Package 'data.table'

September 27, 2021

Version 1.14.2

Title Extension of `data.frame`

Depends R (>= 3.1.0)

Imports methods

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

SystemRequirements zlib

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

ByteCompile TRUE

NeedsCompilation yes

Author Matt Dowle [aut, cre],
Arun Srinivasan [aut],
Jan Gorecki [ctb],
Michael Chirico [ctb],
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],
@marc-outins [ctb],
Roy Storey [ctb],
Manish Saraswat [ctb],
Morgan Jacob [ctb],
Michael Schubmehl [ctb],
Davis Vaughan [ctb],
Toby Hocking [ctb],
Leonardo Silvestri [ctb],
Tyson Barrett [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],
Ben Schwen [ctb]

Maintainer Matt Dowle <mattjdowle@gmail.com>

Repository CRAN

Date/Publication 2021-09-27 16:30:02 UTC

R topics documented:

`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 = getOption("datatable.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]
```

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`. It may be a single comma separated string such as `key="x,y,z"`, or a vector of names such as `key=c("x","y","z")`.
`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` anymore. 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.

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

When `j` is a character vector of column names, a numeric vector of column positions to select or of the form `startcol:endcol`, and the value returned is always a `data.table`. `with=FALSE` is not necessary anymore to select columns dynamically. Note that `x[,cols]` is equivalent to `x[,..cols]` and to `x[,cols,with=FALSE]` and to `x[,.SD,.SDcols=cols]`.

`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`. Use `options(datatable.nomatch=NULL)` to change the default value (used when `nomatch` is not supplied).

`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 or numeric positions. This is useful for speed when applying a function through a subset of (possible very many) columns; 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)].

See examples as well as vignette("datatable-secondary-indices-and-auto-indexing").

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][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` 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://github.com/Rdatatable/data.table/wiki (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

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                 # to refer vars in `j` from the outside of data use `..` prefix
DT[, ..colNum]                 # same, equivalent to DT[, .SD, .SDcols=colNum]
DT[["v"]]                      # same as DT[, v] but much faster

# 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][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)
DT["a", on=.(x)]               # same, for convenience, no need to quote every column
```

```
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
# key(kDT)<-"x"                    # copies whole table, please use set* functions instead
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

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
  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
```

:= *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`.

`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.

Usage

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

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

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 with. If not provided, implies *all rows*. The `:=` form is more powerful as it allows *subsets* and `joins` based add/update columns by reference. 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
```

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}]          # Use the functional form, `:=`(), instead (see above).
```

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 (whether or not fractional data is truncated). The motivation for this is efficiency. 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.

`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.

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]

## 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()))

---

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, will apply not only to data.table but also attributes of the columns. It will skip c("row.names",".internal.selfref") data.table attributes.
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, ...)

## 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.
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 *POSIXct* or *Date* type in first column.

Usage

```
as.xts.data.table(x, ...)
```

Arguments

x	data.table to convert to xts, must have *POSIXct* or *Date* in the first column. All others 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)
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`.

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
```

---

cdt *data.table exported C routines*

Description

Note that this interface is going to be changed in next release. Some of 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 subsetDT(SEXP x, SEXP rows, SEXP cols);
# p_dtCsubsetDT = R_GetCCallable("data.table", "CsubsetDT");
```

Details

For details how to use those see *Writing R Extensions* manual *Linking to native routines in other packages* section.

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(CsubsetDT,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.

See Also

data.table, 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))
```

`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.

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.

At optimisation level >= 1, i.e., `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.

At optimisation level >= 2, i.e., `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,tai` (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.

At optimisation level >= 3, i.e., `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:
# 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="Mb") # 400MB, 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


## 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"))
```

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 message. 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 fun.aggregate is present, takes the value by applying the function on a 0-length vector.
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.

