Package 'data.table'

October 27, 2017

Version 1.10.4-3

Title Extension of `data.frame`

Depends R (>= 3.0.0)

Imports methods

Suggests bit64, knitr, nanotime, chron, ggplot2 (>= 0.9.0), plyr, reshape, reshape2, testthat (>= 0.4), hexbin, fastmatch, nlme, xts, gdata, GenomicRanges, caret, curl, zoo, plm, rmarkdown, parallel

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, a fast friendly file reader and parallel file writer. Offers a natural and flexible syntax, for faster development.

License GPL-3 | file LICENSE

URL http://r-datatable.com

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

MailingList datatable-help@lists.r-forge.r-project.org

VignetteBuilder knitr

ByteCompile TRUE

NeedsCompilation yes

Author Matt Dowle [aut, cre], Arun Srinivasan [aut], Jan Gorecki [ctb], Tom Short [ctb], Steve Lianoglou [ctb], Eduard Antonyan [ctb]

Maintainer Matt Dowle <mattjdowle@gmail.com>

Repository CRAN

Date/Publication 2017-10-27 07:40:34 UTC

R topics documented:

data.table-package . . . 3
:= . . . 12
address . . . 16
all.equal . . . 16
as.data.table . . . 18
as.data.table.xts . . . 19
as.xts.data.table . . . 20
between . . . 21
chmatch . . . 22
copy . . . 24
data.table-class . . . 26
datatable.optimize . . . 26
dcast.data.table . . . 28
duplicated . . . 31
first . . . 34
foverlaps . . . 35
frank . . . 37
fread . . . 39
fsort . . . 47
fwrite . . . 48
IDateTime . . . 52
J . . . 55
last . . . 56
like . . . 57
melt.data.table . . . 58
merge . . . 61
na.omit.data.table . . . 63
patterns . . . 64
print.data.table . . . 65
rbindlist . . . 66
rleid . . . 68
rowid . . . 69
setattr . . . 70
setcolorder . . . 72
setDF . . . 73
setDT . . . 74
setDTthreads . . . 76
setkey . . . 77
setNumericRounding . . . 79
setops . . . 81
setorder . . . 82
shift . . . 84
shouldPrint . . . 85
special-symbols . . . 86
split . . . 87
subset.data.table . . . 89

tables 90
test.data.table 91
timetaken 92
transform.data.table 92
transpose 94
truelength 95
tstrsplit 96

Index **98**

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

Description

`data.table` *inherits* from `data.frame`. It offers fast and nemory 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 feel free to ask questions on Stack Overflow data.table tag or on datatable-help mailing list. To report a bug please type: `bug.report(package = "data.table")`.

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"),              # default: NA_integer_
  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 nc 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 Keys and fast binary search based subset and Secondary indices and auto indexing. *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.

The expression '.()' is a *shorthand* alias to list(); they both mean the same. As long as j returns a list, each element of the list becomes a column in the resulting data.table. This is the default *enhanced* mode.

When with=FALSE, j can only be a vector of column names or positions to select (as in data.frame).

Advanced: j also allows the use of special *read-only* symbols: .SD, .N, .I, .GRP, .BY.

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 Introduction to data.table vignette and examples.

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.

When with=FALSE 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 often useful in data.table to

select columns dynamically. Note that `x[, cols, with=FALSE]` is equivalent to `x[, .SD, .SDcols=cols]`.

`nomatch` Same as `nomatch` in `match`. When a row in `i` has no match to `x`, `nomatch=NA` (default) means `NA` is returned. `0` means no rows will be returned for that row of `i`. Use `options(datatable.nomatch=0)` 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]`

`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.

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

`on`
: Indicate which columns in `i` should be joined with columns in `x` along with the type of binary operator to join with. When specified, this overrides the keys set on `x` and `i`. There are multiple ways of specifying `on` argument:

  - As a character vector, e.g., `X[Y, on=c("a", "b")]`. This assumes both these columns are present in `X` and `Y`.
  - As a *named* character vector, e.g., `X[Y, on=c(x="a", y="b")]`. This is useful when column names to join by are different between the two tables. NB: `X[Y, on=c("a", y="b")]` is also possible if 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 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 ommitted in 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 inclusiv

X[, sum(a), keyby=b]      # get sum(a) grouped by 'b', and sort that result by the grouping column
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
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
# 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)
http://crantastic.org/packages/data-table (User reviews)
http://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, :=, alloc.col, 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 at the prompt
## 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, with=FALSE]            # 2nd column, returns a data.table always
colNum = 2
DT[, colNum, with=FALSE]       # 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=0]      # 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=0]               # inner join
DT[!X, on="x"]                         # not join
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=0]           # 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' 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[, grp := .GRP, by=x]               # add a group counter column
X[, DT[.BY, y, on="x"], by=x]         # join within each group

# 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 lambdain '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]

# Follow r-help posting guide, SUPPORT is here (*not* r-help) :
# http://stackoverflow.com/questions/tagged/data.table
# or
# datatable-help@lists.r-forge.r-project.org

## Not run:
vignette("datatable-intro")
vignette("datatable-reference-semantics")
vignette("datatable-keys-fast-subset")
vignette("datatable-secondary-indices-and-auto-indexing")
vignette("datatable-reshape")
vignette("datatable-faq")


test.data.table()          # over 5700 low level tests

# keep up to date with latest stable version on CRAN
update.packages()
# get the latest devel (needs Rtools for windows, xcode for mac)
install.packages("data.table", repos = "https://Rdatatable.github.io/data.table", type = "source")


## End(Not run)
```

:= *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 loopable 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 added newly.
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. Read the Reference Semantics HTML vignette to follow with examples. Some typical usages are:

```
DT[, col := val]                    # update (or add at the end if doesn't exist) a column call
DT[i, col := val]                   # same as above, but only for those rows specified in i and
 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 ar
```

```
DT[i, colvector := val, with = FALSE]       # OLD syntax. The contents of "colvector" in calling sc
DT[i, (colvector) := val]                  # same (NOW PREFERRED) shorthand syntax. The parens are e
DT[i, colC := mean(colB), by = colA]        # update (or add) column called "colC" by reference by
 DT[,`:=`(new1 = sum(colB), new2 = sum(colC))] # Functional form
```

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, check the FAQs vignette. 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`. See the Reference Semantics HTML vignette 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's 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 synax 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`, `alloc.col`, `truelength`, `set`

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

## 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, programatically.

Value

A character vector length 1.

References

http://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
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)

# 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, ...)

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.
`...`	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`, `:=`, `alloc.col`, `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")

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.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, ...)
```

Arguments

x	xts to convert to data.table
keep.rownames	keep xts index as *index* column in result data.table
...	ignored, just for consistency with as.data.table

See Also

as.xts.data.table

Examples

```
## Not run:
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 dt
print(as.data.table(sample.xts))

## End(Not run)
```

---

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

```
## Not run:
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 dt
print(sample.dt)
# print head of xts
print(head(as.xts.data.table(sample.dt))) # xts might not be attached on search path

## End(Not run)
```

---

between *Convenience functions for range subsets.*

Description

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

between is equivalent to x >= lower & x <= upper when incbounds=TRUE, or x > lower & y < upper when FALSE.

