Package ‘dbi.table’

March 1, 2026

Type Package

Title Database Queries Using 'data.table' Syntax

Version 1.0.7

Depends R (>= 3.6.0)

Imports DBI, bit64, dbplyr, methods, rlang, stringi, utils

Suggests RMariaDB, RPostgres, RSQLite, data.table, duckdb, knitr, rmarkdown, testthat (>= 3.0.0), withr

Description Query database tables over a 'DBI' connection using 'data.table' syntax. Attach database schemas to the search path. Automatically merge using foreign key constraints.

License MPL-2.0

URL `https://github.com/kjellpk/dbi.table`

BugReports `https://github.com/kjellpk/dbi.table/issues`

VignetteBuilder knitr

Encoding UTF-8

Config/testthat/edition 3

RoxygenNote 7.3.3

NeedsCompilation no

Author Kjell P. Konis [aut, cre], Luis Rocha [ctb] (Chinook Database - see example_files/LICENSE)

Maintainer Kjell P. Konis <`kjellk@gmail.com`>

Repository CRAN

Date/Publication 2026-03-01 18:20:03 UTC

Contents

`dbi.table-package` *DBI Table*

Description

A dbi.table is a data structure that describes a SQL query (called the dbi.table's *underlying SQL query*). This query can be manipulated using `data.table`'s `[i, j, by]` syntax.

Usage

```
dbi.table(conn, id, check.names = FALSE, key = NULL, stringsAsFactors = FALSE)

## S3 method for class 'dbi.table'
x[i, j, by, keyby, nomatch = NA, on = NULL]
```

Arguments

`conn`	A `DBIConnection` object, as returned by `dbConnect`. Alternatively, a `dbi.catalog` or a `dbi.table`, in which case the new `dbi.table` will use the connection embedded in the provided object.
`id`	An `Id`, a character string (which will be converted to an `Id` by `Id`), or a `SQL` object (advanced) identifying a database object (e.g., table or view) on conn.
`check.names`	Just as `check.names` in `data.table` and `data.frame`.
`key`	A character vector of one or more column names to set as the resulting `dbi.table`'s key.
`stringsAsFactors`	A logical value (default is `FALSE`). Convert all `character` columns to `factors` when executing the `dbi.table`'s underlying SQL query and retrieving the result set.
`x`	A `dbi.table`.
`i`	A logical expression of the columns of `x`, a `dbi.table`, or a `data.frame`. Use `i` to select a subset of the rows of `x`. Note: unlike `data.table`, `i` *cannot* be a vector. When `i` is a logical expression, the rows where the expression is `TRUE` are returned. If the expression contains a symbol `foo` that is not a column name of

	`x` but that is present in the calling scope, then the value of `foo` will be substituted into the expression if `foo` is a scalar, or if `foo` is a vector and is the right-hand-side argument to `%in%` or `%chin%` (substitution occurs when the extract (`[`) method is evaluated). When `i` inherits from `data.frame`, it is coerced to a `dbi.table`. When `i` is a `dbi.table`, the rows of `x` that match (according to the condition specificed in `on`) the rows of `i` are returned. When `nomatch == NA`, all rows of `i` are returned (right outer join); when `nomatch == NULL`, only the rows of `i` that match a row of `x` are returned (inner join).
`j`	A list of expressions, a literal character vector of column names of `x`, an expression of the form `start_name:end_name`, or a literal numeric vector of integer values indexing the columns of `x`. Use `j` to select (and optionally, transform) the columns of `x`.
`by`	A list of expressions, a literal character vector of column names of `x`, an expression of the form `start_name:end_name`, or a literal numeric vector of integer values indexing the columns of `x`. Use `by` to control grouping when evaluating `j`.
`keyby`	Same as `by`, but additionally sets the key of the resulting `dbi.table` to the columns provided in `by`. May also be `TRUE` or `FALSE` when `by` is provided as an alternative way to accomplish the same operation.
`nomatch`	Either `NA` or `NULL`.
`on`	• An unnamed character vector, e.g., `x[i, on = c("a", "b")]`, used when columns `a` and `b` are common to both `x` and `i`. • Foreign key joins: As a named character vector when the join columns have different names in `x` and `i`. For example, `x[i, on = c(x1 = "i1", x2 = "i2")]` joins `x` and `i` by matching columns `x1` and `x2` in `x` with columns `i1` and `i2` in `i`, respectively. • Foreign key joins can also use the binary operator `==`, e.g., `x[i, on = c("x1 == i1", "x2 == i2")]`. • It is also possible to use `.()` syntax as `x[i, on = .(a, b)]`. • Non-equi joins using binary operators `>=`, `>`, `<=`, `<` are also possible, e.g., `x[i, on = c("x >= a", "y <= b")]`, or `x[i, on = .(x >= a, y <= b)]`.

