Package 'SQLDataFrame'

May 10, 2024

Title Representation of SQL tables in DataFrame metaphor

Version 1.18.0

Description Implements bindings for SQL tables that are compatible with Bioconductor S4 data structures, namely the DataFrame and DelayedArray. This allows SQL-derived data to be easily used inside other Bioconductor objects (e.g., SummarizedExperiments) while keeping everything on disk.

License LGPL (>= 3); File LICENSE

Depends DelayedArray, S4Vectors

Imports stats, utils, methods, BiocGenerics, RSQLite, duckdb, DBI

Suggests knitr, rmarkdown, BiocStyle, testthat

biocViews DataRepresentation, Infrastructure, Software

VignetteBuilder knitr

URL https://github.com/Bioconductor/SQLDataFrame

BugReports https://github.com/Bioconductor/SQLDataFrame/issues

RoxygenNote 7.2.3

Encoding UTF-8

git_url https://git.bioconductor.org/packages/SQLDataFrame

git_branch RELEASE_3_19

git_last_commit 43a68ce

git_last_commit_date 2024-04-30

Repository Bioconductor 3.19

Date/Publication 2024-05-09

Author Qian Liu [aut, cre] (<https://orcid.org/0000-0003-1456-5099>),
Aaron Lun [aut],
Martin Morgan [aut]

Maintainer Qian Liu <Qian.Liu@RoswellPark.org>

Contents

`acquireConn` *Acquire the SQL file connection*

Description

Acquire a (possibly cached) SQL file connection given it's path.

Usage

```
acquireConn(path, dbtype = NULL)

releaseConn(path)
```

Arguments

`path`	String containing a path to a SQL file.
`dbtype`	String containing the SQL database type (case insensitive). Supported types are "SQLite" and "DuckDB".

Details

`acquireConn` will cache the DBIConnection object in the current R session to avoid repeated initialization. This improves efficiency for repeated calls, e.g., when creating a DataFrame with multiple columns from the same SQL table. The cached DBIConnection for any given `path` can be deleted by calling `releaseConn` for the same `path`.

Value

For `acquireConn`, a DBIConnection with backends of SQLite or DuckDB, which are identical to that returned by `DBI::dbConnect(RSQLite::SQLite(), path)` or `DBI::dbConnect(duckdb::duckdb(), path)`.

For `releaseConn`, any existing DBIConnection for the `path` is disconnected and cleared from cache, and `NULL` is invisibly returned. If `path=NULL`, all cached connections are removed.

Author(s)

Qian Liu

Examples

```
############
## SQLite
############

## Mocking up a file
tf <- tempfile()
on.exit(unlink(tf))
con <- DBI::dbConnect(RSQLite::SQLite(), tf)
DBI::dbWriteTable(con, "mtcars", mtcars)
DBI::dbDisconnect(con)

## Acquire or release connection
con <- acquireConn(tf, dbtype = "SQLite")
acquireConn(tf, dbtype = "SQLite") # just re-uses the cache
releaseConn(tf) # clears the cache

############
## DuckDB
############

tf1 <- tempfile()
on.exit(unlist(tf1))
con <- DBI::dbConnect(duckdb::duckdb(), tf1)
DBI::dbWriteTable(con, "mtcars", mtcars)
DBI::dbDisconnect(con)
con <- acquireConn(tf1, dbtype = "DuckDB")
releaseConn(tf1)
```

SQLColumnSeed *Column of a SQL table*

Description

Represent a column of a SQL table as a 1-dimensional DelayedArray. This allows us to use SQL data inside DataFrames without loading them into memory.

Usage

```
SQLColumnSeed(path, dbtype, table, column, length = NULL, type = NULL)

SQLColumnVector(x, ...)
```

Arguments

path	String containing a path to a SQL file.
dbtype	String containing the SQL database type (case insensitive). Supported types are "SQLite" and "DuckDB".

table	String containing the name of the table in SQL file.
column	String containing the name of the column inside the table.
length	Integer containing the number of rows. If NULL, this is determined by inspecting the SQL table. This should only be supplied for efficiency purposes, to avoid a file look-up on construction.
type	String specifying the type of the data. If NULL, this is determined by inspecting the file. Users may specify this to avoid a look-up, or to coerce the output into a different type.
x	A SQLColumnSeed object.
...	Further arguments to be passed to the SQLColumnSeed constructor. #'

Value

For SQLColumnSeed: a SQLColumnSeed. For SQLColumnVector: a SQLColumnVector. #'

Author(s)

Qian Liu

Examples

```
# Mocking up a file:
tf <- tempfile()
on.exit(unlink(tf))
con <- DBI::dbConnect(RSQLite::SQLite(), tf)
DBI::dbWriteTable(con, "mtcars", mtcars)
DBI::dbDisconnect(con)

# Creating a vector:
SQLColumnVector(tf, dbtype = "SQLite", "mtcars", column="gear")

# This happily lives inside DataFrames:
collected <- list()
for (x in colnames(mtcars)) {
    collected[[x]] <- SQLColumnVector(tf, dbtype = "SQLite", "mtcars", column=x)
}
DataFrame(collected)
#'
```

SQLDataFrame *SQL-backed DataFrame*

Description

Create a SQL-backed DataFrame, where the data are kept on disk until requested. Direct extension classes are SQLiteDataFrame and DuckDBDataFrame.

