vectorized input, then it needs to be a vector.
align	Character, default "right". Other alignments have not yet been implemented.
partial	Logical, default FALSE. Should the rolling window size(s) provided in n be trimmed to available observations? For details see froll.
give.names	Logical, default FALSE. When TRUE, names are automatically generated corresponding to names of n. If answer is an integer vector, then the argument is ignored, see examples.

Details

Argument n allows multiple values to generate multiple adaptive windows, unlike x, as mixing different time series would make no sense.

Value

When length(n)==1L then integer vector (*adaptive window* size) of length of x. Otherwise a list of length(n) having integer vectors (*adaptive window* sizes) of length of x for each window size provided in n.

See Also

froll, frollapply

Examples

```
idx = as.Date("2022-10-23") + c(0,1,4,5,6,7,9,10,14)
dt = data.table(index=idx, value=seq_along(idx))
dt
dt[, n3 := frolladapt(index, n=3L)]
dt
dt[, rollmean3 := frollmean(value, n3, adaptive=TRUE)]
dt
dt[, n3p := frolladapt(index, n=3L, partial=TRUE)]
dt[, rollmean3p := frollmean(value, n3p, adaptive=TRUE)]
dt
```

```
n34 = frolladapt(idx, c(small=3, big=4), give.names=TRUE)
n34
dt[, frollmean(value, n34, adaptive=TRUE, give.names=TRUE)]
```

`frollapply` *Rolling user-defined function*

Description

Fast rolling user-defined function (*UDF*) to calculate on a sliding window. Experimental. Please read, at least, *caveats* section below. For "time-aware" (irregularly spaced time series) rolling function see `frolladapt`.

Usage

```
frollapply(X, N, FUN, ..., by.column=TRUE, fill=NA,
  align=c("right","left","center"), adaptive=FALSE, partial=FALSE,
  give.names=FALSE, simplify=TRUE, x, n)
```

Arguments

`X`	Atomic vector, `data.frame`, `data.table` or a `list` on which sliding window calculates `FUN` function. How the `X` is handled depends on the `by.column` argument. It supports vectorized input, for `by.column=TRUE` it needs to be a `data.table`, `data.frame` or a `list`, and for `by.column=FALSE` list of data.frames/data.tables, but not a list of lists.
`N`	Integer, non-negative, non-NA, rolling window size. This is the *total* number of included values in aggregate function. In case of an adaptive rolling function window size has to be provided as a vector for each individual value of `X`. It supports vectorized input, then it needs to be a vector, or in case of an adaptive rolling a `list` of vectors.
`FUN`	The function to be applied on subsets of `X`.
`...`	Extra arguments passed to `FUN`. Note that argument names passed to `...` should not overlap with arguments of `frollapply`.
`by.column`	Logical. When `TRUE` (default) then `X` of types list/data.frame/data.table is treated as vectorized input rather an object to apply rolling window on. Setting to `FALSE` allows rolling window to be applied on multiple variables, using data.frame, data.table or a list, as a whole. For details see `by.column` *argument* section below.
`fill`	An object; value to pad by for an incomplete window iteration. Defaults to `NA`. When `partial=TRUE` this argument is ignored.
`align`	Character, specifying the "alignment" of the rolling window, defaulting to `"right"`. For details see `froll`.
`adaptive`	Logical, default `FALSE`. Should the rolling function be calculated adaptively? For details see `froll`.

partial	Logical, default FALSE. Should the rolling window size(s) provided in N be trimmed to available observations? For details see froll.
give.names	Logical, default FALSE. When TRUE, names are automatically generated corresponding to names of X and names of N. If answer is an atomic vector, then the argument is ignored, see examples.
simplify	Logical or a function. When TRUE (default) then internal simplifylist function is applied on a list storing results of all computations. When FALSE then list is returned without any post-processing. Argument can take a function as well, then the function is applied to a list that would have been returned when simplify=FALSE. If results are not automatically simplified when simplify=TRUE then, for backward compatibility, one should use simplify=FALSE explicitly. See simplify *argument* section below for details.
x	Deprecated, use X instead.
n	Deprecated, use N instead.

Value

Argument simplify impacts the type returned. Its default value TRUE is set for convenience and backward compatibility, but it is advised to use simplify=unlist (or other desired function) instead.

- simplify=FALSE will always return list where each element will be a result of each iteration.
- simplify=unlist (or any other function) will return object returned by provided function as supplied with results of frollapply using simplify=FALSE.
- simplify=TRUE will try to simplify results by unlist, rbind or other functions, its behavior is subject to change, see simplify *argument* section below for more details.

by.column argument

Setting by.column to FALSE allows to apply function on multiple variables rather than a single vector. Then X expects to be data.table, data.frame or a list of equal length vectors, and window size provided in N refers to number of rows (or length of a vectors in a list). See examples for use cases. Error *"incorrect number of dimensions"* can be commonly observed when by.column was not set to FALSE when FUN expects its input to be a data.table or data.frame.

simplify argument

When set to TRUE (default), then results from rolling function which are normally stored in a list may be simplified either with unlist or rbindlist. It also attempts to match type, size and names of fill argument to the results of a function. One should avoid simplify=TRUE when writing robust code. One reason is performance, as explained in *Performance consideration* section below. Another is backward compatibility. For backward compatibility and performance one should always provide desired function to simplify explicitly. In future version we may change internal simplifylist function, then simplify=TRUE may return object of a different type, breaking downstream code.

Caveats

With great power comes great responsibility.

1. An optimization used to avoid repeated allocation of window subsets (explained more deeply in *Implementation* section below) may, in special cases, return rather surprising results:

```
setDTthreads(1)
frollapply(c(1, 9), N=1L, FUN=identity) ## unexpected
#[1] 9 9
frollapply(c(1, 9), N=1L, FUN=list) ## unexpected
#      V1
#   <num>
#1:     9
#2:     9
setDTthreads(2, throttle=1) ## disable throttle
frollapply(c(1, 9), N=1L, FUN=identity) ## good only because threads >= input
#[1] 1 9                                 ## on Linux and Macos
frollapply(c(1, 5, 9), N=1L, FUN=identity) ## unexpected again
#[1] 5 5 9
```

   Problem occurs, in rather unlikely scenarios for rolling computations, when objects returned from a function can be its input (i.e. `identity`), or a reference to it (i.e. `list`), then one has to add extra `copy` call:

```
setDTthreads(1)
frollapply(c(1, 9), N=1L, FUN=function(x) copy(identity(x))) ## only 'copy' would be equivalent her
#[1] 1 9
frollapply(c(1, 9), N=1L, FUN=function(x) copy(list(x)))
#      V1
#   <num>
#1:     1
#2:     9
```

2. `FUN` calls are internally passed to `parallel::mcparallel` to evaluate them in parallel. We inherit few limitations from `parallel` package explained below. This optimization can be disabled completely by calling `setDTthreads(1)`, in which case the limitations listed below do not apply because all evaluations of `FUN` will be made sequentially without use of `parallel` package. Note that on Windows platform this optimization is always disabled due to lack of *fork* used by `parallel` package. One can use `options(datatable.verbose=TRUE)` to get extra information if `frollapply` is running multithreaded or not.

   - Warnings produced inside the function are silently ignored; for consistency we ignore warnings also when running single threaded path.
   - `FUN` should not use any on-screen devices, GUI elements, tcltk, multithreaded libraries.
   - `setDTthreads(1L)` is passed to forked processes, therefore any data.table code inside `FUN` will be forced to be single threaded. It is advised not to call `setDTthreads` inside `FUN`. `frollapply` is already parallelized, and nested parallelism is rarely a good idea.
   - Any operation that could misbehave when run in parallel has to be handled. For example, writing to the same file from multiple CPU threads.

```
old = setDTthreads(1L)
frollapply(iris, 5L, by.column=FALSE, FUN=fwrite, file="rolling-data.csv", append=TRUE)
```

```
setDTthreads(old)
```

- Objects returned from forked processes, FUN, are serialized. This may cause problems for objects that are meant not to be serialized, like data.table. We are handling that for data.table class internally in frollapply whenever FUN is returning data.table (which is checked on the results of the first FUN call so it assumes function is type stable). If data.table is nested in another object returned from FUN then the problem may still manifest, in such case one has to call setDT on objects returned from FUN. This can be also nicely handled via simplify argument when passing a function that calls setDT on nested data.table objects returned from FUN. Anyway, returning data.table from FUN should, in majority of cases, be avoided from the performance reasons, see *UDF optimization* section for details.

```
setDTthreads(2, throttle=1) ## disable throttle
## frollapply will fix DT in most cases
ans = frollapply(1:2, 2, data.table)
.selfref.ok(ans)
#[1] TRUE
ans = frollapply(1:2, 2, data.table, simplify=FALSE)
.selfref.ok(ans[[2L]])
#[1] TRUE

## nested DT not fixed
ans = frollapply(1:2, 2, function(x) list(data.table(x)), fill=list(data.table(NA)), simplify=
.selfref.ok(ans[[2L]][[1L]])
#[1] FALSE
#### now if we want to use it
set(ans[[2L]][[1L]],, "newcol", 1L)
#Error in set(ans[[2L]][[1L]], , "newcol", 1L) :
#  This data.table has either been loaded from disk (e.g. using readRDS()/load()) or constructed
#### fix as explained in error message
ans = lapply(ans, lapply, setDT)
.selfref.ok(ans[[2L]][[1L]])
#[1] TRUE

## fix inside frollapply via simplify
simplifix = function(x) lapply(x, lapply, setDT)
ans = frollapply(1:2, 2, function(x) list(data.table(x)), fill=list(data.table(NA)), simplify=
.selfref.ok(ans[[2L]][[1L]])
#[1] TRUE

## automatic fix may not work for a non-type stable function
f = function(x) (if (x[1L]==1L) data.frame else data.table)(x)
ans = frollapply(1:3, 2, f, fill=data.table(NA), simplify=FALSE)
.selfref.ok(ans[[3L]])
#[1] FALSE
#### fix inside frollapply via simplify
simplifix = function(x) lapply(x, function(y) if (is.data.table(y)) setDT(y) else y)
ans = frollapply(1:3, 2, f, fill=data.table(NA), simplify=simplifix)
.selfref.ok(ans[[3L]])
```

```
#[1] TRUE

setDTthreads(2, throttle=1024) ## enable throttle
```

3. Due to possible future improvements of handling simplification of results returned from rolling function, the default `simplify=TRUE` may not be backward compatible for functions that produce results that haven't been already automatically simplified. See the `simplify` *argument* section for details.

Performance consideration

`frollapply` is meant to run any UDF function. If one needs to use a common function like *mean, sum, max*, etc., then we have highly optimized, implemented in C language, rolling functions described in `froll` manual.
Most crucial optimizations are the ones to be applied on UDF. Those are discussed below in section *UDF optimization*.

- When using `by.column=FALSE`, subset the dataset before passing it to `X` to keep only columns relevant for the computation:

```
x = setDT(lapply(1:1000, function(x) as.double(rep.int(x,1e4L))))
f = function(x) sum(x$V1 * x$V2)
system.time(frollapply(x, 100, f, by.column=FALSE))
#   user  system elapsed
#  0.373   0.069   0.234
system.time(frollapply(x[, c("V1","V2"), with=FALSE], 100, f, by.column=FALSE))
#   user  system elapsed
#  0.050   0.058   0.061
```

- Avoid `partial` argument, see `partial` *argument* section of `froll` manual.
- Avoid `simplify=TRUE` and provide a function instead:

```
x = rnorm(1e5)
system.time(frollapply(x, 2, function(x) 1L, simplify=TRUE))
#   user  system elapsed
#  0.227   0.095   0.236
system.time(frollapply(x, 2, function(x) 1L, simplify=unlist))
#   user  system elapsed
#  0.054   0.049   0.091
```

- CPU threads utilization in `frollapply` can be controlled by `setDTthreads`, which by default uses half of available CPU threads. Usage of multiple CPU threads will be throttled for small input, as described in `setDTthreads` manual.
- Parallel computation of `FUN` is handled by `parallel` package (part of R core since 2.14.0) and its *fork* mechanism. *Fork* is not available on Windows OS, therefore computations will always be single-threaded on that platform.

UDF optimization

FUN will be evaluated many times so should be highly optimized. Tips below are not specific to `frollapply` and can be applied to any code meant to run in many iterations.

- It is usually better to return the most lightweight objects from FUN, for example it will be faster to return a list rather a data.table. In the case presented below, simplify=TRUE is calling rbindlist on the results anyway, which makes the results equal:

```
fun1 = function(x) {tmp=range(x); data.table(min=tmp[1L], max=tmp[2L])}
fun2 = function(x) {tmp=range(x); list(min=tmp[1L], max=tmp[2L])}
fill1 = data.table(min=NA_integer_, max=NA_integer_)
fill2 = list(min=NA_integer_, max=NA_integer_)
system.time(a<-frollapply(1:1e4, 100, fun1, fill=fill1, simplify=rbindlist))
#   user  system elapsed
#  0.934   0.347   0.706
system.time(b<-frollapply(1:1e4, 100, fun2, fill=fill2, simplify=rbindlist))
#   user  system elapsed
#  0.010   0.033   0.094
all.equal(a, b)
#[1] TRUE
```

- Code that is not dependent on a rolling window should be taken out as pre or post computation:

```
x = c(1L,3L)
system.time(for (i in 1:1e6) sum(x+1L))
#   user  system elapsed
#  0.218   0.002   0.221
system.time({y = x+1L; for (i in 1:1e6) sum(y)})
#   user  system elapsed
#  0.160   0.001   0.161
```

- Being strict about data types removes the need for R to handle them automatically:

```
x = vector("integer", 1e6)
system.time(for (i in 1:1e6) x[i] = NA)
#   user  system elapsed
#  0.114   0.000   0.114
system.time(for (i in 1:1e6) x[i] = NA_integer_)
#   user  system elapsed
#  0.029   0.000   0.030
```

- If a function calls another function under the hood, it is usually better to call the latter one directly:

```
x = matrix(c(1L,2L,3L,4L), c(2L,2L))
system.time(for (i in 1:1e4) colSums(x))
#   user  system elapsed
#  0.033   0.000   0.033
system.time(for (i in 1:1e4) .colSums(x, 2L, 2L))
#   user  system elapsed
#  0.010   0.002   0.012
```

- There are many functions that may be optimized for scaling up with larger input, yet for a small input they may incur bigger overhead comparing to simpler counterparts. One may need to experiment on own data, but low overhead functions are likely to be faster when evaluated over many iterations:

```
## uniqueN
x = c(1L,3L,5L)
system.time(for (i in 1:1e4) uniqueN(x))
#   user  system elapsed
#  0.078   0.001   0.080
system.time(for (i in 1:1e4) length(unique(x)))
#   user  system elapsed
#  0.018   0.000   0.018
## column subset
x = data.table(v1 = c(1L,3L,5L))
system.time(for (i in 1:1e4) x[, v1])
#   user  system elapsed
#  1.952   0.011   1.964
system.time(for (i in 1:1e4) x[["v1"]])
#   user  system elapsed
#  0.036   0.000   0.035
```

Implementation

Evaluation of UDF comes with very limited capabilities for optimizations, therefore speed improvements in `frollapply` should not be expected as good as in other data.table fast functions. `frollapply` is implemented almost exclusively in R, rather than C. Its speed improvement comes from two optimizations that have been applied:

1. No repeated allocation of a rolling window subset.
   Object (type of `X` and size of `N`) is allocated once (for each CPU thread), and then for each iteration this object is being re-used by copying expected subset of data into it. This means we still have to subset data on each iteration, but we only copy data into pre-allocated window object, instead of allocating in each iteration. Allocation is carrying much bigger overhead than copy. The faster the `FUN` evaluates the more relative speedup we are getting, because allocation of a subset does not depend on how fast or slow `FUN` evaluates. See *caveats* section for possible edge cases caused by this optimization.
2. Parallel evaluation of `FUN` calls.
   Until September 2025 all the multithreaded code in data.table was using *OpenMP*. It can be used only in C language and it has very low overhead. Unfortunately it could not be applied in `frollapply` because to evaluate UDF from C code one has to call R's C api that is not thread safe (can be run only from single threaded C code). Therefore `frollapply` uses `parallel-package`, which is included in base R, to provide parallelism at the R language level. It uses *fork* parallelism, which has low overhead as well (unless results of computation are big in size which is not an issue for rolling statistics). *Fork* is not available on Windows OS. See *caveats* section for limitations caused by using this optimization.

Note

Be aware that rolling functions operate on the physical order of input. If the intent is to roll values in a vector by a logical window, for example an hour, or a day, then one has to ensure that there are no gaps in the input, or use an adaptive rolling function to handle gaps, for which we provide helper function `frolladapt` to generate adaptive window size.

See Also

froll, frolladapt, shift, data.table, setDTthreads

Examples

```
frollapply(1:16, 4, median)
frollapply(1:9, 3, toString)

## vectorized input
x = list(1:10, 10:1)
n = c(3, 4)
frollapply(x, n, sum)
## give names
x = list(data1 = 1:10, data2 = 10:1)
n = c(small = 3, big = 4)
frollapply(x, n, sum, give.names=TRUE)

## by.column=FALSE
x = as.data.table(iris)
flow = function(x) {
  v1 = x[[1L]]
  v2 = x[[2L]]
  (v1[2L] - v1[1L] * (1+v2[2L])) / v1[1L]
}
x[, "flow" := frollapply(.(Sepal.Length, Sepal.Width), 2L, flow, by.column=FALSE),
  by = Species][]

## rolling regression: by.column=FALSE
f = function(x) coef(lm(v2 ~ v1, data=x))
x = data.table(v1=rnorm(120), v2=rnorm(120))
frollapply(x, 4, f, by.column=FALSE)
```

---

fsort *Fast parallel sort*

Description

Similar to base::sort but fast using parallelism. Experimental.

Usage

```
fsort(x, decreasing = FALSE, na.last = FALSE, internal=FALSE, verbose=FALSE, ...)
```

Arguments

x	A vector. Type double, currently.
decreasing	Decreasing order?

na.last	Control treatment of NAs. If TRUE, missing values in the data are put last; if FALSE, they are put first; if NA, they are removed; if "keep" they are kept with rank NA.
internal	Internal use only. Temporary variable. Will be removed.
verbose	Print tracing information.
...	Not sure yet. Should be consistent with base R.

Details

Process will raise error if x contains negative values. Unless x is already sorted fsort will redirect processing to slower single threaded *order* followed by *subset* in following cases:

- data type other than *double* (*numeric*)
- data having NAs
- decreasing==FALSE

Value

The input in sorted order.

Examples

```
x = runif(1e6)
system.time(ans1 <- sort(x, method="quick"))
system.time(ans2 <- fsort(x))
identical(ans1, ans2)
```

---

fwrite *Fast CSV writer*

Description

As write.csv but much faster (e.g. 2 seconds versus 1 minute) and just as flexible. Modern machines almost surely have more than one CPU so fwrite uses them; on all operating systems including Linux, Mac and Windows.

Usage

```
fwrite(x, file = "", append = FALSE, quote = "auto",
  sep=getOption("datatable.fwrite.sep", ","),
  sep2 = c("","|",""),
  eol = if (.Platform$OS.type=="windows") "\r\n" else "\n",
  na = "", dec = ".", row.names = FALSE, col.names = TRUE,
  qmethod = c("double","escape"),
 logical01 = getOption("datatable.logical01", FALSE),  # due to change to TRUE; see NEWS
  scipen = getOption('scipen', 0L),
  dateTimeAs = c("ISO","squash","epoch","write.csv"),
```

```
  buffMB = 8L, nThread = getDTthreads(verbose),
  showProgress = getOption("datatable.showProgress", interactive()),
  compress = c("auto", "none", "gzip"),
  compressLevel = 6L,
  yaml = FALSE,
  bom = FALSE,
  verbose = getOption("datatable.verbose", FALSE),
  encoding = "",
  forceDecimal = FALSE)
```

Arguments

x	Any list of same length vectors; e.g. data.frame and data.table. If matrix, it gets internally coerced to data.table preserving col names but not row names
file	Output file name. "" indicates output to the console.
append	If TRUE, the file is opened in append mode and column names (header row) are not written.
quote	When "auto", character fields, factor fields and column names will only be surrounded by double quotes when they need to be; i.e., when the field contains the separator sep, a line ending \n, the double quote itself or (when list columns are present) sep2[2] (see sep2 below). If FALSE the fields are not wrapped with quotes even if this would break the CSV due to the contents of the field. If TRUE double quotes are always included other than around numeric fields, as write.csv.
sep	The separator between columns. Default is ",".
sep2	For columns of type list where each item is an atomic vector, sep2 controls how to separate items *within* the column. sep2[1] is written at the start of the output field, sep2[2] is placed between each item and sep2[3] is written at the end. sep2[1] and sep2[3] may be any length strings including empty "" (default). sep2[2] must be a single character and (when list columns are present and therefore sep2 is used) different from both sep and dec. The default (|) is chosen to visually distinguish from the default sep. In speaking, writing and in code comments we may refer to sep2[2] as simply "sep2".
eol	Line separator. Default is "\r\n" for Windows and "\n" otherwise.
na	The string to use for missing values in the data. Default is a blank string "".
dec	The decimal separator, by default ".". See link in references. Cannot be the same as sep.
row.names	Should row names be written? For compatibility with data.frame and write.csv since data.table never has row names. Hence default FALSE unlike write.csv.
col.names	Should the column names (header row) be written? The default is TRUE for new files and when overwriting existing files (append=FALSE). Otherwise, the default is FALSE to prevent column names appearing again mid-file when stacking a set of data.tables or appending rows to the end of a file.
qmethod	A character string specifying how to deal with embedded double quote characters when quoting strings.

