Package 'data.table'

August 29, 2016

Version 1.9.6

Title Extension of Data.frame

Author M Dowle, A Srinivasan, T Short, S Lianoglou with contributions from R Saporta, E Antonyan

Maintainer Matt Dowle <mattjdowle@gmail.com>

Depends R (>= 2.14.1)

Imports methods, chron

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

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 and a fast file reader (fread). Offers a natural and flexible syntax, for faster development.

License GPL (>= 2)

URL https://github.com/Rdatatable/data.table/wiki

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

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

VignetteBuilder knitr

ByteCompile TRUE

NeedsCompilation yes

Repository CRAN

Date/Publication 2015-09-19 22:13:43

R topics documented:

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

Description

`data.table` *inherits* from `data.frame`. It offers fast subset, fast grouping, fast update, fast ordered joins and list columns 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 *only* accept `data.frame`.

The 10 minute quick start guide to `data.table` may be a good place to start: `vignette("datatable-intro")`. Or, the first section of FAQs is intended to be read from start to finish and is considered core documentation: `vignette("datatable-faq")`. If you have read and searched these documents and the help page below, please feel free to ask questions on datatable-help or the Stack Overflow data.table tag. To report a bug please type: `bug.report(package="data.table")`.

Please check the homepage for up to the minute 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)

