Package 'BiocDuckDB'

September 14, 2026

Version 0.99.24

Date 2026-09-14

Title Bioconductor DuckDB Integration and High-Level I/O

Description Integration package providing high-level Parquet I/O functions and optimized methods for single-cell analysis workflows using DuckDB-backed data structures. Includes readParquet and writeParquet functions for seamless serialization of SummarizedExperiment, SingleCellExperiment, MultiAssayExperiment, and MultiAssaySpatialExperiment objects, plus SQL-optimized implementations of scran and scuttle methods for variance modeling, marker detection, QC metrics, and normalization. This package brings together DuckDBDataFrame, DuckDBArray, DuckDBGRanges, and DuckDBSpatial (optional) for complete Bioconductor integration.

License MIT + file LICENSE

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

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

Copyright See the file inst/COPYRIGHTS for third-party schema components bundled in inst/schema/ (Frictionless Data Package specifications, in the public domain under The Unlicense).

Depends R (>= 4.6.0), methods, stats, DuckDBDataFrame, DuckDBArray, DuckDBGRanges

Imports BiocGenerics, BiocParallel, Matrix, S4Vectors, IRanges, GenomicRanges, SparseArray, DelayedArray, MatrixGenerics, SummarizedExperiment, SingleCellExperiment, MultiAssayExperiment, MultiAssaySpatialExperiment, metapod, beachmat, BiocSingular, scran, scuttle, DBI, dplyr, dbplyr, duckdb, arrow, jsonlite

Suggests knitr, rmarkdown, BiocStyle, testthat, jsonvalidate, grDevices, matrixStats, nanoparquet, png, sf, sfarrow, S4Arrays, SpatialExperiment, DuckDBSpatial, ExperimentHub, HDF5Array, ZarrArray, irlba, scater, airway, scRNAseq

biocViews DataImport, DataRepresentation, Infrastructure, RNASeq, Sequencing, SingleCell, Software

VignetteBuilder knitr

Encoding UTF-8

Collate 'BiocDuckDB-package.R'
'MultiAssaySpatialExperiments-internals.R'
'DuckDBDataFrame-spatial.R' 'DuckDBDualSubset-class.R'
'initializeOptions.R' 'initializeCpp.R' 'DuckDBIrlbaParam.R'
'DuckDBMatrix-scran.R' 'DuckDBMatrix-scuttle.R'
'MultiAssaySpatialExperiment-query.R'
'MultiAssaySpatialExperiment-spatial.R'
'SingleCellExperiments-internals.R'
'SingleCellExperiment-colTables.R'
'SingleCellExperiment-loadings.R'
'SingleCellExperiment-pairs.R'
'SingleCellExperiment-rowTables.R' 'fieldtypes.R'
'loadIntoMemory.R' 'readParquet.R' 'writeDatapackage.R'
'writeParquet.R' 'writeStreamingResource.R'

Config/roxygen2/version 8.1.0

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

git_branch devel

git_last_commit 2c1b267

git_last_commit_date 2026-09-14

Repository Bioconductor 3.24

Date/Publication 2026-09-14

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

Maintainer Patrick Aboyoun <aboyounp@gene.com>

Contents

`BiocDuckDB-package` *BiocDuckDB: Bioconductor DuckDB Integration and High-Level I/O*

Description

Integration package providing high-level Parquet I/O functions and optimized methods for single-cell analysis workflows using DuckDB-backed data structures. Includes readParquet and writeParquet functions for seamless serialization of SummarizedExperiment, SingleCellExperiment, MultiAssayExperiment, and MultiAssaySpatialExperiment objects, plus SQL-optimized implementations of scran and scuttle methods for variance modeling, marker detection, QC metrics, and normalization. This package brings together DuckDBDataFrame, DuckDBArray, DuckDBGRanges, and DuckDBSpatial (optional) for complete Bioconductor integration.

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

---

`DuckDBDataFrame-spatial`
DuckDBDataFrame spatial methods

Description

DuckDBDataFrame methods for **MultiAssaySpatialExperiment** spatial generics. Lazy layers delegate to **DuckDBSpatial** for SQL-backed spatial operations.

Value

`spatialOverlaps()` returns the pairwise spatial-overlap result between the lazy layers, `spatialMatch()` the matched table, and `spatialJoin()` a sparse pair table (mapping row indices between layers) when `sparse = TRUE`, or the joined table otherwise. All are computed lazily over the DuckDB-backed layers.

Methods

`spatialOverlaps(x, y, coords = NULL, geom = "geometry")`: Test spatial overlap between lazy spatial layers via `layerSpatialOverlaps`.

`spatialMatch(x, table, coords, geom = "geometry", join = NULL)`: Match points or geometries to a table via `layerSpatialMatch`.

`spatialJoin(x, y, join = st_intersects, sparse = TRUE)`: Spatial join between lazy layers. When `sparse = TRUE`, returns a sparse pair table via `st_intersects_table`; otherwise uses `st_join`.

See Also

`spatialOverlaps`, `spatialMatch`, and `spatialJoin` for generic documentation.

Examples

```
# Spatial predicates need DuckDB's optional spatial extension. Probe for it so
# the example runs only where the extension can actually load; it is skipped
# cleanly when offline or behind a firewall that blocks the extension
# repository, or when DuckDBSpatial is not installed.
spatial_ok <- requireNamespace("DuckDBSpatial", quietly = TRUE) &&
    tryCatch({
        DBI::dbGetQuery(DuckDBDataFrame::acquireDuckDBConn(),
            "SELECT ST_Area(ST_GeomFromText('POLYGON((0 0,0 1,1 1,1 0,0 0))'))")
        TRUE
    }, error = function(e) FALSE)
if (spatial_ok) {
    df <- data.frame(id = 1:3, x = 1:3, y = 1:3)
    path <- tempfile(fileext = ".parquet")
    arrow::write_parquet(df, path)
    ddb <- DuckDBDataFrame(path, datacols = c("x", "y"), keycol = "id")
    polygon <- "POLYGON((0 0, 6 0, 6 6, 0 6, 0 0))"
    spatialOverlaps(ddb, polygon, coords = c("x", "y"))
    unlink(path)
}
```

DuckDBDualSubset-class

DuckDBDualSubset objects

Description

The DuckDBDualSubset class conforms to the DualSubset class from SingleCellExperiment to provide lazy node-based subsetting for DuckDBSelfHits objects.

Details

DuckDBDualSubset is a lightweight wrapper around DuckDBSelfHits that provides node-based subsetting semantics required by SingleCellExperiment's `colPairs`/`rowPairs` framework. Unlike the `DualSubset` class which materializes data immediately upon subsetting, `DuckDBDualSubset` delegates to the `nodes` slot in the wrapped `DuckDBSelfHits` object, enabling lazy SQL-based edge filtering.

When a DuckDBDualSubset is subset via x[i], it delegates to extractNODES(x@hits, i), which updates the nodes slot in the DuckDBSelfHits object. Edge filtering is deferred until materialization via SQL WHERE clauses (only edges where BOTH from and to are in the node subset are retained).

Value

The DuckDBDualSubset() constructor returns a DuckDBDualSubset object wrapping a DuckDBSelfHits. length() returns the number of nodes (an integer); subsetting ([) returns a DuckDBDualSubset, and the [<- and c() methods return the updated DuckDBDualSubset.

Constructor

DuckDBDualSubset(hits): Creates a DuckDBDualSubset object wrapping a DuckDBSelfHits object.

hits A DuckDBSelfHits object containing the edge data.

Accessors

length(x): Returns the number of nodes (nnode(x@hits)).

Subsetting

x[i]: Performs node-based subsetting by delegating to extractNODES(x@hits, i). This updates the nodes slot in the wrapped DuckDBSelfHits object and applies lazy SQL filtering to retain only edges where BOTH endpoints are in the subset.

Usage with SingleCellExperiment

DuckDBDualSubset objects are typically created automatically when assigning DuckDBSelfHits objects to colPairs/rowPairs:

Author(s)

Patrick Aboyoun

See Also

DuckDBSelfHits for the underlying edge storage class.

extractNODES for the node-based subsetting method.

Examples

```
library(SingleCellExperiment)
sce <- SingleCellExperiment(assays = list(counts = matrix(1:50, 10, 5)))
hits <- S4Vectors::SelfHits(from = 1:3, to = 2:4, nnode = 5L)
tmp <- tempfile()
writeParquet(hits, tmp)
ddb_hits <- DuckDBSelfHits(tmp, from = "from", to = "to", nnode = 5L)
colPair(sce, "knn") <- ddb_hits
ds <- colPair(sce, "knn")
length(ds)
ds[1:3]
unlink(tmp, recursive = TRUE)
```

DuckDBIrlbaParam *A fast SVD path for DuckDB-backed matrices*

Description

DuckDBIrlbaParam is a BiocSingularParam for runSVD that, for a DuckDBMatrix, materializes the matrix once into an in-memory sparse matrix and drives **irlba** directly on it, instead of letting **irlba**'s Lanczos iteration call %*% on the lazy DuckDBMatrix once per solver step.

Usage

```
DuckDBIrlbaParam(
  memory_limit = initializeOptions("memory_limit"),
  deferred = FALSE,
  fold = Inf,
  extra.work = 7,
  ...
)

## S4 method for signature 'DuckDBIrlbaParam'
runSVD(
  x,
  k,
  nu = k,
  nv = k,
  center = FALSE,
  scale = FALSE,
  BPPARAM = SerialParam(),
  ...,
  BSPARAM = ExactParam()
)
```

Arguments

memory_limit	Numeric scalar, see the memory_limit option in initializeOptions. Materialization is refused above this estimated size (bytes), falling back to the ordinary lazy path.
deferred, fold, extra.work, ...	Passed to IrlbaParam.
x	A matrix-like object, typically a DuckDBMatrix (the fast path only applies to one; any other input falls back to ordinary IrlbaParam behavior).
k, nu, nv, center, scale, BPPARAM, BSPARAM	See runSVD.

Details

DuckDBMatrix already implements %*%/ crossprod as correct SQL-pushdown methods (see **DuckDBArray**), so BiocSingular::runSVD(x, BSPARAM = BiocSingular::IrlbaParam()) already computes a correct PCA/SVD directly on a DuckDBMatrix. But each of those %*% calls pays real SQL query-construction cost, and **irlba**'s Lanczos loop calls it once per solver iteration, hundreds of

times per solve, so the ordinary IrlbaParam path is markedly slower on a DuckDBMatrix than the same computation in memory (measured: on a real 12,500-cell x 200-HVG benchmark, ~16s vs ~0.3s in-memory).

