Package 'ETLUtils'

May 7, 2026

Title Utility Functions to Execute Standard Extract/Transform/Load Operations (using Package 'ff') on Large Data

Type Package

LazyLoad yes

Description Provides functions to facilitate the use of the 'ff' package in interaction with big data in 'SQL' databases (e.g. in 'Oracle', 'MySQL', 'PostgreSQL', 'Hive') by allowing easy importing directly into 'ffdf' objects using 'DBI', 'RODBC' and 'RJDBC'. Also contains some basic utility functions to do fast left outer join merging based on 'match', factorisation of data and a basic function for re-coding vectors.

Version 1.6

License GPL-2

URL https://github.com/jwijffels/ETLUtils

Depends ff (>= 4.0.0)

Imports bit (>= 4.0.0)

Suggests RSQLite, zoo, DBI, RODBC, RJDBC

RoxygenNote 7.3.2

NeedsCompilation no

Author Jan Wijffels [aut, cre]

Maintainer Jan Wijffels <jwijffels@bnosac.be>

Repository CRAN

Date/Publication 2025-11-26 22:50:02 UTC

Contents

`ETLUtils-package` *Extra utility functions to execute standard ETL operations on large data*

Description

Provides functions to load bigdata (e.g. from Oracle) directly into `ffdf` objects using DBI and some utility functions like recoding and matchmerge which does fast left outer join merging based on `match`.

Author(s)

Jan Wijffels <jwijffels@bnosac.be>

See Also

Useful links:

- https://github.com/jwijffels/ETLUtils

Examples

```
# See the specified functions in the package
```

`factorise` *Put character vectors, columns of a data.frame or list elements as factor*

Description

Put character vectors, columns of a data.frame or list elements as factor if they are character strings or optionally if they are logicals

Usage

```
factorise(x, logicals = FALSE, ...)

## Default S3 method:
factorise(x, logicals = FALSE, ...)

## S3 method for class 'character'
factorise(x, logicals = FALSE, ...)

## S3 method for class 'data.frame'
factorise(x, logicals = FALSE, ...)

## S3 method for class 'list'
factorise(x, logicals = FALSE, ...)
```

Arguments

x	a character vector, a data.frame or a list
logicals	logical indicating if logical vectors should also be converted to factors. Defaults to FALSE.
...	optional arguments passed on to the methods

Value

The updated x vector/data.frame or list where the character vectors or optionally logical elements are converted to factors

See Also

as.factor, factor

Examples

```
x <- data.frame(x = 1:4, y = LETTERS[1:4], b = c(TRUE, FALSE, NA, TRUE), stringsAsFactors=FALSE)
str(factorise(x))
str(factorise(x, logicals = TRUE))
str(factorise(list(a = LETTERS, b = 1:10, c = pi, d = list(x = x))))
```

---

matchmerge *Merge two data frames (fast) by common columns by performing a left (outer) join or an inner join.*

Description

Merge two data frames (fast) by common columns by performing a left (outer) join or an inner join. The data frames are merged on the columns given by by.x and by.y. Columns can be specified only by name. This differs from the merge function from the base package in that merging is done based on 1 column key only. If more than one column is supplied in by.x and by.y, these columns will be concatenated together to form 1 key which will be used to match. Alternatively, by.x and by.y can be 2 vectors of length NROW(x) which will be used as keys.