## 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 # join without setting keys, new feature from v1.9.6+
  ]
```

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")`.
`x`	A `data.table`.

`i` Integer, logical or character vector, single column numeric `matrix`, expression of column names, `list` or `data.table`.

integer and logical vectors work the same way they do in `[.data.frame`. Other than `NA`s in logical `i` are treated as `FALSE` and a single `NA` logical is not recycled to match the number of rows, as it is in `[.data.frame`.

character is matched to the first column of `x`'s key.

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.

When `i` is a `data.table`, `x` must have a key. `i` is *joined* to `x` using `x`'s key and the rows in `x` that match are returned. An equi-join is performed between each column in `i` to each column in `x`'s key; i.e., column 1 of `i` is matched to the 1st column of `x`'s key, column 2 to the second, etc. The match is a binary search in compiled C in O(log n) time. If `i` has *fewer* columns than `x`'s key then not all of `x`'s key columns will be joined to (a common use case) and many rows of `x` will (ordinarily) match to each row of `i`. If `i` has *more* columns than `x`'s key, the columns of `i` not involved in the join are included in the result. If `i` also has a key, it is `i`'s key columns that are used to match to `x`'s key columns (column 1 of `i`'s key is joined to column 1 of `x`'s key, column 2 of `i`'s key to column 2 of `x`'s key, and so on for as long as the shorter key) and a binary merge of the two tables is carried out. In all joins the names of the columns are irrelevant; the columns of `x`'s key are joined to in order, either from column 1 onwards of `i` when `i` is unkeyed, or from column 1 onwards of `i`'s key. In code, the number of join columns is determined by `min(length(key(x)),if (haskey(i)) length(key(i)) else ncol(i))`.

All types of 'i' may be prefixed with `!`. This signals a *not-join* or *not-select* should be performed. Throughout `data.table` documentation, where we refer to the type of 'i', we mean the type of 'i' *after* the '¡, if present. See examples.

Advanced: When `i` is an expression of column names that evaluates to `data.table` or `list`, a join is performed. We call this a *self join*.

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` A single column name, single expresson of column names, `list()` of expressions of column names, an expression or function call that evaluates to `list` (including `data.frame` and `data.table` which are `list`s, too), or (when `with=FALSE`) a vector of names or positions to select.

`j` is evaluated within the frame of the `data.table`; i.e., it sees column names as if they are variables. Use `j=list(...)` to return multiple columns and/or expressions of columns. A single column or single expression returns that type, usually a vector. See the examples.

`by` A single unquoted column name, a `list()` of expressions of column names, 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), or a character vector of column names.

The `list()` of expressions is evaluated within the frame of the `data.table` (i.e. it sees column names as if they are variables). 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. `j=list(...)` may be

omitted when there is just one expression, for convenience, typically a single expression such as `sum(colB)`; e.g., `DT[,sum(colB),by=colA]`.

When `by` contains the first `n` columns of `x`'s key, we call this a *keyed by*. In a keyed by the groups appear contiguously in RAM and memory is copied in bulk internally, for extra speed. Otherwise, we call it an *ad hoc by*. Ad hoc by is still many times faster than `tapply`, for example, but just not as fast as keyed by when datasets are very large, in particular when the size of *each group* is large. Not to be confused with `keyby=` defined below.

Advanced: When `i` is a `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*. It is particularly memory efficient as you don't have to materialise the join result only to aggregate later. Please refer to this Stackoverflow answer for a more detailed explanation until we roll out vignettes.

Advanced: When grouping, symbols `.SD`, `.BY`, `.N`, `.I` and `.GRP` may be used in the `j` expression, 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.

`.SD`, `.BY`, `.N`, `.I` and `.GRP` are *read only*. Their bindings 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).

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` An *ad-hoc-by* or *keyed-by* (just as `by=` defined above) but with an additional `setkey()` run on the `by` columns of the result afterwards, for convenience. It is common practice to use 'keyby=' routinely when you wish the result to be

sorted. Out loud we read keyby= as *by= then setkey*. Otherwise, 'by=' can be relied on to return the groups in the order they appear in the data.

`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 or a numeric vector of column positions to select, and the value returned is always a `data.table`. `with=FALSE` is often useful in `data.table` to select columns dynamically.

`nomatch` Same as `nomatch` in `match`. When a row in `i` has no match to `x`'s key, `nomatch=NA` (default) means `NA` is returned for `x`'s non-join columns for that row of `i`. `0` means no rows will be returned for that row of `i`. The default value (used when `nomatch` is not supplied) can be changed from `NA` to `0` using `options(datatable.nomatch=0)`.

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

`roll` Applies to the last join column, generally a date but can be any ordered variable, irregular and including gaps. If `roll=TRUE` and `i`'s 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 the *prevailing* value in `x` is *rolled* forward. This operation is particularly fast using a modified binary search. The operation is also known as last observation carried forward (LOCF). Usually, there should be no duplicates in `x`'s key, the last key column is a date (or time, or datetime) and all the columns of `x`'s key are joined to. 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*. When `roll` is a positive number, this limits how far values are carried forward. `roll=TRUE` is equivalent to `roll=+Inf`. When `roll` is a negative number, values are rolled backwards; i.e., next observation carried backwards (NOCB). Use `-Inf` for unlimited roll back. When `roll` is `"nearest"`, the nearest value is joined to.

`rollends` A logical vector length 2 (a single logical is recycled). When rolling forward (e.g. `roll=TRUE`) if a value is past the *last* observation within each group defined by the join columns, `rollends[2]=TRUE` will roll the last value forwards. `rollends[1]=TRUE` will roll the first value backwards if the value is before it. If `rollends=FALSE` the value of `i` must fall in a gap in `x` but not after the end or before the beginning of the data, for that group defined by all but the last join column. 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. `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` Advanced. Specifies the columns of `x` included in `.SD`. 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`

`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 `vignette("datatable-faq")`.

`on`
: A named atomic vector of column names indicating which columns in `i` should be joined to which columns in `x`. See `Examples`.

Details

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

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

It combines database like operations such as `subset`, `with` and `by` and provides similar joins that `merge` provides but faster. This is achieved by using R's column based ordered in-memory `data.frame` structure, `eval` within the environment of a `list`, the `[.data.table` mechanism to condense the features, and compiled C to make certain operations fast.

The package can be used just for rapid programming (compact syntax). Largest compute time benefits are on 64bit platforms with plentiful RAM, or when smaller datasets are repeatedly queried within a loop, or when other methods use so much working memory that they fail with an out of memory error.

As with `[.data.frame`, *compound queries* can be concatenated on one line; e.g.,

