Package: DuckDBArray (via r-universe)

November 25, 2026

Version 1.0.0

Date 2026-11-21

Title DuckDB-Backed DelayedArray Implementation

Description Provides DuckDB-backed implementations of DelayedArray and DelayedMatrix classes for efficient, out-of-memory operations on large array and matrix datasets. The package includes DuckDBArray, DuckDBArraySeed, and DuckDBMatrix classes with optimized matrixStats methods. Supports sparse arrays, grid partitioning, and parallel processing for scalable genomic data analysis.

License MIT + file LICENSE

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

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

Depends R (>= 4.5.0), methods, stats, DuckDBDataFrame, DelayedArray, SparseArray

Imports BiocGenerics, S4Vectors, IRanges, S4Arrays, Matrix, MatrixGenerics, DBI, dplyr, dbplyr, arrow

Suggests knitr, rmarkdown, BiocStyle, testthat, nanoparquet, matrixStats, microbenchmark, BiocParallel, DelayedMatrixStats, ExperimentHub, HDF5Array, sparseMatrixStats, SummarizedExperiment, TileDBArray, airway

biocViews DataImport, DataRepresentation, Infrastructure, Software

VignetteBuilder knitr

Encoding UTF-8

Collate 'DuckDBArraySeed-class.R' 'DuckDBArraySeed-utils.R' 'DuckDBArray-class.R' 'DuckDBTable-matrixStats.R' 'DuckDBArraySeed-matrixStats.R' 'DuckDBArray-matrixStats.R' 'DuckDBArray-package.R' 'DuckDBArray-utils.R' 'DuckDBMatrix-class.R' 'DuckDBMatrix-utils.R' 'createDimTables.R' 'rowDeviances.R' 'rowNnzs.R' 'scanKeycols.R' 'writeCoordArray-duckdb.R' 'writeCoordArray-generic.R' 'writeCoordArray-methods.R' 'writeCoordArray-types.R'

Config/roxygen2/version 8.1.1

Config/pak/sysreqs cmake libicu-dev libxml2-dev libssl-dev zlib1g-dev

Repository Bioconductor 3.24

Date/Publication 2026-11-23 22:56:41 UTC

RemoteUrl https://github.com/bioc/DuckDBArray

RemoteRef RELEASE_3_24

RemoteSha ac149ba724d880304fa14a1b2b4742b403e3e221

Contents

createDimTables 2
DuckDBArray-class 4
DuckDBArray-matrixStats 6
DuckDBArray-utils 7
DuckDBArraySeed-class 9
DuckDBArraySeed-matrixStats 11
DuckDBArraySeed-utils 13
DuckDBMatrix-class 14
DuckDBMatrix-utils 16
DuckDBTable-matrixStats 17
rowDeviances 19
rowNnzs 21
scanKeycols 23
writeCoordArray 24

Index **29**

`createDimTables` *Create Dimension Lookup Tables*

Description

Creates lookup tables for array dimensions that map dimension names to their corresponding indices and grid group identifiers. These tables are useful for understanding the structure of partitioned arrays and for reconstructing the original array from coordinate format data.

Usage

```
createDimTables(
  x,
  indexcols = names(dimnames(x)) %||% sprintf("index%d", seq_along(dim(x))),
  grid = defaultAutoGrid(COO_SparseArray(dim(x))),
  grid_suffix = "_group",
  BPPARAM = getAutoBPPARAM()
)
```

Arguments

x	An array-like object with dimensions and dimension names. Must be coercible to a SparseArray object. Common examples include matrices, arrays, and table objects.
indexcols	A character vector of column names for the dimensions of the array.
grid	An optional ArrayGrid to use for partitioning the array. If NULL, uses defaultAutoGrid(COO_SparseArra
grid_suffix	A character string to append to the partitioning directories if the grid contains more than one cell. Defaults to "_group".
BPPARAM	A BiocParallelParam object to use for parallel processing when grid contains multiple cells. Defaults to getAutoBPPARAM().

Details

This function is particularly useful when working with partitioned arrays created by writeCoordArray. The returned lookup tables provide a mapping between dimension names and their positions in the original array, as well as identify which grid partition contains each dimension element.

When the grid contains only one cell, the lookup tables contain only the dimension names and their indices. When multiple grid cells are present, additional columns indicate which grid group each dimension element belongs to.

grid defaults identically to writeCoordArray's default (and should normally be the same grid passed to writeCoordArray, so the lookup tables' group identifiers match the partitions actually on disk); see writeCoordArray's "Sizing grid for large or sparse arrays" section for why that default is a dense-element budget with no sparsity awareness, and how to correct it for a sparse x.

Value

A named list of data frames, one for each dimension in x. Each data frame contains:

- The dimension names as row names
- A column with the corresponding indices
- A column with the grid group identifier (if grid has multiple cells)

The list is named according to indexcols.

Author(s)

Patrick Aboyoun

See Also

writeCoordArray for writing arrays in coordinate format, ArrayGrid for grid partitioning, BiocParallelParam for parallel processing options

Examples

```
# Write the state.x77 matrix to multiple Parquet files using grid partitioning
tf <- tempfile()
state_grid <- RegularArrayGrid(dim(state.x77), c(10, 4))
writeCoordArray(state.x77, file.path(tf, "state"), grid = state_grid)
dimtbls <- createDimTables(state.x77, grid = state_grid)
list.files(tf, full.names = TRUE, recursive = TRUE)
```

DuckDBArray-class *DuckDBArray objects*

Description

The DuckDBArray class is a DelayedArray subclass for representing and operating on a DuckDB table.

All the operations available for DelayedArray objects work on DuckDBArray objects.

Value

The `DuckDBArray()` constructor returns a DuckDBArray object. The accessors return the component of `x` named by the accessor (e.g. `dim(x)` returns an integer vector of the array dimensions).

Constructor

`DuckDBArray(conn, datacol, keycols, dimtbls = NULL, type = NULL)`**:** Creates a DuckDBArray 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.
- `datacol` Either a string specifying the column from `conn` or a named `expression` that will be evaluated in the context of `conn` that defines the values in the array.
- `keycols` Either a character vector of column names from `conn` that will specify the dimension names, or a named list of character vectors where the names of the list specify the dimension names and the character vectors set the distinct values for the dimension names.
- `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` String specifying the type of the data values; one of `"logical"`, `"integer"`, `"integer64"`, `"double"`, or `"character"`. If `NULL`, it is determined by inspecting the data.

Keycol uniqueness

`keycols` must uniquely identify each row of `conn`: for a two-dimensional array, each `(i, j)` combination of key column values is expected to appear at most once. This invariant is not currently enforced at construction time. If it is violated, behavior is unspecified and differs by representation: coercion to `dgCMatrix` (and hence `CsparseMatrix`, via `sparseMatrix`) silently SUMS the duplicate values, while every other materialization path (`as.matrix()`, `as(x, "COO_SparseArray")`, `as(x, "SparseArray")`) silently keeps the LAST one.