Usage

```
matchmerge(
  x,
  y,
  by.x,
  by.y,
  all.x = FALSE,
  by.iskey = FALSE,
  suffix = ".y",
  add.columns = colnames(y),
  check.duplicates = TRUE,
  trace = FALSE
)
```

Arguments

x	the left hand side data frame to merge
y	the right hand side data frame to merge or a vector in which case you always need to supply by.y as a vector, make sure by.iskey is set to TRUE and provide in add.columns the column name for which y will be relabelled to in the joined data frame (see the example).
by.x	either the name of 1 column in x or a character vector of length NROW(x) which will be used as key to merge the 2 data frames
by.y	either the name of 1 column in y or a character vector of length NROW(x) which will be used as key to merge the 2 data frames. Duplicate values in by.y are not allowed.
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 NAs 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. The default value corresponds to an inner join. If TRUE is supplied, this corresponds to a left (outer) join.
by.iskey	Logical, indicating that the by.x and the by.y inputs are vectors of length NROW(x) and NROW(y) instead of column names in x and y. If this is FALSE, the input columns will be pasted together to create a key to merge upon. Otherwise, the function will use the by.x and by.y vectors directly as matching key. Defaults to FALSE indicating the by.x and by.y are column names in x and y.

suffix	a character string to be used for duplicate column names in x and y to make the y columns unique.
add.columns	character vector of column names in y to merge to the x data frame. Defaults to all columns in y.
check.duplicates	checks if by.y contains duplicates which is not allowed. Defaults to TRUE.
trace	logical, indicating to print some informative messages about the progress

Details

The rows in the right hand side data frame that match on the specific key are extracted, and joined together with the left hand side data frame.

Merging is done based on the match function on the key value. This makes the function a lot faster when compared to applying merge, especially for large data frames (see the example). And also the memory consumption is a lot smaller.

In SQL database terminology, the default value of all.x = FALSE gives a natural join, a special case of an inner join. Specifying all.x = FALSE gives a left (outer) join. Right (outer) join or (full) outer join are not provided in this function.

Value

data frame with x joined with y based on the supplied columns. The output columns are the columns in x followed by the extra columns in y.

See Also

cbind, match, merge

Examples

```
left <- data.frame(idlhs = c(1:4, 3:5), a = LETTERS[1:7], stringsAsFactors = FALSE)
right <- data.frame(idrhs = c(1:4), b = LETTERS[8:11], stringsAsFactors = FALSE)
## Inner join
matchmerge(x=left, y=right, by.x = "idlhs", by.y = "idrhs")

## Left outer join in 2 ways
matchmerge(x=left, y=right, by.x = "idlhs", by.y = "idrhs", all.x=TRUE)
matchmerge(x=left, y=right, by.x = left$idlhs, by.y = right$idrhs, all.x=TRUE, by.iskey=TRUE)

## Show usage when y is just a vector instead of a data.frame
matchmerge(x=left, y=right$b, by.x = left$idlhs, by.y = right$idrhs, all.x=TRUE,
by.iskey=TRUE, add.columns="b.renamed")

## Show speedup difference with merge
## Not run:
size <- 100000
dimension <- seq(Sys.Date(), Sys.Date()+10, by = "day")
```

```
left <- data.frame(date = rep(dimension, size), sales = rnorm(size))
right <- data.frame(date = dimension, feature = dimension-7, feature = dimension-14)
dim(left)
dim(right)
print(system.time(merge(left, right, by.x="date", by.y="date", all.x=TRUE, all.y=FALSE)))
print(system.time(matchmerge(left, right, by.x="date", by.y="date", all.x=TRUE, by.iskey=FALSE)))

## End(Not run)
## Show example usage
products <- expand.grid(product = c("Pepsi", "Coca Cola"), type = c("Can","Bottle"),
size = c("6Ml","8Ml"), distributor = c("Distri X","Distri Y"), salesperson = c("Mr X","Mr Y"),
stringsAsFactors=FALSE)
products <- products[!duplicated(products[, c("product","type","size")]), ]
products$key <- paste(products$product, products$type, products$size, sep=".")
sales <- expand.grid(item = unique(products$key), sales = rnorm(10000, mean = 100))
str(products)
str(sales)
info <- matchmerge(x=sales, y=products,
  by.x=sales$item, by.y=products$key, all.x=TRUE, by.iskey=TRUE,
  add.columns=c("size","distributor"), check.duplicates=FALSE)
str(info)
tapply(info$sales, info$distributor, FUN=sum)
```

`naLOCFPlusone` *Performs NA replacement by last observation carried forward but adds 1 to the last observation carried forward.*

Description

Performs NA replacement by last observation carried forward but adds 1 to the last observation carried forward.