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; 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 deprecated, 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=2:4) # calls melt.data.table

# dcast is an S3 method in data.table from v1.9.6
```

```
dcast(DT, time ~ variable, fun=mean) # using partial matching of argument
dcast(DT, diet ~ variable, fun=mean)
dcast(DT, diet+chick ~ time, drop=FALSE)
dcast(DT, diet+chick ~ time, drop=FALSE, fill=0)

# using subset
dcast(DT, chick ~ time, fun=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)                      # default is drop=TRUE
dcast(DT, v1 + v2 ~ v3, drop=FALSE)         # all missing combinations of both LHS and RHS
dcast(DT, v1 + v2 ~ v3, drop=c(FALSE, TRUE)) # all missing combinations of only LHS
dcast(DT, v1 + v2 ~ v3, drop=c(TRUE, FALSE)) # all missing combinations of only RHS

# 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, 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', fill = 0L)  #  0L --> 0
dcast(DT, v2 ~ v3, value.var = 'v4', 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=sum, value.var=c("d1","d2"))
# multiple fun.aggregate
dcast(DT, x + y ~ z, fun=list(sum, mean), value.var="d1")
# multiple fun.agg and value.var (all combinations)
dcast(DT, x + y ~ z, fun=list(sum, mean), value.var=c("d1", "d2"))
# multiple fun.agg and value.var (one-to-one)
dcast(DT, x + y ~ z, fun=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), ...)

## 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, i.e., the last (or rightmost) of identical elements would correspond to duplicated = FALSE.
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).
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.

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

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
)
```

---

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(...)
```

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.

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

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
```

---

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 length 1. Default value NA. NULL is treated as NA.

Details

In contrast to ifelse attributes are copied from yes to the output. This is useful when returning Date, factor or other classes.

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

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)
```

---

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 = if (!is.null(key(x))) key(x) else key(y),
    by.y = key(y), maxgap = 0L, minoverlap = 1L,
    type = c("any", "within", "start", "end", "equal"),
    mult = c("all", "first", "last"),
    nomatch = getOption("datatable.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. And the start column should always be <= end column. If x is keyed, by.x is equal to key(x), else key(y). by.y defaults to key(y).
maxgap	It should be a non-negative integer value, >= 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	It should be a positive integer value, > 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. Use options(datatable.nomatch=NULL) to change the default value (used when nomatch is not supplied).
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 or 0 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.table` 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=".", 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,
key=NULL, index=NULL,
showProgress=getOption("datatable.showProgress", interactive()),
data.table=getOption("datatable.fread.datatable", TRUE),
```

```
nThread=getDTthreads(verbose),
logical01=getOption("datatable.logical01", FALSE),  # due to change to TRUE; see NEWS
keepLeadingZeros = getOption("datatable.keepLeadingZeros", FALSE),
yaml=FALSE, autostart=NA, 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 character columns to factors?

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 `NA`s, 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`. If not "." (default) then usually ",". 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` default is `TRUE`. Strips leading and trailing whitespaces of unquoted fields. If `FALSE`, only header trailing spaces are removed.

`fill` logical (default is `FALSE`). If `TRUE` then in case the rows have unequal length, blank fields are implicitly filled.

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

`key` Character vector of one or more column names which is passed to `setkey`. It may be a single comma separated string such as `key="x,y,z"`, or a vector of names such as `key=c("x","y","z")`. 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.

`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`.

`autostart` Deprecated and ignored with warning. Please use `skip` instead.

`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, 2GB).

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 and locale: `fread(...,dec=",")` should just work. `fread` uses C function

strtod to read numeric data; e.g., 1.23 or 1,23. strtod retrieves the decimal separator (. or , usually) from the locale of the R session rather than as an argument passed to the strtod function. So for fread(...,dec=",") to work, fread changes this (and only this) R session's locale temporarily to a locale which provides the desired decimal separator.

On Windows, "French_France.1252" is tried which should be available as standard (any locale with comma decimal separator would suffice) and on unix "fr_FR.utf8" (you may need to install this locale on unix). fread() is very careful to set the locale back again afterwards, even if the function fails with an error. The choice of locale is determined by options()$datatable.fread.dec.locale. This may be a *vector* of locale names and if so they will be tried in turn until the desired dec is obtained; thus allowing more than two different decimal separators to be selected. This is a new feature in v1.9.6 and is experimental. In case of problems, turn it off with options(datatable.fread.dec.experiment=FALSE).

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. Secure URLS (ftps:// and https://) are downloaded with curl::curl_download; 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.