Accessors

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

`dim(x)`**:** An integer vector of the array dimensions.

`dimnames(x)`**,** `dimnames(x) <- value`**:** Get or set the array dimension names.

`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 array elements; one of `"logical"`, `"integer"`, `"integer64"`, `"double"`, or `"character"`.

Subsetting

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

`x[i, j, ..., drop = TRUE]`**:** Returns a new DuckDBArray object. Empty dimensions are dropped if `drop = TRUE`.

Transposition

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

`aperm(a, perm)`**:** Returns a new DuckDBArray object with the dimensions permuted according to the `perm` vector.

`t(x)`**:** For two-dimensional arrays, returns a new DuckDBArray object with the dimensions transposed.

Author(s)

Patrick Aboyoun

See Also

- `DuckDBArray-utils` for the utilities
- `DuckDBMatrix-class` for the matrix class
- `DuckDBArraySeed-class` for the internal seed class
- `DuckDBArraySeed-utils` for the internal seed class utilities
- `DelayedArray` 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 <- Titanic[as.matrix(df)]

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

pqarray <- DuckDBArray(tf, datacol = "fate", keycols = c("Class", "Sex", "Age", "Survived"))
```

`DuckDBArray-matrixStats` *DuckDBArray row / column summarization methods*

Description

Row / column summarization methods for DuckDBArray objects.

Value

Method return types are documented in the sections above.

Row / Column Summarization Methods

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

`rowCounts(x, value = TRUE)`**:** Calculates the row counts of `x` that are equal to `value`.

`value` The value to count.

`colCounts(x, value = TRUE)`**:** Calculates the column counts of `x` that are equal to `value`.

`value` The value to count.

`rowMaxs(x)`**:** Calculates the row maxima of `x`.

`colMaxs(x)`**:** Calculates the column maxima of `x`.

`rowMeans(x, dims = 1)`**:** Calculates the row means of `x`.

`dims` An integer specifying which dimensions to average over, namely `dims + 1, ...`.

`colMeans(x, dims = 1)`**:** Calculates the column means of `x`.

`dims` An integer specifying which dimensions to average over, namely `1:dims`.

`rowMins(x)`**:** Calculates the row minima of `x`.

`colMins(x)`**:** Calculates the column minima of `x`.

`rowSums(x, dims = 1)`**:** Calculates the row sums of `x`.

`dims` An integer specifying which dimensions to sum over, namely `dims + 1, ...`.

`colSums(x, dims = 1)`: Calculates the column sums of `x`.

`dims` An integer specifying which dimensions to sum over, namely `1:dims`.

`rowSds(x)`: Calculates the row standard deviations of `x`.

`colSds(x)`: Calculates the column standard deviations of `x`.

`rowVars(x)`: Calculates the row variances of `x`.

`colVars(x)`: Calculates the column variances of `x`.

Author(s)

Patrick Aboyoun

See Also

- `DuckDBArray-class` for the main class
- `DuckDBMatrix-class` for the matrix class
- `DuckDBArraySeed-class` for the internal seed class
- `DuckDBArraySeed-matrixStats` for the internal seed class matrixStats methods
- `DelayedArray` for the base class

Examples

```
df <- do.call(expand.grid, c(dimnames(Titanic), stringsAsFactors = FALSE))
df$fate <- as.integer(Titanic[as.matrix(df)])
df <- df[df$fate != 0L, ]
tf <- tempfile(fileext = ".parquet")
on.exit(unlink(tf))
arrow::write_parquet(df, tf)
pqarray <- DuckDBArray(tf, datacol = "fate",
                       keycols = c("Class", "Sex", "Age", "Survived"))
colMeans(pqarray)
rowSums(pqarray, dims = 2L)
```

DuckDBArray-utils *Common operations on DuckDBArray objects*

Description

Common operations on DuckDBArray objects.

Value

Method return types are documented in the sections above.

Group Generics

DuckDBArray 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 DuckDBArray object:

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

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

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

`sweep(x, MARGIN, STATS, FUN = "/")`: Sweeps out array summaries from `x`. Applies `FUN` to each element of `x` using the corresponding value from `STATS` based on `MARGIN`.

`MARGIN` integer specifying the dimension (1 for rows, 2 for columns)

`STATS` numeric vector with length equal to the extent of dimension `MARGIN`

`FUN` function to be used to carry out the sweep

Sparsity Methods

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

`is_nonzero(x)`: Returns a DuckDBArray 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.

`nzvals(x)`: Returns the non-zero values.

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

Author(s)

Patrick Aboyoun

See Also

- DuckDBArray-class for the main class
- DuckDBMatrix-class for the matrix class
- DuckDBArraySeed-class for the internal seed class
- DuckDBArraySeed-utils for the internal seed class utilities
- DelayedArray for the base class

Examples

```
df <- do.call(expand.grid, c(dimnames(Titanic), stringsAsFactors = FALSE))
df$fate <- as.integer(Titanic[as.matrix(df)])
df <- df[df$fate != 0L, ]
tf <- tempfile(fileext = ".parquet")
on.exit(unlink(tf))
arrow::write_parquet(df, tf)
pqarray <- DuckDBArray(tf, datacol = "fate",
                       keycols = c("Class", "Sex", "Age", "Survived"))
is_sparse(pqarray)
rowSums(pqarray)
pqarray + 1L
```

DuckDBArraySeed-class *DuckDBArraySeed objects*

Description

DuckDBArraySeed is a low-level helper class for representing a pointer to a DuckDB table.

Note that a DuckDBArraySeed object is not intended to be used directly. Most end users will typically create and manipulate a higher-level DuckDBArray object instead. See ?DuckDBArray for more information.

Value

The DuckDBArraySeed() constructor returns a DuckDBArraySeed object; the accessors return the component of x named by the accessor.

Constructor

DuckDBArraySeed(conn, datacol, keycols, dimtbls = NULL, type = NULL): Creates a DuckDBArraySeed 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.

datacol Either a string specifying the column from conn or a named expression that will be evaluated in the context of conn that defines the values in the array.

`keycols` Either a character vector of column names from `conn` that will specify the dimension names, or a named list of character vectors where the names of the list specify the dimension names and the character vectors set the distinct values for the dimension names.

`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` String specifying the type of the data values; one of `"logical"`, `"integer"`, `"integer64"`, `"double"`, or `"character"`. If `NULL`, it is determined by inspecting the data.

Keycol uniqueness

`keycols` must uniquely identify each row of `conn`: for a two-dimensional array, each `(i, j)` combination of key column values is expected to appear at most once. This invariant is not currently enforced at construction time. If it is violated, behavior is unspecified and differs by representation: coercion of the owning DuckDBArray/DuckDBMatrix to `dgCMatrix` (and hence `CsparseMatrix`, via `sparseMatrix`) silently SUMS the duplicate values, while every other materialization path (`as.matrix()`, `as(x, "COO_SparseArray")`, `as(x, "SparseArray")`) silently keeps the LAST one.

Accessors

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

`dim(x)`**:** An integer vector of the array dimensions.

`dimnames(x)`**,** `dimnames(x) <- value`**:** Get or set the array dimension names.

`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 array elements; one of `"logical"`, `"integer"`, `"integer64"`, `"double"`, or `"character"`.

Subsetting

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

`x[i, j, ..., drop = TRUE]`**:** Returns a new DuckDBArraySeed object. Empty dimensions are dropped if `drop = TRUE`.

Transposition

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

`aperm(a, perm)`**:** Returns a new DuckDBArraySeed object with the dimensions permuted according to the `perm` vector.

`t(x)`**:** For two-dimensional arrays, returns a new DuckDBArraySeed object with the dimensions transposed.

Author(s)

Patrick Aboyoun

See Also

- DuckDBArraySeed-utils for the utilities
- DuckDBArray-class for the main class
- DuckDBArray-utils for the main class utilities
- Array 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)

pqaseed <- DuckDBArraySeed(tf, datacol = "fate", keycols = c("Class", "Sex", "Age", "Survived"))
```

