Package 'DBI'

December 15, 2019

Title R Database Interface

Version 1.1.0

Date 2019-12-15

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.

License LGPL (>= 2.1)

URL http://r-dbi.github.io/DBI, https://github.com/r-dbi/DBI

BugReports https://github.com/r-dbi/DBI/issues

Depends methods,
R (>= 3.0.0)

Suggests blob,
covr,
hms,
knitr,
magrittr,
rmarkdown,
rprojroot,
RSQLite (>= 1.1-2),
testthat,
xml2

VignetteBuilder knitr

Encoding UTF-8

Roxygen list(markdown = TRUE, roclets = c(``collate'',
``namespace'', ``rd''))

RoxygenNote 7.0.2

Collate 'hidden.R'
'DBObject.R'
'DBDriver.R'
'table.R'

'DBConnection.R'
'ANSI.R'
'DBConnector.R'
'DBI-package.R'
'DBResult.R'
'data-types.R'
'data.R'
'deprecated.R'
'interpolate.R'
'list-pairs.R'
'quote.R'
'rd.R'
'rownames.R'
'table-create.R'
'table-insert.R'
'transactions.R'

R topics documented:

`DBI-package` *DBI: R Database Interface*

Description

DBI defines an interface 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 (so-called *DBI backends*).

Definition

A DBI backend is an R package which imports 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 backend author to adopt this convention or not.

DBI classes and methods

A backend defines three classes, which are subclasses of DBIDriver, DBIConnection, and DBIResult. The backend provides implementation for all methods of these base classes that are defined but not implemented by DBI. All methods defined in **DBI** are reexported (so that the package can be used without having to attach **DBI**), and have an ellipsis . . . in their formals for extensibility.

Construction of the DBIDriver object

The backend must support creation of an instance of its DBIDriver subclass 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. However, backend authors may choose a different name. The constructor must be exported, and it must be a function that is callable without arguments. DBI recommends to define a constructor with an empty argument list.

Author(s)

Maintainer: Kirill Müller <krlmlr+r@mailbox.org> (ORCID)

Authors:

- R Special Interest Group on Databases (R-SIG-DB)
- Hadley Wickham

Other contributors:

- R Consortium [funder]

See Also

Important generics: `dbConnect()`, `dbGetQuery()`, `dbReadTable()`, `dbWriteTable()`, `dbDisconnect()`

Formal specification (currently work in progress and incomplete): `vignette("spec",package = "DBI")`

Examples

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

---

dbAppendTable *Insert rows into a table*

Description

The `dbAppendTable()` method assumes that the table has been created beforehand, e.g. with `dbCreateTable()`. The default implementation calls `sqlAppendTableTemplate()` and then `dbExecute()` with the `param` argument. Backends compliant to ANSI SQL 99 which use ? as a placeholder for prepard queries don't need to override it. Backends with a different SQL syntax which use ? as a placeholder for prepared queries can override `sqlAppendTable()`. Other backends (with different placeholders or with entirely different ways to create tables) need to override the `dbAppendTable()` method.

Usage

```
dbAppendTable(conn, name, value, ..., row.names = NULL)
```

Arguments

conn	A DBIConnection object, as returned by dbConnect().
name	Name of the table, escaped with dbQuoteIdentifier().
value	A data frame of values. The column names must be consistent with those in the target table in the database.
...	Other arguments used by individual methods.
row.names	Must be NULL.

Details

The row.names argument is not supported by this method. Process the values with sqlRownamesToColumn() before calling this method.

See Also

Other DBIConnection generics: DBIConnection-class, dbCreateTable(), dbDataType(), dbDisconnect(), dbExecute(), dbExistsTable(), dbGetException(), dbGetInfo(), dbGetQuery(), dbIsReadOnly(), dbIsValid(), dbListFields(), dbListObjects(), dbListResults(), dbListTables(), dbReadTable(), dbRemoveTable(), dbSendQuery(), dbSendStatement(), dbWriteTable()

Examples

```
con <- dbConnect(RSQLite::SQLite(), ":memory:")
dbCreateTable(con, "iris", iris)
dbAppendTable(con, "iris", iris)
dbReadTable(con, "iris")
dbDisconnect(con)
```

---

dbBind *Bind values to a parameterized/prepared statement*

Description

For parametrized or prepared statements, the dbSendQuery() and dbSendStatement() functions can be called with statements that contain placeholders for values. The dbBind() function binds these placeholders to actual values, and is intended to be called on the result set before calling dbFetch() or dbGetRowsAffected().

Usage

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

Arguments

res	An object inheriting from DBIResult.
params	A list of bindings, named or unnamed.
...	Other arguments passed on to methods.

Details

DBI supports parametrized (or prepared) queries and statements via the `dbBind()` generic. Parametrized queries are different from normal queries in that they allow an arbitrary number of placeholders, which are later substituted by actual values. Parametrized queries (and statements) serve two purposes:

- The same query can be executed more than once with different values. The DBMS may cache intermediate information for the query, such as the execution plan, and execute it faster.
- Separation of query syntax and parameters protects against SQL injection.

The placeholder format is currently not specified by **DBI**; in the future, a uniform placeholder syntax may be supported. Consult the backend documentation for the supported formats. For automated testing, backend authors specify the placeholder syntax with the `placeholder_pattern` tweak. Known examples are:

- ? (positional matching in order of appearance) in **RMySQL** and **RSQLite**
- $1 (positional matching by index) in **RPostgres** and **RSQLite**
- :name and $name (named matching) in **RSQLite**

Value

`dbBind()` returns the result set, invisibly, for queries issued by `dbSendQuery()` and also for data manipulation statements issued by `dbSendStatement()`. Calling `dbBind()` for a query without parameters raises an error. Binding too many or not enough values, or parameters with wrong names or unequal length, also raises an error. If the placeholders in the query are named, all parameter values must have names (which must not be empty or `NA`), and vice versa, otherwise an error is raised. The behavior for mixing placeholders of different types (in particular mixing positional and named placeholders) is not specified.

Calling `dbBind()` on a result set already cleared by `dbClearResult()` also raises an error.

Specification

DBI clients execute parametrized statements as follows:

1. Call `dbSendQuery()` or `dbSendStatement()` with a query or statement that contains placeholders, store the returned DBIResult object in a variable. Mixing placeholders (in particular, named and unnamed ones) is not recommended. It is good practice to register a call to `dbClearResult()` via `on.exit()` right after calling `dbSendQuery()` or `dbSendStatement()` (see the last enumeration item). Until `dbBind()` has been called, the returned result set object has the following behavior:
   - `dbFetch()` raises an error (for `dbSendQuery()`)
   - `dbGetRowCount()` returns zero (for `dbSendQuery()`)
   - `dbGetRowsAffected()` returns an integer `NA` (for `dbSendStatement()`)
   - `dbIsValid()` returns `TRUE`
   - `dbHasCompleted()` returns `FALSE`
2. Construct a list with parameters that specify 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 parameters, unnamed values are matched by position in the list of parameters. All

elements in this list must have the same lengths and contain values supported by the backend; a data.frame is internally stored as such a list. The parameter list is passed to a call to `dbBind()` on the `DBIResult` object.

3. Retrieve the data or the number of affected rows from the `DBIResult` object.
   - For queries issued by `dbSendQuery()`, call `dbFetch()`.
   - For statements issued by `dbSendStatements()`, call `dbGetRowsAffected()`. (Execution begins immediately after the `dbBind()` call, the statement is processed entirely before the function returns.)
4. Repeat 2. and 3. as necessary.
5. Close the result set via `dbClearResult()`.

The elements of the `params` argument do not need to be scalars, vectors of arbitrary length (including length 0) are supported. For queries, calling `dbFetch()` binding such parameters returns concatenated results, equivalent to binding and fetching for each set of values and connecting via `rbind()`. For data manipulation statements, `dbGetRowsAffected()` returns the total number of rows affected if binding non-scalar parameters. `dbBind()` also accepts repeated calls on the same result set for both queries and data manipulation statements, even if no results are fetched between calls to `dbBind()`.

If the placeholders in the query are named, their order in the `params` argument is not important.

At least the following data types are accepted on input (including NA):

- integer
- numeric
- logical for Boolean values
- character
- factor (bound as character, with warning)
- Date
- POSIXct timestamps
- POSIXlt timestamps
- lists of raw for blobs (with `NULL` entries for SQL NULL values)
- objects of type blob::blob

See Also

Other DBIResult generics: `DBIResult-class`, `dbClearResult()`, `dbColumnInfo()`, `dbFetch()`, `dbGetInfo()`, `dbGetRowCount()`, `dbGetRowsAffected()`, `dbGetStatement()`, `dbHasCompleted()`, `dbIsReadOnly()`, `dbIsValid()`, `dbQuoteIdentifier()`, `dbQuoteLiteral()`, `dbQuoteString()`, `dbUnquoteIdentifier()`

Examples

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

dbWriteTable(con, "iris", iris)
```

```
# Using the same query for different values
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)

# Executing the same statement with different values at once
iris_result <- dbSendStatement(con, "DELETE FROM iris WHERE [Species] = $species")
dbBind(iris_result, list(species = c("setosa", "versicolor", "unknown")))
dbGetRowsAffected(iris_result)
dbClearResult(iris_result)

nrow(dbReadTable(con, "iris"))

dbDisconnect(con)
```

`dbCanConnect` *Check if a connection to a DBMS can be established*

Description

Like `dbConnect()`, but only checks validity without actually returning a connection object. The default implementation opens a connection and disconnects on success, but individual backends might implement a lighter-weight check.

Usage

```
dbCanConnect(drv, ...)
```

Arguments

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

Value

A scalar logical. If `FALSE`, the `"reason"` attribute indicates a reason for failure.

See Also

Other DBIDriver generics: `DBIDriver-class`, `dbConnect()`, `dbDataType()`, `dbDriver()`, `dbGetInfo()`, `dbIsReadOnly()`, `dbIsValid()`, `dbListConnections()`

Examples