DuckDBIrlbaParam closes that gap for matrices that fit in memory, by reusing the same loadIntoMemory machinery (.duckdb_seed_to_sparse_matrix(), size-gated by "memory_limit") to materialize the matrix exactly once, then calling **BiocSingular**'s own exported irlba-driving function (runIrlbaSVD) directly on the materialized matrix. Reusing that function, rather than reimplementing its nu/nv trimming, edge cases, and output packaging, avoids a larger and riskier undertaking than depending on one stable, public function.

Whenever the fast path is not applicable (x is not a DuckDBMatrix, its seed is not zero-filled, or the estimated materialized size is over "memory_limit"), this falls back to ordinary IrlbaParam behavior (the lazy SQL-pushdown %*% path), so it can never turn previously-*working* (if slower) code into a failure. It can still fail the same way the ordinary path already does, e.g. a non-zero-filled seed is not supported by %*% either, that case falls back to, and fails identically to, the ordinary path rather than silently producing an incorrect result.

Value

A DuckDBIrlbaParam object, for use as the BSPARAM argument to runSVD (and so scran::fixedPCA(), scater::runPCA()/calculatePCA()).

Author(s)

Patrick Aboyoun

See Also

- IrlbaParam for the ordinary, lazy-pushdown behavior
- loadIntoMemory for the materialization helper this reuses
- initializeOptions for the shared "memory_limit" default

Examples

```
df <- data.frame(row = c("r1", "r2", "r1"), col = c("c1", "c1", "c2"),
                 value = c(1, 2, 3))
tf <- tempfile(fileext = ".parquet")
on.exit(unlink(tf))
arrow::write_parquet(df, tf)
pqmat <- DuckDBArray::DuckDBMatrix(tf, datacol = "value",
                                   keycols = list(row = c("r1", "r2"), col = c("c1", "c2")))
BiocSingular::runSVD(pqmat, k = 1, BSPARAM = DuckDBIrlbaParam())
```

DuckDBMatrix-scran *DuckDBMatrix scran methods*

Description

scran methods for DuckDBMatrix objects.

Details

The DuckDBMatrix methods compute per-gene statistics using SQL aggregation, which is efficient for large disk-backed matrices:

- For `pairwiseTTests`: per-group means and variances via SQL
- For `pairwiseBinom`: per-group detection counts via SQL

The trend fitting and p-value computations are then performed in R using the standard scran functions.

For `findMarkers`, both `test.type="t"` and `test.type="binom"` use optimized DuckDBMatrix implementations. The Wilcoxon test (`test.type="wilcox"`) falls back to the default method because it requires pairwise cell comparisons that are not efficiently expressed in SQL.

Value

These methods mirror their **scran** generics. `correlatePairs()` returns a `DataFrame` of gene pairs with columns `gene1`, `gene2`, `rho` (Pearson's r), `p.value`, and `FDR`. `modelGeneVar()`, `modelGeneVarByPoisson()`, and `modelGeneCV2()` return a `DataFrame` of per-gene variance statistics, with the fitted mean-variance trend in `metadata()`. `pairwiseTTests()` and `pairwiseBinom()` return a list with `statistics` and `pairs` elements; `findMarkers()` and `scoreMarkers()` return a List of per-group `DataFrames` of marker statistics; and `summaryMarkerStats()` returns a `DataFrame` summarizing marker statistics per group.

Pairwise Correlation Methods

The following pairwise correlation methods have optimized DuckDBMatrix implementations:

`correlatePairs(x, subset.row = NULL, pairings = NULL, use.names = TRUE, BPPARAM = SerialParam())`:
Compute pairwise Pearson correlations between genes using sparse-aware SQL aggregation, with a standard t-test significance for each correlation (`p.value`, BH-adjusted as `FDR`). Only `fill = 0` is supported. This is Pearson's r on `x` as given (typically log-normalized values), not **scran**'s Spearman's rho on ranked expression – results are not expected to numerically match `scran::correlatePairs()`.

`subset.row` rows (genes) to correlate; required for >1000 genes

`pairings` optional matrix specifying specific gene pairs

`use.names` whether to use row names in output

Variance Modelling Methods

The following variance modelling methods have optimized DuckDBMatrix implementations:

`modelGeneVar(x, block = NULL, design = NULL, subset.row = NULL, subset.fit = NULL, ..., equiweight = TRU`
Model the variance of log-expression profiles for each gene, decomposing it into technical and biological components. Uses SQL aggregation for computing per-gene means and variances.

`block` factor specifying blocking levels for each cell

`design` design matrix for blocking (not supported with block)

`subset.row` rows for which to model the variance

`subset.fit` rows to be used for trend fitting

`equiweight` whether blocks should be weighted equally

`method` method for combining p-values across blocks

`modelGeneVarByPoisson(x, size.factors = NULL, block = NULL, design = NULL, subset.row = NULL, ..., equi`
Model the variance of log-expression profiles for each gene, using a Poisson-based trend to estimate technical variance. Uses SQL aggregation for computing per-gene means and variances.

`size.factors` numeric vector of size factors for normalization

`block` factor specifying blocking levels for each cell

`design` design matrix for blocking (not supported with block)

`subset.row` rows for which to model the variance

`equiweight` whether blocks should be weighted equally

`method` method for combining p-values across blocks

`modelGeneCV2(x, size.factors = NULL, block = NULL, subset.row = NULL, subset.fit = NULL, ..., equiweigh`
Model the squared coefficient of variation (CV2) of normalized expression for each gene, fitting a trend to account for the mean-CV2 relationship. Uses SQL aggregation for computing per-gene means and variances on size-factor normalized counts.

`size.factors` numeric vector of size factors for normalization

`block` factor specifying blocking levels for each cell

`subset.row` rows for which to model the CV2

`subset.fit` rows to be used for trend fitting

`equiweight` whether blocks should be weighted equally

`method` method for combining p-values across blocks

Marker Gene Detection Methods

The following marker gene detection methods have optimized DuckDBMatrix implementations:

`pairwiseTTests(x, groups, block = NULL, design = NULL, restrict = NULL, exclude = NULL, direction = c("a`
Perform pairwise Welch t-tests between groups of cells. Uses SQL aggregation for computing per-gene, per-group means and variances.

`groups` vector specifying group assignment for each cell

`block` factor specifying blocking levels

`restrict` character vector of groups to restrict comparisons to

`exclude` character vector of groups to exclude from comparisons

`direction` direction of log-fold changes to consider

`lfc` log-fold change threshold

`std.lfc` whether to standardize log-fold changes

`log.p` whether to return log-transformed p-values

`gene.names` character vector of gene names for output

`pairwiseBinom(x, groups, block = NULL, restrict = NULL, exclude = NULL, direction = c("any", "up", "dow`
Perform pairwise binomial tests between groups of cells. Uses SQL aggregation for computing per-gene, per-group detection counts.

`groups` vector specifying group assignment for each cell

`block` factor specifying blocking levels

`restrict` character vector of groups to restrict comparisons to

`exclude` character vector of groups to exclude from comparisons

`direction` direction of log-fold changes to consider

`threshold` expression threshold for detection

`lfc` log-fold change threshold for proportions

`log.p` whether to return log-transformed p-values

`gene.names` character vector of gene names for output

`findMarkers(x, groups, test.type = c("t", "wilcox", "binom"), ..., pval.type = c("any", "some", "all")`
Find candidate marker genes for groups of cells.

`groups` vector specifying group assignment for each cell

`test.type` type of pairwise test ("t" and "binom" are optimized)

`pval.type` how to combine p-values across pairwise tests

`sorted` whether to sort results by significance

`add.summary` whether to add summary statistics

`scoreMarkers(x, groups, block = NULL, pairings = NULL, lfc = 0, row.data = NULL, full.stats = FALSE, subs`
Compute effect size summary statistics for potential marker genes. Uses SQL aggregation for computing per-gene, per-group means, variances, and detection proportions.

AUC computation options: By default (`true.auc = FALSE`), AUC is computed using a normal approximation which is fast (~97% correlation with scran). Set `true.auc = TRUE` to compute true rank-based AUC using SQL window functions (Wilcoxon rank-sum statistic). This is slower (~15-25x) but provides exact AUC values identical to scran (100% correlation).

`groups` vector specifying group assignment for each cell

`block` factor specifying blocking levels

`pairings` specification of pairwise comparisons to compute

`lfc` log-fold change threshold for effect sizes

`row.data` additional row data to include in output

`full.stats` whether to return full pairwise statistics

`true.auc` logical indicating whether to compute true rank-based AUC via SQL window functions (default FALSE uses normal approximation with ~97% correlation; TRUE gives 100% exact match but is ~15-25x slower)

`summaryMarkerStats(x, groups, row.data = NULL, average = "mean", BPPARAM = SerialParam())`:
Compute summary statistics for marker gene selection. Uses SQL aggregation for per-group means and detection proportions.

`groups` vector specifying group assignment for each cell

`row.data` additional row data to include in output

`average` type of average to compute ("mean" or "median")

AUC Computation in scoreMarkers

The `scoreMarkers` method for DuckDBMatrix provides two AUC computation modes:

Normal approximation (default, `true.auc = FALSE`):

$$AUC \approx \Phi\left(\frac{\mu_1 - \mu_2}{\sqrt{\sigma_1^2/n_1 + \sigma_2^2/n_2}}\right)$$

where Φ is the standard normal CDF. This is fast but may be inaccurate for sparse/bimodal data.

True rank-based AUC (`true.auc = TRUE`): Computes exact AUC via the Wilcoxon rank-sum statistic using SQL window functions:

$$AUC = \frac{R_1 - n_1(n_1+1)/2}{n_1 \cdot n_2}$$

where R_1 is the sum of ranks in group 1. This is ~15-25x slower than the normal approximation but provides exact results identical to scran (correlation = 1.0).

Benchmarks (5000 genes x 5000 cells, 12 pairwise comparisons):

- Default (normal approx): 0.68s, ~97% correlation with scran
- `true.auc = TRUE`: 17.7s, 100% exact match with scran

The true AUC is preferred for:

- Genes with bimodal expression (expressed vs. not expressed)
- Sparse data with many zeros
- Applications where exact effect sizes are important

Alternatively, use `findMarkers(x, groups, test.type = "wilcox")` for rank-based p-values.

Author(s)

Patrick Aboyoun

See Also

- `DuckDBMatrix-class` for the main class
- `DuckDBMatrix-scuttle` for the scuttle methods
- `correlatePairs` for the scran generic
- `modelGeneVar` for the scran generic
- `modelGeneVarByPoisson` for the scran generic
- `modelGeneCV2` for the scran generic
- `pairwiseTTests` for the scran generic
- `pairwiseBinom` for the scran generic
- `findMarkers` for the scran generic
- `scoreMarkers` for the scran generic
- `summaryMarkerStats` for the scran generic

Examples

```
mat <- matrix(rpois(500, 5), 25, 20)
dimnames(mat) <- list(paste0("G", 1:25), paste0("C", 1:20))
names(dimnames(mat)) <- c("index1", "index2")
tmp <- tempfile()
writeParquet(mat, tmp)
pqmat <- DuckDBMatrix(tmp, datacol = "value",
    keycols = lapply(dimnames(mat), function(x) setNames(seq_along(x), x)))
head(scran::modelGeneVarByPoisson(pqmat))
unlink(tmp, recursive = TRUE)
```

DuckDBMatrix-scuttle *DuckDBMatrix scuttle methods*

Description

scuttle methods for DuckDBMatrix objects.