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)
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.
upper	Upper range bound.
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.

Details

From v1.9.8+, between is vectorised. lower and upper are recycled to length(x) if necessary.

non-equi joins were recently implemented in v1.9.8. It extends 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 as 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
```

---

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%, fmatch

Examples

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

# N is set small here (1e5) because CRAN runs all examples and tests every night, to catch
# any problems early as R itself changes and other packages run.
# The comments here apply when N has been changed to 1e7.
N = 1e5

u = as.character(as.hexmode(1:10000))
y = sample(u,N,replace=TRUE)
x = sample(u)
                                          #  With N=1e7 ...
system.time(a <- match(x,y))              #  4.8s
system.time(b <- chmatch(x,y))            #  0.9s   Faster than 1st fmatch
identical(a,b)
if (fastmatchloaded<-suppressWarnings(require(fastmatch))) {
    print(system.time(c <- fmatch(x,y)))  #  2.1s   Builds and caches hash
    print(system.time(c <- fmatch(x,y)))  #  0.00s  Uses hash
    identical(a,c)
}

system.time(a <- x %in% y)                #  4.8s
system.time(b <- x %chin% y)              #  0.9s
identical(a,b)
if (fastmatchloaded) {
    match <- fmatch                       # fmatch is drop in replacement
    print(system.time(c <- match(x,y)))   #  0.00s
    print(system.time(c <- x %in% y))     #  4.8s   %in% still prefers base::match
    # Anyone know how to get %in% to use fmatch (without masking %in% too)?
    rm(match)
    identical(a,c)
}

# 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))              # 34.0s
system.time(b <- chmatch(x,y))            #  6.4s
identical(a,b)
if (fastmatchloaded) {
    print(system.time(c <- fmatch(x,y)))  #  7.9s
    print(system.time(c <- fmatch(x,y)))  #  4.0s
    identical(a,c)
}
```

---

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 inheritence.
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 inorder 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.

- When expressions in `j` which contains only these functions `min, max, mean, median, var, sd, prod`, 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 replaced with `gmean, gmedian, gmin, gmax` instead.

  Normally, once the rows belonging to each groups are identified, the values corresponding to the group is 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. It when used separately or combined with the functions mentioned above still uses GForce.
- 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.

Auto indexing: `data.table` also allows for blazing fast subsets by creating an *index* on the first run. Any successive subsets on the same column then reuses this index to *binary search* (instead of *vector scan*) and is therefore fast.

At the moment, expressions of the form `dt[col == val]` and `dt[col %in% val]` are both optimised. We plan to expand this to more operators and conditions in the future.

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)

# restore optimization
options(datatable.optimize = Inf)

# auto indexing
options(datatable.auto.index = FALSE)
system.time(ans1 <- dt[id == 100L]) # vector scan
system.time(ans2 <- dt[id == 100L]) # vector scan
system.time(dt[id

options(datatable.auto.index = TRUE)
system.time(ans1 <- dt[id == 100L]) # index + binary search subset
system.time(ans2 <- dt[id == 100L]) # only binary search subset
system.time(dt[id


## End(Not run)
```

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

Description

dcast.data.table is a much faster version of reshape2::dcast, but for data.tables. More importantly, it's capable of handling very large data quite efficiently in terms of memory usage in comparison to reshape2::dcast.

From 1.9.6, dcast is implemented as an S3 generic in data.table. To melt or cast data.tables, it is not necessary to load reshape2 anymore. If you have load reshape2, do so before loading data.table to prevent unwanted masking.

NEW: dcast.data.table can now cast multiple value.var columns and also 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. **NEW**: it is possible to provide a list of functions to fun.aggregate. 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. **NEW:** Following #1512, c(FALSE, TRUE) will only include all missing combinations of formula LHS. And 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. **NEW**: it is now possible to cast multiple value.var columns simultaneously. 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: `.` and `....` . represents no variable; `...` represents all variables not otherwise mentioned in `formula`; see Examples.

`dcast` also allows `value.var` columns of type `list`.

When variable combinations in `formula` doesn't identify a unique value in a cell, `fun.aggregate` will have to be specified, which defaults to `length` if unspecified. 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 whereever possible.

NEW: 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=1, or a list of length equal to `length(fun.aggregate)`. When `value.var` is a character vector or a list of length 1, 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 appled to each element of `value.var` column.

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

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

# dcast is a S3 method in data.table from v1.9.6
dcast(DT, time ~ variable, fun=mean)
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)

## Not run:
# benchmark against reshape2's dcast, minimum of 3 runs
set.seed(45)
DT <- data.table(aa=sample(1e4, 1e6, TRUE),
      bb=sample(1e3, 1e6, TRUE),
      cc = sample(letters, 1e6, TRUE), dd=runif(1e6))
system.time(dcast(DT, aa ~ cc, fun=sum)) # 0.12 seconds
system.time(dcast(DT, bb ~ cc, fun=mean)) # 0.04 seconds
# reshape2::dcast takes 31 seconds
system.time(dcast(DT, aa + bb ~ cc, fun=sum)) # 1.2 seconds

## End(Not run)

# NEW FEATURE - 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 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
```

`first` *First item of an object*

Description

Returns the first item of a vector or list, or the first row of a data.frame or data.table.

Usage

```
first(x, ...)
```

Arguments

`x`	A vector, list, data.frame or data.table. Otherwise the S3 method of `xts::first` is deployed.
`...`	Not applicable for `data.table::first`. Any arguments here are passed through to `xts::first`.

Value

If no other arguments are supplied it depends on the type of x. The first item of a vector or list. The first row of a `data.frame` or `data.table`. Otherwise, whatever `xts::first` returns (if package xts has been loaded, otherwise a helpful error).

If any argument is supplied in addition to `x` (such as `n` or `keep` in `xts::first`), regardless of `x`'s type, then `xts::first` is called if xts has been loaded, otherwise a helpful error.