Usage

```
naLOCFPlusone(x)
```

Arguments

x	a numeric vector

Value

a vector where NA's are replaced with the LOCF + 1

See Also

`na.locf`

Examples

```
require(zoo)
x <- c(2,NA,NA,4,5,2,NA)
naLOCFPlusone(x)
```

`read.dbi.ffdf` *Read data from a DBI connection into an ffdf.*

Description

Read data from a DBI connection into an `ffdf`. This can for example be used to import large datasets from Oracle, SQLite, MySQL, PostgreSQL, Hive or other SQL databases into R.

Usage

```
read.dbi.ffdf(
  query = NULL,
  dbConnect.args = list(drv = NULL, dbname = NULL, username = "", password = ""),
  dbSendQuery.args = list(),
  dbFetch.args = list(),
  x = NULL,
  nrows = -1,
  first.rows = NULL,
  next.rows = NULL,
  levels = NULL,
  appendLevels = TRUE,
  asffdf_args = list(),
  BATCHBYTES = getOption("ffbatchbytes"),
  VERBOSE = FALSE,
  colClasses = NULL,
  transFUN = NULL,
  ...
)
```

Arguments

`query`	the SQL query to execute on the DBI connection
`dbConnect.args`	a list of arguments to pass to DBI's `dbConnect` (like drv, dbname, username, password). See the examples.
`dbSendQuery.args`	a list containing database-specific parameters which will be passed to to pass to `dbSendQuery`. Defaults to an empty list.
`dbFetch.args`	a list containing optional database-specific parameters which will be passed to to pass to `dbFetch`. Defaults to an empty list.

x	NULL or an optional ffdf object to which the read records are appended. See documentation in read.table.ffdf for more details and the example below.
nrows	Number of rows to read from the query resultset. Default value of -1 reads in all rows.
first.rows	chunk size (rows) to read for first chunk from the query resultset
next.rows	chunk size (rows) to read sequentially for subsequent chunks from the query resultset. Currently, this must be specified.
levels	optional specification of factor levels. A list with as names the names the columns of the data.frame fetched in the first.rows, containing levels of the factors.
appendLevels	logical. A vector of permissions to expand levels for factor columns. See documentation in read.table.ffdf for more details.
asffdf_args	further arguments passed to as.ffdf (ignored if 'x' gives an ffdf object)
BATCHBYTES	integer: bytes allowed for the size of the data.frame storing the result of reading one chunk. See documentation in read.table.ffdf for more details.
VERBOSE	logical: TRUE to verbose timings for each processed chunk (default FALSE).
colClasses	See documentation in read.table.ffdf
transFUN	function applied to the data frame after each chunk is retreived by dbFetch
...	optional parameters passed on to transFUN

Details

Opens up the DBI connection using DBI::dbConnect, sends the query using DBI::dbSendQuery and DBI::dbFetch-es the results in batches of next.rows rows. Heavily borrowed from read.table.ffdf

Value

An ffdf object unless the query returns zero records in which case the function will return the data.frame returned by dbFetch and possibly transFUN.