Value

These methods mirror their **scuttle** generics. `perCellQCMetrics()` and `perFeatureQCMetrics()` return a `DataFrame` of QC metrics. `librarySizeFactors()`, `geometricSizeFactors()`, and `calculateAverage()` return a numeric vector. `normalizeCounts()`, `calculateCPM()`, and `calculateTPM()` return a matrix-like object of transformed values, and `numDetectedAcrossFeatures()` and `sumCountsAcrossFeatures` a matrix-like object of per-group summaries. `summarizeAssayByGroup()` returns a SummarizedExperiment of per-group assay summaries.

QC Metrics Methods

The following QC metrics methods have optimized DuckDBMatrix implementations:

`perCellQCMetrics(x, subsets = NULL, percent.top = integer(0), threshold = 0, ..., flatten = TRUE)`**:** Compute per-cell quality control metrics for a count matrix.

- `subsets` named list of vectors specifying feature subsets (e.g., mitochondrial genes). Each vector can be character (feature names), logical, or integer (indices). Returns sum, detected, and percent for each subset.
- `percent.top` integer vector specifying the sizes of the top sets of high-abundance genes (e.g., `c(50, 100, 200)`). For each size, computes the percentage of total counts assigned to the most highly expressed genes in each cell. Uses SQL window functions for efficient computation.
- `threshold` numeric scalar specifying the detection threshold
- `flatten` logical scalar indicating whether nested DataFrames in the output should be flattened. If `TRUE` (default), subset and percent.top statistics are returned as top-level columns (e.g., `subsets_Mito_sum`, `percent.top_50`). If `FALSE`, nested DataFrames/matrices are returned.

`perFeatureQCMetrics(x, subsets = NULL, threshold = 0, ..., flatten = TRUE)`**:** Compute per-feature quality control metrics for a count matrix.

- `subsets` named list of vectors specifying cell subsets (e.g., control wells). Each vector can be character (cell names), logical, or integer (indices). Returns mean, detected, and ratio for each subset.
- `threshold` numeric scalar specifying the detection threshold
- `flatten` logical scalar indicating whether nested DataFrames in the output should be flattened. If `TRUE` (default), subset statistics are returned as top-level columns (e.g., `subsets_SetA_mean`). If `FALSE`, a nested `subsets` DataFrame is returned.

Normalization Methods

The following normalization methods have optimized DuckDBMatrix implementations:

`librarySizeFactors(x)`**:** Define per-cell size factors from the library sizes.

`geometricSizeFactors(x, pseudo.count = 1)`: Define per-cell size factors from the geometric mean of counts per cell.

`pseudo.count` numeric scalar specifying the pseudo-count to add during log-transformation

`normalizeCounts(x, size.factors = NULL, log = TRUE, transform = c("log", "none", "asinh"), pseudo.coun`
Normalizes counts by dividing by size factors.

`size.factors` numeric vector of cell-specific size factors

`log` logical scalar indicating whether normalized values should be log2-transformed

`transform` string specifying the transformation to apply to the normalized expression values.

`pseudo.count` numeric scalar specifying the pseudo-count to add when `transform = "log"`

`center.size.factors` logical scalar indicating whether size factors should be centered at unity before being used

`subset.row` vector specifying the subset of rows of x for which to return normalized values

`calculateTPM(x, lengths = NULL, ...)`: Calculates transcripts per million using SQL-optimized row-wise sweep.

`lengths` Numeric vector of gene lengths.

`calculateCPM(x, size.factors = NULL, subset.row = NULL, ...)`: Calculates counts per million using DelayedArray-safe sweep operations.

`size.factors` numeric vector containing size factors to adjust the library sizes. If NULL, library sizes are used directly.

`subset.row` vector specifying the subset of rows of x for which to return normalized values

`calculateAverage(x, size.factors = NULL, subset.row = NULL, BPPARAM = SerialParam(), ...)`:
Calculates average counts per feature using DelayedArray-safe sweep operations.

`size.factors` numeric vector containing size factors. If NULL, library size factors are computed automatically.

`subset.row` vector specifying the subset of rows of x for which to return averages

`BPPARAM` BiocParallelParam object for parallel computation

The following normalization methods will dispatch to optimized DuckDBMatrix implementations of `normalizeCounts`, `librarySizeFactors`, `calculateCPM`, `calculateAverage`, and `calculateTPM`.

Aggregation Methods

The following aggregation methods have optimized DuckDBMatrix implementations that use SQL GROUP BY operations for efficient pseudo-bulk analysis:

`numDetectedAcrossFeatures(x, ids, subset.row = NULL, subset.col = NULL, average = FALSE, threshold = 0`
Count detected features across feature sets for each cell.

`ids` factor or list specifying feature groupings

`average` logical indicating whether to compute the proportion

`threshold` threshold for detection

`sumCountsAcrossFeatures(x, ids, subset.row = NULL, subset.col = NULL, average = FALSE, ...)`:
Sum counts across feature sets for each cell.

`ids` factor or list specifying feature groupings

`average` logical indicating whether to compute the average

`summarizeAssayByGroup(x, ids, subset.row = NULL, subset.col = NULL, statistics = c("mean", "sum", "num`
From an assay matrix, compute summary statistics for groups of cells.

`ids` factor specifying the group for each cell

`statistics` character vector of statistics to compute

`threshold` threshold for detection

Author(s)

Patrick Aboyoun

See Also

- DuckDBMatrix-class for the main class
- DuckDBArray-matrixStats for the matrixStats methods
- perCellQCMetrics for the scuttle generic
- perFeatureQCMetrics for the scuttle generic
- librarySizeFactors for the scuttle generic
- normalizeCounts for the scuttle generic
- calculateCPM for the scuttle generic
- calculateAverage for the scuttle generic
- calculateTPM for the scuttle generic
- numDetectedAcrossFeatures for the scuttle generic
- sumCountsAcrossFeatures for the scuttle generic
- summarizeAssayByGroup for the scuttle generic

Examples

```
mat <- matrix(rpois(500, 5), 25, 20)
dimnames(mat) <- list(paste0("G", 1:25), paste0("C", 1:20))
names(dimnames(mat)) <- c("index1", "index2")
tmp <- tempfile()
writeParquet(mat, tmp)
pqmat <- DuckDBMatrix(tmp, datacol = "value",
    keycols = lapply(dimnames(mat), function(x) setNames(seq_along(x), x)))
head(librarySizeFactors(pqmat))
unlink(tmp, recursive = TRUE)
```

`initializeCpp` *Initialize DuckDB-backed matrices for beachmat*

Description

Registers a `initializeCpp` method for DuckDBArraySeed objects (the seed underlying every DuckDBMatrix), giving compiled C++ code that goes through **beachmat/tatami** (e.g. `BiocSingular`'s `compute_center/compute_scale`) a fast path instead of always falling back to **beachmat**'s generic "unknown matrix" block-processing path.

Usage

```
## S4 method for signature 'DuckDBArraySeed'
initializeCpp(
  x,
  ...,
  duckdb.realize = initializeOptions("realize"),
  duckdb.memory_limit = initializeOptions("memory_limit")
)
```

Arguments

x	A DuckDBArraySeed object.
...	Further arguments, passed to **beachmat**'s generic fallback method when the fast path is not applicable.
duckdb.realize	Logical scalar, see the realize option in initializeOptions.
duckdb.memory_limit	Numeric scalar, see the memory_limit option in initializeOptions.

Details

Unlike beachmat.hdf5/beachmat.tiledb, there is no dedicated tatami_duckdb C++ library implementing genuine on-demand, out-of-core reads. This method instead materializes the seed's underlying COO table into an in-memory sparse matrix via a single bulk SQL pull (loadIntoMemory), then delegates to **beachmat**'s own native initializeCpp,dgCMatrix-method. This mirrors beachmat.hdf5's *opt-in* realize path, not its default on-demand streaming path, and is only attempted for a zero-filled (sparse) seed within initializeOptions's "memory_limit".

Unlike beachmat.hdf5/beachmat.tiledb, the materialized pointer is **not** cached across calls with checkMemoryCache. Verified empirically that doing so is unsafe here: two DuckDBMatrix objects built from the same underlying query (the common case, e.g. re-reading the same release twice) render identical SQL and would hit the same cache entry, but reusing that cached pointer across a later, logically separate computation silently corrupted results. Every call re-materializes, which costs a fraction of a second for a "memory_limit"-sized matrix, negligible next to the speedup this fast path already provides.

This is purely a performance optimization for anything that already goes through initializeCpp (e.g. BiocSingular::runSVD's center/scale computation). It does **not** speed up irlba's Lanczos iteration itself, which calls %*% directly on the DuckDBMatrix object rather than through initializeCpp; use the SQL-pushdown %*%/crossprod methods in **DuckDBArray** for that, or materialize small-enough matrices with loadIntoMemory and drive irlba directly.

Whenever the fast path is not applicable (a non-zero-filled seed, more than 2 keyed dimensions, or an estimated size over "memory_limit"), this method falls back to **beachmat**'s existing generic path rather than raising an error, so it can never make previously-working (if slower) code stop working.

Value

An external pointer that can be used in any **tatami**-compatible function.

Author(s)

Patrick Aboyoun

See Also

- initializeCpp for the generic
- loadIntoMemory to force materialization directly
- initializeOptions for the tunable defaults

Examples

```
# A genuinely sparse 2x2 matrix: (r2, c2) is an implicit zero.
df <- data.frame(row = c("r1", "r2", "r1"), col = c("c1", "c1", "c2"),
                 value = c(1, 2, 3))
tf <- tempfile(fileext = ".parquet")
on.exit(unlink(tf))
arrow::write_parquet(df, tf)
pqmat <- DuckDBArray::DuckDBMatrix(tf, datacol = "value",
                                   keycols = list(row = c("r1", "r2"), col = c("c1", "c2")))
ptr <- beachmat::initializeCpp(pqmat@seed)
```

`initializeOptions` *Options for DuckDB-backed matrices*

Description

Options for initializing DuckDB-backed matrices in `initializeCpp`.

Usage

```
initializeOptions(option, value)
```

Arguments

`option`	String specifying the name of the option.
`value`	Value of the option.