See Also

`NROW`, `head`, `tail`, `last`

Examples

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

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's 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"),
    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. Note: equal is not yet implemented. But this is just a normal join of the type y[x, ...], unless you require also using maxgap and minoverlap arguments. 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 requirments, 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 — Same as nomatch in match. 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=0 means no rows will be returned for that row of x. The default value (used when nomatch is not supplied) can be changed from NA to 0 using options(datatable.nomatch=0).

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=0, 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, http://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=0L)
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".

bit64::integer64 type is also supported.

Usage

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

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

Arguments

x	A vector, or list with all it's 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 a "-", e.g., frank(x, a, -b, c). The -b works when b is of type character as well.
cols	A character vector of column names (or numbers) of x, to which obtain ranks for.
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's 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 types are also detected and read directly without needing to read as character before converting.

Dates are read as character currently. They can be converted afterwards using the excellent fasttime package or standard base functions.

'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, sep="auto", sep2="auto", nrows=-1L, header="auto", na.strings="NA", file,
stringsAsFactors=FALSE, verbose=getOption("datatable.verbose"), autostart=1L,
skip=0L, select=NULL, drop=NULL, colClasses=NULL,
integer64=getOption("datatable.integer64"),         # default: "integer64"
dec=if (sep!=".") "." else ",", col.names,
check.names=FALSE, encoding="unknown", quote="\"",
strip.white=TRUE, fill=FALSE, blank.lines.skip=FALSE, key=NULL,
showProgress=getOption("datatable.showProgress"),   # default: TRUE
data.table=getOption("datatable.fread.datatable")   # default: TRUE
)
```

Arguments

input	Either the file name to read (containing no \n character), a shell command that preprocesses the file (e.g. fread("grep blah filename")) or the input itself as a string (containing at least one \n), see examples. In both cases, a length 1 character string. A filename input is passed through path.expand for convenience and may be a URL starting http:// or file:///.
sep	The separator between columns. Defaults to the first character in the set [,\t |;:] that exists on line autostart outside quoted ("") regions, and separates the rows above autostart into a consistent number of fields, too.
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 on line autostart. NB: sep2 is not yet implemented.
nrows	The number of rows to read, by default -1 means all. Unlike read.table, it doesn't help speed to set this to the number of rows in the file (or an estimate), since the number of rows is automatically determined and is already fast. Only set nrows if you require the first 10 rows, for example. 'nrows=0' is a special case that just returns the column names and types; e.g., a dry run for a large file or to quickly check format consistency of a set of files before starting to read any.
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 read as type character is read as a blank string ("") and ",NA," is read as NA. Typical alternatives might be na.strings=NULL (no coercion to NA at all!) or perhaps na.strings=c("NA","N/A","null").

`file` File path, useful when we want to ensure that no shell commands will be executed. File path can also be provided to `input` argument.

`stringsAsFactors` Convert all character columns to factors?

`verbose` Be chatty and report timings?

`autostart` Any line number within the region of machine readable delimited text, by default 30. If the file is shorter or this line is empty (e.g. short files with trailing blank lines) then the last non empty line (with a non empty line above that) is used. This line and the lines above it are used to auto detect `sep`, `sep2` and the number of fields. It's extremely unlikely that `autostart` should ever need to be changed, we hope.

`skip` If 0 (default) use the procedure described below starting on line `autostart` to find the first data row. `skip>0` means ignore `autostart` and take line `skip+1` as the first data row (or column names according to header="auto"|TRUE|FALSE as usual). `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` Vector of column names or numbers to keep, drop the rest.

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

`colClasses` A character vector of classes (named or unnamed), as read.csv. Or a named list of vectors of column names or numbers, see examples. colClasses in fread is intended for rare overrides, not for routine use. fread will only promote a column to a higher type if colClasses requests it. It won't downgrade a column to a lower type since NAs would result. You have to coerce such columns afterwards yourself, if you really require data loss.

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

`dec` The decimal separator as in `base::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.

`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 doublequote, `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.

`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`.

`showProgress` `TRUE` displays progress on the console using `\r`. 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`.

Details

Once the separator is found on line `autostart`, the number of columns is determined. Then the file is searched backwards from `autostart` until a row is found that doesn't have that number of columns. Thus, the first data row is found and any human readable banners are automatically skipped. This feature can be particularly useful for loading a set of files which may not all have consistently sized banners. Setting `skip>0` overrides this feature by setting `autostart=skip+1` and turning off the search upwards step.

A sample of 1,000 rows is used to determine column types (100 rows from 10 points). The lowest type for each column is chosen from the ordered list: `logical`, `integer`, `integer64`, `double`, `character`. This enables `fread` to allocate exactly the right number of rows, with columns of the right type, up front once. The file may of course still contain data of a higher type in rows outside the sample. In that case, the column types are bumped mid read and the data read on previous rows is coerced. Setting `verbose=TRUE` reports the line and field number of each mid read type bump and how long this type bumping took (if any).

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 (currenly 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.

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 = "".

Value

A data.table by default. A data.frame when argument data.table=FALSE; e.g. options(datatable.fread.datatable

References

Background :
https://cran.r-project.org/doc/manuals/R-data.html
http://stackoverflow.com/questions/1727772/quickly-reading-very-large-tables-as-dataframes-in-r
http://www.biostat.jhsph.edu/~rpeng/docs/R-large-tables.html
https://stat.ethz.ch/pipermail/r-help/2007-August/138315.html
http://www.cerebralmastication.com/2009/11/loading-big-data-into-r/
http://stackoverflow.com/questions/9061736/faster-than-scan-with-rcpp
http://stackoverflow.com/questions/415515/how-can-i-read-and-manipulate-csv-file-data-in-c

http://stackoverflow.com/questions/9352887/strategies-for-reading-in-csv-files-in-pieces
http://stackoverflow.com/questions/11782084/reading-in-large-text-files-in-r
http://stackoverflow.com/questions/45972/mmap-vs-reading-blocks
http://stackoverflow.com/questions/258091/when-should-i-use-mmap-for-file-access
http://stackoverflow.com/a/9818473/403310
http://stackoverflow.com/questions/9608950/reading-huge-files-using-memory-mapped-files

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

See Also

read.csv, url, Sys.setlocale

Examples

```
## Not run:

# Demo speedup
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))
# 60 sec (first time in fresh R session)

system.time(DF1 <- read.csv("test.csv",stringsAsFactors=FALSE))
# 30 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")))
# 10 sec (consistently). All known tricks and known nrows, see references.

require(data.table)
system.time(DT <- fread("test.csv"))
#  3 sec (faster and friendlier)
```