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://cerebralmastication.com/2009/11/loading-big-data-into-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

finagler = "to get or achieve by guile or manipulation" https://dictionary.reference.com/browse/finagler

On YAML, see https://yaml.org/; on csvy, see https://csvy.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 (MB):", round(file.info("test.csv")$size/1024^2),"\n")
# 50 MB (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)
# 500MB 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))


# Real data example (Airline data)
# https://stat-computing.org/dataexpo/2009/the-data.html

download.file("https://stat-computing.org/dataexpo/2009/2008.csv.bz2",
              destfile="2008.csv.bz2")
# 109MB (compressed)

system("bunzip2 2008.csv.bz2")
# 658MB (7,009,728 rows x 29 columns)

colClasses = sapply(read.csv("2008.csv",nrows=100,stringsAsFactors=FALSE),class)
# 4 character, 24 integer, 1 logical. Incorrect.

colClasses = sapply(read.csv("2008.csv",nrows=200,stringsAsFactors=FALSE),class)
# 5 character, 24 integer. Correct. Might have missed data only using 100 rows
# since read.table assumes colClasses is correct.

system.time(DF <- read.table("2008.csv", header=TRUE, sep=",",
    quote="",stringsAsFactors=FALSE,comment.char="",nrows=7009730,
    colClasses=colClasses))
# 24.4 secs

system.time(DT <- fread("2008.csv"))
#  1.9 secs

table(sapply(DT,class))
# 5 character and 24 integer columns. Correct without needing to worry about colClasses
# issue above.


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

# Decompresses .gz and .bz2 automatically :
fread("https://stat-computing.org/dataexpo/2009/1987.csv.bz2")


## End(Not run)
```

`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 = ",", 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
  logicalAsInt = logical01,  # deprecated
  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"),
  yaml = FALSE,
  bom = FALSE,
  verbose = getOption("datatable.verbose", 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"?

logicalAsInt — Deprecated. Old name for 'logical01'. Name change for consistency with 'fread' for which 'logicalAsInt' would not make sense.

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 (MB) per thread in the range 1 to 1024, default 8MB. 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"`.

`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?

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://www.h2o.ai/blog/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))                          #     382MB
```

```
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
)                                                                   #     774MB
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, ...)
cube(x, ...)
## S3 method for class 'data.table'
cube(x, j, by, .SDcols, id = FALSE, ...)
groupingsets(x, ...)
## S3 method for class 'data.table'
groupingsets(x, j, by, sets, .SDcols, id = FALSE, jj, ...)
```

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.

Details

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

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

```
# 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

Date and time classes with integer storage for fast sorting and grouping. Still experimental!

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)
month(x)
quarter(x)
year(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.
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.

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.

Conversions to and from Date and POSIXct formats are provided.

ITime does not account for time zones. 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.

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, 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.

These values are all taken directly from the 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/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"), as("2001-01-01", "IDate"))

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

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

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

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

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 = "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")
```

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

z = 0:1 + (0:1)*1i
CJ(x, z, sorted = FALSE) # support for sorting complex is not yet implemented
```

`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`.

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)
vector %like% pattern
vector %ilike% pattern
vector %flike% 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)?

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"]
```

---

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. 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 syntax; multiple patterns will produce multiple columns.

	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 for the measured variable names column. The default name is 'variable'.
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. Note that, if the type of value column is a list, na.rm = TRUE will have no effect.

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. The function patterns 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 deprecated, 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),
      i_2 = c(NA,6,7,8,9,10),
      f_1 = factor(sample(c(letters[1:3], NA), 6, TRUE)),
      f_2 = factor(c("z", "a", "x", "c", "x", "x"), ordered=TRUE),
      c_1 = sample(c(letters[1:3], NA), 6, 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(5,1)))), by = i_1]$c]
DT[, l_2 := DT[, list(c=list(rep(c_1, sample(5,1)))), by = i_1]$c]

# id, measure as character/integer/numeric vectors
melt(DT, id=1:2, measure="f_1")
melt(DT, id=c("i_1", "i_2"), measure=3) # same as above
melt(DT, id=1:2, measure=3L, value.factor=TRUE) # same, but 'value' is factor
melt(DT, id=1:2, measure=3:4, value.factor=TRUE) # 'value' is *ordered* factor