Details

The following options are supported:

- `realize`, a logical scalar specifying whether to materialize a DuckDBArraySeed's underlying COO table into an in-memory sparse matrix with `loadIntoMemory` on every `initializeCpp` call (unlike `beachmat.hdf5`/`beachmat.tiledb`, this is deliberately **not** cached across calls, see `initializeCpp`'s details for why). Unlike `beachmat.hdf5`/`beachmat.tiledb` (which default this to `FALSE` because they have a genuine on-demand streaming backend), this defaults to `TRUE`: BiocDuckDB has no native out-of-core **tatami** backend, so setting this to `FALSE` falls back to **beachmat**'s generic "unknown matrix" block-processing path rather than a faster streaming one.
- `memory_limit`, a numeric scalar specifying the maximum estimated size, in bytes, of the sparse matrix materialized by `loadIntoMemory`. Above this limit, materialization is refused so a genuinely out-of-core matrix fails loudly (and falls back to the generic path, when reached via `initializeCpp`) instead of risking an OOM.

Value

If `value` is missing, the current setting of `option` is returned. If `value` is supplied, it is used to set the option, and the previous value of the option is invisibly returned.

Author(s)

Patrick Aboyoun

See Also

initializeCpp, loadIntoMemory

Examples

```
initializeOptions("memory_limit")
old <- initializeOptions("memory_limit", 4 * 1024^3)
initializeOptions("memory_limit")
initializeOptions("memory_limit", old)
```

---

linkSpatialMap *Link assay observations to their spatial layer rows*

Description

Joins the observations recorded in spatialMap to the rows of the spatial layer they reference, through the full junction key (assay, colname, element_type, region, instance_id). Resolves a single (element_type, region) layer (inferred from the filtered spatialMap when unambiguous, else specify); the join is pushed to DuckDB and returned as a lazy DuckDBDataFrame with the assay identity columns plus the layer's coordinate columns.

Usage

```
linkSpatialMap(
  mase,
  assay = NULL,
  element_type = NULL,
  region = NULL,
  conn = acquireDuckDBConn()
)
```

Arguments

mase	A MultiAssaySpatialExperiment.
assay, element_type, region	Optional filters selecting the spatialMap rows (and thus the single layer) to link.
conn	A DuckDB connection (default the shared BiocDuckDB connection).

Value

A lazy DuckDBDataFrame of linked observations.

Examples

```
mase <- readParquet(system.file("extdata", "spatial_mase",
                                package = "BiocDuckDB"))
linkSpatialMap(mase, assay = "assay1")
```

`loadIntoMemory` *Load a DuckDB-backed matrix into memory*

Description

Load a DuckDBMatrix's (or its DuckDBArraySeed's) underlying COO table into memory as an external pointer to a **tatami**-compatible representation, via a single bulk SQL pull. This is what `initializeCpp` calls internally when `duckdb.realize = TRUE` (the default); call it directly when you want the materialized pointer without going through the `initializeCpp` generic. Each call re-materializes; the result is deliberately not cached (see `initializeCpp`'s details for why).

Usage

```
loadIntoMemory(x, memory_limit = initializeOptions("memory_limit"))
```

Arguments

`x`	A DuckDBMatrix or DuckDBArraySeed object.
`memory_limit`	Numeric scalar, see the `memory_limit` option in `initializeOptions`. Materialization is refused above this estimated size (bytes) rather than risking an OOM.

Value

An external pointer that can be used in **tatami**-based functions.

Author(s)

Patrick Aboyoun

See Also

`initializeCpp`, `initializeOptions`

Examples

```
# A genuinely sparse 2x2 matrix: (r2, c2) is an implicit zero.
df <- data.frame(row = c("r1", "r2", "r1"), col = c("c1", "c1", "c2"),
                 value = c(1, 2, 3))
tf <- tempfile(fileext = ".parquet")
on.exit(unlink(tf))
arrow::write_parquet(df, tf)
pqmat <- DuckDBArray::DuckDBMatrix(tf, datacol = "value",
                                   keycols = list(row = c("r1", "r2"), col = c("c1", "c2")))
ptr <- loadIntoMemory(pqmat)
```

`MultiAssaySpatialExperiment-spatial` *MultiAssaySpatialExperiment lazy spatial methods*

Description

Lazy MultiAssaySpatialExperiment methods that keep spatial layers as DuckDBDataFrame objects when possible. In-memory MASE defaults are used when no lazy layers are present.

Value

`readParquetForMASE()` returns the parsed table — a lazy `DuckDBDataFrame` for large files, or an in-memory object for small ones — and `readGeoParquetForMASE()` returns a geometry-bearing object (an **sf** data frame). `annotateWithRegions()`, `subsetByBoundingBox()`, and `subsetByPolygon()` return a MultiAssaySpatialExperiment (annotated or spatially subset).

Parquet readers

`readParquetForMASE(file_path, col_select = NULL, ...)`**:** Files larger than 50 MB are read as lazy `DuckDBDataFrame` objects; smaller files use the in-memory **MultiAssaySpatialExperiment** method.

`readGeoParquetForMASE(file_path, ...)`**:** Uses `readGeoParquet` when **DuckDBSpatial** is installed; otherwise falls back to **sfarrow**.

Spatial operations

`annotateWithRegions(x, points, shapes, ...)`**:** Annotate points with shape regions using lazy `spatialMatch`.

`subsetByBoundingBox(x, xmin, xmax, ymin, ymax, ...)`**:** Subset by bounding box with lazy layer filtering when applicable.

`subsetByPolygon(x, polygon, ...)`**:** Subset by polygon with lazy layer filtering when applicable.

See Also

`readParquetForMASE`, `readGeoParquetForMASE`, `annotateWithRegions`, `subsetByBoundingBox`, and `subsetByPolygon`.

Examples

```
# subsetByBoundingBox on a DuckDB-backed layer uses DuckDB's optional spatial
# extension; probe for it so the example is skipped cleanly where the
# extension cannot load (offline / firewalled repository / not installed).
spatial_ok <- requireNamespace("MultiAssaySpatialExperiment", quietly = TRUE) &&
    requireNamespace("sf", quietly = TRUE) &&
    requireNamespace("DuckDBSpatial", quietly = TRUE) &&
    tryCatch({
        DBI::dbGetQuery(DuckDBDataFrame::acquireDuckDBConn(),
            "SELECT ST_Area(ST_GeomFromText('POLYGON((0 0,0 1,1 1,1 0,0 0))'))")
        TRUE
    }, error = function(e) FALSE)
```

```
if (spatial_ok) {
    pts <- S4Vectors::DataFrame(
        x = 1:3, y = 1:3, instance_id = c("A", "B", "C"))
    mat <- matrix(1:9, 3, 3,
        dimnames = list(paste0("G", 1:3), c("A", "B", "C")))
    mase <- MultiAssaySpatialExperiment::MultiAssaySpatialExperiment(
        experiments = MultiAssayExperiment::ExperimentList(rna = mat),
        colData = S4Vectors::DataFrame(row.names = "s1"),
        sampleMap = S4Vectors::DataFrame(
            assay = "rna", primary = "s1", colname = c("A", "B", "C")),
        points = MultiAssaySpatialExperiment::PointsLayerList(centroids = pts))
    tmp <- tempfile()
    writeParquet(mase, tmp)
    mase2 <- readParquet(tmp)
    subsetByBoundingBox(mase2, xmin = 1, xmax = 3, ymin = 1, ymax = 3)
    unlink(tmp, recursive = TRUE)
}
```

`readParquet` *Read Parquet Representation of a Bioconductor Object*

Description

Reads a parquet representation of various Bioconductor objects from disk using the Frictionless Data Package metadata model, reconstructing them as DuckDB-backed objects. This function is the counterpart to `writeParquet` and supports reading `SummarizedExperiment`, `RangedSummarizedExperiment`, `SingleCellExperiment`, and `MultiAssayExperiment` objects that have been written to parquet format with comprehensive metadata.

Usage

```
readParquet(
  path,
  package = .readDatapackage(path),
  model = package[["model"]],
  ...
)
```

Arguments

`path`	A character string specifying the path to the directory containing the parquet files. The directory must contain a `datapackage.json` file written by `writeParquet`.
`package`	A list containing the parsed `datapackage.json` contents. Defaults to reading `file.path(path, "datapackage.json")`. Contains the package-level `model` field and a `resources` list describing each parquet dataset.
`model`	A character string identifying the package schema — which container class to reconstruct. Defaults to `package[["model"]]`. Supported values: `"summarized_experiment"`, `"ranged_summarized_experiment"`, `"single_cell_experiment"`, `"experiment_list"`, `"multi_assay_experiment"`, `"multi_assay_spatial_experiment"`. When `NULL` (absent from the package JSON), all resources are returned as a `SimpleList` with no imposed schema.
`...`	Additional arguments passed to internal helper functions.

Details

This function reads parquet files that were created by writeParquet using the Frictionless Data Package metadata model and reconstructs the original Bioconductor object structure. This function supports different object types through a switch statement that routes to appropriate helper functions:

SummarizedExperiment **objects:** Reads feature data, sample data, and assay data from separate parquet files, reconstructing the object with DuckDB-backed components. For ranged experiments, genomic coordinates are reconstructed as DuckDBGRanges objects.

SingleCellExperiment **objects:** Extends SummarizedExperiment functionality by also reading reduced dimensions, loadings, alternative experiments, row / column tables, and row / column pairings from their respective parquet files.

MultiAssayExperiment **objects:** Reads multiple experiments along with sample data and sample mapping information, reconstructing the multi-experiment structure.

This function uses the Frictionless Data Package metadata to understand the structure of the data, including resource schemas with field definitions, primary keys, and semantic meanings (sequence_name, start, end, strand for genomic data) encoded in the datapackage.json file.

Value

A Bioconductor object of the appropriate class, reconstructed using DuckDB backend classes:

- DuckDBMatrix objects for assay data
- DuckDBDataFrame objects for row and column metadata
- DuckDBGRanges objects for genomic ranges
- DuckDBGRangesList objects for grouped genomic ranges
- DuckDBSelfHits objects for graph edge lists
- DuckDBDualSubset objects for pairwise graphs (wrapping DuckDBSelfHits)

Supported Object Types

GenomicRanges Read with genomic coordinates (seqnames, start, end, strand) and optional metadata columns.

GenomicRangesList Read with genomic coordinates and metadata columns.

DuckDBSelfHits Graph edge lists with from, to columns and optional metadata. Node count (nnode) is stored in the schema graphEdges property.

SummarizedExperiment Feature metadata from features/, sample metadata from samples/, and assays from flat assay_<name>/ directories. Any complex objects stored in metadata() are written to unbound_<name>/ directories and restored on read.

RangedSummarizedExperiment As SummarizedExperiment, with rowRanges reconstructed as a DuckDBGRanges from features/.