```
require(sqldf)
system.time(SQLDF <- read.csv.sql("test.csv",dbname=NULL))
# 20 sec (friendly too, good defaults)

require(ff)
system.time(FFDF <- read.csv.ffdf(file="test.csv",nrows=n))
# 20 sec (friendly too, good defaults)

identical(DF1,DF2)
all.equal(as.data.table(DF1), DT)
identical(DF1,within(SQLDF,{b<-as.integer(b);c<-as.numeric(c)}))
identical(DF1,within(as.data.frame(FFDF),d<-as.character(d)))

# 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 (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")))
# 100-200 sec (varies)

system.time(DT <- fread("testbig.csv"))
# 30-40 sec

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


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

download.file("http://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),class)
# 4 character, 24 integer, 1 logical. Incorrect.

colClasses = sapply(read.csv("2008.csv",nrows=200),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)
```

```
# 360 secs

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

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


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


## End(Not run)

# 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")   # silent loss of precision
DT[,sprintf("%.15E",A)]   # stored accurately as far as double precision allows

DT = fread("A\n1.46761e-313\n")   # detailed warning about ERANGE; 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

# 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
```

`fsort` *Fast parallel sort*

Description

Similar to `base::sort` but parallel. 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 `NA`s. 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

Returns the input in sorted order. Fast using parallelism.

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.

This is new functionality as of Nov 2016. We may need to refine argument names and defaults.

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"),
  logicalAsInt = FALSE, dateTimeAs = c("ISO","squash","epoch","write.csv"),
  buffMB = 8L, nThread = getDTthreads(),
  showProgress = getOption("datatable.showProgress"),
  verbose = getOption("datatable.verbose"))
```

Arguments

`x`	Any `list` of same length vectors; e.g. `data.frame` and `data.table`.
`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? If missing, append=TRUE and the file already exists, the default is set to FALSE for convenience to prevent column names appearing again mid 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.

logicalAsInt — Should logical values be written as 1 and 0 rather than "TRUE" and "FALSE"?

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==""`.

`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: `http://blog.h2o.ai/2016/04/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.

References

`http://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("
  str2=sample(sprintf("
  str3=sample(sapply(sample(2:30, 100, TRUE), function(n)
     paste0(sample(LETTERS, n, TRUE), collapse=""")), N, TRUE),
  str4=sprintf("
  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",row.names=F,quote=F))  #     63.2s
system("diff /dev/shm/tmp1.csv /dev/shm/tmp2.csv")                # identical

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

```
## End(Not run)
```

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, ...)
## 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'
as.chron(x, time = NULL, ...)
## 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 'ITime'
as.POSIXct(x, tz = "UTC", date = as.Date(Sys.time()), ...)
## S3 method for class 'ITime'
as.chron(x, date = NULL, ...)
## 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 convertable with as.IDate.
time	time-of-day object convertable with as.ITime.
digits	really units; one of the units listed for rounding. May be abbreviated.

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, POSIXct, and chron 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.

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 arguement 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.

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, http://gist.github.com/10238.

ISO 8601, http://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")

if (require(chron)) {
    as.chron(as.IDate("2000-01-01"))
    as.chron(as.ITime("10:45"))
    as.chron(as.IDate("2000-01-01"), as.ITime("10:45"))
    as.chron(as.ITime("10:45"), as.IDate("2000-01-01"))
    as.ITime(chron(times = "11:01:01"))
    IDateTime(chron("12/31/98","10:45:00"))
}
```

J *Creates a Join data table*

Description

Creates a `data.table` to be passed in as the `i` to a `[.data.table` join.

Usage

```
# DT[J(...)]                            # J() only for use inside DT[...].
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 repetition is applied.
`sorted`	logical. Should the input order be retained?
`unique`	logical. When `TRUE`, only unique values of each vectors are used (automatically).

Details

SJ and CJ are convenience functions for creating a data.table in the context of 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)].

x must have a key when passing in a join table as the i. See [.data.table

Value

J : the same result as calling list. J is a direct alias for list but results in clearer more readable code.

SJ : (S)orted (J)oin. The same value as J() but additionally setkey() is called on all the columns in the order they were passed in 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, 10 ids, and 100 dates, CJ returns a 1000 row table containing all the dates for all the ids. It gains sorted, which by default is TRUE for backwards compatibility. FALSE retains input order.

See Also

data.table, test.data.table

Examples

```
DT = data.table(A=5:1,B=letters[5:1])
setkey(DT,B)    # re-orders table and marks it sorted.
DT[J("b")]      # returns the 2nd row
DT[.("b")]      # same. Style of package plyr.
DT[list("b")]   # same

# 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, unique=TRUE) # unique(x) and unique(y) are computed automatically
```

---

last *Last item of an object*

Description

Returns the last item of a vector or list, or the last row of a data.frame or data.table.

Usage

```
last(x, ...)
```

Arguments

x	A vector, list, data.frame or data.table. Otherwise the S3 method of `xts::last` is deployed.
...	Not applicable for `data.table::last`. Any arguments here are passed through to `xts::last`.

Value

If no other arguments are supplied it depends on the type of x. The last item of a vector or list. The last row of a `data.frame` or `data.table`. Otherwise, whatever `xts::last` returns (if package xts has been loaded, otherwise a helpful error).

If any argument is supplied in addition to `x` (such as `n` or `keep` in `xts::last`), regardless of `x`'s type, then `xts::last` is called if xts has been loaded, otherwise a helpful error.

See Also

`NROW`, `head`, `tail`, `first`

Examples

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

---

like *Convenience function for calling regexpr.*

Description

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

Usage

```
like(vector,pattern)
vector %like% pattern
```

Arguments

vector	Either a `character` vector or a `factor`. A `factor` is faster.
pattern	Passed on to `grepl`.

Value

Logical vector, TRUE for items that match pattern.

Note

Current implementation does not make use of sorted keys.

See Also

data.table, grepl

Examples

```
DT = data.table(Name=c("Mary","George","Martha"), Salary=c(2,3,4))
DT[Name %like% "^Mar"]
```

---

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

Description

An S3 method for melting data.tables written in C for speed and memory efficiency. Since v1.9.6, melt.data.table allows melting into multiple columns simultaneously.

It is not necessary to load reshape2 anymore. But if you have to, then load reshape2 package *before* loading data.table.

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.
measure.vars	vector of measure variables. Can be integer (corresponding measure column numbers) or character (measure column names) vector. If missing, all non-id columns will be assigned to it. measure.vars also now accepts a list of character/integer vectors to melt into multiple columns - i.e., melt into more than one value columns simultaneously. Use patterns to provide multiple patterns conveniently. 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. The default name is 'value'.
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. This is to be compatible with reshape2's melt.data.frame. To get a factor column, set value.factor = TRUE. melt.data.table also preserves ordered factors.

Value

An unkeyed data.table containing the molten data.

