Package ‘DBI’

September 10, 2016

Version 0.5-1

Date 2016-09-09

Title R Database Interface

Description A database interface definition for communication between R and relational database management systems. All classes in this package are virtual and need to be extended by the various R/DBMS implementations.

Depends R (>= 2.15.0), methods

Suggests testthat, RSQLite, knitr, rmarkdown, covr

Encoding UTF-8

License LGPL (>= 2)

URL `http://rstats-db.github.io/DBI`

URLNote https://github.com/rstats-db/DBI

BugReports `https://github.com/rstats-db/DBI/issues`

Collate 'DBObject.R' 'DBDriver.R' 'DBConnection.R' 'ANSI.R' 'DBI-package.R' 'DBResult.R' 'data-types.R' 'data.R' 'deprecated.R' 'hidden.R' 'interpolate.R' 'quote.R' 'rownames.R' 'table-create.R' 'table-insert.R' 'table.R' 'transactions.R' 'util.R'

VignetteBuilder knitr

RoxygenNote 5.0.1.9000

NeedsCompilation no

Author R Special Interest Group on Databases (R-SIG-DB) [aut],
Hadley Wickham [aut],
Kirill Müller [aut, cre]

Maintainer Kirill Müller <`krlmlr+r@mailbox.org`>

Repository CRAN

Date/Publication 2016-09-10 01:27:20

R topics documented:

DBI-package *R Database Interface*

Description

A database interface definition for communication between R and relational database management systems. All classes in this package are virtual and need to be extended by the various R/DBMS implementations.

See Also

Important generics: `dbConnect`, `dbGetQuery`, `dbWriteTable`, `dbDisconnect`

Formal specification (currently work in progress and incomplete): DBIspec

Examples

```
con <- dbConnect(RSQLite::SQLite(), ":memory:")

dbWriteTable(con, "iris", iris)
dbGetQuery(con, "SELECT * FROM iris WHERE [Petal.Width] > 2.3")

dbDisconnect(con)
```

dbBind *Bind values to a parameterised/prepared statement*

Description

The `dbSendQuery` function can be called with queries that contain placeholders for values. This function binds these placeholders to actual values, and is intended to be called on the result of `dbSendQuery` before calling `dbFetch`.

Usage

```
dbBind(res, params, ...)
```

Arguments

`res`	An object inheriting from `DBIResult`.
`params`	A list of bindings
`...`	Other arguments passed on to methods.

Details

Parametrised or prepared statements are executed as follows:

1. Call dbSendQuery with a query that contains placeholders, store the returned DBIResult object in a variable. Currently, the syntax for the placeholders is backend-specific, e.g., ?, $, $name and :name. Mixing placeholders (in particular, named and unnamed ones) is not recommended.
2. Call dbBind on the DBIResult object with a list that specifies actual values for the placeholders. The list must be named or unnamed, depending on the kind of placeholders used. Named values are matched to named paramters, unnamed values are matched by position.
3. Call dbFetch on the same DBIResult object.
4. Repeat 2. and 3. as necessary.
5. Close the result set via dbClearResult.

See Also

Other DBIResult generics: DBIResult-class, SQL, dbClearResult, dbColumnInfo, dbFetch, dbGetInfo, dbGetRowCount, dbGetRowsAffected, dbGetStatement, dbHasCompleted, dbIsValid

Examples

```
## Not run:
con <- dbConnect(RSQLite::SQLite(), ":memory:")

dbWriteTable(con, "iris", iris)
iris_result <- dbSendQuery(con, "SELECT * FROM iris WHERE [Petal.Width] > ?")
dbBind(iris_result, list(2.3))
dbFetch(iris_result)
dbBind(iris_result, list(3))
dbFetch(iris_result)

dbClearResult(iris_result)
dbDisconnect(con)

## End(Not run)
```

---

dbClearResult *Clear a result set*

Description

Frees all resources (local and remote) associated with a result set. In some cases (e.g., very large result sets) this can be a critical step to avoid exhausting resources (memory, file descriptors, etc.)

Usage

```
dbClearResult(res, ...)
```

Arguments

`res`	An object inheriting from `DBIResult`.
`...`	Other arguments passed on to methods.

Value

a logical indicating whether clearing the result set was successful or not.

See Also

Other DBIResult generics: `DBIResult-class`, `SQL`, `dbBind`, `dbColumnInfo`, `dbFetch`, `dbGetInfo`, `dbGetRowCount`, `dbGetRowsAffected`, `dbGetStatement`, `dbHasCompleted`, `dbIsValid`

Examples

```
con <- dbConnect(RSQLite::SQLite(), ":memory:")

rs <- dbSendQuery(con, "SELECT 1")
print(dbFetch(rs))

dbClearResult(rs)
dbDisconnect(con)
```

`dbColumnInfo` *Information about result types*

Description

Produces a data.frame that describes the output of a query. The data.frame should have as many rows as there are output fields in the result set, and each column in the data.frame should describe an aspect of the result set field (field name, type, etc.)

Usage

```
dbColumnInfo(res, ...)
```

Arguments

`res`	An object inheriting from `DBIResult`.
`...`	Other arguments passed on to methods.

Value

A data.frame with one row per output field in `res`. Methods MUST include `name`, `field.type` (the SQL type), and `data.type` (the R data type) columns, and MAY contain other database specific information like scale and precision or whether the field can store `NULL`s.

See Also

Other DBIResult generics: DBIResult-class, SQL, dbBind, dbClearResult, dbFetch, dbGetInfo, dbGetRowCount, dbGetRowsAffected, dbGetStatement, dbHasCompleted, dbIsValid

Examples

```
con <- dbConnect(RSQLite::SQLite(), ":memory:")

rs <- dbSendQuery(con, "SELECT 1 AS a, 2 AS b")
dbColumnInfo(rs)
dbFetch(rs)

dbClearResult(rs)
dbDisconnect(con)
```

---

dbConnect *Create a connection to a DBMS*

Description

Connect to a DBMS going through the appropriate authorization procedure. Some implementations may allow you to have multiple connections open, so you may invoke this function repeatedly assigning its output to different objects. The authorization mechanism is left unspecified, so check the documentation of individual drivers for details.

Usage

```
dbConnect(drv, ...)
```

Arguments

`drv`	an object that inherits from DBIDriver, or an existing DBIConnection object (in order to clone an existing connection).
`...`	authorization arguments needed by the DBMS instance; these typically include `user`, `password`, `dbname`, `host`, `port`, etc. For details see the appropriate `DBIDriver`.

Details

Each driver will define what other arguments are required, e.g., `"dbname"` for the database name, `"username"`, and `"password"`.

Value

An object that extends DBIConnection in a database-specific manner. For instance `dbConnect(RMySQL::MySQL())` produces an object of class `MySQLConnection`. This object is used to direct commands to the database engine.

See Also

dbDisconnect to disconnect from a database.