Value

A `dbi.table`.

Keys

A key marks a `dbi.table` as sorted with an attribute `"sorted"`. The sorted columns are the key. The key can be any number of columns. Unlike `data.table`, the underlying data are not physically sorted, so there is no performance improvement. However, there remain benefits to using keys:

1. The key provides a default order for window queries so that functions like `shift` and `cumsum` give reproducible output.

2. dbi.table's merge method uses a dbi.table's key to determin the default columns to merge on in the same way that data.table's merge method does. Note: if a dbi.table has a foreign key relationship, that will be used to determin the default columns to merge on before the dbi.table's key is considered.

A table's primary key is used as the default key when it can be determined.

Differences vs. data.table Keys

There are a few key differences between dbi.table keys and data.table keys.

1. In data.table, NAs are always first. Some databases (e.g., PostgreSQL) sort NULLs last by default and some databases (e.g., SQLite) sort them first. as.data.frame does not change the order of the result set returned by the database. Note that as.data.table uses the dbi.table's key so that the resulting data.table is sorted in the usual data.table way.
2. The sort is *not* stable: the order of ties may change on subsequent evaluations of the dbi.table's underlying SQL query.

Strict Processing of Keys

By default, when previewing data (dbi.table's print method), the key is not included in the underlying SQL query's ORDER BY clause. However, the result set is sorted locally to resepct the key. This behavior is referred to as a *non-strict* evaluation of the key and the printed output labels the key (non-strict). To override the default behavior for a single preview, call print explicitly and provide the optional argument strict = TRUE. To change the default behavior, set the option dbitable.print.strict to TRUE.

Non-strict evaluation of keys reduces the time taken to retrieve the preview.

See Also

- as.data.frame to retrieve the *results set* as a data.frame,
- csql to see the underlying SQL query.

Examples

```
# open a connection to the Chinook example database using duckdb
duck <- chinook.duckdb()

# create a dbi.table corresponding to the Album table on duck
Album <- dbi.table(duck, DBI::Id(table_name = "Album"))

# the print method displays a 5 row preview
# print(Album)
Album

# 'id' can also be 'SQL'; use the same DBI connection as Album
Genre <- dbi.table(Album, DBI::SQL("chinook_duckdb.main.Genre"))

# use the extract ([...]) method to subset the dbi.table
Album[AlbumId < 5, .(Title, nchar = paste(nchar(Title), "characters"))]

# use csql to see the underlying SQL query
```

```
csql(Album[AlbumId < 5, #WHERE
           .(Title, #SELECT
             nchar = paste(nchar(Title), "characters"))])
```

`as.data.frame` *Coerce to a Data Frame*

Description

Execute a `dbi.table`'s underlying SQL query and return the result set as a `data.frame`. By default, the result set is limited to 10,000 rows. See Details.

Usage

```
## S3 method for class 'dbi.table'
as.data.frame(
  x,
  row.names = NULL,
  optional = FALSE,
  ...,
  n = getOption("dbitable.max.fetch", 10000L)
)
```

Arguments

`x`	a `dbi.table`.
`row.names`	a logical value. This argument is not used.
`optional`	a logical value. This argument is not used.
`...`	additional arguments are ignored.
`n`	an integer value. When nonnegative, the underlying SQL query includes a 'LIMIT `n`' clause and `n` is also passed to `dbFetch`. When negative, the underlying SQL query does not include a LIMIT clause and all rows in the result set are returned.

