Package 'DuckDBDataFrame'

August 24, 2026

Version 0.99.23

Date 2026-08-19

Title DuckDB-Backed DataFrame and Table Structures

Description Provides DuckDB-backed implementations of core tabular data structures including DuckDBTable, DuckDBDataFrame, and DuckDBColumn. These classes extend Bioconductor's S4Vectors framework to enable efficient, out-of-memory operations on large tabular datasets using DuckDB's analytical database engine. The package includes SQL function discovery, connection management, and support for list columns and embeddings stored in ARRAY[] columns.

License MIT + file LICENSE

URL https://github.com/Genentech/DuckDBDataFrame

BugReports https://github.com/Genentech/DuckDBDataFrame/issues

Depends R (>= 4.6.0), methods, stats, bit64, S4Vectors, IRanges

Imports tools, BiocGenerics, S4Arrays, SparseArray, DelayedArray, Matrix, DBI, rlang, dplyr, dbplyr, duckdb, arrow, tibble

Suggests knitr, rmarkdown, BiocStyle, testthat

biocViews DataImport, DataRepresentation, Infrastructure, Software

VignetteBuilder knitr

Encoding UTF-8

Collate 'sql_fun.R' 'sql_call.R' 'tblconn.R' 'keynames.R' 'DuckDBConnection.R' 'DuckDBTable-class.R' 'DuckDBTable-utils.R' 'DuckDBColumn-class.R' 'DuckDBAtomicList-class.R' 'DuckDBColumn-utils.R' 'DuckDBEmbeddings-class.R' 'DuckDBDataFrame-class.R' 'DuckDBDataFrame-internals.R' 'DuckDBDataFrame-package.R' 'DuckDBDataFrameList-class.R' 'DuckDBSelfHits-class.R' 'DuckDBTransposedDataFrame-class.R' 'arrow-types.R' 'parquet-io-cluster.R' 'parquet-io.R'

Config/roxygen2/version 8.1.0

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

git_branch devel

git_last_commit 2f7702e

git_last_commit_date 2026-08-19

Repository Bioconductor 3.24

Date/Publication 2026-08-24

Author Patrick Aboyoun [aut, cre],
Genentech, Inc. [cph]

Maintainer Patrick Aboyoun <aboyounp@gene.com>

Contents

`DuckDBDataFrame-package`

DuckDBDataFrame: DuckDB-Backed DataFrame and Table Structures

---

Description

Provides DuckDB-backed implementations of core tabular data structures including DuckDBTable, DuckDBDataFrame, and DuckDBColumn. These classes extend Bioconductor's S4Vectors framework to enable efficient, out-of-memory operations on large tabular datasets using DuckDB's analytical database engine. The package includes SQL function discovery, connection management, and support for list columns and embeddings stored in ARRAY[] columns.

Author(s)

Maintainer: Patrick Aboyoun <aboyounp@gene.com>

Authors:

- Patrick Aboyoun <aboyounp@gene.com>

Other contributors:

- Genentech, Inc. [copyright holder]

See Also

Useful links:

- https://github.com/Genentech/DuckDBDataFrame
- Report bugs at https://github.com/Genentech/DuckDBDataFrame/issues

---

`arrow-types` *Arrow type helpers for the DuckDB package suite*

Description

Utilities for narrowing integer types and converting between `DataType` objects and Arrow type names. Used by **BiocDuckDB** and **DuckDBArray** for Parquet schema selection and parallel-safe type passing.

Usage

```
arrowIntType(range)

arrowType(x)

arrowTypeToName(type)

arrowTypeFromName(name)
```

Arguments

`range`	Length-two numeric vector `c(min, max)` for `arrowIntType`.
`x`	An R vector for `arrowType`.
`type`	An `DataType` or Arrow type name (character) for `arrowTypeToName`.
`name`	Length-one character Arrow type name (e.g. `"uint16"`) for `arrowTypeFromName`.

Details

`arrowIntType` selects the narrowest unsigned or signed Arrow integer type that can represent `range`. `arrowType` infers an Arrow type from an R vector, narrowing integers when possible.

`arrowTypeFromName` and `arrowTypeToName` implement a small registry of scalar Arrow types used when passing type information to BiocParallel workers (where `DataType` external pointers do not serialize reliably).

Value

`arrowIntType`, `arrowType` An `DataType`.

`arrowTypeToName` A length-one character Arrow type name.

`arrowTypeFromName` An `DataType`.

Author(s)

Patrick Aboyoun

See Also

parquet-io, writeParquet, writeCoordArray

Examples

```
arrowIntType(c(0L, 10L))
arrowType(1:10L)
arrowTypeToName(arrow::uint16())
arrowTypeFromName("uint16")
```

DuckDBAtomicList-class

DuckDBAtomicList objects

Description

The DuckDBAtomicList class extends AtomicList and DuckDBColumn to represent list columns extracted from a DuckDBDataFrame object.

Value

Objects of class `DuckDBAtomicList` extend AtomicList.

Accessors

In the code snippets below, `x` is a DuckDBAtomicList object:

`length(x)`**:** Get the number of list elements in `x`.

`names(x)`**:** Get the names of the list elements of `x`.

`elementNROWS(x)`**:** Get the length of each list element in `x`.

`elementType(x)`**:** Get the type of the list elements; one of `"logical"`, `"integer"`, `"integer64"`, `"numeric"`, `"character"`, or `"factor"`.

`dimtbls(x, drop = TRUE)`**,** `dimtbls(x) <- value`**:** Get or set the list of dimension tables used to define partitions for efficient queries. If `drop = TRUE`, then it returns a named `DataFrameList` object, else it returns an environment containing a `dimtbls` named `DataFrameList` element.

Coercion

`as.list(x)`**:** Coerces `x` to a list.

`realize(x, BACKEND = getAutoRealizationBackend())`**:** Realize an object into memory or on disk using the equivalent of `realize(as.list(x), BACKEND)`.

Subsetting

In the code snippets below, `x` is a DuckDBAtomicList object:

`x[i]`**:** Returns a DuckDBAtomicList object containing the selected list elements.

`x[[i]]`**:** Extracts the list element at position `i` as an atomic vector.

`head(x, n = 6L)`**:** If `n` is non-negative, returns the first `n` list elements of `x`. If `n` is negative, returns all but the last `abs(n)` list elements of `x`.

`tail(x, n = 6L)`**:** If `n` is non-negative, returns the last `n` list elements of `x`. If `n` is negative, returns all but the first `abs(n)` list elements of `x`.

Author(s)

Patrick Aboyoun

See Also

- DuckDBColumn-class for atomic columns
- AtomicList for the base class
- List for the List class hierarchy

Examples

```
tf <- tempfile(fileext = ".parquet")
on.exit(unlink(tf))
df <- data.frame(
    id = sprintf("gene%02d", 1:5),
    int_list = I(lapply(1:5, function(i) seq_len(i)))
)
arrow::write_parquet(df, tf)
ddf <- DuckDBDataFrame(tf, keycol = "id")
lst <- ddf[["int_list"]]
lst
elementNROWS(lst)
```

DuckDBColumn-class *DuckDBColumn objects*

Description

The DuckDBColumn class extends Vector to represent a column extracted from a DuckDBDataFrame object.

Value

Objects of class DuckDBColumn extend Vector.

Accessors

In the code snippets below, x is a DuckDBColumn object:

`length(x)`: Get the number of elements in x.

`names(x)`: Get the names of the elements of x.

`dimtbls(x, drop = TRUE)`, `dimtbls(x) <- value`: Get or set the list of dimension tables used to define partitions for efficient queries. If `drop = TRUE`, then it returns a named `DataFrameList` object, else it returns an environment containing a `dimtbls` named `DataFrameList` element.

`type(x)`, `type(x) <- value`: Get or set the data type of the elements; one of `"logical"`, `"integer"`, `"integer64"`, `"double"`, `"character"`, or `"factor"` (a `"character"` column with a recorded `collevels` entry).

`levels(x)`, `nlevels(x)`: Get the recorded factor levels (or their count), or `NULL` (`0` for `nlevels`) if x is not a `"factor"`-typed column. Does not materialize x. There is no `levels<-` replacement method; relabeling levels would need to remap the underlying stored values, not just the recorded level set.

Coercion

`as.vector(x)`: Coerces `x` to a vector.

`realize(x, BACKEND = getAutoRealizationBackend())`: Realize an object into memory or on disk using the equivalent of `realize(as.vector(x), BACKEND)`.

Subsetting

In the code snippets below, `x` is a DuckDBColumn object:

`x[i]`: Returns a DuckDBColumn object containing the selected elements.