See Also

dcast, http://had.co.nz/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)

# 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"),
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 behavior works in a similar fashion as the merge.data.frame method does.
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 was not the key (or head of the key) of x or y, then a copy is first rekeyed prior to performing the merge. This was less performant and 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 anymore and therefore very performant and memory efficient. Also there is better control for providing the columns to merge on with the help of 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 1.12 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)
```

---

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 implictly 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
    quote=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.
quote	If TRUE, all output will appear in quotes, as in print.default.
...	Other arguments ultimately passed to format.

Details

By default, with an eye to the typically large number of observations in a codedata.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)
```

---

rbindlist *Makes one data.table from a list of many*

Description

Same as do.call("rbind", l) on data.frames, but much faster. See DETAILS for more.

Usage

```
rbindlist(l, use.names=fill, fill=FALSE, idcol=NULL)
# rbind(..., use.names=TRUE, fill=FALSE, idcol=NULL)
```

Arguments

l	A list containing data.table, data.frame or list objects. At least one of the inputs should have column names set. ... is the same but you pass the objects by name separately.
use.names	If TRUE items will be bound by matching column names. By default FALSE for rbindlist (for backwards compatibility) and TRUE for rbind (consistency with base). Columns with duplicate names are bound in the order of occurrence, similar to base. When TRUE, at least one item of the input list has to have non-null column names.
fill	If TRUE fills missing columns with NAs. By default FALSE. When TRUE, use.names has to be TRUE, and all items of the input list has to have non-null column names.
idcol	Generates an index column. Default (NULL) is not to. If idcol=TRUE then the column is auto named .id. Alternatively the column name can be directly provided, e.g., idcol = "id". If input is a named list, ids are generated using them, else using integer vector from 1 to length of input list. 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 a variable number of (potentially many) objects to stack, such as returned by lapply(fileNames, fread). rbind however 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.

In versions <= v1.9.2, each item for rbindlist should have the same number of columns as the first non empty item. rbind.data.table gained a fill argument to fill missing columns with NA in v1.9.2, which allowed for rbind(...) binding unequal number of columns.

In version > v1.9.2, these functionalities were extended to rbindlist (and written entirely in C for speed). rbindlist has use.names argument, which is set to FALSE by default for backwards compatibility. It also contains fill argument as well and can bind unequal columns when set to TRUE.

With these changes, the only difference between rbind(...) and rbindlist(l) is their *default argument* use.names.

If column i of input items do not all have the same type; e.g, a data.table may be bound with a list or a column is factor while others are character types, they are coerced to the highest type (SEXPTYPE).

Note that any additional attributes that might exist on individual items of the input list would not be preserved in the result.

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, http://stackoverflow.com/q/21421047/559784

Examples

```
DT = data.table(grp=rep(c("A", "B", "C", "A", "B"), c(2,2,3,1,2)), value=1:10)
rleid(DT$grp) # get run-length ids
rleidv(DT, "grp") # same as above

rleid(DT$grp, prefix="grp") # prefix with 'grp'

# get sum of value over run-length groups
DT[, sum(value), by=.(grp, rleid(grp))]
DT[, sum(value), by=.(grp, rleid(grp, prefix="grp"))]
```

---

rowid *Generate unique row ids within each group*

Description

Convenience functions for generating a unique row ids within each group. It accepts atomic vectors, lists, data.frames or data.tables as input.

rowid is intended for interactive use, particularly along with the function dcast to generate unique ids directly in the formula.

rowidv(dt, cols=c("x", "y")) is equivalent to column N in the code dt[, N := seq_len(.N), by=c("x", "y")].

See examples for more.

Usage

```
rowid(..., prefix=NULL)
rowidv(x, cols=seq_along(x), prefix=NULL)
```

Arguments

x	A vector, list, data.frame or data.table.
...	A sequence of numeric, integer64, character or logical vectors, all of same length. For interactive use.
cols	Only meaningful for lists, data.frames or data.tables. A character vector of column names (or numbers) of x.
prefix	Either NULL (default) or a character vector of length=1 which is prefixed to the row ids, returning a character vector (instead of an integer vector).

Value

When `prefix = NULL`, an integer vector with same length as `NROW(x)`, else a character vector with the value in `prefix` prefixed to the ids obtained.

See Also

`dcast.data.table`, `rleid`

Examples

```
DT = data.table(x=c(20,10,10,30,30,20), y=c("a", "a", "a", "b", "b", "b"), z=1:6)

rowid(DT$x) # 1,1,2,1,2,2
rowidv(DT, cols="x") # same as above

rowid(DT$x, prefix="group") # prefixed with 'group'

rowid(DT$x, DT$y) # 1,1,2,1,2,1
rowidv(DT, cols=c("x","y")) # same as above
DT[, .(N=seq_len(.N)), by=.(x,y)]$N # same as above

# convenient usage with dcast
dcast(DT, x ~ rowid(x, prefix="group"), value.var="z")
#     x group1 group2
# 1: 10      2      3
# 2: 20      1      6
# 3: 30      4      5
```

---

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)
```

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, the new column names, which must be the same length as the number of columns. See examples.
new	Optional. New column names, must be the same length as columns provided to old argument.

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

```
DF = data.frame(a=1:2,b=3:4)       # base data.frame to demo copies and syntax
try(tracemem(DF))                  # try() for R sessions opted out of memory profiling
```

```
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
try(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))

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)
```

Arguments

x	A data.table.
neworder	Character vector of the new column name ordering. May also be column numbers.

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"))
```

---

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

This feature request came up on the data.table mailing list. 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.
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. Default value 0 means to utilize all CPU available with an appropriate number of threads calculated by OpenMP. `getDTthreads()` returns the number of threads that will be used. This affects `data.table` only and does not change R itself or other packages using OpenMP. The most common usage expected is `setDTthreads(1)` to limit `data.table` to one thread for pre-existing explictly parallel user code; e.g. via packages parallel and foreach. Otherwise, nested parallelism may bite. As `data.table` becomes more parallel automatically internally, we expect explicit user parallelism to be needed less often.

Usage

```
setDTthreads(threads)
getDTthreads()
```

Arguments

`threads`	An integer >= 0. Default 0 means use all CPU available and leave the operating system to multi task.

Value

A length 1 `integer`. The old value is returned by `setDTthreads` so you can store that value and pass it to `setDTthreads` again after the section of your code where you, probably, limited to one thread.

setkey *Create key on a data table*

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.

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

key() returns the data.table's key if it exists, and NULL if none exist.

haskey() returns a logical TRUE/FALSE depending on whether the data.table has a key (or not).