```
DT[,sum(v),by=colA][V1<300][tail(order(V1))]
# sum(v) by colA then return the 6 largest which are under 300
```

The `j` expression does not have to return data; e.g.,

```
DT[,plot(colB,colC),by=colA]
# produce a set of plots (likely to pdf) returning no data
```

Multiple `data.tables` (e.g. `X`, `Y` and `Z`) can be joined in many ways; e.g.,

```
X[Y][Z]
X[Z][Y]
X[Y[Z]]
X[Z[Y]]
```

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

1. it never has rownames. Instead it may have one *key* of one or more columns. This key can be used for row indexing instead of rownames.
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.

Since a `list` *is* a `vector`, `data.table` columns may be type `list`. Columns of type `list` can contain mixed types. Each item in a column of type `list` may be different lengths. This is true of `data.frame`, too.

Several *methods* are provided for `data.table`, including `is.na`, `na.omit`, `t`, `rbind`, `cbind`, `merge` and others.

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. Unexpected errors may occur if you manage to create a column of type POSIXlt. Please see NEWS for 1.6.3, and `IDateTime` instead. IDateTime has methods to convert to and from POSIXlt.

References

`data.table` homepage: http://datatable.r-forge.r-project.org/
User reviews: http://crantastic.org/packages/data-table
http://en.wikipedia.org/wiki/Binary_search
http://en.wikipedia.org/wiki/Radix_sort

See Also

`data.frame`, `[.data.frame` , `setkey`, `J`, `SJ`, `CJ`, `merge.data.table`, `tables`, `test.data.table`, `IDateTime`, `unique.data.table`, `copy`, `:=`, `alloc.col`, `truelength`, `rbindlist`, `setNumericRounding`

Examples

```
## Not run:
example(data.table)  # to run these examples at the prompt
## End(Not run)

DF = data.frame(x=rep(c("a","b","c"),each=3), y=c(1,3,6), v=1:9)
DT = data.table(x=rep(c("a","b","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()

DT[2]                      # 2nd row
DT[,v]                     # v column (as vector)
DT[,list(v)]               # v column (as data.table)
DT[2:3,sum(v)]             # sum(v) over rows 2 and 3
DT[2:5,cat(v,"\n")]        # just for j's side effect
DT[c(FALSE,TRUE)]          # even rows (usual recycling)

DT[,2,with=FALSE]          # 2nd column
colNum = 2
DT[,colNum,with=FALSE]     # same

setkey(DT,x)               # set a 1-column key. No quotes, for convenience.
setkeyv(DT,"x")            # same (v in setkeyv stands for vector)
v="x"
setkeyv(DT,v)              # same
# key(DT)<-"x"             # copies whole table, please use set* functions instead

DT["a"]                    # binary search (fast)
DT[x=="a"]                 # same; i.e. binary search (fast)

DT[,sum(v),by=x]           # keyed by
DT[,sum(v),by=key(DT)]     # same
DT[,sum(v),by=y]           # ad hoc by

DT["a",sum(v)]                    # j for one group
DT[c("a","b"),sum(v),by=.EACHI]   # j for two groups

X = data.table(c("b","c"),foo=c(4,2))
X

DT[X]                      # join
DT[X,sum(v),by=.EACHI]     # join and eval j for each row in i
DT[X,mult="first"]         # first row of each group
DT[X,mult="last"]          # last row of each group
DT[X,sum(v)*foo,by=.EACHI] # join inherited scope

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

DT["a"]                    # join to 1st column of key
DT[.("a")]                 # same, .() is an alias for list()
DT[list("a")]              # same
DT[.("a",3)]               # join to 2 columns
DT[.("a",3:6)]             # join 4 rows (2 missing)
DT[.("a",3:6),nomatch=0]   # remove missing
DT[.("a",3:6),roll=TRUE]   # rolling join (locf)