`head(x, n = 6L)`: If `n` is non-negative, returns the first `n` elements of `x`. If `n` is negative, returns all but the last `abs(n)` elements of `x`.

`tail(x, n = 6L)`: If `n` is non-negative, returns the last `n` elements of `x`. If `n` is negative, returns all but the first `abs(n)` elements of `x`.

Author(s)

Patrick Aboyoun

See Also

- `DuckDBColumn-utils` for the utilities
- `Vector` for the base class

Examples

```
tf <- tempfile(fileext = ".parquet")
on.exit(unlink(tf))
arrow::write_parquet(cbind(model = rownames(mtcars), mtcars), tf)
df <- DuckDBDataFrame(tf, datacols = colnames(mtcars), keycol = "model")
cyl <- df[["cyl"]]
cyl
length(cyl)
mean(cyl)
head(cyl, 3)
```

DuckDBColumn-utils *Common operations on DuckDBColumn objects*

Description

Common operations on DuckDBColumn objects.

Value

Method return types are documented in the sections above.

SQL Methods

In the code snippets below, x is a DuckDBColumn object:

`sql_call(x, fun, ...)`: Applies the specified SQL function to all data columns of x.

`sql_fun(x, function_type = NULL, return_type = NULL, description = FALSE)`: Finds DuckDB SQL functions compatible with the data type of x. Returns a data.frame of matching functions with optional filtering by function type and return type.

Group Generics

DuckDBColumn objects have support for S4 group generic functionality:

`Arith` "+", "-", "*", "^", "%%", "%/%", "/"

`Compare` "==", ">", "<", "!=", "<=", ">="

`Logic` "&", "|", "!"

`Ops` "Arith", "Compare", "Logic"

`Math` "abs", "sign", "sqrt", "ceiling", "floor", "trunc", "log", "log10", "log2", "acos", "acosh", "asin", "asinh", "atan", "atanh", "exp", "expm1", "cos", "cosh", "sin", "sinh", "tan", "tanh", "gamma", "lgamma"

`Summary` "max", "min", "range", "prod", "sum", "any", "all"

See S4groupGeneric for more details.

Numerical Data Methods

In the code snippets below, x is a DuckDBColumn object:

`is.finite(x)`: Returns a DuckDBColumn containing logicals that indicate which values are finite.

`is.infinite(x)`: Returns a DuckDBColumn containing logicals that indicate which values are infinite.

`is.nan(x)`: Returns a DuckDBColumn containing logicals that indicate which values are Not a Number.

`mean(x)`: Calculates the mean of x.

`var(x)`: Calculates the variance of x.

`sd(x)`: Calculates the standard deviation of x.

`median(x)`: Calculates the median of x.

`quantile(x, probs = seq(0, 1, 0.25), names = TRUE, type = 7)`: Calculates the specified quantiles of x.

- `probs` A numeric vector of probabilities with values in [0,1].
- `names` If TRUE, the result has names describing the quantiles.
- `type` Either 1 or 7 that specifies the quantile algorithm detailed in `quantile`.

`mad(x, constant = 1.4826)`: Calculates the median absolute deviation of x.

- `constant` The scale factor.

`IQR(x, type = 7)`: Calculates the interquartile range of x.

- `type` Either 1 or 7 that specifies the quantile algorithm detailed in `quantile`.

`pmax(..., na.rm = FALSE)`: Returns the parallel maxima of multiple DuckDBColumn objects. All arguments must be DuckDBColumn objects with compatible dimensions.

`pmin(..., na.rm = FALSE)`: Returns the parallel minima of multiple DuckDBColumn objects. All arguments must be DuckDBColumn objects with compatible dimensions.

Character Methods

In the code snippets below, x is a DuckDBColumn object:

`nchar(x)`: Returns a DuckDBColumn containing the number of characters in each string.

`tolower(x)`: Returns a DuckDBColumn with all strings converted to lowercase.

`toupper(x)`: Returns a DuckDBColumn with all strings converted to uppercase.

`chartr(old, new, x)`: Returns a DuckDBColumn with characters translated.

`old` Characters to be translated.

`new` Characters to translate to.

`substr(x, start, stop)`: Returns a DuckDBColumn containing substrings extracted by position.

`start` Integer starting position (1-indexed).

`stop` Integer ending position (inclusive).

`substring(x, first, last = 1000000L)`: Returns a DuckDBColumn containing substrings extracted by position.

`first` Integer starting position (1-indexed).

`last` Integer ending position (inclusive).

`grepl(pattern, x, ignore.case = FALSE, fixed = FALSE)`: Returns a DuckDBColumn containing logicals indicating pattern matches.

`pattern` Character string containing a regular expression.

`ignore.case` If `TRUE`, case-insensitive matching.

`fixed` If `TRUE`, pattern is a fixed string not regex.

`sub(pattern, replacement, x, ignore.case = FALSE, fixed = FALSE)`: Returns a DuckDBColumn with first match of pattern replaced.

`pattern` Character string containing a regular expression.

`replacement` Replacement string.

`ignore.case` If `TRUE`, case-insensitive matching.

`fixed` If `TRUE`, pattern is a fixed string not regex.

`gsub(pattern, replacement, x, ignore.case = FALSE, fixed = FALSE)`: Returns a DuckDBColumn with all matches of pattern replaced.

`pattern` Character string containing a regular expression.

`replacement` Replacement string.

`ignore.case` If `TRUE`, case-insensitive matching.

`fixed` If `TRUE`, pattern is a fixed string not regex.

`startsWith(x, prefix)`: Returns a DuckDBColumn containing logicals indicating if strings start with the specified prefix.

`endsWith(x, suffix)`: Returns a DuckDBColumn containing logicals indicating if strings end with the specified suffix.

`paste(..., sep = " ", collapse = NULL)`: Concatenates multiple DuckDBColumn objects using the specified separator. The `collapse` argument is not supported. All arguments must be DuckDBColumn objects with compatible dimensions.

General Methods

In the code snippets below, x is a DuckDBColumn object:

`unique(x)`: Returns a DuckDBColumn containing the distinct rows.

`x %in% table`: Returns a DuckDBColumn containing logicals that indicate the elements of `x` in `table`.

`table(...)`: Returns a table containing the counts across the distinct values.

Sparsity Methods

In the code snippets below, x is a DuckDBColumn object:

`is_nonzero(x)`: Returns a DuckDBColumn containing logicals that indicate if the elements of x are non-zero.

`nzcount(x)`: Returns the total number of non-zero values.

Author(s)

Patrick Aboyoun

See Also

- DuckDBColumn-class for the main class
- Vector for the base class

Examples

```
tf <- tempfile(fileext = ".parquet")
on.exit(unlink(tf))
arrow::write_parquet(cbind(model = rownames(mtcars), mtcars), tf)
df <- DuckDBDataFrame(tf, datacols = colnames(mtcars), keycol = "model")
mpg <- df[["mpg"]]
mean(mpg)
sd(mpg)
unique(mpg)[1:5]
```

DuckDBConnection *Acquire and Release the DuckDB Connection*

Description

Acquire and Release the DuckDB connection used by the **BiocDuckDB** package.

Usage

```
loadExtension(conn, extension, optional = FALSE)

configureOutOfCore(conn)

configureCloud(conn)

acquireDuckDBConn(
  conn = dbConnect(duckdb(), bigint = "integer64", array = "matrix")
)

releaseDuckDBConn()
```

Arguments

conn	A duckdb_connection object.
extension	A character string naming a DuckDB extension (e.g. "spatial").
optional	If TRUE, an extension that is unknown to the build or cannot be installed (permission / network) produces a warning and is skipped rather than an error. Defaults to FALSE.

Details

acquireDuckDBConn will add a duckdb_connection object to the **BiocDuckDB** package cache. releaseDuckDBConn will remove the duckdb_connection object from that cache. loadExtension installs and loads a DuckDB extension on the shared connection; it is used by companion packages (e.g. **DuckDBSpatial**) to ensure their required extension ("spatial") is available.

configureOutOfCore is called automatically by acquireDuckDBConn when the shared connection is first created; call it again (on acquireDuckDBConn()) after changing an option to re-apply. Each setting is read from an R option first, then an environment variable, and left at the DuckDB default when neither is set:

Setting	Option	Environment variable
memory limit	DuckDBDataFrame.memory_limit	BIOCDUCKDB_MEMORY_LIMIT
threads	DuckDBDataFrame.threads	BIOCDUCKDB_THREADS
spill directory	DuckDBDataFrame.temp_directory	BIOCDUCKDB_TEMP_DIRECTORY
preserve insertion order	\ DuckDBDataFrame.preserve_insertion_order	BIOCDUCKDB_PRESERVE_INSERTION_ORD

Setting memory_limit (e.g. "16GB" or "80%") and temp_directory guards against the OS killing the process before DuckDB can spill a large aggregation/sort/join; preserve_insertion_order = FALSE avoids buffering a whole result to preserve row order on a Tahoe-scale export where order is not significant.