# preserves attribute when types are identical, ex: Date
melt(DT, id=3:4, measure=c("d_1", "d_2"))
melt(DT, id=3:4, measure=c("i_1", "d_1")) # attribute not preserved

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

# on character
melt(DT, id=1, measure=c("c_1", "f_1")) # value is char
melt(DT, id=1, measure=c("c_1", "i_2")) # i2 coerced to char

# on na.rm=TRUE. NAs are removed efficiently, from within C
melt(DT, id=1, measure=c("c_1", "i_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=1:2, measure=patterns("^f_", "^d_"), value.factor=TRUE)
# same as above, but provide list of columns directly by column names or indices
melt(DT, id=1:2, measure=list(3:4, c("d_1", "d_2")), value.factor=TRUE)
# same as above, but provide names directly:
melt(DT, id=1:2, measure=patterns(f="^f_", d="^d_"), value.factor=TRUE)
```

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

# return 'NA' for missing columns, 'na.rm=TRUE' ignored due to list column
melt(DT, id=1:2, measure=patterns("l_", "c_"), na.rm=TRUE)
```

---

merge *Merge two data.tables*

Description

Fast merge of two data.tables. The data.table method behaves very similarly to that of data.frames except that, by default, it attempts to merge

- 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.

Set the by, or 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
...)
```

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, then extra rows will be added to the output, one for each row in x that has no matching row in y. These rows will have 'NA's in those 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 merged data.table is sorted by setting the key to the by / by.x columns. If FALSE, the result is not sorted.

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.
...	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, NA is matched against NA (and NaN against NaN) while merging.

In versions <= v1.9.4, if the specified columns in by were not the key (or head of the key) of x or y, then a copy is first re-keyed prior to performing the merge. This was less performant as well as memory inefficient. The concept of secondary keys (implemented in v1.9.4) was used to overcome this limitation from v1.9.6+. No deep copies are made any more, thereby improving performance and memory efficiency. Also, there is better control for providing the columns to merge on with the help of the newly implemented by.x and by.y arguments.

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, as.data.table, [.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 = "A,B"))
(dt2 <- data.table(A = c(rep(1L, 5), 2L), B = letters[rep(2:4, 2)], Y = 6:1, key = "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="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="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)

# new feature, no need to set keys anymore
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)

# new feature, using by.x and by.y arguments
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")
```

`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 or integer, value to be used to fill when type=="const".
nan	(numeric x only) Either NaN or NA; if the former, NaN is treated as distinct from NA, otherwise, they are treated the same during replacement?
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

Examples

```
x = 1:10
x[c(1:2, 5:6, 9:10)] = NA
nafill(x, "locf")

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
```

patterns *Obtain matching indices corresponding to patterns*

Description

patterns returns the matching indices in the argument cols corresponding to the regular expression patterns provided. The patterns 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))
```

Arguments

...	A set of regular expression patterns.
cols	A character vector of names to which each pattern is matched.

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.

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: FALSE
    row.names=getOption("datatable.print.rownames"),    # default: TRUE
    col.names=getOption("datatable.print.colnames"),    # default: "auto"
    print.keys=getOption("datatable.print.keys"),       # default: FALSE
    trunc.cols=getOption("datatable.print.trunc.cols"), # default: FALSE
    quote=FALSE,
    timezone=FALSE, ...)
```

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).
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. "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).
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).
...	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).

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="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 = "aasdfasdfasdfasdfasdfasdfasdfasdfasdf",
                 thing_51 = vector("integer", 3),
                 thing_61 = vector("complex", 3))
print(DT, trunc.cols=TRUE)
options(old_width)
```

`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)
# 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. By default FALSE. When TRUE, use.names is set to TRUE.
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.

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.

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")
```

`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"))]
```

---

roll *Rolling functions*

Description

Fast rolling functions to calculate aggregates on sliding window. Function name and arguments are experimental.

Usage