```
# SQLite only needs a path to the database. (Here, ":memory:" is a special
# path that creates an in-memory database.) Other database drivers
# will require more details (like user, password, host, port, etc.)
dbCanConnect(RSQLite::SQLite(), ":memory:")
```

`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

`dbClearResult()` returns `TRUE`, invisibly, for result sets obtained from both `dbSendQuery()` and `dbSendStatement()`. An attempt to close an already closed result set issues a warning in both cases.

Specification

`dbClearResult()` frees all resources associated with retrieving the result of a query or update operation. The DBI backend can expect a call to `dbClearResult()` for each `dbSendQuery()` or `dbSendStatement()` call.

See Also

Other DBIResult generics: `DBIResult-class`, `dbBind()`, `dbColumnInfo()`, `dbFetch()`, `dbGetInfo()`, `dbGetRowCount()`, `dbGetRowsAffected()`, `dbGetStatement()`, `dbHasCompleted()`, `dbIsReadOnly()`, `dbIsValid()`, `dbQuoteIdentifier()`, `dbQuoteLiteral()`, `dbQuoteString()`, `dbUnquoteIdentifier()`

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 describes 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

`dbColumnInfo()` returns a data frame with at least two columns `"name"` and `"type"` (in that order) (and optional columns that start with a dot). The `"name"` and `"type"` columns contain the names and types of the R columns of the data frame that is returned from `dbFetch()`. The `"type"` column is of type `character` and only for information. Do not compute on the `"type"` column, instead use `dbFetch(res,n = 0)` to create a zero-row data frame initialized with the correct data types.

An attempt to query columns for a closed result set raises an error.

Specification

A column named `row_names` is treated like any other column.

The column names are always consistent with the data returned by `dbFetch()`. If the query returns unnamed columns, unique non-empty and non-`NA` names are assigned. In the case of a duplicate column name, the first occurrence retains the original name, and unique names are assigned for the other occurrences. Column names that correspond to SQL or R keywords are left unchanged.

See Also

Other DBIResult generics: DBIResult-class, dbBind(), dbClearResult(), dbFetch(), dbGetInfo(), dbGetRowCount(), dbGetRowsAffected(), dbGetStatement(), dbHasCompleted(), dbIsReadOnly(), dbIsValid(), dbQuoteIdentifier(), dbQuoteLiteral(), dbQuoteString(), dbUnquoteIdentifier()

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 authentication 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 authentication mechanism is left unspecified, so check the documentation of individual drivers for details. Use dbCanConnect() to check if a connection can be established.

Usage

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

Arguments

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

Value

dbConnect() returns an S4 object that inherits from DBIConnection. This object is used to communicate with the database engine.

A format() method is defined for the connection object. It returns a string that consists of a single line of text.

Specification

DBI recommends using the following argument names for authentication parameters, with `NULL` default:

- `user` for the user name (default: current user)
- `password` for the password
- `host` for the host name (default: local connection)
- `port` for the port number (default: local connection)
- `dbname` for the name of the database on the host, or the database file name

The defaults should provide reasonable behavior, in particular a local connection for `host = NULL`. For some DBMS (e.g., PostgreSQL), this is different to a TCP/IP connection to `localhost`.

In addition, DBI supports the `bigint` argument that governs how 64-bit integer data is returned. The following values are supported:

- `"integer"`: always return as `integer`, silently overflow
- `"numeric"`: always return as `numeric`, silently round
- `"character"`: always return the decimal representation as `character`
- `"integer64"`: return as a data type that can be coerced using `as.integer()` (with warning on overflow), `as.numeric()` and `as.character()`

See Also

`dbDisconnect()` to disconnect from a database.

Other DBIDriver generics: `DBIDriver-class`, `dbCanConnect()`, `dbDataType()`, `dbDriver()`, `dbGetInfo()`, `dbIsReadOnly()`, `dbIsValid()`, `dbListConnections()`

Other DBIConnector generics: `DBIConnector-class`, `dbDataType()`, `dbGetConnectArgs()`, `dbIsReadOnly()`

Examples

```
# SQLite only needs a path to the database. (Here, ":memory:" is a special
# path that creates an in-memory database.) Other database drivers
# will require more details (like user, password, host, port, etc.)
con <- dbConnect(RSQLite::SQLite(), ":memory:")
con

dbListTables(con)

dbDisconnect(con)
```

dbCreateTable *Create a table in the database*

Description

The default dbCreateTable() method calls sqlCreateTable() and dbExecute(). Backends compliant to ANSI SQL 99 don't need to override it. Backends with a different SQL syntax can override sqlCreateTable(), backends with entirely different ways to create tables need to override this method.

Usage

```
dbCreateTable(conn, name, fields, ..., row.names = NULL, temporary = FALSE)
```

Arguments

conn	A DBIConnection object, as returned by dbConnect().
name	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().
...	Other arguments used by individual methods.
row.names	Must be NULL.
temporary	If TRUE, will generate a temporary table statement.

Details

The row.names argument is not supported by this method. Process the values with sqlRownamesToColumn() before calling this method.

The argument order is different from the sqlCreateTable() method, the latter will be adapted in a later release of DBI.

See Also

Other DBIConnection generics: DBIConnection-class, dbAppendTable(), dbDataType(), dbDisconnect(), dbExecute(), dbExistsTable(), dbGetException(), dbGetInfo(), dbGetQuery(), dbIsReadOnly(), dbIsValid(), dbListFields(), dbListObjects(), dbListResults(), dbListTables(), dbReadTable(), dbRemoveTable(), dbSendQuery(), dbSendStatement(), dbWriteTable()

Examples

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

dbDataType *Determine the SQL data type of an object*

Description

Returns an SQL string that describes the SQL data type to be used for an object. The default implementation of this generic determines the SQL type of an R object according to the SQL 92 specification, which may serve as a starting point for driver implementations. DBI also provides an implementation 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 primitive 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

`dbDataType()` returns the SQL type that corresponds to the `obj` argument as a non-empty character string. For data frames, a character vector with one element per column is returned. An error is raised for invalid values for the `obj` argument such as a `NULL` value.

Specification

The backend can override the `dbDataType()` generic for its driver class.

This generic expects an arbitrary object as second argument. 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. If the database supports blobs, this method also must accept lists of raw vectors, and blob::blob objects. As-is objects (i.e., wrapped by `I()`) must be supported and return the same results as their

unwrapped counterparts. The SQL data type for factor and ordered is the same as for character. The behavior for other object types is not specified.

All data types returned by `dbDataType()` are usable in an SQL statement of the form `"CREATE TABLE test (a ...)"`.

See Also

Other DBIDriver generics: `DBIDriver-class`, `dbCanConnect()`, `dbConnect()`, `dbDriver()`, `dbGetInfo()`, `dbIsReadOnly()`, `dbIsValid()`, `dbListConnections()`

Other DBIConnection generics: `DBIConnection-class`, `dbAppendTable()`, `dbCreateTable()`, `dbDisconnect()`, `dbExecute()`, `dbExistsTable()`, `dbGetException()`, `dbGetInfo()`, `dbGetQuery()`, `dbIsReadOnly()`, `dbIsValid()`, `dbListFields()`, `dbListObjects()`, `dbListResults()`, `dbListTables()`, `dbReadTable()`, `dbRemoveTable()`, `dbSendQuery()`, `dbSendStatement()`, `dbWriteTable()`

Other DBIConnector generics: `DBIConnector-class`, `dbConnect()`, `dbGetConnectArgs()`, `dbIsReadOnly()`

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 returned by dbConnect().
...	Other parameters passed on to methods.

Value

dbDisconnect() returns TRUE, invisibly.

Specification

A warning is issued on garbage collection when a connection has been released without calling dbDisconnect(), but this cannot be tested automatically. A warning is issued immediately when calling dbDisconnect() on an already disconnected or invalid connection.

See Also

Other DBIConnection generics: DBIConnection-class, dbAppendTable(), dbCreateTable(), dbDataType(), dbExecute(), dbExistsTable(), dbGetException(), dbGetInfo(), dbGetQuery(), dbIsReadOnly(), dbIsValid(), dbListFields(), dbListObjects(), dbListResults(), dbListTables(), dbReadTable(), dbRemoveTable(), dbSendQuery(), dbSendStatement(), dbWriteTable()

Examples

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

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

Description

Executes a statement and returns the number of rows affected. `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 returned by `dbConnect()`.
`statement`	a character string containing SQL.
`...`	Other parameters passed on to methods.

Details

You can also use `dbExecute()` to call a stored procedure that performs data manipulation or other actions that do not return a result set. To execute a stored procedure that returns a result set use `dbGetQuery()` instead.

Value

`dbExecute()` always returns a scalar numeric that specifies the number of rows affected by the statement. An error is raised when issuing a statement over a closed or invalid connection, if the syntax of the statement is invalid, or if the statement is not a non-`NA` string.

Implementation notes

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

Additional arguments

The following arguments are not part of the `dbExecute()` generic (to improve compatibility across backends) but are part of the DBI specification:

- `params` (default: `NULL`)
- `immediate` (default: `NULL`)

They must be provided as named arguments. See the "Specification" sections for details on their usage.

Specification

The `param` argument allows passing query parameters, see `dbBind()` for details.

Specification for the `immediate` argument

The `immediate` argument supports distinguishing between "direct" and "prepared" APIs offered by many database drivers. Passing `immediate = TRUE` leads to immediate execution of the query or statement, via the "direct" API (if supported by the driver). The default `NULL` means that the backend should choose whatever API makes the most sense for the database, and (if relevant) tries the other API if the first attempt fails. A successful second attempt should result in a message that suggests passing the correct `immediate` argument. Examples for possible behaviors:

1. DBI backend defaults to `immediate = TRUE` internally
   (a) A query without parameters is passed: query is executed
   (b) A query with parameters is passed:
      i. `params` not given: rejected immediately by the database because of a syntax error in the query, the backend tries `immediate = FALSE` (and gives a message)
      ii. `params` given: query is executed using `immediate = FALSE`
2. DBI backend defaults to `immediate = FALSE` internally
   (a) A query without parameters is passed:
      i. simple query: query is executed
      ii. "special" query (such as setting a config options): fails, the backend tries `immediate = TRUE` (and gives a message)
   (b) A query with parameters is passed:
      i. `params` not given: waiting for parameters via `dbBind()`
      ii. `params` given: query is executed

See Also

For queries: `dbSendQuery()` and `dbGetQuery()`.

Other DBIConnection generics: `DBIConnection-class`, `dbAppendTable()`, `dbCreateTable()`, `dbDataType()`, `dbDisconnect()`, `dbExistsTable()`, `dbGetException()`, `dbGetInfo()`, `dbGetQuery()`, `dbIsReadOnly()`, `dbIsValid()`, `dbListFields()`, `dbListObjects()`, `dbListResults()`, `dbListTables()`, `dbReadTable()`, `dbRemoveTable()`, `dbSendQuery()`, `dbSendStatement()`, `dbWriteTable()`

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
```

```
# Pass values using the param argument:
dbExecute(
  con,
  "INSERT INTO cars (speed, dist) VALUES (?, ?)",
  params = list(4:7, 5:8)
)
dbReadTable(con, "cars")   # there are now 10 rows

dbDisconnect(con)
```

---

dbExistsTable *Does a table exist?*

Description

Returns if a table given by name exists in the database.