- "escape" - the quote character (as well as the backslash character) is escaped in C style by a backslash, or
- "double" (default, same as `write.csv`), in which case the double quote is doubled with another one.

`logical01` Should `logical` values be written as `1` and `0` rather than `"TRUE"` and `"FALSE"`?

`scipen` `integer` In terms of printing width, how much of a bias should there be towards printing whole numbers rather than scientific notation? See Details.

`dateTimeAs` How `Date`/`IDate`, `ITime` and `POSIXct` items are written.

- "ISO" (default) - `2016-09-12`, `18:12:16` and `2016-09-12T18:12:16.999999Z`. 0, 3 or 6 digits of fractional seconds are printed if and when present for convenience, regardless of any R options such as `digits.secs`. The idea being that if milli and microseconds are present then you most likely want to retain them. R's internal UTC representation is written faithfully to encourage ISO standards, stymie timezone ambiguity and for speed. An option to consider is to start R in the UTC timezone simply with `"$ TZ='UTC' R"` at the shell (NB: it must be one or more spaces between `TZ='UTC'` and `R`, anything else will be silently ignored; this TZ setting applies just to that R process) or `Sys.setenv(TZ='UTC')` at the R prompt and then continue as if UTC were local time.
- "squash" - `20160912`, `181216` and `20160912181216999`. This option allows fast and simple extraction of `yyyy`, `mm`, `dd` and (most commonly to group by) `yyyymm` parts using integer div and mod operations. In R for example, one line helper functions could use `%/%10000`, `%/%100%%100`, `%%100` and `%/%100` respectively. POSIXct UTC is squashed to 17 digits (including 3 digits of milliseconds always, even if `000`) which may be read comfortably as `integer64` (automatically by `fread()`).
- "epoch" - `17056`, `65536` and `1473703936.999999`. The underlying number of days or seconds since the relevant epoch (1970-01-01, 00:00:00 and 1970-01-01T00:00:00Z respectively), negative before that (see `?Date`). 0, 3 or 6 digits of fractional seconds are printed if and when present.
- "write.csv" - this currently affects `POSIXct` only. It is written as `write.csv` does by using the `as.character` method which heeds `digits.secs` and converts from R's internal UTC representation back to local time (or the `"tzone"` attribute) as of that historical date. Accordingly this can be slow. All other column types (including `Date`, `IDate` and `ITime` which are independent of timezone) are written as the "ISO" option using fast C code which is already consistent with `write.csv`.

The first three options are fast due to new specialized C code. The epoch to date-part conversion uses a fast approach by Howard Hinnant (see references) using a day-of-year starting on 1 March. You should not be able to notice any difference in write speed between those three options. The date range supported for `Date` and `IDate` is [0000-03-01, 9999-12-31]. Every one of these 3,652,365 dates have been tested and compared to base R including all 2,790 leap days in this range.

This option applies to vectors of date/time in list column cells, too.

	A fully flexible format string (such as "%m/%d/%Y") is not supported. This is to encourage use of ISO standards and because that flexibility is not known how to make fast at C level. We may be able to support one or two more specific options if required.
buffMB	The buffer size (MiB) per thread in the range 1 to 1024, default 8MiB. Experiment to see what works best for your data on your hardware.
nThread	The number of threads to use. Experiment to see what works best for your data on your hardware.
showProgress	Display a progress meter on the console? Ignored when file="".
compress	If compress = "auto" and if file ends in .gz then output format is gzipped csv else csv. If compress = "none", output format is always csv. If compress = "gzip" then format is gzipped csv. Output to the console is never gzipped even if compress = "gzip". By default, compress = "auto".
compressLevel	Level of compression between 1 and 9, 6 by default. See https://www.gnu.org/software/gzip/manual/html_node/Invoking-gzip.html for details.
yaml	If TRUE, fwrite will output a CSVY file, that is, a CSV file with metadata stored as a YAML header, using as.yaml. See Details.
bom	If TRUE a BOM (Byte Order Mark) sequence (EF BB BF) is added at the beginning of the file; format 'UTF-8 with BOM'.
verbose	Be chatty and report timings?
encoding	The encoding of the strings written to the CSV file. Default is "", which means writing raw bytes without considering the encoding. Other possible options are "UTF-8" and "native".
forceDecimal	Should decimal points be forced for whole numbers in numeric columns? When FALSE, the default, whole numbers like c(1.0, 2.0, 3.0) will be written as '1, 2, 3' i.e., dropping dec.

Details

fwrite began as a community contribution with pull request #1613 by Otto Seiskari. This gave Matt Dowle the impetus to specialize the numeric formatting and to parallelize: https://web.archive.org/web/20250623031725/https://h2o.ai/blog/2016/fast-csv-writing-for-r/. Final items were tracked in issue #1664 such as automatic quoting, bit64::integer64 support, decimal/scientific formatting exactly matching write.csv between 2.225074e-308 and 1.797693e+308 to 15 significant figures, row.names, dates (between 0000-03-01 and 9999-12-31), times and sep2 for list columns where each cell can itself be a vector.

To save space, fwrite prefers to write wide numeric values in scientific notation – e.g. 10000000000 takes up much more space than 1e+10. Most file readers (e.g. fread) understand scientific notation, so there's no fidelity loss. Like in base R, users can control this by specifying the scipen argument, which follows the same rules as options('scipen'). fwrite will see how much space a value will take to write in scientific vs. decimal notation, and will only write in scientific notation if the latter is more than scipen characters wider. For 10000000000, then, 1e+10 will be written whenever scipen<6.

CSVY Support:

The following fields will be written to the header of the file and surrounded by --- on top and bottom:

- source - Contains the R version and data.table version used to write the file
- creation_time_utc - Current timestamp in UTC time just before the header is written
- schema with element fields giving name-type (class) pairs for the table; multi-class objects (e.g. c('POSIXct', 'POSIXt')) will have their first class written.
- header - same as col.names (which is header on input)
- sep
- sep2
- eol
- na.strings - same as na
- dec
- qmethod
- logical01

References

https://howardhinnant.github.io/date_algorithms.html
https://en.wikipedia.org/wiki/Decimal_mark

See Also

setDTthreads, fread, write.csv, write.table, bit64::integer64

Examples

```
DF = data.frame(A=1:3, B=c("foo","A,Name","baz"))
fwrite(DF)
write.csv(DF, row.names=FALSE, quote=FALSE)  # same

fwrite(DF, row.names=TRUE, quote=TRUE)
write.csv(DF)                                # same

DF = data.frame(A=c(2.1,-1.234e-307,pi), B=c("foo","A,Name","bar"))
fwrite(DF, quote='auto')        # Just DF[2,2] is auto quoted
write.csv(DF, row.names=FALSE)  # same numeric formatting

DT = data.table(A=c(2,5.6,-3),B=list(1:3,c("foo","A,Name","bar"),round(pi*1:3,2)))
fwrite(DT)
fwrite(DT, sep="|", sep2=c("{",",","}"))

## Not run:

set.seed(1)
DT = as.data.table( lapply(1:10, sample,
         x=as.numeric(1:5e7), size=5e6))                          #    382MiB
system.time(fwrite(DT, "/dev/shm/tmp1.csv"))                      #      0.8s
system.time(write.csv(DT, "/dev/shm/tmp2.csv",                    #     60.6s
                      quote=FALSE, row.names=FALSE))
system("diff /dev/shm/tmp1.csv /dev/shm/tmp2.csv")                # identical
```

```
set.seed(1)
N = 1e7
DT = data.table(
  str1=sample(sprintf("%010d",sample(N,1e5,replace=TRUE)), N, replace=TRUE),
  str2=sample(sprintf("%09d",sample(N,1e5,replace=TRUE)), N, replace=TRUE),
  str3=sample(sapply(sample(2:30, 100, TRUE), function(n)
     paste0(sample(LETTERS, n, TRUE), collapse="")), N, TRUE),
  str4=sprintf("%05d",sample(sample(1e5,50),N,TRUE)),
  num1=sample(round(rnorm(1e6,mean=6.5,sd=15),2), N, replace=TRUE),
  num2=sample(round(rnorm(1e6,mean=6.5,sd=15),10), N, replace=TRUE),
  str5=sample(c("Y","N"),N,TRUE),
  str6=sample(c("M","F"),N,TRUE),
  int1=sample(ceiling(rexp(1e6)), N, replace=TRUE),
  int2=sample(N,N,replace=TRUE)-N/2
)                                                                    #    775MiB
system.time(fwrite(DT,"/dev/shm/tmp1.csv"))                          #      1.1s
system.time(write.csv(DT,"/dev/shm/tmp2.csv",                        #     63.2s
                      row.names=FALSE, quote=FALSE))
system("diff /dev/shm/tmp1.csv /dev/shm/tmp2.csv")                   # identical

unlink("/dev/shm/tmp1.csv")
unlink("/dev/shm/tmp2.csv")

## End(Not run)
```

---

groupingsets *Grouping Set aggregation for data tables*

Description

Calculate aggregates at various levels of groupings producing multiple (sub-)totals. Reflects SQLs *GROUPING SETS* operations.

Usage

```
rollup(x, ...)
## S3 method for class 'data.table'
rollup(x, j, by, .SDcols, id = FALSE, label = NULL, ...)
cube(x, ...)
## S3 method for class 'data.table'
cube(x, j, by, .SDcols, id = FALSE, label = NULL, ...)
groupingsets(x, ...)
## S3 method for class 'data.table'
groupingsets(x, j, by, sets, .SDcols,
  id = FALSE, jj, label = NULL, enclos = parent.frame(), ...)
```

Arguments

x	data.table.
...	argument passed to custom user methods. Ignored for data.table methods.
j	expression passed to data.table j.
by	character column names by which we are grouping.
sets	list of character vector reflecting grouping sets, used in groupingsets for flexibility.
.SDcols	columns to be used in j expression in .SD object.
id	logical default FALSE. If TRUE it will add leading column with bit mask of grouping sets.
jj	quoted version of j argument, for convenience. When provided function will ignore j argument.
label	label(s) to be used in the 'total' rows in the grouping variable columns of the output, that is, in rows where the grouping variable has been aggregated. Can be a named list of scalars, or a scalar, or NULL. Defaults to NULL, which results in the grouping variables having NA in their 'total' rows. See Details.
enclos	the environment containing the symbols referenced by jj. When writing functions that accept a j environment for non-standard evaluation by **data.table**, substitute() it and forward it to groupingsets using the jj argument, set this to the parent.frame() of the function that captures j.

Details

All three functions rollup, cube, groupingsets are generic methods, data.table methods are provided.

The label argument can be a named list of scalars, or a scalar, or NULL. When label is a list, each element name must be (1) a variable name in by, or (2) the first element of the class in the data.table x of a variable in by, or (3) one of 'character', 'integer', 'numeric', 'factor', 'Date', 'IDate'. The order of the list elements is not important. A label specified by variable name will apply only to that variable, while a label specified by first element of a class will apply to all variables in by for which the first element of the class of the variable in x matches the label element name, except for variables that have a label specified by variable name (that is, specification by variable name takes precedence over specification by class). For label elements with name in by, the class of the label value must be the same as the class of the variable in x. For label elements with name not in by, the first element of the class of the label value must be the same as the label element name. For example, label = list(integer = 999, IDate = as.Date("3000-01-01")) would produce an error because class(999)[1] is not "integer" and class(as.Date("3000-01-01"))[1] is not "IDate". A corrected specification would be label = list(integer = 999L, IDate = as.IDate("3000-01-01")).

The label = <scalar> option provides a shorter alternative in the case where only one class of grouping variable requires a label. For example, label = list(character = "Total") can be shortened to label = "Total". When this option is used, the label will be applied to all variables in by for which the first element of the class of the variable in x matches the first element of the class of the scalar.

Value

A data.table with various aggregates.

References

https://www.postgresql.org/docs/9.5/static/queries-table-expressions.html#QUERIES-GROUPING-SETS
https://www.postgresql.org/docs/9.5/static/functions-aggregate.html#FUNCTIONS-GROUPING-TABLE

See Also

data.table, rbindlist

Examples

```
n = 24L
set.seed(25)
DT <- data.table(
  color = sample(c("green","yellow","red"), n, TRUE),
  year = as.Date(sample(paste0(2011:2015,"-01-01"), n, TRUE)),
  status = as.factor(sample(c("removed","active","inactive","archived"), n, TRUE)),
  amount = sample(1:5, n, TRUE),
  value = sample(c(3, 3.5, 2.5, 2), n, TRUE)
)

# rollup
by_vars = c("color", "year", "status")
rollup(DT, j=sum(value), by=by_vars) # default id=FALSE
rollup(DT, j=sum(value), by=by_vars, id=TRUE)
rollup(DT, j=lapply(.SD, sum), by=by_vars, id=TRUE, .SDcols="value")
rollup(DT, j=c(list(count=.N), lapply(.SD, sum)), by=by_vars, id=TRUE)
rollup(DT, j=sum(value), by=by_vars,
       # specify label by variable name
       label=list(color="total", year=as.Date("3000-01-01"), status=factor("total")))
rollup(DT, j=sum(value), by=by_vars,
       # specify label by variable name and first element of class
       label=list(color="total", Date=as.Date("3000-01-01"), factor=factor("total")))
# label is character scalar so applies to color only
rollup(DT, j=sum(value), by=by_vars, label="total")
rollup(DT, j=.N, by=c("color", "year", "status", "value"),
       # label can be explicitly specified as NA or NaN
      label = list(color=NA_character_, year=as.Date(NA), status=factor(NA), value=NaN))

# cube
cube(DT, j = sum(value), by = c("color","year","status"), id=TRUE)
cube(DT, j = lapply(.SD, sum), by = c("color","year","status"), id=TRUE, .SDcols="value")
cube(DT, j = c(list(count=.N), lapply(.SD, sum)), by = c("color","year","status"), id=TRUE)

# groupingsets
groupingsets(DT, j = c(list(count=.N), lapply(.SD, sum)), by = c("color","year","status"),
             sets = list("color", c("year","status"), character()), id=TRUE)
```

IDateTime *Integer based date class*

Description

Classes (IDate and ITime) with *integer* storage for fast sorting and grouping.

IDate inherits from the base class Date; the main difference is that the latter uses double storage, allowing e.g. for fractional dates at the cost of storage & sorting inefficiency.

Using IDate, if sub-day granularity is needed, use a second ITime column. IDateTime() facilitates building such paired columns.

Lastly, there are date-time helpers for extracting parts of dates as integers, for example the year (year()), month (month()), or day in the month (mday()); see Usage and Examples.

Usage

```
as.IDate(x, ...)
## Default S3 method:
as.IDate(x, ..., tz = attr(x, "tzone", exact=TRUE))
## S3 method for class 'Date'
as.IDate(x, ...)
## S3 method for class 'IDate'
as.Date(x, ...)
## S3 method for class 'IDate'
as.POSIXct(x, tz = "UTC", time = 0, ...)
## S3 method for class 'IDate'
round(x, digits = c("weeks", "months", "quarters","years"), ...)

as.ITime(x, ...)
## Default S3 method:
as.ITime(x, ...)
## S3 method for class 'POSIXlt'
as.ITime(x, ms = 'truncate', ...)
## S3 method for class 'ITime'
round(x, digits = c("hours", "minutes"), ...)
## S3 method for class 'ITime'
trunc(x, units = c("hours", "minutes"), ...)

## S3 method for class 'ITime'
as.POSIXct(x, tz = "UTC", date = Sys.Date(), ...)
## S3 method for class 'ITime'
as.character(x, ...)
## S3 method for class 'ITime'
format(x, ...)

IDateTime(x, ...)
## Default S3 method:
```

```
IDateTime(x, ...)

second(x)
minute(x)
hour(x)
yday(x)
wday(x)
mday(x)
week(x)
isoweek(x)
isoyear(x)
month(x)
quarter(x)
year(x)
yearmon(x)
yearqtr(x)
```

Arguments

x	an object
...	arguments to be passed to or from other methods. For as.IDate.default, arguments are passed to as.Date. For as.ITime.default, arguments are passed to as.POSIXlt.
tz	time zone (see strptime).
date	date object convertible with as.IDate.
time	time-of-day object convertible with as.ITime.
digits	really units; one of the units listed for rounding. May be abbreviated. Named digits for consistency with the S3 generic.
units	one of the units listed for truncating. May be abbreviated.
ms	For as.ITime methods, what should be done with sub-second fractions of input? Valid values are 'truncate' (floor), 'nearest' (round), and 'ceil' (ceiling). See Details.

Details

IDate is a date class derived from Date. It has the same internal representation as the Date class, except the storage mode is integer. IDate is a relatively simple wrapper, and it should work in almost all situations as a replacement for Date. The main limitations of integer storage are (1) fractional dates are not supported (use IDateTime() instead) and (2) the range of supported dates is bounded by .Machine$integer.max dates away from January 1, 1970 (a rather impractical limitation as these dates are roughly 6 million years in the future/past, but consider this your caveat).

Functions that use Date objects generally work for IDate objects. This package provides specific methods for IDate objects for mean, cut, seq, c, rep, and split to return an IDate object.

ITime is a time-of-day class stored as the integer number of seconds in the day. as.ITime does not allow days longer than 24 hours. Because ITime is stored in seconds, you can add it to a POSIXct object, but you should not add it to a Date object.

For as.ITime, note that the string "24:00:00" is parsed as "00:00:00". This is because the conversion uses as.POSIXct, which treats "24:00:00" as midnight of the next day. This differs from ISO 8601 (which allows "24:00:00" to represent end-of-day), but aligns with POSIX standards. To represent end-of-day intervals, use "23:59:59" or arithmetic (e.g., as.ITime("23:59:59") + 1L).

We also provide S3 methods to convert to and from Date and POSIXct.

ITime is time zone-agnostic. When converting ITime and IDate to POSIXct with as.POSIXct, a time zone may be specified.

Inputs like '2018-05-15 12:34:56.789' are ambiguous from the perspective of an ITime object – the method of coercion of the 789 milliseconds is controlled by the ms argument to relevant methods. The default behavior (ms = 'truncate') is to use as.integer, which has the effect of truncating anything after the decimal. Alternatives are to round to the nearest integer (ms = 'nearest') or to round up (ms = 'ceil').

In as.POSIXct methods for ITime and IDate, the second argument is required to be tz based on the generic template, but to make converting easier, the second argument is interpreted as a date instead of a time zone if it is of type IDate or ITime. Therefore, you can use either of the following: as.POSIXct(time, date) or as.POSIXct(date,time).

IDateTime takes a date-time input and returns a data table with columns date and time.

Using integer storage allows dates and/or times to be used as data table keys. With positive integers with a range less than 100,000, grouping and sorting is fast because radix sorting can be used (see sort.list).

Several convenience functions like hour and quarter are provided to group or extract by hour, month, and other date-time intervals. as.POSIXlt is also useful. For example, as.POSIXlt(x)$mon is the integer month. The R base convenience functions weekdays, months, and quarters can also be used, but these return character values, so they must be converted to factors for use with data.table. isoweek is ISO 8601-consistent.