```
frollmean(x, n, fill=NA, algo=c("fast", "exact"), align=c("right",
  "left", "center"), na.rm=FALSE, hasNA=NA, adaptive=FALSE)
frollsum(x, n, fill=NA, algo=c("fast","exact"), align=c("right", "left",
  "center"), na.rm=FALSE, hasNA=NA, adaptive=FALSE)
frollapply(x, n, FUN, ..., fill=NA, align=c("right", "left", "center"))
```

Arguments

x	vector, list, data.frame or data.table of numeric or logical columns.
n	integer vector, for adaptive rolling function also list of integer vectors, rolling window size.
fill	numeric, value to pad by. Defaults to NA.
algo	character, default "fast". When set to "exact", then slower algorithm is used. It suffers less from floating point rounding error, performs extra pass to adjust rounding error correction and carefully handles all non-finite values. If available it will use multiple cores. See details for more information.
align	character, define if rolling window covers preceding rows ("right"), following rows ("left") or centered ("center"). Defaults to "right".
na.rm	logical. Should missing values be removed when calculating window? Defaults to FALSE. For details on handling other non-finite values, see details below.
hasNA	logical. If it is known that x contains NA then setting to TRUE will speed up. Defaults to NA.
adaptive	logical, should adaptive rolling function be calculated, default FALSE. See details below.
FUN	the function to be applied in rolling fashion; see Details for restrictions
...	extra arguments passed to FUN in frollapply.

Details

froll* functions accepts vectors, lists, data.frames or data.tables. They always return a list except when the input is a vector and length(n)==1 in which case a vector is returned, for convenience. Thus rolling functions can be used conveniently within data.table syntax.

Argument n allows multiple values to apply rolling functions on multiple window sizes. If adaptive=TRUE, then it expects a list. Each list element must be integer vector of window sizes corresponding to every single observation in each column.

When algo="fast" then *on-line* algorithm is used, also any NaN,+Inf,-Inf is treated as NA. Setting algo="exact" will make rolling functions to use compute-intensive algorithm that suffers less from floating point rounding error. It also handles NaN,+Inf,-Inf consistently to base R. In case of some functions (like *mean*), it will additionally make 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 are special cases where for each single observation has own corresponding rolling window width. Due to the logic of adaptive rolling functions, following restrictions apply:

- align only "right".
- if list of vectors is passed to x, then all list vectors must have equal length.

When multiple columns or multiple windows width are provided, then they are run in parallel. Except for the algo="exact" which runs in parallel already.

frollapply computes rolling aggregate on arbitrary R functions. The input x (first argument) to the function FUN is coerced to *numeric* beforehand and FUN has to return a scalar *numeric* value.

Checks for that are made only during the first iteration when FUN is evaluated. Edge cases can be found in examples below. Any R function is supported, but it is not optimized using our own C implementation – hence, for example, using frollapply to compute a rolling average is inefficient. It is also always single-threaded because there is no thread-safe API to R's C eval. Nevertheless we've seen the computation speed up vis-a-vis versions implemented in base R.

Value

A list except when the input is a vector and length(n)==1 in which case a vector is returned.

Note

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 double.
- 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.
- partial window feature is not supported, although it can be accomplished by using adaptive=TRUE, see examples.

Be aware that rolling functions operates 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, one has to ensure that there are no gaps in input. For details see issue #3241.

References

Round-off error

See Also

shift, data.table

Examples

```
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

# partial window using adaptive rolling function
an = function(n, len) c(seq.int(n), rep(n, len-n))
n = an(3, nrow(d))
frollmean(d, n, adaptive=TRUE)

# frollsum
frollsum(d, 3:4)

# frollapply
frollapply(d, 3:4, sum)
f = function(x, ...) if (sum(x, ...)>5) min(x, ...) else max(x, ...)
frollapply(d, 3:4, f, na.rm=TRUE)

# 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

# frollapply corner cases
f = function(x) head(x, 2)     ## FUN returns non length 1
try(frollapply(1:5, 3, f))
f = function(x) {              ## FUN sometimes returns non length 1
  n = length(x)
  # length 1 will be returned only for first iteration where we check length
```