SingleCellExperiment Extends SummarizedExperiment with: reduced dimensions from sample_embeddings/; row loadings from feature_embeddings/; alternative experiments from experiment_<name>/ directories; row/column tables from flat feature_table_<name>/ and sample_table_<name>/ directories; row/column pairwise graphs from flat feature_graph_<name>/ and sample_graph_<name>/ directories.

MultiAssayExperiment Experiments written directly to root as experiment_<name>/ directories (flattened). Each SummarizedExperiment has its own datapackage.json (layout = "nested_experiment"). Also includes subjects (subject metadata) and sample_map (subject-to-sample mapping) as unbound resources.

`MultiAssaySpatialExperiment` Extends `MultiAssayExperiment` with spatial points from flat `sample_points_<name>/` directories, shapes from flat `sample_shapes_<name>/` directories, image metadata from `img_data/`, and spatial mapping from `spatial_map/`. Requires the `MultiAssaySpatialExperiment` package.

Frictionless Data Package Metadata

This function reads data packages that follow the Frictionless Data Package specification (extended by the BiocDuckDB profile), parsing the `datapackage.json` metadata file. Dispatch operates at two levels:

Package level (root `datapackage.json`):

- `model` — which container class to reconstruct; absent means `SimpleList` fallback.
- `annotations` — non-relational metadata from `metadata(x)` (scalars, vectors, 1-D arrays, JSON-safe nested lists). Tabular items are referenced via `parquet_ref` stubs pointing at unbound Parquet sidecars; mixed nested lists use a `nested_mapping` wrapper.

Resource level (entries in `resources` array):

- `dimension` — biological axis (`"feature"`, `"sample"`, `"crossed"`, `"unbound"`).
- `layout` — physical storage pattern (e.g., `"data_frame"`, `"coord_array"`, `"graph_edges"`, `"nested_experiment"`); routes each resource to the appropriate low-level reader via `.readParquetResource`.
- `schema` — field definitions, primary keys, and BiocDuckDB annotations (`genomicCoords`, `graphEdges`, `arrayItem`).

Author(s)

Patrick Aboyoun

See Also

- `writeParquet` for writing Bioconductor objects to parquet
- `DuckDBMatrix` for matrix storage
- `DuckDBDataFrame` for metadata storage
- `DuckDBGRanges` for genomic ranges
- `DuckDBGRangesList` for grouped genomic ranges
- `DuckDBSelfHits` for graph edge lists

Examples

```
se <- SummarizedExperiment::SummarizedExperiment(
    assays = list(counts = matrix(rpois(500, 5), 25, 20)),
    colData = S4Vectors::DataFrame(sample = rep(c("A", "B"), each = 10)))
tmpdir <- tempfile()
writeParquet(se, tmpdir)
se2 <- readParquet(tmpdir)
class(se2)
dim(se2)
unlink(tmpdir, recursive = TRUE)
```

`SingleCellExperiment-colTables` *Sample table methods*

Description

Methods to get or set nested sample-level tables in a SingleCellExperiment object. These tables enable storage of multi-column relational data associated with samples (cells/perturbations), which would otherwise require nested DataFrames in `colData()`. By extracting nested tables into a separate slot, they can be serialized to independent Parquet tables for efficient querying.

Arguments

`x`	A SingleCellExperiment object.
`type`	String or integer scalar specifying the name or index of the table to get or set.
`withDimnames`	Logical scalar indicating whether row names should be extracted from (getters) or set to (setters) the column names of `x`.
`...`	Additional arguments, currently ignored.
`value`	For the getter, a DataFrame-like object with number of rows equal to `ncol(x)`, containing the table data. For `colTables<-`, a list of such DataFrames. For `colTableNames<-`, a character vector of names.

Details

Sample tables (`colTables`) are stored in `int_colData(x)$colTables` and provide a mechanism to unnest complex relational data from `colData()`. This is particularly useful when sample metadata contains multi-valued properties or hierarchical structures that are difficult to query when embedded as nested DataFrames.

Common use cases include:

- Patient metadata: multiple disease diagnoses, treatment histories, or demographic details per sample
- Experimental metadata: replicate-level measurements, batch details, or quality control metrics
- Cell lineage: developmental trajectories or cell state transitions

Each table is a DataFrame with `ncol(x)` rows, maintaining alignment with samples. Tables are automatically subset when the parent SingleCellExperiment is subset by columns.

Value

For `colTable`, a DataFrame containing sample-level relational data.

For `colTables`, a SimpleList of such DataFrames.

For `colTableNames`, a character vector of table names.

For all setters, `x` is returned with the modified tables or names.

Getters

In the following examples, x is a SingleCellExperiment object.

`colTable(x, type, withDimnames=TRUE)`: Retrieves a DataFrame containing sample-level relational data. `type` is either a string specifying the name of the table in `x` to retrieve, or a numeric scalar specifying the index of the desired table.

If `withDimnames=TRUE`, row names of the output DataFrame are replaced with the column names of `x`.

`colTableNames(x)`: Returns a character vector containing the names of all tables in `x`. This is guaranteed to be of the same length as the number of tables, though the names may not be unique.

`colTables(x, withDimnames=TRUE)`: Returns a named SimpleList of DataFrames containing one or more tables. Each table has the same number of rows as `ncol(x)`.

If `withDimnames=TRUE`, row names of each DataFrame are replaced with the column names of `x`.

Single-table setter

`colTable(x, type, withDimnames=TRUE) <- value` will add or replace a table in a SingleCellExperiment object `x`. The value of `type` determines how the table is added or replaced:

- If `type` is a numeric scalar, it must be within the range of existing tables, and `value` will be assigned to the table at that index.
- If `type` is a string and a table exists with this name, `value` is assigned to that table. Otherwise a new table with this name is appended to the existing list of tables.

`value` is expected to be a DataFrame or data.frame-like object with number of rows equal to `ncol(x)`.

If `withDimnames=TRUE`, row names of `value` are set to `colnames(x)`.

Other setters

In the following examples, `x` is a SingleCellExperiment object.

`colTables(x, withDimnames=TRUE) <- value`: Replaces all tables in `x` with those in `value`. The latter should be a list-like object containing any number of DataFrames or data.frame-like objects with number of rows equal to `ncol(x)`.

If `value` is named, those names will be used to name the tables in `x`.

If `value` is a Annotated object, any `metadata` will be retained in `colTables(x)`. If `value` is a Vector object, any `mcols` will also be retained.

If `withDimnames=TRUE`, row names in each entry of `value` are set to `colnames(x)`.

`colTableNames(x) <- value`: Replaces all names for tables in `x` with a character vector `value`. This should be of length equal to the number of tables currently in `x`.

Author(s)

Patrick Aboyoun

See Also

`colData` for simple sample metadata.

`rowTables` for feature-level nested tables.

`int_colData` for the internal column metadata storage.

Examples

```
library(SingleCellExperiment)

# Create example SCE
sce <- SingleCellExperiment(
    assays = list(counts = matrix(rpois(1000, 5), 100, 10))
)
rownames(sce) <- paste0("Gene", 1:100)
colnames(sce) <- paste0("Cell", 1:10)

# Add a table of patient disease history (multiple diseases per patient)
diseases <- DataFrame(
    disease_id = paste0("MONDO:", sample(100000:999999, 10)),
    disease_label = paste0("Disease_", 1:10),
    onset_age = sample(20:80, 10, replace = TRUE)
)
colTable(sce, "diseases") <- diseases

# Add a table of per-sample QC metrics
qc_metrics <- DataFrame(
    metric_name = rep(c("n_genes", "n_counts", "pct_mito"), length.out = 10),
    value = runif(10, 1000, 5000),
    passed = sample(c(TRUE, FALSE), 10, replace = TRUE)
)
colTable(sce, "qc") <- qc_metrics

# Retrieve all tables
all_tables <- colTables(sce)
names(all_tables)

# Retrieve specific table by name
dis <- colTable(sce, "diseases")
dim(dis)  # 10 samples × 3 columns

# Retrieve by index
qc <- colTable(sce, 2)

# Get table names
colTableNames(sce)

# Subset SCE - tables are automatically subset
sce_sub <- sce[, 1:5]
dim(colTable(sce_sub, "diseases"))  # 5 samples × 3 columns
```

SingleCellExperiment-loadings

Feature loading methods

Description

Methods to get or set feature-level loading matrices in a SingleCellExperiment object. These matrices represent per-feature contributions to latent factors, embeddings, or statistical models, where feature identity (e.g., gene names) carries semantic meaning. Each row of a loading matrix corresponds to a row (feature) of the SingleCellExperiment object.

Arguments

x	A SingleCellExperiment object.
type	String or integer scalar specifying the name or index of the loading result to get or set.
withDimnames	Logical scalar indicating whether row names should be extracted from (getters) or set to (setters) the row names of x.
...	Additional arguments, currently ignored.
value	For the getter, a matrix-like object with number of rows equal to nrow(x), containing the loading values. For rowLoadings<-, a list of such matrices. For rowLoadingNames<-, a character vector of names.

Details

Feature loadings (rowLoadings) are stored in int_elementMetadata(x)$rowLoadings and represent the AnnData .varm equivalent in R. Unlike sample embeddings (reducedDims) where dimension labels are arbitrary coordinates, loading matrices have **meaningful row names** that identify which features contribute to each latent dimension.

Common use cases include:

- PCA loadings: which genes load on each principal component
- Factor analysis: gene weights for latent factors
- Statistical summaries: per-gene effect sizes or log fold changes across conditions

Loading matrices are automatically subset when the parent SingleCellExperiment is subset by rows, maintaining alignment between features and their loadings.

Value

For rowLoading, a matrix containing loading values for features (rows) and dimensions (columns).

For rowLoadings, a SimpleList of such matrices.

For rowLoadingNames, a character vector of loading names.

For all setters, x is returned with the modified loading results or names.

Getters

In the following examples, x is a SingleCellExperiment object.

rowLoading(x, type, withDimnames=TRUE): Retrieves a matrix containing feature loadings (rows) for each latent dimension (columns). type is either a string specifying the name of the loading result in x to retrieve, or a numeric scalar specifying the index of the desired result.

If withDimnames=TRUE, row names of the output matrix are replaced with the row names of x.

rowLoadingNames(x): Returns a character vector containing the names of all loading results in x. This is guaranteed to be of the same length as the number of results, though the names may not be unique.

rowLoadings(x, withDimnames=TRUE): Returns a named SimpleList of matrices containing one or more loading results. Each result is a matrix with the same number of rows as nrow(x).

If withDimnames=TRUE, row names of each matrix are replaced with the row names of x.