Details

By default, `as.data.frame` returns up to 10,000 rows (see the `n` argument). To override this limit, either call `as.data.frame` and provide the `n` argument (e.g., `n = -1` to return the entire result set), or set the option `dbitable.max.fetch` to the desired default value of `n`.

Value

a `data.frame`.

See Also

`as.data.frame` (the generic method in the **base** package).

Examples

```
duck <- chinook.duckdb()
Artist <- dbi.table(duck, DBI::Id("Artist"))

as.data.frame(Artist, n = 7)[]
```

`as.dbi.table` *Coerce to DBI Table*

Description

Test whether an object is a `dbi.table`, or coerce it if possible.

Usage

```
is.dbi.table(x)

as.dbi.table(conn, x, type = c("auto", "query", "temporary"))
```

Arguments

`x`	any R object.
`conn`	a connection handle returned by `dbConnect`. Alternatively, `conn` may be a `dbi.table` or a `dbi.catalog`; in these cases, the connection handle is extracted from the provided object.
`type`	a character string. Possible choices are `"auto"`, `"query"`, and `"temporary"`. See Details. The default `"auto"` uses *In Query* tables when `x` has 500 or fewer rows or when creating a temporary table on the database fails.

Details

Two types of tables are provided: *Temporary* (when `type == "temporary"`) and *In Query* (when `type == "query"`). For *Temporary*, the data are written to a SQL temporary table and the associated `dbi.table` is returned. For *In Query*, the data are written into a CTE as part of the query itself - useful when the connection does not permit creating temporary tables.

Value

a `dbi.table`.

Note

The temporary tables created by this function are dropped (by calling `dbRemoveTable`) during garbage collection when they are no longer referenced.

Examples

```
duck <- dbi.catalog(chinook.duckdb)
csql(as.dbi.table(duck, iris[1:4, 1:3], type = "query"))
```

`csql` *See SQL*

Description

View a `dbi.table`'s underlying SQL query.

Usage

```
csql(x, n = getOption("dbitable.max.fetch", 10000L), strict = FALSE)
```

Arguments

`x`	a `dbi.table`.
`n`	a single integer value. When nonnegative, limits the number of rows returned by the query to `n`.
`strict`	a logical value. If `TRUE` and `x` has a key, the key is appended to the ORDER BY clause.

Value

none (invisible `NULL`).

`dbi.attach` *Attach a Database Schema to the Search Path*

Description

The database schema is attached to the R search path. This means that the schema is searched by R when evaluating a variable, so that `dbi.table`s in the schema can be accessed by simply giving their names.

Usage

```
dbi.attach(
  what,
  pos = 2L,
  name = NULL,
  warn.conflicts = FALSE,
  schema = NULL,
  graphics = TRUE
)
```

Arguments

`what`	a connection handle returned by `dbConnect` or a zero-argument function that returns a connection handle.
`pos`	an integer specifying position in `search`() where to attach.
`name`	a character string specifying the name to use for the attached database.
`warn.conflicts`	a logical value. If `TRUE`, warnings are printed about `conflicts` from attaching the database, unless that database contains an object `.conflicts.OK`. A conflict is a function masking a function, or a non-function masking a non-function.
`schema`	a character string specifying the name of the schema to attach.
`graphics`	a logical value; passed to `menu`. In interactive sessions, when `schema` is `NULL` and multiple schemas are found on `what`, a menu is displayed to select a schema.

Value

an `environment`, the attached schema is invisibly returned.

See Also

`attach`

`dbi.catalog` *Create a* `dbi.catalog`

Description

A `dbi.catalog` represents a database catalog.

Usage

```
dbi.catalog(conn, schemas)
```

Arguments

`conn`	a connection handle returned by `dbConnect` or a zero-argument function that returns a connection handle.
`schemas`	a character vector of distinct schema names. These schemas will be loaded into the `dbi.catalog`. By default (when `schemas` is missing), `dbi.catalog` loads all available schemas.

Value

a `dbi.catalog`.

Examples

```
# chinook.duckdb is a zero-argument function that returns a DBI handle
(db <- dbi.catalog(chinook.duckdb))

# list schemas
ls(db)

# list the tables in the schema 'main'
ls(db$main)
```

`dbi.table.DBI` *DBI Methods for* `dbi.table`*s*

Description

Call DBI methods using the underlying DBI connection.

