Package ‘data.table’

September 3, 2013

Version 1.8.10

Title Extension of data.frame for fast indexing, fast ordered joins,fast assignment, fast grouping and list columns.

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

Maintainer Matthew Dowle <mdowle@mdowle.plus.com>

Depends R (>= 2.12.0)

Imports methods

Suggests chron, ggplot2 (>= 0.9.0), plyr, reshape, testthat (>= 0.4),hexbin, fastmatch, nlme, xts, bit64

Description
Enhanced data.frame. Fast indexing, fast ordered joins, fast assignment by reference, fast grouping and list columns in a short and flexible syntax. i and j may be expressions of column names directly, for faster development. Example: X[Y] is a fast join for large data.

License GPL (>= 2)

URL http://datatable.r-forge.r-project.org/,
http://stackoverflow.com/questions/tagged/data.table

BugReports https://r-forge.r-project.org/tracker/?group_id=240

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

ByteCompile TRUE

NeedsCompilation yes

Repository CRAN

Date/Publication 2013-09-03 06:40:57

R topics documented:

:= *Assignment by reference*

Description

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

Usage

```
#   DT[i, LHS:=RHS, 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 colum
DT[i,"colname
DT[i,(3:6):=value]                        # update existing columns 3:6 with value. Aside: pare
DT[i,colnamevector:=value,with=FALSE]     # old syntax. The contents of colnamevector in callin
DT[i,(colnamevector):=value]              # same, shorthand. Now preferred. The parens are enou
DT[i,colC:=mean(colB),by=colA]            # update (or add) column called "colC" by reference b
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 :=
```

`:=` 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) and search data.table's wiki.

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.table`s 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))
```

`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

Integer vector containing the locations of `x` which 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)
    }
```

---

data.table *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.

NEW :

- help page for :=
- keyby argument
- character and numeric now allowed as key column types
- := by group

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,
    rolltolast = FALSE   # deprecated
    ]
```

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, 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 `lists`, 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.
	Advanced: Aggregation for a subset of known groups is particularly efficient when passing those groups in `i`. When `i` is a `data.table`, `DT[i,j]` evaluates `j` for each row of `i`. We call this *by without by* or *grouping by i*. Hence, the self join `DT[data.table(unique(colA)),j]` is identical to `DT[,j,by=colA]`.
	Advanced: When grouping by `by` or by `i`, 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 length `.N` holding the row locations in `x` for this group. 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* just as `by` but with an additional `setkey()` on the `by` columns of the result, for convenience. Not to be confused with a *keyed by* as defined above.
`with`	By default `with=TRUE` and `j` is evaluated within the frame of `x`. The column names can be used as variables. When `with=FALSE`, `j` is a vector of names or positions to select.
`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 `max(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")`.
`rolltolast`	Deprecated. Setting `rolltolast=TRUE` is converted to `roll=TRUE;rollends=FALSE` for backwards compatibility.

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

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"]                 # vector scan (slow)

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)]      # j for two groups

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

DT[X]                      # join
DT[X,sum(v)]               # 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]           # 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[J("a")]                 # same. J() stands for Join, an alias for list()
DT[list("a")]              # same
DT[.("a")]                 # same. In the style of package plyr.
```

```
DT[J("a",3)]               # join to 2 columns
DT[.("a",3)]               # same
DT[J("a",3:6)]             # join 4 rows (2 missing)
DT[J("a",3:6),nomatch=0]   # remove missing
DT[J("a",3:6),roll=TRUE]   # rolling join (locf)

DT[,sum(v),by=list(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=list(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[!J("a")]                # not join
DT[!"a"]                   # same
DT[!2:4]                   # all rows other than 2:4
DT[x!="b" | y!=3]          # multiple vector scanning approach, slow
DT[!J("b",3)]              # same result but much faster


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

## Not run:
vignette("datatable-intro")
vignette("datatable-faq")
vignette("datatable-timings")

test.data.table()          # over 700 low level tests

update.packages()          # keep up to date

## End(Not run)
```

`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))
```

`duplicated` *Determine Duplicate Rows*

Description

`duplicated` returns a logical vector indicating which rows of a `data.table` have duplicate rows (by key).

`unique` returns a data table with duplicated rows (by key) removed, or (when no key) duplicated rows by all columns removed.

Usage

```
## S3 method for class 'data.table'
duplicated(x, incomparables=FALSE,
                                tolerance=.Machine$double.eps ^ 0.5,
                                by=key(x), ...)