Other DBIDriver generics: DBIDriver-class, dbDataType, dbDriver, dbGetInfo, dbIsValid, dbListConnections

Examples

```
# SQLite only needs a path to the database. Other database drivers
# will require more details (like username, password, host, port etc)
con <- dbConnect(RSQLite::SQLite(), ":memory:")
con

dbListTables(con)

dbDisconnect(con)
```

---

dbDataType *Determine the SQL data type of an object*

Description

This is a generic function. The default method determines the SQL type of an R object according to the SQL 92 specification, which may serve as a starting point for driver implementations. The default method also provides a method for data.frame which will return a character vector giving the type for each column in the dataframe.

Usage

```
dbDataType(dbObj, obj, ...)
```

Arguments

dbObj	A object inheriting from DBIDriver or DBIConnection
obj	An R object whose SQL type we want to determine.
...	Other arguments passed on to methods.

Details

The data types supported by databases are different than the data types in R, but the mapping between the primitve types is straightforward: Any of the many fixed and varying length character types are mapped to character vectors. Fixed-precision (non-IEEE) numbers are mapped into either numeric or integer vectors.

Notice that many DBMS do not follow IEEE arithmetic, so there are potential problems with under/overflows and loss of precision.

Value

A character string specifying the SQL data type for `obj`.

See Also

Other DBIDriver generics: `DBIDriver-class`, `dbConnect`, `dbDriver`, `dbGetInfo`, `dbIsValid`, `dbListConnections`

Other DBIConnection generics: `DBIConnection-class`, `dbDisconnect`, `dbExecute`, `dbExistsTable`, `dbGetException`, `dbGetInfo`, `dbGetQuery`, `dbIsValid`, `dbListFields`, `dbListResults`, `dbListTables`, `dbReadTable`, `dbRemoveTable`, `dbSendQuery`, `dbSendStatement`

Examples

```
dbDataType(ANSI(), 1:5)
dbDataType(ANSI(), 1)
dbDataType(ANSI(), TRUE)
dbDataType(ANSI(), Sys.Date())
dbDataType(ANSI(), Sys.time())
dbDataType(ANSI(), Sys.time() - as.POSIXct(Sys.Date()))
dbDataType(ANSI(), c("x", "abc"))
dbDataType(ANSI(), list(raw(10), raw(20)))
dbDataType(ANSI(), I(3))

dbDataType(ANSI(), iris)

con <- dbConnect(RSQLite::SQLite(), ":memory:")

dbDataType(con, 1:5)
dbDataType(con, 1)
dbDataType(con, TRUE)
dbDataType(con, Sys.Date())
dbDataType(con, Sys.time())
dbDataType(con, Sys.time() - as.POSIXct(Sys.Date()))
dbDataType(con, c("x", "abc"))
dbDataType(con, list(raw(10), raw(20)))
dbDataType(con, I(3))

dbDataType(con, iris)

dbDisconnect(con)
```

`dbDisconnect` *Disconnect (close) a connection*

Description

This closes the connection, discards all pending work, and frees resources (e.g., memory, sockets).

Usage

```
dbDisconnect(conn, ...)
```

Arguments

`conn`	A `DBIConnection` object, as produced by `dbConnect`.
`...`	Other parameters passed on to methods.

Value

a logical vector of length 1, indicating success or failure.

See Also

Other DBIConnection generics: `DBIConnection-class`, `dbDataType`, `dbExecute`, `dbExistsTable`, `dbGetException`, `dbGetInfo`, `dbGetQuery`, `dbIsValid`, `dbListFields`, `dbListResults`, `dbListTables`, `dbReadTable`, `dbRemoveTable`, `dbSendQuery`, `dbSendStatement`

Examples

```
con <- dbConnect(RSQLite::SQLite(), ":memory:")
dbDisconnect(con)
```

`dbDriver` *Load and unload database drivers*

Description

`dbDriver` is a helper method used to create an new driver object given the name of a database or the corresponding R package. It works through convention: all DBI-extending packages should provide an exported object with the same name as the package. `dbDriver` just looks for this object in the right places: if you know what database you are connecting to, you should call the function directly.

Usage

```
dbDriver(drvName, ...)

dbUnloadDriver(drv, ...)
```

Arguments

`drvName`	character name of the driver to instantiate.
`...`	any other arguments are passed to the driver `drvName`.
`drv`	an object that inherits from `DBIDriver` as created by `dbDriver`.

Value

In the case of `dbDriver`, an driver object whose class extends `DBIDriver`. This object may be used to create connections to the actual DBMS engine.

In the case of `dbUnloadDriver`, a logical indicating whether the operation succeeded or not.

Side Effects

The client part of the database communication is initialized (typically dynamically loading C code, etc.) but note that connecting to the database engine itself needs to be done through calls to `dbConnect`.

See Also

Other DBIDriver generics: `DBIDriver-class`, `dbConnect`, `dbDataType`, `dbGetInfo`, `dbIsValid`, `dbListConnections`

Other DBIDriver generics: `DBIDriver-class`, `dbConnect`, `dbDataType`, `dbGetInfo`, `dbIsValid`, `dbListConnections`

Examples

```
# Create a RSQLite driver with a string
d <- dbDriver("SQLite")
d

# But better, access the object directly
RSQLite::SQLite()
```

`dbExecute` *Execute an update statement, query number of rows affected, and then close result set*

Description

`dbExecute` comes with a default implementation (which should work with most backends) that calls `dbSendStatement`, then `dbGetRowsAffected`, ensuring that the result is always free-d by `dbClearResult`.

Usage

```
dbExecute(conn, statement, ...)
```

Arguments

`conn`	A `DBIConnection` object, as produced by `dbConnect`.
`statement`	a character vector of length 1 containing SQL.
`...`	Other parameters passed on to methods.

Value

The number of rows affected by the `statement`

See Also

For queries: `dbSendQuery` and `dbGetQuery`.

Other DBIConnection generics: `DBIConnection-class`, `dbDataType`, `dbDisconnect`, `dbExistsTable`, `dbGetException`, `dbGetInfo`, `dbGetQuery`, `dbIsValid`, `dbListFields`, `dbListResults`, `dbListTables`, `dbReadTable`, `dbRemoveTable`, `dbSendQuery`, `dbSendStatement`

Examples

```
con <- dbConnect(RSQLite::SQLite(), ":memory:")

dbWriteTable(con, "cars", head(cars, 3))
dbReadTable(con, "cars")   # there are 3 rows
dbExecute(con,
  "INSERT INTO cars (speed, dist) VALUES (1, 1), (2, 2), (3, 3);")
dbReadTable(con, "cars")   # there are now 6 rows

dbDisconnect(con)
```

`dbExistsTable` *Does a table exist?*

Description

Does a table exist?

Usage

```
dbExistsTable(conn, name, ...)
```

Arguments

`conn`	A `DBIConnection` object, as produced by `dbConnect`.
`name`	A character string specifying a DBMS table name.
`...`	Other parameters passed on to methods.

Value

a logical vector of length 1.

See Also

Other DBIConnection generics: `DBIConnection-class`, `dbDataType`, `dbDisconnect`, `dbExecute`, `dbGetException`, `dbGetInfo`, `dbGetQuery`, `dbIsValid`, `dbListFields`, `dbListResults`, `dbListTables`, `dbReadTable`, `dbRemoveTable`, `dbSendQuery`, `dbSendStatement`

Examples

```
con <- dbConnect(RSQLite::SQLite(), ":memory:")

dbExistsTable(con, "iris")
dbWriteTable(con, "iris", iris)
dbExistsTable(con, "iris")

dbDisconnect(con)
```

dbFetch *Fetch records from a previously executed query*

Description

Fetch the next n elements (rows) from the result set and return them as a data.frame.