DuckDBArraySeed-matrixStats

DuckDBArraySeed row / column summarization methods

Description

Row / column summarization methods for DuckDBArraySeed objects.

Value

Method return types are documented in the sections above.

Row / Column Summarization Methods

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

`rowCounts(x, value = TRUE)`: Calculates the row counts of x that are equal to `value`.

`value` The value to count.

`colCounts(x, value = TRUE)`: Calculates the column counts of x that are equal to `value`.

`value` The value to count.

`rowMaxs(x)`: Calculates the row maxima of x.

`colMaxs(x)`: Calculates the column maxima of x.

`rowMeans(x, dims = 1)`: Calculates the row means of `x`.

`dims` An integer specifying which dimensions to average over, namely `dims + 1`, ...

`colMeans(x, dims = 1)`: Calculates the column means of `x`.

`dims` An integer specifying which dimensions to average over, namely `1:dims`.

`rowMins(x)`: Calculates the row minima of `x`.

`colMins(x)`: Calculates the column minima of `x`.

`rowSums(x, dims = 1)`: Calculates the row sums of `x`.

`dims` An integer specifying which dimensions to sum over, namely `dims + 1`, ...

`colSums(x, dims = 1)`: Calculates the column sums of `x`.

`dims` An integer specifying which dimensions to sum over, namely `1:dims`.

`rowSds(x)`: Calculates the row standard deviations of `x`.

`colSds(x)`: Calculates the column standard deviations of `x`.

`rowVars(x)`: Calculates the row variances of `x`.

`colVars(x)`: Calculates the column variances of `x`.

Author(s)

Patrick Aboyoun

See Also

- `DuckDBArraySeed-class` for the internal seed class
- `DuckDBArray-class` for the main class
- `DuckDBArray-matrixStats` for the main class matrixStats methods
- `Array` for the base class

Examples

```
df <- do.call(expand.grid, c(dimnames(Titanic), stringsAsFactors = FALSE))
df$fate <- as.integer(Titanic[as.matrix(df)])
df <- df[df$fate != 0L, ]
tf <- tempfile(fileext = ".parquet")
on.exit(unlink(tf))
arrow::write_parquet(df, tf)
seed <- DuckDBArraySeed(tf, datacol = "fate",
                        keycols = c("Class", "Sex", "Age", "Survived"))
colMeans(seed)
rowSums(seed, dims = 2L)
```

DuckDBArraySeed-utils *Common operations on DuckDBArraySeed objects*

Description

Common operations on DuckDBArraySeed objects.

Value

Method return types are documented in the sections above.

Group Generics

DuckDBArraySeed 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 DuckDBArraySeed object:

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

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

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

`sweep(x, MARGIN, STATS, FUN = "/")`: Sweeps out array summaries from `x`. Applies `FUN` to each element of `x` using the corresponding value from `STATS` based on `MARGIN`.

`MARGIN` integer specifying the dimension (1 for rows, 2 for columns)

`STATS` numeric vector with length equal to the extent of dimension `MARGIN`

`FUN` function to be used to carry out the sweep

Sparsity Methods

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

`is_nonzero(x)`: Returns a DuckDBArraySeed 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

- `DuckDBArraySeed-class` for the internal seed class
- `DuckDBArray-class` for the main class
- `DuckDBArray-utils` for the main class utilities
- `Array` for the base class

Examples

```
df <- do.call(expand.grid, c(dimnames(Titanic), stringsAsFactors = FALSE))
df$fate <- as.integer(Titanic[as.matrix(df)])
df <- df[df$fate != 0L, ]
tf <- tempfile(fileext = ".parquet")
on.exit(unlink(tf))
arrow::write_parquet(df, tf)
seed <- DuckDBArraySeed(tf, datacol = "fate",
                        keycols = c("Class", "Sex", "Age", "Survived"))
is_sparse(seed)
nzcount(seed)
seed + 1L
```

DuckDBMatrix-class *DuckDBMatrix objects*

Description

The DuckDBMatrix class is a DelayedMatrix subclass for representing and operating on a DuckDB table.

All the operations available for DelayedMatrix objects work on DuckDBMatrix objects.

Value

The `DuckDBMatrix()` constructor returns a DuckDBMatrix object; the accessors return the component of x named by the accessor.

Constructor

`DuckDBMatrix(conn, datacol, row, col, keycols = c(row, col), dimtbls = NULL, type = NULL)`**:** Creates a DuckDBMatrix 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.
- `datacol` Either a string specifying the column from `conn` or a named `expression` that will be evaluated in the context of `conn` that defines the values in the matrix.
- `row` Either a string that specifies the column in `conn` that specifies the row names of the matrix, or a named list containing a single character vector that defines the column in `conn` for the row names and its values.
- `col` Either a string that specifies the column in `conn` that specifies the column names of the matrix, or a named list containing a single character vector that defines the column in `conn` for the column names and its values.
- `keycols` An optional character vector that define the names of the columns in `conn` for the rows and columns of the matrix, or a named list of character vectors where the names of the list define rows and columns and the character vectors define distinct values for the rows and columns.
- `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` String specifying the type of the data values; one of `"logical"`, `"integer"`, `"integer64"`, `"double"`, or `"character"`. If `NULL`, it is determined by inspecting the data.

Accessors

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

`dim(x)`**:** An integer vector of the array dimensions.

`dimnames(x)`**,** `dimnames(x) <- value`**:** Get or set the array dimension names.

`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 array elements; one of `"logical"`, `"integer"`, `"integer64"`, `"double"`, or `"character"`.

Subsetting

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

`x[i, j, ..., drop = TRUE]`**:** Returns a new DuckDBMatrix object. Empty dimensions are dropped if `drop = TRUE`.

Transposition

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

`aperm(a, perm)`: Returns a new DuckDBMatrix object with the dimensions permuted according to the perm vector.

`t(x)`: Returns a new DuckDBMatrix object with the rows and columns transposed.