Usage

```
## S4 method for signature 'dbi.catalog'
dbAppendTable(conn, name, value, ..., row.names = NULL)

## S4 method for signature 'dbi.schema'
dbAppendTable(conn, name, value, ..., row.names = NULL)

## S4 method for signature 'dbi.table'
dbAppendTable(conn, name, value, ..., row.names = NULL)

## S4 method for signature 'dbi.catalog'
dbCreateTable(conn, name, fields, ..., row.names = NULL, temporary = FALSE)

## S4 method for signature 'dbi.schema'
dbCreateTable(conn, name, fields, ..., row.names = NULL, temporary = FALSE)
```

```
## S4 method for signature 'dbi.table'
dbCreateTable(conn, name, fields, ..., row.names = NULL, temporary = FALSE)

## S4 method for signature 'dbi.catalog,ANY'
dbExecute(conn, statement, ...)

## S4 method for signature 'dbi.schema,ANY'
dbExecute(conn, statement, ...)

## S4 method for signature 'dbi.table,ANY'
dbExecute(conn, statement, ...)

## S4 method for signature 'dbi.catalog'
dbGetInfo(dbObj, ...)

## S4 method for signature 'dbi.schema'
dbGetInfo(dbObj, ...)

## S4 method for signature 'dbi.table'
dbGetInfo(dbObj, ...)

## S4 method for signature 'dbi.table,missing'
dbGetQuery(conn, statement, ...)

## S4 method for signature 'dbi.catalog,ANY'
dbGetQuery(conn, statement, ...)

## S4 method for signature 'dbi.schema,ANY'
dbGetQuery(conn, statement, ...)

## S4 method for signature 'dbi.table,ANY'
dbGetQuery(conn, statement, ...)

## S4 method for signature 'dbi.catalog,ANY'
dbListFields(conn, name, ...)

## S4 method for signature 'dbi.schema,ANY'
dbListFields(conn, name, ...)

## S4 method for signature 'dbi.table,ANY'
dbListFields(conn, name, ...)

## S4 method for signature 'dbi.catalog'
dbListObjects(conn, prefix = NULL, ...)

## S4 method for signature 'dbi.schema'
dbListObjects(conn, prefix = NULL, ...)
```

```
## S4 method for signature 'dbi.table'
dbListObjects(conn, prefix = NULL, ...)

## S4 method for signature 'dbi.catalog,ANY'
dbQuoteIdentifier(conn, x, ...)

## S4 method for signature 'dbi.schema,ANY'
dbQuoteIdentifier(conn, x, ...)

## S4 method for signature 'dbi.table,ANY'
dbQuoteIdentifier(conn, x, ...)

## S4 method for signature 'dbi.catalog'
dbQuoteLiteral(conn, x, ...)

## S4 method for signature 'dbi.schema'
dbQuoteLiteral(conn, x, ...)

## S4 method for signature 'dbi.table'
dbQuoteLiteral(conn, x, ...)

## S4 method for signature 'dbi.catalog,ANY'
dbQuoteString(conn, x, ...)

## S4 method for signature 'dbi.schema,ANY'
dbQuoteString(conn, x, ...)

## S4 method for signature 'dbi.table,ANY'
dbQuoteString(conn, x, ...)

## S4 method for signature 'dbi.catalog,ANY'
dbReadTable(conn, name, ...)

## S4 method for signature 'dbi.schema,ANY'
dbReadTable(conn, name, ...)

## S4 method for signature 'dbi.table,ANY'
dbReadTable(conn, name, ...)

## S4 method for signature 'dbi.catalog,ANY'
dbRemoveTable(conn, name, ...)

## S4 method for signature 'dbi.schema,ANY'
dbRemoveTable(conn, name, ...)

## S4 method for signature 'dbi.table,ANY'
dbRemoveTable(conn, name, ...)
```

```
## S4 method for signature 'dbi.table,missing'
dbSendStatement(conn, statement, ...)

## S4 method for signature 'dbi.catalog,ANY'
dbSendStatement(conn, statement, ...)

## S4 method for signature 'dbi.schema,ANY'
dbSendStatement(conn, statement, ...)

## S4 method for signature 'dbi.table,ANY'
dbSendStatement(conn, statement, ...)

## S4 method for signature 'dbi.catalog'
dbWithTransaction(conn, code, ...)

## S4 method for signature 'dbi.schema'
dbWithTransaction(conn, code, ...)

## S4 method for signature 'dbi.table'
dbWithTransaction(conn, code, ...)

## S4 method for signature 'dbi.catalog,ANY'
dbWriteTable(conn, name, value, ...)

## S4 method for signature 'dbi.schema,ANY'
dbWriteTable(conn, name, value, ...)

## S4 method for signature 'dbi.table,ANY'
dbWriteTable(conn, name, value, ...)
```