Usage

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

Arguments

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

Value

dbExistsTable() returns a logical scalar, TRUE if the table or view specified by the name argument exists, FALSE otherwise. This includes temporary tables if supported by the database.

An error is raised when calling this method for a closed or invalid connection. An error is also raised if name cannot be processed with dbQuoteIdentifier() or if this results in a non-scalar.

Specification

The name argument is processed as follows, to support databases that allow non-syntactic names for their objects:

- If an unquoted table name as string: dbExistsTable() will do the quoting, perhaps by calling dbQuoteIdentifier(conn,x = name)
- If the result of a call to dbQuoteIdentifier(): no more quoting is done

For all tables listed by dbListTables(), dbExistsTable() returns TRUE.

See Also

Other DBIConnection generics: DBIConnection-class, dbAppendTable(), dbCreateTable(), dbDataType(), dbDisconnect(), dbExecute(), dbGetException(), dbGetInfo(), dbGetQuery(), dbIsReadOnly(), dbIsValid(), dbListFields(), dbListObjects(), dbListResults(), dbListTables(), dbReadTable(), dbRemoveTable(), dbSendQuery(), dbSendStatement(), dbWriteTable()

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, created by dbSendQuery().
n	maximum number of records to retrieve per fetch. Use n = -1 or n = Inf 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 implementation for dbFetch() calls fetch() so that it is compatible with existing code. Modern backends should implement for dbFetch() only.

Value

`dbFetch()` always returns a data.frame with as many rows as records were fetched and as many columns as fields in the result set, even if the result is a single value or has one or zero rows. An attempt to fetch from a closed result set raises an error. If the `n` argument is not an atomic whole number greater or equal to -1 or Inf, an error is raised, but a subsequent call to `dbFetch()` with proper `n` argument succeeds. Calling `dbFetch()` on a result set from a data manipulation query created by `dbSendStatement()` can be fetched and return an empty data frame, with a warning.

Specification

Fetching multi-row queries with one or more columns by default returns the entire result. Multi-row queries can also be fetched progressively by passing a whole number (integer or numeric) as the `n` argument. A value of Inf for the `n` argument is supported and also returns the full result. If more rows than available are fetched, the result is returned in full without warning. If fewer rows than requested are returned, further fetches will return a data frame with zero rows. If zero rows are fetched, the columns of the data frame are still fully typed. Fetching fewer rows than available is permitted, no warning is issued when clearing the result set.

A column named `row_names` is treated like any other column.

The column types of the returned data frame depend on the data returned:

- integer (or coercible to an integer) for integer values between -2^31 and 2^31 - 1, with NA for SQL `NULL` values
- numeric for numbers with a fractional component, with NA for SQL `NULL` values
- logical for Boolean values (some backends may return an integer); with NA for SQL `NULL` values
- character for text, with NA for SQL `NULL` values
- lists of raw for blobs with NULL entries for SQL NULL values
- coercible using `as.Date()` for dates, with NA for SQL `NULL` values (also applies to the return value of the SQL function `current_date`)
- coercible using `hms::as_hms()` for times, with NA for SQL `NULL` values (also applies to the return value of the SQL function `current_time`)
- coercible using `as.POSIXct()` for timestamps, with NA for SQL `NULL` values (also applies to the return value of the SQL function `current_timestamp`)

If dates and timestamps are supported by the backend, the following R types are used:

- Date for dates (also applies to the return value of the SQL function `current_date`)
- POSIXct for timestamps (also applies to the return value of the SQL function `current_timestamp`)

R has no built-in type with lossless support for the full range of 64-bit or larger integers. If 64-bit integers are returned from a query, the following rules apply:

- Values are returned in a container with support for the full range of valid 64-bit values (such as the `integer64` class of the **bit64** package)
- Coercion to numeric always returns a number that is as close as possible to the true value
- Loss of precision when converting to numeric gives a warning
- Conversion to character always returns a lossless decimal representation of the data

See Also

Close the result set with dbClearResult() as soon as you finish retrieving the records you want.

Other DBIResult generics: DBIResult-class, dbBind(), dbClearResult(), dbColumnInfo(), dbGetInfo(), dbGetRowCount(), dbGetRowsAffected(), dbGetStatement(), dbHasCompleted(), dbIsReadOnly(), dbIsValid(), dbQuoteIdentifier(), dbQuoteLiteral(), dbQuoteString(), dbUnquoteIdentifier()

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)
```

---

dbGetConnectArgs *Get connection arguments*

Description

Returns the arguments stored in a DBIConnector object for inspection, optionally evaluating them. This function is called by dbConnect() and usually does not need to be called directly.

Usage

```
dbGetConnectArgs(drv, eval = TRUE, ...)
```

Arguments

drv	A object inheriting from DBIConnector.
eval	Set to FALSE to return the functions that generate the argument instead of evaluating them.
...	Other arguments passed on to methods. Not otherwise used.

See Also

Other DBIConnector generics: DBIConnector-class, dbConnect(), dbDataType(), dbIsReadOnly()

Examples

```
cnr <- new("DBIConnector",
  .drv = RSQLite::SQLite(),
  .conn_args = list(dbname = ":memory:", password = function() "supersecret")
)
dbGetConnectArgs(cnr)
dbGetConnectArgs(cnr, eval = FALSE)
```

---

dbGetInfo *Get DBMS metadata*

Description

Retrieves information on objects of class DBIDriver, DBIConnection or DBIResult.

Usage

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

Arguments

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

Value

For objects of class DBIDriver, dbGetInfo() returns a named list that contains at least the following components:

- driver.version: the package version of the DBI backend,
- client.version: the version of the DBMS client library.

For objects of class DBIConnection, dbGetInfo() returns a named list that contains at least the following components:

- db.version: version of the database server,
- dbname: database name,
- username: username to connect to the database,
- host: hostname of the database server,
- port: port on the database server. It must not contain a password component. Components that are not applicable should be set to NA.

For objects of class DBIResult, dbGetInfo() returns a named list that contains at least the following components:

- statatment: the statement used with dbSendQuery() or dbExecute(), as returned by dbGetStatement(),
- row.count: the number of rows fetched so far (for queries), as returned by dbGetRowCount(),
- rows.affected: the number of rows affected (for statements), as returned by dbGetRowsAffected()
- has.completed: a logical that indicates if the query or statement has completed, as returned by dbHasCompleted().

Implementation notes

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, dbCanConnect(), dbConnect(), dbDataType(), dbDriver(), dbIsReadOnly(), dbIsValid(), dbListConnections()

Other DBIConnection generics: DBIConnection-class, dbAppendTable(), dbCreateTable(), dbDataType(), dbDisconnect(), dbExecute(), dbExistsTable(), dbGetException(), dbGetQuery(), dbIsReadOnly(), dbIsValid(), dbListFields(), dbListObjects(), dbListResults(), dbListTables(), dbReadTable(), dbRemoveTable(), dbSendQuery(), dbSendStatement(), dbWriteTable()

Other DBIResult generics: DBIResult-class, dbBind(), dbClearResult(), dbColumnInfo(), dbFetch(), dbGetRowCount(), dbGetRowsAffected(), dbGetStatement(), dbHasCompleted(), dbIsReadOnly(), dbIsValid(), dbQuoteIdentifier(), dbQuoteLiteral(), dbQuoteString(), dbUnquoteIdentifier()

---

dbGetQuery *Send query, retrieve results and then clear result set*

Description

Returns the result of a query as a data frame. dbGetQuery() comes with a default implementation (which should work with most backends) 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 returned by dbConnect().
statement	a character string containing SQL.
...	Other parameters passed on to methods.

Details

This method is for SELECT queries only (incl. other SQL statements that return a SELECT-alike result, e. g. execution of a stored procedure).

To execute a stored procedure that does not return a result set, use dbExecute().

Some backends may support data manipulation statements through this method for compatibility reasons. However, callers are strongly advised to use dbExecute() for data manipulation statements.

Value

dbGetQuery() always returns a data.frame with as many rows as records were fetched and as many columns as fields in the result set, even if the result is a single value or has one or zero rows. An error is raised when issuing a query over a closed or invalid connection, if the syntax of the query is invalid, or if the query is not a non-NA string. If the n argument is not an atomic whole number greater or equal to -1 or Inf, an error is raised, but a subsequent call to dbGetQuery() with proper n argument succeeds.

Implementation notes

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

Additional arguments

The following arguments are not part of the dbGetQuery() generic (to improve compatibility across backends) but are part of the DBI specification:

- n (default: -1)
- params (default: NULL)
- immediate (default: NULL)

They must be provided as named arguments. See the "Specification" and "Value" sections for details on their usage.

Specification

A column named row_names is treated like any other column.

The n argument specifies the number of rows to be fetched. If omitted, fetching multi-row queries with one or more columns returns the entire result. A value of Inf for the n argument is supported and also returns the full result. If more rows than available are fetched (by passing a too large value for n), the result is returned in full without warning. If zero rows are requested, the columns of the data frame are still fully typed. Fetching fewer rows than available is permitted, no warning is issued.

The param argument allows passing query parameters, see dbBind() for details.

Specification for the `immediate` argument

The `immediate` argument supports distinguishing between "direct" and "prepared" APIs offered by many database drivers. Passing `immediate = TRUE` leads to immediate execution of the query or statement, via the "direct" API (if supported by the driver). The default `NULL` means that the backend should choose whatever API makes the most sense for the database, and (if relevant) tries the other API if the first attempt fails. A successful second attempt should result in a message that suggests passing the correct `immediate` argument. Examples for possible behaviors:

1. DBI backend defaults to `immediate = TRUE` internally
   (a) A query without parameters is passed: query is executed
   (b) A query with parameters is passed:
      i. `params` not given: rejected immediately by the database because of a syntax error in the query, the backend tries `immediate = FALSE` (and gives a message)
      ii. `params` given: query is executed using `immediate = FALSE`
2. DBI backend defaults to `immediate = FALSE` internally
   (a) A query without parameters is passed:
      i. simple query: query is executed
      ii. "special" query (such as setting a config options): fails, the backend tries `immediate = TRUE` (and gives a message)
   (b) A query with parameters is passed:
      i. `params` not given: waiting for parameters via `dbBind()`
      ii. `params` given: query is executed

See Also

For updates: `dbSendStatement()` and `dbExecute()`.

Other DBIConnection generics: `DBIConnection-class`, `dbAppendTable()`, `dbCreateTable()`, `dbDataType()`, `dbDisconnect()`, `dbExecute()`, `dbExistsTable()`, `dbGetException()`, `dbGetInfo()`, `dbIsReadOnly()`, `dbIsValid()`, `dbListFields()`, `dbListObjects()`, `dbListResults()`, `dbListTables()`, `dbReadTable()`, `dbRemoveTable()`, `dbSendQuery()`, `dbSendStatement()`, `dbWriteTable()`

Examples

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

dbWriteTable(con, "mtcars", mtcars)
dbGetQuery(con, "SELECT * FROM mtcars")
dbGetQuery(con, "SELECT * FROM mtcars", n = 6)

# Pass values using the param argument:
# (This query runs eight times, once for each different
# parameter. The resulting rows are combined into a single
# data frame.)
dbGetQuery(
  con,
  "SELECT COUNT(*) FROM mtcars WHERE cyl = ?",
  params = list(1:8)
)
```

```
dbDisconnect(con)
```