Usage

```
dbFetch(res, n = -1, ...)

fetch(res, n = -1, ...)
```

Arguments

res	An object inheriting from DBIResult.
n	maximum number of records to retrieve per fetch. Use n = -1 to retrieve all pending records. Some implementations may recognize other special values.
...	Other arguments passed on to methods.

Details

fetch is provided for compatibility with older DBI clients - for all new code you are strongly encouraged to use dbFetch. The default method for dbFetch calls fetch so that it is compatible with existing code. Implementors are free to provide methods for dbFetch only.

Value

a data.frame with as many rows as records were fetched and as many columns as fields in the result set.

See Also

close the result set with dbClearResult as soon as you finish retrieving the records you want.

Other DBIResult generics: DBIResult-class, SQL, dbBind, dbClearResult, dbColumnInfo, dbGetInfo, dbGetRowCount, dbGetRowsAffected, dbGetStatement, dbHasCompleted, dbIsValid

Examples

```
con <- dbConnect(RSQLite::SQLite(), ":memory:")

dbWriteTable(con, "mtcars", mtcars)

# Fetch all results
rs <- dbSendQuery(con, "SELECT * FROM mtcars WHERE cyl = 4")
dbFetch(rs)
dbClearResult(rs)

# Fetch in chunks
rs <- dbSendQuery(con, "SELECT * FROM mtcars")
while (!dbHasCompleted(rs)) {
  chunk <- dbFetch(rs, 10)
  print(nrow(chunk))
}

dbClearResult(rs)
dbDisconnect(con)
```

`dbGetException` *Get DBMS exceptions*

Description

Get DBMS exceptions

Usage

```
dbGetException(conn, ...)
```

Arguments

`conn`	A `DBIConnection` object, as produced by `dbConnect`.
`...`	Other parameters passed on to methods.

Value

a list with elements `errorNum` (an integer error number) and `errorMsg` (a character string) describing the last error in the connection `conn`.

See Also

Other DBIConnection generics: `DBIConnection-class`, `dbDataType`, `dbDisconnect`, `dbExecute`, `dbExistsTable`, `dbGetInfo`, `dbGetQuery`, `dbIsValid`, `dbListFields`, `dbListResults`, `dbListTables`, `dbReadTable`, `dbRemoveTable`, `dbSendQuery`, `dbSendStatement`

dbGetInfo *Get DBMS metadata*

Description

Get DBMS metadata

Usage

```
dbGetInfo(dbObj, ...)
```

Arguments

`dbObj`	An object inheriting from `DBIObject`, i.e. `DBIDriver`, `DBIConnection`, or a `DBIResult`
`...`	Other arguments to methods.

Value

a named list

Implementation notes

For `DBIDriver` subclasses, this should include the version of the package (`driver.version`), the version of the underlying client library (`client.version`), and the maximum number of connections (`max.connections`).

For `DBIConnection` objects this should report the version of the DBMS engine (`db.version`), database name (`dbname`), username, (`username`), host (`host`), port (`port`), etc. It MAY also include any other arguments related to the connection (e.g., thread id, socket or TCP connection type). It MUST NOT include the password.

For `DBIResult` objects, this should include the statement being executed (`statement`), how many rows have been fetched so far (in the case of queries, `row.count`), how many rows were affected (deleted, inserted, changed, (`rows.affected`), and if the query is complete (`has.completed`).

The default implementation for `DBIResult objects` constructs such a list from the return values of the corresponding methods, `dbGetStatement`, `dbGetRowCount`, `dbGetRowsAffected`, and `dbHasCompleted`.

See Also

Other DBIDriver generics: `DBIDriver-class`, `dbConnect`, `dbDataType`, `dbDriver`, `dbIsValid`, `dbListConnections`

Other DBIConnection generics: `DBIConnection-class`, `dbDataType`, `dbDisconnect`, `dbExecute`, `dbExistsTable`, `dbGetException`, `dbGetQuery`, `dbIsValid`, `dbListFields`, `dbListResults`, `dbListTables`, `dbReadTable`, `dbRemoveTable`, `dbSendQuery`, `dbSendStatement`

Other DBIResult generics: `DBIResult-class`, `SQL`, `dbBind`, `dbClearResult`, `dbColumnInfo`, `dbFetch`, `dbGetRowCount`, `dbGetRowsAffected`, `dbGetStatement`, `dbHasCompleted`, `dbIsValid` `dbGetQuery` *Send query, retrieve results and then clear result set*

Description

`dbGetQuery` comes with a default implementation that calls `dbSendQuery`, then `dbFetch`, ensuring that the result is always free-d by `dbClearResult`.

Usage

```
dbGetQuery(conn, statement, ...)
```

Arguments

`conn`	A `DBIConnection` object, as produced by `dbConnect`.
`statement`	a character vector of length 1 containing SQL.
`...`	Other parameters passed on to methods.

Details

This function is for `SELECT` queries only. Some backends may support data manipulation statements through this function for compatibility reasons. However callers are strongly advised to use `dbExecute` for data manipulation statements.

Implementation notes

Subclasses should override this method only if they provide some sort of performance optimisation.

See Also

For updates: `dbSendStatement` and `dbExecute`.

Other DBIConnection generics: `DBIConnection-class`, `dbDataType`, `dbDisconnect`, `dbExecute`, `dbExistsTable`, `dbGetException`, `dbGetInfo`, `dbIsValid`, `dbListFields`, `dbListResults`, `dbListTables`, `dbReadTable`, `dbRemoveTable`, `dbSendQuery`, `dbSendStatement`

Examples

```
con <- dbConnect(RSQLite::SQLite(), ":memory:")

dbWriteTable(con, "mtcars", mtcars)
dbGetQuery(con, "SELECT * FROM mtcars")

dbDisconnect(con)
```

dbGetRowCount *The number of rows fetched so far*

Description

This value is increased by calls to dbFetch. For a data modifying query, the return value is 0.

Usage

```
dbGetRowCount(res, ...)
```

Arguments

res	An object inheriting from DBIResult.
...	Other arguments passed on to methods.

Value

a numeric vector of length 1

See Also

Other DBIResult generics: DBIResult-class, SQL, dbBind, dbClearResult, dbColumnInfo, dbFetch, dbGetInfo, dbGetRowsAffected, dbGetStatement, dbHasCompleted, dbIsValid

Examples

```
con <- dbConnect(RSQLite::SQLite(), ":memory:")

dbWriteTable(con, "mtcars", mtcars)
rs <- dbSendQuery(con, "SELECT * FROM mtcars")

dbGetRowCount(rs)
ret1 <- dbFetch(rs, 10)
dbGetRowCount(rs)
ret2 <- dbFetch(rs)
dbGetRowCount(rs)
nrow(ret1) + nrow(ret2)

dbClearResult(rs)
dbDisconnect(con)
```

`dbGetRowsAffected` *The number of rows affected*

Description

This function returns the number of rows that were added, deleted, or updated by a data manipulation statement. For a selection query, this function returns 0.