Author(s)

Patrick Aboyoun

See Also

- DuckDBArray-class for the array class
- DuckDBArray-utils for the array utilities
- DuckDBArraySeed-class for the internal seed class
- DuckDBArraySeed-utils for the internal seed class utilities
- DelayedMatrix for the base class

Examples

```
# Create a data.frame from a matrix
df <- data.frame(
  rowname = rep(rownames(state.x77), times = ncol(state.x77)),
  colname = rep(colnames(state.x77), each = nrow(state.x77)),
  value = as.vector(state.x77)
)

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

pqmat <- DuckDBMatrix(tf, row = "rowname", col = "colname", datacol = "value")
```

DuckDBMatrix-utils *DuckDBMatrix utility methods*

Description

Matrix multiplication methods for DuckDBMatrix objects.

Value

Method return types are documented in the sections above.

Matrix Multiplication Methods

In the code snippets below, `x` is a DuckDBMatrix object and `y` is an ordinary matrix or vector:

`x %*% y`: Matrix multiplication via SQL aggregation.

`crossprod(x, y)`: Cross-product, equivalent to `t(x) %*% y`.

`tcrossprod(x, y)`: Transposed cross-product, equivalent to `x %*% t(y)`.

`crossprod(x)`: Self cross-product via SQL self-join.

`tcrossprod(x)`: Self transposed cross-product via SQL self-join.

Author(s)

Patrick Aboyoun

See Also

- DuckDBMatrix-class for the main class
- DuckDBArray-utils for array utilities

Examples

```
state_df <- data.frame(
  index1 = rep(rownames(state.x77), times = ncol(state.x77)),
  index2 = rep(colnames(state.x77), each = nrow(state.x77)),
  value = as.vector(state.x77)
)
state_df <- subset(state_df, value != 0)
tf <- tempfile(fileext = ".parquet")
on.exit(unlink(tf))
arrow::write_parquet(state_df, tf)
pqmat <- DuckDBMatrix(tf, datacol = "value",
                      keycols = list(index1 = rownames(state.x77),
                                     index2 = colnames(state.x77)))
y <- rep(1, ncol(pqmat))
pqmat %*% y
dim(crossprod(pqmat))
```

DuckDBTable-matrixStats

DuckDBTable row / column summarization methods

Description

Row / column summarization methods for DuckDBTable-class objects.

Value

Method return types are documented in the sections above.

Row / Column Summarization Methods

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

`rowCounts(x, value = TRUE)`: Calculates the row counts of x that are equal to value.

value The value to count.

`colCounts(x, value = TRUE)`: Calculates the column counts of x that are equal to value.

value The value to count.

`rowMaxs(x)`: Calculates the row maxima of x.

`colMaxs(x)`: Calculates the column maxima of x.

`rowMeans(x, dims = 1)`: Calculates the row means of x.

dims An integer specifying which dimensions to average over, namely dims + 1, ...

`colMeans(x, dims = 1)`: Calculates the column means of x.

dims An integer specifying which dimensions to average over, namely 1:dims.

`rowMins(x)`: Calculates the row minima of x.

`colMins(x)`: Calculates the column minima of x.

`rowSums(x, dims = 1)`: Calculates the row sums of x.

dims An integer specifying which dimensions to sum over, namely dims + 1, ...

`colSums(x, dims = 1)`: Calculates the column sums of x.

dims An integer specifying which dimensions to sum over, namely 1:dims.

`rowSds(x)`: Calculates the row standard deviations of x.

`colSds(x)`: Calculates the column standard deviations of x.

`rowVars(x)`: Calculates the row variances of x.

`colVars(x)`: Calculates the column variances of x.

Sweep Method

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

`sweep(x, MARGIN, STATS, FUN = "/")`: Sweeps out array summaries from x. Applies FUN to each element of x using the corresponding value from STATS based on MARGIN.

MARGIN integer specifying the dimension (1 for rows, 2 for columns)

STATS numeric vector with length equal to the extent of dimension MARGIN

FUN function to be used to carry out the sweep

Author(s)

Patrick Aboyoun

See Also

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

Examples

```
df <- do.call(expand.grid, c(dimnames(Titanic), stringsAsFactors = FALSE))
df$fate <- as.integer(Titanic[as.matrix(df)])
df <- df[df$fate != 0L, ]
tf <- tempfile(fileext = ".parquet")
on.exit(unlink(tf))
arrow::write_parquet(df, tf)
tbl <- DuckDBTable(tf, datacols = "fate",
                   keycols = c("Class", "Sex", "Age", "Survived"))
rowSums(tbl)
colMeans(tbl)
```

---

rowDeviances *Row-wise Deviance Statistics*

Description

Compute per-row (per-gene) deviance statistics for feature selection in single-cell RNA-seq data. Genes with high deviance are likely to be informative for downstream analysis.

Usage

```
rowDeviances(x, family = c("binomial", "poisson"), ...)

## S4 method for signature 'matrix'
rowDeviances(x, family = c("binomial", "poisson"), ...)

## S4 method for signature 'dgCMatrix'
rowDeviances(x, family = c("binomial", "poisson"), ...)

## S4 method for signature 'DelayedMatrix'
rowDeviances(
  x,
  family = c("binomial", "poisson"),
  ...,
  grid = NULL,
  as.sparse = NA,
  BPPARAM = getAutoBPPARAM()
)

## S4 method for signature 'DuckDBMatrix'
rowDeviances(x, family = c("binomial", "poisson"), ...)
```

Arguments

x	A matrix-like object with genes in rows and cells in columns. Typically a count matrix (not log-transformed).

family	Character string specifying the null model distribution. Either "binomial" (default) or "poisson".
...	Additional arguments (currently unused).
grid	For DelayedMatrix objects, an optional ArrayGrid object specifying the block structure. Defaults to colAutoGrid(x) for column-wise processing since deviance is additive across cells.
as.sparse	For DelayedMatrix objects, logical indicating whether blocks should be returned as sparse matrices. Default is NA, which lets blockApply decide based on data sparsity. Set to TRUE to force sparse blocks or FALSE to force dense blocks.
BPPARAM	For DelayedMatrix objects, a BiocParallelParam object specifying the parallel backend. Defaults to getAutoBPPARAM().

Details

The deviance statistic measures how poorly each gene fits a simple null model where expression is constant across cells (after accounting for library size). Genes with high deviance show more variation than expected under the null model and are good candidates for highly variable gene selection.

For family = "binomial", the null model treats each UMI as a Bernoulli trial with gene-specific success probability. This is the closest approximation to multinomial sampling.

For family = "poisson", the null model assumes Poisson-distributed counts with rate proportional to library size. This is faster to compute and often gives similar results to binomial.

Value

A numeric vector of deviance statistics, one per row of x. Names are preserved from rownames(x) if present.