---

dbGetRowCount *The number of rows fetched so far*

Description

Returns the total number of rows actually fetched with calls to dbFetch() for this result set.

Usage

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

Arguments

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

Value

dbGetRowCount() returns a scalar number (integer or numeric), the number of rows fetched so far. After calling dbSendQuery(), the row count is initially zero. After a call to dbFetch() without limit, the row count matches the total number of rows returned. Fetching a limited number of rows increases the number of rows by the number of rows returned, even if fetching past the end of the result set. For queries with an empty result set, zero is returned even after fetching. For data manipulation statements issued with dbSendStatement(), zero is returned before and after calling dbFetch(). Attempting to get the row count for a result set cleared with dbClearResult() gives an error.

See Also

Other DBIResult generics: DBIResult-class, dbBind(), dbClearResult(), dbColumnInfo(), dbFetch(), dbGetInfo(), dbGetRowsAffected(), dbGetStatement(), dbHasCompleted(), dbIsReadOnly(), dbIsValid(), dbQuoteIdentifier(), dbQuoteLiteral(), dbQuoteString(), dbUnquoteIdentifier()

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 method returns the number of rows that were added, deleted, or updated by a data manipulation statement.

Usage

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

Arguments

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

Value

dbGetRowsAffected() returns a scalar number (integer or numeric), the number of rows affected by a data manipulation statement issued with dbSendStatement(). The value is available directly after the call and does not change after calling dbFetch(). For queries issued with dbSendQuery(), zero is returned before and after the call to dbFetch(). Attempting to get the rows affected for a result set cleared with dbClearResult() gives an error.

See Also

Other DBIResult generics: DBIResult-class, dbBind(), dbClearResult(), dbColumnInfo(), dbFetch(), dbGetInfo(), dbGetRowCount(), dbGetStatement(), dbHasCompleted(), dbIsReadOnly(), dbIsValid(), dbQuoteIdentifier(), dbQuoteLiteral(), dbQuoteString(), dbUnquoteIdentifier()

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() or dbSendStatement().

Usage

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

Arguments

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

Value

dbGetStatement() returns a string, the query used in either dbSendQuery() or dbSendStatement(). Attempting to query the statement for a result set cleared with dbClearResult() gives an error.

See Also

Other DBIResult generics: DBIResult-class, dbBind(), dbClearResult(), dbColumnInfo(), dbFetch(), dbGetInfo(), dbGetRowCount(), dbGetRowsAffected(), dbHasCompleted(), dbIsReadOnly(), dbIsValid(), dbQuoteIdentifier(), dbQuoteLiteral(), dbQuoteString(), dbUnquoteIdentifier()

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 always completed.

Usage

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

Arguments

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

Value

dbHasCompleted() returns a logical scalar. For a query initiated by dbSendQuery() with non-empty result set, dbHasCompleted() returns FALSE initially and TRUE after calling dbFetch() without limit. For a query initiated by dbSendStatement(), dbHasCompleted() always returns TRUE. Attempting to query completion status for a result set cleared with dbClearResult() gives an error.

Specification

The completion status for a query is only guaranteed to be set to FALSE after attempting to fetch past the end of the entire result. Therefore, for a query with an empty result set, the initial return value is unspecified, but the result value is TRUE after trying to fetch only one row. Similarly, for a query with a result set of length n, the return value is unspecified after fetching n rows, but the result value is TRUE after trying to fetch only one more row.

See Also

Other DBIResult generics: DBIResult-class, dbBind(), dbClearResult(), dbColumnInfo(), dbFetch(), dbGetInfo(), dbGetRowCount(), dbGetRowsAffected(), dbGetStatement(), dbIsReadOnly(), dbIsValid(), dbQuoteIdentifier(), dbQuoteLiteral(), dbQuoteString(), dbUnquoteIdentifier()

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: DBIConnector-class, DBIDriver-class, DBIObject-class, DBIResult-class

Other DBIConnection generics: dbAppendTable(), dbCreateTable(), dbDataType(), dbDisconnect(), dbExecute(), dbExistsTable(), dbGetException(), dbGetInfo(), dbGetQuery(), dbIsReadOnly(), dbIsValid(), dbListFields(), dbListObjects(), dbListResults(), dbListTables(), dbReadTable(), dbRemoveTable(), dbSendQuery(), dbSendStatement(), dbWriteTable()

Examples

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

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

## End(Not run)
```

DBIConnector-class *DBIConnector class*

Description

Wraps objects of the DBIDriver class to include connection options. The purpose of this class is to store both the driver and the connection options. A database connection can be established with a call to `dbConnect()`, passing only that object without additional arguments.

Details

To prevent leakage of passwords and other credentials, this class supports delayed evaluation. All arguments can optionally be a function (callable without arguments). In such a case, the function is evaluated transparently when connecting in `dbGetConnectArgs()`.

See Also

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

Other DBIConnector generics: `dbConnect()`, `dbDataType()`, `dbGetConnectArgs()`, `dbIsReadOnly()`

Examples

```
# Create a connector:
cnr <- new("DBIConnector",
  .drv = RSQLite::SQLite(),
  .conn_args = list(dbname = ":memory:")
)
cnr

# Establish a connection through this connector:
con <- dbConnect(cnr)
con

# Access the database through this connection:
dbGetQuery(con, "SELECT 1 AS a")
dbDisconnect(con)
```

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 `RPostgres`, `RMariaDB`, etc. implement these operations in a DBMS-specific manner.

See Also

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

Other DBIDriver generics: dbCanConnect(), dbConnect(), dbDataType(), dbDriver(), dbGetInfo(), dbIsReadOnly(), 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, DBIConnector-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`, `DBIConnector-class`, `DBIDriver-class`, `DBIObject-class`

Other DBIResult generics: `dbBind()`, `dbClearResult()`, `dbColumnInfo()`, `dbFetch()`, `dbGetInfo()`, `dbGetRowCount()`, `dbGetRowsAffected()`, `dbGetStatement()`, `dbHasCompleted()`, `dbIsReadOnly()`, `dbIsValid()`, `dbQuoteIdentifier()`, `dbQuoteLiteral()`, `dbQuoteString()`, `dbUnquoteIdentifier()`

dbIsReadOnly *Is this DBMS object read only?*

Description

This generic tests whether a database object is read only.

Usage

```
dbIsReadOnly(dbObj, ...)
```

Arguments

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

See Also

Other DBIDriver generics: DBIDriver-class, dbCanConnect(), dbConnect(), dbDataType(), dbDriver(), dbGetInfo(), dbIsValid(), dbListConnections()

Other DBIConnection generics: DBIConnection-class, dbAppendTable(), dbCreateTable(), dbDataType(), dbDisconnect(), dbExecute(), dbExistsTable(), dbGetException(), dbGetInfo(), dbGetQuery(), dbIsValid(), dbListFields(), dbListObjects(), dbListResults(), dbListTables(), dbReadTable(), dbRemoveTable(), dbSendQuery(), dbSendStatement(), dbWriteTable()

Other DBIResult generics: DBIResult-class, dbBind(), dbClearResult(), dbColumnInfo(), dbFetch(), dbGetInfo(), dbGetRowCount(), dbGetRowsAffected(), dbGetStatement(), dbHasCompleted(), dbIsValid(), dbQuoteIdentifier(), dbQuoteLiteral(), dbQuoteString(), dbUnquoteIdentifier()

Other DBIConnector generics: DBIConnector-class, dbConnect(), dbDataType(), dbGetConnectArgs()

Examples