Usage

```
setkey(x, ..., verbose=getOption("datatable.verbose"), physical = TRUE)
setkeyv(x, cols, verbose=getOption("datatable.verbose"), physical = TRUE)
setindex(...)
setindexv(...)
key(x)
indices(x, vectors = FALSE)
haskey(x)
key(x) <- value   #  DEPRECATED, please use setkey or setkeyv instead.
```

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 (only) of column names.
value	In (deprecated) key<-, a character vector (only) of column names.
verbose	Output status and information.
physical	TRUE changes the order of the data in RAM. FALSE adds a secondary key a.k.a. index.
vectors	logical scalar default FALSE, when set to TRUE then list of character vectors is returned, each vector refers to one index.

Details

setkey reorders (or sorts) the rows of a data.table by the columns provided. In versions 1.9+, for integer columns, a modified version of base's counting sort is implemented, which allows negative values as well. It is extremely fast, but is limited by the range of integer values being <=

1e5. If that fails, it falls back to a (fast) 4-pass radix sort for integers, implemented based on Pierre Terdiman's and Michael Herf's code (see links below). Similarly, a very fast 6-pass radix order for columns of type `double` is also implemented. This gives a speed-up of about 5-8x compared to `1.8.10` on `setkey` and all internal `order`/`sort` operations. Fast radix sorting is also implemented for `character` and `bit64::integer64` types.

The sort is *stable*; i.e., the order of ties (if any) is preserved, in both versions - `<=1.8.10` and `>= 1.9.0`.

In `data.table` versions `<= 1.8.10`, for columns of type `integer`, the sort is attempted with the very fast `"radix"` method in `sort.list`. If that fails, the sort reverts to the default method in `order`. For character vectors, `data.table` takes advantage of R's internal global string cache and implements a very efficient order, also exported as `chorder`.

In v1.7.8, the `key<-` syntax was deprecated. The `<-` method copies the whole table and we know of no way to avoid that copy without a change in R itself. Please use the `set*` functions instead, which make no copy at all. `setkey` accepts unquoted column names for convenience, whilst `setkeyv` accepts one vector of column names.

The problem (for `data.table`) with the copy by `key<-` (other than being slower) is that R doesn't maintain the over allocated truelength, but it looks as though it has. Adding a column by reference using `:=` after a `key<-` was therefore a memory overwrite and eventually a segfault; the over allocated memory wasn't really there after `key<-`'s copy. `data.tables` now have an attribute `.internal.selfref` to catch and warn about such copies. This attribute has been implemented in a way that is friendly with `identical()` and `object.size()`.

For the same reason, please use the other `set*` functions which modify objects by reference, rather than using the `<-` operator which results in copying the entire object.

It isn't good programming practice, in general, to use column numbers rather than names. 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's possible but deliberately a little harder; e.g., `setkeyv(DT,colnames(DT)[1:2])`.

Value

The input is modified by reference, and returned (invisibly) so it can be used in compound statements; e.g., `setkey(DT,a)[J("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. See `?copy`.

Note

Despite its name, `base::sort.list(x,method="radix")` actually invokes a *counting sort* in R, not a radix sort. See do_radixsort in src/main/sort.c. A counting sort, however, is particularly suitable for sorting integers and factors, and we like it. In fact we like it so much that `data.table` contains a counting sort algorithm for character vectors using R's internal global string cache. This is particularly fast for character vectors containing many duplicates, such as grouped data in a key

column. This means that character is often preferred to factor. Factors are still fully supported, in particular ordered factors (where the levels are not in alphabetic order).

References

http://en.wikipedia.org/wiki/Radix_sort
http://en.wikipedia.org/wiki/Counting_sort
http://cran.at.r-project.org/web/packages/bit/index.html
http://stereopsis.com/radix.html

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 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)   # rather than key(DT)<-keycols (which copies entire table)

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's bytes rather than bits because it's 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
http://en.wikipedia.org/wiki/Double-precision_floating-point_format
http://en.wikipedia.org/wiki/Floating_point
http://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's set functions, `union`, `intersect`, `setdiff` and `setequal` but for `data.tables`. Additional `all` argument controls if/how `duplicate` rows are returned. `bit64::integer64` is also supported.

Unlike SQL, data.table functions will retain order of rows in result.

Usage

```
fintersect(x, y, all = FALSE)
fsetdiff(x, y, all = FALSE)
funion(x, y, all = FALSE)
fsetequal(x, y)
```

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.

Details

Columns of type `complex` and `list` are not supported 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))
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, y)             # setequal
```

---

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.

Also x[order(.)] is now optimised internally to use data.table's fast order by default. data.table always reorders in C-locale. To sort by session locale, use x[base::order(.)] instead.

bit64::integer64 type is also supported for reordering rows of a data.table.

Usage

```
setorder(x, ..., na.last=FALSE)
setorderv(x, cols, 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 a "-", 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, to which to order by. 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's 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 it's default is TRUE. setorder and setorderv only accept TRUE/FALSE with default FALSE.

Details

data.table implements fast radix based ordering. In versions <= 1.9.2, it was only capable of increasing order (ascending). From 1.9.4 on, the functionality has been extended to decreasing order (descending) as well.

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). Note that -b also works with columns of type character unlike base::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's a subset operation as opposed to setorder or setorderv which reorders the data.table by reference.

If setorder results in reordering of the rows of a keyed data.table, then it's 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.

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"), give.names=FALSE)
```

Arguments

x	A vector, list, data.frame or data.table.
n	Non-negative integer vector denoting the offset to lead or lag the input by. To create multiple lead/lag vectors, provide multiple values to n.
fill	Value to pad by.
type	default is "lag". The other possible value is "lead".
give.names	default is FALSE which returns an unnamed list. When TRUE, names are automatically generated corresponding to type and n.

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")

# 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")
```

`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 and .GRP 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.

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 are used in j and 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 0. 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 explicity 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, it's 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.

See Also

`data.table`, `:=`, `set`, `datatable-optimize`

Examples

```
## Not run:
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
X[, DT[.BY, y, on="x"], by=x]           # join within each group

## End(Not run)
```

`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 refered 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 all objects of class 'data.table'*

Description

Lists all data.table's in memory, including number of rows, column names and any keys.

Usage

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

Arguments

mb	TRUE adds size of the data.table in MB to the output (slow in older versions of R).
order.col	Quoted column name to sort the output by
width	Number of characters to truncate the COLS output
env	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, or set it to .GlobalEnv
silent	By default tables() is expected to be called at the prompt for its compact print output. silent=TRUE prints nothing. The data statistics are returned as a data.table, silently, 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.data.table *Runs a set of tests.*

Description

Runs a set of tests to check data.table is working correctly.