Usage

```
dbGetRowsAffected(res, ...)
```

Arguments

`res`	An object inheriting from `DBIResult`.
`...`	Other arguments passed on to methods.

Value

a numeric vector of length 1

See Also

Other DBIResult generics: `DBIResult-class`, `SQL`, `dbBind`, `dbClearResult`, `dbColumnInfo`, `dbFetch`, `dbGetInfo`, `dbGetRowCount`, `dbGetStatement`, `dbHasCompleted`, `dbIsValid`

Examples

```
con <- dbConnect(RSQLite::SQLite(), ":memory:")

dbWriteTable(con, "mtcars", mtcars)
rs <- dbSendStatement(con, "DELETE FROM mtcars")
dbGetRowsAffected(rs)
nrow(mtcars)

dbClearResult(rs)
dbDisconnect(con)
```

`dbGetStatement` *Get the statement associated with a result set*

Description

Returns the statement that was passed to `dbSendQuery`.

Usage

```
dbGetStatement(res, ...)
```

Arguments

`res`	An object inheriting from `DBIResult`.
`...`	Other arguments passed on to methods.

Value

a character vector

See Also

Other DBIResult generics: `DBIResult-class`, `SQL`, `dbBind`, `dbClearResult`, `dbColumnInfo`, `dbFetch`, `dbGetInfo`, `dbGetRowCount`, `dbGetRowsAffected`, `dbHasCompleted`, `dbIsValid`

Examples

```
con <- dbConnect(RSQLite::SQLite(), ":memory:")

dbWriteTable(con, "mtcars", mtcars)
rs <- dbSendQuery(con, "SELECT * FROM mtcars")
dbGetStatement(rs)

dbClearResult(rs)
dbDisconnect(con)
```

`dbHasCompleted` *Completion status*

Description

This method returns if the operation has completed. A `SELECT` query is completed if all rows have been fetched. A data manipulation statement is completed if it has been executed.

Usage

```
dbHasCompleted(res, ...)
```

Arguments

`res`	An object inheriting from `DBIResult`.
`...`	Other arguments passed on to methods.

Value

a logical vector of length 1

See Also

Other DBIResult generics: `DBIResult-class`, `SQL`, `dbBind`, `dbClearResult`, `dbColumnInfo`, `dbFetch`, `dbGetInfo`, `dbGetRowCount`, `dbGetRowsAffected`, `dbGetStatement`, `dbIsValid`

Examples

```
con <- dbConnect(RSQLite::SQLite(), ":memory:")

dbWriteTable(con, "mtcars", mtcars)
rs <- dbSendQuery(con, "SELECT * FROM mtcars")

dbHasCompleted(rs)
ret1 <- dbFetch(rs, 10)
dbHasCompleted(rs)
ret2 <- dbFetch(rs)
dbHasCompleted(rs)

dbClearResult(rs)
dbDisconnect(con)
```

`DBIConnection-class` *DBIConnection class*

Description

This virtual class encapsulates the connection to a DBMS, and it provides access to dynamic queries, result sets, DBMS session management (transactions), etc.

Implementation note

Individual drivers are free to implement single or multiple simultaneous connections.

See Also

Other DBI classes: `DBIDriver-class`, `DBIObject-class`, `DBIResult-class`

Other DBIConnection generics: `dbDataType`, `dbDisconnect`, `dbExecute`, `dbExistsTable`, `dbGetException`, `dbGetInfo`, `dbGetQuery`, `dbIsValid`, `dbListFields`, `dbListResults`, `dbListTables`, `dbReadTable`, `dbRemoveTable`, `dbSendQuery`, `dbSendStatement`

Examples

```
con <- dbConnect(RSQLite::SQLite(), ":memory:")
con
dbDisconnect(con)

## Not run:
con <- dbConnect(RPostgreSQL::PostgreSQL(), "username", "passsword")
con
dbDisconnect(con)

## End(Not run)
```

DBIDriver-class *DBIDriver class*

Description

Base class for all DBMS drivers (e.g., RSQLite, MySQL, PostgreSQL). The virtual class `DBIDriver` defines the operations for creating connections and defining data type mappings. Actual driver classes, for instance `RPgSQL`, `RMySQL`, etc. implement these operations in a DBMS-specific manner.

See Also

Other DBI classes: `DBIConnection-class`, `DBIObject-class`, `DBIResult-class`

Other DBIDriver generics: `dbConnect`, `dbDataType`, `dbDriver`, `dbGetInfo`, `dbIsValid`, `dbListConnections`

DBIObject-class *DBIObject class*

Description

Base class for all other DBI classes (e.g., drivers, connections). This is a virtual Class: No objects may be created from it.

Details

More generally, the DBI defines a very small set of classes and generics that allows users and applications access DBMS with a common interface. The virtual classes are `DBIDriver` that individual drivers extend, `DBIConnection` that represent instances of DBMS connections, and `DBIResult` that represent the result of a DBMS statement. These three classes extend the basic class of `DBIObject`, which serves as the root or parent of the class hierarchy.

Implementation notes

An implementation MUST provide methods for the following generics:

- dbGetInfo.

It MAY also provide methods for:

- summary. Print a concise description of the object. The default method invokes dbGetInfo(dbObj) and prints the name-value pairs one per line. Individual implementations may tailor this appropriately.

See Also

Other DBI classes: DBIConnection-class, DBIDriver-class, DBIResult-class

Examples

```
drv <- RSQLite::SQLite()
con <- dbConnect(drv)

rs <- dbSendQuery(con, "SELECT 1")
is(drv, "DBIObject")   ## True
is(con, "DBIObject")   ## True
is(rs, "DBIObject")

dbClearResult(rs)
dbDisconnect(con)
```

DBIResult-class *DBIResult class*

Description

This virtual class describes the result and state of execution of a DBMS statement (any statement, query or non-query). The result set keeps track of whether the statement produces output how many rows were affected by the operation, how many rows have been fetched (if statement is a query), whether there are more rows to fetch, etc.

Implementation notes

Individual drivers are free to allow single or multiple active results per connection.

The default show method displays a summary of the query using other DBI generics.

See Also

Other DBI classes: DBIConnection-class, DBIDriver-class, DBIObject-class

Other DBIResult generics: SQL, dbBind, dbClearResult, dbColumnInfo, dbFetch, dbGetInfo, dbGetRowCount, dbGetRowsAffected, dbGetStatement, dbHasCompleted, dbIsValid

`DBIspec` *DBI specification*

Description

The **DBI** package defines the generic DataBase Interface for R. The connection to individual DBMS is made by packages that import **DBI** (so-called *DBI backends*). This document formalizes the behavior expected by the functions declared in **DBI** and implemented by the individal backends.

To ensure maximum portability and exchangeability, and to reduce the effort for implementing a new DBI backend, the **DBItest** package defines a comprehensive set of test cases that test conformance to the DBI specification. In fact, this document is derived from comments in the test definitions of the **DBItest** package. This ensures that an extension or update to the tests will be reflected in this document.