The round method for IDate's is useful for grouping and plotting. It can round to weeks, months, quarters, and years. Similarly, the round and trunc methods for ITime's are useful for grouping and plotting. They can round or truncate to hours and minutes. Note for ITime's with 30 seconds, rounding is inconsistent due to rounding off a 5. See 'Details' in round for more information.

Functions like week() and isoweek() provide week numbering functionality. week() computes completed or fractional weeks within the year, while isoweek() calculates week numbers according to ISO 8601 standards, which specify that the first week of the year is the one containing the first Thursday. This convention ensures that week boundaries align consistently with year boundaries, accounting for both year transitions and varying day counts per week.

Similarly, isoyear() returns the ISO 8601 year corresponding to the ISO week.

Value

For as.IDate, a class of IDate and Date with the date stored as the number of days since some origin.

For as.ITime, a class of ITime stored as the number of seconds in the day.

For IDateTime, a data table with columns idate and itime in IDate and ITime format.

second, minute, hour, yday, wday, mday, week, isoweek, isoyear, month, quarter, and year return integer values for second, minute, hour, day of year, day of week, day of month, week,

month, quarter, and year, respectively. `yearmon` and `yearqtr` return double values representing respectively `year + (month-1) / 12` and `year + (quarter-1) / 4`.

`second`, `minute`, `hour` are taken directly from the `POSIXlt` representation. All other values are computed from the underlying integer representation and comparable with the values of their `POSIXlt` representation of `x`, with the notable difference that while `yday`, `wday`, and `mon` are all 0-based, here they are 1-based.

Author(s)

Tom Short, t.short@ieee.org

References

G. Grothendieck and T. Petzoldt, "Date and Time Classes in R", R News, vol. 4, no. 1, June 2004.

H. Wickham, https://gist.github.com/hadley/10238.

ISO 8601, https://www.iso.org/iso/home/standards/iso8601.htm

See Also

`as.Date`, `as.POSIXct`, `strptime`, `DateTimeClasses`

Examples

```
# create IDate:
(d <- as.IDate("2001-01-01"))

# S4 coercion also works
identical(as.IDate("2001-01-01"), methods::as("2001-01-01", "IDate"))

# create ITime:
(t <- as.ITime("10:45"))

# S4 coercion also works
identical(as.ITime("10:45"), methods::as("10:45", "ITime"))

(t <- as.ITime("10:45:04"))

(t <- as.ITime("10:45:04", format = "%H:%M:%S"))

# "24:00:00" is parsed as "00:00:00"
as.ITime("24:00:00")

# Workaround for end-of-day: add 1 second to "23:59:59"
as.ITime("23:59:59") + 1L

as.POSIXct("2001-01-01") + as.ITime("10:45")

datetime <- seq(as.POSIXct("2001-01-01"), as.POSIXct("2001-01-03"), by = "5 hour")
(af <- data.table(IDateTime(datetime), a = rep(1:2, 5), key = c("a", "idate", "itime")))

af[, mean(a), by = "itime"]
```

```
af[, mean(a), by = list(hour = hour(itime))]
af[, mean(a), by = list(wday = factor(weekdays(idate)))]
af[, mean(a), by = list(wday = wday(idate))]

as.POSIXct(af$idate)
as.POSIXct(af$idate, time = af$itime)
as.POSIXct(af$idate, af$itime)
as.POSIXct(af$idate, time = af$itime, tz = "GMT")

as.POSIXct(af$itime, af$idate)
as.POSIXct(af$itime) # uses today's date

(seqdates <- seq(as.IDate("2001-01-01"), as.IDate("2001-08-03"), by = "3 weeks"))
round(seqdates, "months")

(seqtimes <- seq(as.ITime("07:00"), as.ITime("08:00"), by = 20))
round(seqtimes, "hours")
trunc(seqtimes, "hours")

# Examples for isoyear() and isoweek()
d1 = as.IDate("2019-12-30")
year(d1)
isoweek(d1)
isoyear(d1)

d2 = as.IDate("2016-01-01")
year(d2)
isoweek(d2)
isoyear(d2)
```

J *Creates a join* data.table

Description

Creates a data.table for use in i in a [.data.table join.

Usage

```
# DT[J(...)]                          # J() only for use inside DT[...]
# DT[.(...)]                          # .() only for use inside DT[...]
# DT[list(...)]                       # same; .(), list() and J() are identical
SJ(...)                             # DT[SJ(...)]
CJ(..., sorted=TRUE, unique=FALSE)  # DT[CJ(...)]
```

Arguments

...	Each argument is a vector. Generally each vector is the same length, but if they are not then the usual silent recycling is applied.

sorted	logical. Should setkey() be called on all the columns in the order they were passed to CJ?
unique	logical. When TRUE, only unique values of each vectors are used (automatically).

Details

SJ and CJ are convenience functions to create a data.table to be used in i when performing a data.table 'query' on x.

x[data.table(id)] is the same as x[J(id)] but the latter is more readable. Identical alternatives are x[list(id)] and x[.(id)].

When using a join table in i, x must either be keyed or the on argument be used to indicate the columns in x and i which should be joined. See [.data.table.

Value

J : the same result as calling list, for which J is a direct alias.

SJ : **S**orted **J**oin. The same value as J() but additionally setkey() is called on all columns in the order they were passed to SJ. For efficiency, to invoke a binary merge rather than a repeated binary full search for each row of i.

CJ : **C**ross **J**oin. A data.table is formed from the cross product of the vectors. For example, CJ on 10 ids and 100 dates, returns a 1000 row table containing all dates for all ids. If sorted = TRUE (default), setkey() is called on all columns in the order they were passed in to CJ. If sorted = FALSE, the result is unkeyed and input order is retained.

See Also

data.table, test.data.table

Examples

```
DT = data.table(A=5:1, B=letters[5:1])
setkey(DT, B)   # reorders table and marks it sorted
DT[J("b")]      # returns the 2nd row
DT[list("b")]   # same
DT[.("b")]      # same using the dot alias for list

# CJ usage examples
CJ(c(5, NA, 1), c(1, 3, 2))                 # sorted and keyed data.table
do.call(CJ, list(c(5, NA, 1), c(1, 3, 2)))  # same as above
CJ(c(5, NA, 1), c(1, 3, 2), sorted=FALSE)   # same order as input, unkeyed
# use for 'unique=' argument
x = c(1, 1, 2)
y = c(4, 6, 4)
CJ(x, y)              # output columns are automatically named 'x' and 'y'
CJ(x, y, unique=TRUE) # unique(x) and unique(y) are computed automatically
CJ(x, y, sorted = FALSE) # retain input order for y
```

`last` *First/last item of an object*

Description

Returns the first/last item of a vector or list, or the first/last row of a data.frame or data.table. The main difference to head/tail is that the default for `n` is 1 rather than 6.

Usage

```
first(x, n=1L, ...)
last(x, n=1L, ...)
```

Arguments

`x`	A vector, list, data.frame or data.table. Otherwise the S3 method of `xts::first` is deployed.
`n`	A numeric vector length 1. How many items to select.
`...`	Not applicable for `data.table` first/last. Any arguments here are passed through to `xts`'s first/last.

Value

If no other arguments are supplied it depends on the type of `x`. The first/last item of a vector or list. The first/last row of a `data.frame` or `data.table`. For other types, or if any argument is supplied in addition to `x` (such as `n`, or `keep` in `xts`) regardless of `x`'s type, then `xts::first`/ `xts::last` is called if `xts` has been loaded, otherwise `utils::head`/`utils::tail`.

Note

For zero-length vectors, `first(x)` and `last(x)` mimic `head(x, 1)` and `tail(x, 1)` by returning an empty vector instead of `NA`. However, unlike `head()`/`tail()` and base R subsetting (e.g., `x[1]`), they do not preserve attributes like names.

See Also

`NROW`, `head`, `tail`

Examples

```
first(1:5) # [1] 1
x = data.table(x=1:5, y=6:10)
first(x) # same as head(x, 1)

last(1:5) # [1] 5
x = data.table(x=1:5, y=6:10)
last(x) # same as tail(x, 1)
```

like *Convenience function for calling grep*

Description

Intended for use in `i` in `[.data.table`, i.e., for subsetting/filtering.

Syntax should be familiar to SQL users, with interpretation as regex.

Usage

```
like(vector, pattern, ignore.case = FALSE, fixed = FALSE, perl = FALSE)
vector %like% pattern
vector %ilike% pattern
vector %flike% pattern
vector %plike% pattern
```

Arguments

`vector`	Either a `character` or a `factor` vector.
`pattern`	Pattern to be matched
`ignore.case`	`logical`; is `pattern` case-sensitive?
`fixed`	`logical`; should `pattern` be interpreted as a literal string (i.e., ignoring regular expressions)?
`perl`	`logical`; is `pattern` Perl-compatible regular expression?

Details

Internally, `like` is essentially a wrapper around `base::grepl`, except that it is smarter about handling `factor` input (`base::grep` uses slow `as.character` conversion).

Value

Logical vector, `TRUE` for items that match `pattern`.

Note

Current implementation does not make use of sorted keys.

See Also

`base::grepl`

Examples

```
DT = data.table(Name=c("Mary","George","Martha"), Salary=c(2,3,4))
DT[Name %like% "^Mar"]
DT[Name %ilike% "mar"]
DT[Name %flike% "Mar"]
DT[Name %plike% "(?=Ma)(?=.*y)"]
```

`measure` *Specify measure.vars via regex or separator*

Description

These functions compute an integer vector or list for use as the `measure.vars` argument to `melt`. Each measured variable name is converted into several groups that occupy different columns in the output melted data. `measure` allows specifying group names/conversions in R code (each group and conversion specified as an argument) whereas `measurev` allows specifying group names/conversions using data values (each group and conversion specified as a list element). See `vignette("datatable-reshape")` for more info.

Usage

```
measure(..., sep, pattern, cols, multiple.keyword="value.name")
measurev(fun.list, sep, pattern, cols, multiple.keyword="value.name")
```

Arguments

`...`	One or more (1) symbols (without argument name; symbol is used for group name) or (2) functions to convert the groups (with argument name that is used for group name). Must have same number of arguments as groups that are specified by either `sep` or `pattern` arguments.
`fun.list`	Named list which must have the same number of elements as groups that are specified by either `sep` or `pattern` arguments. Each name used for a group name, and each value must be either a function (to convert the group from a character vector to an atomic vector of the same size) or NULL (no conversion).
`sep`	Separator to split each element of `cols` into groups. Columns that result in the maximum number of groups are considered measure variables.
`pattern`	Perl-compatible regex with capture groups to match to `cols`. Columns that match the regex are considered measure variables.
`cols`	A character vector of column names.
`multiple.keyword`	A string, if used as a group name, then measure returns a list and melt returns multiple value columns (with names defined by the unique values in that group). Otherwise if the string not used as a group name, then measure returns a vector and melt returns a single value column.

See Also

melt, https://github.com/Rdatatable/data.table/wiki/Getting-started

Examples

```
(two.iris = data.table(datasets::iris)[c(1,150)])
# melt into a single value column.
melt(two.iris, measure.vars = measure(part, dim, sep="."))
# do the same, programmatically with measurev
my.list = list(part=NULL, dim=NULL)
melt(two.iris, measure.vars=measurev(my.list, sep="."))
# melt into two value columns, one for each part.
melt(two.iris, measure.vars = measure(value.name, dim, sep="."))
# melt into two value columns, one for each dim.
melt(two.iris, measure.vars = measure(part, value.name, sep="."))
# melt using sep, converting child number to integer.
(two.families = data.table(sex_child1="M", sex_child2="F", age_child1=10, age_child2=20))
print(melt(two.families, measure.vars = measure(
  value.name, child=as.integer,
  sep="_child"
)), class=TRUE)
# same melt using pattern.
print(melt(two.families, measure.vars = measure(
  value.name, child=as.integer,
  pattern="(.*)_child(.)"
)), class=TRUE)
# same melt with pattern and measurev function list.
print(melt(two.families, measure.vars = measurev(
  list(value.name=NULL, child=as.integer),
  pattern="(.*)_child(.)"
)), class=TRUE)
# inspired by data(who, package="tidyr")
(who <- data.table(id=1, new_sp_m5564=2, newrel_f65=3))
# melt to three variable columns, all character.
melt(who, measure.vars = measure(diagnosis, gender, ages, pattern="new_?(.*)_(.)(.*)"))
# melt to five variable columns, two numeric (with custom conversion).
print(melt(who, measure.vars = measure(
  diagnosis, gender, ages,
  ymin=as.numeric,
  ymax=function(y)ifelse(nzchar(y), as.numeric(y), Inf),
  pattern="new_?(.*)_(.)(([0-9]{2})([0-9]{0,2}))"
)), class=TRUE)
```

`melt.data.table` *Fast melt for data.table*

Description

`melt` is `data.table`'s wide-to-long reshaping tool. We provide an S3 method for melting `data.tables`. It is written in C for speed and memory efficiency. Since `v1.9.6`, `melt.data.table` allows melting into multiple columns simultaneously.

Usage

```
## fast melt a data.table
## S3 method for class 'data.table'
melt(data, id.vars, measure.vars,
    variable.name = "variable", value.name = "value",
    ..., na.rm = FALSE, variable.factor = TRUE,
    value.factor = FALSE,
    verbose = getOption("datatable.verbose"))
```

Arguments

data	A data.table object to melt.
id.vars	vector of id variables. Can be integer (corresponding id column numbers) or character (id column names) vector, perhaps created using patterns(). If missing, all non-measure columns will be assigned to it. If integer, must be positive; see Details.
measure.vars	Measure variables for melting. Can be missing, vector, list, or pattern-based. • When missing, measure.vars will become all columns outside id.vars. • Vector can be integer (implying column numbers) or character (column names). • list is a generalization of the vector version – each element of the list (which should be integer or character as above) will become a melted column. • Pattern-based column matching can be achieved with the regular expression-based patterns (regex without capture groups; matching column names are used in the variable.name output column), or measure (regex with capture groups; each capture group becomes an output column). For convenience/clarity in the case of multiple melted columns, resulting column names can be supplied as names to the elements measure.vars (in the list and patterns usages). See also Examples.
variable.name	name (default 'variable') of output column containing information about which input column(s) were melted. If measure.vars is an integer/character vector, then each entry of this column contains the name of a melted column from data. If measure.vars is a list of integer/character vectors, then each entry of this column contains an integer indicating an index/position in each of those vectors. If measure.vars has attribute variable_table then it must be a data table with nrow = length of measure.vars vector(s), each row describing the corresponding measured variables(s), (typically created via measure) and its columns will be output instead of the variable.name column.
value.name	name for the molten data values column(s). The default name is 'value'. Multiple names can be provided here for the case when measure.vars is a list, though note well that the names provided in measure.vars take precedence.
na.rm	If TRUE, NA values will be removed from the molten data.
variable.factor	If TRUE, the variable column will be converted to factor, else it will be a character column.

value.factor	If TRUE, the value column will be converted to factor, else the molten value type is left unchanged.
verbose	TRUE turns on status and information messages to the console. Turn this on by default using options(datatable.verbose=TRUE). The quantity and types of verbosity may be expanded in future.
...	any other arguments to be passed to/from other methods.

Details

If id.vars and measure.vars are both missing, all non-numeric/integer/logical columns are assigned as id variables and the rest as measure variables. If only one of id.vars or measure.vars is supplied, the rest of the columns will be assigned to the other. Both id.vars and measure.vars can have the same column more than once and the same column can be both as id and measure variables.

melt.data.table also accepts list columns for both id and measure variables.

When all measure.vars are not of the same type, they'll be coerced according to the hierarchy list > character > numeric > integer > logical. For example, if any of the measure variables is a list, then entire value column will be coerced to a list.

From version 1.9.6, melt gains a feature with measure.vars accepting a list of character or integer vectors as well to melt into multiple columns in a single function call efficiently. If a vector in the list contains missing values, or is shorter than the max length of the list elements, then the output will include runs of missing values at the specified position, or at the end. The functions patterns and measure can be used to provide regular expression patterns. When used along with melt, if cols argument is not provided, the patterns will be matched against names(data), for convenience.

Attributes are preserved if all value columns are of the same type. By default, if any of the columns to be melted are of type factor, it'll be coerced to character type. To get a factor column, set value.factor = TRUE. melt.data.table also preserves ordered factors.

Historical note: melt.data.table was originally designed as an enhancement to reshape2::melt in terms of computing and memory efficiency. reshape2 has since been superseded in favour of tidyr, and melt has had a generic defined within data.table since v1.9.6 in 2015, at which point the dependency between the packages became more etymological than programmatic. We thank the reshape2 authors for the inspiration.

Value

An unkeyed data.table containing the molten data.

See Also

dcast, https://cran.r-project.org/package=reshape

Examples

```
set.seed(45)
require(data.table)
DT <- data.table(
```

```
  i_1 = c(1:5, NA),
  n_1 = c(NA, 6, 7, 8, 9, 10),
  f_1 = factor(sample(c(letters[1:3], NA), 6L, TRUE)),
  f_2 = factor(c("z", "a", "x", "c", "x", "x"), ordered=TRUE),
  c_1 = sample(c(letters[1:3], NA), 6L, TRUE),
  c_2 = sample(c(LETTERS[1:2], NA), 6L, TRUE),
  d_1 = as.Date(c(1:3,NA,4:5), origin="2013-09-01"),
  d_2 = as.Date(6:1, origin="2012-01-01")
)
# add a couple of list cols
DT[, l_1 := DT[, list(c=list(rep(i_1, sample(5, 1L)))), by = i_1]$c]
DT[, l_2 := DT[, list(c=list(rep(c_1, sample(5, 1L)))), by = i_1]$c]

# id.vars, measure.vars as character/integer/numeric vectors
melt(DT, id.vars=1:2, measure.vars="f_1")
melt(DT, id.vars=c("i_1", "n_1"), measure.vars=3) # same as above
melt(DT, id.vars=1:2, measure.vars=3L, value.factor=TRUE) # same, but 'value' is factor
melt(DT, id.vars=1:2, measure.vars=3:4, value.factor=TRUE) # 'value' is *ordered* factor

# preserves attribute when types are identical, ex: Date
melt(DT, id.vars=3:4, measure.vars=c("d_1", "d_2"))
melt(DT, id.vars=3:4, measure.vars=c("n_1", "d_1")) # attribute not preserved

# on list
melt(DT, id.vars=1, measure.vars=c("l_1", "l_2")) # value is a list
suppressWarnings(
  melt(DT, id.vars=1, measure.vars=c("c_1", "l_1")) # c1 coerced to list, with warning
)

# on character
melt(DT, id.vars=1, measure.vars=c("c_1", "f_1")) # value is char
suppressWarnings(
  melt(DT, id.vars=1, measure.vars=c("c_1", "n_1")) # n_1 coerced to char, with warning
)

# on na.rm=TRUE. NAs are removed efficiently, from within C
melt(DT, id.vars=1, measure.vars=c("c_1", "c_2"), na.rm=TRUE) # remove NA

# measure.vars can be also a list
# melt "f_1,f_2" and "d_1,d_2" simultaneously, retain 'factor' attribute
# convenient way using internal function patterns()
melt(DT, id.vars=1:2, measure.vars=patterns("^f_", "^d_"), value.factor=TRUE)
melt(DT, id.vars=patterns("[in]"), measure.vars=patterns("^f_", "^d_"), value.factor=TRUE)
# same as above, but provide list of columns directly by column names or indices
melt(DT, id.vars=1:2, measure.vars=list(3:4, c("d_1", "d_2")), value.factor=TRUE)
# same as above, but provide names directly:
melt(DT, id.vars=1:2, measure.vars=patterns(f="^f_", d="^d_"), value.factor=TRUE)

# na.rm=TRUE removes rows with NAs in any 'value' columns
melt(DT, id.vars=1:2, measure.vars=patterns("f_", "d_"), value.factor=TRUE, na.rm=TRUE)

# 'na.rm=TRUE' also works with list column, but note that is.na only
# returns TRUE if the list element is a length=1 vector with an NA.
```

```
is.na(list(one.NA=NA, two.NA=c(NA,NA)))
melt(DT, id.vars=1:2, measure.vars=patterns("l_", "d_"), na.rm=FALSE)
melt(DT, id.vars=1:2, measure.vars=patterns("l_", "d_"), na.rm=TRUE)

# measure list with missing/short entries results in output with runs of NA
DT.missing.cols <- DT[, .(d_1, d_2, c_1, f_2)]
melt(DT.missing.cols, measure.vars=list(d=1:2, c="c_1", f=c(NA, "f_2")))

# specifying columns to melt via separator.
melt(DT.missing.cols, measure.vars=measure(value.name, number=as.integer, sep="_"))

# specifying columns to melt via regex.
melt(DT.missing.cols, measure.vars=measure(value.name, number=as.integer, pattern="(.)_(.)"))
melt(DT.missing.cols, measure.vars=measure(value.name, number=as.integer, pattern="([dc])_(.)"))

# cols arg of measure can be used if you do not want to use regex
melt(DT.missing.cols, measure.vars=measure(
  value.name, number=as.integer, sep="_", cols=c("d_1","d_2","c_1")))
```

`merge` *Merge two data.tables*

Description

Fast merge of two `data.tables`. The `data.table` method behaves similarly to `data.frame` except that row order is specified, and by default the columns to merge on are chosen:

- at first based on the shared key columns, and if there are none,
- then based on key columns of the first argument `x`, and if there are none,
- then based on the common columns between the two `data.tables`.