Usage

```
test.data.table(verbose=FALSE, pkg="pkg", silent=FALSE)
```

Arguments

verbose	If TRUE sets datatable.verbose to TRUE for the duration of the tests.
pkg	Root directory name under which all package content (ex: DESCRIPTION, src/, R/, inst/ etc..) resides.
silent	Logical, default FASLE, when TRUE it will not raise error on in case of test fails.

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.

Value

When silent equals to TRUE it will return TRUE if all tests were successful. FALSE otherwise. If silent equals to FALSE it will return TRUE if all tests were successful. Error otherwise.

See Also

data.table

Examples

```
## Not run:
test.data.table()

## End(Not run)
```

`timetaken` *Pretty print of time taken*

Description

Pretty print of time taken since last started.at.

Usage

```
timetaken(started.at)
```

Arguments

`started.at`	The result of proc.time() taken some time earlier.

Value

A character vector of the form hh:mm:ss, or ss.mmm if under 60 seconds.

Examples

```
started.at=proc.time()
Sys.sleep(1)
cat("Finished in",timetaken(started.at),"\n")
```

`transform.data.table` *Data table utilities*

Description

Utilities for `data.table` transformation.

`transform` **by group is particularly slow. Please use** `:=` **by group instead.**

`within`, `transform` and other similar functions in `data.table` are not just provided for users who expect them to work, but for non-data.table-aware packages to retain keys, for example. Hopefully the (much) faster and more convenient `data.table` syntax will be used in time. See examples.

Usage

```
## S3 method for class 'data.table'
transform(`_data`, ...)
## S3 method for class 'data.table'
within(data, expr, ...)
```

Arguments

data, _data	data.table to be transformed.
...	for transform, Further arguments of the form tag=value. Ignored for within.
expr	expression to be evaluated within the data.table.

Details

within is like with, but modifications (columns changed, added, or removed) are updated in the returned data.table.

Note that transform will keep the key of the data.table provided the *targets* of the transform (i.e. the columns that appear in ...) are not in the key of the data.table. within also retains the key provided the key columns are not *touched*.

Value

The modified value of a copy of data.

See Also

transform, within and :=

Examples

```
DT <- data.table(a=rep(1:3, each=2), b=1:6)

DT2 <- transform(DT, c = a^2)
DT[, c:=a^2]
identical(DT,DT2)

DT2 <- within(DT, {
  b <- rev(b)
  c <- a*2
  rm(a)
})
DT[,`:=`(b = rev(b),
         c = a*2,
         a = NULL)]
identical(DT,DT2)

DT$d = ave(DT$b, DT$c, FUN=max)              # copies entire DT, even if it is 10GB in RAM
DT = DT[, transform(.SD, d=max(b)), by="c"]  # same, but even worse as .SD is copied for each group
DT[, d:=max(b), by="c"]                  # same result, but much faster, shorter and scales
```

```
# Multiple update by group. Convenient, fast, scales and easy to read.
DT[, `:=`(minb = min(b),
          meanb = mean(b),
          bplusd = sum(b+d)),  by=c%/%5]
DT
```

`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)
```

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.

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.

Usage

```
truelength(x)
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 negligibe 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 datatable-help). 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 `alloc.col` 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's 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's 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`.