Arguments

`conn`	A `dbi.catalog`, `dbi.schema`, or `dbi.table`.
`name`	Please refer to the documentation for the generic function (links can be found in the 'See Also' section).
`value`	Please refer to the documentation for the generic function (links can be found in the 'See Also' section).
`...`	Additional parameters to pass to methods.
`row.names`	Please refer to the documentation for the generic function (links can be found in the 'See Also' section).
`fields`	Please refer to the documentation for the generic function (links can be found in the 'See Also' section).
`temporary`	Please refer to the documentation for the generic function (links can be found in the 'See Also' section).
`statement`	Please refer to the documentation for the generic function (links can be found in the 'See Also' section).

`dbObj`	A `dbi.catalog`, `dbi.schema`, or `dbi.table`.
`prefix`	Please refer to the documentation for the generic function (links can be found in the 'See Also' section).
`x`	Please refer to the documentation for the generic function (links can be found in the 'See Also' section).
`code`	Please refer to the documentation for the generic function (links can be found in the 'See Also' section).

See Also

`dbAppendTable`, `dbCreateTable`, `dbExecute`, `dbGetInfo`, `dbGetQuery`, `dbListObjects`, `dbReadTable`, `dbQuoteIdentifier`, `dbQuoteLiteral`, `dbQuoteString`, `dbRemoveTable`, `dbSendStatement`, `dbWithTransaction`

`example_databases` *Example Databases*

Description

These zero-argument functions return connections to the example databases included in the **dbi.table** package.

Usage

```
chinook.sqlite()

chinook.duckdb()
```

Value

a `DBIConnection` object, as returned by `dbConnect`.

`merge` *Merge two dbi.tables*

Description

Merge two `dbi.table`s. By default, the columns to merge on are determined by the first of the following cases to apply.

1. If `x` and `y` are each unmodified `dbi.table`s in the same `dbi.catalog` and if there is a single foreign key relating `x` and `y` (either `x` referencing `y`, or `y` referencing `x`), then it is used to set `by.x` and `by.y`.
2. If `x` and `y` have shared key columns, then they are used to set `by` (that is, `by = intersect(key(x), key(y))` when `intersect(key(x), key(y))` has length greater than zero).

3. If x has a key, then it is used to set by (that is, by = key(x) when key(x) has length greater than zero).
4. If x and y have columns in common, then they are used to set by (that is, by = intersect(names(x), names(y)) when intersect(names(x), names(y)) has length greater than zero).

Use the by, by.x, and by.y arguments explicitly to override this default.

Usage

```
## S3 method for class 'dbi.table'
merge(
  x,
  y,
  by = NULL,
  by.x = NULL,
  by.y = NULL,
  all = FALSE,
  all.x = all,
  all.y = all,
  sort = TRUE,
  suffixes = c(".x", ".y"),
  no.dups = TRUE,
  recursive = FALSE,
  ...
)
```

Arguments

x, y	dbi.tables sharing the same DBI connection. If y is not a dbi.table but does inherit from data.frame, then it is coerced to a dbi.table using as.dbi.table. If y is missing, a merge is performed for each of x's foreign keys.
by	a character vector of shared column names in x and y to merge on.
by.x, by.y	character vectors of column names in x and y to merge on.
all	a logical value. all = TRUE is shorthand to save setting both all.x = TRUE and all.y = TRUE.
all.x	a logical value. When TRUE, rows from x that do not have a matching row in y are included. These rows will have NAs in the columns that are 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	a logical value. Analogous to all.x above.
sort	a logical value. When TRUE (default), the key of the merged dbi.table is set to the by / by.x columns.
suffixes	a length-2 character vector. The suffixes to be used for making non-by column names unique. The suffix behavior works in a similar fashion to the merge.data.frame method.
no.dups	a logical value. When TRUE, suffixes are also appended to non-by.y column names in y when they have the same column name as any by.x.