See Also

read.table.ffdf, read.odbc.ffdf

Examples

```
require(ff)
require(DBI)

##
## Example query using data in sqlite
##
require(RSQLite)
dbfile <- system.file("smalldb.sqlite3", package="ETLUtils")
drv <- dbDriver("SQLite")
query <- "select * from testdata limit 10000"
x <- read.dbi.ffdf(query = query, dbConnect.args = list(drv = drv, dbname = dbfile),
```

```
first.rows = 100, next.rows = 1000, VERBOSE=TRUE)
class(x)
x[1:10, ]

## show it is the same as getting the data directly using RSQLite
## apart from characters which are factors in ffdf objects
directly <- dbGetQuery(dbConnect(drv = drv, dbname = dbfile), query)
directly <- as.data.frame(as.list(directly), stringsAsFactors=TRUE)
all.equal(x[,], directly)

## show how to use the transFUN argument to transform the data before saving into the ffdf
## and shows the use of the levels argument
query <- "select * from testdata limit 10"
x <- read.dbi.ffdf(query = query, dbConnect.args = list(drv = drv, dbname = dbfile),
first.rows = 100, next.rows = 1000, VERBOSE=TRUE, levels = list(a = rev(LETTERS)),
transFUN = function(x, subtractdays){
x$b <- as.Date(x$b)
x$b.subtractdaysago <- x$b - subtractdays
x
}, subtractdays=7)
class(x)
x[1:10, ]
## remark that the levels of column a are reversed due to specifying the levels argument correctly
levels(x$a)

## show how to append data to an existing ffdf object
transformexample <- function(x, subtractdays){
x$b <- as.Date(x$b)
x$b.subtractdaysago <- x$b - subtractdays
x
}
dim(x)
x[,]
combined <- read.dbi.ffdf(query = query,
 dbConnect.args = list(drv = drv, dbname = dbfile),
 first.rows = 100, next.rows = 1000, x = x, VERBOSE=TRUE,
 transFUN = transformexample, subtractdays=1000)
dim(combined)
combined[,]

##
## Example query using ROracle. Do try this at home with some larger data :)
##
## Not run:
require(ROracle)
query <- "select OWNER, TABLE_NAME, TABLESPACE_NAME, NUM_ROWS, LAST_ANALYZED from all_all_tables"
x <- read.dbi.ffdf(query=query,
dbConnect.args = list(drv = dbDriver("Oracle"),
user = "YourUser", password = "YourPassword", dbname = "Mydatabase"),
first.rows = 100, next.rows = 50000, nrows = -1, VERBOSE=TRUE)

## End(Not run)
```

`read.jdbc.ffdf` *Read data from a JDBC connection into an ffdf.*

Description

Read data from a JDBC connection into an `ffdf`. This can for example be used to import large datasets from Oracle, SQLite, MySQL, PostgreSQL, Hive or other SQL databases into R.

Usage

```
read.jdbc.ffdf(
  query = NULL,
  dbConnect.args = list(drv = NULL, dbname = NULL, username = "", password = ""),
  dbSendQuery.args = list(),
  dbFetch.args = list(),
  x = NULL,
  nrows = -1,
  first.rows = NULL,
  next.rows = NULL,
  levels = NULL,
  appendLevels = TRUE,
  asffdf_args = list(),
  BATCHBYTES = getOption("ffbatchbytes"),
  VERBOSE = FALSE,
  colClasses = NULL,
  transFUN = NULL,
  ...
)
```

Arguments

`query`	the SQL query to execute on the JDBC connection
`dbConnect.args`	a list of arguments to pass to JDBC's `RJDBC::dbConnect` (like drv, dbname, username, password). See the examples.
`dbSendQuery.args`	a list containing database-specific parameters which will be passed to to pass to `RJDBC::dbSendQuery`. Defaults to an empty list.
`dbFetch.args`	a list containing optional database-specific parameters which will be passed to to pass to `RJDBC::dbFetch`. Defaults to an empty list.
`x`	NULL or an optional ffdf object to which the read records are appended. See documentation in read.table.ffdf for more details and the example below.
`nrows`	Number of rows to read from the query resultset. Default value of -1 reads in all rows.
`first.rows`	chunk size (rows) to read for first chunk from the query resultset