DT[,sum(v),by=.(y%%2)]   # by expression
DT[,.SD[2],by=x]           # 2nd row of each group
DT[,tail(.SD,2),by=x]      # last 2 rows of each group
```

```
DT[,lapply(.SD,sum),by=x]  # apply through columns by group

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

DT[,sum(v),x][V1<20]       # compound query
DT[,sum(v),x][order(-V1)]  # ordering results

print(DT[,z:=42L])         # add new column by reference
print(DT[,z:=NULL])        # remove column by reference
print(DT["a",v:=42L])      # subassign to existing v column by reference
print(DT["b",v2:=84L])     # 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()

DT[,.SD[which.min(v)],by=x][]  # nested query by group

DT[!("a")]                 # not join
DT[!"a"]                   # same
DT[!2:4]                   # all rows other than 2:4
DT[x!="b" | y!=3]          # not yet optimized, currently vector scans
DT[!("b",3)]               # same result but much faster


# new feature: 'on' argument, from v1.9.6+
DT1 = data.table(x=c("c", "a", "b", "a", "b"), a=1:5)
DT2 = data.table(x=c("d", "c", "b"), mul=6:8)

DT1[DT2, on=c(x="x")] # join on columns 'x'
DT1[DT2, on="x"] # same as above
DT1[DT2, .(sum(a) * mul), by=.EACHI, on="x"] # using by=.EACHI
DT1[DT2, .(sum(a) * mul), by=.EACHI, on="x", nomatch=0L] # using by=.EACHI

# 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-faq")

test.data.table()          # over 1,300 low level tests

update.packages()          # keep up to date

## End(Not run)
```

:= *Assignment by reference*

Description

Fast add, remove and modify subsets of columns, by reference.

Usage

```
#   DT[i, LHS:=RHS, by=...]

#   DT[i, c("LHS1","LHS2") := list(RHS1, RHS2), by=...]

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

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

Arguments

LHS	A single column name. Or, when `with=FALSE`, a 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 vector of replacement values. It is recycled in the usual way to fill the number of rows satisfying `i`, if any. Or, when `with=FALSE`, a `list` of replacement vectors which are applied (the `list` is recycled if necessary) to each column of `LHS` . To remove a column use `NULL`.
x	A `data.table`. Or, `set()` accepts `data.frame`, too.
i	Optional. In set(), integer row numbers to be assigned `value`. `NULL` represents all rows more efficiently than creating a vector such as `1:nrow(x)`.
j	In set(), integer column number to be assigned `value`.
value	Value to assign by reference to `x[i,j]`.

Details

`:=` is defined for use in `j` only. It *updates* or *adds* the column(s) by reference. It makes no copies of any part of memory at all. Typical usages are :

```
DT[i,colname:=value]                  # update (or add at the end if doesn't exist) a column called
 DT[i,"colname %":=value]                   # same. column called "colname %"
DT[i,(3:6):=value]                   # update existing columns 3:6 with value. Aside: parens are r
DT[i,colnamevector:=value,with=FALSE]    # old syntax. The contents of colnamevector in calling sc
DT[i,(colnamevector):=value]           # same, shorthand. Now preferred. The parens are enough to
DT[i,colC:=mean(colB),by=colA]          # update (or add) column called "colC" by reference by grou
 DT[,`:=`(new1=sum(colB), new2=sum(colC))] # multiple :=.
```

The following all result in a friendly error (by design) :

```
x := 1L                                  # friendly error
DT[i,colname] := value                   # friendly error
DT[i]$colname := value                   # friendly error
DT[,{col1:=1L;col2:=2L}]              # friendly error. Use `:=`() instead for multiple := (see ab
```

`:=` 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 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`.

Additional resources: search for `":="` in the FAQs vignette (3 FAQs mention `:=`), search Stack Overflow's data.table tag for "reference" (6 questions).

Advanced (internals) : sub assigning to existing columns is easy to see how that is done internally. Removing columns by reference is also straightforward by modifying the vector of column pointers only (using memmove in C). Adding columns is more tricky to see how that 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 also it bypasses `[<-` dispatch via `*tmp*` 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 flexible than `set()` because `:=` is intended to be combined with `i` and `by` in single queries on large datasets.