```
dbIsReadOnly(ANSI())
```

---

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

dbIsValid() returns a logical scalar, TRUE if the object specified by dbObj is valid, FALSE otherwise. A DBIConnection object is initially valid, and becomes invalid after disconnecting with dbDisconnect(). For an invalid connection object (e.g., for some drivers if the object is saved to a file and then restored), the method also returns FALSE. A DBIResult object is valid after a call to dbSendQuery(), and stays valid even after all rows have been fetched; only clearing it with dbClearResult() invalidates it. A DBIResult object is also valid after a call to dbSendStatement(), and stays valid after querying the number of rows affected; only clearing it with dbClearResult() invalidates it. If the connection to the database system is dropped (e.g., due to connectivity problems, server failure, etc.), dbIsValid() should return FALSE. This is not tested automatically.

See Also

Other DBIDriver generics: DBIDriver-class, dbCanConnect(), dbConnect(), dbDataType(), dbDriver(), dbGetInfo(), dbIsReadOnly(), dbListConnections()

Other DBIConnection generics: DBIConnection-class, dbAppendTable(), dbCreateTable(), dbDataType(), dbDisconnect(), dbExecute(), dbExistsTable(), dbGetException(), dbGetInfo(), dbGetQuery(), dbIsReadOnly(), dbListFields(), dbListObjects(), dbListResults(), dbListTables(), dbReadTable(), dbRemoveTable(), dbSendQuery(), dbSendStatement(), dbWriteTable()

Other DBIResult generics: DBIResult-class, dbBind(), dbClearResult(), dbColumnInfo(), dbFetch(), dbGetInfo(), dbGetRowCount(), dbGetRowsAffected(), dbGetStatement(), dbHasCompleted(), dbIsReadOnly(), dbQuoteIdentifier(), dbQuoteLiteral(), dbQuoteString(), dbUnquoteIdentifier()

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)
```

---

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 returned by dbConnect().
name	a character string with the name of the remote table.
...	Other parameters passed on to methods.

Value

dbListFields() returns a character vector that enumerates all fields in the table in the correct order. This also works for temporary tables if supported by the database. The returned names are suitable for quoting with dbQuoteIdentifier(). If the table does not exist, an error is raised. Invalid types for the name argument (e.g., character of length not equal to one, or numeric) lead to an error. An error is also raised when calling this method for a closed or invalid connection.

Specification

The name argument can be

- a string
- the return value of dbQuoteIdentifier()
- a value from the table column from the return value of dbListObjects() where is_prefix is FALSE

A column named row_names is treated like any other column.

See Also

dbColumnInfo() to get the type of the fields.

Other DBIConnection generics: DBIConnection-class, dbAppendTable(), dbCreateTable(), dbDataType(), dbDisconnect(), dbExecute(), dbExistsTable(), dbGetException(), dbGetInfo(), dbGetQuery(), dbIsReadOnly(), dbIsValid(), dbListObjects(), dbListResults(), dbListTables(), dbReadTable(), dbRemoveTable(), dbSendQuery(), dbSendStatement(), dbWriteTable()

Examples

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

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

dbDisconnect(con)
```

---

dbListObjects *List remote objects*

Description

Returns the names of remote objects accessible through this connection as a data frame. This should include temporary objects, but not all database backends (in particular **RMariaDB** and **RMySQL**) support this. Compared to dbListTables(), this method also enumerates tables and views in schemas, and returns fully qualified identifiers to access these objects. This allows exploration of all database objects available to the current user, including those that can only be accessed by giving the full namespace.

Usage

```
dbListObjects(conn, prefix = NULL, ...)
```

Arguments

conn	A DBIConnection object, as returned by dbConnect().
prefix	A fully qualified path in the database's namespace, or NULL. This argument will be processed with dbUnquoteIdentifier(). If given the method will return all objects accessible through this prefix.
...	Other parameters passed on to methods.

Value

dbListObjects() returns a data frame with columns table and is_prefix (in that order), optionally with other columns with a dot (.) prefix. The table column is of type list. Each object in this list is suitable for use as argument in dbQuoteIdentifier(). The is_prefix column is a logical. This data frame contains one row for each object (schema, table and view) accessible from the prefix (if passed) or from the global namespace (if prefix is omitted). Tables added with dbWriteTable() are part of the data frame, including temporary objects if supported by the database. As soon a table is removed from the database, it is also removed from the data frame of database objects.

The returned names are suitable for quoting with dbQuoteIdentifier(). An error is raised when calling this method for a closed or invalid connection.

Specification

The table object can be quoted with dbQuoteIdentifier(). The result of quoting can be passed to dbUnquoteIdentifier(). The unquoted results are equal to the original table object. (For backends it may be convenient to use the Id class, but this is not required.)

The prefix column indicates if the table value refers to a table or a prefix. For a call with the default prefix = NULL, the table values that have is_prefix == FALSE correspond to the tables returned from dbListTables(),

Values in table column that have is_prefix == TRUE can be passed as the prefix argument to another call to dbListObjects(). For the data frame returned from a dbListObject() call with the prefix argument set, all table values where is_prefix is FALSE can be used in a call to dbExistsTable() which returns TRUE.

See Also

Other DBIConnection generics: DBIConnection-class, dbAppendTable(), dbCreateTable(), dbDataType(), dbDisconnect(), dbExecute(), dbExistsTable(), dbGetException(), dbGetInfo(), dbGetQuery(), dbIsReadOnly(), dbIsValid(), dbListFields(), dbListResults(), dbListTables(), dbReadTable(), dbRemoveTable(), dbSendQuery(), dbSendStatement(), dbWriteTable()

Examples

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

dbListObjects(con)
```

```
dbWriteTable(con, "mtcars", mtcars)
dbListObjects(con)

dbDisconnect(con)
```

`dbListTables` *List remote tables*

Description

Returns the unquoted names of remote tables accessible through this connection. This should include views and temporary objects, but not all database backends (in particular **RMariaDB** and **RMySQL**) support this.

Usage

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

Arguments

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

Value

`dbListTables()` returns a character vector that enumerates all tables and views in the database. Tables added with `dbWriteTable()` are part of the list, including temporary tables if supported by the database. As soon a table is removed from the database, it is also removed from the list of database tables.

The returned names are suitable for quoting with `dbQuoteIdentifier()`. An error is raised when calling this method for a closed or invalid connection.

See Also

Other DBIConnection generics: `DBIConnection-class`, `dbAppendTable()`, `dbCreateTable()`, `dbDataType()`, `dbDisconnect()`, `dbExecute()`, `dbExistsTable()`, `dbGetException()`, `dbGetInfo()`, `dbGetQuery()`, `dbIsReadOnly()`, `dbIsValid()`, `dbListFields()`, `dbListObjects()`, `dbListResults()`, `dbReadTable()`, `dbRemoveTable()`, `dbSendQuery()`, `dbSendStatement()`, `dbWriteTable()`

Examples

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

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

dbDisconnect(con)
```

dbQuoteIdentifier *Quote identifiers*

Description

Call this method to generate a string that is suitable for use in a query as a column or table name, to make sure that you generate valid SQL and protect against SQL injection attacks. The inverse operation is dbUnquoteIdentifier().

Usage

```
dbQuoteIdentifier(conn, x, ...)
```

Arguments

conn	A DBIConnection object, as returned by dbConnect().
x	A character vector, SQL or Id object to quote as identifier.
...	Other arguments passed on to methods.

Value

dbQuoteIdentifier() returns an object that can be coerced to character, of the same length as the input. For an empty character vector this function returns a length-0 object. The names of the input argument are preserved in the output. When passing the returned object again to dbQuoteIdentifier() as x argument, it is returned unchanged. Passing objects of class SQL should also return them unchanged. (For backends it may be most convenient to return SQL objects to achieve this behavior, but this is not required.)

An error is raised if the input contains NA, but not for an empty string.

Specification

Calling dbGetQuery() for a query of the format SELECT 1 AS ... returns a data frame with the identifier, unquoted, as column name. Quoted identifiers can be used as table and column names in SQL queries, in particular in queries like SELECT 1 AS ... and SELECT * FROM (SELECT 1) .... The method must use a quoting mechanism that is unambiguously different from the quoting mechanism used for strings, so that a query like SELECT ... FROM (SELECT 1 AS ...) throws an error if the column names do not match.

The method can quote column names that contain special characters such as a space, a dot, a comma, or quotes used to mark strings or identifiers, if the database supports this. In any case, checking the validity of the identifier should be performed only when executing a query, and not by dbQuoteIdentifier().

See Also

Other DBIResult generics: DBIResult-class, dbBind(), dbClearResult(), dbColumnInfo(), dbFetch(), dbGetInfo(), dbGetRowCount(), dbGetRowsAffected(), dbGetStatement(), dbHasCompleted(), dbIsReadOnly(), dbIsValid(), dbQuoteLiteral(), dbQuoteString(), dbUnquoteIdentifier()

Examples

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

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

dbQuoteIdentifier(ANSI(), var_name)

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

---

dbQuoteLiteral *Quote literal values*

Description

Call these methods to generate a string that is suitable for use in a query as a literal value of the correct type, to make sure that you generate valid SQL and protect against SQL injection attacks.

Usage