next.rows	chunk size (rows) to read sequentially for subsequent chunks from the query resultset. Currently, this must be specified.
levels	optional specification of factor levels. A list with as names the names the columns of the data.frame fetched in the first.rows, containing levels of the factors.
appendLevels	logical. A vector of permissions to expand levels for factor columns. See documentation in read.table.ffdf for more details.
asffdf_args	further arguments passed to as.ffdf (ignored if 'x' gives an ffdf object)
BATCHBYTES	integer: bytes allowed for the size of the data.frame storing the result of reading one chunk. See documentation in read.table.ffdf for more details.
VERBOSE	logical: TRUE to verbose timings for each processed chunk (default FALSE).
colClasses	See documentation in read.table.ffdf
transFUN	function applied to the data frame after each chunk is retreived by RJDBC::dbFetch
...	optional parameters passed on to transFUN

Details

Opens up the JDBC connection using RJDBC::dbConnect, sends the query using RJDBC::dbSendQuery and RJDBC::dbFetch-es the results in batches of next.rows rows. Heavily borrowed from read.table.ffdf

Value

An ffdf object unless the query returns zero records in which case the function will return the data.frame returned by RJDBC::dbFetch and possibly transFUN.

See Also

read.table.ffdf, read.jdbc.ffdf

Examples

```
## Not run:
require(ff)
require(DBI)

##
## Example query using data in sqlite
##
require(RSQLite)
dbfile <- system.file("smalldb.sqlite3", package="ETLUtils")
drv <- JDBC(driverClass = "org.sqlite.JDBC", classPath = "/usr/local/lib/sqlite-jdbc-3.7.2.jar")
query <- "select * from testdata limit 10000"
x <- read.jdbc.ffdf(query = query,
 dbConnect.args = list(drv = drv, url = sprintf("jdbc:sqlite:%s", dbfile)),
 first.rows = 100, next.rows = 1000, VERBOSE=TRUE)
class(x)
x[1:10, ]

## End(Not run)
```

`read.odbc.ffdf` *Read data from a ODBC connection into an ffdf.*

Description

Read data from a ODBC connection into an `ffdf`. This can for example be used to import large datasets from Oracle, SQLite, MySQL, PostgreSQL, Hive or other SQL databases into R.

Usage

```
read.odbc.ffdf(
  query = NULL,
  odbcConnect.args = list(dsn = NULL, uid = "", pwd = ""),
  odbcDriverConnect.args = list(connection = ""),
  odbcQuery.args = list(),
  sqlGetResults.args = list(),
  x = NULL,
  nrows = -1,
  first.rows = NULL,
  next.rows = NULL,
  levels = NULL,
  appendLevels = TRUE,
  asffdf_args = list(),
  BATCHBYTES = getOption("ffbatchbytes"),
  VERBOSE = FALSE,
  colClasses = NULL,
  transFUN = NULL,
  ...
)
```

Arguments

`query`	the SQL query to execute on the ODBC connection
`odbcConnect.args`	a list of arguments to pass to ODBC's `odbcConnect` (like dsn, uid, pwd). See the examples.
`odbcDriverConnect.args`	a list of arguments to pass to ODBC's `odbcDriverConnect` (like connection). If you want to connect using odbcDriverConnect instead of odbcConnect.
`odbcQuery.args`	a list of arguments to pass to ODBC's `odbcQuery`, like rows_at_time. Defaults to an empty list.
`sqlGetResults.args`	a list containing optional parameters which will be passed to `sqlGetResults`. Defaults to an empty list. The max parameter will be overwritten with first.rows and next.rows when importing in batches.

x	NULL or an optional ffdf object to which the read records are appended. See documentation in read.table.ffdf for more details and the example below.
nrows	Number of rows to read from the query resultset. Default value of -1 reads in all rows.
first.rows	chunk size (rows) to read for first chunk from the query resultset
next.rows	chunk size (rows) to read sequentially for subsequent chunks from the query resultset. Currently, this must be specified.
levels	optional specification of factor levels. A list with as names the names the columns of the data.frame fetched in the first.rows, containing levels of the factors.
appendLevels	logical. A vector of permissions to expand levels for factor columns. See documentation in read.table.ffdf for more details.
asffdf_args	further arguments passed to as.ffdf (ignored if 'x' gives an ffdf object)
BATCHBYTES	integer: bytes allowed for the size of the data.frame storing the result of reading one chunk. See documentation in read.table.ffdf for more details.
VERBOSE	logical: TRUE to verbose timings for each processed chunk (default FALSE).
colClasses	See documentation in read.table.ffdf
transFUN	function applied to the data frame after each chunk is retreived by sqlGetResults
...	optional parameters passed on to transFUN