Use the `by`, `by.x` and `by.y` arguments explicitly to override this default.

Usage

```
## S3 method for class 'data.table'
merge(x, y, by = NULL, by.x = NULL, by.y = NULL, all = FALSE,
all.x = all, all.y = all, sort = TRUE, suffixes = c(".x", ".y"), no.dups = TRUE,
allow.cartesian=getOption("datatable.allow.cartesian"),  # default FALSE
incomparables = NULL, ...)
```

Arguments

`x, y`	`data tables`. `y` is coerced to a `data.table` if it isn't one already.
`by`	A vector of shared column names in `x` and `y` to merge on. This defaults to the shared key columns between the two tables. If `y` has no key columns, this defaults to the key of `x`.
`by.x, by.y`	Vectors of column names in `x` and `y` to merge on.

all	logical; all = TRUE is shorthand to save setting both all.x = TRUE and all.y = TRUE.
all.x	logical; if TRUE, rows from x which have no matching row in y are included. These rows will have 'NA's in the columns that are usually filled with values from y. The default is FALSE so that only rows with data from both x and y are included in the output.
all.y	logical; analogous to all.x above.
sort	logical. If TRUE (default), the rows of the merged data.table are sorted by setting the key to the by / by.x columns. If FALSE, unlike base R's merge for which row order is unspecified, the row order in x is retained (including retaining the position of missing entries when all.x=TRUE), followed by y rows that don't match x (when all.y=TRUE) retaining the order those appear in y.
suffixes	A character(2) specifying the suffixes to be used for making non-by column names unique. The suffix behaviour works in a similar fashion as the merge.data.frame method does.
no.dups	logical indicating that suffixes are also appended to non-by.y column names in y when they have the same column name as any by.x.
allow.cartesian	See allow.cartesian in [.data.table.
incomparables	values which cannot be matched and therefore are excluded from by columns.
...	Not used at this time.

Details

merge is a generic function in base R. It dispatches to either the merge.data.frame method or merge.data.table method depending on the class of its first argument. Note that, unlike SQL join, NA is matched against NA (and NaN against NaN) while merging.

For a more data.table-centric way of merging two data.tables, see [.data.table; e.g., x[y, ...]. See FAQ 1.11 for a detailed comparison of merge and x[y, ...].

Value

A new data.table based on the merged data tables, and sorted by the columns set (or inferred for) the by argument if argument sort is set to TRUE.

See Also

data.table, setkey, [.data.table, merge.data.frame

Examples

```
(dt1 <- data.table(A = letters[1:10], X = 1:10, key = "A"))
(dt2 <- data.table(A = letters[5:14], Y = 1:10, key = "A"))
merge(dt1, dt2)
merge(dt1, dt2, all = TRUE)

(dt1 <- data.table(A = letters[rep(1:3, 2)], X = 1:6, key = "A"))
```

```
(dt2 <- data.table(A = letters[rep(2:4, 2)], Y = 6:1, key = "A"))
merge(dt1, dt2, allow.cartesian=TRUE)

(dt1 <- data.table(A = c(rep(1L, 5), 2L), B = letters[rep(1:3, 2)], X = 1:6, key = c("A", "B")))
(dt2 <- data.table(A = c(rep(1L, 5), 2L), B = letters[rep(2:4, 2)], Y = 6:1, key = c("A", "B")))
merge(dt1, dt2)
merge(dt1, dt2, by="B", allow.cartesian=TRUE)

# test it more:
d1 <- data.table(a=rep(1:2,each=3), b=1:6, key=c("a", "b"))
d2 <- data.table(a=0:1, bb=10:11, key="a")
d3 <- data.table(a=0:1, key="a")
d4 <- data.table(a=0:1, b=0:1, key=c("a", "b"))

merge(d1, d2)
merge(d2, d1)
merge(d1, d2, all=TRUE)
merge(d2, d1, all=TRUE)

merge(d3, d1)
merge(d1, d3)
merge(d1, d3, all=TRUE)
merge(d3, d1, all=TRUE)

merge(d1, d4)
merge(d1, d4, by="a", suffixes=c(".d1", ".d4"))
merge(d4, d1)
merge(d1, d4, all=TRUE)
merge(d4, d1, all=TRUE)

# setkey is automatic by default
set.seed(1L)
d1 <- data.table(a=sample(rep(1:3,each=2)), z=1:6)
d2 <- data.table(a=2:0, z=10:12)
merge(d1, d2, by="a")
merge(d1, d2, by="a", all=TRUE)

# using by.x and by.y
setnames(d2, "a", "b")
merge(d1, d2, by.x="a", by.y="b")
merge(d1, d2, by.x="a", by.y="b", all=TRUE)
merge(d2, d1, by.x="b", by.y="a")

# using incomparables values
d1 <- data.table(a=c(1,2,NA,NA,3,1), z=1:6)
d2 <- data.table(a=c(1,2,NA), z=10:12)
merge(d1, d2, by="a")
merge(d1, d2, by="a", incomparables=NA)
```

`mergelist` *Merge multiple data.tables*

Description

Faster merge of multiple data.tables.

Usage

```
mergelist(l, on, cols=NULL,
  how=c("left", "inner", "full", "right", "semi", "anti", "cross"),
  mult, join.many=getOption("datatable.join.many"))
setmergelist(l, on, cols=NULL,
  how=c("left", "inner", "full", "right", "semi", "anti", "cross"),
  mult, join.many=getOption("datatable.join.many"))
```

Arguments

l	list of data.tables to merge.
on	character vector of column names to merge on; when missing, the key of the *join-to* table is used (see Details).
cols	Optional list of character column names corresponding to tables in l, used to subset columns during merges. NULL means all columns, all tables; NULL entries in a list means all columns for the corresponding table.
how	Character string, controls how to merge tables. Allowed values are "left" (default), "inner", "full", "right", "semi", "anti", and "cross". See Details.
mult	Character string, controls how to proceed when multiple rows in the *join-to* table match to the row in the *join-from* table. Allowed values are "error", "all", "first", "last". The default value depends on how; see Details. See Examples for how to detect duplicated matches. When using "all", we recommend specifying join.many=FALSE as a precaution to prevent unintended explosion of rows.
join.many	logical, defaulting to getOption("datatable.join.many"), which is TRUE by default; when FALSE and mult="all", an error is thrown when any *many-to-many* matches are detected between pairs of tables. This is essentially a stricter version of the allow.cartesian option in [.data.table. Note that the option "datatable.join.many" also controls the behavior of joins in [.data.table.

Details

Note: these functions should be considered experimental. Users are encouraged to provide feedback in our issue tracker.

Merging is performed sequentially from "left to right", so that for l of 3 tables, it will do something like merge(merge(l[[1L]], l[[2L]]), l[[3L]]). *Non-equi joins* are not supported. Column names to merge on must be common in both tables on each merge.

Arguments on, how, mult, join.many could be lists as well, each of length length(l)-1L, to provide argument to be used for each single tables pair to merge, see examples.

The terms *join-to* and *join-from* indicate which in a pair of tables is the "baseline" or "authoritative" source – this governs the ordering of rows and columns. Whether each refers to the "left" or "right" table of a pair depends on the how argument:

1. how %in% c("left", "semi", "anti"): *join-to* is *RHS*, *join-from* is *LHS*.
2. how %in% c("inner", "full", "cross"): *LHS* and *RHS* tables are treated equally, so that the terms are interchangeable.
3. how == "right": *join-to* is *LHS*, *join-from* is *RHS*.

Using mult="error" will throw an error when multiple rows in *join-to* table match to the row in *join-from* table. It should not be used just to detect duplicates, which might not have matching row, and thus would silently be missed.

When not specified, mult takes its default depending on the how argument:

1. When how %in% c("left", "inner", "full", "right"), mult="error".
2. When how %in% c("semi", "anti"), mult="last", although this is equivalent to mult="first".
3. When how == "cross", mult="all".

When the on argument is missing, it will be determined based how argument:

1. When how %in% c("left", right", "semi", "anti")}, \code{on} becomes the key column(s) of the \emph{join-to} table. \item When \code{how %in% c("inner", full"), if only one table has a key, then this key is used; if both tables have keys, then on = intersect(key(lhs), key(rhs)), having its order aligned to shorter key.

When joining tables that are not directly linked to a single table, e.g. a snowflake schema (see References), a *right* outer join can be used to optimize the sequence of merges, see Examples.

Value

A new data.table based on the merged objects.

For setmergelist, if possible, a copy of the inputs is avoided.

Note

Using how="inner" or how="full" together with mult!="all" is sub-efficient. Unlike during joins in [.data.table, it will apply mult on both tables. This ensures that the join is symmetric so that the *LHS* and *RHS* tables can be swapped, regardless of mult. It is always possible to apply a mult-like filter manually and join using mult="all".

Using join.many=FALSE is also sub-efficient. Note that it only takes effect when mult="all". If input data are verified not to have duplicate matches, then this can safely use the default TRUE. Otherwise, for mult="all" merges it is recommended to use join.many=FALSE, unless of course *many-to-many* joins, duplicating rows, are intended.

References

https://en.wikipedia.org/wiki/Snowflake_schema, https://en.wikipedia.org/wiki/Star_schema

See Also

[.data.table, merge.data.table

Examples

```
l = list(
  data.table(id1=c(1:4, 2:5), v1=1:8),
  data.table(id1=2:3, v2=1:2),
  data.table(id1=3:5, v3=1:3)
)
mergelist(l, on="id1")

## using keys
l = list(
  data.table(id1=c(1:4, 2:5), v1=1:8),
  data.table(id1=3:5, id2=1:3, v2=1:3, key="id1"),
  data.table(id2=1:4, v3=4:1, key="id2")
)
mergelist(l)

## select columns
l = list(
  data.table(id1=c(1:4, 2:5), v1=1:8, v2=8:1),
  data.table(id1=3:5, v3=1:3, v4=3:1, v5=1L, key="id1")
)
mergelist(l, cols=list(NULL, c("v3", "v5")))

## different arguments for each merge pair
l = list(
  data.table(id1=1:4, id2=4:1),
  data.table(id1=c(1:3, 1:2), v2=c(1L, 1L, 1:2, 2L)),
  data.table(id2=4:5)
)
mergelist(l,
  on = list("id1", "id2"),     ## first merge on id1, second on id2
  how = list("inner", "anti"), ## first inner join, second anti join
  mult = list("last", NULL))   ## use default 'mult' in second join

## detecting duplicates matches
l = list(
  data.table(id1=c(1:4, 2:5), v1=1:8), ## dups in LHS are fine
  data.table(id1=c(2:3, 2L), v2=1:3),  ## dups in RHS
  data.table(id1=3:5, v3=1:3)
)
lapply(l[-1L], `[`, j = if (.N>1L) .SD, by = "id1") ## duplicated rows
try(mergelist(l, on="id1"))


## 'star schema' and 'snowflake schema' examples (realistic data sizes)

### populate fact: US population by state and date

gt = state.x77[, "Population"]
gt = data.table(state_id=seq_along(state.name), p=gt[state.name] / sum(gt), k=1L)
tt = as.IDate(paste0(as.integer(time(uspop)), "-01-01"))
tt = as.data.table(stats::approx(tt, c(uspop), tt[1L]:tt[length(tt)]))
```

```
tt = tt[, .(date=as.IDate(x), date_id=seq_along(x), pop=y, k=1L)]
fact = tt[gt, on="k", allow.cartesian=TRUE,
          .(state_id=i.state_id, date_id=x.date_id, population=x.pop * i.p)]
setkeyv(fact, c("state_id", "date_id"))

### populate dimensions: time and geography

time = data.table(key="date_id",
  date_id= seq_along(tt$date), date=tt$date,
  month_id=month(tt$date), month=month.name[month(tt$date)],
  year_id=year(tt$date)-1789L, year=as.character(year(tt$date)),
  week_id=week(tt$date), week=as.character(week(tt$date)),
  weekday_id=wday(tt$date)-1L, weekday=weekdays(tt$date)
)
time[weekday_id == 0L, weekday_id := 7L][]
geog = data.table(key="state_id",
  state_id=seq_along(state.name), state_abb=state.abb, state_name=state.name,
  division_id=as.integer(state.division),
  division_name=as.character(state.division),
  region_id=as.integer(state.region),
  region_name=as.character(state.region)
)
rm(gt, tt)

### denormalize 'star schema'

l = list(fact, time, geog)
str(l)
mergelist(l)

rm(l)

### turn 'star schema' into 'snowflake schema'

make.lvl = function(x, cols) {
  stopifnot(is.data.table(x))
  lvl = x[, unique(.SD), .SDcols=cols]
  setkeyv(lvl, cols[1L])
  setindexv(lvl, as.list(cols))
  lvl
}
time = list(
  date = make.lvl(
    time, c("date_id", "date", "year_id", "month_id", "week_id", "weekday_id")),
  weekday = make.lvl(time, c("weekday_id", "weekday")),
  week = make.lvl(time, c("week_id", "week")),
  month = make.lvl(time, c("month_id", "month")),
  year = make.lvl(time, c("year_id", "year"))
)
geog = list(
  state = make.lvl(geog, c("state_id", "state_abb", "state_name", "division_id")),
  division = make.lvl(geog, c("division_id", "division_name", "region_id")),
  region = make.lvl(geog, c("region_id", "region_name"))
```

```
)

### denormalize 'snowflake schema'

#### left join all
l = c(list(fact=fact), time, geog)
str(l)
mergelist(l)

rm(l)
#### merge hierarchies alone, reduce sizes in merges of geog dimension
ans = mergelist(list(
  fact,
  mergelist(time),
  mergelist(rev(geog), how="right")
))

rm(ans)
#### same but no unnecessary copies
ans = mergelist(list(
  fact,
  setmergelist(time),
  setmergelist(rev(geog), how="right")
))
```

`na.omit.data.table` *Remove rows with missing values on columns specified*

Description

This is a `data.table` method for the S3 generic `stats::na.omit`. The internals are written in C for speed. See examples for benchmark timings.

`bit64::integer64` type is also supported.

Usage

```
## S3 method for class 'data.table'
na.omit(object, cols=seq_along(object), invert=FALSE, ...)
```

Arguments

`object`	A `data.table`.
`cols`	A vector of column names (or numbers) on which to check for missing values. Default is all the columns.
`invert`	logical. If `FALSE` omits all rows with any missing values (default). `TRUE` returns just those rows with missing values instead.
`...`	Further arguments special methods could require.

Details

The `data.table` method consists of an additional argument `cols`, which when specified looks for missing values in just those columns specified. The default value for `cols` is all the columns, to be consistent with the default behaviour of `stats::na.omit`.

It does not add the attribute `na.action` as `stats::na.omit` does.

Value

A data.table with just the rows where the specified columns have no missing value in any of them.

See Also

`data.table`

Examples

```
DT = data.table(x=c(1,NaN,NA,3), y=c(NA_integer_, 1:3), z=c("a", NA_character_, "b", "c"))
# default behaviour
na.omit(DT)
# omit rows where 'x' has a missing value
na.omit(DT, cols="x")
# omit rows where either 'x' or 'y' have missing values
na.omit(DT, cols=c("x", "y"))

## Not run:
# Timings on relatively large data
set.seed(1L)
DT = data.table(x = sample(c(1:100, NA_integer_), 5e7L, TRUE),
                y = sample(c(rnorm(100), NA), 5e7L, TRUE))
system.time(ans1 <- na.omit(DT)) ## 2.6 seconds
system.time(ans2 <- stats:::na.omit.data.frame(DT)) ## 29 seconds
# identical? check each column separately, as ans2 will have additional attribute
all(sapply(1:2, function(i) identical(ans1[[i]], ans2[[i]]))) ## TRUE

## End(Not run)
```

`nafill` *Fill missing values*

Description

Fast fill missing values using constant value, *last observation carried forward* or *next observation carried backward*.

Usage

```
nafill(x, type=c("const", "locf", "nocb"), fill=NA, nan=NA)
setnafill(x, type=c("const", "locf", "nocb"), fill=NA, nan=NA, cols=seq_along(x))
```

Arguments

x	Vector, list, data.frame or data.table of numeric columns.
type	Character, one of *"const"*, *"locf"* or *"nocb"*. Defaults to "const".
fill	Numeric value to be used to replace missing observations. See examples.
nan	Either NaN or NA; if the former, NaN is treated as distinct from NA, otherwise, they are treated the same during replacement. See Examples.
cols	Numeric or character vector specifying columns to be updated.

Details

Only *double* and *integer* data types are currently supported.

Note that both nafill and setnafill provide some verbose output when getOption('datatable.verbose') is TRUE.

Value

A list except when the input is a vector in which case a vector is returned. For setnafill the input argument is returned, updated by reference.

See Also

shift, data.table, fcoalesce

Examples

```
x = 1:10
x[c(1:2, 5:6, 9:10)] = NA
nafill(x, "locf")

x = c(1, NA, NaN, 3, NaN, NA, 4)
nafill(x, "locf")
nafill(x, "locf", nan=NaN)

# fill= applies to any leftover NA
nafill(c(NA, x), "locf")
nafill(c(NA, x), "locf", fill=0)

dt = data.table(v1=x, v2=shift(x)/2, v3=shift(x, -1L)/2)
nafill(dt, "nocb")

setnafill(dt, "locf", cols=c("v2","v3"))
dt
```

notin *Convenience operator for checking if an example is not in a set of elements*

Description

Check whether an object is absent from a table, i.e., the logical inverse of `in`. See examples on how missing values are being handled.

Usage

```
x %notin% table
```

Arguments

x	Vector or NULL: the values to be matched.
table	Vector or NULL: the values to be matched against.

Value

Logical vector, TRUE for each element of x *absent* from table, and FALSE for each element of x *present* in table.

See Also

`match`, `chmatch`

Examples

```
11 %notin% 1:10 # TRUE
"a" %notin% c("a", "b") # FALSE

## NAs on the LHS
NA %in% 1:2
NA %notin% 1:2
## NAs on the RHS
NA %in% c(1:2,NA)
NA %notin% c(1:2,NA)
```

patterns *Obtain matching indices corresponding to patterns*

Description

patterns returns the elements of cols that match the regular expression patterns, which must be supported by grep.

From v1.9.6, melt.data.table has an enhanced functionality in which measure.vars argument can accept a *list of column names* and melt them into separate columns. See the Efficient reshaping using data.tables vignette linked below to learn more.

Usage

```
patterns(
  ..., cols=character(0),
  ignore.case=FALSE, perl=FALSE,
  fixed=FALSE, useBytes=FALSE)
```

Arguments

...	A set of regular expression patterns.
cols	A character vector of names to which each pattern is matched.
ignore.case, perl, fixed, useBytes	Passed to grep.

See Also

melt, https://github.com/Rdatatable/data.table/wiki/Getting-started

Examples

```
DT = data.table(x1 = 1:5, x2 = 6:10, y1 = letters[1:5], y2 = letters[6:10])
# melt all columns that begin with 'x' & 'y', respectively, into separate columns
melt(DT, measure.vars = patterns("^x", "^y", cols=names(DT)))
# when used with melt, 'cols' is implicitly assumed to be names of input
# data.table, if not provided.
melt(DT, measure.vars = patterns("^x", "^y"))
```

print.data.table *data.table Printing Options*

Description

print.data.table extends the functionalities of print.data.frame.