`alloc.col` *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
alloc.col(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)][]
```

Index

∗Topic **chron**
IDateTime, 52
∗Topic **classes**
data.table-class, 26
∗Topic **data**
:=, 12
address, 16
as.data.table, 18
between, 21
chmatch, 22
copy, 24
data.table-package, 3
datatable.optimize, 26
dcast.data.table, 28
duplicated, 31
first, 34
foverlaps, 35
frank, 37
fread, 39
fwrite, 48
J, 55
last, 56
like, 57
melt.data.table, 58
merge, 61
na.omit.data.table, 63
patterns, 64
rbindlist, 66
rleid, 68
rowid, 69
setattr, 70
setcolorder, 72
setDF, 73
setDT, 74
setDTthreads, 76
setkey, 77
setNumericRounding, 79
setops, 81
setorder, 82
shift, 84
special-symbols, 86
split, 87
subset.data.table, 89
tables, 90
test.data.table, 91
timetaken, 92
transform.data.table, 92
transpose, 94
truelength, 95
tstrsplit, 96
∗Topic **methods**
data.table-class, 26
∗Topic **utilities**
IDateTime, 52
.BY, *5*
.BY *(special-symbols)*, 86
.GRP, *5*
.GRP *(special-symbols)*, 86
.I, *5*
.I *(special-symbols)*, 86
.N, *5*
.N *(special-symbols)*, 86
.SD, *5, 6*
.SD *(special-symbols)*, 86
:=, *8*, 12, *19*, *25*, *70–75*, *77*, *79*, *82, 83*, *87*, *93*
[.data.frame, *4*, *8*
[.data.table, *56*, *61, 62*
[.data.table *(data.table-package)*, 3
%between% *(between)*, 21
%chin% *(chmatch)*, 22
%inrange% *(between)*, 21
%like% *(like)*, 57
%chin%, *22*
%in%, *23*

address, 16
all.equal, 16, *17*, *33*
all.equal.data.table, *82*
alloc.col, *8*, *15*, *19*

alloc.col *(truelength)*, 95
anyDuplicated, *8*, *82*
anyDuplicated *(duplicated)*, 31
as.character.ITime *(IDateTime)*, 52
as.chron.IDate *(IDateTime)*, 52
as.chron.ITime *(IDateTime)*, 52
as.data.table, *4*, *8*, 18, *62*, *74, 75*
as.data.table.xts, 19, *21*
as.Date, *54*
as.Date.IDate *(IDateTime)*, 52
as.IDate *(IDateTime)*, 52
as.ITime *(IDateTime)*, 52
as.list.IDate *(IDateTime)*, 52
as.POSIXct, *54*
as.POSIXct.IDate *(IDateTime)*, 52
as.POSIXct.ITime *(IDateTime)*, 52
as.POSIXlt.ITime *(IDateTime)*, 52
as.xts.data.table, *20*, 20
auto-index *(datatable.optimize)*, 26
auto-indexing *(datatable.optimize)*, 26
autoindex *(datatable.optimize)*, 26
autoindexing *(datatable.optimize)*, 26

between, 21

c.IDate *(IDateTime)*, 52
charmatch, *23*
chgroup *(chmatch)*, 22
chmatch, 22
chorder, *78, 79*
chorder *(chmatch)*, 22
CJ, *6*, *8*, *19*
CJ *(J)*, 55
class:data.table *(data.table-class)*, 26
copy, *8*, *14, 15*, *19*, 24, *62*, *71*, *73–75*, *79*, *83*, *96*
cut.IDate *(IDateTime)*, 52

data.frame, *4*, *8*, *75*
data.table, *15*, *19*, *22*, *25, 26*, *33*, *37*, *39*, *56*, *58*, *62*, *64*, *67*, *69*, *71*, *74, 75*, *79*, *82*, *84*, *86–88*, *91*, *94*, *97*
data.table *(data.table-package)*, 3
data.table-class, 26
data.table-optimize *(datatable.optimize)*, 26
data.table-package, 3
data.table.optimize *(datatable.optimize)*, 26
datatable-optimize *(datatable.optimize)*, 26
datatable-symbols *(special-symbols)*, 86
datatable.optimize, 26
DateTimeClasses, *54*
dcast, *59*
dcast *(dcast.data.table)*, 28
dcast.data.table, 28, *70*
duplicated, 31, *33*, *82*

except *(setops)*, 81

fastorder *(setorder)*, 82
fexcept *(setops)*, 81
fintersect, *8*, *33*
fintersect *(setops)*, 81
first, 34, *57*
fmatch, *23*
forder *(setorder)*, 82
format.ITime *(IDateTime)*, 52
foverlaps, 35
frank, *8*, 37
frankv *(frank)*, 37
fread, 39, *51*
fsetdiff, *8*, *33*
fsetdiff *(setops)*, 81
fsetequal, *8*, *33*
fsetequal *(setops)*, 81
fsort, 47
funion, *8*, *33*
funion *(setops)*, 81
fwrite, 48

getDTthreads *(setDTthreads)*, 76
getNumericRounding, *27*, *73*
getNumericRounding *(setNumericRounding)*, 79
GForce *(datatable.optimize)*, 26
gforce *(datatable.optimize)*, 26
grep, *64*
grepl, *57, 58*

haskey *(setkey)*, 77
head, *34*, *57*
hour *(IDateTime)*, 52

IDate *(IDateTime)*, 52
IDate-class *(IDateTime)*, 52
IDateTime, *8*, 52

indices *(setkey)*, 77
inrange *(between)*, 21
integer64, *17*
intersect *(setops)*, 81
is.data.table *(as.data.table)*, 18
is.na.data.table *(data.table-package)*, 3
isoweek *(IDateTime)*, 52
ITime *(IDateTime)*, 52
ITime-class *(IDateTime)*, 52

J, *8*, *19*, 55, *79*

key *(setkey)*, 77
key2 *(setkey)*, 77
key<- *(setkey)*, 77

lag *(shift)*, 84
last, *34*, 56
lead *(shift)*, 84
like, *22*, 57
ls, *91*

make.names, *41*
match, *6*, *23*, *36*
mday *(IDateTime)*, 52
mean.IDate *(IDateTime)*, 52
melt, *65*
melt *(melt.data.table)*, 58
melt.data.table, *30*, 58, *64*
merge, 61, *62*
merge.data.frame, *61, 62*
merge.data.table, *8*, *19*
minute *(IDateTime)*, 52
month *(IDateTime)*, 52

na.omit, *8*
na.omit *(na.omit.data.table)*, 63
na.omit.data.table, 63
NROW, *34*, *57*

object.size, *91*
objects, *91*
Ops.data.table *(data.table-package)*, 3
order, *78*
order *(setorder)*, 82

path.expand, *40*
patterns, *58, 59*, 64
print.data.table, 65
print.default, *66*
print.ITime *(IDateTime)*, 52

quarter *(IDateTime)*, 52

rank, *38*
rank *(frank)*, 37
rbind *(rbindlist)*, 66
rbindlist, *8*, *19*, 66, *82*, *88*
read.csv, *44*
rep.IDate *(IDateTime)*, 52
rep.ITime *(IDateTime)*, 52
rle, *69*
rleid, *8*, 68, *70*
rleidv *(rleid)*, 68
round.IDate *(IDateTime)*, 52
rounding *(datatable.optimize)*, 26
rowid, *8*, *30*, *69*, 69
rowidv *(rowid)*, 69

second *(IDateTime)*, 52
seq.IDate *(IDateTime)*, 52
set, *15*, *25*, *71*, *73–75*, *79*, *83*, *87*
set *(:=)*, 12
set2key *(setkey)*, 77
set2keyv *(setkey)*, 77
setattr, *25*, 70, *73–75*, *79*, *83*
setcolorder, *71*, 72, *74, 75*, *79*, *83*
setDF, *8*, *19*, *25*, *71*, *73*, 73, *75*, *79*, *83*
setdiff *(setops)*, 81
setDT, *8*, *19*, *25*, *71*, *73, 74*, 74, *79*, *83*
setDTthreads, *51*, 76
setequal *(setops)*, 81
setindex *(setkey)*, 77
setindexv *(setkey)*, 77
setkey, *4*, *8*, *19*, *23*, *25*, *39*, *42*, *71*, *73–75*, 77, *83*, *91*
setkeyv, *75*
setkeyv *(setkey)*, 77
setnames, *25*, *73–75*, *79*, *83*
setnames *(setattr)*, 70
setNumericRounding, *8*, *19*, *27*, *32, 33*, *37*, *73*, *79*, 79, *83*
setops, 81
setorder, *8*, *25*, *39*, *71*, *73–75*, *79*, 82
setorderv *(setorder)*, 82
shift, 84
shouldPrint, 85
SJ, *8*, *19*
SJ *(J)*, 55

sort.list, *78, 79*
special-symbols, 86
split, 87
split.data.frame, *88*
split.data.table, *67*
split.IDate *(*IDateTime*)*, 52
storage.mode, *36*
strptime, *54*
strsplit *(*tstrsplit*)*, 96
subset, *90*
subset *(*subset.data.table*)*, 89
subset.data.table, 89
Sys.setlocale, *44*

tables, *8*, *79*, 90
tail, *34*, *57*
test.data.table, *8*, *56*, 91
timetaken, 92
transform, *93*
transform *(*transform.data.table*)*, 92
transform.data.table, 92
transpose, 94, *97*
truelength, *8*, *14, 15*, *19*, 95
tstrsplit, *94*, 96
type.convert, *97*

union *(*setops*)*, 81
unique, *33*, *82*
unique *(*duplicated*)*, 31
unique.data.frame, *32*
unique.data.table, *8*
uniqueN, *8*, *82*
uniqueN *(*duplicated*)*, 31
url, *44*

wday *(*IDateTime*)*, 52
week *(*IDateTime*)*, 52
within, *93*
within *(*transform.data.table*)*, 92
write.csv, *51*
write.table, *51*

yday *(*IDateTime*)*, 52
year *(*IDateTime*)*, 52