Value

DT is modified by reference and the new value is returned. If you require a copy, take a copy first (using DT2=copy(DT)). Recall that this package is for large data (of mixed column types, with multi-column keys) where updates by reference can be many orders of magnitude faster than copying the entire table.

See Also

data.table, copy, alloc.col, truelength, set

Examples

```
DT = data.table(a=LETTERS[c(1,1:3)],b=4:7,key="a")
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:=10L]     # subassign by reference to column d
DT               # DT changed by reference

DT[b>4,b:=d*2L]  # subassign to b using d, where b>4
DT["A",b:=0L]    # binary search for group "A" and set column b

DT[,e:=mean(d),by=a]  # add new column by group by reference
DT["B",f:=mean(d)]    # subassign to new column, NA initialized

## Not run:
# Speed example ...

m = matrix(1,nrow=100000,ncol=100)
DF = as.data.frame(m)
DT = as.data.table(m)

system.time(for (i in 1:1000) DF[i,1] <- i)
# 591 seconds
system.time(for (i in 1:1000) DT[i,V1:=i])
# 2.4 seconds  ( 246 times faster, 2.4 is overhead in [.data.table )
system.time(for (i in 1:1000) set(DT,i,1L,i))
# 0.03 seconds  ( 19700 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

Performs some factor level "stripping" and other operations to allow for a convenient test of data equality between data.table objects.

Usage

```
  ## S3 method for class 'data.table'
all.equal(target, current, trim.levels = TRUE, ...)
```

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.
...	Passed down to internal call of all.equal.list

Details

This function is used primarily to make life easy with a testing harness built around `test_that`. A call to `test_that::(expect_equal|equal)` will ultimately dispatch to this method when making an "equality" check.

Value

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

See Also

`all.equal.list`

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")
identical(all.equal(dt1, dt1), TRUE)
is.character(all.equal(dt1, dt2))
```

`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.

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 function for range subset logic.*

Description

Intended for use in [.data.table i.

Usage

```
between(x,lower,upper,incbounds=TRUE)
x
```

Arguments

`x`	Any vector e.g. numeric, character, date, ...
`lower`	Lower range bound.
`upper`	Upper range bound.
`incbounds`	`TRUE` means inclusive bounds i.e. [lower,upper]. `FALSE` means exclusive bounds i.e. (lower,upper).

Value

Logical vector as the same length as `x` with value `TRUE` for those that lie within the range [lower,upper] or (lower,upper).

Note

Current implementation does not make use of ordered keys.

See Also

`data.table`, `like`

Examples

```
DT = data.table(a=1:5, b=6:10)
DT[b %between% c(7,9)]
```

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))
```

`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 a implemented as a S3 generic in `data.table`. To melt or cast data.tables, it is not necessary to load `reshape2` anymore. If you have to, then load `reshape2` package before loading `data.table`.

NEW: `dcast.data.table` can now cast multiple `value.var` columns and also accepts multiple functions under `fun.aggregate` argument. 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 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` argument. See `examples`.
`sep`	Default is _ for backwards compatibility. Character vector of length 1, indicating the separating character in variable names generated during casting.
`...`	Any other arguments that maybe 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 subset of the data. Ex: subset = .(col1 <= 5) or subset = .(variable != "January").
`fill`	Value to fill missing cells with. If `fun.aggregate` is present, takes the value by applying the function on 0-length vector.
`drop`	`FALSE` will cast by including all missing combinations.

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 possible to cast multiple value.var columns simultaneously now. See examples.
verbose	Not used yet. Maybe dropped in the future or used to provide information messages onto 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 ... Their functionality is identical to that of reshape2::dcast.

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, http://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))

## 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` (by key columns or when no key all columns) are duplicates of a row with smaller subscripts.

`unique` returns a `data.table` with duplicated rows (by key) removed, or (when no key) duplicated rows by all columns 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))` but much faster for `atomic vectors`, `data.frames` and `data.tables`, for other types it dispatch to `length(unique(x))`. The number of unique rows are computed directly without materialising the intermediate unique data.table and is therefore memory efficient as well.

Usage

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

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

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

uniqueN(x, by=if (is.data.table(x)) key(x) else NULL)
```