Single-result setter

`rowLoading(x, type, withDimnames=TRUE) <- value` will add or replace a loading result in a SingleCellExperiment object `x`. The value of `type` determines how the result is added or replaced:

- If `type` is a numeric scalar, it must be within the range of existing results, and `value` will be assigned to the result at that index.
- If `type` is a string and a result exists with this name, `value` is assigned to that result. Otherwise a new result with this name is appended to the existing list of results.

`value` is expected to be a matrix or matrix-like object with number of rows equal to `nrow(x)`.

If `withDimnames=TRUE`, row names of `value` are set to `rownames(x)`.

Other setters

In the following examples, `x` is a SingleCellExperiment object.

`rowLoadings(x, withDimnames=TRUE) <- value`**:** Replaces all loading results in `x` with those in `value`. The latter should be a list-like object containing any number of matrices or matrix-like objects with number of rows equal to `nrow(x)`.

If `value` is named, those names will be used to name the loading results in `x`.

If `value` is a Annotated object, any `metadata` will be retained in `rowLoadings(x)`. If `value` is a Vector object, any `mcols` will also be retained.

If `withDimnames=TRUE`, row names in each entry of `value` are set to `rownames(x)`.

`rowLoadingNames(x) <- value`**:** Replaces all names for loading results in `x` with a character vector `value`. This should be of length equal to the number of results currently in `x`.

Author(s)

Patrick Aboyoun

See Also

`reducedDims` for sample-level embeddings (observation matrices).

`int_elementMetadata` for the internal row metadata storage.

Examples

```
library(SingleCellExperiment)

# Create example SCE with perturbation data
sce <- SingleCellExperiment(
    assays = list(beta_Z = matrix(rnorm(1000), 100, 10))
)
rownames(sce) <- paste0("Gene", 1:100)
colnames(sce) <- paste0("Pert", 1:10)

# Add PCA loadings (genes × components)
pca_loadings <- matrix(rnorm(100 * 5), 100, 5)
rownames(pca_loadings) <- rownames(sce)
colnames(pca_loadings) <- paste0("PC", 1:5)
rowLoading(sce, "PCA") <- pca_loadings

# Add factor analysis loadings (genes × factors)
factor_loadings <- matrix(rnorm(100 * 20), 100, 20)
```

```
rownames(factor_loadings) <- rownames(sce)
colnames(factor_loadings) <- paste0("Factor", 1:20)
rowLoading(sce, "FactorAnalysis") <- factor_loadings

# Retrieve all loadings
all_loadings <- rowLoadings(sce)
names(all_loadings)

# Retrieve specific loading by name
pca <- rowLoading(sce, "PCA")
dim(pca)  # 100 genes × 5 components

# Retrieve by index
factors <- rowLoading(sce, 2)

# Get loading names
rowLoadingNames(sce)

# Subset SCE - loadings are automatically subset
sce_sub <- sce[1:50, ]
dim(rowLoading(sce_sub, "PCA"))  # 50 genes × 5 components
```

`SingleCellExperiment-pairs` *colPairs/rowPairs setters for DuckDBSelfHits*

Description

Specialized setter methods for `colPairs` and `rowPairs` that automatically wrap DuckDBSelfHits objects in DuckDBDualSubset containers. This enables lazy node-based subsetting when a SingleCellExperiment is subset.

Arguments

`x`	A SingleCellExperiment object.
`type`	String or integer scalar specifying the name or index of the pairing to replace.
`...`	Additional arguments passed to the default setter.
`value`	A DuckDBSelfHits object to be stored as a column/row pairing. This will be automatically wrapped in a DuckDBDualSubset.

Details

These methods extend the default `colPairs<-` and `rowPairs<-` setters from SingleCellExperiment to recognize `DuckDBSelfHits` objects and wrap them in `DuckDBDualSubset` containers. This ensures that when a SingleCellExperiment is subset (e.g., `sce[, 1:100]`), the wrapped `DuckDBSelfHits` objects are lazily filtered via SQL rather than materialized in memory.

Value

For the setter methods, `x` is returned with the modified pairings.

Usage

```
# Create SCE with lazy KNN graph
library(SingleCellExperiment)
library(BiocDuckDB)

sce <- SingleCellExperiment(assays = list(counts = matrix(1:100, 10, 10)))

# Create DuckDBSelfHits from parquet
knn_hits <- DuckDBSelfHits("knn_edges.parquet",
                           from = "from",
                           to = "to",
                           nnode = ncol(sce))

# Assign to colPairs (auto-wraps in DuckDBDualSubset)
colPairs(sce, "knn") <- knn_hits

# Subsetting SCE automatically subsets the graph lazily
sce_sub <- sce[, 1:5]
pairs_sub <- colPairs(sce_sub, "knn")  # Lazy filtered to nodes 1:5
```

Author(s)

Patrick Aboyoun

See Also

DuckDBSelfHits for the lazy edge storage class.

DuckDBDualSubset for the wrapper that enables node-based subsetting.

`colPairs` and `rowPairs` for the base SingleCellExperiment methods.

Examples

```
library(SingleCellExperiment)
sce <- SingleCellExperiment(assays = list(counts = matrix(1:50, 10, 5)))
hits <- S4Vectors::SelfHits(
    from = rep(1:5, each = 2),
    to = sample(1:5, 10, replace = TRUE),
    nnode = 5L)
tmp <- tempfile()
writeParquet(hits, tmp)
ddb_hits <- DuckDBSelfHits(tmp, from = "from", to = "to", nnode = 5L)
colPair(sce, "knn") <- ddb_hits
class(colPair(sce, "knn"))
unlink(tmp, recursive = TRUE)
```

`SingleCellExperiment-rowTables`

Feature table methods

Description

Methods to get or set nested feature-level tables in a SingleCellExperiment object. These tables enable storage of multi-column relational data associated with features (genes), which would otherwise require nested DataFrames in `rowData()`. By extracting nested tables into a separate slot, they can be serialized to independent Parquet tables for efficient querying.

Arguments

`x`	A SingleCellExperiment object.
`type`	String or integer scalar specifying the name or index of the table to get or set.
`withDimnames`	Logical scalar indicating whether row names should be extracted from (getters) or set to (setters) the row names of `x`.
`...`	Additional arguments, currently ignored.
`value`	For the getter, a DataFrame-like object with number of rows equal to `nrow(x)`, containing the table data. For `rowTables<-`, a list of such DataFrames. For `rowTableNames<-`, a character vector of names.

Details

Feature tables (`rowTables`) are stored in `int_elementMetadata(x)$rowTables` and provide a mechanism to unnest complex relational data from `rowData()`. This is particularly useful when feature metadata contains multi-valued properties or hierarchical structures that are difficult to query when embedded as nested DataFrames.

Common use cases include:

- Gene annotation tables: transcript isoforms, protein domains, or GO terms
- Experimental design tables: per-gene treatment conditions or batch effects
- Statistical results: detailed per-gene statistics across multiple contrasts

Each table is a DataFrame with `nrow(x)` rows, maintaining alignment with features. Tables are automatically subset when the parent SingleCellExperiment is subset by rows.

Value

For `rowTable`, a DataFrame containing feature-level relational data.

For `rowTables`, a SimpleList of such DataFrames.

For `rowTableNames`, a character vector of table names.

For all setters, `x` is returned with the modified tables or names.

Getters

In the following examples, `x` is a SingleCellExperiment object.

`rowTable(x, type, withDimnames=TRUE)`**:** Retrieves a DataFrame containing feature-level relational data. `type` is either a string specifying the name of the table in `x` to retrieve, or a numeric scalar specifying the index of the desired table.

If `withDimnames=TRUE`, row names of the output DataFrame are replaced with the row names of `x`.

`rowTableNames(x)`**:** Returns a character vector containing the names of all tables in `x`. This is guaranteed to be of the same length as the number of tables, though the names may not be unique.

`rowTables(x, withDimnames=TRUE)`: Returns a named SimpleList of DataFrames containing one or more tables. Each table has the same number of rows as `nrow(x)`.

If `withDimnames=TRUE`, row names of each DataFrame are replaced with the row names of `x`.

Single-table setter

`rowTable(x, type, withDimnames=TRUE) <- value` will add or replace a table in a SingleCellExperiment object `x`. The value of `type` determines how the table is added or replaced:

- If `type` is a numeric scalar, it must be within the range of existing tables, and `value` will be assigned to the table at that index.
- If `type` is a string and a table exists with this name, `value` is assigned to that table. Otherwise a new table with this name is appended to the existing list of tables.

`value` is expected to be a DataFrame or data.frame-like object with number of rows equal to `nrow(x)`.

If `withDimnames=TRUE`, row names of `value` are set to `rownames(x)`.

Other setters

In the following examples, `x` is a SingleCellExperiment object.

`rowTables(x, withDimnames=TRUE) <- value`: Replaces all tables in `x` with those in `value`. The latter should be a list-like object containing any number of DataFrames or data.frame-like objects with number of rows equal to `nrow(x)`.

If `value` is named, those names will be used to name the tables in `x`.

If `value` is a Annotated object, any `metadata` will be retained in `rowTables(x)`. If `value` is a Vector object, any `mcols` will also be retained.

If `withDimnames=TRUE`, row names in each entry of `value` are set to `rownames(x)`.

`rowTableNames(x) <- value`: Replaces all names for tables in `x` with a character vector `value`. This should be of length equal to the number of tables currently in `x`.

Author(s)

Patrick Aboyoun

See Also

`rowData` for simple feature metadata.

`colTables` for sample-level nested tables.

`int_elementMetadata` for the internal row metadata storage.

Examples

```
library(SingleCellExperiment)

# Create example SCE
sce <- SingleCellExperiment(
    assays = list(counts = matrix(rpois(1000, 5), 100, 10))
)
rownames(sce) <- paste0("Gene", 1:100)
colnames(sce) <- paste0("Cell", 1:10)
```

```
# Add a table of gene isoforms (multiple isoforms per gene)
isoforms <- DataFrame(
    isoform_id = paste0("ISO", 1:100),
    length = sample(500:5000, 100),
    is_canonical = sample(c(TRUE, FALSE), 100, replace = TRUE)
)
rowTable(sce, "isoforms") <- isoforms

# Add a table of differential expression results across conditions
de_results <- DataFrame(
    condition = rep(c("treated", "control"), each = 50),
    log2fc = rnorm(100),
    pvalue = runif(100)
)
rowTable(sce, "de_stats") <- de_results

# Retrieve all tables
all_tables <- rowTables(sce)
names(all_tables)

# Retrieve specific table by name
iso <- rowTable(sce, "isoforms")
dim(iso)  # 100 genes × 3 columns

# Retrieve by index
de <- rowTable(sce, 2)

# Get table names
rowTableNames(sce)

# Subset SCE - tables are automatically subset
sce_sub <- sce[1:50, ]
dim(rowTable(sce_sub, "isoforms"))  # 50 genes × 3 columns
```

`spatialCoordinateSystems` *Coordinate systems of a MultiAssaySpatialExperiment*

Description

Returns the names of the coordinate systems referenced by the per-element transforms stored in `metadata(x)$transforms` (the targets each element is mapped into). Empty when no transforms are recorded.