```
dbQuoteLiteral(conn, x, ...)
```

Arguments

conn	A DBIConnection object, as returned by dbConnect().
x	A vector to quote as string.
...	Other arguments passed on to methods.

Value

dbQuoteLiteral() returns an object that can be coerced to character, of the same length as the input. For an empty character vector this function returns a length-0 object.

When passing the returned object again to dbQuoteLiteral() as x argument, it is returned unchanged. Passing objects of class SQL should also return them unchanged. (For backends it may be most convenient to return SQL objects to achieve this behavior, but this is not required.)

Specification

The returned expression can be used in a SELECT ... query, and the value of dbGetQuery(paste0("SELECT ",dbQuoteLiteral(x)))[[1]] must be equal to x for any scalar integer, numeric, string, and logical. If x is NA, the result must merely satisfy is.na(). The literals "NA" or "NULL" are not treated specially.

NA should be translated to an unquoted SQL NULL, so that the query SELECT * FROM (SELECT 1) a WHERE ... IS NULL returns one row.

Passing a list for the x argument raises an error.

See Also

Other DBIResult generics: DBIResult-class, dbBind(), dbClearResult(), dbColumnInfo(), dbFetch(), dbGetInfo(), dbGetRowCount(), dbGetRowsAffected(), dbGetStatement(), dbHasCompleted(), dbIsReadOnly(), dbIsValid(), dbQuoteIdentifier(), dbQuoteString(), dbUnquoteIdentifier()

Examples

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

# NAs become NULL
dbQuoteLiteral(ANSI(), c(1:3, NA))

# Logicals become integers by default
dbQuoteLiteral(ANSI(), c(TRUE, FALSE, NA))

# Raw vectors become hex strings by default
dbQuoteLiteral(ANSI(), list(as.raw(1:3), NULL))

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

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

---

dbQuoteString *Quote literal strings*

Description

Call this method to generate a string that is suitable for use in a query as a string literal, to make sure that you generate valid SQL and protect against SQL injection attacks.

Usage

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

Arguments

conn	A DBIConnection object, as returned by dbConnect().
x	A character vector to quote as string.
...	Other arguments passed on to methods.

Value

dbQuoteString() returns an object that can be coerced to character, of the same length as the input. For an empty character vector this function returns a length-0 object.

When passing the returned object again to dbQuoteString() as x argument, it is returned unchanged. Passing objects of class SQL should also return them unchanged. (For backends it may be most convenient to return SQL objects to achieve this behavior, but this is not required.)

Specification

The returned expression can be used in a SELECT ... query, and for any scalar character x the value of dbGetQuery(paste0("SELECT ",dbQuoteString(x)))[[1]] must be identical to x, even if x contains spaces, tabs, quotes (single or double), backticks, or newlines (in any combination) or is itself the result of a dbQuoteString() call coerced back to character (even repeatedly). If x is NA, the result must merely satisfy is.na(). The strings "NA" or "NULL" are not treated specially.

NA should be translated to an unquoted SQL NULL, so that the query SELECT * FROM (SELECT 1) a WHERE ... IS NULL returns one row.

Passing a numeric, integer, logical, or raw vector, or a list for the x argument raises an error.

See Also

Other DBIResult generics: DBIResult-class, dbBind(), dbClearResult(), dbColumnInfo(), dbFetch(), dbGetInfo(), dbGetRowCount(), dbGetRowsAffected(), dbGetStatement(), dbHasCompleted(), dbIsReadOnly(), dbIsValid(), dbQuoteIdentifier(), dbQuoteLiteral(), dbUnquoteIdentifier()

Examples

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

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

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

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

---

dbReadTable *Copy data frames from database tables*

Description

Reads a database table to a data frame, optionally converting a column to row names and converting the column names to valid R identifiers.

Usage

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

Arguments

conn	A DBIConnection object, as returned by dbConnect().
name	A character string specifying the unquoted DBMS table name, or the result of a call to dbQuoteIdentifier().
...	Other parameters passed on to methods.

Value

dbReadTable() returns a data frame that contains the complete data from the remote table, effectively the result of calling dbGetQuery() with SELECT * FROM <name>. An error is raised if the table does not exist. An empty table is returned as a data frame with zero rows.

The presence of rownames depends on the row.names argument, see sqlColumnToRownames() for details:

- If FALSE or NULL, the returned data frame doesn't have row names.
- If TRUE, a column named "row_names" is converted to row names, an error is raised if no such column exists.
- If NA, a column named "row_names" is converted to row names if it exists, otherwise no translation occurs.
- If a string, this specifies the name of the column in the remote table that contains the row names, an error is raised if no such column exists.

The default is row.names = FALSE.

If the database supports identifiers with special characters, the columns in the returned data frame are converted to valid R identifiers if the check.names argument is TRUE, otherwise non-syntactic column names can be returned unquoted.

An error is raised when calling this method for a closed or invalid connection. An error is raised if name cannot be processed with dbQuoteIdentifier() or if this results in a non-scalar. Unsupported values for row.names and check.names (non-scalars, unsupported data types, NA for check.names) also raise an error.

Additional arguments

The following arguments are not part of the dbReadTable() generic (to improve compatibility across backends) but are part of the DBI specification:

- row.names (default: FALSE)
- check.names

They must be provided as named arguments. See the "Value" section for details on their usage.

Specification

The name argument is processed as follows, to support databases that allow non-syntactic names for their objects:

- If an unquoted table name as string: dbReadTable() will do the quoting, perhaps by calling dbQuoteIdentifier(conn,x = name)
- If the result of a call to dbQuoteIdentifier(): no more quoting is done

See Also

Other DBIConnection generics: DBIConnection-class, dbAppendTable(), dbCreateTable(), dbDataType(), dbDisconnect(), dbExecute(), dbExistsTable(), dbGetException(), dbGetInfo(), dbGetQuery(), dbIsReadOnly(), dbIsValid(), dbListFields(), dbListObjects(), dbListResults(), dbListTables(), dbRemoveTable(), dbSendQuery(), dbSendStatement(), dbWriteTable()

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

Remove a remote table (e.g., created by dbWriteTable()) from the database.

Usage

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

Arguments

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

Value

dbRemoveTable() returns TRUE, invisibly. If the table does not exist, an error is raised. An attempt to remove a view with this function may result in an error.

An error is raised when calling this method for a closed or invalid connection. An error is also raised if name cannot be processed with dbQuoteIdentifier() or if this results in a non-scalar.

Additional arguments

The following arguments are not part of the dbRemoveTable() generic (to improve compatibility across backends) but are part of the DBI specification:

- temporary (default: FALSE)
- fail_if_missing (default: TRUE)

These arguments must be provided as named arguments.

If temporary is TRUE, the call to dbRemoveTable() will consider only temporary tables. Not all backends support this argument. In particular, permanent tables of the same name are left untouched.

If fail_if_missing is FALSE, the call to dbRemoveTable() succeeds if the table does not exist.

Specification

A table removed by dbRemoveTable() doesn't appear in the list of tables returned by dbListTables(), and dbExistsTable() returns FALSE. The removal propagates immediately to other connections to the same database. This function can also be used to remove a temporary table.

The name argument is processed as follows, to support databases that allow non-syntactic names for their objects:

- If an unquoted table name as string: dbRemoveTable() will do the quoting, perhaps by calling dbQuoteIdentifier(conn, x = name)
- If the result of a call to dbQuoteIdentifier(): no more quoting is done

See Also

Other DBIConnection generics: DBIConnection-class, dbAppendTable(), dbCreateTable(), dbDataType(), dbDisconnect(), dbExecute(), dbExistsTable(), dbGetException(), dbGetInfo(), dbGetQuery(), dbIsReadOnly(), dbIsValid(), dbListFields(), dbListObjects(), dbListResults(), dbListTables(), dbReadTable(), dbSendQuery(), dbSendStatement(), dbWriteTable()

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 dbSendQuery() method 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 dbFetch() method, 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 returned by dbConnect().
statement	a character string containing SQL.
...	Other parameters passed on to methods.

Details

This method is for SELECT queries only. Some backends may support data manipulation queries through this method for compatibility reasons. However, callers are strongly encouraged to use dbSendStatement() for data manipulation statements.

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.

Value

dbSendQuery() returns an S4 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 clear it with dbClearResult(). An error is raised when issuing a query over a closed or invalid connection, or if the query is not a non-NA string. An error is also raised if the syntax of the query is invalid and all query parameters are given (by passing the params argument) or the immediate argument is set to TRUE.

Additional arguments

The following arguments are not part of the dbSendQuery() generic (to improve compatibility across backends) but are part of the DBI specification:

- params (default: NULL)
- immediate (default: NULL)

They must be provided as named arguments. See the "Specification" sections for details on their usage.

Specification

No warnings occur under normal conditions. When done, the DBIResult object must be cleared with a call to `dbClearResult()`. Failure to clear the result set leads to a warning when the connection is closed.

If the backend supports only one open result set per connection, issuing a second query invalidates an already open result set and raises a warning. The newly opened result set is valid and must be cleared with `dbClearResult()`.

The `param` argument allows passing query parameters, see `dbBind()` for details.

Specification for the `immediate` argument

The `immediate` argument supports distinguishing between "direct" and "prepared" APIs offered by many database drivers. Passing `immediate = TRUE` leads to immediate execution of the query or statement, via the "direct" API (if supported by the driver). The default `NULL` means that the backend should choose whatever API makes the most sense for the database, and (if relevant) tries the other API if the first attempt fails. A successful second attempt should result in a message that suggests passing the correct `immediate` argument. Examples for possible behaviors:

1. DBI backend defaults to `immediate = TRUE` internally
   (a) A query without parameters is passed: query is executed
   (b) A query with parameters is passed:
      i. `params` not given: rejected immediately by the database because of a syntax error in the query, the backend tries `immediate = FALSE` (and gives a message)
      ii. `params` given: query is executed using `immediate = FALSE`
2. DBI backend defaults to `immediate = FALSE` internally
   (a) A query without parameters is passed:
      i. simple query: query is executed
      ii. "special" query (such as setting a config options): fails, the backend tries `immediate = TRUE` (and gives a message)
   (b) A query with parameters is passed:
      i. `params` not given: waiting for parameters via `dbBind()`
      ii. `params` given: query is executed

See Also

For updates: `dbSendStatement()` and `dbExecute()`.

Other DBIConnection generics: `DBIConnection-class`, `dbAppendTable()`, `dbCreateTable()`, `dbDataType()`, `dbDisconnect()`, `dbExecute()`, `dbExistsTable()`, `dbGetException()`, `dbGetInfo()`, `dbGetQuery()`, `dbIsReadOnly()`, `dbIsValid()`, `dbListFields()`, `dbListObjects()`, `dbListResults()`, `dbListTables()`, `dbReadTable()`, `dbRemoveTable()`, `dbSendStatement()`, `dbWriteTable()`

Examples

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

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

# Pass one set of values with the param argument:
rs <- dbSendQuery(
  con,
  "SELECT * FROM mtcars WHERE cyl = ?",
  params = list(4L)
)
dbFetch(rs)
dbClearResult(rs)

# Pass multiple sets of values with dbBind():
rs <- dbSendQuery(con, "SELECT * FROM mtcars WHERE cyl = ?")
dbBind(rs, list(6L))
dbFetch(rs)
dbBind(rs, list(8L))
dbFetch(rs)
dbClearResult(rs)

dbDisconnect(con)
```

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

Description

The `dbSendStatement()` method 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 returned by `dbConnect()`.
`statement`	a character string 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

`dbSendStatement()` returns an S4 object that inherits from DBIResult. The result set can be used with `dbGetRowsAffected()` to determine the number of rows affected by the query. Once you have finished using a result, make sure to clear it with `dbClearResult()`. An error is raised when issuing a statement over a closed or invalid connection, or if the statement is not a non-NA string. An error is also raised if the syntax of the query is invalid and all query parameters are given (by passing the `params` argument) or the `immediate` argument is set to `TRUE`.