The spill temp_directory is always set and created (recursively): when neither the option nor the environment variable is given it defaults to a temp subdirectory of R_user_dir("DuckDBDataFrame", "cache") rather than DuckDB's default under the R session tempdir, which on batch schedulers (e.g. SLURM's per-job /tmp) can be small or cleaned mid-session and make a spill fail to create its directory. Point BIOCDUCKDB_TEMP_DIRECTORY at roomy scratch for large out-of-core sorts.

The memory_limit likewise gets a package default when unset: 80% of the most-restrictive detected ceiling – an explicit SLURM allocation (SLURM_MEM_PER_NODE, or SLURM_MEM_PER_CPU times SLURM_CPUS_ON_NODE), the cgroup limit (v2 memory.max then v1 memory.limit_in_bytes, read from the job's own cgroup resolved via /proc/self/cgroup before the mount root), then physical RAM. DuckDB's own default is 80% of *physical* RAM, which ignores a SLURM / cgroup cap and can over-commit (near-OOM on the first large scan); the detected default keeps a big aggregation spilling within the job's allocation. When nothing can be detected (e.g. macOS) DuckDB's default is left in place.

threads is likewise defaulted when unset: the most-restrictive detected CPU allocation – a SLURM allocation (SLURM_CPUS_PER_TASK, or SLURM_CPUS_ON_NODE) then the cgroup CFS quota (v2 cpu.max, v1 cpu.cfs_quota_us/cpu.cfs_period_us, read from the job's own cgroup subpath before the mount root). DuckDB otherwise defaults to hardware concurrency, which ignores a SLURM/cgroup cpuset and over-subscribes on a shared node (each extra thread also carries working memory against the memory cap). When no allocation is detected, DuckDB's default is left in place.

Value

`acquireDuckDBConn` returns a cached `duckdb_connection` object. `releaseDuckDBConn` returns `NULL`, invisibly. `loadExtension` installs (if needed) and loads `extension` on `conn`, returning the load status invisibly. `configureOutOfCore` applies out-of-core engine settings (buffer-pool memory limit, worker threads, spill directory, insertion-order preservation) to `conn` from R options or environment variables, returning `NULL` invisibly.

Author(s)

Patrick Aboyoun

Examples

```
releaseDuckDBConn()
conn <- acquireDuckDBConn()
identical(conn, acquireDuckDBConn())
releaseDuckDBConn()
releaseDuckDBConn()
```

DuckDBDataFrame-class *DuckDBDataFrame objects*

Description

The DuckDBDataFrame class extends both DuckDBTable and DataFrame to represent a DuckDB table as a DataFrame object.

Details

DuckDBDataFrame adds DataFrame semantics to a DuckDB table. It achieves a balance between the flexibility of a tbl_duckdb_connection object and the familiarity of a DataFrame object.

Value

Objects of class `DuckDBDataFrame` extend DataFrame.

Constructor

`DuckDBDataFrame(conn, datacols = colnames(conn), keycol = NULL, dimtbl = NULL, type = NULL, collevels =`

Creates a DuckDBDataFrame object.

`conn` Either a character vector containing the paths to parquet, csv, or gzipped csv data files; a string that defines a duckdb `read_*` data source; a DuckDBDataFrame object; or a tbl_duckdb_connection object.

`datacols` Either a character vector of column names from `conn` or a named `expression` that will be evaluated in the context of `conn` that defines the data.

`keycol` An optional string specifying the column name from `conn` that will define the foreign key in the underlying table, or a named list containing a character vector where the name of the list element defines the foreign key and the character vector set the distinct values for that key. If missing, a `row_number` column is created as an identifier.

`dimtbl` A optional named `DataFrameList` that specifies the dimension table associated with the `keycol`. The name of the list element must match the name of the `keycol` list. Additionally, the `DataFrame` object must have row names that match the distinct values of the `keycol` list element and columns that define partitions in the data table for efficient querying.

`type` An optional named character vector where the names specify the column names and the values specify the column type; one of `"logical"`, `"integer"`, `"integer64"`, `"double"`, or `"character"`.

`collevels` An optional named list, keyed by data column name, that restores `factor` columns on materialization (each element a list with a character `levels` vector and a `TRUE`/`FALSE` `ordered` flag). Primarily used by `readParquet()`.

Accessors

In the code snippets below, `x` is a DuckDBDataFrame object:

`dim(x)`**:** Length two integer vector defined as `c(nrow(x), ncol(x))`.

`nrow(x)`**,** `ncol(x)`**:** Get the number of rows and columns, respectively.

`NROW(x)`**,** `NCOL(x)`**:** Same as `nrow(x)` and `ncol(x)`, respectively.

`dimnames(x)`**:** Length two list of character vectors defined as `list(rownames(x), colnames(x))`.

`rownames(x)`**,** `colnames(x)`**:** Get the names of the rows and columns, respectively.

`coltypes(x)`**,** `coltypes(x) <- value`**:** Get or set the data type of the columns; one of `"logical"`, `"integer"`, `"integer64"`, `"double"`, `"character"`, or `"factor"` (a `"character"` column with a recorded `collevels` entry).

`dimtbls(x, drop = TRUE)`**,** `dimtbls(x) <- value`**:** Get or set the list of dimension tables used to define partitions for efficient queries. If `drop = TRUE`, then it returns a named `DataFrameList` object, else it returns an environment containing a `dimtbls` named `DataFrameList` element.

Coercion

`as.data.frame(x)`**:** Coerces `x` to a data.frame.

`as.matrix(x)`**:** Coerces `x` to a matrix.

`as(from, "DFrame")`**:** Converts a DuckDBDataFrame object to a DFrame object. This process involves first loading the data into memory using `as.data.frame`. The resulting data.frame is then coerced into a DFrame. Additionally, any associated metadata and mcols (metadata columns) are preserved and added to the DFrame, if they exist.

`realize(x, BACKEND = getAutoRealizationBackend())`**:** Realize an object into memory or on disk using the equivalent of `realize(as(x, "DFrame"), BACKEND)`.

Subsetting

In the code snippets below, `x` is a DuckDBDataFrame object:

`x[i, j, drop = TRUE]`**:** Returns either a new DuckDBDataFrame object or a DuckDBColumn if selecting a single column and `drop = TRUE`.

`x[[i]]`**:** Extracts a DuckDBColumn object from `x`.

`x[[i]] <- value`**:** Replaces column `i` in `x` with `value`.

`head(x, n = 6L)`**:** If `n` is non-negative, returns the first `n` rows of `x`. If `n` is negative, returns all but the last `abs(n)` rows of `x`.

`tail(x, n = 6L)`: If `n` is non-negative, returns the last `n` rows of `x`. If `n` is negative, returns all but the first `abs(n)` rows of `x`.

`subset(x, subset, select, drop = FALSE)`: Return a new DuckDBDataFrame using:

subset logical expression indicating rows to keep, where missing values are taken as FALSE.

select expression indicating columns to keep.

drop passed on to `[` indexing operator.

Combining

`cbind(...)`: Creates a new DuckDBDataFrame object by concatenating the columns of the input objects. The returned DuckDBDataFrame object concatenates the metadata across the input objects. The metadata columns of the returned DuckDBDataFrame object are obtained by combining the metadata columns of the input object with `combineRows()`.

Displaying

The `show()` method for DuckDBDataFrame objects obeys global options `showHeadLines` and `showTailLines` for controlling the number of head and tail rows and columns to display.

Author(s)

Patrick Aboyoun, Aaron Lun

See Also

- `DuckDBTable-class` for the table base class
- `DuckDBTable-utils` for the table base class utilities
- `DuckDBColumn-class` for the extracted column class
- `DuckDBColumn-utils` for the extracted column utilities
- `DuckDBTransposedDataFrame-class` for the transposed DataFrame class
- `DuckDBDataFrameList-class` for the split DataFrame class
- `DataFrame` for the base class

Examples

```
# Mocking up a file:
tf <- tempfile(fileext = ".parquet")
on.exit(unlink(tf))
arrow::write_parquet(cbind(model = rownames(mtcars), mtcars), tf)

# Creating our DuckDB-backed data frame:
df <- DuckDBDataFrame(tf, datacols = colnames(mtcars), keycol = "model")
df

# Extraction yields a DuckDBColumn:
df$carb

# Slicing DuckDBDataFrame objects:
df[,1:5]
df[1:5,]

# Combining by DuckDBDataFrame and DuckDBColumn objects:
```

```
combined <- cbind(df, df)
class(combined)
combined2 <- cbind(df, some_new_name=df[,1])
class(combined2)
```

DuckDBDataFrameList-class *DuckDBDataFrameList objects*

Description

The DuckDBDataFrame class extends SplitDataFrameList.