Author(s)

Patrick Aboyoun

References

Townes FW, Hicks SC, Aryee MJ, and Irizarry RA (2019). Feature Selection and Dimension Reduction for Single Cell RNA-Seq based on a Multinomial Model. *Genome Biology* https://doi.org/10.1186/s13059-019-1861-6

See Also

rowVars for variance-based feature selection.

Examples

```
# Generate example count data
set.seed(123)
mat <- matrix(rpois(1000, lambda = 5), nrow = 100, ncol = 10)
```

```
rownames(mat) <- paste0("Gene", 1:100)

# Compute deviances
dev_binom <- rowDeviances(mat, family = "binomial")
dev_pois <- rowDeviances(mat, family = "poisson")

# Select top variable genes
top_genes <- head(order(dev_binom, decreasing = TRUE), 20)
```

---

rowNnzs *Row and Column Non-Zero Counts*

Description

Count the number of non-zero elements per row or column of a matrix-like object. This is useful for filtering genes by detection rate (number of cells where the gene is expressed) or filtering cells by complexity (number of genes detected).

Usage

```
rowNnzs(x, value = vector(type(x), 1L), ...)

colNnzs(x, value = vector(type(x), 1L), ...)

## S4 method for signature 'ANY'
rowNnzs(x, value = vector(type(x), 1L), ...)

## S4 method for signature 'DuckDBArray'
rowNnzs(
  x,
  value = vector(type(x), 1L),
  na.rm = FALSE,
  dims = 1,
  ...,
  useNames = TRUE
)

## S4 method for signature 'ANY'
colNnzs(x, value = vector(type(x), 1L), ...)

## S4 method for signature 'DuckDBArray'
colNnzs(
  x,
  value = vector(type(x), 1L),
  na.rm = FALSE,
  dims = 1,
  ...,
```

```
  useNames = TRUE
)
```

Arguments

x	A matrix-like object.
value	The value to treat as "zero" (i.e., elements NOT equal to this value are counted). Defaults to the zero value for the storage mode of x.
...	Additional arguments passed to rowCounts or colCounts.
na.rm	If TRUE, missing values are excluded first.
dims	A single integer specifying the dimension(s) over which to operate. For rowNnzs, dims = 1 counts non-zeros in each row. For colNnzs, dims = 1 counts non-zeros in each column. Only used for DuckDBArray objects.
useNames	If TRUE, the names of the input are preserved in the output.

Details

These functions are convenience wrappers around rowCounts and colCounts that count elements not equal to zero. The zero value is determined by the storage mode of x:

- For logical matrices: FALSE
- For integer matrices: 0L
- For numeric matrices: 0
- For character matrices: ""

Value

An integer vector with length equal to nrow(x) for rowNnzs or ncol(x) for colNnzs.

Author(s)

Patrick Aboyoun

See Also

rowCounts for counting specific values.

Examples

```
# Dense matrix
mat <- matrix(c(0L, 1L, 2L, 0L, 0L, 3L), nrow = 2)
rowNnzs(mat)
colNnzs(mat)
```

scanKeycols *Scan a Coordinate Parquet File for keycols*

Description

Introspects a coordinate (COO) Parquet file or dataset and returns a `keycols` list, in the exact shape expected by the `keycols` argument of `DuckDBArray`/`DuckDBMatrix`: one element per key column, each holding that column's distinct values in sorted order.

Usage

```
scanKeycols(path, keycols)
```

Arguments

`path`	A single string giving the path to a Parquet file or directory of Parquet files (as accepted by `open_dataset`), or an already-open Arrow `Dataset`/`Table`.
`keycols`	A character vector of key column names to scan. Each must match a column of `path`.

Details

Given a COO Parquet file produced by another tool, the key columns' extent isn't known ahead of time the way it is for a file this package wrote itself. `scanKeycols` automates the per-column `dplyr::distinct()`/`dplyr::arrange()` scan needed to recover it, so the result can be passed directly as `keycols`.

Value

A named list with one element per name in `keycols`, each a vector of that column's distinct values in ascending order.

Author(s)

Patrick Aboyoun

See Also

`DuckDBArray` and `DuckDBMatrix`, whose `keycols` argument this function's return value is shaped for.

Examples

```
arr_df <- expand.grid(i = 1:4, j = 1:3, k = 1:2)
arr_df$value <- rpois(nrow(arr_df), lambda = 2)
arr_path <- tempfile(fileext = ".parquet")
arrow::write_parquet(arr_df, arr_path)
```

```
scanKeycols(arr_path, keycols = c("i", "j", "k"))

arr <- DuckDBArray(arr_path, datacol = "value",
                   keycols = scanKeycols(arr_path, c("i", "j", "k")))
dim(arr)
```

---

writeCoordArray *Write an Array-Like Object in Coordinate Format*

Description

An `arrow::write_dataset` wrapper function to write array-like objects in coordinate format. If dimension names are present, they are substituted for the corresponding indices.

Usage

```
writeCoordArray(
  x,
  path,
  indexcols = names(dimnames(x)) %||% sprintf("index%d", seq_along(dim(x))),
  datacol = "value",
  grid = defaultAutoGrid(COO_SparseArray(dim(x))),
  grid_suffix = "_group",
  arrowtype = NULL,
  max_dim = NULL,
  append = FALSE,
  along = NULL,
  offset = 0L,
  group_offset = 0L,
  ...
)

## S4 method for signature 'ANY'
writeCoordArray(
  x,
  path,
  indexcols = names(dimnames(x)) %||% sprintf("index%d", seq_along(dim(x))),
  datacol = "value",
  grid = defaultAutoGrid(COO_SparseArray(dim(x))),
  grid_suffix = "_group",
  arrowtype = NULL,
  max_dim = NULL,
  append = FALSE,
  along = NULL,
  offset = 0L,
  group_offset = 0L,
```

```
  BPPARAM = getAutoBPPARAM(),
  ...
)

## S4 method for signature 'DuckDBArray'
writeCoordArray(
  x,
  path,
  indexcols = names(dimnames(x)) %||% sprintf("index%d", seq_along(dim(x))),
  datacol = "value",
  grid = defaultAutoGrid(COO_SparseArray(dim(x))),
  grid_suffix = "_group",
  arrowtype = NULL,
  max_dim = NULL,
  append = FALSE,
  along = NULL,
  offset = 0L,
  group_offset = 0L,
  cluster_by = NULL,
  ...
)
```