Getting started

A DBI backend is an R package, which should import the **DBI** and **methods** packages. For better or worse, the names of many existing backends start with 'R', e.g., **RSQLite**, **RMySQL**, **RSQLServer**; it is up to the package author to adopt this convention or not.

Driver

Each DBI backend implements a *driver class*, which must be an S4 class and inherit from the `DBIDriver` class. This section describes the construction of, and the methods defined for, this driver class.

Construction: The backend must support creation of an instance of this driver class with a *constructor function*. By default, its name is the package name without the leading 'R' (if it exists), e.g., `SQLite` for the **RSQLite** package. For the automated tests, the constructor name can be tweaked using the `constructor_name` tweak.

The constructor must be exported, and it must be a function that is callable without arguments. For the automated tests, unless the `constructor_relax_args` tweak is set to `TRUE`, an empty argument list is expected. Otherwise, an argument list where all arguments have default values is also accepted.

`dbDataType("DBIDriver", "ANY")`: The backend can override the `dbDataType` generic for its driver class. This generic expects an arbitrary object as second argument and returns a corresponding SQL type as atomic character value with at least one character. As-is objects (i.e., wrapped by `I`) must be supported and return the same results as their unwrapped counterparts.

To query the values returned by the default implementation, run `example(dbDataType, package = "DBI")`. If the backend needs to override this generic, it must accept all basic R data types as its second argument, namely `logical`, `integer`, `numeric`, `character`, dates (see `Dates`), date-time (see `DateTimeClasses`), and `difftime`. It also must accept lists of `raw` vectors and map them to the BLOB (binary large object) data type. The behavior for other object types is not specified.

Transactions

`dbBegin("DBIConnection")` **and** `dbCommit("DBIConnection")`**:** Transactions are available in DBI, but actual support may vary between backends. A transaction is initiated by a call to `dbBegin` and committed by a call to `dbCommit`. Both generics expect an object of class `DBIConnection` and return `TRUE` (invisibly) upon success.

The implementations are expected to raise an error in case of failure, but this is difficult to test in an automated way. In any way, both generics should throw an error with a closed connection. In addition, a call to `dbCommit` without a call to `dbBegin` should raise an error. Nested transactions are not supported by DBI, an attempt to call `dbBegin` twice should yield an error.

Data written in a transaction must persist after the transaction is committed. For example, a table that is missing when the transaction is started but is created and populated during the transaction must exist and contain the data added there both during and after the transaction.

The transaction isolation level is not specified by DBI.

`dbIsValid` *Is this DBMS object still valid?*

Description

This generic tests whether a database object is still valid (i.e. it hasn't been disconnected or cleared).

Usage

```
dbIsValid(dbObj, ...)
```

Arguments

`dbObj`	An object inheriting from `DBIObject`, i.e. `DBIDriver`, `DBIConnection`, or a `DBIResult`
`...`	Other arguments to methods.

Value

a logical of length 1

See Also

Other DBIDriver generics: `DBIDriver-class`, `dbConnect`, `dbDataType`, `dbDriver`, `dbGetInfo`, `dbListConnections`

Other DBIConnection generics: `DBIConnection-class`, `dbDataType`, `dbDisconnect`, `dbExecute`, `dbExistsTable`, `dbGetException`, `dbGetInfo`, `dbGetQuery`, `dbListFields`, `dbListResults`, `dbListTables`, `dbReadTable`, `dbRemoveTable`, `dbSendQuery`, `dbSendStatement`

Other DBIResult generics: `DBIResult-class`, `SQL`, `dbBind`, `dbClearResult`, `dbColumnInfo`, `dbFetch`, `dbGetInfo`, `dbGetRowCount`, `dbGetRowsAffected`, `dbGetStatement`, `dbHasCompleted`

Examples

```
dbIsValid(RSQLite::SQLite())

con <- dbConnect(RSQLite::SQLite(), ":memory:")
dbIsValid(con)

rs <- dbSendQuery(con, "SELECT 1")
dbIsValid(rs)

dbClearResult(rs)
dbIsValid(rs)

dbDisconnect(con)
dbIsValid(con)
```

`dbListConnections` *List currently open connections*

Description

Drivers that implement only a single connections MUST return a list containing a single element. If no connection are open, methods MUST return an empty list.

Usage

```
dbListConnections(drv, ...)
```

Arguments

`drv`	A object inheriting from `DBIDriver`
`...`	Other arguments passed on to methods.

Value

a list

See Also

Other DBIDriver generics: `DBIDriver-class`, `dbConnect`, `dbDataType`, `dbDriver`, `dbGetInfo`, `dbIsValid`

dbListFields *List field names of a remote table*

Description

List field names of a remote table

Usage

```
dbListFields(conn, name, ...)
```

Arguments

conn	A DBIConnection object, as produced by dbConnect.
name	a character string with the name of the remote table.
...	Other parameters passed on to methods.

Value

a character vector

See Also

dbColumnInfo to get the type of the fields.

Other DBIConnection generics: DBIConnection-class, dbDataType, dbDisconnect, dbExecute, dbExistsTable, dbGetException, dbGetInfo, dbGetQuery, dbIsValid, dbListResults, dbListTables, dbReadTable, dbRemoveTable, dbSendQuery, dbSendStatement

Examples

```
con <- dbConnect(RSQLite::SQLite(), ":memory:")

dbWriteTable(con, "mtcars", mtcars)
dbListFields(con, "mtcars")

dbDisconnect(con)
```

`dbListResults` *A list of all pending results*

Description

List of DBIResult objects currently active on the connection.

Usage

```
dbListResults(conn, ...)
```

Arguments

`conn`	A `DBIConnection` object, as produced by `dbConnect`.
`...`	Other parameters passed on to methods.

Value

a list. If no results are active, an empty list. If only a single result is active, a list with one element.

See Also

Other DBIConnection generics: `DBIConnection-class`, `dbDataType`, `dbDisconnect`, `dbExecute`, `dbExistsTable`, `dbGetException`, `dbGetInfo`, `dbGetQuery`, `dbIsValid`, `dbListFields`, `dbListTables`, `dbReadTable`, `dbRemoveTable`, `dbSendQuery`, `dbSendStatement`

`dbListTables` *List remote tables*

Description

This should, where possible, include temporary tables.

Usage

```
dbListTables(conn, ...)
```

Arguments

`conn`	A `DBIConnection` object, as produced by `dbConnect`.
`...`	Other parameters passed on to methods.

Value

a character vector. If no tables present, a character vector of length 0.

See Also

Other DBIConnection generics: `DBIConnection-class`, `dbDataType`, `dbDisconnect`, `dbExecute`, `dbExistsTable`, `dbGetException`, `dbGetInfo`, `dbGetQuery`, `dbIsValid`, `dbListFields`, `dbListResults`, `dbReadTable`, `dbRemoveTable`, `dbSendQuery`, `dbSendStatement`

Examples

```
con <- dbConnect(RSQLite::SQLite(), ":memory:")

dbListTables(con)
dbWriteTable(con, "mtcars", mtcars)
dbListTables(con)

dbDisconnect(con)
```

`dbReadTable` *Copy data frames to and from database tables*

Description

`dbReadTable`: database table -> data frame; `dbWriteTable`: data frame -> database table.