Key enhancements include automatic output compression of many observations and concise column-wise class summary.

format_col and format_list_item generics provide flexibility for end-users to define custom printing methods for generic classes.

Note also the option datatable.prettyprint.char; character columns entries exceeding this limit will be truncated, with ... indicating the truncation. Note that the truncation is done with strtrim; be cognizant of potential limitations when dealing with non-printable characters like new-lines or tabs.

Usage

```
  ## S3 method for class 'data.table'
print(x,
    topn=getOption("datatable.print.topn"),             # default: 5
    nrows=getOption("datatable.print.nrows"),           # default: 100
    class=getOption("datatable.print.class"),           # default: TRUE
    row.names=getOption("datatable.print.rownames"),    # default: TRUE
    col.names=getOption("datatable.print.colnames"),    # default: "auto"
    print.keys=getOption("datatable.print.keys"),       # default: TRUE
    trunc.cols=getOption("datatable.print.trunc.cols"), # default: FALSE
    show.indices=getOption("datatable.show.indices"),   # default: FALSE
    quote=FALSE,
    na.print=NULL,
    timezone=FALSE, ...)

  format_col(x, ...)
  ## Default S3 method:
format_col(x, ...)
  ## S3 method for class 'POSIXct'
format_col(x, ..., timezone=FALSE)
  ## S3 method for class 'expression'
format_col(x, ...)

  format_list_item(x, ...)
  ## Default S3 method:
format_list_item(x, ...)
```

Arguments

x	A data.table.

topn	The number of rows to be printed from the beginning and end of tables with more than nrows rows.
nrows	The number of rows which will be printed before truncation is enforced.
class	If TRUE, the resulting output will include above each column its storage class (or a self-evident abbreviation thereof). When combined with col.names="auto" and tables >20 rows, classes will also appear at the bottom.
row.names	If TRUE, row indices will be printed alongside x.
col.names	One of three flavours for controlling the display of column names in output. "auto" includes column names above the data, as well as below the table if nrow(x) > 20 (when class=TRUE, column classes will also appear at the bottom). "top" excludes this lower register when applicable, and "none" suppresses column names altogether (as well as column classes if class = TRUE.
print.keys	If TRUE, any key and/or index currently assigned to x will be printed prior to the preview of the data.
trunc.cols	If TRUE, only the columns that can be printed in the console without wrapping the columns to new lines will be printed (similar to tibbles).
show.indices	If TRUE, indices will be printed as columns alongside x.
quote	If TRUE, all output will appear in quotes, as in print.default.
timezone	If TRUE, time columns of class POSIXct or POSIXlt will be printed with their timezones (if attribute is available).
na.print	The string to be printed in place of NA values, as in print.default.
...	Other arguments ultimately passed to format.

Details

By default, with an eye to the typically large number of observations in a data.table, only the beginning and end of the object are displayed (specifically, head(x, topn) and tail(x, topn) are displayed unless nrow(x) < nrows, in which case all rows will print).

format_col is applied at a column level; for example, format_col.POSIXct is used to tag the time zones of POSIXct columns. format_list_item is applied to the elements (rows) of list columns; see Examples. The default format_col method uses getS3method to test if a format method exists for the column, and if so uses it. Otherwise, the default format_list_item method uses the S3 format method (if one exists) for each item of a list column.

Value

print.data.table returns x invisibly.

format_col returns a length(x)-size character vector.

format_list_item returns a length-1 character scalar.

See Also

print.default

Examples

```
#output compression
DT <- data.table(a = 1:1000)
print(DT, nrows = 100, topn = 4)

#`quote` can be used to identify whitespace
DT <- data.table(blanks = c(" 12", " 34"),
                 noblanks = c("12", "34"))
print(DT, quote = TRUE)

#`class` provides handy column type summaries at a glance
DT <- data.table(a = vector("integer", 3),
                 b = vector("complex", 3),
                 c = as.IDate(paste0("2016-02-0", 1:3)))
print(DT, class = TRUE)

#`row.names` can be eliminated to save space
DT <- data.table(a = 1:3)
print(DT, row.names = FALSE)

#`print.keys` can alert which columns are currently keys
DT <- data.table(a=1:3, b=4:6, c=7:9, key=c("b", "a"))
setindexv(DT, c("a", "b"))
setindexv(DT, "a")
print(DT, print.keys=TRUE)

# `trunc.cols` will make it so only columns that fit in console will be printed
#    with a message that states the variables not shown
old_width = options("width" = 40)
DT <- data.table(thing_11 = vector("integer", 3),
                 thing_21 = vector("complex", 3),
                 thing_31 = as.IDate(paste0("2016-02-0", 1:3)),
                 thing_41 = "aasdfasdfasdfasdfasdfasdfasdfasdfasdfasdf",
                 thing_51 = vector("integer", 3),
                 thing_61 = vector("complex", 3))
print(DT, trunc.cols=TRUE)
options(old_width)

# `char.trunc` will truncate the strings,
# if their lengths exceed the given limit: `datatable.prettyprint.char`
# For example:

old = options(datatable.prettyprint.char=5L)
DT = data.table(x=1:2, y=c("abcdefghij", "klmnopqrstuv"))
DT
options(old)

# Formatting customization
format_col.complex = function(x, ...) sprintf('(%.1f, %.1fi)', Re(x), Im(x))
x = data.table(z = c(1 + 3i, 2 - 1i, pi + 2.718i))
print(x)
```

```
old = options(datatable.show.indices=TRUE)
NN = 200
set.seed(2024)
DT = data.table(
  grp1 = sample(100, NN, TRUE),
  grp2 = sample(90, NN, TRUE),
  grp3 = sample(80, NN, TRUE)
)
setkey(DT, grp1, grp2)
setindex(DT, grp1, grp3)
print(DT)
options(old)

iris = as.data.table(iris)
iris_agg = iris[ , .(reg = list(lm(Sepal.Length ~ Petal.Length))), by = Species]
format_list_item.lm = function(x, ...) sprintf('<lm:%s>', format(x$call$formula))
print(iris_agg)
```

`rbindlist` *Makes one data.table from a list of many*

Description

Same as `do.call(rbind, l)` on `data.frames`, but much faster.

Usage

```
rbindlist(l, use.names="check", fill=FALSE, idcol=NULL, ignore.attr=FALSE)
# rbind(..., use.names=TRUE, fill=FALSE, idcol=NULL)
```

Arguments

`l`	A list containing `data.table`, `data.frame` or `list` objects. `...` is the same but you pass the objects by name separately.
`use.names`	`TRUE` binds by matching column name, `FALSE` by position. `"check"` (default) warns if all items don't have the same names in the same order and then currently proceeds as if `use.names=FALSE` for backwards compatibility (`TRUE` in future); see news for v1.12.2.
`fill`	`TRUE` fills missing columns with NAs, or NULL for missing list columns. By default `FALSE`.
`idcol`	Creates a column in the result showing which list item those rows came from. `TRUE` names this column `".id"`. `idcol="file"` names this column `"file"`. If the input list has names, those names are the values placed in this id column, otherwise the values are an integer vector `1:length(l)`. See `examples`.
`ignore.attr`	Logical, default `FALSE`. When `TRUE`, allows binding columns with different attributes (e.g. class).

Details

Each item of l can be a data.table, data.frame or list, including NULL (skipped) or an empty object (0 rows). rbindlist is most useful when there are an unknown number of (potentially many) objects to stack, such as returned by lapply(fileNames, fread). rbind is most useful to stack two or three objects which you know in advance. ... should contain at least one data.table for rbind(...) to call the fast method and return a data.table, whereas rbindlist(l) always returns a data.table even when stacking a plain list with a data.frame, for example.

Columns with duplicate names are bound in the order of occurrence, similar to base. The position (column number) that each duplicate name occurs is also retained.

If column i does not have the same type in each of the list items; e.g, the column is integer in item 1 while others are numeric, they are coerced to the highest type.

If a column contains factors then a factor is created. If any of the factors are also ordered factors then the longest set of ordered levels are found (the first if this is tied). Then the ordered levels from each list item are checked to be an ordered subset of these longest levels. If any ambiguities are found (e.g. blue<green vs green<blue), or any ordered levels are missing from the longest, then a regular factor is created with warning. Any strings in regular factor and character columns which are missing from the longest ordered levels are added at the end.

When binding lists of data.table or data.frame objects containing objects with units defined by class attributes (e.g., difftime objects with different units), the resulting data.table may not preserve the original units correctly. Instead, values will be converted to a common unit without proper conversion of the values themselves. This issue applies to any class where the unit or precision is determined by attributes. Users should manually ensure that objects with unit-dependent attributes have consistent units before using rbindlist.

Value

An unkeyed data.table containing a concatenation of all the items passed in.

See Also

data.table, split.data.table

Examples

```
# default case
DT1 = data.table(A=1:3,B=letters[1:3])
DT2 = data.table(A=4:5,B=letters[4:5])
l = list(DT1,DT2)
rbindlist(l)

# bind correctly by names
DT1 = data.table(A=1:3,B=letters[1:3])
DT2 = data.table(B=letters[4:5],A=4:5)
l = list(DT1,DT2)
rbindlist(l, use.names=TRUE)

# fill missing columns, and match by col names
DT1 = data.table(A=1:3,B=letters[1:3])
DT2 = data.table(B=letters[4:5],C=factor(1:2))
```

```
l = list(DT1,DT2)
rbindlist(l, use.names=TRUE, fill=TRUE)

# generate index column, auto generates indices
rbindlist(l, use.names=TRUE, fill=TRUE, idcol=TRUE)
# let's name the list
setattr(l, 'names', c("a", "b"))
rbindlist(l, use.names=TRUE, fill=TRUE, idcol="ID")

# bind different classes
DT1 = data.table(A=1:3,B=letters[1:3])
DT2 = data.table(A=4:5,B=letters[4:5])
setattr(DT1[["A"]], "class", c("a", "integer"))
rbind(DT1, DT2, ignore.attr=TRUE)
```

`rleid` *Generate run-length type group id*

Description

A convenience function for generating a *run-length* type *id* column to be used in grouping operations. It accepts atomic vectors, lists, data.frames or data.tables as input.

Usage

```
rleid(..., prefix=NULL)
rleidv(x, cols=seq_along(x), prefix=NULL)
```

Arguments

`x`	A vector, list, data.frame or data.table.
`...`	A sequence of numeric, integer64, character or logical vectors, all of same length. For interactive use.
`cols`	Only meaningful for lists, data.frames or data.tables. A character vector of column names (or numbers) of x.
`prefix`	Either `NULL` (default) or a character vector of length=1 which is prefixed to the row ids, returning a character vector (instead of an integer vector).

Details

At times aggregation (or grouping) operations need to be performed where consecutive runs of identical values should belong to the same group (See `rle`). The use for such a function has come up repeatedly on StackOverflow, see the `See Also` section. This function allows to generate "run-length" groups directly.

`rleid` is designed for interactive use and accepts a sequence of vectors as arguments. For programming, `rleidv` might be more useful.

Value

When prefix = NULL, an integer vector with same length as NROW(x), else a character vector with the value in prefix prefixed to the ids obtained.

See Also

data.table, rowid, https://stackoverflow.com/q/21421047/559784

Examples

```
DT = data.table(grp=rep(c("A", "B", "C", "A", "B"), c(2,2,3,1,2)), value=1:10)
rleid(DT$grp) # get run-length ids
rleidv(DT, "grp") # same as above

rleid(DT$grp, prefix="grp") # prefix with 'grp'

# get sum of value over run-length groups
DT[, sum(value), by=.(grp, rleid(grp))]
DT[, sum(value), by=.(grp, rleid(grp, prefix="grp"))]
```

---

rowid *Generate unique row ids within each group*

Description

Convenience functions for generating a unique row ids within each group. It accepts atomic vectors, lists, data.frames or data.tables as input.

rowid is intended for interactive use, particularly along with the function dcast to generate unique ids directly in the formula.

rowidv(DT, cols=c("x", "y")) is equivalent to column N in the code DT[, N := seq_len(.N), by=c("x", "y")].

See examples for more.

Usage

```
rowid(..., prefix=NULL)
rowidv(x, cols=seq_along(x), prefix=NULL)
```

Arguments

x	A vector, list, data.frame or data.table.
...	A sequence of numeric, integer64, character or logical vectors, all of same length. For interactive use.
cols	Only meaningful for lists, data.frames or data.tables. A character vector of column names (or numbers) of x.
prefix	Either NULL (default) or a character vector of length=1 which is prefixed to the row ids, returning a character vector (instead of an integer vector).

Value

When `prefix = NULL`, an integer vector with same length as `NROW(x)`, else a character vector with the value in `prefix` prefixed to the ids obtained.

See Also

`dcast.data.table`, `rleid`

Examples

```
DT = data.table(x=c(20,10,10,30,30,20), y=c("a", "a", "a", "b", "b", "b"), z=1:6)

rowid(DT$x) # 1,1,2,1,2,2
rowidv(DT, cols="x") # same as above

rowid(DT$x, prefix="group") # prefixed with 'group'

rowid(DT$x, DT$y) # 1,1,2,1,2,1
rowidv(DT, cols=c("x","y")) # same as above
DT[, .(N=seq_len(.N)), by=.(x,y)]$N # same as above

# convenient usage with dcast
dcast(DT, x ~ rowid(x, prefix="group"), value.var="z")
#     x group1 group2
# 1: 10      2      3
# 2: 20      1      6
# 3: 30      4      5
```

---

rowwiseDT *Create a data.table row-wise*

Description

`rowwiseDT` creates a `data.table` object by specifying a row-by-row layout. This is convenient and highly readable for small tables.

Usage

```
rowwiseDT(...)
```

Arguments

`...`	Arguments that define the structure of a `data.table`. The column names come from named arguments (like `col=`), which must precede the data. See Examples.

Value

A `data.table`. The default is for each column to return as a vector. However, if any entry has a length that is not one (e.g., `list(1, 2)`), the whole column will be converted to a list column.

See Also

data.table

Examples

```
rowwiseDT(
  A=,B=, C=,
  1, "a",2:3,
  2, "b",list(5)
)
```

---

setattr *Set attributes of objects by reference*

Description

In data.table, all set* functions change their input *by reference*. That is, no copy is made at all, other than temporary working memory which is as large as one column. The only other data.table operator that modifies input by reference is :=. Check out the See Also section below for other set* function that data.table provides.

Usage

```
setattr(x, name, value)
setnames(x, old, new, skip_absent=FALSE)
```

Arguments

x	setnames accepts data.frame and data.table. setattr accepts any input; e.g, list, columns of a data.frame or data.table.
name	The character attribute name.
value	The value to assign to the attribute or NULL removes the attribute, if present.
old	When new is provided, character names or numeric positions of column names to change. When new is not provided, a function or the new column names (i.e., it's implicitly treated as new; excluding old and explicitly naming new is equivalent). If a function, it will be called with the current column names and is supposed to return the new column names. The new column names must be the same length as the number of columns. See examples.
new	Optional. It can be a function or the new column names. If a function, it will be called with old and expected to return the new column names. The new column names must be the same length as columns provided to old argument. Missing values in new mean to not rename that column, note: missing values are only allowed when old is not provided.
skip_absent	Skip items in old that are missing (i.e. absent) in names(x). Default FALSE halts with error if any are missing.

Details

setnames operates on data.table and data.frame not other types like list and vector. It can be used to change names *by name* with built-in checks and warnings (e.g., if any old names are missing or appear more than once).

setattr is a more general function that allows setting of any attribute to an object *by reference*.

A very welcome change in R 3.1+ was that names<- and colnames<- no longer copy the *entire* object as they used to (up to 4 times), see examples below. They now take a shallow copy. The 'set*' functions in data.table are still useful because they don't even take a shallow copy. This allows changing names and attributes of a (usually very large) data.table in the global environment *from within functions*. Like a database.

Value

The input is modified by reference, and returned (invisibly) so it can be used in compound statements; e.g., setnames(DT,"V1", "Y")[, .N, by=Y]. If you require a copy, take a copy first (using DT2=copy(DT)). See ?copy.

Note that setattr is also in package bit. Both packages merely expose R's internal setAttrib function at C level but differ in return value. bit::setattr returns NULL (invisibly) to remind you the function is used for its side effect. data.table::setattr returns the changed object (invisibly) for use in compound statements.

See Also

data.table, setkey, setorder, setcolorder, set, :=, setDT, setDF, copy

Examples

```
DT <- data.table(a = 1, b = 2, d = 3)

old <- c("a", "b", "c", "d")
new <- c("A", "B", "C", "D")

setnames(DT, old, new, skip_absent = TRUE) # skips old[3] because "c" is not a column name of DT

DF = data.frame(a=1:2,b=3:4)       # base data.frame to demo copies and syntax
if (capabilities()["profmem"])    # usually memory profiling is available but just in case
  tracemem(DF)
colnames(DF)[1] <- "A"            # 4 shallow copies (R >= 3.1, was 4 deep copies before)
names(DF)[1] <- "A"                # 3 shallow copies
names(DF) <- c("A", "b")           # 1 shallow copy
`names<-`(DF,c("A","b"))           # 1 shallow copy

DT = data.table(a=1:2,b=3:4,c=5:6) # compare to data.table
if (capabilities()["profmem"])
  tracemem(DT)                     # by reference, no deep or shallow copies
setnames(DT,"b","B")              # by name, no match() needed (warning if "b" is missing)
setnames(DT,3,"C")                 # by position with warning if 3 > ncol(DT)
setnames(DT,2:3,c("D","E"))        # multiple
setnames(DT,c("a","E"),c("A","F")) # multiple by name (warning if either "a" or "E" is missing)
setnames(DT,c("X","Y","Z"))        # replace all (length of names must be == ncol(DT))
```

```
setnames(DT,tolower)               # replace all names with their lower case
setnames(DT,2:3,toupper)           # replace the 2nd and 3rd names with their upper case

DT <- data.table(x = 1:3, y = 4:6, z = 7:9)
setnames(DT, -2, c("a", "b"))      # NEW FR #1443, allows -ve indices in 'old' argument

DT = data.table(a=1:3, b=4:6)
f = function(...) {
    # ...
    setattr(DT,"myFlag",TRUE)  # by reference
    # ...
    localDT = copy(DT)
    setattr(localDT,"myFlag2",TRUE)
    # ...
    invisible()
}
f()
attr(DT,"myFlag")   # TRUE
attr(DT,"myFlag2")  # NULL
```

`setcolorder` *Fast column reordering of a data.table by reference*

Description

In `data.table` parlance, all `set*` functions change their input *by reference*. That is, no copy is made at all, other than temporary working memory, which is as large as one column. The only other `data.table` operator that modifies input by reference is `:=`. Check out the `See Also` section below for other `set*` function `data.table` provides.

`setcolorder` reorders the columns of data.table, *by reference*, to the new order provided.

Usage

```
setcolorder(x, neworder=key(x), before=NULL, after=NULL, skip_absent=FALSE)
```

Arguments

`x`	A `data.table`.
`neworder`	Character vector of the new column name ordering. May also be column numbers. If `length(neworder)` < `length(x)`, the specified columns are moved in order to the "front" of `x`. By default, `setcolorder` without a specified `neworder` moves the key columns in order to the "front" of `x`.
`before, after`	If one of them (not both) was provided with a column name or number, `neworder` will be inserted before or after that column.
`skip_absent`	Logical, default `FALSE`. If `neworder` includes columns not present in `x`, `TRUE` will silently ignore them, whereas `FALSE` will throw an error.

Details

To reorder data.table columns, the idiomatic way is to use setcolorder(x, neworder), instead of doing x <- x[, ..neworder] (or x <- x[, neworder, with=FALSE]). This is because the latter makes an entire copy of the data.table, which maybe unnecessary in most situations. setcolorder also allows column numbers instead of names for neworder argument, although we recommend using names as a good programming practice.

Value

The input is modified by reference, and returned (invisibly) so it can be used in compound statements. If you require a copy, take a copy first (using DT2 = copy(DT)). See ?copy.

See Also

setkey, setorder, setattr, setnames, set, :=, setDT, setDF, copy, getNumericRounding, setNumericRounding

Examples

```
set.seed(45L)
DT = data.table(A=sample(3, 10, TRUE),
         B=sample(letters[1:3], 10, TRUE), C=sample(10))

setcolorder(DT, c("C", "A", "B"))

#incomplete specification
setcolorder(DT, "A")

# insert new column as first column
set(DT, j="D", value=sample(10))
setcolorder(DT, "D", before=1)

# move column to last column place
setcolorder(DT, "A", after=ncol(DT))
```

---

setDF *Coerce a data.table to data.frame by reference*

Description

In data.table parlance, all set* functions change their input *by reference*. That is, no copy is made at all, other than temporary working memory, which is as large as one column. The only other data.table operator that modifies input by reference is :=. Check out the See Also section below for other set* function data.table provides.