Value

Objects of class `DuckDBDataFrameList` extend List.

Constructor

`split(x, f)`: Creates a DuckDBDataFrameList object.

- `x` A DuckDBDataFrame object to split.
- `f` A DuckDBColumn object to split `x` by.

Accessors

In the code snippets below, `x` is a DuckDBDataFrameList object:

`length(x)`: Get the number of elements in `x`.

`names(x)`, `names(x) <- value`: Get or set the names of the elements of `x`.

`mcols(x)`, `mcols(x) <- value`: Get or set the metadata columns.

`elementNROWS(x)`: Get the length (or nb of row for a matrix-like object) of each of the elements.

`dimtbls(x, drop = TRUE)`, `dimtbls(x) <- value`: Get or set the list of dimension tables used to define partitions for efficient queries. If `drop = TRUE`, then it returns a named `DataFrameList` object, else it returns an environment containing a `dimtbls` named `DataFrameList` element.

`dims(x)`: Get the two-column matrix indicating the number of rows and columns for each of the list elements.

`nrows(x)`, `ncols(x)`: Get the number of rows and columns, respectively, for each of the list elements.

`dimnames(x)`: Get the list of two CharacterLists, the first holding the rownames and the second the column names.

`rownames(x)`, `colnames(x)`: Get the CharacterList of row and colum names, respectively.

`commonColnames(x)`, `commonColnames(x) <- value`: Get or set the character vector of column names present in the individual DataFrames in `x`.

`columnMetadata(x)`, `columnMetadata(x) <- value`: Get the DataFrame of metadata along the columns, i.e., where each column in `x` is represented by a row in the metadata. The metadata is common across all elements of `x`. Note that calling `mcols(x)` returns the metadata on the DataFrame elements of `x`.

Coercion

In the code snippets below, x is a DuckDBDataFrameList object:

`unlist(x)`: Returns the underlying DuckDBDataFrame object.

`as(from, "DFrameList")`: Converts a DuckDBDataFrameList object to a DFrameList object. This process involves first loading the data into memory using `as(from, "DFrame")`. The resulting DFrame is then split into a DFrameList. Additionally, any associated metadata and mcols (metadata columns) are preserved and added to the DFrameList, if they exist.

`realize(x, BACKEND = getAutoRealizationBackend())`: Realize an object into memory or on disk using the equivalent of `realize(as(x, "DFrameList"), BACKEND)`.

Subsetting

In the code snippets below, x is a DuckDBDataFrameList object:

`x[i]`: Returns a DuckDBDataFrameList object containing the selected elements.

`x[[i]]`: Return the selected DuckDBDataFrame by `i`, where `i` is an numeric or character vector of length 1.

`x$name`: Similar to `x[[name]]`, but `name` is taken literally as an element name.

`head(x, n = 6L)`: If `n` is non-negative, returns the first `n` elements of `x`. If `n` is negative, returns all but the last `abs(n)` elements of `x`.

`tail(x, n = 6L)`: If `n` is non-negative, returns the last `n` elements of `x`. If `n` is negative, returns all but the first `abs(n)` elements of `x`.

Author(s)

Patrick Aboyoun

See Also

- DuckDBDataFrame-class for the unsplit class
- SplitDataFrameList for the base class

Examples

```
# Mocking up a file:
tf <- tempfile(fileext = ".parquet")
on.exit(unlink(tf))
arrow::write_parquet(cbind(model = rownames(mtcars), mtcars), tf)

# Creating our DuckDB-backed data frame:
df <- DuckDBDataFrame(tf, datacols = colnames(mtcars), keycol = "model")
dflist <- split(df, df$cyl)
dflist
```

DuckDBEmbeddings-class *DuckDBEmbeddings objects*

Description

The DuckDBEmbeddings class extends RectangularData and DuckDBColumn to represent a single dimensionality reduction embedding (e.g., PCA, UMAP, tSNE) stored in DuckDB using an ARRAY[] column.

Details

DuckDBEmbeddings provides a DuckDB-backed container for storing a single embedding as a 2D rectangular object where rows represent cells and columns represent dimensions within that embedding. The embedding is stored as a single ARRAY[] column in the underlying DuckDB table, enabling efficient storage and lazy evaluation.

Value

Objects of class `DuckDBEmbeddings` extend RectangularData.

Accessors

In the code snippets below, `x` is a DuckDBEmbeddings object:

`dim(x)`: Returns integer vector of length 2: `c(nrow(x), ncol(x))` where nrow is the number of cells and ncol is the ARRAY size (number of dimensions in the embedding).

`nrow(x)`, `ncol(x)`: Get the number of rows (cells) and columns (dimensions), respectively.

`NROW(x)`, `NCOL(x)`: Same as `nrow(x)` and `ncol(x)`, respectively.

`dimnames(x)`: Returns list of length 2: `list(rownames(x), colnames(x))`.

`rownames(x)`, `colnames(x)`: Get the names of the rows (cell IDs) and columns (dimension names), respectively.

`length(x)`: Get the number of cells (same as `nrow(x)`).

`names(x)`: Get the cell IDs (same as `rownames(x)`).

`type(x)`: Get the ARRAY data type. Returns enhanced format `"array<double,50>"` or raw DuckDB format `"DOUBLE[50]"` depending on schema detection.

`coltypes(x)`: Get the R element type (e.g., `"numeric"` for DOUBLE arrays, `"integer"` for INTEGER arrays). Handles both type formats.

`dimtbls(x, drop = TRUE)`, `dimtbls(x) <- value`: Get or set the list of dimension tables used to define partitions for efficient queries. If `drop = TRUE`, then it returns a named `DataFrameList` object, else it returns an environment containing a `dimtbls` named `DataFrameList` element.

Subsetting

In the code snippets below, `x` is a DuckDBEmbeddings object:

`x[i, j, drop=TRUE]`: Subset cells (rows) and/or dimensions (columns). When both `i` and `j` are missing, returns `x`. When `drop=TRUE` and the result is a single cell or single dimension, returns a vector. Otherwise returns a matrix or DuckDBEmbeddings object.

`x[i]`: Subset cells (rows), returns a DuckDBEmbeddings object with the selected cells.

`head(x, n = 6L)`: Returns the first n cells of the embedding.

`tail(x, n = 6L)`: Returns the last n cells of the embedding.

Coercion

`as.matrix(x)`: Coerces the embedding to a matrix with cells as rows and dimensions as columns.

`realize(x, BACKEND = getAutoRealizationBackend())`: Realize an object into memory or on disk using the equivalent of `realize(as.matrix(x), BACKEND)`.

Author(s)

Patrick Aboyoun

See Also

- DuckDBNumericList-class for the base class
- DuckDBTable-class for the underlying table representation

Examples

```
tf <- tempfile(fileext = ".parquet")
on.exit(unlink(tf))
n <- 10L
mat <- matrix(rnorm(n * 5), nrow = n)
df <- data.frame(
    cell_id = sprintf("cell_%02d", seq_len(n)),
    pca = I(asplit(mat, 1L)),
    stringsAsFactors = FALSE
)
arrow::write_parquet(df, tf)
emb <- DuckDBDataFrame(tf, keycol = "cell_id")[["pca"]]
emb
dim(emb)
head(emb, 3)
```

DuckDBSelfHits-class *DuckDBSelfHits objects*

Description

The DuckDBSelfHits class extends the SelfHits-class class from S4Vectors for DuckDB tables.

Details

DuckDBSelfHits provides lazy evaluation for large graph structures (e.g., = KNN graphs, similarity matrices) by storing edge lists in DuckDB-backed tables. This enables efficient storage and querying of pairwise relationships without materializing all edges in memory.

The class stores the parallel slots (‘from‘, ‘to‘, and any metadata columns) in a DuckDBDataFrame, while keeping the scalar ‘nLnode‘ and ‘nRnode‘ slots materialized for fast validation.

Value

Objects of class `DuckDBSelfHits` extend SelfHits-class.

Constructor

`DuckDBSelfHits(conn, from, to, nnode, mcols = NULL, keycol = NULL, dimtbl = NULL, nodes = NULL)`: Creates a DuckDBSelfHits object.