Additional arguments

The following arguments are not part of the `dbSendStatement()` generic (to improve compatibility across backends) but are part of the DBI specification:

- `params` (default: `NULL`)
- `immediate` (default: `NULL`)

They must be provided as named arguments. See the "Specification" sections for details on their usage.

Specification

No warnings occur under normal conditions. When done, the DBIResult object must be cleared with a call to `dbClearResult()`. Failure to clear the result set leads to a warning when the connection is closed. If the backend supports only one open result set per connection, issuing a second query invalidates an already open result set and raises a warning. The newly opened result set is valid and must be cleared with `dbClearResult()`.

The `param` argument allows passing query parameters, see `dbBind()` for details.

Specification for the `immediate` argument

The `immediate` argument supports distinguishing between "direct" and "prepared" APIs offered by many database drivers. Passing `immediate = TRUE` leads to immediate execution of the query or statement, via the "direct" API (if supported by the driver). The default `NULL` means that the backend should choose whatever API makes the most sense for the database, and (if relevant) tries the other API if the first attempt fails. A successful second attempt should result in a message that suggests passing the correct `immediate` argument. Examples for possible behaviors:

1. DBI backend defaults to `immediate = TRUE` internally
   (a) A query without parameters is passed: query is executed
   (b) A query with parameters is passed:
      i. `params` not given: rejected immediately by the database because of a syntax error in the query, the backend tries `immediate = FALSE` (and gives a message)
      ii. `params` given: query is executed using `immediate = FALSE`

2. DBI backend defaults to `immediate = FALSE` internally
   (a) A query without parameters is passed:
      i. simple query: query is executed
      ii. "special" query (such as setting a config options): fails, the backend tries `immediate = TRUE` (and gives a message)
   (b) A query with parameters is passed:
      i. `params` not given: waiting for parameters via `dbBind()`
      ii. `params` given: query is executed

See Also

For queries: `dbSendQuery()` and `dbGetQuery()`.

Other DBIConnection generics: `DBIConnection-class`, `dbAppendTable()`, `dbCreateTable()`, `dbDataType()`, `dbDisconnect()`, `dbExecute()`, `dbExistsTable()`, `dbGetException()`, `dbGetInfo()`, `dbGetQuery()`, `dbIsReadOnly()`, `dbIsValid()`, `dbListFields()`, `dbListObjects()`, `dbListResults()`, `dbListTables()`, `dbReadTable()`, `dbRemoveTable()`, `dbSendQuery()`, `dbWriteTable()`

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

# Pass one set of values directly using the param argument:
rs <- dbSendStatement(
  con,
  "INSERT INTO cars (speed, dist) VALUES (?, ?)",
  params = list(4L, 5L)
)
dbClearResult(rs)

# Pass multiple sets of values using dbBind():
rs <- dbSendStatement(
  con,
  "INSERT INTO cars (speed, dist) VALUES (?, ?)"
)
dbBind(rs, list(5:6, 6:7))
dbBind(rs, list(7L, 8L))
dbClearResult(rs)
dbReadTable(con, "cars")   # there are now 10 rows

dbDisconnect(con)
```

dbUnquoteIdentifier *Unquote identifiers*

Description

Call this method to convert a SQL object created by dbQuoteIdentifier() back to a list of Id objects.

Usage

```
dbUnquoteIdentifier(conn, x, ...)
```

Arguments

conn	A DBIConnection object, as returned by dbConnect().
x	An SQL or Id object.
...	Other arguments passed on to methods.

Value

dbUnquoteIdentifier() returns a list of objects of the same length as the input. For an empty character vector this function returns a length-0 object. The names of the input argument are preserved in the output. When passing the first element of a returned object again to dbUnquoteIdentifier() as x argument, it is returned unchanged (but wrapped in a list). Passing objects of class Id should also return them unchanged (but wrapped in a list). (For backends it may be most convenient to return Id objects to achieve this behavior, but this is not required.)

An error is raised if plain character vectors are passed as the x argument.

Specification

For any character vector of length one, quoting (with dbQuoteIdentifier()) then unquoting then quoting the first element is identical to just quoting. This is also true for strings that contain special characters such as a space, a dot, a comma, or quotes used to mark strings or identifiers, if the database supports this.

Unquoting simple strings (consisting of only letters) wrapped with SQL() and then quoting via dbQuoteIdentifier() gives the same result as just quoting the string. Similarly, unquoting expressions of the form SQL("schema.table") and then quoting gives the same result as quoting the identifier constructed by Id(schema = "schema",table = "table").

See Also

Other DBIResult generics: DBIResult-class, dbBind(), dbClearResult(), dbColumnInfo(), dbFetch(), dbGetInfo(), dbGetRowCount(), dbGetRowsAffected(), dbGetStatement(), dbHasCompleted(), dbIsReadOnly(), dbIsValid(), dbQuoteIdentifier(), dbQuoteLiteral(), dbQuoteString()

Examples

```
# Unquoting allows to understand the structure of a
# possibly complex quoted identifier
dbUnquoteIdentifier(
  ANSI(),
  SQL(c('"Catalog"."Schema"."Table"', '"Schema"."Table"', '"UnqualifiedTable"'))
)

# The returned object is always a list,
# also for Id objects
dbUnquoteIdentifier(
  ANSI(),
  Id(catalog = "Catalog", schema = "Schema", table = "Table")
)

# Quoting is the inverse operation to unquoting the elements
# of the returned list
dbQuoteIdentifier(
  ANSI(),
  dbUnquoteIdentifier(ANSI(), SQL("UnqualifiedTable"))[[1]]
)

dbQuoteIdentifier(
  ANSI(),
  dbUnquoteIdentifier(ANSI(), Id(schema = "Schema", table = "Table"))[[1]]
)
```

---

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 returned by `dbConnect()`.
code	An arbitrary block of R code.
...	Other parameters passed on to methods.

Details

DBI implements `dbWithTransaction()`, backends should need to override this generic only if they implement specialized handling.

Value

`dbWithTransaction()` returns the value of the executed code. Failure to initiate the transaction (e.g., if the connection is closed or invalid of if `dbBegin()` has been called already) gives an error.

Specification

`dbWithTransaction()` initiates a transaction with `dbBegin()`, executes the code given in the `code` argument, and commits the transaction with `dbCommit()`. If the code raises an error, the transaction is instead aborted with `dbRollback()`, and the error is propagated. If the code calls `dbBreak()`, execution of the code stops and the transaction is silently aborted. All side effects caused by the code (such as the creation of new variables) propagate to the calling environment.

Examples

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

dbWriteTable(con, "cash", data.frame(amount = 100))
dbWriteTable(con, "account", data.frame(amount = 2000))

# All operations are carried out as logical unit:
dbWithTransaction(
  con,
  {
    withdrawal <- 300
    dbExecute(con, "UPDATE cash SET amount = amount + ?", list(withdrawal))
    dbExecute(con, "UPDATE account SET amount = amount - ?", list(withdrawal))
  }
)

# The code is executed as if in the curent environment:
withdrawal

# The changes are committed to the database after successful execution:
dbReadTable(con, "cash")
dbReadTable(con, "account")

# Rolling back with dbBreak():
dbWithTransaction(
  con,
  {
    withdrawal <- 5000
    dbExecute(con, "UPDATE cash SET amount = amount + ?", list(withdrawal))
    dbExecute(con, "UPDATE account SET amount = amount - ?", list(withdrawal))
    if (dbReadTable(con, "account")$amount < 0) {
      dbBreak()
    }
```

```
  }
)

# These changes were not committed to the database:
dbReadTable(con, "cash")
dbReadTable(con, "account")

dbDisconnect(con)
```

`dbWriteTable` *Copy data frames to database tables*

Description

Writes, overwrites or appends a data frame to a database table, optionally converting row names to a column and specifying SQL data types for fields. New code should prefer `dbCreateTable()` and `dbAppendTable()`.

Usage

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

Arguments

`conn`	A DBIConnection object, as returned by `dbConnect()`.
`name`	A character string specifying the unquoted DBMS table name, or the result of a call to `dbQuoteIdentifier()`.
`value`	a data.frame (or coercible to data.frame).
`...`	Other parameters passed on to methods.

Value

`dbWriteTable()` returns `TRUE`, invisibly. If the table exists, and both `append` and `overwrite` arguments are unset, or `append = TRUE` and the data frame with the new data has different column names, an error is raised; the remote table remains unchanged.

An error is raised when calling this method for a closed or invalid connection. An error is also raised if `name` cannot be processed with `dbQuoteIdentifier()` or if this results in a non-scalar. Invalid values for the additional arguments `row.names`, `overwrite`, `append`, `field.types`, and `temporary` (non-scalars, unsupported data types, `NA`, incompatible values, duplicate or missing names, incompatible columns) also raise an error.

Additional arguments

The following arguments are not part of the `dbWriteTable()` generic (to improve compatibility across backends) but are part of the DBI specification:

- `row.names` (default: `FALSE`)

- `overwrite` (default: `FALSE`)
- `append` (default: `FALSE`)
- `field.types` (default: `NULL`)
- `temporary` (default: `FALSE`)

They must be provided as named arguments. See the "Specification" and "Value" sections for details on their usage.