Arguments

x	Atomic vectors, lists, data.frames or data.tables.
...	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 form x to use for uniqueness checks. Defaults to key(x)) which, by default, only uses the keyed columns. by=NULL uses all columns and acts like the analogous data.frame methods.

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 (for speed) whilst avoiding unexpected behaviour due to floating point representation by rounding the last two bytes off the significand (default) as explained in setNumericRounding.

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.

Any combination of columns can be used to test for uniqueness (not just the key columns) and are specified via the by parameter. To get the analagous data.frame functionality, set by to NULL.

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

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

---

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 an 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.

Columns of numeric types (i.e., double) have their last two bytes rounded off while computing overlap joins, by defalult, to avoid any unexpected behaviour due to limitations in representing floating point numbers precisely. Have a look at setNumericRounding to learn more.

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",
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", strip.white=TRUE,
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")`.
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 -1 (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, it uses the base function make.unique to ensure that column names are all unique.
encoding	default is "unknown". Other possible options are "UTF-8" and "Latin-1".
strip.white	default is TRUE. Strips leading and trailing whitespaces of unquoted fields. If FALSE, only header trailing spaces are removed.
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.

The first 5 rows, middle 5 rows and last 5 rows are then read to determine column types. The lowest type for each column is chosen from the ordered list 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 different type in rows other than first,

middle and last 5. 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:

- Spaces and othe 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,...`.

Quoting may be used to signify that numeric data should be read as text.

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.

Value

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

References

Background :
http://cran.r-project.org/doc/manuals/R-data.html
http://stackoverflow.com/questions/1727772/quickly-reading-very-large-tables-as-dataframes-in-r
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
```

`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, ...)

hour(x)
yday(x)
wday(x)
mday(x)
week(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.

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.

hour, codeyday, wday, mday, week, month, quarter, and year return integer values for hour, day of year, day of week, day of month, week, month, quarter, and year.

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.

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 usual silent repitition 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

like *Convenience function for calling regexpr.*

Description

Intended for use in [.data.table i.

Usage

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

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 entirely in C for speed. It also avoids any unnecessary copies by handling all arguments internally in a memory efficient manner.

From 1.9.6, to melt or cast data.tables, it is not necessary to load reshape2 anymore. If you have to, then load reshape2 package before loading data.table.

NEW: melt.data.table now allows melting into multiple columns simultaneously. See the details and examples section.

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. **NEW:** 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 the function patterns to provide multiple patterns conveniently. See the examples section.
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. See the examples section for the usage.

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

# NEW FEATURE: measure.vars can be 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`.

This `merge` method for `data.table` behaves very similarly to that of `data.frames` with one major exception: By default, the columns used to merge the `data.tables` are the shared key columns rather than the shared columns with the same names. Set the `by`, or `by.x`, `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.

In versions < v1.9.6, 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.

In version v1.9.4 secondary keys was implemented. In v1.9.6, the concept of secondary keys has been extended to merge. 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, ...].

Merges on numeric columns: Columns of numeric types (i.e., double) have their last two bytes rounded off while computing order, by defalult, to avoid any unexpected behaviour due to limitations in representing floating point numbers precisely. For large numbers (integers > 2^31), we recommend using bit64::integer64. Have a look at setNumericRounding to learn more.

Value

A new data.table based on the merged data tables, sorted by the columns set (or inferred for) the by argument.

See Also

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` *Regex patterns to extract columns from data.table*

Description

From `v1.9.6`, `melt.data.table` has a new 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.ta` vignette linked below to learn more.

`patterns` is designed purely for convenience, to be used only within the `measure.vars` argument of `melt.data.table`. Column names corresponding to each pattern from the `data.table` is melted into a separate column.

Usage

```
patterns(...)
```

Arguments

`...`	A set of patterns. See example.

See Also

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

Examples

```
# makes sense only in the context of melt at the moment
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"))
```

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`

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(...)
rleidv(x, cols=seq_along(x))
```

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.