Arguments

x	An array-like object.
path	The path to write the array-like object to.
indexcols	A character vector of column names for the dimensions of the array.
datacol	A character string specifying the column name containing the array values in the resulting table. Defaults to "value".
grid	An optional ArrayGrid to use for partitioning the array. If NULL, uses defaultAutoGrid(COO_SparseArra
grid_suffix	A character string to append to the partitioning directories if the grid contains more than one cell. Defaults to "_group".
arrowtype	An optional DataType for the value column (e.g. arrow::int32(), arrow::uint16(), arrow::float64()). When NULL and append = FALSE, the ANY method infers the type from the data: the narrowest type capable of representing the range of non-zero values if they are all integer-valued, and otherwise double. The DuckDBArray method does not materialize values to compute this range; it instead takes the value column's existing declared type from x's backing table as-is (e.g. a column declared DOUBLE writes double even if every non-zero value would fit in a narrower integer type). Pass arrowtype explicitly to narrow the on-disk type for a DuckDBArray source. When NULL and append = TRUE, the type is read from an existing parquet file under path so that the appended slab always matches the existing value-column schema. When supplied explicitly in append mode, it is verified to match the existing schema; a mismatch aborts the call before any file is written.
max_dim	Optional integer vector of length length(dim(x)) giving an upper bound on the coordinates in each dimension. Controls the arrow type chosen for each index

column: the narrowest unsigned integer type that can represent `0..max_dim[j]` is picked for column `indexcols[j]`. When `NULL` and `append = FALSE`, the `ANY` method uses `dim(x)`; the `DuckDBArray` method instead takes each index column's existing declared type from `x`'s backing table as-is, without regard to `dim(x)`. Pass `max_dim` explicitly to narrow the on-disk index types for a `DuckDBArray` source. When `NULL` and `append = TRUE`, the index-column types are read from an existing parquet file under `path` so that the appended slab always matches the existing index-column schema. When supplied explicitly in append mode, the resulting types are verified to match the existing schema; a mismatch aborts the call before any file is written. The caller does not need to know the exact final extent, only an upper bound on it; setting `max_dim` too high only costs a slightly wider integer type on disk.

`append` Logical; when `TRUE`, write additional hive-style partitions into an existing directory previously created by `writeCoordArray`. Append mode is only supported for hive-partitioned output (i.e. `length(grid) > 1L`); each cell of the supplied `grid` produces a new partition directory along dimension `along`. Before writing, `writeCoordArray` performs several corruption-prevention checks against the existing dataset:

- `path` must already exist and be laid out as a hive partitioning (top-level entries must all be `paste0(indexcols[1], grid_suffix, "=", <integer>)` subdirectories).
- At least one parquet file must exist under `path`, and its schema must contain every `indexcols[j]` and `datacol`.
- The value-column and every index-column arrow type must match the existing schema. If `arrowtype`/`max_dim` were supplied, a mismatch errors; if not, the existing types are adopted.
- No partition directory that this call would write to may already exist (otherwise `group_offset` is too small and the call would silently overwrite previously-written data).

Because arrow's dataset reader requires a uniform schema across files, these checks make silent schema drift and silent partition collisions impossible. Defaults to `FALSE`.

`along` Integer index of the dimension along which to append new partitions. Required when `append = TRUE`.

`offset` Non-negative integer added to the coordinates in dimension `along` for every non-zero written in this call. Typically the total extent already written along `along` by previous calls. Defaults to `0L`.

`group_offset` Non-negative integer added to the partition group index along dimension `along` (i.e. the value embedded in the `paste0(indexcols[along], grid_suffix)` subdirectory) so that new partition directories do not collide with existing ones. Typically the total number of partition groups already written along `along` by previous calls. Defaults to `0L`.

`...` Additional arguments to pass to `arrow::write_dataset`. Note that `existing_data_behavior` defaults to `"error"` (rather than arrow's default of `"overwrite_or_ignore"`) so that a re-run into a populated `path` fails loudly instead of silently clobbering or merging existing data. Pass `existing_data_behavior = "delete_matching"` or similar to override.

BPPARAM — A BiocParallelParam object to use for parallel processing when grid contains multiple cells (ANY method only). Defaults to getAutoBPPARAM(). Note: The DuckDBArray method does not support this parameter because DuckDB connections are not thread-safe for concurrent execution. The DuckDBArray method uses sequential execution and relies on DuckDB's internal parallelism for performance.

cluster_by — Optional within-partition row ordering for the DuckDBArray fast path: a zorder() / hilbert() spec or a character vector of index columns (e.g. one dimension's index to cluster the COO on that axis, tightening its zonemaps so a range/point query on that axis prunes row groups). Columns refer to indexcols / datacol. Default NULL keeps the existing index ordering (output byte-identical). A faithful reorder only; the COO is order-agnostic on read.

Details

Sizing grid for large or sparse arrays: when grid is NULL, defaultAutoGrid(COO_SparseArray(dim(x))) sizes partitions from dim(x) alone: a dense element-count budget (DelayedArray::getAutoBlockSize() bytes, divided by the value type's element size) that DelayedArray::getAutoBlockShape() distributes across axes ("scale" proportionally to each axis's extent; "hypercube", the default, splits every axis by the same factor). Neither setting looks at how many non-zero entries x actually has, so for a sparse x the defaults budget partitions as if x were fully dense, which can produce far more or far fewer partitions than the on-disk (non-zero-only) data volume warrants.

To size partitions off the real, sparse data volume instead, call DelayedArray::setAutoBlockSize(target_bytes / density) and DelayedArray::setAutoBlockShape("scale") before grid is computed (i.e. before this call, since grid = NULL computes it internally), where density = nnz(x) / prod(dim(x)) and target_bytes is the desired size, in bytes, of each partition's non-zero payload. Inflating the nominal (dense) budget by 1 / density means that once only the non-zero entries are actually stored, each partition lands near target_bytes. A literal, fixed setAutoBlockSize() value tuned for one dataset (as opposed to one derived from that dataset's own density) does not generalize to a dataset of different size or sparsity.

This correction has a hard ceiling: block.length (the dense element count defaultAutoGrid() budgets per partition) cannot exceed .Machine$integer.max (about 2.15 billion) regardless of setAutoBlockSize(); larger values error with "Automatic block length is too big". When prod(dim(x)) is large enough that prod(dim(x)) / .Machine$integer.max alone exceeds the number of partitions wanted, no setAutoBlockSize() / setAutoBlockShape() combination can reach it – e.g. for a 378345 x 14628532 array, that ratio is already about 2,577, so even at the legal maximum block size the default (hypercube) shape reaches only on the order of 2,600-2,844 partitions, far short of a target in the low hundreds. In that regime grid must be constructed some other way, e.g. an explicit RegularArrayGrid sized from nnz(x) directly rather than from dim(x).

Value

Called for its side effect of writing x to path as coordinate-format Parquet; returns NULL invisibly.