Specification

The `name` argument is processed as follows, to support databases that allow non-syntactic names for their objects:

- If an unquoted table name as string: `dbWriteTable()` will do the quoting, perhaps by calling `dbQuoteIdentifier(conn, x = name)`
- If the result of a call to `dbQuoteIdentifier()`: no more quoting is done

If the `overwrite` argument is `TRUE`, an existing table of the same name will be overwritten. This argument doesn't change behavior if the table does not exist yet.

If the `append` argument is `TRUE`, the rows in an existing table are preserved, and the new data are appended. If the table doesn't exist yet, it is created.

If the `temporary` argument is `TRUE`, the table is not available in a second connection and is gone after reconnecting. Not all backends support this argument. A regular, non-temporary table is visible in a second connection and after reconnecting to the database.

SQL keywords can be used freely in table names, column names, and data. Quotes, commas, and spaces can also be used in the data, and, if the database supports non-syntactic identifiers, also for table names and column names.

The following data types must be supported at least, and be read identically with `dbReadTable()`:

- integer
- numeric (the behavior for `Inf` and `NaN` is not specified)
- logical
- `NA` as NULL
- 64-bit values (using `"bigint"` as field type); the result can be
  - converted to a numeric, which may lose precision,
  - converted a character vector, which gives the full decimal representation
  - written to another table and read again unchanged
- character (in both UTF-8 and native encodings), supporting empty strings
- factor (returned as character)
- list of raw (if supported by the database)
- objects of type blob::blob (if supported by the database)
- date (if supported by the database; returned as `Date`)
- time (if supported by the database; returned as objects that inherit from `difftime`)

- timestamp (if supported by the database; returned as `POSIXct` respecting the time zone but not necessarily preserving the input time zone)

Mixing column types in the same table is supported.

The `field.types` argument must be a named character vector with at most one entry for each column. It indicates the SQL data type to be used for a new column. If a column is missed from `field.types`, the type is inferred from the input data with `dbDataType()`.

The interpretation of rownames depends on the `row.names` argument, see `sqlRownamesToColumn()` for details:

- If `FALSE` or `NULL`, row names are ignored.
- If `TRUE`, row names are converted to a column named "row_names", even if the input data frame only has natural row names from 1 to `nrow(...)`.
- If `NA`, a column named "row_names" is created if the data has custom row names, no extra column is created in the case of natural row names.
- If a string, this specifies the name of the column in the remote table that contains the row names, even if the input data frame only has natural row names.

The default is `row.names = FALSE`.

See Also

Other DBIConnection generics: `DBIConnection-class`, `dbAppendTable()`, `dbCreateTable()`, `dbDataType()`, `dbDisconnect()`, `dbExecute()`, `dbExistsTable()`, `dbGetException()`, `dbGetInfo()`, `dbGetQuery()`, `dbIsReadOnly()`, `dbIsValid()`, `dbListFields()`, `dbListObjects()`, `dbListResults()`, `dbListTables()`, `dbReadTable()`, `dbRemoveTable()`, `dbSendQuery()`, `dbSendStatement()`

Examples

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

dbWriteTable(con, "mtcars", mtcars[1:5, ])
dbReadTable(con, "mtcars")

dbWriteTable(con, "mtcars", mtcars[6:10, ], append = TRUE)
dbReadTable(con, "mtcars")

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

# No row names
dbWriteTable(con, "mtcars", mtcars[1:10, ], overwrite = TRUE, row.names = FALSE)
dbReadTable(con, "mtcars")
```

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

Description

Objects of class Id have a single slot name, which is a named character vector. The dbQuoteIdentifier() method converts Id objects to strings. Support for Id objects depends on the database backend. They can be used in the following methods as name argument:

- dbCreateTable()
- dbAppendTable()
- dbReadTable()
- dbWriteTable()
- dbExistsTable()
- dbRemoveTable() Objects of this class are also returned from dbListObjects().

Usage

```
Id(...)
```

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/

Examples

```
Id(schema = "dbo", table = "Customer")
dbQuoteIdentifier(ANSI(), Id(schema = "nycflights13", table = "flights"))
Id(cluster = "mycluster", catalog = "mycatalog", schema = "myschema", table = "mytable")
```

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 an 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 may 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, ..., names = NULL)
```

Arguments

x	A character vector to label as being escaped SQL.
...	Other arguments passed on to methods. Not otherwise used.
names	Names for the returned object, must have the same length as x.

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.

Examples

```
dbQuoteIdentifier(ANSI(), "SELECT")
dbQuoteString(ANSI(), "SELECT")

# 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(), "SELECT"))
```

---

sqlAppendTable *Compose query to 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(). The default methods are ANSI SQL 99 compliant. These methods are mostly useful for backend implementers.

Usage

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

sqlAppendTableTemplate(
  con,
```

```
  table,
  values,
  row.names = NA,
  prefix = "?",
  ...,
  pattern = ""
)
```

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 use for placeholders.
pattern	Parameter pattern to use for placeholders: • "": no pattern • "1": position • anything else: field name

Details

The row.names argument must be passed explicitly in order to avoid a compatibility warning. The default will be changed in a later release.

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` *Compose query to create a simple table*

Description

Exposes an 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.

Details

The `row.names` argument must be passed explicitly in order to avoid a compatibility warning. The default will be changed in a later release.

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 an 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 parameterized 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

Accepts a query string with placeholders for values, and returns a string with the values embedded. The function is careful to quote all of its inputs with `dbQuoteLiteral()` to protect against SQL injection attacks.

Placeholders can be specified with one of two syntaxes:

- ?: each occurrence of a standalone ? is replaced with a value
- `?name1`, `?name2`, ...: values are given as named arguments or a named list, the names are used to match the values

Mixing ? and `?name` syntaxes is an error. The number and names of values supplied must correspond to the placeholders used in the query.

Usage

```
sqlInterpolate(conn, sql, ..., .dots = list())
```

Arguments

`conn`	A DBIConnection object, as returned by `dbConnect()`.
`sql`	A SQL string 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 _.
`..., .dots`	Values (for `...`) or a list (for `.dots`) to interpolate into a string. Names are required if `sql` uses the `?name` syntax for placeholders. All values will be first escaped with `dbQuoteLiteral()` prior to interpolation to protect against SQL injection attacks. Arguments created by `SQL()` or `dbQuoteIdentifier()` remain unchanged.

Value

The `sql` query with the values from `...` and `.dots` safely embedded.

Backend authors

If you are implementing an 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--;")

# Using paste0() could lead to dangerous SQL with carefully crafted inputs
# (SQL injection)
name <- "H'); DROP TABLE--;"
paste0("SELECT * FROM X WHERE name = '", name, "'")

# Use SQL() or dbQuoteIdentifier() to avoid escaping
sql2 <- "SELECT * FROM ?table WHERE name in ?names"
sqlInterpolate(ANSI(), sql2,
  table = dbQuoteIdentifier(ANSI(), "X"),
  names = SQL("('a', 'b')")
)

# Don't use SQL() to escape identifiers to avoid SQL injection
sqlInterpolate(ANSI(), sql2,
  table = SQL("X; DELETE FROM X; SELECT * FROM X"),
  names = SQL("('a', 'b')")
)

# Use dbGetQuery() or dbExecute() to process these queries:
if (requireNamespace("RSQLite", quietly = TRUE)) {
  con <- dbConnect(RSQLite::SQLite())
  sql <- "SELECT ?value AS value"
  query <- sqlInterpolate(con, sql, value = 3)
  print(dbGetQuery(con, query))
  dbDisconnect(con)
}
```

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

Description

A transaction encapsulates several SQL statements in an atomic unit. It is initiated with `dbBegin()` and either made persistent with `dbCommit()` or undone with `dbRollback()`. In any case, the DBMS guarantees that either all or none of the statements have a permanent effect. This helps ensuring consistency of write operations to multiple tables.

Usage

```
dbBegin(conn, ...)

dbCommit(conn, ...)
```

```
dbRollback(conn, ...)
```

Arguments

conn	A DBIConnection object, as returned by dbConnect().
...	Other parameters passed on to methods.

Details

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

Value

dbBegin(), dbCommit() and dbRollback() return TRUE, invisibly. The implementations are expected to raise an error in case of failure, but this is not tested. In any way, all generics throw an error with a closed or invalid connection. In addition, a call to dbCommit() or dbRollback() without a prior call to dbBegin() raises an error. Nested transactions are not supported by DBI, an attempt to call dbBegin() twice yields an error.

Specification

Actual support for transactions may vary between backends. A transaction is initiated by a call to dbBegin() and committed by a call to dbCommit(). Data written in a transaction must persist after the transaction is committed. For example, a record that is missing when the transaction is started but is created during the transaction must exist both during and after the transaction, and also in a new connection.

A transaction can also be aborted with dbRollback(). All data written in such a transaction must be removed after the transaction is rolled back. For example, a record that is missing when the transaction is started but is created during the transaction must not exist anymore after the rollback.

Disconnection from a connection with an open transaction effectively rolls back the transaction. All data written in such a transaction must be removed after the transaction is rolled back.

The behavior is not specified if other arguments are passed to these functions. In particular, **RSQLite** issues named transactions with support for nesting if the name argument is set.

The transaction isolation level is not specified by DBI.

See Also

Self-contained transactions: dbWithTransaction()

Examples

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

dbWriteTable(con, "cash", data.frame(amount = 100))
dbWriteTable(con, "account", data.frame(amount = 2000))

# All operations are carried out as logical unit:
```

```
dbBegin(con)
withdrawal <- 300
dbExecute(con, "UPDATE cash SET amount = amount + ?", list(withdrawal))
dbExecute(con, "UPDATE account SET amount = amount - ?", list(withdrawal))
dbCommit(con)

dbReadTable(con, "cash")
dbReadTable(con, "account")

# Rolling back after detecting negative value on account:
dbBegin(con)
withdrawal <- 5000
dbExecute(con, "UPDATE cash SET amount = amount + ?", list(withdrawal))
dbExecute(con, "UPDATE account SET amount = amount - ?", list(withdrawal))
if (dbReadTable(con, "account")$amount >= 0) {
  dbCommit(con)
} else {
  dbRollback(con)
}

dbReadTable(con, "cash")
dbReadTable(con, "account")

dbDisconnect(con)
```

Index