A helper function to convert a data.table or list of equal length to data.frame by reference.

Usage

```
setDF(x, rownames=NULL)
```

Arguments

x	A data.table, data.frame or list of equal length.
rownames	A character vector to assign as the row names of x.

Details

All data.table attributes including any keys and indices of the input data.table are stripped off.

When using rownames, recall that the row names of a data.frame must be unique. By default, the assigned set of row names is simply the sequence 1, ..., nrow(x) (or length(x) for lists).

Value

The input data.table is modified by reference to a data.frame and returned (invisibly). If you require a copy, take a copy first (using DT2 = copy(DT)). See ?copy.

See Also

data.table, as.data.table, setDT, copy, setkey, setcolorder, setattr, setnames, set, :=, setorder

Examples

```
X = data.table(x=1:5, y=6:10)
## convert 'X' to data.frame, without any copy.
setDF(X)

X = data.table(x=1:5, y=6:10)
## idem, assigning row names
setDF(X, rownames = LETTERS[1:5])

X = list(x=1:5, y=6:10)
# X is converted to a data.frame without any copy.
setDF(X)
```

---

setDT *Coerce lists and data.frames to data.table by reference*

Description

In data.table parlance, all set* functions change their input *by reference*. That is, no copy is made at all, other than temporary working memory, which is as large as one column. The only other data.table operator that modifies input by reference is :=. Check out the See Also section below for other set* function data.table provides.

setDT converts lists (both named and unnamed) and data.frames to data.tables *by reference*. This feature was requested on Stackoverflow.

Usage

```
setDT(x, keep.rownames=FALSE, key=NULL, check.names=FALSE)
```

Arguments

x	A named or unnamed list, data.frame or data.table.
keep.rownames	For data.frames, TRUE retains the data.frame's row names under a new column rn. keep.rownames = "id" names the column "id" instead.
key	Character vector of one or more column names which is passed to setkeyv.
check.names	Just as check.names in data.frame.

Details

When working on large lists or data.frames, it might be both time- and memory-consuming to convert them to a data.table using as.data.table(.), which will make a complete copy of the input object before converting it to a data.table. setDT takes care of this issue by converting any list (named or unnamed, data.frame or not) *by reference* instead. That is, the input object is modified in place with no copy.

This should come with low overhead, but note that setDT does check that the input is valid by looking for inconsistent input lengths and inadmissible column types (e.g. matrix).

Value

The input is modified by reference, and returned (invisibly) so it can be used in compound statements; e.g., setDT(X)[, sum(B), by=A]. If you require a copy, take a copy first (using DT2 = copy(DT)). See ?copy.

See Also

data.table, as.data.table, setDF, copy, setkey, setcolorder, setattr, setnames, set, :=, setorder, See the FAQ vignette: vignette("datatable-faq", package = "data.table").

Examples

```
set.seed(45L)
X = data.frame(
  A=sample(3, 10, TRUE),
  B=sample(letters[1:3], 10, TRUE),
  C=sample(10))

# Convert X to data.table by reference and
# get the frequency of each "A,B" combination
setDT(X)[, .N, by=.(A,B)]

# convert list to data.table
# autofill names
X = list(1:4, letters[1:4])
setDT(X)
# don't provide names
```

```
X = list(a=1:4, letters[1:4])
setDT(X, FALSE)

# setkey directly
X = list(a = 4:1, b=runif(4))
setDT(X, key="a")[]

# check.names argument
X = list(a=1:5, a=6:10)
setDT(X, check.names=TRUE)[]

# Example demonstrating setDT after loading from RDS
rds_file = tempfile(fileext = ".rds")
X = data.table(a = 1:5, b = letters[1:5])
saveRDS(X, rds_file)
X_loaded = readRDS(rds_file)
setDT(X_loaded)  # restore internal data.table attributes
print(X_loaded)
unlink(rds_file)
```

`setDTthreads` *Set or get number of threads that data.table should use*

Description

Set and get number of threads to be used in `data.table` functions that are parallelized with OpenMP. The number of threads is initialized when `data.table` is first loaded in the R session using optional environment variables. Thereafter, the number of threads may be changed by calling `setDTthreads`. If you change an environment variable using `Sys.setenv` you will need to call `setDTthreads` again to reread the environment variables.

Usage

```
setDTthreads(threads = NULL, restore_after_fork = NULL, percent = NULL, throttle = NULL)
 getDTthreads(verbose = getOption("datatable.verbose"))
```

Arguments

`threads`	NULL (default) rereads environment variables. 0 means to use all logical CPUs available. Otherwise a number >= 1
`restore_after_fork`	Should data.table be multi-threaded after a fork has completed? NULL leaves the current setting unchanged which by default is TRUE. See details below.
`percent`	If provided it should be a number between 2 and 100; the percentage of logical CPUs to use. By default on startup, 50%.
`throttle`	1024 (default) means that, roughly speaking, a single thread will be used when `nrow(DT)<=1024`, 2 threads when `nrow(DT)<=2048`, etc. The throttle is to speed up small data tasks (especially when repeated many times) by not incurring the

overhead of managing multiple threads. Hence the number of threads is throttled (restricted) for small tasks.

`verbose` Display the value of relevant OpenMP settings plus the `restore_after_fork` internal option.

Details

`data.table` automatically switches to single threaded mode upon fork (the mechanism used by `parallel::mclapply` and the foreach package). Otherwise, nested parallelism would very likely overload your CPUs and result in much slower execution. As `data.table` becomes more parallel internally, we expect explicit user parallelism to be needed less often. The `restore_after_fork` option controls what happens after the explicit fork parallelism completes. It needs to be at C level so it is not a regular R option using `options()`. By default `data.table` will be multi-threaded again; restoring the prior setting of `getDTthreads()`. But problems have been reported in the past on Mac with Intel OpenMP libraries whereas success has been reported on Linux. If you experience problems after fork, start a new R session and change the default behaviour by calling `setDTthreads(restore_after_fork=FALSE)` before retrying. Please raise issues on the data.table GitHub issues page.

The number of logical CPUs is determined by the OpenMP function `omp_get_num_procs()` whose meaning may vary across platforms and OpenMP implementations. `setDTthreads()` will not allow more than this limit. Neither will it allow more than `omp_get_thread_limit()` nor the current value of `Sys.getenv("OMP_THREAD_LIMIT")`. Note that CRAN's daily test system (results for data.table here) sets `OMP_THREAD_LIMIT` to 2 and should always be respected; e.g., if you have written a package that uses data.table and your package is to be released on CRAN, you should not change `OMP_THREAD_LIMIT` in your package to a value greater than 2.

Some hardware allows CPUs to be removed and/or replaced while the server is running. If this happens, our understanding is that `omp_get_num_procs()` will reflect the new number of processors available. But if this happens after data.table started, `setDTthreads(...)` will need to be called again by you before data.table will reflect the change. If you have such hardware, please let us know your experience via GitHub issues / feature requests.

Use `getDTthreads(verbose=TRUE)` to see the relevant environment variables, their values and the current number of threads data.table is using. For example, the environment variable `R_DATATABLE_NUM_PROCS_PERCENT` can be used to change the default number of logical CPUs from 50% to another value between 2 and 100. If you change these environment variables using `Sys.setenv()` after data.table and/or OpenMP has initialized then you will need to call `setDTthreads(threads=NULL)` to reread their current values. `getDTthreads()` merely retrieves the internal value that was set by the last call to `setDTthreads()`. `setDTthreads(threads=NULL)` is called when data.table is first loaded and is not called again unless you call it.

`setDTthreads()` affects `data.table` only and does not change R itself or other packages using OpenMP. We have followed the advice of section 1.2.1.1 in the R-exts manual: "...or, better, for the regions in your code as part of their specification...num_threads(nthreads)...That way you only control your own code and not that of other OpenMP users." Every parallel region in data.table contain a `num_threads(getDTthreads())` directive. This is mandated by a `grep` in data.table's quality control script.

`setDTthreads(0)` is the same as `setDTthreads(percent=100)`; i.e. use all logical CPUs, subject to `Sys.getenv("OMP_THREAD_LIMIT")`. Please note again that CRAN's daily test system sets

OMP_THREAD_LIMIT to 2, so developers of CRAN packages should never change OMP_THREAD_LIMIT inside their package to a value greater than 2.

Internally parallelized code is used in the following places:

- 'between.c' - between()
- 'cj.c' - CJ()
- 'coalesce.c' - fcoalesce()
- 'fifelse.c' - fifelse()
- 'fread.c', 'freadR.c' - fread(). Parallelized across row-based chunks of the file.
- 'forder.c', 'fsort.c', and 'reorder.c' - forder() and related
- 'froll.c', 'frolladaptive.c', and 'frollR.c' - froll() and family
- 'fwrite.c' - fwrite(). Parallelized across rows.
- 'gsumm.c' - GForce in various places, see GForce. Parallelized across groups.
- 'nafill.c' - nafill()
- 'subset.c' - Used in [.data.table subsetting
- 'types.c' - Internal testing usage

We endeavor to keep this list up to date, but note that the canonical reference here is the source code itself.

Value

A length 1 integer. The old value is returned by setDTthreads so you can store that prior value and pass it to setDTthreads() again after the section of your code where you control the number of threads.

Examples

```
getDTthreads(verbose=TRUE)
```

---

setkey *Create key on a data.table*

Description

setkey sorts a data.table and marks it as sorted with an attribute "sorted". The sorted columns are the key. The key can be any number of columns. The data is always sorted in *ascending* order with NAs (if any) always first. The table is changed *by reference* and there is no memory used for the key (other than marking which columns the data is sorted by).

There are three reasons setkey is desirable:

- binary search and joins are faster when they detect they can use an existing key
- grouping by a leading subset of the key columns is faster because the groups are already gathered contiguously in RAM

- simpler shorter syntax; e.g. DT["id",] finds the group "id" in the first column of DT's key using binary search. It may be helpful to think of a key as super-charged rownames: multi-column and multi-type.

NAs are always first because:

- NA is internally INT_MIN (a large negative number) in R. Keys and indexes are always in increasing order so if NAs are first, no special treatment or branch is needed in many data.table internals involving binary search. It is not optional to place NAs last for speed, simplicity and robustness of internals at C level.
- if any NAs are present then we believe it is better to display them up front (rather than hiding them at the end) to reduce the risk of not realizing NAs are present.

In data.table parlance, all set* functions change their input *by reference*. That is, no copy is made at all other than for temporary working memory, which is as large as one column. The only other data.table operator that modifies input by reference is :=. Check out the See Also section below for other set* functions data.table provides.

setindex creates an index for the provided columns. This index is simply an ordering vector of the dataset's rows according to the provided columns. This order vector is stored as an attribute of the data.table and the dataset retains the original order of rows in memory. See the vignette("datatable-secondary-indices-and-auto-indexing") for more details.

key returns the data.table's key if it exists; NULL if none exists.

haskey returns TRUE/FALSE if the data.table has a key.

Usage

```
setkey(x, ..., verbose=getOption("datatable.verbose"), physical = TRUE)
setkeyv(x, cols, verbose=getOption("datatable.verbose"), physical = TRUE)
setindex(...)
setindexv(x, cols, verbose=getOption("datatable.verbose"))
key(x)
indices(x, vectors = FALSE)
haskey(x)
```

Arguments

x	A data.table.
...	The columns to sort by. Do not quote the column names. If ... is missing (i.e. setkey(DT)), all the columns are used. NULL removes the key.
cols	A character vector of column names. For setindexv, this can be a list of character vectors, in which case each element will be applied as an index in turn.
verbose	Output status and information.
physical	TRUE changes the order of the data in RAM. FALSE adds an index.
vectors	logical scalar, default FALSE; when set to TRUE, a list of character vectors is returned, each referring to one index.

Details

setkey reorders (i.e. sorts) the rows of a data.table by the columns provided. The sort method used has developed over the years and we have contributed to base R too; see sort. Generally speaking we avoid any type of comparison sort (other than insert sort for very small input) preferring instead counting sort and forwards radix. We also avoid hash tables.

Note that setkey always uses "C-locale"; see the Details in the help for setorder for more on why.

The sort is *stable*; i.e., the order of ties (if any) is preserved.

For character vectors, data.table takes advantage of R's internal global string cache, also exported as chorder.

Value

The input is modified by reference and returned (invisibly) so it can be used in compound statements; e.g., setkey(DT,a)[.("foo")]. If you require a copy, take a copy first (using DT2=copy(DT)). copy may also sometimes be useful before := is used to subassign to a column by reference.

Keys vs. Indices

Setting a key (with setkey) and an index (with setindex) are similar, but have very important distinctions.

Setting a key physically reorders the data in RAM.

Setting an index computes the sort order, but instead of applying the reordering, simply *stores* this computed ordering. That means that multiple indices can coexist, and that the original row order is preserved.

Good practice

In general, it's good practice to use column names rather than numbers. This is why setkey and setkeyv only accept column names. If you use column numbers then bugs (possibly silent) can more easily creep into your code as time progresses if changes are made elsewhere in your code; e.g., if you add, remove or reorder columns in a few months time, a setkey by column number will then refer to a different column, possibly returning incorrect results with no warning. (A similar concept exists in SQL, where "select * from ..." is considered poor programming style when a robust, maintainable system is required.)

If you really wish to use column numbers, it is possible but deliberately a little harder; e.g., setkeyv(DT,names(DT)[1:2]).

If you want to subset rows based on values of an integer key column, it should be done with the dot (.) syntax, because integers are otherwise interpreted as row numbers (see example).

If you wanted to use grep to select key columns according to a pattern, note that you can just set value = TRUE to return a character vector instead of the default integer indices.

References

https://en.wikipedia.org/wiki/Radix_sort
https://en.wikipedia.org/wiki/Counting_sort
http://stereopsis.com/radix.html

https://codercorner.com/RadixSortRevisited.htm
https://cran.r-project.org/package=bit64
https://github.com/Rdatatable/data.table/wiki/Presentations

See Also

data.table, tables, J, sort.list, copy, setDT, setDF, set :=, setorder, setcolorder, setattr, setnames, chorder, setNumericRounding

Examples

```
# Type 'example(setkey)' to run these at the prompt and browse output

DT = data.table(A=5:1,B=letters[5:1])
DT # before
setkey(DT,B)          # re-orders table and marks it sorted.
DT # after
tables()              # KEY column reports the key'd columns
key(DT)
keycols = c("A","B")
setkeyv(DT,keycols)

DT = data.table(A=5:1,B=letters[5:1])
DT2 = DT              # does not copy
setkey(DT2,B)         # does not copy-on-write to DT2
identical(DT,DT2)     # TRUE. DT and DT2 are two names for the same keyed table

DT = data.table(A=5:1,B=letters[5:1])
DT2 = copy(DT)        # explicit copy() needed to copy a data.table
setkey(DT2,B)         # now just changes DT2
identical(DT,DT2)     # FALSE. DT and DT2 are now different tables

DT = data.table(A=5:1,B=letters[5:1])
setindex(DT)          # set indices
setindex(DT, A)
setindex(DT, B)
indices(DT)           # get indices single vector
indices(DT, vectors = TRUE) # get indices list

# Setting multiple indices at once
DT = data.table(A = 5:1, B = letters[5:1], C = 10:6)
setindexv(DT, list(c("A", "B"), c("B", "C")))
print(DT, show.indices=TRUE)

# Use the dot .(subset_value) syntax with integer keys:
DT = data.table(id = 2:1)
setkey(DT, id)
subset_value <- 1
DT[subset_value]  # treats subset_value as an row number
DT[.(subset_value)]  # matches subset_value against key column (id)
```

`setNumericRounding` *Change or turn off numeric rounding*

Description

Change rounding to 0, 1 or 2 bytes when joining, grouping or ordering numeric (i.e. double, POSIXct) columns.

Usage

```
setNumericRounding(x)
getNumericRounding()
```

Arguments

x	integer or numeric vector: 0 (default), 1 or 2 byte rounding

Details

Computers cannot represent some floating point numbers (such as 0.6) precisely, using base 2. This leads to unexpected behaviour when joining or grouping columns of type 'numeric'; i.e. 'double', see example below. In cases where this is undesirable, data.table allows rounding such data up to approximately 11 significant figures which is plenty of digits for many cases. This is achieved by rounding the last 2 bytes off the significand. Other possible values are 1 byte rounding, or no rounding (full precision, default).

It is bytes rather than bits because it is tied in with the radix sort algorithm for sorting numerics which sorts byte by byte. With the default rounding of 0 bytes, at most 8 passes are needed. With rounding of 2 bytes, at most 6 passes are needed (and therefore might be a tad faster).

For large numbers (integers > 2^31), we recommend using `bit64::integer64`, even though the default is to round off 0 bytes (full precision).

Value

`setNumericRounding` returns no value; the new value is applied. `getNumericRounding` returns the current value: 0, 1 or 2.

See Also

`datatable-optimize`
https://en.wikipedia.org/wiki/Double-precision_floating-point_format
https://en.wikipedia.org/wiki/Floating_point
https://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html

Examples

```
DT = data.table(a=seq(0,1,by=0.2),b=1:2, key="a")
DT
setNumericRounding(0)   # By default, rounding is turned off
DT[.(0.4)]   # works
DT[.(0.6)]   # no match, can be confusing since 0.6 is clearly there in DT
             # happens due to floating point representation limitations

setNumericRounding(2)   # round off last 2 bytes
DT[.(0.6)]   # works

# using type 'numeric' for integers > 2^31 (typically ids)
DT = data.table(id = c(1234567890123, 1234567890124, 1234567890125), val=1:3)
print(DT, digits=15)
DT[,.N,by=id]   # 1 row, (last 2 bytes rounded)
setNumericRounding(0)
DT[,.N,by=id]   # 3 rows, (no rounding, default)
# better to use bit64::integer64 for such ids
```

---

setops *Set operations for data tables*

Description

Similar to base R set functions, `union`, `intersect`, `setdiff` and `setequal` but for `data.tables`. Additional `all` argument controls how duplicated rows are handled. Functions `fintersect`, `setdiff` (`MINUS` or `EXCEPT` in SQL) and `funion` are meant to provide functionality of corresponding SQL operators. Unlike SQL, data.table functions will retain row order.

Usage

```
fintersect(x, y, all = FALSE)
fsetdiff(x, y, all = FALSE)
funion(x, y, all = FALSE)
fsetequal(x, y, all = TRUE)
```

Arguments

`x, y`	`data.tables`.
`all`	Logical. Default is `FALSE` and removes duplicate rows on the result. When `TRUE`, if there are `xn` copies of a particular row in `x` and `yn` copies of the same row in `y`, then: • `fintersect` will return `min(xn, yn)` copies of that row. • `fsetdiff` will return `max(0, xn-yn)` copies of that row. • `funion` will return `xn+yn` copies of that row. • `fsetequal` will return `FALSE` unless `xn == yn`.

Details

bit64::integer64 columns are supported but not complex and list, except for funion.

Value

A data.table in case of fintersect, funion and fsetdiff. Logical TRUE or FALSE for fsetequal.

References

https://db.apache.org/derby/papers/Intersect-design.html

See Also

data.table, rbindlist, all.equal.data.table, unique, duplicated, uniqueN, anyDuplicated

Examples

```
x = data.table(c(1,2,2,2,3,4,4))
x2 = data.table(c(1,2,3,4)) # same set of rows as x
y = data.table(c(2,3,4,4,4,5))
fintersect(x, y)            # intersect
fintersect(x, y, all=TRUE)  # intersect all
fsetdiff(x, y)              # except
fsetdiff(x, y, all=TRUE)    # except all
funion(x, y)                # union
funion(x, y, all=TRUE)      # union all
fsetequal(x, x2, all=FALSE) # setequal
fsetequal(x, x2)            # setequal all
```

---

setorder *Fast row reordering of a data.table by reference*

Description

In data.table parlance, all set* functions change their input *by reference*. That is, no copy is made at all, other than temporary working memory, which is as large as one column. The only other data.table operator that modifies input by reference is :=. Check out the See Also section below for other set* function data.table provides.

setorder (and setorderv) reorders the rows of a data.table based on the columns (and column order) provided. It reorders the table *by reference* and is therefore very memory efficient.