- `conn` Either a character vector containing the paths to parquet, csv, or gzipped csv data files; a string that defines a duckdb `read_*` data source; a DuckDBDataFrame object; or a tbl_duckdb_connection object.
- `from` Either `NULL` or a string specifying the column from `conn` that defines the query/left nodes (1-indexed).
- `to` Either `NULL` or a string specifying the column from `conn` that defines the subject/right nodes (1-indexed).
- `nnode` Single integer specifying the number of nodes in the graph. Must be positive. This is the only slot that remains materialized.
- `mcols` Optional character vector specifying additional columns that define edge metadata (e.g., "weight", "distance", "x").
- `keycol` An optional string specifying the column name from `conn` that will define the foreign key in the underlying table, or a named list containing a character vector where the name of the list element defines the foreign key and the character vector sets the distinct values for that key. If missing, a `row_number` column is created as an identifier for each edge.
- `dimtbl` An optional named `DataFrameList` that specifies the dimension table associated with the `keycol`. The name of the list element must match the name of the `keycol` list. Additionally, the `DataFrame` object must have row names that match the distinct values of the `keycol` list element.
- `nodes` An optional integer vector specifying the node ids. If `NULL` (default), uses implicit encoding `c(NA_integer_, -nnode)` representing 1:nnode. Can be an explicit integer vector for non-contiguous node subsets, optionally named for node aliases.

Accessors

In the code snippets below, `x` is a DuckDBSelfHits object:

`length(x)`: Get the number of edges (hits).

`from(x)`, `queryHits(x)`: Get the query/left node indices as a DuckDBColumn.

`to(x)`, `subjectHits(x)`: Get the subject/right node indices as a DuckDBColumn.

`nnode(x)`, `nLnode(x)`, `nRnode(x)`: Get the number of nodes. All three return the same value for SelfHits.

`queryLength(x)`, `subjectLength(x)`: Get the number of left/right nodes. For SelfHits, both return `nnode(x)`.

`countLnodeHits(x)`, `countQueryHits(x)`: Count the number of hits for each left/query node. Returns an integer vector of length `nLnode(x)`. When nodes are implicit (1:nnode), returns an unnamed vector with positional indexing. When nodes are explicit, returns a named vector where names are the node IDs as characters. This operation materializes the `from` column.

`countRnodeHits(x)`, `countSubjectHits(x)`: Count the number of hits for each right/subject node. Returns an integer vector of length `nRnode(x)`. When nodes are implicit (1:nnode), returns an unnamed vector with positional indexing. When nodes are explicit, returns a named vector where names are the node IDs as characters. This operation materializes the `to` column.

`mcols(x)`, `mcols(x) <- value`: Get or set the edge metadata columns.

Coercion

`as(from, "DuckDBDataFrame")`: Creates a DuckDBDataFrame object with from, to, and mcols.

`as.data.frame(x)`: Coerces `x` to a data.frame.

`as(from, "SelfHits")`: Converts a DuckDBSelfHits object to a materialized SelfHits object. This will load all edges into memory.

`as(from, "dgCMatrix")`: Converts to a sparse matrix. Equivalent to `as(as(from, "SelfHits"), "dgCMatrix")`.

`realize(x, BACKEND = getAutoRealizationBackend())`: Realize an object into memory or on disk using the equivalent of `realize(as(x, "SelfHits"), BACKEND)`.

Subsetting

In the code snippets below, `x` is a DuckDBSelfHits object:

`x[i]` **or** `extractROWS(x, i)`: Returns a DuckDBSelfHits object with edge-based subsetting (subsets edges directly by index, does NOT filter by nodes).

`extractNODES(x, i)`: Returns a DuckDBSelfHits object with node-based subsetting. Updates the `nodes` slot to track the subset, and filters edges lazily via SQL to include only edges where BOTH from and to are in the node subset. Filtering is applied when data is materialized via `tblconn()`, `as.data.frame()`, or coercion methods.

`head(x, n = 6L)`: If `n` is non-negative, returns the first `n` edges. If `n` is negative, returns all but the last `abs(n)` edges.

`tail(x, n = 6L)`: If `n` is non-negative, returns the last `n` edges. If `n` is negative, returns all but the first `abs(n)` edges.

Operations

`sort(x, decreasing = FALSE)`: Returns a DuckDBSelfHits with edges sorted by from node, then by to node. The sorting is performed lazily using SQL ORDER BY.

`t(x)`: Transpose the graph by swapping from and to columns.

`c(x, ...)`: Concatenate multiple DuckDBSelfHits objects.

Displaying

The `show()` method for DuckDBSelfHits objects obeys global options `showHeadLines` and `showTailLines` for controlling the number of edges to display.

Author(s)

Patrick Aboyoun

See Also

- `SelfHits-class` for the base class
- `colPairs` for usage in SingleCellExperiment
- `DuckDBDataFrame` for the underlying storage

Examples

```
# Create an example edge list:
edges <- data.frame(
  from = c(1, 1, 2, 3, 4, 4, 5),
  to = c(2, 5, 3, 4, 2, 5, 1),
  weight = c(0.8, 0.6, 0.9, 0.7, 0.85, 0.75, 0.65),
  distance = c(1.2, 2.3, 1.5, 1.8, 1.4, 2.1, 2.5)
)
tf <- tempfile(fileext = ".parquet")
on.exit(unlink(tf))
arrow::write_parquet(edges, tf)

# Create the DuckDBSelfHits object
hits <- DuckDBSelfHits(tf, from = "from", to = "to", nnode = 5,
                       mcols = c("weight", "distance"))
hits

# Access edge data
from(hits)
to(hits)
mcols(hits)

# Convert to sparse matrix (materializes)
mat <- as(hits, "dgCMatrix")
```

DuckDBTable-class *DuckDBTable objects*

Description

The DuckDBTable class extends the RectangularData virtual class for DuckDB tables by wrapping a tbl_duckdb_connection object.

Details

The DuckDBTable class provides a way to define a DuckDB table as a RectangularData object. It supports *standard 2D API* such as `dim()`, `nrow()`, `ncol()`, `dimnames()`, `x[i, j]` and `cbind()`, but does not support `rbind()`.

Value

Objects of class `DuckDBTable` extend RectangularData.

Constructor

`DuckDBTable(conn, datacols = colnames(conn), keycols = NULL, dimtbls = NULL, type = NULL, collevels = NU`

Creates a DuckDBTable object.

`conn` Either a character vector containing the paths to parquet, csv, or gzipped csv data files; a string that defines a duckdb `read_*` data source; a DuckDBDataFrame object; or a tbl_duckdb_connection object.

`datacols` Either a character vector of column names from `conn` or a named `expression` that will be evaluated in the context of `conn` that defines the data.

keycols An optional character vector of column names from conn that will define the set of foreign keys in the underlying table, or a named list of character vectors where the names of the list define the foreign keys and the character vectors set the distinct values for those keys. If missing, a row_number column is created as an identifier.

dimtbls Either NULL, a named DataFrameList object, or an environment containing a named DataFrameList "dimtbls" element that specifies the dimension tables associated with the keycols. The name of the list elements match the names of the keycols list. Additionally, the DataFrame objects have row names that match the distinct values of the corresponding keycols list element and columns that define partitions in the data table for efficient querying.

type An optional named character vector where the names specify the column names and the values specify the column type; one of "logical", "integer", "integer64", "double", or "character".

collevels An optional named list, keyed by data column name, that restores factor columns on materialization. Each element is a list with a character levels vector and a TRUE/FALSE ordered flag; columns not named are left unchanged. Primarily used by readParquet() to recover factors recorded in a product's schema.

Accessors

In the code snippets below, x is a DuckDBTable object:

dim(x): Length two integer vector defined as c(nrow(x), ncol(x)).

nrow(x), ncol(x): Get the number of rows and columns, respectively.

NROW(x), NCOL(x): Same as nrow(x) and ncol(x), respectively.

dimnames(x): Length two list of character vectors defined as list(rownames(x), colnames(x)).

rownames(x), colnames(x): Get the names of the rows and columns, respectively.

coltypes(x), coltypes(x) <- value: Get or set the data type of the columns. Getter returns one of "logical", "integer", "integer64", "double", "character", "factor" (a "character" column with a recorded collevels entry; see DuckDBTable), or complex type strings like "list<integer>", "array<numeric,128>", "struct", "map", "blob", or "geometry". Setter accepts atomic types only (not "factor"; recasting a factor column's underlying type drops its collevels entry, per coltypes<-'s documented behavior).

dimtbls(x, drop = TRUE), dimtbls(x) <- value: Get or set the list of dimension tables used to define partitions for efficient queries. If drop = TRUE, then it returns a named DataFrameList object, else it returns an environment containing a dimtbls named DataFrameList element.