Usage

```
dbReadTable(conn, name, ...)

dbWriteTable(conn, name, value, ...)
```

Arguments

`conn`	A `DBIConnection` object, as produced by `dbConnect`.
`name`	A character string specifying a DBMS table name.
`...`	Other parameters passed on to methods.
`value`	a data.frame (or coercible to data.frame).

Value

a data.frame.

Note

The translation of identifiers between R and SQL is done through calls to `make.names` and `make.db.names`, but we cannot guarantee that the conversion is reversible. For details see `make.db.names`.

See Also

Other DBIConnection generics: `DBIConnection-class`, `dbDataType`, `dbDisconnect`, `dbExecute`, `dbExistsTable`, `dbGetException`, `dbGetInfo`, `dbGetQuery`, `dbIsValid`, `dbListFields`, `dbListResults`, `dbListTables`, `dbRemoveTable`, `dbSendQuery`, `dbSendStatement`

Examples

```
con <- dbConnect(RSQLite::SQLite(), ":memory:")

dbWriteTable(con, "mtcars", mtcars[1:10, ])
dbReadTable(con, "mtcars")

dbDisconnect(con)
```

`dbRemoveTable` *Remove a table from the database*

Description

Executes the sql `DROP TABLE name`.

Usage

```
dbRemoveTable(conn, name, ...)
```

Arguments

`conn`	A `DBIConnection` object, as produced by `dbConnect`.
`name`	A character string specifying a DBMS table name.
`...`	Other parameters passed on to methods.

Value

a logical vector of length 1 indicating success or failure.

See Also

Other DBIConnection generics: `DBIConnection-class`, `dbDataType`, `dbDisconnect`, `dbExecute`, `dbExistsTable`, `dbGetException`, `dbGetInfo`, `dbGetQuery`, `dbIsValid`, `dbListFields`, `dbListResults`, `dbListTables`, `dbReadTable`, `dbSendQuery`, `dbSendStatement`

Examples

```
con <- dbConnect(RSQLite::SQLite(), ":memory:")

dbExistsTable(con, "iris")
dbWriteTable(con, "iris", iris)
dbExistsTable(con, "iris")
dbRemoveTable(con, "iris")
dbExistsTable(con, "iris")

dbDisconnect(con)
```

`dbSendQuery` *Execute a query on a given database connection*

Description

The function `dbSendQuery` only submits and synchronously executes the SQL query to the database engine. It does *not* extract any records — for that you need to use the function `dbFetch`, and then you must call `dbClearResult` when you finish fetching the records you need. For interactive use, you should almost always prefer `dbGetQuery`.

Usage

```
dbSendQuery(conn, statement, ...)
```

Arguments

`conn`	A `DBIConnection` object, as produced by `dbConnect`.
`statement`	a character vector of length 1 containing SQL.
`...`	Other parameters passed on to methods.

Details

This function is for `SELECT` queries only. Some backends may support data manipulation queries through this function for compatibility reasons. However, callers are strongly advised to use `dbSendStatement` for data manipulation statements.

Value

An object that inherits from `DBIResult`. The result set can be used with `dbFetch` to extract records. Once you have finished using a result, make sure to disconnect it with `dbClearResult`.

Side Effects

The query is submitted to the database server and the DBMS executes it, possibly generating vast amounts of data. Where these data live is driver-specific: some drivers may choose to leave the output on the server and transfer them piecemeal to R, others may transfer all the data to the client – but not necessarily to the memory that R manages. See individual drivers' `dbSendQuery` documentation for details.

See Also

For updates: dbSendStatement and dbExecute.

Other DBIConnection generics: DBIConnection-class, dbDataType, dbDisconnect, dbExecute, dbExistsTable, dbGetException, dbGetInfo, dbGetQuery, dbIsValid, dbListFields, dbListResults, dbListTables, dbReadTable, dbRemoveTable, dbSendStatement

Examples

```
con <- dbConnect(RSQLite::SQLite(), ":memory:")

dbWriteTable(con, "mtcars", mtcars)
rs <- dbSendQuery(con, "SELECT * FROM mtcars WHERE cyl = 4;")
dbFetch(rs)
dbClearResult(rs)

dbDisconnect(con)
```

---

dbSendStatement *Execute a data manipulation statement on a given database connection*

Description

The function `dbSendStatement` only submits and synchronously executes the SQL data manipulation statement (e.g., `UPDATE`, `DELETE`, `INSERT INTO`, `DROP TABLE`, ...) to the database engine. To query the number of affected rows, call dbGetRowsAffected on the returned result object. You must also call dbClearResult after that. For interactive use, you should almost always prefer dbExecute.

Usage

```
dbSendStatement(conn, statement, ...)
```

Arguments

`conn`	A DBIConnection object, as produced by dbConnect.
`statement`	a character vector of length 1 containing SQL.
`...`	Other parameters passed on to methods.

Details

dbSendStatement comes with a default implementation that simply forwards to dbSendQuery, to support backends that only implement the latter.

Value

An object that inherits from DBIResult. Once you have finished using a result, make sure to disconnect it with dbClearResult.

See Also

For queries: dbSendQuery and dbGetQuery.

Other DBIConnection generics: DBIConnection-class, dbDataType, dbDisconnect, dbExecute, dbExistsTable, dbGetException, dbGetInfo, dbGetQuery, dbIsValid, dbListFields, dbListResults, dbListTables, dbReadTable, dbRemoveTable, dbSendQuery

Examples

```
con <- dbConnect(RSQLite::SQLite(), ":memory:")

dbWriteTable(con, "cars", head(cars, 3))
rs <- dbSendStatement(con,
  "INSERT INTO cars (speed, dist) VALUES (1, 1), (2, 2), (3, 3);")
dbHasCompleted(rs)
dbGetRowsAffected(rs)
dbClearResult(rs)
dbReadTable(con, "cars")   # there are now 6 rows

dbDisconnect(con)
```

---

dbWithTransaction *Self-contained SQL transactions*

Description

Given that transactions are implemented, this function allows you to pass in code that is run in a transaction. The default method of `dbWithTransaction` calls dbBegin before executing the code, and dbCommit after successful completion, or dbRollback in case of an error. The advantage is that you don't have to remember to do `dbBegin` and `dbCommit` or `dbRollback` – that is all taken care of. The special function `dbBreak` allows an early exit with rollback, it can be called only inside `dbWithTransaction`.

Usage

```
dbWithTransaction(conn, code)

dbBreak()
```

Arguments

conn	A DBIConnection object, as produced by dbConnect.
code	An arbitrary block of R code

Value

The result of the evaluation of code

Side Effects

The transaction in code on the connection conn is committed or rolled back. The code chunk may also modify the local R environment.