Usage

```
spatialCoordinateSystems(x)
```

Arguments

x	A MultiAssaySpatialExperiment.

Value

A character vector of coordinate-system names.

Examples

```
mase <- readParquet(system.file("extdata", "spatial_mase",
                                package = "BiocDuckDB"))
spatialCoordinateSystems(mase)
```

`spatialElementJoin` *Spatial join between a points element and a shapes element of a MASE*

Description

Joins a points element to a shapes element (each named `"<element_type>/<region>"`) by a spatial predicate, pushed to DuckDB via **DuckDBSpatial**. When `coordinate_system` is given and the MASE carries transforms, each element is first aligned into that coordinate system through the coordinate-transform graph (`spatialCoordinateSystems`), so elements defined in different intrinsic frames are compared correctly. This generalizes `annotateWithRegions` (point-in-polygon with the default predicate only) to any **sf** binary predicate.

Usage

```
spatialElementJoin(
  mase,
  x,
  y,
  join = sf::st_intersects,
  coordinate_system = NULL,
  x_col = "x",
  y_col = "y"
)
```

Arguments

`mase`	A MultiAssaySpatialExperiment.
`x`	A points element spec, `"points/<region>"` (or `list(element_type = "points", region = )`).
`y`	A shapes element spec, `"shapes/<region>"` (or `list(element_type = "shapes", region = )`).
`join`	A spatial predicate function (default `st_intersects`).
`coordinate_system`	Optional target coordinate-system name to align both elements into before joining.
`x_col, y_col`	Coordinate column names in `x`, the points element.

Value

The spatial-join result from **DuckDBSpatial** (`spatialMatch` indices matching each point in `x` to a row of `y`, `NA` where unmatched).

Examples

```
pts <- S4Vectors::DataFrame(x = c(1.5, 2.5, 3.5), y = c(1.5, 2.5, 3.5),
                            instance_id = c("A", "B", "C"))
shp <- S4Vectors::DataFrame(instance_id = c("cell1", "cell2", "cell3"),
    geometry = sf::st_sfc(
        sf::st_polygon(list(matrix(c(1,1,2,1,2,2,1,2,1,1), ncol = 2, byrow = TRUE))),
        sf::st_polygon(list(matrix(c(2,1,3,1,3,2,2,2,2,1), ncol = 2, byrow = TRUE))),
        sf::st_polygon(list(matrix(c(2,2,3,2,3,3,2,3,2,2), ncol = 2, byrow = TRUE)))))
mase <- MultiAssaySpatialExperiment::MultiAssaySpatialExperiment(
    experiments = MultiAssayExperiment::ExperimentList(assay1 = matrix(1:9, 3, 3,
        dimnames = list(paste0("G", 1:3), c("A", "B", "C")))),
    colData = S4Vectors::DataFrame(row.names = "s1"),
    sampleMap = S4Vectors::DataFrame(assay = "assay1", primary = "s1",
                                     colname = c("A", "B", "C")),
    points = MultiAssaySpatialExperiment::PointsLayerList(centroids = pts),
    shapes = MultiAssaySpatialExperiment::ShapesLayerList(cells = shp))
spatialElementJoin(mase, "points/centroids", "shapes/cells")
```

`spatialViews` *On-the-fly DuckDB views over a MASE's spatial elements*

Description

Registers each spatial layer (points, shapes) and the `spatialMap` junction of a MultiAssaySpatialExperiment as DuckDB temp views on a shared connection, so cross-element queries can be expressed as SQL. Lazy (DuckDBDataFrame) layers are registered as views over their rendered SQL (no materialization); in-memory layers and the materialized `spatialMap` are registered as temp tables. This is the substrate used by `linkSpatialMap` / `validateSpatialMap` and a handle for power-user raw SQL.

Usage

```
spatialViews(mase, conn = acquireDuckDBConn(), prefix = "mase_")
```

Arguments

`mase`	A MultiAssaySpatialExperiment.
`conn`	A DuckDB connection (default the shared BiocDuckDB connection).
`prefix`	View-name prefix.

Value

A `MASESpatialViews` registry (a list of view names + `conn`).

Examples

```
mase <- readParquet(system.file("extdata", "spatial_mase",
                                package = "BiocDuckDB"))
v <- spatialViews(mase)
v$spatial_map
```

validateSpatialMap *Validate a MASE's spatialMap referential integrity*

Description

Checks the spatialMap foreign keys against the spatial layers and the assays, out-of-core via DuckDB anti-joins on lazy layers. Reports four violation classes: unknown_layer (a (element_type, region) with no matching layer), orphan_instance (a spatialMap row whose instance_id is absent from its layer), orphan_colname (an (assay, colname) that is not a column of that experiment), and duplicate_instance (an instance_id occurring more than once within a layer).

Usage

```
validateSpatialMap(mase, strict = FALSE, conn = acquireDuckDBConn())
```

Arguments

mase	A MultiAssaySpatialExperiment.
strict	If TRUE, error when any violation is found instead of returning the report.
conn	A DuckDB connection (default the shared BiocDuckDB connection).

Value

A DataFrame of violations (type, assay, colname, element_type, region, instance_id, detail); empty when the spatialMap is valid.

Examples

```
mase <- readParquet(system.file("extdata", "spatial_mase",
                                package = "BiocDuckDB"))
validateSpatialMap(mase)  # empty report: the spatialMap is valid
```

writeDatapackage *Write a Frictionless datapackage.json envelope*

Description

Assembles and writes the top-level datapackage.json manifest from a list of already-written Frictionless resource descriptors. This is the envelope step shared by the experiment-level writeParquet methods, exposed so that producers who accumulate resources incrementally (streaming large datasets, or promoting from another store) can emit a conformant manifest without reconstructing an in-memory Bioconductor object.

Usage

```
writeDatapackage(
  model,
  resources,
  path,
  main_exp_name = NULL,
  annotations = NULL
)
```

Arguments

model	Character(1) package-level schema identifier that selects the readParquet reader used to reconstruct the container (e.g. "summarized_experiment", "single_cell_experiment", "multi_assay_experiment"). See the storage-layout vignette for the documented model values.
resources	A list of Frictionless resource descriptors (each a list), as returned/accumulated from writeParquet. NULL entries are removed.
path	Character(1) directory to write datapackage.json into; created recursively if it does not exist.
main_exp_name	Optional character(1) naming the main experiment (used by the single_cell_experiment reader). Omitted from the manifest when NULL.
annotations	Optional list of non-relational metadata elements (as produced during metadata serialization). Omitted when NULL.

Details

Each entry of resources is a Frictionless resource descriptor – a list with name, path, dimension, layout, format, mediatype, and schema – exactly as returned by the primitive writeParquet methods (array, data.frame, DataFrame, SelfHits). NULL entries are dropped, so the NULL returned by append/streaming parts (see writeParquet) can be accumulated and passed straight through. Descriptors are written verbatim otherwise; strip any private, non-Frictionless keys before calling.

Value

Invisibly, the assembled package list that was written.

Author(s)

Patrick Aboyoun

See Also

writeParquet for writing resources, and readParquet for reading a written package (the DuckDBMatrix/DuckDBArray/D constructors attach an existing coord-array in place).

Examples

```
# Assemble a manifest from a hand-built resource descriptor.
tf <- tempfile()
resources <- list(list(
    name = "features", path = "features",
    dimension = "feature", layout = "data_frame",
    format = "parquet",
    mediatype = "application/vnd.apache.parquet",
    schema = list(fields = list(list(name = "id", type = "integer")))))
writeDatapackage("summarized_experiment", resources, tf)
cat(readLines(file.path(tf, "datapackage.json")), sep = "\n")
```

writeParquet *Write Parquet Representation of a Bioconductor Object*

Description

An `arrow::write_dataset` wrapper function to write the Parquet representation of various R objects using the Frictionless [Data Package](https://datapackage.org) metadata model. This function supports multiple object types including arrays, data frames, genomic ranges, and more, converting them into efficient Parquet format for analysis and storage with comprehensive metadata descriptions.