Details

Opens up the ODBC connection using RODBC::odbcConnect or RODBC::odbcDriverConnect, sends the query using RODBC::odbcQuery and retrieves the results in batches of next.rows rows using RODBC::sqlGetResults. Heavily borrowed from read.table.ffdf

Value

An ffdf object unless the query returns zero records in which case the function will return the data.frame returned by sqlGetResults and possibly transFUN.

See Also

read.table.ffdf, read.dbi.ffdf

Examples

```
##
## Using the sqlite database (smalldb.sqlite3) in the /inst folder of the package
## set up the sqlite ODBC driver (www.stats.ox.ac.uk/pub/bdr/RODBC-manual.pd)
## and call it 'smalltestsqlitedb'
##
## Not run:
require(RODBC)
x <- read.odbc.ffdf(
query = "select * from testdata limit 10000",
odbcConnect.args = list(
```

```
 dsn="smalltestsqlitedb", uid = "", pwd = "",
 believeNRows = FALSE, rows_at_time = 1),
nrows = -1,
first.rows = 100, next.rows = 1000, VERBOSE = TRUE)

## End(Not run)
```

---

recoder *Recodes the values of a character vector*

Description

Recodes the values of a character vector

Usage

```
recoder(x, from = c(), to = c())
```

Arguments

x	character vector
from	character vector with old values
to	character vector with new values

Value

x where from values are recoded to the supplied to values

See Also

match

Examples

```
recoder(x=append(LETTERS, NA, 5), from = c("A","B"), to = c("a.123","b.123"))
```

renameColumns *Renames variables in a data frame.*

Description

Renames variables in a data frame.

Usage

```
renameColumns(x, from = "", to = "")
```

Arguments

x	data frame to be modified.
from	character vector containing the current names of each variable to be renamed.
to	character vector containing the new names of each variable to be renamed.

Value

The updated data frame x where the variables listed in from are renamed to the corresponding to column names.

See Also

colnames, recoder

Examples

```
x <- data.frame(x = 1:4, y = LETTERS[1:4])
renameColumns(x, from = c("x","y"), to = c("digits","letters"))
```

write.dbi.ffdf *Write ffdf data to a database table by using a DBI connection.*

Description

Write ffdf data to a database table by using a DBI connection. This can for example be used to store large ffdf datasets from R in Oracle, SQLite, MySQL, PostgreSQL, Hive or other SQL databases.
Mark that for very large datasets, these SQL databases might have tools to speed up by bulk loading. You might also consider that as an alternative to using this procedure.

Usage

```
write.dbi.ffdf(
  x,
  name,
  dbConnect.args = list(drv = NULL, dbname = NULL, username = "", password = ""),
  RECORDBYTES = sum(.rambytes[vmode(x)]),
  BATCHBYTES = getOption("ffbatchbytes"),
  by = NULL,
  VERBOSE = FALSE,
  ...
)
```

Arguments

x	the ffdf to write to the database
name	character string with the name of the table to store the data in. Passed on to dbWriteTable.
dbConnect.args	a list of arguments to pass to DBI's dbConnect (like drv, dbname, username, password). See the examples.
RECORDBYTES	optional integer scalar representing the bytes needed to process a single row of the ffdf
BATCHBYTES	integer: bytes allowed for the size of the data.frame storing the result of reading one chunk. See documentation in read.table.ffdf for more details.
by	integer passed on to chunk indicating to write to the database in chunks of this size. Overwrites the behaviour of BATCHBYTES and RECORDBYTES.
VERBOSE	logical: TRUE to verbose timings for each processed chunk (default FALSE).
...	optional parameters passed on to dbWriteTable

Details

Opens up the DBI connection using DBI::dbConnect, writes data to the SQL table using DBI::dbWriteTable by extracting the data in batches from the ffdf and appending them to the table.