Coercion

as.data.frame(x): Coerces x to a data.frame.

Subsetting

In the code snippets below, x is a DuckDBTable object:

x[i, j]: Return a new DuckDBTable of the same class as x made of the selected rows and columns.

head(x, n = 6L): If n is non-negative, returns the first n rows of x. If n is negative, returns all but the last abs(n) rows of x.

tail(x, n = 6L): If n is non-negative, returns the last n rows of x. If n is negative, returns all but the first abs(n) rows of x.

Author(s)

Patrick Aboyoun

See Also

- DuckDBTable-utils for the utilities
- RectangularData for the base class

Examples

```
# Create a data.frame from the Titanic data
df <- do.call(expand.grid, c(dimnames(Titanic), stringsAsFactors = FALSE))
df$fate <- as.integer(Titanic[as.matrix(df)])

# Write data to a parquet file
tf <- tempfile(fileext = ".parquet")
on.exit(unlink(tf))
arrow::write_parquet(df, tf)

tbl <- DuckDBTable(tf, datacols = "fate", keycols = c("Class", "Sex", "Age", "Survived"))
```

DuckDBTable-utils *Common operations on DuckDBTable objects*

Description

Common operations on DuckDBTable objects.

Value

Method return types are documented in the sections above.

SQL Methods

In the code snippets below, x is a DuckDBTable object:

`sql_call(x, fun, ...)`: Applies the specified SQL function to all data columns of x.

`sql_fun(x, function_type = NULL, return_type = NULL, description = FALSE)`: Finds DuckDB SQL functions compatible with the data type of x. Returns a data.frame of matching functions with optional filtering by function type and return type.

Group Generics

DuckDBTable objects have support for S4 group generic functionality:

`Arith "+", "-", "*", "^", "%%", "%/%", "/"`

`Compare "==", ">", "<", "!=", "<=", ">="`

`Logic "&", "|", "!"`

`Ops "Arith", "Compare", "Logic"`

`Math` "abs", "sign", "sqrt", "ceiling", "floor", "trunc", "log", "log10", "log2", "acos", "acosh", "asin", "asinh", "atan", "atanh", "exp", "expm1", "cos", "cosh", "sin", "sinh", "tan", "tanh", "gamma", "lgamma"

`Summary` "max", "min", "range", "prod", "sum", "any", "all"

See S4groupGeneric for more details.

Numerical Data Methods

In the code snippets below, `x` is a DuckDBTable object:

`is.finite(x)`: Returns a DuckDBTable containing logicals that indicate which values are finite.

`is.infinite(x)`: Returns a DuckDBTable containing logicals that indicate which values are infinite.

`is.nan(x)`: Returns a DuckDBTable containing logicals that indicate which values are Not a Number.

`mean(x)`: Calculates the mean of `x`.

`var(x)`: Calculates the variance of `x`.

`sd(x)`: Calculates the standard deviation of `x`.

`median(x)`: Calculates the median of `x`.

`quantile(x, probs = seq(0, 1, 0.25), names = TRUE, type = 7)`: Calculates the specified quantiles of `x`.

`probs` A numeric vector of probabilities with values in [0,1].

`names` If `TRUE`, the result has names describing the quantiles.

`type` Either 1 or 7 that specifies the quantile algorithm detailed in `quantile`.

`mad(x, constant = 1.4826)`: Calculates the median absolute deviation of `x`.

`constant` The scale factor.

`IQR(x, type = 7)`: Calculates the interquartile range of `x`.

`type` Either 1 or 7 that specifies the quantile algorithm detailed in `quantile`.

`pmax(..., na.rm = FALSE)`: Returns the parallel maxima of multiple DuckDBTable objects. All arguments must be DuckDBTable objects with compatible dimensions.

`pmin(..., na.rm = FALSE)`: Returns the parallel minima of multiple DuckDBTable objects. All arguments must be DuckDBTable objects with compatible dimensions.

Character Methods

In the code snippets below, `x` is a DuckDBTable object:

`nchar(x)`: Returns a DuckDBTable containing the number of characters in each string.

`tolower(x)`: Returns a DuckDBTable with all strings converted to lowercase.

`toupper(x)`: Returns a DuckDBTable with all strings converted to uppercase.

`chartr(old, new, x)`: Returns a DuckDBTable with characters translated.

`old` Characters to be translated.

`new` Characters to translate to.

`substr(x, start, stop)`: Returns a DuckDBTable containing substrings extracted by position.

`start` Integer starting position (1-indexed).

`stop` Integer ending position (inclusive).

`substring(x, first, last = 1000000L)`: Returns a DuckDBTable containing substrings extracted by position.

`first` Integer starting position (1-indexed).

`last` Integer ending position (inclusive).

`grepl(pattern, x, ignore.case = FALSE, fixed = FALSE)`: Returns a DuckDBTable containing logicals indicating pattern matches.

`pattern` Character string containing a regular expression.

`ignore.case` If `TRUE`, case-insensitive matching.

`fixed` If `TRUE`, pattern is a fixed string not regex.

`sub(pattern, replacement, x, ignore.case = FALSE, fixed = FALSE)`: Returns a DuckDBTable with first match of pattern replaced.

`pattern` Character string containing a regular expression.

`replacement` Replacement string.

`ignore.case` If `TRUE`, case-insensitive matching.

`fixed` If `TRUE`, pattern is a fixed string not regex.

`gsub(pattern, replacement, x, ignore.case = FALSE, fixed = FALSE)`: Returns a DuckDBTable with all matches of pattern replaced.

`pattern` Character string containing a regular expression.

`replacement` Replacement string.

`ignore.case` If `TRUE`, case-insensitive matching.

`fixed` If `TRUE`, pattern is a fixed string not regex.

`startsWith(x, prefix)`: Returns a DuckDBTable containing logicals indicating if strings start with the specified prefix.

`endsWith(x, suffix)`: Returns a DuckDBTable containing logicals indicating if strings end with the specified suffix.

`paste(..., sep = " ", collapse = NULL)`: Concatenates multiple DuckDBTable objects column-wise using the specified separator. The `collapse` argument is not supported. All arguments must be DuckDBTable objects with compatible dimensions.

General Methods

In the code snippets below, `x` is a DuckDBTable object:

`unique(x)`: Returns a DuckDBTable containing the distinct rows.

`x %in% table`: Returns a DuckDBTable containing logicals that indicate if the values in each of the columns of `x` are in `table`.

`table(...)`: Returns a table containing the counts across the distinct values.

List Methods

In the code snippets below, `x` is a DuckDBTable object:

`elementNROWS(x)`: Returns a DuckDBTable containing the number of elements in each list column of `x`.

Sparsity Methods

In the code snippets below, x is a DuckDBTable object:

`is_nonzero(x)`: Returns a DuckDBTable containing logicals that indicate if the values in each of the columns of x are non-zero.

`nzcount(x)`: Returns the total number of non-zero values.

`is_sparse(x)`: Returns `TRUE` since data are stored in a sparse array representation.

Author(s)

Patrick Aboyoun

See Also

- `DuckDBTable-class` for the main class
- `RectangularData` for the base class

Examples

```
tf <- tempfile(fileext = ".parquet")
on.exit(unlink(tf))
arrow::write_parquet(cbind(model = rownames(mtcars), mtcars), tf)
tbl <- DuckDBTable(tf, datacols = colnames(mtcars), keycol = "model")
mean(tbl[, "mpg"])
head(tbl, 3)
```

DuckDBTransposedDataFrame-class

DuckDBTransposedDataFrame objects

Description

The DuckDBTransposedDataFrame class extends TransposedDataFrame to represent a DuckDB table as a TransposedDataFrame object.

Details

DuckDBTransposedDataFrame objects are constructed by calling `t()` on a DuckDBDataFrame object.

Value

Objects of class `DuckDBTransposedDataFrame` extend RectangularData.

Constructor

`t(x)`: Creates a DuckDBTransposedDataFrame object from a DuckDBDataFrame object.

Accessors

In the code snippets below, x is a DuckDBTransposedDataFrame object:

`dim(x)`: Length two integer vector defined as `c(nrow(x), ncol(x))`.

`nrow(x)`, `ncol(x)`: Get the number of rows and columns, respectively.

`NROW(x)`, `NCOL(x)`: Same as `nrow(x)` and `ncol(x)`, respectively.

`dimnames(x)`: Length two list of character vectors defined as `list(rownames(x), colnames(x))`.

`rownames(x)`, `colnames(x)`: Get the names of the rows and columns, respectively.