Usage

```
SQLDataFrame(path, dbtype = NULL, table = NULL, columns = NULL, nrows = NULL)
```

Arguments

path	String containing a path to a SQL file.
dbtype	String containing the SQL database type (case insensitive). Supported types are "SQLite" and "DuckDB".
table	String containing the name of SQL table.
columns	Character vector containing the names of columns in a SQL table. If NULL, this is determined from path.
nrows	Integer scalar specifying the number of rows in a SQL table. If NULL, this is determined from path.

Details

The SQLDataFrame is essentially just a DataFrame of SQLColumnVector objects. It is primarily useful for indicating that the in-memory representation is consistent with the underlying SQL file (e.g., no delayed filter/mutate operations have been applied, no data has been added from other files). Thus, users can specialize code paths for a SQLDataFrame to operate directly on the underlying SQL table.

In that vein, operations on a SQLDataFrame may return another SQLDataFrame if the operation does not introduce inconsistencies with the file-backed data. For example, slicing or combining by column will return a SQLDataFrame as the contents of the retained columns are unchanged. In other cases, the SQLDataFrame will collapse to a regular DFrame of SQLColumnVector objects before applying the operation; these are still file-backed but lack the guarantee of file consistency.

Value

A SQLDataFrame where each column is a SQLColumnVector.

Author(s)

Qian Liu

Examples

```
## Mocking up a file:

### SQLite
tf <- tempfile()
on.exit(unlink(tf))
con <- DBI::dbConnect(RSQLite::SQLite(), tf)
DBI::dbWriteTable(con, "mtcars", mtcars)
DBI::dbDisconnect(con)

### DuckDB
tf1 <- tempfile()
```

```
on.exit(unlist(tf1))
con <- DBI::dbConnect(duckdb::duckdb(), tf1)
DBI::dbWriteTable(con, "mtcars", mtcars)
DBI::dbDisconnect(con)

## Creating a SQLite-backed data frame:

df <- SQLDataFrame(tf, dbtype = "SQLite", table = "mtcars")
df1 <- SQLiteDataFrame(tf, "mtcars")
identical(df, df1)

## DuckDB-backed data frame:
df2 <- SQLDataFrame(tf1, dbtype = "duckdb", table = "mtcars")
df3 <- DuckDBDataFrame(tf1, "mtcars")
identical(df2, df3)
## Extraction yields a SQLiteColumnVector:
df$carb

## Some operations preserve the SQLDataFrame:
df[,1:5]
combined <- cbind(df, df)
class(combined)

## ... but most operations collapse to a regular DFrame:
df[1:5,]
combined2 <- cbind(df, some_new_name=df[,1])
class(combined2)

df1 <- df
rownames(df1) <- paste0("row", seq_len(nrow(df1)))
class(df1)

df2 <- df
colnames(df2) <- letters[1:ncol(df2)]
class(df2)

df3 <- df
df3$carb <- mtcars$carb
class(df3)

## Utility functions
path(df)
dbtype(df)
sqltable(df)
dim(df)
names(df)

as.data.frame(df)
```

`SQLiteColumnSeed-class` *SQL extensions*

Description

Extensions of SQLDataFrame, SQLColumnVector, SQLColumnSeed with different SQL backends. Currently supporting SQLite and DuckDB, with which the definition coding can be followed for added extension of other SQL backends.

Arguments

`path`	String containing a path to a SQL file.
`table`	String containing the name of the table in SQL file.
`column`	String containing the name of the column inside the table.
`length`	Integer containing the number of rows. If `NULL`, this is determined by inspecting the SQL table. This should only be supplied for efficiency purposes, to avoid a file look-up on construction.
`type`	String specifying the type of the data. If `NULL`, this is determined by inspecting the file. Users may specify this to avoid a look-up, or to coerce the output into a different type.

Examples

```
## Mocking up a file:

### SQLite
tf <- tempfile()
on.exit(unlink(tf))
con <- DBI::dbConnect(RSQLite::SQLite(), tf)
DBI::dbWriteTable(con, "mtcars", mtcars)
DBI::dbDisconnect(con)

### DuckDB
tf1 <- tempfile()
on.exit(unlist(tf1))
con <- DBI::dbConnect(duckdb::duckdb(), tf1)
DBI::dbWriteTable(con, "mtcars", mtcars)
DBI::dbDisconnect(con)

## Constructor of xxColumnSeed and xxColumnVector

sd <- SQLiteColumnSeed(tf, "mtcars", "gear")
scv <- SQLiteColumnVector(sd)
scv1 <- SQLiteColumnVector(tf, "mtcars", "gear")
identical(scv, scv1)

DuckDBColumnSeed(tf1, "mtcars", "mpg")
DuckDBColumnVector(tf1, "mtcars", "mpg")
```

```
## Constructor of xxDataFrame

SQLiteDataFrame(tf, "mtcars")
DuckDBDataFrame(tf1, "mtcars")
```

Index