## S3 method for class 'data.table'
unique(x, incomparables=FALSE,
                            tolerance=.Machine$double.eps ^ 0.5,
                            by=key(x), ...)
```

Arguments

`x`	A data.table.
`...`	Not used at this time.
`incomparables`	Not used. Here for S3 method consistency.
`tolerance`	Double precision values are considered equal if they are within this tolerance. Same default as `all.equal`.
`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. A `NULL` or `FALSE` value 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 considred. Unlike `unique.data.frame`, `paste` is not used to ensure equality of floating point data. This is done directly (for speed) whilst still respecting tolerance in the same spirit as `all.equal`.

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 for `unique` and `duplicated`, set by to `NULL` or `FALSE`.

Value

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

`unique` returns a data table with duplicated rows removed.

See Also

`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
```

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.

This function is still under development. For example, dates are read as character (they can be converted afterwards using the excellent `fasttime` package or standard base functions) and embedded quotes (`"\""` and `""""`) have problems. There are other known issues that haven't been fixed and features not yet implemented. But, you may find it works in many cases. Please report problems to datatable-help or Stack Overflow's data.table tag.

Not for production use yet. Not because it's unstable in the sense that it crashes or is buggy (your testing will show whether it is stable in your cases or not) but because `fread`'s arguments and behaviour is likely to change in future; i.e., we expect to make (hopefully minor) non-backwards-compatible changes. Why has it been released to CRAN then? Because a maintenance release was asked for by CRAN maintainers to comply with new stricter tests in R-devel, and a few Bioconductor packages depend on data.table and Bioconductor requires packages to pass R-devel checks. It was quicker to leave `fread` in and write these paragraphs, than take `fread` out.

Usage

```
fread(input, sep="auto", sep2="auto", nrows=-1L, header="auto", na.strings="NA",
stringsAsFactors=FALSE, verbose=FALSE, autostart=30L, skip=-1L, select=NULL,
colClasses=NULL, integer64=getOption("datatable.integer64"))
```

Arguments

`input`	Either the file name to read (containing no \n character) 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 to convert to `NA_character_`. By default for columns read as type character `",,"` is read as a blank string (`""`) and `",NA,"` is read as `NA_character_`. Typical alternatives might be `na.strings=NULL` or perhaps `na.strings=c("NA","N/A","")`.
`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`	Not yet implemented. Vector of column names or positions to keep. Or, use `NULL` in colClasses to drop specified columns (by number if type list is used).
`colClasses`	A character vector of classes (named or unnamed), as read.csv. Or, type list enables setting ranges of columns by numeric position. 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`	Not yet implemented. "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, possibly with loss of precision. Or, "character".

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).

`character` columns can be quoted (`...,2,"Joe Bloggs",3.14,...`) or not quoted (`...,2,Joe Bloggs,3.14,...`). Spaces and other whitepace (other than `sep` and `\n`) may appear in an unquoted character field, provided the field doesn't contain `sep` itself. Therefore quoting character values is only required if `sep` itself appears in the string value. Quoting may also be used to signify that numeric data should be read as text (or that can be achieved by specifying the column type via `colClasses`). Field quoting is automatically detected and no arguments are needed to control it.

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`.

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.

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.

Furthermore, these few features are for fostering friendliness. Facilitated by a fair farthingsworth of (far from flaky, flawed or fatuous) finagling. Finally, it's frustrating to forget but fear not fine friends, fortunately the (free) fread function's first facet is f; for fast, friendly, file or finagle.

Value

A `data.table`.

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

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")

# 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
")


## End(Not run)
```

IDateTime *Integer based date class*

Description

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

Usage

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

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

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

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)  # 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?

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
```

---

last — *Last item of an object*

Description

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

Usage

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

Arguments

`x`	A vector, list or data.table. Otherwise S3 method is dispatched, for compatibility with `xts::last`.
`...`	If any arguments other than `x` are supplied, such as `n` or `keep` regardless of `x`'s type, then S3 dispatch is deployed.

Value

The last item of a vector or list. If `x` is a `data.table`, the last row as a one row `data.table`. Otherwise, whatever `xts::last` returns.