Value

invisible()

See Also

dbWriteTable, chunk

Examples

```
require(ff)
require(DBI)

##
```

```
## Example query using data in sqlite
##
require(RSQLite)
dbfile <- system.file("smalldb.sqlite3", package="ETLUtils")
drv <- dbDriver("SQLite")
query <- "select * from testdata limit 10000"
x <- read.dbi.ffdf(query = query, dbConnect.args = list(drv = drv, dbname = dbfile),
first.rows = 100, next.rows = 1000, VERBOSE=TRUE)

## copy db in package folder to temp folder as CRAN does not allow writing in package dirs
dbfile <- tempfile(fileext = ".sqlite3")
file.copy(from = system.file("smalldb.sqlite3", package="ETLUtils"), to = dbfile)
Sys.chmod(dbfile, mode = "777")
write.dbi.ffdf(x = x, name = "helloworld", row.names = FALSE, overwrite = TRUE,
  dbConnect.args = list(drv = drv, dbname = dbfile),
  by = 1000, VERBOSE=TRUE)
test <- read.dbi.ffdf(query = "select * from helloworld",
  dbConnect.args = list(drv = drv, dbname = dbfile))

## clean up for CRAN
file.remove(dbfile)
## Not run:
require(ROracle)
write.dbi.ffdf(x = x, name = "hellooracle", row.names = FALSE, overwrite = TRUE,
  dbConnect.args = list(drv = dbDriver("Oracle"),
                    user = "YourUser", password = "YourPassword", dbname = "Mydatabase"),
  VERBOSE=TRUE)

## End(Not run)
```

`write.jdbc.ffdf` *Write ffdf data to a database table by using a JDBC connection.*

Description

Write ffdf data to a database table by using a JDBC connection. This can for example be used to store large ffdf datasets from R in Oracle, SQLite, MySQL, PostgreSQL, Hive or other SQL databases.
Mark that for very large datasets, these SQL databases might have tools to speed up by bulk loading. You might also consider that as an alternative to using this procedure.

Usage

```
write.jdbc.ffdf(
  x,
  name,
  dbConnect.args = list(drv = NULL, dbname = NULL, username = "", password = ""),
  RECORDBYTES = sum(.rambytes[vmode(x)]),
  BATCHBYTES = getOption("ffbatchbytes"),
```

```
  by = NULL,
  VERBOSE = FALSE,
  ...
)
```

Arguments

x	the ffdf to write to the database
name	character string with the name of the table to store the data in. Passed on to dbWriteTable.
dbConnect.args	a list of arguments to pass to JDBC's RJDBC::dbConnect (like drv, dbname, username, password). See the examples.
RECORDBYTES	optional integer scalar representing the bytes needed to process a single row of the ffdf
BATCHBYTES	integer: bytes allowed for the size of the data.frame storing the result of reading one chunk. See documentation in read.table.ffdf for more details.
by	integer passed on to chunk indicating to write to the database in chunks of this size. Overwrites the behaviour of BATCHBYTES and RECORDBYTES.
VERBOSE	logical: TRUE to verbose timings for each processed chunk (default FALSE).
...	optional parameters passed on to dbWriteTable

Details

Opens up the JDBC connection using RJDBC::dbConnect, writes data to the SQL table using RJDBC::dbWriteTable by extracting the data in batches from the ffdf and appending them to the table.