`recursive`	a logical value. Only used when `y` is missing. When `TRUE`, `merge` is called on each `dbi.table` prior to merging with `x`. See examples.
`...`	additional arguments are passed to `as.dbi.table`.

Details

`merge.dbi.table` uses `sql.join` to join `x` and `y` then formats the result set to match the typical `merge` output.

Value

a `dbi.table`.

See Also

`merge.data.table`, `merge.data.frame`

Examples

```
chinook <- dbi.catalog(chinook.duckdb)

#The Album table has a foreign key constriant that references Artist
merge(chinook$main$Album, chinook$main$Artist)

#When y is omitted, x's foreign key relationship is used to determine y
merge(chinook$main$Album)

#Track has 3 foreign keys: merge with Album, Genre, and MediaType
merge(chinook$main$Track)

#Track references Album but not Artist, Album references Artist
#This dbi.table includes the artist name
merge(chinook$main$Track, recursive = TRUE)
```

`reference.test` *Test* `dbi.table` *vs. Reference Implementation*

Description

Evaluate an expression including at least one `dbi.table` and compare the result with the *Reference Implementation*. This function is primarily for testing and is potentially very slow for large tables.

Usage

```
reference.test(
  expr,
  envir = parent.frame(),
  ignore.row.order = TRUE,
  verbose = TRUE
)
```

Arguments

`expr`	an expression involving at least one `dbi.table` and whose result can be coerced into a `data.table`.
`envir`	an environment. Where to evaluate `expr`.
`ignore.row.order`	a logical value. This argument is passed to `all.equal`.
`verbose`	a logical value. When `TRUE`, the output from `all.equal` is displayed in a message when `all.equal` returns anything other than `TRUE`.

Value

a logical value.

Reference Implementation

Suppose that `id1` identifies a table in a SQL database and that `[i, j, by]` describes a subset/select/summarize operation using `data.table` syntax. The *Reference Implementation* for this operation is:

`setDT(dbReadTable(conn, id1))[i, j, by]`

More generally, for an expression involving multiple SQL database objects and using `data.table` syntax, the *Reference Implementation* would be to download each of these objects in their entirety, convert them to `data.tables`, then evaluate the expression.

The goal of the **dbi.table** is to generate an SQL query that produces the same results set as the Reference Implementation up to row ordering.

Examples

```
library(data.table)
duck <- dbi.catalog(chinook.duckdb)
Album <- duck$main$Album
Artist <- duck$main$Artist

reference.test(merge(Album, Artist, by = "ArtistId"))
```

sql.join *Join* dbi.table*s*

Description

A SQL-like join of two dbi.tables that share the same DBI connection. All columns from both dbi.tables are returned.

Usage

```
sql.join(x, y, type = "inner", on = NULL, prefixes = c("x.", "y."))
```

Arguments

x, y	dbi.tables to join. x and y must share the same DBI connection.
type	a character string specifying the join type. Valid choices are "inner", "left", "right", "outer", and "cross".
on	a call specifying the join predicate. The symbols in on should be column names of x or column names of y, use prefixes as necessary.
prefixes	a 2-element character vector of distinct values. When x and y both have a column with the same name (e.g., common_name) then, when specifing the join predicate in on, use \`prefixes[1]\`common_name to refer to the common_name column in x and \`prefixes[2]\`common_name to refer to the common_name column in y. prefixes are also used to disambiguate the output column names.

Value

a dbi.table.

Examples

```
chinook <- dbi.catalog(chinook.duckdb)
Album <- chinook$main$Album
Artist <- chinook$main$Artist

sql.join(Album, Artist, type = "inner",
         on = Album.ArtistId == Artist.ArtistId,
         prefixes = c("Album.", "Artist."))
```

Index