```
  if (n==x[n]) x[1L] else range(x) # range(x)[2L] is silently ignored!
}
frollapply(1:5, 3, f)
options(datatable.verbose=TRUE)
x = c(1,2,1,1,1,2,3,2)
frollapply(x, 3, uniqueN)     ## FUN returns integer
numUniqueN = function(x) as.numeric(uniqueN(x))
frollapply(x, 3, numUniqueN)
x = c(1,2,1,1,NA,2,NA,2)
frollapply(x, 3, anyNA)       ## FUN returns logical
as.logical(frollapply(x, 3, anyNA))
options(datatable.verbose=FALSE)
f = function(x) {             ## FUN returns character
  if (sum(x)>5) "big" else "small"
}
try(frollapply(1:5, 3, f))
f = function(x) {             ## FUN is not type-stable
  n = length(x)
  # double type will be returned only for first iteration where we check type
  if (n==x[n]) 1 else NA # NA logical turns into garbage without coercion to double
}
try(frollapply(1:5, 3, f))
```

`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
```

---

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.
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))
```

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.

Details

To reorder data.table columns, the idiomatic way is to use setcolorder(x,neworder), instead of doing 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")
```

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 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. It may be a single comma separated string such as key="x,y,z", or a vector of names such as key=c("x","y","z").
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(.), as this will make a complete copy of the input object before to convert it to a data.table. The setDT function takes care of this issue by allowing to convert lists - both named and unnamed lists and data.frames *by reference* instead. That is, the input object is modified in place, no copy is being made.

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

Examples

```
set.seed(45L)
X = data.frame(A=sample(3, 10, TRUE),
         B=sample(letters[1:3], 10, TRUE),
         C=sample(10), stringsAsFactors=FALSE)

# 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)[]
```

---

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 envioronment 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' - `fread()`
- 'forder.c', 'fsort.c', and 'reorder.c' - `forder()` and related
- 'froll.c', 'frolladaptive.c', and 'frollR.c' - `froll()` and family
- 'fwrite.c' - `fwrite()`
- 'gsumm.c' - GForce in various places, see GForce
- 'nafill.c' - `nafill()`
- 'subset.c' - Used in `[.data.table` subsetting
- 'types.c' - Internal testing usage

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.

`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 columns are always sorted in *ascending* order. The table is changed *by reference* and `setkey` is very memory efficient.

There are three reasons `setkey` is desirable: i) binary search and joins are faster when they detect they can use an existing key, ii) grouping by a leading subset of the key columns is faster because the groups are already gathered contiguously in RAM, iii) 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 rownames.

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.

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 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
```

`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 s.f. 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)]
```

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 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(.)] to be consistent with base::order. Only x[order(.)] 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.

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=NA, type=c("lag", "lead", "shift"), 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	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") and "shift" (behave same as "lag" except given names).
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.

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)

# 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

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. See the vignettes and examples here and in data.table. .EACHI is a symbol passed to by; i.e. by=.EACHI.

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].
- .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.

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' and 'y'
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[, .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
```

`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`	factor or list of factors. 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.
`...`	passed to data.frame way of processing when using `f` argument.
`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))
```

tables *Display 'data.table' metadata*

Description

Convenience function for concisely summarizing some metadata of all `data.tables` in memory (or an optionally specified environment).

Usage

```
tables(mb=TRUE, order.col="NAME", width=80,
       env=parent.frame(), silent=FALSE, index=FALSE)
```

Arguments

`mb`	`logical`; `TRUE` adds the rough size of each `data.table` in megabytes to the output under column `MB`.
`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.

Note that on older versions of R, `object.size` may be slow, so setting `mb=FALSE` may speed up execution of `tables` significantly.

Setting `silent=TRUE` prints nothing; the metadata are 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)
```

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.

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)
```

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?

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

---

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)
```

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.

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`.

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)
```

---

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 is what v1.7.2 did. 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. So, as from v1.7.3 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 in R 2.14.0+ (and random in R <= 2.13.2), 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.
`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`

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)][]
```

`update.dev.pkg` *Perform update of development version of a package*

Description

It will download and install package from devel repository only when new commit is available there, otherwise only PACKAGES file is transferred. Defaults are set to update `data.table`, other packages can be used as well. Their repository has to include git commit information in PACKAGES file.

Usage

```
## S3 method for class 'dev.pkg'
update(object="data.table",
       repo="https://Rdatatable.gitlab.io/data.table",
       field="Revision", type=getOption("pkgType"), lib=NULL, ...)
```

Arguments

`object`	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, and eventually set `type="source"`.

Value

NULL.

Note

Package namespace is unloaded before attempting to install newer version.

See Also

data.table

Examples

```
# data.table::update.dev.pkg()
```

Index