Author(s)

Patrick Aboyoun

Examples

```
# Write the state.x77 matrix to multiple Parquet files using grid partitioning
tf <- tempfile()
state_grid <- RegularArrayGrid(dim(state.x77), c(10, 4))
writeCoordArray(state.x77, file.path(tf, "state"), grid = state_grid)
list.files(tf, full.names = TRUE, recursive = TRUE)

# Append mode: write two column-slabs as distinct hive partitions without
# first materializing cbind(slab1, slab2). Omitting 'arrowtype' and
# 'max_dim' is safe -- both are read from the existing schema so the
# appended slab is guaranteed to match.
tf2 <- tempfile()
slab1 <- state.x77[, 1:4]
slab2 <- state.x77[, 5:8]
grid1 <- RegularArrayGrid(dim(slab1), c(10L, 2L))  # 2 col partitions
grid2 <- RegularArrayGrid(dim(slab2), c(10L, 2L))  # 2 more col partitions
writeCoordArray(slab1, file.path(tf2, "state"), grid = grid1)
writeCoordArray(slab2, file.path(tf2, "state"), grid = grid2,
                append = TRUE, along = 2L,
                offset = ncol(slab1), group_offset = dim(grid1)[2L])

# Pinning 'arrowtype' / 'max_dim' is still useful when the first slab's
# inferred schema is narrower than you want the final dataset to have
# (e.g. a small first slab that picks uint8 but you know later slabs
# will need uint16). Pin both on every call for a fully-predictable
# dataset schema.
tf3 <- tempfile()
int_slab   <- state.x77[, c("Population", "Income")]
float_slab <- state.x77[, c("Illiteracy", "Life Exp")]
grid_i <- RegularArrayGrid(dim(int_slab),   c(10L, 2L))
grid_f <- RegularArrayGrid(dim(float_slab), c(10L, 2L))
max_dim <- c(nrow(state.x77), ncol(state.x77))
writeCoordArray(int_slab, file.path(tf3, "state"), grid = grid_i,
                arrowtype = arrow::float64(), max_dim = max_dim)
writeCoordArray(float_slab, file.path(tf3, "state"), grid = grid_f,
                arrowtype = arrow::float64(), max_dim = max_dim,
                append = TRUE, along = 2L,
                offset = ncol(int_slab),
                group_offset = dim(grid_i)[2L])
```

Index

!,DuckDBArray-method
(DuckDBArray-utils), 7
!,DuckDBArraySeed-method
(DuckDBArraySeed-utils), 13
∗ **IO**
writeCoordArray, 24
∗ **classes**
DuckDBArray-class, 4
DuckDBArraySeed-class, 9
DuckDBMatrix-class, 14
∗ **methods**
DuckDBArray-class, 4
DuckDBArray-matrixStats, 6
DuckDBArray-utils, 7
DuckDBArraySeed-class, 9
DuckDBArraySeed-matrixStats, 11
DuckDBArraySeed-utils, 13
DuckDBMatrix-class, 14
DuckDBMatrix-utils, 16
DuckDBTable-matrixStats, 17
∗ **utilities**
DuckDBArray-matrixStats, 6
DuckDBArray-utils, 7
DuckDBArraySeed-matrixStats, 11
DuckDBArraySeed-utils, 13
DuckDBMatrix-utils, 16
DuckDBTable-matrixStats, 17
[,DuckDBArray,ANY,ANY,ANY-method
(DuckDBArray-class), 4
[,DuckDBArraySeed,ANY,ANY,ANY-method
(DuckDBArraySeed-class), 9
[,DuckDBMatrix,ANY,ANY,ANY-method
(DuckDBMatrix-class), 14
%*%,DuckDBMatrix,numeric-method
(DuckDBMatrix-utils), 16
%*%,numeric,DuckDBMatrix-method
(DuckDBMatrix-utils), 16

aperm,DuckDBArray-method
(DuckDBArray-class), 4
aperm,DuckDBArraySeed-method
(DuckDBArraySeed-class), 9
Array, *11*, *12*, *14*
ArrayGrid, *3*, *20*, *25*

BiocParallelParam, *3*, *20*, *27*
blockApply, *20*

coerce,DuckDBArray,COO_SparseArray-method
(DuckDBArray-class), 4
coerce,DuckDBArray,COO_SparseMatrix-method
(DuckDBArray-class), 4
coerce,DuckDBArray,dgCMatrix-method
(DuckDBArray-class), 4
colAutoGrid, *20*
colCounts, *22*
colCounts,DuckDBArray-method
(DuckDBArray-matrixStats), 6
colCounts,DuckDBArraySeed-method
(DuckDBArraySeed-matrixStats), 11
colCounts,DuckDBTable-method
(DuckDBTable-matrixStats), 17
colMaxs,DuckDBArray-method
(DuckDBArray-matrixStats), 6
colMaxs,DuckDBArraySeed-method
(DuckDBArraySeed-matrixStats), 11
colMaxs,DuckDBTable-method
(DuckDBTable-matrixStats), 17
colMeans,DuckDBArray-method
(DuckDBArray-matrixStats), 6
colMeans,DuckDBArraySeed-method
(DuckDBArraySeed-matrixStats), 11
colMeans,DuckDBTable-method
(DuckDBTable-matrixStats), 17
colMins,DuckDBArray-method
(DuckDBArray-matrixStats), 6

colMins,DuckDBArraySeed-method
(DuckDBArraySeed-matrixStats), 11
colMins,DuckDBTable-method
(DuckDBTable-matrixStats), 17
colNnzs (rowNnzs), 21
colNnzs,ANY-method (rowNnzs), 21
colNnzs,DuckDBArray-method (rowNnzs), 21
colSds,DuckDBArray-method
(DuckDBArray-matrixStats), 6
colSds,DuckDBArraySeed-method
(DuckDBArraySeed-matrixStats), 11
colSds,DuckDBTable-method
(DuckDBTable-matrixStats), 17
colSums,DuckDBArray-method
(DuckDBArray-matrixStats), 6
colSums,DuckDBArraySeed-method
(DuckDBArraySeed-matrixStats), 11
colSums,DuckDBTable-method
(DuckDBTable-matrixStats), 17
colVars,DuckDBArray-method
(DuckDBArray-matrixStats), 6
colVars,DuckDBArraySeed-method
(DuckDBArraySeed-matrixStats), 11
colVars,DuckDBTable-method
(DuckDBTable-matrixStats), 17
createDimTables, 2
crossprod,DuckDBMatrix,missing-method
(DuckDBMatrix-utils), 16
crossprod,DuckDBMatrix,numeric-method
(DuckDBMatrix-utils), 16
crossprod,numeric,DuckDBMatrix-method
(DuckDBMatrix-utils), 16