Examples

```
con <- dbConnect(RSQLite::SQLite(), ":memory:")

dbWriteTable(con, "cars", head(cars, 3))
dbReadTable(con, "cars")   # there are 3 rows

## successful transaction
dbWithTransaction(con, {
  dbExecute(con, "INSERT INTO cars (speed, dist) VALUES (1, 1);")
  dbExecute(con, "INSERT INTO cars (speed, dist) VALUES (2, 2);")
  dbExecute(con, "INSERT INTO cars (speed, dist) VALUES (3, 3);")
})
dbReadTable(con, "cars")   # there are now 6 rows

## failed transaction -- note the missing comma
tryCatch(
  dbWithTransaction(con, {
    dbExecute(con, "INSERT INTO cars (speed, dist) VALUES (1, 1);")
    dbExecute(con, "INSERT INTO cars (speed dist) VALUES (2, 2);")
    dbExecute(con, "INSERT INTO cars (speed, dist) VALUES (3, 3);")
  }),
  error = identity
)
dbReadTable(con, "cars")   # still 6 rows

## early exit, silently
dbWithTransaction(con, {
  dbExecute(con, "INSERT INTO cars (speed, dist) VALUES (1, 1);")
  dbExecute(con, "INSERT INTO cars (speed, dist) VALUES (2, 2);")
  if (nrow(dbReadTable(con, "cars")) > 7) dbBreak()
  dbExecute(con, "INSERT INTO cars (speed, dist) VALUES (3, 3);")
})
dbReadTable(con, "cars")   # still 6 rows

dbDisconnect(con)
```

`make.db.names` *Make R identifiers into legal SQL identifiers*

Description

These methods are DEPRECATED. Please use `dbQuoteIdentifier` (or possibly `dbQuoteString`) instead.

Usage

```
make.db.names(dbObj, snames, keywords = .SQL92Keywords, unique = TRUE,
  allow.keywords = TRUE, ...)

make.db.names.default(snames, keywords = .SQL92Keywords, unique = TRUE,
  allow.keywords = TRUE)

isSQLKeyword(dbObj, name, keywords = .SQL92Keywords, case = c("lower",
  "upper", "any")[3], ...)

isSQLKeyword.default(name, keywords = .SQL92Keywords, case = c("lower",
  "upper", "any")[3])
```

Arguments

`dbObj`	any DBI object (e.g., `DBIDriver`).
`snames`	a character vector of R identifiers (symbols) from which we need to make SQL identifiers.
`keywords`	a character vector with SQL keywords, by default it's `.SQL92Keywords` defined by the DBI.
`unique`	logical describing whether the resulting set of SQL names should be unique. Its default is `TRUE`. Following the SQL 92 standard, uniqueness of SQL identifiers is determined regardless of whether letters are upper or lower case.
`allow.keywords`	logical describing whether SQL keywords should be allowed in the resulting set of SQL names. Its default is `TRUE`
`name`	a character vector with database identifier candidates we need to determine whether they are legal SQL identifiers or not.
`case`	a character string specifying whether to make the comparison as lower case, upper case, or any of the two. it defaults to `any`.
`...`	any other argument are passed to the driver implementation.

Details

The algorithm in `make.db.names` first invokes `make.names` and then replaces each occurrence of a dot "." by an underscore "_". If `allow.keywords` is `FALSE` and identifiers collide with SQL keywords, a small integer is appended to the identifier in the form of `"_n"`.

The set of SQL keywords is stored in the character vector `.SQL92Keywords` and reflects the SQL ANSI/ISO standard as documented in "X/Open SQL and RDA", 1994, ISBN 1-872630-68-8. Users can easily override or update this vector.

Value

`make.db.names` returns a character vector of legal SQL identifiers corresponding to its `snames` argument.

`SQLKeywords` returns a character vector of all known keywords for the database-engine associated with `dbObj`.

`isSQLKeyword` returns a logical vector parallel to `name`.

Bugs

The current mapping is not guaranteed to be fully reversible: some SQL identifiers that get mapped into R identifiers with `make.names` and then back to SQL with `make.db.names` will not be equal to the original SQL identifiers (e.g., compound SQL identifiers of the form `username.tablename` will loose the dot ".").

References

The set of SQL keywords is stored in the character vector `.SQL92Keywords` and reflects the SQL ANSI/ISO standard as documented in "X/Open SQL and RDA", 1994, ISBN 1-872630-68-8. Users can easily override or update this vector.

rownames — *Convert row names back and forth between columns*

Description

These functions provide a reasonably automatic way of preserving the row names of data frame during back-and-forth translation to a SQL table. By default, row names will be converted to an explicit column called "row_names", and any query returning a column called "row_names" will have those automatically set as row names. These methods are mostly useful for backend implementers.

Usage

```
sqlRownamesToColumn(df, row.names = NA)

sqlColumnToRownames(df, row.names = NA)
```

Arguments

`df`	A data frame
`row.names`	Either `TRUE`, `FALSE`, `NA` or a string. If `TRUE`, always translate row names to a column called "row_names". If `FALSE`, never translate row names. If `NA`, translate rownames only if they're a character vector. A string is equivalent to `TRUE`, but allows you to override the default name. For backward compatibility, `NULL` is equivalent to `FALSE`.

Examples

```
# If have row names
sqlRownamesToColumn(head(mtcars))
sqlRownamesToColumn(head(mtcars), FALSE)
sqlRownamesToColumn(head(mtcars), "ROWNAMES")

# If don't have
sqlRownamesToColumn(head(iris))
sqlRownamesToColumn(head(iris), TRUE)
sqlRownamesToColumn(head(iris), "ROWNAMES")
```

SQL *SQL quoting*

Description

This set of classes and generics make it possible to flexibly deal with SQL escaping needs. By default, any user supplied input to a query should be escaped using either `dbQuoteIdentifier` or `dbQuoteString` depending on whether it refers to a table or variable name, or is a literal string. These functions return an object of the `SQL` class, which tells DBI functions that a character string does not need to be escaped anymore, to prevent double escaping. The `SQL` class has associated the `SQL()` constructor function.

Usage

```
SQL(x)

dbQuoteIdentifier(conn, x, ...)

dbQuoteString(conn, x, ...)
```

Arguments

`x`	A character vector to label as being escaped SQL.
`conn`	A subclass of `DBIConnection`, representing an active connection to an DBMS.
`...`	Other arguments passed on to methods. Not otherwise used.

Value

An object of class `SQL`.

Implementation notes

DBI provides default generics for SQL-92 compatible quoting. If the database uses a different convention, you will need to provide your own methods. Note that because of the way that S4 dispatch finds methods and because SQL inherits from character, if you implement (e.g.) a method for `dbQuoteString(MyConnection, character)`, you will also need to implement `dbQuoteString(MyConnection, SQL)` - this should simply return `x` unchanged.

If you implement your own method, make sure to convert NA to NULL (unquoted).

See Also

Other DBIResult generics: DBIResult-class, dbBind, dbClearResult, dbColumnInfo, dbFetch, dbGetInfo, dbGetRowCount, dbGetRowsAffected, dbGetStatement, dbHasCompleted, dbIsValid

Other DBIResult generics: DBIResult-class, dbBind, dbClearResult, dbColumnInfo, dbFetch, dbGetInfo, dbGetRowCount, dbGetRowsAffected, dbGetStatement, dbHasCompleted, dbIsValid

Examples

```
# Quoting ensures that arbitrary input is safe for use in a query
name <- "Robert'); DROP TABLE Students;--"
dbQuoteString(ANSI(), name)
dbQuoteIdentifier(ANSI(), name)

# NAs become NULL
dbQuoteString(ANSI(), c("x", NA))

# SQL vectors are always passed through as is
var_name <- SQL("select")
var_name

dbQuoteIdentifier(ANSI(), var_name)
dbQuoteString(ANSI(), var_name)

# This mechanism is used to prevent double escaping
dbQuoteString(ANSI(), dbQuoteString(ANSI(), name))
```

---

sqlAppendTable *Insert rows into a table*

Description

sqlAppendTable generates a single SQL string that inserts a data frame into an existing table. sqlAppendTableTemplate generates a template suitable for use with dbBind. These methods are mostly useful for backend implementers.