Value

invisible()

See Also

JDBCConnection-methods, chunk

Examples

```
## Not run:
require(ff)
require(DBI)

##
## Example query using data in sqlite
##
require(RJDBC)
dbfile <- system.file("smalldb.sqlite3", package="ETLUtils")
drv <- JDBC(driverClass = "org.sqlite.JDBC", classPath = "/usr/local/lib/sqlite-jdbc-3.7.2.jar")
query <- "select * from testdata limit 10000"
```

```
x <- read.jdbc.ffdf(query = query,
 dbConnect.args = list(drv = drv, url = sprintf("jdbc:sqlite:%s", dbfile)),
 first.rows = 100, next.rows = 1000, VERBOSE=TRUE)

write.jdbc.ffdf(x = x, name = "helloworld", row.names = FALSE, overwrite = TRUE,
  dbConnect.args = list(drv = drv, url = sprintf("jdbc:sqlite:%s", dbfile)),
  by = 1000, VERBOSE=TRUE)
test <- read.jdbc.ffdf(query = "select * from helloworld",
  dbConnect.args = list(drv = drv, url = sprintf("jdbc:sqlite:%s", dbfile)))

## End(Not run)
```

`write.odbc.ffdf` *Write ffdf data to a database table by using a ODBC connection.*

Description

Write `ffdf` data to a database table by using a ODBC connection. This can for example be used to store large ffdf datasets from R in Oracle, SQLite, MySQL, PostgreSQL, Hive or other SQL databases.
Mark that for very large datasets, these SQL databases might have tools to speed up by bulk loading. You might also consider that as an alternative to using this procedure.

Usage

```
write.odbc.ffdf(
  x,
  tablename,
  odbcConnect.args = list(dsn = NULL, uid = "", pwd = ""),
  RECORDBYTES = sum(.rambytes[vmode(x)]),
  BATCHBYTES = getOption("ffbatchbytes"),
  by = NULL,
  VERBOSE = FALSE,
  ...
)
```

Arguments

`x`	the `ffdf` to write to the database
`tablename`	character string with the name of the table to store the data in. Passed on to `sqlSave`.
`odbcConnect.args`	a list of arguments to pass to ODBC's `odbcConnect` (like dsn, uid, pwd). See the examples.
`RECORDBYTES`	optional integer scalar representing the bytes needed to process a single row of the ffdf

BATCHBYTES	integer: bytes allowed for the size of the data.frame storing the result of reading one chunk. See documentation in read.table.ffdf for more details.
by	integer passed on to chunk indicating to write to the database in chunks of this size. Overwrites the behaviour of BATCHBYTES and RECORDBYTES.
VERBOSE	logical: TRUE to verbose timings for each processed chunk (default FALSE).
...	optional parameters passed on to sqlSave

Details

Opens up the ODBC connection using RODBC::odbcConnect, writes data to the SQL table using RODBC::sqlSave by extracting the data in batches from the ffdf and appending them to the table.

Value

invisible()

See Also

sqlSave, chunk

Examples

```
##
## Using the sqlite database (smalldb.sqlite3) in the /inst folder of the package
## set up the sqlite ODBC driver (www.stats.ox.ac.uk/pub/bdr/RODBC-manual.pd)
## and call it 'smalltestsqlitedb'
##
## Not run:
require(RODBC)
x <- read.odbc.ffdf(
  query = "select * from testdata limit 10000",
  odbcConnect.args = list(
   dsn="smalltestsqlitedb", uid = "", pwd = "",
   believeNRows = FALSE, rows_at_time = 1),
  nrows = -1,
  first.rows = 100, next.rows = 1000, VERBOSE = TRUE)

write.odbc.ffdf(x = x, tablename = "testdata", rownames = FALSE, append = TRUE,
  odbcConnect.args = list(
   dsn="smalltestsqlitedb", uid = "", pwd = "",
   believeNRows = FALSE, rows_at_time = 1),
  by = 1000, VERBOSE=TRUE)

## End(Not run)
```

Index