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

An integer vector with same length as NROW(x).

See Also

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

---

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, the same length as old.

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(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 *Convert 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: http://bit.ly/1xkokNQ. 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

setkey, setcolorder, setattr, setnames, set, :=, setorder, copy, setDT

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 *Convert 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

setkey, setcolorder, setattr, setnames, set, :=, setorder, copy, setDF

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)[]
```

---

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)
set2key(...)
set2keyv(...)
key(x)
key2(x)
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.

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.

Note that columns of numeric types (i.e., double) have their last two bytes rounded off while computing order, by defalult, to avoid any unexpected behaviour due to limitations in representing floating point numbers precisely. Have a look at setNumericRounding to learn more.

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
```

`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: 2 (default), 1 or 0 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. To deal with this automatically for convenience, when joining or grouping, data.table rounds such data to apx 11 s.f. which is plenty of digits for many cases. This is achieved by rounding the last 2 bytes off the significand. Where this is not enough, `setNumericRounding` can be used to reduce to 1 byte rounding, or no rounding (0 bytes rounded) for full precision.

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 2 bytes, at most 6 passes are needed. With no rounding, at most 8 passes are needed and hence may be slower. The choice of default is not for speed however, but to avoid surprising results such as in the example below.

For large numbers (integers > 2^31), we recommend using `bit64::integer64` rather than setting rounding to `0`.

If you're using `POSIXct` type column with *millisecond* (or lower) resolution, you might want to consider setting `setNumericRounding(1)` . This'll become the default for `POSIXct` types in the future, instead of the default `2`.

Value

`setNumericRounding` returns no value; the new value is applied. `getNumericRounding` returns the current value: 0, 1 or 2.

See Also

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)   # turn off rounding
DT[.(0.4)]   # works
DT[.(0.6)]   # no match, confusing since 0.6 is clearly there in DT
```

```
setNumericRounding(2)   # restore default
DT[.(0.6)]   # works as expected

# 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
setNumericRounding(0)
DT[,.N,by=id]   # 3 rows
# better to use bit64::integer64 for such ids
setNumericRounding(2)
```

`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. Columns of numeric types (i.e., double) have their last two bytes rounded off while computing order, by defalult, to avoid any unexpected behaviour due to limitations in representing floating point numbers precisely. Have a look at setNumericRounding to learn more.

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 providing the periods to lead/lag 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 period 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 periods 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 period=1 and pad with NA (returns vector)
shift(x, n=1, fill=NA, type="lag")
# lag with period=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")
```

`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")
```

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.

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

TRUE if all tests were successful. FALSE 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: quote(max(100L,ncol(DT)+64L))
    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 column pointer slots to reserve in memory, including existing columns. May be a numeric, or a quote()-ed expression (see default). If DT is a 10 column data.table, n=1000 means grow the spare slots from 90 to 990, assuming the default of 100 has not been changed.
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=1000).

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; i.e., in both cases perhaps unexpected. data.table detects this state and over-allocates the loaded data.table when the next column addition or deletion 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)             # 100 column pointer slots allocated
alloc.col(DT,200)
length(DT)                 # 2 used
truelength(DT)             # 200 allocated, 198 free
DT[,c:=7L]                 # add new column by assigning to spare slot
truelength(DT)-length(DT)  # 197 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)
```

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.

Details

It internally calls `strsplit` first, and then `transpose` on the result.

Value

A transposed list.

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>")

DT = data.table(x=c("A/B", "A", "B"), y=1:3)
DT[, c("c1", "c2") := tstrsplit(x, "/", fixed=TRUE)][]
```

Index