See Also

NROW

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"]
```

merge *Merge Two Data Tables*

Description

Relatively quick merge of two `data.tables` based on common key columns (by default).

This `merge` method for `data.table` is meant to act very similarly to the `merge` method for `data.frame`, with the major exception being that the default columns used to merge two `data.table` inputs are the shared key columns rather than the shared columns with the same names.

For a more `data.table`-centric (and faster) way of merging two `data.tables`, see `[.data.table`; e.g., `x[y, ...]`. In recent versions, however, `merge()` is much closer to the speed of `x[y, ...]`. See FAQ 1.12 for a detailed comparison of `merge` and `x[y, ...]`.

Note that `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. Typing `?merge` at the prompt should present a choice of two links: the help pages for each of these `merge` methods. You don't need to use the full name of the method although you may if you wish; i.e., `merge(DT1,DT2)` is idiomatic R but you can bypass method dispatch by going direct if you wish: `merge.data.table(DT1,DT2)`.

Usage

```
## S3 method for class 'data.table'
merge(x, y, by = NULL, all = FALSE, all.x = all, all.y = all, 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`.
`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.
`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

Note that if the specified columns in `by` is not the key (or head of the key) of `x` or `y`, then a copy is first rekeyed prior to performing the merge. This might make this function perform slower than you are expecting. When secondary keys are implemented in future we expect performance in this case to improve.

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)
```

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

Description

Same as `do.call("rbind",l)`, but much faster.

Usage

```
rbindlist(l)
```

Arguments

`l`	A list of `data.table`, `data.frame` or `list` objects.

Details

Each item of `l` may be either `NULL` (skipped), an empty object (0 rows) (skipped), or, have the same number of columns as the first non empty item. All items do not have to be the same type; e.g, a `data.table` may be bound with a `list`. The column types of the result are taken from the first non-empty item. If subsequent non-empty items have columns that mismatch in type, they are coerced to the first non-empty item's column types.

Value

An unkeyed `data.table` containing a concatenation of all the items passed in.

See Also

`data.table`

Examples

```
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)
```

setkey *Create key on a data table*

Description

`setkey()` sorts a `data.table` and marks it as 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*. No copy is made at all, other than temporary working memory as large as one column.

All `set*` functions similarly change their input by reference with no copy at all, and are documented here. Other than `set()`, which is documented in `:=`.

Usage

```
setkey(x, ..., verbose=getOption("datatable.verbose"))
setkeyv(x, cols, verbose=getOption("datatable.verbose"))
key(x)
haskey(x)
copy(x)
setattr(x,name,value)
setnames(x,old,new)
setcolorder(x,neworder)
key(x) <- value   #  DEPRECATED, please use setkey or setkeyv instead.
```

Arguments

`x`	A `data.table`. Other than `setattr` which accepts any input; e.g, columns of a `data.frame` or `data.table`, and `setnames` which accepts `data.frame`, too.
`...`	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. In `setattr`, the value to assign to the attribute or `NULL` removes the attribute, if present.
`name`	The character attribute name.
`verbose`	Output status and information.
`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`.
`neworder`	Character vector of the new column name ordering. May also be column numbers.

Details

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. That logic is repeated column by column.

The sort is *stable*; i.e., the order of ties (if any) is preserved.

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 setattr() rather than attr(x,name)<-value, setnames() rather than names(x)<-value or colnames(x)<-value, and setcolorder() rather than DT<-DT[,neworder,with=FALSE]. In particular, setattr() is useful in many situations to set attributes by reference and can be used on any object or part of an object, not just data.tables.

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, and why old in setnames() is recommended to be 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 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.

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

See Also

data.table, tables, J, sort.list, copy, :=

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

DF = data.frame(a=1:2,b=3:4)       # base data.frame to demo copies, as of R 2.15.1
try(tracemem(DF))                  # try() for R sessions opted out of memory profiling
colnames(DF)[1] <- "A"             # 4 copies of entire object
names(DF)[1] <- "A"                # 3 copies of entire object
names(DF) <- c("A", "b")           # 1 copy of entire object
`names<-`(DF,c("A","b"))           # 1 copy of entire object

# What if DF is large, say 10GB in RAM. Copy 10GB, even once, just to change a column name?

DT = data.table(a=1:2,b=3:4,c=5:6)
try(tracemem(DT))
setnames(DT,"b","B")               # by name; no match() needed
setnames(DT,3,"C")                 # by position
setnames(DT,2:3,c("D","E"))        # multiple
setnames(DT,c("a","E"),c("A","F")) # multiple by name
setnames(DT,c("X","Y","Z"))        # replace all

# And, no copy of DT was made by setnames() at all.
```

subset.data.table *Subsetting data.tables*

Description

Retruns 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)
```

Arguments

`verbose`	If TRUE sets datatable.verbose to TRUE for the duration of the tests.

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
```

`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
```

Index