Subsetting

In the code snippets below, x is a DuckDBTransposedDataFrame object:

`x[i, j, drop = TRUE]`: Returns either a new DuckDBTransposedDataFrame object or a DuckDB-Column if selecting a single row and `drop = TRUE`.

Displaying

The `show()` method for DuckDBTransposedDataFrame objects obeys global options `showHeadLines` and `showTailLines` for controlling the number of head and tail rows to display.

Author(s)

Patrick Aboyoun

See Also

- `DuckDBDataFrame-class` for the DataFrame class
- `TransposedDataFrame` for the base class

Examples

```
# Mocking up a file:
tf <- tempfile(fileext = ".parquet")
on.exit(unlink(tf))
arrow::write_parquet(cbind(model = rownames(mtcars), mtcars), tf)

# Creating our DuckDB-backed data frame:
tdf <- t(DuckDBDataFrame(tf, datacols = colnames(mtcars), keycol = "model"))
tdf

# Slicing DuckDBTransposedDataFrame objects:
tdf[,1:5]
tdf[1:5,]
```

keynames *Key Names and Key Count*

Description

Get the keycols names, keycols dimension names, or keycols count of an object.

Usage

```
keynames(x)

keydimnames(x, value)

keydimnames(x) <- value

nkey(x)

nkeydim(x)
```

Arguments

x	An object to get the keycols related information.
value	A character vector of keycols dimension names.

Value

keynames and keydimnames return character vectors. nkey and nkeydim return a single integer.

Author(s)

Patrick Aboyoun

Examples

```
keynames
showMethods("keynames")

keydimnames
showMethods("keydimnames")

nkey
showMethods("nkey")
```

parquet-io *Parquet I/O utilities for the DuckDB package suite*

Description

Shared append validation, SQL `COPY TO` helpers, and lazy DuckDBTable export used by **BiocDuckDB** and **DuckDBArray**. End users should call `writeParquet` or `writeCoordArray` rather than these functions directly.

Usage

```
validateAppendOffset(offset)

readParquetSchema(path, columns = NULL)

reconcileParquetSchema(path, columns, arrowtypes)

checkHiveAppendPartitions(
  path,
  indexcols,
  grid_suffix,
  grid,
  along,
  group_offset
)

checkAppendPart(path, part, part_digits = 0L)

parquetPartPath(path, part, part_digits = 0L)

validateWriteParquetPart(part)

setupFlatParquetWrite(
  path,
  append = FALSE,
  offset = 0L,
  part = NULL,
  part_digits = 0L,
  indexcol = NULL,
  reconcile_columns = NULL,
  create = TRUE
)

escapeSQLPath(path)

quoteSQLColumns(conn, cols)

buildParquetCopySQL(
  query_sql,
  target_path,
  order_cols = NULL,
```

```
  partition_by = NULL,
  row_group_size = NULL,
  append = FALSE
)

writeDuckDBTableParquet(
  x,
  path,
  indexcol = "__index__",
  keycol = "__name__",
  dimtbl = NULL,
  append = FALSE,
  offset = 0L,
  part = NULL,
  part_digits = 0L,
  cluster_by = NULL,
  index_max = NULL,
  row_group_size = 491520L,
  ...
)

splitParquetPart(
  path,
  n_parts,
  part_digits = 0L,
  conn = acquireDuckDBConn(),
  row_group_size = 491520L
)
```

Arguments

offset	Non-negative integer scalar; global index shift for flat table append (validateAppendOffset).
path	Directory containing Parquet files or hive partitions.
columns	Character vector of column names required in an existing dataset schema.
arrowtypes	Named list of DataType objects or length-one character Arrow type names (names must match columns). NULL elements adopt the on-disk type; non-NULL elements must match the existing type exactly (reconcileParquetSchema).
indexcols	Character vector of index column names for hive layout.
grid_suffix	Partition directory suffix (e.g. "group__").
grid	ArrayGrid for the current write slab.
along	Integer index of the append dimension along which group_offset applies.
group_offset	Partition-group offset along along for hive append.
part	Non-negative integer; flat part-<n>.parquet index.
part_digits	Zero-padding width for part in the filename.
append	Logical; enable flat multi-part append (setupFlatParquetWrite).
indexcol	Optional index column name for lazy table export.
reconcile_columns	Character vector of columns to pin on append via reconcileParquetSchema.
create	Logical; create path when missing (setupFlatParquetWrite).

conn	DBI connection for quoteSQLColumns and splitParquetPart.
cols	Character vector of column names to quote.
query_sql	Inner SELECT (or SELECT ... WHERE ...) for buildParquetCopySQL.
target_path	Destination file or directory path for buildParquetCopySQL.
order_cols	Optional character vector of ORDER BY expressions (pre-quoted identifiers or SQL expressions).
partition_by	Optional character vector of hive partition column names (pre-quoted identifiers) for PARTITION_BY.
row_group_size	Optional integer ROW_GROUP_SIZE. Used by buildParquetCopySQL and by writeDuckDBTableParq which defaults it to 491520L (240 x the 2048 DuckDB STANDARD_VECTOR_SIZE) to match the convention used by the coord-array and flat arrow writers; previously the lazy export fell back to DuckDB's 122880 default.
x	A DuckDBTable object (writeDuckDBTableParquet).
keycol	Optional primary-key column name for lazy table export.
dimtbl	Optional single-row DataFrame of partition columns.
n_parts	Number of files to split into (splitParquetPart).

Details

These functions implement the shared append contract (fail before write, no silent overwrite):

- Flat sample/feature tables use part-*.parquet files; checkAppendPart and reconcileParquetSchema apply.
- Hive-partitioned coord arrays use <index><suffix>=<n>/ directories; checkHiveAppendPartitions guards partition collisions.
- readParquetSchema reads the first *.parquet file found under path (recursive search).

Value

validateAppendOffset Integer scalar.

readParquetSchema Schema object.

reconcileParquetSchema Named list of resolved DataType objects.

checkHiveAppendPartitions, checkAppendPart NULL, invisibly.

parquetPartPath Character path to a part-*.parquet file.

validateWriteParquetPart Integer scalar or NULL.

setupFlatParquetWrite Named list with validated flat-write parameters (path, part, offset, pq_path, flat_part, subsequent_part).

escapeSQLPath Escaped path string.

quoteSQLColumns Character vector of quoted identifiers.

buildParquetCopySQL Complete COPY TO SQL string.

writeDuckDBTableParquet List with path, dir, nrow, sample_df, colnames, part, append, and subsequent_part.

splitParquetPart Character vector of the resulting part-*.parquet paths, invisibly.

Functions

- `writeDuckDBTableParquet()`: Lazy DuckDBTable SQL Parquet export.
- `splitParquetPart()`: Split a directory's single flat `part-*.parquet` file into `n_parts` smaller ones, in place, preserving row order and factor columns.

Author(s)

Patrick Aboyoun

See Also

`writeParquet`, `writeCoordArray`

Examples

```
path <- tempfile()
dir.create(path)
on.exit(unlink(path, recursive = TRUE), add = TRUE)
pq <- file.path(path, "part-0.parquet")
arrow::write_parquet(data.frame(x = 1:3L, y = letters[1:3]), pq)
validateAppendOffset(0L)
readParquetSchema(path, columns = c("x", "y"))
tbl <- DuckDBTable(pq)
quoteSQLColumns(dbconn(tbl), c("x", "y"))

path2 <- tempfile()
dir.create(path2)
on.exit(unlink(path2, recursive = TRUE), add = TRUE)
arrow::write_parquet(data.frame(x = 1:100L), file.path(path2, "part-0.parquet"))
splitParquetPart(path2, n_parts = 4L)
list.files(path2)
```

---

parquet-io-cluster *Cluster-on-write keys and helpers*

Description

Clustering keys for `writeParquet`'s `cluster_by` argument, plus the in-memory reorder they imply. Writing with a clustering key physically orders rows so each Parquet row group covers a tight region, letting DuckDB's row-group zonemaps prune groups outside a range query.

Usage

```
zorder(cols, bits = 16L, by = NULL)

hilbert(cols, bits = 16L, by = NULL)

clusterSort(df, cluster_by)

clusterOrderSQL(conn, subquery_sql, cluster_by, available = NULL)

columnExtents(conn, subquery_sql, cols)
```

Arguments

cols	Character vector of numeric column names to cluster by. hilbert() requires exactly two.
bits	Grid resolution per axis (default 16; a 2^{16} grid). Higher = finer; keep length(cols) * bits <= 62 so the Morton code fits a 64-bit integer.
by	Optional character vector of columns ordered lexicographically before the curve (a composite categorical-prefix key); NULL for a pure space-filling key.
df	A data.frame or DataFrame (clusterSort).
cluster_by	A zorder()/hilbert() spec or a character vector of columns (clusterSort).
conn	A DuckDB DBIConnection (clusterOrderSQL, columnExtents).
subquery_sql	A SQL SELECT string whose output rows are to be ordered (clusterOrderSQL) or measured (columnExtents).
available	Optional character vector of the subquery's output column names, used to validate that the cluster_by columns exist (clusterOrderSQL).