DataType, *25*
dbconn,DuckDBArray-method
(DuckDBArray-class), 4
dbconn,DuckDBArraySeed-method
(DuckDBArraySeed-class), 9
DelayedArray, *4*, *5*, *7*, *9*
DelayedArray,DuckDBArraySeed-method
(DuckDBArraySeed-class), 9
DelayedMatrix, *14*, *16*
dim,DuckDBArraySeed-method
(DuckDBArraySeed-class), 9
dimnames,DuckDBArraySeed-method
(DuckDBArraySeed-class), 9
dimnames<-,DuckDBArray,ANY-method
(DuckDBArray-class), 4
dimnames<-,DuckDBArraySeed,ANY-method
(DuckDBArraySeed-class), 9
dimtbls,DuckDBArray-method
(DuckDBArray-class), 4
dimtbls,DuckDBArraySeed-method
(DuckDBArraySeed-class), 9
dimtbls<-,DuckDBArray-method
(DuckDBArray-class), 4
dimtbls<-,DuckDBArraySeed-method
(DuckDBArraySeed-class), 9
DuckDBArray, *6*, *7*, *9*, *10*, *23*
DuckDBArray (DuckDBArray-class), 4
DuckDBArray-class, 4
DuckDBArray-matrixStats, 6
DuckDBArray-utils, 7
DuckDBArraySeed, *11*, *13*
DuckDBArraySeed
(DuckDBArraySeed-class), 9
DuckDBArraySeed-class, 9
DuckDBArraySeed-matrixStats, 11
DuckDBArraySeed-utils, 13
DuckDBMatrix, *10*, *16*, *23*
DuckDBMatrix (DuckDBMatrix-class), 14
DuckDBMatrix-class, 14
DuckDBMatrix-utils, 16
DuckDBTable, *18*
DuckDBTable-class, *17*
DuckDBTable-matrixStats, 17

extract_array,DuckDBArraySeed-method
(DuckDBArraySeed-class), 9
extract_sparse_array,DuckDBArraySeed-method
(DuckDBArraySeed-class), 9

getAutoBPPARAM, *20*

is.finite,DuckDBArray-method
(DuckDBArray-utils), 7
is.finite,DuckDBArraySeed-method
(DuckDBArraySeed-utils), 13
is.infinite,DuckDBArray-method
(DuckDBArray-utils), 7
is.infinite,DuckDBArraySeed-method
(DuckDBArraySeed-utils), 13

is.nan,DuckDBArray-method
(DuckDBArray-utils), 7
is.nan,DuckDBArraySeed-method
(DuckDBArraySeed-utils), 13
is_nonzero,DuckDBArray-method
(DuckDBArray-utils), 7
is_nonzero,DuckDBArraySeed-method
(DuckDBArraySeed-utils), 13
is_sparse,DuckDBArraySeed-method
(DuckDBArraySeed-utils), 13

Math,DuckDBArray-method
(DuckDBArray-utils), 7
Math,DuckDBArraySeed-method
(DuckDBArraySeed-utils), 13
matrixClass,DuckDBArray-method
(DuckDBMatrix-class), 14

nzcount,DuckDBArray-method
(DuckDBArray-utils), 7
nzcount,DuckDBArraySeed-method
(DuckDBArraySeed-utils), 13
nzvals,DuckDBArray-method
(DuckDBArray-utils), 7

open_dataset, *23*
Ops,atomic,DuckDBArray-method
(DuckDBArray-utils), 7
Ops,atomic,DuckDBArraySeed-method
(DuckDBArraySeed-utils), 13
Ops,DuckDBArray,atomic-method
(DuckDBArray-utils), 7
Ops,DuckDBArray,DuckDBArray-method
(DuckDBArray-utils), 7
Ops,DuckDBArray,missing-method
(DuckDBArray-utils), 7
Ops,DuckDBArraySeed,atomic-method
(DuckDBArraySeed-utils), 13
Ops,DuckDBArraySeed,DuckDBArraySeed-method
(DuckDBArraySeed-utils), 13
Ops,DuckDBArraySeed,missing-method
(DuckDBArraySeed-utils), 13

RectangularData, *18*
RegularArrayGrid, *27*
rowCounts, *22*
rowCounts,DuckDBArray-method
(DuckDBArray-matrixStats), 6
rowCounts,DuckDBArraySeed-method
(DuckDBArraySeed-matrixStats), 11
rowCounts,DuckDBTable-method
(DuckDBTable-matrixStats), 17
rowDeviances, 19
rowDeviances,DelayedMatrix-method
(rowDeviances), 19
rowDeviances,dgCMatrix-method
(rowDeviances), 19
rowDeviances,DuckDBMatrix-method
(rowDeviances), 19
rowDeviances,matrix-method
(rowDeviances), 19
rowMaxs,DuckDBArray-method
(DuckDBArray-matrixStats), 6
rowMaxs,DuckDBArraySeed-method
(DuckDBArraySeed-matrixStats), 11
rowMaxs,DuckDBTable-method
(DuckDBTable-matrixStats), 17
rowMeans,DuckDBArray-method
(DuckDBArray-matrixStats), 6
rowMeans,DuckDBArraySeed-method
(DuckDBArraySeed-matrixStats), 11
rowMeans,DuckDBTable-method
(DuckDBTable-matrixStats), 17
rowMins,DuckDBArray-method
(DuckDBArray-matrixStats), 6
rowMins,DuckDBArraySeed-method
(DuckDBArraySeed-matrixStats), 11
rowMins,DuckDBTable-method
(DuckDBTable-matrixStats), 17
rowNnzs, 21
rowNnzs,ANY-method (rowNnzs), 21
rowNnzs,DuckDBArray-method (rowNnzs), 21
rowSds,DuckDBArray-method
(DuckDBArray-matrixStats), 6
rowSds,DuckDBArraySeed-method
(DuckDBArraySeed-matrixStats), 11
rowSds,DuckDBTable-method
(DuckDBTable-matrixStats), 17
rowSums,DuckDBArray-method
(DuckDBArray-matrixStats), 6
rowSums,DuckDBArraySeed-method

(DuckDBArraySeed-matrixStats), 11
rowSums,DuckDBTable-method (DuckDBTable-matrixStats), 17
rowVars, *20*
rowVars,DuckDBArray-method (DuckDBArray-matrixStats), 6
rowVars,DuckDBArraySeed-method (DuckDBArraySeed-matrixStats), 11
rowVars,DuckDBTable-method (DuckDBTable-matrixStats), 17

S4groupGeneric, *8*, *13*
scanKeycols, 23
show,DuckDBArraySeed-method (DuckDBArraySeed-class), 9
sparseMatrix, *5*, *10*
sweep,DuckDBArray-method (DuckDBArray-utils), 7
sweep,DuckDBArraySeed-method (DuckDBArraySeed-utils), 13
sweep,DuckDBTable-method (DuckDBTable-matrixStats), 17

t,DuckDBArray-method (DuckDBArray-class), 4
t,DuckDBArraySeed-method (DuckDBArraySeed-class), 9
tblconn,DuckDBArray-method (DuckDBArray-class), 4
tblconn,DuckDBArraySeed-method (DuckDBArraySeed-class), 9
tcrossprod,DuckDBMatrix,missing-method (DuckDBMatrix-utils), 16
tcrossprod,DuckDBMatrix,numeric-method (DuckDBMatrix-utils), 16
tcrossprod,numeric,DuckDBMatrix-method (DuckDBMatrix-utils), 16
type,DuckDBArray-method (DuckDBArray-class), 4
type,DuckDBArraySeed-method (DuckDBArraySeed-class), 9
type<-,DuckDBArray-method (DuckDBArray-class), 4
type<-,DuckDBArraySeed-method (DuckDBArraySeed-class), 9

writeCoordArray, *3*, 24
writeCoordArray,ANY-method (writeCoordArray), 24
writeCoordArray,DuckDBArray-method (writeCoordArray), 24

zorder, *27*