Note that queries like x[order(.)] are optimised internally to use data.table's fast order.

Also note that data.table always reorders in "C-locale" (see Details). To sort by session locale, use x[base::order(.)].

bit64::integer64 type is also supported for reordering rows of a data.table.

Usage

```
setorder(x, ..., na.last=FALSE)
setorderv(x, cols = colnames(x), order=1L, na.last=FALSE)
# optimised to use data.table's internal fast order
# x[order(., na.last=TRUE)]
# x[order(., decreasing=TRUE)]
# sort_by(x, ., na.last=TRUE, decreasing=FALSE)    # R >= 4.4.0
```

Arguments

x	A data.table.
...	The columns to sort by. Do not quote column names. If ... is missing (ex: setorder(x)), x is rearranged based on all columns in ascending order by default. To sort by a column in descending order prefix the symbol "-" which means "descending" (*not* "negative", in this context), i.e., setorder(x, a, -b, c). The -b works when b is of type character as well.
cols	A character vector of column names of x by which to order. By default, sorts over all columns; cols = NULL will return x untouched. Do not add "-" here. Use order argument instead.
order	An integer vector with only possible values of 1 and -1, corresponding to ascending and descending order. The length of order must be either 1 or equal to that of cols. If length(order) == 1, it is recycled to length(cols).
na.last	logical. If TRUE, missing values in the data are placed last; if FALSE, they are placed first; if NA they are removed. na.last=NA is valid only for x[order(., na.last)] and related sort_by(x, .) (R $\geq$ 4.4.0) and its default is TRUE. setorder and setorderv only accept TRUE/FALSE with default FALSE.

Details

data.table implements its own fast radix-based ordering. See the references for some exposition on the concept of radix sort.

setorder accepts unquoted column names (with names preceded with a - sign for descending order) and reorders data.table rows *by reference*, for e.g., setorder(x, a, -b, c). We emphasize that this means "descending" and not "negative" because the implementation simply reverses the sort order, as opposed to sorting the opposite of the input (which would be inefficient). Note that -b also works with columns of type character unlike order, which requires -xtfrm(y) instead (which is slow).

setorderv in turn accepts a character vector of column names and an integer vector of column order separately.

Note that setkey still requires and will always sort only in ascending order, and is different from setorder in that it additionally sets the sorted attribute.

na.last argument, by default, is FALSE for setorder and setorderv to be consistent with data.table's setkey and is TRUE for x[order(.)] and sort_by(x, .) (R $\geq$ 4.4.0) to be consistent with base::order. Only x[order(.)] (and related sort_by(x, .)) can have na.last = NA as it is a subset operation as opposed to setorder or setorderv which reorders the data.table by reference.

data.table always reorders in "C-locale". As a consequence, the ordering may be different to that obtained by base::order. In English locales, for example, sorting is case-sensitive in C-locale. Thus, sorting c("c", "a", "B") returns c("B", "a", "c") in data.table but c("a", "B", "c") in base::order. Note this makes no difference in most cases of data; both return identical results on ids where only upper-case or lower-case letters are present ("AB123" < "AC234" is true in both), or on country names and other proper nouns which are consistently capitalized. For example, neither "America" < "Brazil" nor "america" < "brazil" are affected since the first letter is consistently capitalized.

Using C-locale makes the behaviour of sorting in data.table more consistent across sessions and locales. The behaviour of base::order depends on assumptions about the locale of the R session. In English locales, "america" < "BRAZIL" is true by default but false if you either type Sys.setlocale(locale="C") or the R session has been started in a C locale for you – which can happen on servers/services since the locale comes from the environment the R session was started in. By contrast, "america" < "BRAZIL" is always FALSE in data.table regardless of the way your R session was started.

If setorder results in reordering of the rows of a keyed data.table, then its key will be set to NULL.

Starting from R 4.4.0, sort_by(x, y, ...) is the S3 method for the generic sort_by for data.table's. It uses the same formula or list interfaces as data.frame's sort_by but internally uses data.table's fast ordering, hence it behaves the same as x[order(.)] and takes the same optional named arguments and their defaults.

Value

The input is modified by reference, and returned (invisibly) so it can be used in compound statements; e.g., setorder(DT,a,-b)[, cumsum(c), by=list(a,b)]. If you require a copy, take a copy first (using DT2 = copy(DT)). See copy.

References

https://en.wikipedia.org/wiki/Radix_sort
https://en.wikipedia.org/wiki/Counting_sort
http://stereopsis.com/radix.html
https://codercorner.com/RadixSortRevisited.htm
https://medium.com/basecs/getting-to-the-root-of-sorting-with-radix-sort-f8e9240d4224

See Also

setkey, setcolorder, setattr, setnames, set, :=, setDT, setDF, copy, setNumericRounding

Examples

```
set.seed(45L)
DT = data.table(A=sample(3, 10, TRUE),
         B=sample(letters[1:3], 10, TRUE), C=sample(10))

# setorder
setorder(DT, A, -B)
```

```
# same as above, but using setorderv
setorderv(DT, c("A", "B"), c(1, -1))
```

---

shift *Fast lead/lag for vectors and lists*

Description

lead or lag vectors, lists, data.frames or data.tables implemented in C for speed.

bit64::integer64 is also supported.

Usage

```
shift(x, n=1L, fill, type=c("lag", "lead", "shift", "cyclic"), give.names=FALSE)
```

Arguments

x	A vector, list, data.frame or data.table.
n	integer vector denoting the offset by which to lead or lag the input. To create multiple lead/lag vectors, provide multiple values to n; negative values of n will "flip" the value of type, i.e., n=-1 and type='lead' is the same as n=1 and type='lag'.
fill	default is NA. Value to use for padding when the window goes beyond the input length.
type	default is "lag" (look "backwards"). The other possible values "lead" (look "forwards"), "shift" (behave same as "lag" except given names) and "cyclic" where pushed out values are re-introduced at the front/back.
give.names	default is FALSE which returns an unnamed list. When TRUE, names are automatically generated corresponding to type and n. If answer is an atomic vector, then the argument is ignored.

Details

shift accepts vectors, lists, data.frames or data.tables. It always returns a list except when the input is a vector and length(n) == 1 in which case a vector is returned, for convenience. This is so that it can be used conveniently within data.table's syntax. For example, DT[, (cols) := shift(.SD, 1L), by=id] would lag every column of .SD by 1 for each group and DT[, newcol := colA + shift(colB)] would assign the sum of two *vectors* to newcol.

Argument n allows multiple values. For example, DT[, (cols) := shift(.SD, 1:2), by=id] would lag every column of .SD by 1 and 2 for each group. If .SD contained four columns, the first two elements of the list would correspond to lag=1 and lag=2 for the first column of .SD, the next two for second column of .SD and so on. Please see examples for more.

shift is designed mainly for use in data.tables along with := or set. Therefore, it returns an unnamed list by default as assigning names for each group over and over can be quite time consuming

with many groups. It may be useful to set names automatically in other cases, which can be done by setting `give.names` to `TRUE`.

Note that when using `shift` with a list, it should be a list of lists rather than a flattened list. The function was not designed to handle flattened lists directly. This also applies to the use of list columns in a data.table. For example, `DT = data.table(x=as.list(1:4))` is a data.table with four rows. Applying `DT[, shift(x)]` now lags every entry individually, rather than shifting the full columns like `DT[, shift(as.integer(x))]` does. Using `DT = data.table(x=list(1:4))` creates a data.table with one row. Now `DT[, shift(x)]` returns a data.table with four rows where x is lagged. To get a shifted data.table with the same number of rows, wrap the `shift` function in `list` or `dot`, e.g., `DT[, .(shift(x))]`.

Value

A list containing the lead/lag of input `x`.

See Also

`data.table`

Examples

```
# on vectors, returns a vector as long as length(n) == 1, #1127
x = 1:5
# lag with n=1 and pad with NA (returns vector)
shift(x, n=1, fill=NA, type="lag")
# lag with n=1 and 2, and pad with 0 (returns list)
shift(x, n=1:2, fill=0, type="lag")
# getting a window by using positive and negative n:
shift(x, n = -1:1)
shift(x, n = -1:1, type = "shift", give.names = TRUE)
# cyclic shift where pad uses pushed out values
shift(x, n = -1:1, type = "cyclic")

# on data.tables
DT = data.table(year=2010:2014, v1=runif(5), v2=1:5, v3=letters[1:5])
# lag columns 'v1,v2,v3' DT by 1 and fill with 0
cols = c("v1","v2","v3")
anscols = paste("lead", cols, sep="_")
DT[, (anscols) := shift(.SD, 1, 0, "lead"), .SDcols=cols]

# return a new data.table instead of updating
# with names automatically set
DT = data.table(year=2010:2014, v1=runif(5), v2=1:5, v3=letters[1:5])
DT[, shift(.SD, 1:2, NA, "lead", TRUE), .SDcols=2:4]

# lag/lead in the right order
DT = data.table(year=2010:2014, v1=runif(5), v2=1:5, v3=letters[1:5])
DT = DT[sample(nrow(DT))]
# add lag=1 for columns 'v1,v2,v3' in increasing order of 'year'
cols = c("v1","v2","v3")
anscols = paste("lag", cols, sep="_")
```

```
DT[order(year), (cols) := shift(.SD, 1, type="lag"), .SDcols=cols]
DT[order(year)]

# while grouping
DT = data.table(year=rep(2010:2011, each=3), v1=1:6)
DT[, c("lag1", "lag2") := shift(.SD, 1:2), by=year]

# on lists
ll = list(1:3, letters[4:1], runif(2))
shift(ll, 1, type="lead")
shift(ll, 1, type="lead", give.names=TRUE)
shift(ll, 1:2, type="lead")

# fill using first or last by group
DT = data.table(x=1:6, g=rep(1:2, each=3))
DT[ , shift(x, fill=x[1L]), by=g]
DT[ , shift(x, fill=x[.N], type="lead"), by=g]
```

`shouldPrint` *For use by packages that mimic/divert auto printing e.g. IRkernel and knitr*

Description

Not for use by users. Exported only for use by IRkernel (Jupyter) and knitr.

Usage

```
shouldPrint(x)
```

Arguments

`x`	A `data.table`.

Details

Should IRkernel/Jupyter print a data.table returned invisibly by DT[,:=] ? This is a read-once function since it resets an internal flag. If you need the value more than once in your logic, store the value from the first call.

Value

TRUE or FALSE.

References

https://github.com/IRkernel/IRkernel/issues/127
https://github.com/Rdatatable/data.table/issues/933

Examples

```
# dummy example section to pass release check that all .Rd files have examples
```

---

special-symbols *Special symbols*

Description

.SD, .BY, .N, .I, .GRP, and .NGRP are *read-only* symbols for use in j. .N can be used in i as well. .I can be used in by as well. See the vignettes, Details and Examples here and in data.table. .EACHI is a symbol passed to by; i.e. by=.EACHI, .NATURAL is a symbol passed to on; i.e. on=.NATURAL

Details

The bindings of these variables are locked and attempting to assign to them will generate an error. If you wish to manipulate .SD before returning it, take a copy(.SD) first (see FAQ 4.5). Using := in the j of .SD is reserved for future use as a (tortuously) flexible way to update DT by reference by group (even when groups are not contiguous in an ad hoc by).

These symbols used in j are defined as follows.

- .SD is a data.table containing the **S**ubset of x's **D**ata for each group, excluding any columns used in by (or keyby).
- .BY is a list containing a length 1 vector for each item in by. This can be useful when by is not known in advance. The by variables are also available to j directly by name; useful for example for titles of graphs if j is a plot command, or to branch with if() depending on the value of a group variable.
- .N is an integer, length 1, containing the number of rows in the group. This may be useful when the column names are not known in advance and for convenience generally. When grouping by i, .N is the number of rows in x matched to, for each row of i, regardless of whether nomatch is NA or NULL. It is renamed to N (no dot) in the result (otherwise a column called ".N" could conflict with the .N variable, see FAQ 4.6 for more details and example), unless it is explicitly named; e.g., DT[,list(total=.N),by=a].
- .I is an integer vector equal to seq_len(nrow(x)). While grouping, it holds for each item in the group, its row location in x. This is useful to subset in j; e.g. DT[, .I[which.max(somecol)], by=grp]. If used in by it corresponds to applying a function rowwise.
- .GRP is an integer, length 1, containing a simple group counter. 1 for the 1st group, 2 for the 2nd, etc.
- .NGRP is an integer, length 1, containing the number of groups.

.EACHI is defined as NULL but its value is not used. Its usage is by=.EACHI (or keyby=.EACHI) which invokes grouping-by-each-row-of-i; see data.table's by argument for more details.

.NATURAL is defined as NULL but its value is not used. Its usage is on=.NATURAL (alternative of X[on=Y]) which joins two tables on their common column names, performing a natural join; see data.table's on argument for more details.

Note that `.N` in `i` is computed up-front, while that in `j` applies *after filtering in* `i`. That means that even absent grouping, `.N` in `i` can be different from `.N` in `j`. See Examples.

Note also that you should consider these symbols read-only and of limited scope – internal data.table code might manipulate them in unexpected ways, and as such their bindings are locked. There are subtle ways to wind up with the wrong object, especially when attempting to copy their values outside a grouping context. See examples; when in doubt, `copy()` is your friend.

See Also

`data.table`, `:=`, `set`, `datatable-optimize`

Examples

```
DT = data.table(x=rep(c("b","a","c"),each=3), v=c(1,1,1,2,2,1,1,2,2), y=c(1,3,6), a=1:9, b=9:1)
DT
X = data.table(x=c("c","b"), v=8:7, foo=c(4,2))
X

DT[.N]                                  # last row, only special symbol allowed in 'i'
DT[, .N]                                # total number of rows in DT
DT[, .N, by=x]                          # number of rows in each group
DT[, .SD, .SDcols=x:y]                  # select columns 'x' through 'y'
DT[, .SD[1]]                            # first row of all columns
DT[, .SD[1], by=x]                      # first row of all columns for each group in 'x'
DT[, c(.N, lapply(.SD, sum)), by=x]     # get rows *and* sum all columns by group
DT[, .I[1], by=x]                       # row number in DT corresponding to each group
DT[, .N, by=rleid(v)]                   # get count of consecutive runs of 'v'
DT[, c(.(y=max(y)), lapply(.SD, min)),
        by=rleid(v), .SDcols=v:b]       # compute 'j' for each consecutive runs of 'v'
DT[, grp := .GRP, by=x]                 # add a group counter
DT[, grp_pct := .GRP/.NGRP, by=x]       # add a group "progress" counter
X[, DT[.BY, y, on="x"], by=x]           # join within each group
DT[X, on=.NATURAL]                    # join X and DT on common column similar to X[on=Y]

# .N can be different in i and j
DT[{cat(sprintf('in i, .N is %d\n', .N)); a < .N/2},
   {cat(sprintf('in j, .N is %d\n', .N)); mean(a)}]

# .I can be different in j and by, enabling rowwise operations in by
DT[, .(.I, min(.SD[,-1]))]
DT[, .(min(.SD[,-1])), by=.I]

# Do not expect this to correctly append the value of .BY in each group; copy(.BY) will work.
by_tracker = list()
DT[, { append(by_tracker, .BY); sum(v) }, by=x]
```

`split` *Split data.table into chunks in a list*

Description

Split method for data.table. Faster and more flexible. Be aware that processing list of data.tables will be generally much slower than manipulation in single data.table by group using by argument, read more on data.table.

Usage

```
## S3 method for class 'data.table'
split(x, f, drop = FALSE,
      by, sorted = FALSE, keep.by = TRUE, flatten = TRUE,
      ..., verbose = getOption("datatable.verbose"))
```

Arguments

x	data.table
f	Same as split.data.frame. Use by argument instead, this is just for consistency with data.frame method.
drop	logical. Default FALSE will not drop empty list elements caused by factor levels not referred by that factors. Works also with new arguments of split data.table method.
by	character vector. Column names on which split should be made. For length(by) > 1L and flatten FALSE it will result nested lists with data.tables on leafs.
sorted	When default FALSE it will retain the order of groups we are splitting on. When TRUE then sorted list(s) are returned. Does not have effect for f argument.
keep.by	logical default TRUE. Keep column provided to by argument.
flatten	logical default TRUE will unlist nested lists of data.tables. When using f results are always flattened to list of data.tables.
...	When using f, passed to split.data.frame. When using by, sep is recognized as with the default method.
verbose	logical default FALSE. When TRUE it will print to console data.table split query used to split data.

Details

Argument f is just for consistency in usage to data.frame method. Recommended is to use by argument instead, it will be faster, more flexible, and by default will preserve order according to order in data.

Value

List of data.tables. If using flatten FALSE and length(by) > 1L then recursively nested lists having data.tables as leafs of grouping according to by argument.

See Also

data.table, rbindlist

Examples

```
set.seed(123)
DT = data.table(x1 = rep(letters[1:2], 6),
                x2 = rep(letters[3:5], 4),
                x3 = rep(letters[5:8], 3),
                y = rnorm(12))
DT = DT[sample(.N)]
DF = as.data.frame(DT)

# split consistency with data.frame: `x, f, drop`
all.equal(
    split(DT, list(DT$x1, DT$x2)),
    lapply(split(DF, list(DF$x1, DF$x2)), setDT)
)

# nested list using `flatten` arguments
split(DT, by=c("x1", "x2"))
split(DT, by=c("x1", "x2"), flatten=FALSE)

# dealing with factors
fdt = DT[, c(lapply(.SD, as.factor), list(y=y)), .SDcols=x1:x3]
fdf = as.data.frame(fdt)
sdf = split(fdf, list(fdf$x1, fdf$x2))
all.equal(
    split(fdt, by=c("x1", "x2"), sorted=TRUE),
    lapply(sdf[sort(names(sdf))], setDT)
)

# factors having unused levels, drop FALSE, TRUE
fdt = DT[, .(x1 = as.factor(c(as.character(x1), "c"))[-13L],
             x2 = as.factor(c("a", as.character(x2)))[-1L],
             x3 = as.factor(c("a", as.character(x3), "z"))[c(-1L,-14L)],
             y = y)]
fdf = as.data.frame(fdt)
sdf = split(fdf, list(fdf$x1, fdf$x2))
all.equal(
    split(fdt, by=c("x1", "x2"), sorted=TRUE),
    lapply(sdf[sort(names(sdf))], setDT)
)
sdf = split(fdf, list(fdf$x1, fdf$x2), drop=TRUE)
all.equal(
    split(fdt, by=c("x1", "x2"), sorted=TRUE, drop=TRUE),
    lapply(sdf[sort(names(sdf))], setDT)
)
```

`subset.data.table` *Subsetting data.tables*

Description

Returns subsets of a `data.table`.

Usage

```
  ## S3 method for class 'data.table'
subset(x, subset, select, ...)
```

Arguments

`x`	`data.table` to subset.
`subset`	logical expression indicating elements or rows to keep
`select`	expression indicating columns to select from `data.table`
`...`	further arguments to be passed to or from other methods

Details

The `subset` argument works on the rows and will be evaluated in the `data.table` so columns can be referred to (by name) as variables in the expression.

The `data.table` that is returned will maintain the original keys as long as they are not `select`-ed out.

Value

A `data.table` containing the subset of rows and columns that are selected.

See Also

`subset`

Examples

```
DT <- data.table(a=sample(c('a', 'b', 'c'), 20, replace=TRUE),
                 b=sample(c('a', 'b', 'c'), 20, replace=TRUE),
                 c=sample(20), key=c('a', 'b'))

sub <- subset(DT, a == 'a')
all.equal(key(sub), key(DT))
```

`substitute2` *Substitute expression*

Description

Experimental, more robust, and more user-friendly version of base R `substitute`.

Usage

```
substitute2(expr, env)
```

Arguments

expr	Unevaluated expression in which substitution has to take place.
env	List, or an environment that will be coerced to list, from which variables will be taken to inject into expr.

Details

For convenience function will turn any character elements of env argument into symbols. In case if character is of length 2 or more, it will raise an error. It will also turn any list elements into list calls instead. Behaviour can be changed by wrapping env into I call. In such case any symbols must be explicitly created, for example using as.name function. Alternatively it is possible to wrap particular elements of env into I call, then only those elements will retain their original class.