Details

zorder() orders by a Morton (Z-order) code over cols (any number of numeric columns), lowered SQL-side to a generated bit-interleave ORDER BY (no extension needed). hilbert() orders by the native DuckDB spatial ST_Hilbert curve (exactly two numeric columns, better locality, requires the spatial extension). A plain character vector is a lexicographic ordering (best for the leading column). clusterSort() is the in-memory counterpart used by the materializing data.frame / DataFrame write path (Hilbert falls back to Morton there). Its double-precision Morton code is exact only for length(cols) * bits <= 52; the lazy SQL path (unsigned 64-bit) is exact to 62, so for a total above 52 bits the two write paths may order rows differently. Keep length(cols) * bits at or below 52 when both paths must agree byte-for-byte.

by adds a *composite* key: the named (typically low-cardinality categorical) columns are ordered lexicographically *before* the space-filling curve, i.e. ORDER BY by..., curve(cols). This keeps each by-group's rows contiguous so the group's own zonemaps / run-length encoding stay tight (e.g. zorder(c("x","y"), by = "gene") for a viewport+gene workload, or zorder(c("x","y"), by = "__sample__") to keep a COO assay's sample-slice pruning). The prefix columns are **not** interleaved into the Morton code, so they cost none of the bits budget.

clusterOrderSQL() is the SQL-side counterpart of clusterSort(): it lowers a cluster_by spec (from zorder() / hilbert() or a plain character vector) plus an inner subquery_sql into a character vector of ORDER BY expressions suitable for buildParquetCopySQL's order_cols. It is the primitive a lazy or coordinate-array writer (e.g. **DuckDBArray**) calls to cluster a COPY TO without re-deriving the Morton/Hilbert generator, so the whole suite keeps one curve implementation. Returns NULL for a NULL spec. columnExtents() returns each column's finite [min, max] over the subquery (one cheap scalar pass), useful for building bounding-box windows or grid resolutions downstream.

Value

zorder, hilbert A DuckDBClusterSpec for writeParquet(cluster_by=).

clusterSort df reordered by the clustering key (same class, same columns); a no-op when cluster_by is NULL, df is empty, or a key column is absent.

clusterOrderSQL Character vector of ORDER BY expressions for buildParquetCopySQL(order_cols=), or NULL when cluster_by is NULL.

columnExtents List (one element per column) of numeric c(min, max) finite extents.

Examples

```
zorder(c("x", "y"))
hilbert(c("x", "y"), bits = 12L)
clusterSort(data.frame(x = c(9, 1, 5), y = c(9, 1, 5)), zorder(c("x", "y")))
```

`set_row_number` *Internal generics and helpers for the BiocDuckDB suite*

Description

These generics and helper functions are the low-level extension points shared across the BiocDuckDB packages (for example **DuckDBArray** and **DuckDBGRanges** define methods on the generics). They are exported so companion packages can dispatch methods on them across package boundaries, but they are not part of the public user-facing API and are not intended for direct use by end users.

Usage

```
set_row_number(conn)

keycols(x)

has_row_number(x)

makePrettyCharacterMatrixForDisplay(x)
```

Arguments

`conn`	A `tbl_duckdb_connection`.
`x`	A DuckDBTable-derived object.

Value

`keycols()` returns the named list of key-dimension index vectors. `has_row_number()` returns a logical scalar indicating whether the object is addressed by a generated row number (rather than named keys). `set_row_number()` returns the `integer64` row-number encoding for a connection. `makePrettyCharacterMatrixForDisplay()` returns a character matrix used by the `show()` method.

`sql_call` *Call DuckDB SQL Functions on BiocDuckDB Objects*

Description

Apply any DuckDB SQL function to a BiocDuckDB object. This is a low-level interface that provides direct access to DuckDB's extensive SQL function library. The function is applied lazily and only executed when the result is materialized.

Usage

```
sql_call(x, fun, ...)
```

Arguments

x	A BiocDuckDB object (DuckDBTable, DuckDBColumn, DuckDBDataFrame, or DuckDBAtomicList).
fun	A character string naming the SQL function to call.
...	Additional arguments passed to the SQL function. Arguments are coerced to SQL literals or column references as appropriate.

Details

This function applies the specified SQL function to all data columns in the object. For common operations, consider using dedicated methods or convenience wrappers (e.g., `sort`, `unique`, `%in%`).

Available SQL functions are documented in DuckDB's function reference: https://duckdb.org/docs/sql/functions/overview

Value

An object of the same class as `x` with the SQL function applied to all data columns. The operation is lazy and only executed when the result is materialized via `as.vector()`, `as.list()`, or similar.

Argument Handling

Arguments in `...` are converted to SQL expressions as follows:

- **Character**: SQL string literal (automatically quoted)
- **Numeric**: SQL numeric literal
- **Logical**: SQL boolean literal (TRUE/FALSE)
- **NULL**: SQL NULL
- **name/symbol**: SQL column reference (unquoted, from `as.name()`)
- **sql()**: Raw SQL expression (from `dplyr::sql()`, not quoted)

Author(s)

Patrick Aboyoun

Examples

```
# sql_call() applies any DuckDB SQL function to a column
df <- data.frame(x = c(1.4, 2.6, 3.5), y = c(-1, 2, -3))
path <- tempfile(fileext = ".parquet")
arrow::write_parquet(df, path)
ddf <- DuckDBDataFrame(path, datacols = c("x", "y"))

sql_call(ddf$x, "round", 0)   # ROUND(x, 0)
sql_call(ddf$y, "abs")        # ABS(y)

showMethods("sql_call")
unlink(path)
```

sql_fun *Find Compatible DuckDB SQL Functions for BiocDuckDB Objects*

Description

Query DuckDB's function catalog to find SQL functions that are compatible with a BiocDuckDB object's data type. This is useful for discovering what operations can be performed on a particular column type.

Usage

```
sql_fun(x, function_type = NULL, return_type = NULL, description = FALSE)
```

Arguments

x	A BiocDuckDB object (DuckDBTable, DuckDBColumn, DuckDBDataFrame, or DuckDBAtomicList).
function_type	Optional character vector of function types to filter for. If NULL (default), returns all function types. Options: "scalar", "aggregate", "macro", "table", "pragma", "table_macro"
return_type	Optional character vector of return types to filter for. If NULL (default), returns all functions. Common values: "BOOLEAN", "INTEGER", "DOUBLE", "VARCHAR", "ANY", "INTEGER[]", "DOUBLE[]", "VARCHAR[]", "ANY[]", etc.
description	Logical; if TRUE, include function descriptions in output. Default is FALSE.

Details

This function queries DuckDB directly via DESCRIBE to get the column's exact type (e.g., INTEGER[], DOUBLE[], VARCHAR[]), then searches duckdb_functions() for functions that accept that type as the first parameter. It matches both the exact type and generic types (T[], ANY[], ANY) that work with any type.

The results are grouped by function name with distinct return types collected into a list for functions that have multiple overloads.

Value

A data.frame with the following columns:

- **function_name**: Name of the function
- **alias_of**: If this is an alias, the canonical function name
- **function_type**: Type of function (scalar, aggregate, etc.)
- **return_type**: List of distinct return types for this function
- **description**: (if `description = TRUE`) Function description

Author(s)

Patrick Aboyoun

Examples

```
# sql_fun() lists the DuckDB SQL functions applicable to a column
df <- data.frame(x = c(1.4, 2.6, 3.5), y = c(-1, 2, -3))
path <- tempfile(fileext = ".parquet")
arrow::write_parquet(df, path)
ddf <- DuckDBDataFrame(path, datacols = c("x", "y"))

head(sql_fun(ddf$x))                            # all applicable functions
head(sql_fun(ddf$x, function_type = "scalar"))  # filter by function type

showMethods("sql_fun")
unlink(path)
```

---

tblconn *Accessing DB Table Information*

Description

Get a database table connection object

Usage

```
tblconn(x, select = TRUE, filter = TRUE)
```

Arguments

x	An object with a database table connection.
select	A logical value indicating whether to apply column selections to the database table connection.
filter	A logical value indicating whether to apply key column filtering to the database table connection.

Value

returns a database table connection object.

Author(s)

Patrick Aboyoun

Examples

```
tblconn
showMethods("tableconn")
```

Index