Usage

```
sqlAppendTable(con, table, values, row.names = NA, ...)

sqlAppendTableTemplate(con, table, values, row.names = NA, prefix = "?",
  ...)
```

Arguments

con	A database connection.
table	Name of the table. Escaped with dbQuoteIdentifier.
values	A data frame. Factors will be converted to character vectors. Character vectors will be escaped with dbQuoteString.

`row.names`	Either `TRUE`, `FALSE`, `NA` or a string. If `TRUE`, always translate row names to a column called "row_names". If `FALSE`, never translate row names. If `NA`, translate rownames only if they're a character vector. A string is equivalent to `TRUE`, but allows you to override the default name. For backward compatibility, `NULL` is equivalent to `FALSE`.
`...`	Other arguments used by individual methods.
`prefix`	Parameter prefix to put in front of column id.

Examples

```
sqlAppendTable(ANSI(), "iris", head(iris))

sqlAppendTable(ANSI(), "mtcars", head(mtcars))
sqlAppendTable(ANSI(), "mtcars", head(mtcars), row.names = FALSE)
sqlAppendTableTemplate(ANSI(), "iris", iris)

sqlAppendTableTemplate(ANSI(), "mtcars", mtcars)
sqlAppendTableTemplate(ANSI(), "mtcars", mtcars, row.names = FALSE)
```

`sqlCreateTable` *Create a simple table*

Description

Exposes interface to simple `CREATE TABLE` commands. The default method is ANSI SQL 99 compliant. This method is mostly useful for backend implementers.

Usage

```
sqlCreateTable(con, table, fields, row.names = NA, temporary = FALSE, ...)
```

Arguments

`con`	A database connection.
`table`	Name of the table. Escaped with `dbQuoteIdentifier`.
`fields`	Either a character vector or a data frame. A named character vector: Names are column names, values are types. Names are escaped with `dbQuoteIdentifier`. Field types are unescaped. A data frame: field types are generated using `dbDataType`.
`row.names`	Either `TRUE`, `FALSE`, `NA` or a string. If `TRUE`, always translate row names to a column called "row_names". If `FALSE`, never translate row names. If `NA`, translate rownames only if they're a character vector. A string is equivalent to `TRUE`, but allows you to override the default name. For backward compatibility, `NULL` is equivalent to `FALSE`.

`temporary`	If `TRUE`, will generate a temporary table statement.
`...`	Other arguments used by individual methods.

DBI-backends

If you implement one method (i.e. for strings or data frames), you need to implement both, otherwise the S4 dispatch rules will be ambiguous and will generate an error on every call.

Examples

```
sqlCreateTable(ANSI(), "my-table", c(a = "integer", b = "text"))
sqlCreateTable(ANSI(), "my-table", iris)

# By default, character row names are converted to a row_names colum
sqlCreateTable(ANSI(), "mtcars", mtcars[, 1:5])
sqlCreateTable(ANSI(), "mtcars", mtcars[, 1:5], row.names = FALSE)
```

`sqlData` *Convert a data frame into form suitable for upload to a SQL database*

Description

This is a generic method that coerces R objects into vectors suitable for upload to the database. The output will vary a little from method to method depending on whether the main upload device is through a single SQL string or multiple parameterised queries. This method is mostly useful for backend implementers.

Usage

```
sqlData(con, value, row.names = NA, ...)
```

Arguments

`con`	A database connection.
`value`	A data frame
`row.names`	Either `TRUE`, `FALSE`, `NA` or a string. If `TRUE`, always translate row names to a column called "row_names". If `FALSE`, never translate row names. If `NA`, translate rownames only if they're a character vector. A string is equivalent to `TRUE`, but allows you to override the default name. For backward compatibility, `NULL` is equivalent to `FALSE`.
`...`	Other arguments used by individual methods.

Details

The default method:

- Converts factors to characters
- Quotes all strings
- Converts all columns to strings
- Replaces NA with NULL

Examples

```
con <- dbConnect(RSQLite::SQLite(), ":memory:")

sqlData(con, head(iris))
sqlData(con, head(mtcars))

dbDisconnect(con)
```

---

sqlInterpolate *Safely interpolate values into an SQL string*

Description

Safely interpolate values into an SQL string

Usage

```
sqlInterpolate(`_con`, `_sql`, ...)
```

Arguments

`_con`	A database connection.
`...`	Named values to interpolate into string. All strings will be first escaped with `dbQuoteString` prior to interpolation to protect against SQL injection attacks.
`‘_sql‘`	A SQL string containing containing variables to interpolate. Variables must start with a question mark and can be any valid R identifier, i.e. it must start with a letter or `.`, and be followed by a letter, digit, `.` or `_`.

Backend authors

If you are implementing a SQL backend with non-ANSI quoting rules, you’ll need to implement a method for `sqlParseVariables`. Failure to do so does not expose you to SQL injection attacks, but will (rarely) result in errors matching supplied and interpolated variables.

Examples

```
sql <- "SELECT * FROM X WHERE name = ?name"
sqlInterpolate(ANSI(), sql, name = "Hadley")

# This is safe because the single quote has been double escaped
sqlInterpolate(ANSI(), sql, name = "H'); DROP TABLE--;")
```

`Table-class` *Refer to a table nested in a hierarchy (e.g. within a schema)*

Description

Refer to a table nested in a hierarchy (e.g. within a schema)

Usage

```
Table(...)
```

Arguments

`...`	Components of the hierarchy, e.g. `schema`, `table`, or `cluster`, `catalog`, `schema`, `table`. For more on these concepts, see `http://stackoverflow.com/questions/7022755/`

`transactions` *Begin/commit/rollback SQL transactions*

Description

Not all database engines implement transaction management, in which case these methods should not be implemented for the specific `DBIConnection` subclass.

Usage

```
dbBegin(conn, ...)

dbCommit(conn, ...)

dbRollback(conn, ...)
```

Arguments

`conn`	A `DBIConnection` object, as produced by `dbConnect`.
`...`	Other parameters passed on to methods.

Value

a logical indicating whether the operation succeeded or not.

Side Effects

The current transaction on the connection con is committed or rolled back.

See Also

Self-contained transactions: dbWithTransaction

Examples

```
## Not run:
ora <- dbDriver("Oracle")
con <- dbConnect(ora)

rs <- dbSendQuery(con,
      "delete * from PURGE as p where p.wavelength<0.03")
if (dbGetRowsAffected(rs) > 250) {
  warning("dubious deletion -- rolling back transaction")
  dbRollback(con)
} else {
  dbCommit(con)
}

dbClearResult(rs)
dbDisconnect(con)

## End(Not run)
```

Index