Comparing to base R substitute, substitute2 function:

1. substitutes calls argument names as well
2. by default converts character elements of env argument to symbols
3. by default converts list elements of env argument to list calls
4. does not accept missing env argument
5. evaluates elements of env argument

Value

Quoted expression having variables and call argument names substituted.

Note

Conversion of *character to symbol* and *list to list call* works recursively for each list element in env list. If this behaviour is not desired for your use case, we would like to hear about that via our issue tracker. For the present moment there is an option to disable that: options(datatable.enlist=FALSE). This option is provided only for debugging and will be removed in future. Please do not write code that depends on it, but use I calls instead.

See Also

substitute, I, call, name, eval

Examples

```
## base R substitute vs substitute2
substitute(list(var1 = var2), list(var1 = "c1", var2 = 5L))
substitute2(list(var1 = var2), list(var1 = "c1", var2 = 5L)) ## works also on names

substitute(var1, list(var1 = "c1"))
substitute2(var1, list(var1 = I("c1"))) ## enforce character with I

substitute(var1, list(var1 = as.name("c1")))
substitute2(var1, list(var1 = "c1")) ## turn character into symbol, for convenience
```

```
## mix symbols and characters using 'I' function, both lines will yield same result
substitute2(list(var1 = var2), list(var1 = "c1", var2 = I("some_character")))
substitute2(list(var1 = var2), I(list(var1 = as.name("c1"), var2 = "some_character")))

## list elements are enlist'ed into list calls
(cl1 = substitute(f(lst), list(lst = list(1L, 2L))))
(cl2 = substitute2(f(lst), I(list(lst = list(1L, 2L)))))
(cl3 = substitute2(f(lst), list(lst = I(list(1L, 2L)))))
(cl4 = substitute2(f(lst), list(lst = quote(list(1L, 2L)))))
(cl5 = substitute2(f(lst), list(lst = list(1L, 2L))))
cl1[[2L]] ## base R substitute with list element
cl2[[2L]] ## same
cl3[[2L]] ## same
cl4[[2L]] ## desired
cl5[[2L]] ## automatically

## character to name and list into list calls works recursively
(cl1 = substitute2(f(lst), list(lst = list(1L, list(2L)))))
(cl2 = substitute2(f(lst), I(list(lst = list(1L, list(2L))))) ## unless I() used
last(cl1[[2L]]) ## enlisted recursively
last(cl2[[2L]]) ## AsIs

## using substitute2 from another function
f = function(expr, env) {
  eval(substitute(
    substitute2(.expr, env),
    list(.expr = substitute(expr))
  ))
}
f(list(var1 = var2), list(var1 = "c1", var2 = 5L))
```

`tables` *Display 'data.table' metadata*

Description

Convenience function for concisely summarizing some metadata of all `data.table`s in memory (or an optionally specified environment).

Usage

```
tables(mb=type_size, order.col="NAME", width=80,
       env=parent.frame(), silent=FALSE, index=FALSE)
```

Arguments

`mb`	a function which accepts a `data.table` and returns its size in bytes. By default, `type_size` (same as `TRUE`) provides a fast lower bound by excluding the size of character strings in R's global cache (which may be shared) and excluding the size of list column items (which also may be shared). A column `"MB"` is included in the output unless `FALSE` or `NULL`.

order.col	Column name (character) by which to sort the output.
width	integer; number of characters beyond which the output for each of the columns COLS, KEY, and INDICES are truncated.
env	An environment, typically the .GlobalEnv by default, see Details.
silent	logical; should the output be printed?
index	logical; if TRUE, the column INDICES is added to indicate the indices assorted with each object, see indices.

Details

Usually tables() is executed at the prompt, where parent.frame() returns .GlobalEnv. tables() may also be useful inside functions where parent.frame() is the local scope of the function; in such a scenario, simply set it to .GlobalEnv to get the same behaviour as at prompt.

mb = utils::object.size provides a higher and more accurate estimate of size, but may take longer. Its default units="b" is appropriate.

Setting silent=TRUE prints nothing; the metadata is returned as a data.table invisibly whether silent is TRUE or FALSE.

Value

A data.table containing the information printed.

See Also

data.table, setkey, ls, objects, object.size

Examples

```
DT = data.table(A=1:10, B=letters[1:10])
DT2 = data.table(A=1:10000, ColB=10000:1)
setkey(DT,B)
tables()
```

---

test *Test assertions for equality, exceptions and console output*

Description

An internal testing function used in data.table test scripts that are run by test.data.table.

Usage

```
test(num, x, y = TRUE,
     error = NULL, warning = NULL, message = NULL,
     output = NULL, notOutput = NULL, ignore.warning = NULL,
     options = NULL, env = NULL)
```

Arguments

num	A unique identifier for a test, helpful in identifying the source of failure when testing is not working. Currently, we use a manually-incremented system with tests formatted as n.m, where essentially n indexes an issue and m indexes aspects of that issue. For the most part, your new PR should only have one value of n (scroll to the end of inst/tests/tests.Rraw to see the next available ID) and then index the tests within your PR by increasing m. Note – n.m is interpreted as a number, so 123.4 and 123.40 are actually the same – please 0-pad as appropriate. Test identifiers are checked to be in increasing order at runtime to prevent duplicates being possible.
x	An input expression to be evaluated.
y	Pre-defined value to compare to x, by default TRUE.
error	When you are testing behaviour of code that you expect to fail with an error, supply the expected error message to this argument. It is interpreted as a regular expression, so you can be abbreviated, but try to include the key portion of the error so as not to accidentally include a different error message.
warning	Same as error, in the case that you expect your code to issue a warning. Note that since the code evaluates successfully, you should still supply y.
message	Same as warning but expects message exception.
output	If you are testing the printing/console output behaviour; e.g. with verbose=TRUE or options(datatable.verbose=TRUE). Again, regex-compatible and case sensitive.
notOutput	Or if you are testing that a feature does *not* print particular console output. Case insensitive (unlike output) so that the test does not incorrectly pass just because the string is not found due to case.
ignore.warning	A single character string. Any warnings emitted by x that contain this string are dropped. Remaining warnings are compared to the expected warning as normal.
options	A named list of options to set for the duration of the test. Any code evaluated during this call to test() (usually, x, or maybe y) will run with the named options set, and the original options will be restored on return. This is a named list since different options can have different types in general, but in typical usage, only one option is set at a time, in which case a named vector is also accepted.
env	A named list of environment variables to set for the duration of the test, much like options. A list entry set to NULL will unset (i.e., Sys.unsetenv) the corresponding variable.

Value

Logical TRUE when test passes, FALSE when test fails. Invisibly.

Note

NA_real_ and NaN are treated as equal, use identical if distinction is needed. See examples below.

If warning= is not supplied then you are automatically asserting no warning is expected; the test will fail if any warning does occur. Similarly for message=.

Multiple warnings are supported; supply a vector of strings to warning=. If x does not produce the correct number of warnings in the correct order, the test will fail.

Strings passed to notOutput= should be minimal; e.g. pick out single words from the output that you desire to check does not occur. The reason being so that the test does not incorrectly pass just because the output has slightly changed. For example notOutput="revised" is better than notOutput="revised flag to true". notOutput= is automatically case insensitive for this reason.

See Also

test.data.table

Examples

```
test = data.table:::test
test(1, x = sum(1:5), y = 15L)
test(2, log(-1), NaN, warning="NaNs")
test(3, sum("a"), error="invalid.*character")
# test failure example
stopifnot(
  test(4, TRUE, FALSE) == FALSE
)
# NA_real_ vs NaN
test(5.01, NA_real_, NaN)
test(5.03, all.equal(NaN, NA_real_))
test(5.02, identical(NaN, NA_real_), FALSE)
```

---

test.data.table *Runs a set of tests*

Description

Runs a set of tests to check data.table is working correctly.

Usage

```
test.data.table(script = "tests.Rraw", verbose = FALSE, pkg = ".",
                silent = FALSE,
                showProgress = interactive() && !silent,
                testPattern = NULL,
                memtest = Sys.getenv("TEST_DATA_TABLE_MEMTEST", 0),
                memtest.id = NULL, optional = FALSE)
```

Arguments

script	Run arbitrary R test script.
verbose	TRUE sets options(datatable.verbose=TRUE) for the duration of the tests. This tests there are no errors in the branches that produce the verbose output, and produces a lot of output. The output is normally used for tracing bugs or performance tuning. Tests which specifically test the verbose output is correct (typically looking for an expected substring) always run regardless of this option.
pkg	Root directory name under which all package content (ex: DESCRIPTION, src/, R/, inst/ etc..) resides. Used only in *dev-mode*.
silent	Controls what happens if a test fails. Like silent in try, TRUE causes the error message to be suppressed and FALSE to be returned, otherwise the error is returned.
showProgress	Output 'Running test <n> ...\r' at the start of each test?
testPattern	When present, a regular expression tested against the number of each test for inclusion. Useful for running only a small portion of a large test script.
memtest	Measure and report memory usage of tests (1:gc before ps, 2:gc after ps) rather than time taken (0) by default. Intended for and tested on Linux. See PR #5515 for more details.
memtest.id	An id for which to print memory usage for every sub id. May be a range of ids.
optional	If TRUE, the test will only run when the environment variable RUN_ALL_DATATABLE_TESTS is set to "yes". This allows certain optional tests to be skipped on CRAN but run in development or CI environments.

Details

Runs a series of tests. These can be used to see features and examples of usage, too. Running test.data.table will tell you the full location of the test file(s) to open.

Setting silent=TRUE sets showProgress=FALSE too, via the default of showProgress.

Value

If all tests were successful, TRUE is returned. Otherwise, see the silent argument above. silent=TRUE is intended for use at the start of production scripts; e.g. stopifnot(test.data.table(silent=TRUE)) to check data.table is passing its own tests before proceeding.

See Also

data.table, test

Examples

```
## Not run:
test.data.table()

## End(Not run)
```

timetaken *Pretty print of time taken*

Description

Pretty print of time taken since last started.at.

Usage

```
timetaken(started.at)
```

Arguments

started.at	The result of proc.time() taken some time earlier.

Value

A character vector of the form HH:MM:SS, or SS.MMMsec if under 60 seconds.

Examples

```
started.at=proc.time()
Sys.sleep(1)
cat("Finished in",timetaken(started.at),"\n")
```

transpose *Efficient transpose of list*

Description

transpose is an efficient way to transpose lists, data.frames or data.tables.

Usage

```
transpose(l, fill=NA, ignore.empty=FALSE, keep.names=NULL,
          make.names=NULL, list.cols=FALSE)
```

Arguments

l	A list, data.frame or data.table.
fill	Default is NA. It is used to fill shorter list elements so as to return each element of the transposed result of equal lengths.
ignore.empty	Default is FALSE. TRUE will ignore length-0 list elements.
keep.names	The name of the first column in the result containing the names of the input; e.g. keep.names="rn". By default NULL and the names of the input are discarded.

make.names	The name or number of a column in the input to use as names of the output; e.g. make.names="rn". By default NULL and default names are given to the output columns.
list.cols	Default is FALSE. TRUE will avoid promoting types and return columns of type list instead. factor will always be cast to character.

Details

The list elements (or columns of data.frame/data.table) should be all atomic. If list elements are of unequal lengths, the value provided in fill will be used so that the resulting list always has all elements of identical lengths. The class of input object is also preserved in the transposed result.

The ignore.empty argument can be used to skip or include length-0 elements.

This is particularly useful in tasks that require splitting a character column and assigning each part to a separate column. This operation is quite common enough that a function tstrsplit is exported.

factor columns are converted to character type. Attributes are not preserved at the moment. This may change in the future.

Value

A transposed list, data.frame or data.table.

list outputs will only be named according to make.names.

See Also

data.table, tstrsplit

Examples

```
ll = list(1:5, 6:8)
transpose(ll)
setDT(transpose(ll, fill=0))[]

DT = data.table(x=1:5, y=6:10)
transpose(DT)

DT = data.table(x=1:3, y=c("a","b","c"))
transpose(DT, list.cols=TRUE)

# base R equivalent of transpose
l = list(1:3, c("a", "b", "c"))
lapply(seq(length(l[[1]])), function(x) lapply(l, `[[`, x))
transpose(l, list.cols=TRUE)

ll = list(nm=c('x', 'y'), 1:2, 3:4)
transpose(ll, make.names="nm")
```

truelength *Over-allocation access*

Description

These functions are experimental and somewhat advanced. By *experimental* we mean their names might change and perhaps the syntax, argument names and types. So if you write a lot of code using them, you have been warned! They should work and be stable, though, so please report problems with them. `alloc.col` is just an alias to `setalloccol`. We recommend to use `setalloccol` (though `alloc.col` will continue to be supported) because the `set*` prefix in `setalloccol` makes it clear that its input argument is modified in-place.

Usage

```
truelength(x)
setalloccol(DT,
    n = getOption("datatable.alloccol"),        # default: 1024L
    verbose = getOption("datatable.verbose"))   # default: FALSE
alloc.col(DT,
    n = getOption("datatable.alloccol"),        # default: 1024L
    verbose = getOption("datatable.verbose"))   # default: FALSE
```

Arguments

`x`	Any type of vector, including `data.table` which is a `list` vector of column pointers.
`DT`	A `data.table`.
`n`	The number of spare column pointer slots to ensure are available. If `DT` is a 1,000 column `data.table` with 24 spare slots remaining, `n=1024L` means grow the 24 spare slots to be 1024. `truelength(DT)` will then be 2024 in this example.
`verbose`	Output status and information.

Details

When adding columns by reference using `:=`, we *could* simply create a new column list vector (one longer) and memcpy over the old vector, with no copy of the column vectors themselves. That requires negligible use of space and time, and long ago we did just that. However, that copy of the list vector of column pointers only (but not the columns themselves), a *shallow copy*, resulted in inconsistent behaviour in some circumstances. Therefore, data.table over-allocates the list vector of column pointers so that columns can be added fully by reference, consistently.

When the allocated column pointer slots are used up, to add a new column `data.table` must reallocate that vector. If two or more variables are bound to the same data.table this shallow copy may or may not be desirable, but we don't think this will be a problem very often (more discussion may be required on data.table issue tracker). Setting `options(datatable.verbose=TRUE)` includes messages if and when a shallow copy is taken. To avoid shallow copies there are several options: use

copy to make a deep copy first, use setalloccol to reallocate in advance, or, change the default allocation rule (perhaps in your .Rprofile); e.g., options(datatable.alloccol=10000L).

Please note: over-allocation of the column pointer vector is not for efficiency *per se*; it is so that := can add columns by reference without a shallow copy.

Value

truelength(x) returns the length of the vector allocated in memory. length(x) of those items are in use. Currently, it is just the list vector of column pointers that is over-allocated (i.e. truelength(DT)), not the column vectors themselves, which would in future allow fast row insert(). For tables loaded from disk however, truelength is 0, which is perhaps unexpected. data.table detects this state and over-allocates the loaded data.table when the next column addition occurs. All other operations on data.table (such as fast grouping and joins) do not need truelength.

setalloccol *reallocates* DT by reference. This may be useful for efficiency if you know you are about to going to add a lot of columns in a loop. It also returns the new DT, for convenience in compound queries.

See Also

copy

Examples

```
DT = data.table(a=1:3,b=4:6)
length(DT)                 # 2 column pointer slots used
truelength(DT)             # 1026 column pointer slots allocated
setalloccol(DT, 2048)
length(DT)                 # 2 used
truelength(DT)             # 2050 allocated, 2048 free
DT[,c:=7L]                 # add new column by assigning to spare slot
truelength(DT)-length(DT)  # 2047 slots spare
```

---

tstrsplit *strsplit and transpose the resulting list efficiently*

Description

This is equivalent to transpose(strsplit(...)). This is a convenient wrapper function to split a column using strsplit and assign the transposed result to individual columns. See examples.

Usage

```
tstrsplit(x, ..., fill=NA, type.convert=FALSE, keep, names=FALSE)
```

Arguments

x	The vector to split (and transpose).
...	All the arguments to be passed to strsplit.
fill	Default is NA. It is used to fill shorter list elements so as to return each element of the transposed result of equal lengths.
type.convert	TRUE calls type.convert with as.is=TRUE on the columns. May also be a function, list of functions, or named list of functions to apply to each part; see examples.
keep	Specify indices corresponding to just those list elements to retain in the transposed result. Default is to return all.
names	TRUE auto names the list with V1, V2 etc. Default (FALSE) is to return an unnamed list.

Details

It internally calls strsplit first, and then transpose on the result.

names argument can be used to return an auto named list, although this argument does not have any effect when used with :=, which requires names to be provided explicitly. It might be useful in other scenarios.

Value

A transposed list after splitting by the pattern provided.

See Also

data.table, transpose, type.convert

Examples

```
x = c("abcde", "ghij", "klmnopq")
strsplit(x, "", fixed=TRUE)
tstrsplit(x, "", fixed=TRUE)
tstrsplit(x, "", fixed=TRUE, fill="<NA>")

# using keep to return just 1,3,5
tstrsplit(x, "", fixed=TRUE, keep=c(1,3,5))

# names argument
tstrsplit(x, "", fixed=TRUE, keep=c(1,3,5), names=LETTERS[1:3])

DT = data.table(x=c("A/B", "A", "B"), y=1:3)
DT[, c("c1") := tstrsplit(x, "/", fixed=TRUE, keep=1L)][]
DT[, c("c1", "c2") := tstrsplit(x, "/", fixed=TRUE)][]

# type.convert argument
DT = data.table(
  w = c("Yes/F", "No/M"),
  x = c("Yes 2000-03-01 A/T", "No 2000-04-01 E/R"),
```

```
  y = c("1/1/2", "2/5/2.5"),
  z = c("Yes/1/2", "No/5/3.5"),
  v = c("Yes 10 30.5 2000-03-01 A/T", "No 20 10.2 2000-04-01 E/R"))

# convert each element in the transpose list to type factor
DT[, tstrsplit(w, "/", type.convert=as.factor)]

# convert part and leave any others
DT[, tstrsplit(z, "/", type.convert=list(as.numeric=2:3))]

# convert part with one function and any others with another
DT[, tstrsplit(z, "/", type.convert=list(as.factor=1L, as.numeric))]

# convert the remaining using 'type.convert(x, as.is=TRUE)' (i.e. what type.convert=TRUE does)
DT[, tstrsplit(v, " ", type.convert=list(as.IDate=4L, function(x) type.convert(x, as.is=TRUE)))]
```

`update_dev_pkg` *Perform update of development version of a package*

Description

Downloads and installs latest development version, only when a new commit is available. Defaults are set to update `data.table`, other packages can be used as well. Repository of a package has to include git commit SHA information in PACKAGES file.

Usage

```
update_dev_pkg(pkg="data.table",
       repo="https://Rdatatable.gitlab.io/data.table",
       field="Revision", type=getOption("pkgType"), lib=NULL, ...)
```

Arguments

`pkg`	character scalar, package name.
`repo`	character scalar, url of package devel repository.
`field`	character scalar, metadata field to use in PACKAGES file and DESCRIPTION file, default `"Revision"`.
`type`	character scalar, default `getOption("pkgType")`, used to define if package has to be installed from sources, binaries or both.
`lib`	character scalar, library location where package is meant to be upgraded.
`...`	passed to `install.packages`.

Details

In case if a devel repository does not provide binaries user will need development tools installed for package compilation, like *Rtools* on Windows, or alternatively eventually set `type="source"`.

Value

Invisibly `TRUE` if package was updated, otherwise `FALSE`.

data.table repositories

By default the function uses our GitLab-hosted R repository at `https://Rdatatable.gitlab.io/data.table`. This repository is updated nightly. It runs multiple test jobs (on top of GitHub tests jobs run upstream) and publish the package (sources and binaries), even if GitLab test jobs are failing. Status of GitLab test jobs can be checked at Package Check Results.
We also publish bleeding edge version of the package on GitHub-hosted R repository at `https://Rdatatable.gitlab.io/d` (just minor change in url from *lab* to *hub*). GitHub version should be considered less stable than GitLab one. It publishes only package sources.
There are also other repositories maintained by R community, for example `https://rdatatable.r-universe.dev`. Those can be used as well, but as they are unlikely to provide git commit SHA, the function will install the package even if latest version is already installed.

Note

Package namespace is unloaded before attempting to install newer version.

See Also

`data.table`

Examples

```
if (FALSE) data.table::update_dev_pkg()
```

Index