Usage

```
writeParquet(x, path, ...)

## S4 method for signature 'ANY'
writeParquet(
  x,
  path,
  indexcols = names(dimnames(x)) %||% sprintf("index%d", seq_along(dim(x))),
  indexrefs = NULL,
  datacol = "value",
  grid = defaultAutoGrid(COO_SparseArray(dim(x))),
  grid_suffix = "_group",
  BPPARAM = getAutoBPPARAM(),
  name = basename(path),
  dimension = c("crossed", "unbound"),
  layout = "coord_array",
  ...
)

## S4 method for signature 'data.frame'
writeParquet(
  x,
  path,
  indexcol = "__index__",
  keycol = "__name__",
  dimtbl = NULL,
  refs = NULL,
  append = FALSE,
  offset = 0L,
  part = NULL,
  part_digits = 0L,
  name = basename(path),
  dimension = c("unbound", "sample", "feature", "crossed"),
  layout = "data_frame",
  ...
)

## S4 method for signature 'DataFrame'
writeParquet(
```

```
  x,
  path,
  indexcol = "__index__",
  keycol = "__name__",
  dimtbl = NULL,
  refs = NULL,
  append = FALSE,
  offset = 0L,
  part = NULL,
  part_digits = 0L,
  name = basename(path),
  dimension = c("unbound", "sample", "feature", "crossed"),
  layout = "data_frame",
  ...
)

## S4 method for signature 'DuckDBTable'
writeParquet(
  x,
  path,
  indexcol = "__index__",
  keycol = "__name__",
  dimtbl = NULL,
  append = FALSE,
  offset = 0L,
  part = NULL,
  part_digits = 0L,
  name = basename(path),
  dimension = c("unbound", "sample", "feature", "crossed"),
  layout = "data_frame",
  ...
)

## S4 method for signature 'DuckDBDataFrame'
writeParquet(
  x,
  path,
  indexcol = "__index__",
  keycol = "__name__",
  dimtbl = NULL,
  append = FALSE,
  offset = 0L,
  part = NULL,
  part_digits = 0L,
  name = basename(path),
  dimension = c("unbound", "sample", "feature", "crossed"),
  layout = "data_frame",
  ...
)

## S4 method for signature 'DuckDBTransposedDataFrame'
writeParquet(
```

```
  x,
  path,
  indexcol = "__index__",
  keycol = "__name__",
  dimtbl = NULL,
  append = FALSE,
  offset = 0L,
  part = NULL,
  part_digits = 0L,
  name = basename(path),
  dimension = "crossed",
  layout = "transposed_data_frame",
  ...
)

## S4 method for signature 'DuckDBSelfHits'
writeParquet(
  x,
  path,
  indexcol = "__index__",
  keycol = "__name__",
  dimtbl = NULL,
  append = FALSE,
  offset = 0L,
  part = NULL,
  part_digits = 0L,
  name = basename(path),
  dimension = c("unbound", "sample", "feature"),
  layout = "graph_edges",
  ...
)

## S4 method for signature 'DuckDBGRanges'
writeParquet(
  x,
  path,
  indexcol = "__index__",
  keycol = "__name__",
  dimtbl = NULL,
  append = FALSE,
  offset = 0L,
  part = NULL,
  part_digits = 0L,
  name = basename(path),
  dimension = c("feature", "unbound"),
  layout = "genomic_ranges",
  ...
)

## S4 method for signature 'DuckDBGRangesList'
writeParquet(
  x,
```

```
  path,
  indexcol = "__index__",
  keycol = "__name__",
  dimtbl = NULL,
  append = FALSE,
  offset = 0L,
  part = NULL,
  part_digits = 0L,
  name = basename(path),
  dimension = c("feature", "unbound"),
  layout = "genomic_ranges_list",
  ...
)

## S4 method for signature 'TransposedDataFrame'
writeParquet(
  x,
  path,
  indexcol = "__index__",
  keycol = "__name__",
  dimtbl = NULL,
  name = basename(path),
  dimension = "crossed",
  layout = "transposed_data_frame",
  ...
)

## S4 method for signature 'list'
writeParquet(
  x,
  path,
  dimension = c("unbound", "sample", "feature", "crossed"),
  ...
)

## S4 method for signature 'List'
writeParquet(
  x,
  path,
  dimension = c("unbound", "sample", "feature", "crossed"),
  ...
)

## S4 method for signature 'GenomicRanges'
writeParquet(
  x,
  path,
  indexcol = "__index__",
  keycol = "__name__",
  dimtbl = NULL,
  name = basename(path),
  dimension = c("feature", "unbound"),
```

```
  layout = "genomic_ranges",
  ...
)

## S4 method for signature 'GenomicRangesList'
writeParquet(
  x,
  path,
  indexcol = "__index__",
  keycol = "__name__",
  dimtbl = NULL,
  name = basename(path),
  dimension = c("feature", "unbound"),
  layout = "genomic_ranges_list",
  ...
)

## S4 method for signature 'SelfHits'
writeParquet(
  x,
  path,
  indexcol = "__index__",
  keycol = "__name__",
  dimtbl = NULL,
  name = basename(path),
  dimension = c("unbound", "sample", "feature"),
  layout = "graph_edges",
  ...
)

## S4 method for signature 'Assays'
writeParquet(
  x,
  path,
  indexcols = c("__feature__", "__sample__"),
  indexrefs = NULL,
  datacol = "value",
  grid = defaultAutoGrid(COO_SparseArray(dim(x[[1L]]))),
  grid_suffix = "group__",
  dimension = "crossed",
  ...
)

## S4 method for signature 'SummarizedExperiment'
writeParquet(
  x,
  path,
  indexcols = c("__feature__", "__sample__"),
  package = list(model = ifelse(is(x, "RangedSummarizedExperiment"),
   "ranged_summarized_experiment", "summarized_experiment"), resources = list()),
  ...
)
```

```
## S4 method for signature 'SingleCellExperiment'
writeParquet(
  x,
  path,
  indexcols = c("__feature__", "__sample__"),
  package = list(model = "single_cell_experiment", resources = list()),
  ...
)

## S4 method for signature 'ExperimentList'
writeParquet(x, path, indexcols = c("__feature__", "__sample__"), ...)

## S4 method for signature 'MultiAssayExperiment'
writeParquet(
  x,
  path,
  indexcols = c("__feature__", "__sample__"),
  package = list(model = "multi_assay_experiment", resources = list()),
  ...
)

## S4 method for signature 'PointsLayerList'
writeParquet(x, path, ...)

## S4 method for signature 'ShapesLayerList'
writeParquet(x, path, ...)

## S4 method for signature 'MultiAssaySpatialExperiment'
writeParquet(
  x,
  path,
  indexcols = c("__feature__", "__sample__"),
  package = list(model = "multi_assay_spatial_experiment", resources = list()),
  ...
)
```

Arguments

`x` The object to write. Supported types include:

- Array-like objects - converted to coordinate (long) format
- `data.frame` / `DataFrame` objects - written with field type metadata
- `DuckDBTable` / `DuckDBDataFrame` objects - lazy SQL `COPY TO` export via `writeDuckDBTableParquet` in `parquet-io` (same append, `offset`, `part` contract as in-memory tables)
- `DuckDBTransposedDataFrame`, `DuckDBSelfHits`, `DuckDBGRanges`, `DuckDBGRangesList` - delegate to the underlying lazy table without materializing
- `TransposedDataFrame` objects - transposed before writing
- `list` / `List` objects - each element dispatched individually; requires `dimension`; `layout` defaults to `NULL` (each element's method supplies its own default layout)
- `GenomicRanges` objects - genomic coordinates with metadata

- GenomicRangesList objects - lists of genomic ranges stored as LIST-typed columns
- SelfHits objects - graph edge lists with node count metadata
- Assays objects - collection of feature-by-sample matrices
- SummarizedExperiment objects - feature/sample metadata with assays
- RangedSummarizedExperiment objects - as above with genomic ranges for features
- SingleCellExperiment objects - extends SummarizedExperiment with embeddings, alt experiments, dimension tables, and pairwise graphs
- ExperimentList objects - named collection of experiments; primitive method used internally by MultiAssayExperiment and SingleCellExperiment (altExps); returns resources without writing datapackage.json
- MultiAssayExperiment objects - multi-experiment study with subject metadata and sample map
- PointsLayerList / ShapesLayerList objects - spatial point and polygon layers
- MultiAssaySpatialExperiment objects - spatially-resolved multi-assay experiment

path — A character string specifying the path where the data will be written. The path will be created if it doesn't exist.

... — Additional arguments. For data.frame methods with part set: passed to write_parquet. Otherwise for tables: passed to write_dataset. For array-like objects (ANY method): passed to writeCoordArray, including:

append Logical; hive-partition append when TRUE (requires length(grid) > 1L). See ?writeCoordArray.

along Integer; append dimension (required when append = TRUE).

group_offset Non-negative integer; partition-group offset along along; defaults to 0L.

arrowtype Optional DataType for the value column; schema pinning on append is described in ?writeCoordArray.

max_dim Optional length-ndim(x) integer vector of index upper bounds.

existing_data_behavior Passed through to write_dataset when applicable.

Note: offset for array-like objects is documented above; table methods use offset as an explicit formal argument.

indexcols — For array-like objects, a character vector of column names for the integer index columns in the resulting table. Defaults to the names(dimnames(x)) or sprintf("index%d", seq_along(dim(x))) if dimension names are not available.

indexrefs — For array-like objects, an optional list of foreign key references for the index columns in the schema. Defaults to NULL.

datacol — For array-like objects, a character string specifying the column name containing the array values in the resulting table. Defaults to "value".

grid — For array-like objects, an optional ArrayGrid to use for partitioning array-like objects. If provided, the array will be split into chunks and written as separate files. Defaults to defaultAutoGrid(COO_SparseArray(dim(x))).

grid_suffix — For array-like objects, a character string to append to the partitioning directories if the grid contains more than one cell. Defaults to "_group".

BPPARAM — For array-like objects, a BiocParallelParam object to use for parallel processing when grid contains multiple cells. Defaults to getAutoBPPARAM().

name — A character string specifying the name of the resource. Defaults to the basename of the path argument.

dimension — A character string describing which biological axis the resource is indexed by. One of "feature", "sample", "crossed" (feature × sample Cartesian product), or "unbound" (not tied to either axis).

layout — A character string identifying the physical storage layout of the resource. Recorded in datapackage.json and used by readParquet to dispatch the correct reader. Examples: "data_frame", "coord_array", "embedding_table", "genomic_ranges", "graph_edges", "nested_data_frame", "nested_experiment".

indexcol — For data.frame and Vector objects, a character string specifying the column name for the integer index column in the resulting table. Defaults to "__index__". Set to NULL to omit this column.

keycol — For data.frame and Vector objects, a character string specifying the column name for the row names (data frames) / names (genomic ranges) column in the resulting table. Defaults to "__name__". Set to NULL to omit this column.

dimtbl — For data.frame and Vector objects, an optional data.frame containing a dimension table with partitioning information.

refs — For flat (data.frame / DataFrame) tables, an optional list of foreign key references to emit in the schema, the flat analog of indexrefs. Each entry is list(fields = <local>, reference = list(fields = <target>, resource = <res>)). Used, for example, to declare the MultiAssayExperiment sample_map primary -> subjects correspondence. Defaults to NULL.

append — Logical. For data.frame, DataFrame, and DuckDBTable methods: when TRUE, append a new flat part-*.parquet file under path (requires part). For array-like objects (ANY method): forwarded to writeCoordArray for hive-partition append (requires length(grid) > 1L; see along, group_offset there). Defaults to FALSE.

offset — Non-negative integer. For data.frame and DuckDBTable methods: added to the index column when indexcol is set (offset + seq_len(nrow(x)) in memory; offset + row_number() in SQL for lazy tables); ignored when indexcol is NULL. For array-like objects: forwarded to writeCoordArray as the coordinate shift along along. Defaults to 0L.

part — Optional non-negative integer; data.frame and DuckDBTable methods. When set, write a single part-<n>.parquet file (via arrow::write_parquet for in-memory tables, or DuckDB COPY TO for lazy tables). Required when append = TRUE on table writes.

part_digits — Zero-padding width for part in the filename (e.g. 2L yields part-00.parquet). Table methods (in-memory and lazy).

package — For experiment-level methods (SummarizedExperiment, etc.), a list used to accumulate datapackage.json contents before writing. Primitive data.frame methods ignore this argument; capture the return value on the first flat part and append later parts with append = TRUE. Contains:

- model - Package-level schema identifier (e.g. "single_cell_experiment"). Determines which readParquet reader reconstructs the object. Distinct from the resource-level layout field — model describes the whole package; layout describes one resource within it.

- `resources` - Accumulating list of resource entries, each contributed by a primitive `writeParquet` method call.

Callers should not normally set this; the default is supplied by each experiment method. Experiment methods accumulate `package[["resources"]]` within a single call and write `datapackage.json` at the end. Primitive `data.frame` methods do not update a caller's `package` across separate invocations; capture the list returned from the first `writeParquet` call and merge manually (see **Flat table append** below).

Details

This function provides specialized handling for different object types:

Array-like objects: Converts multi-dimensional arrays into a coordinate (long) format where each non-zero element is represented as a row with columns for each dimension and the value. For sparse arrays, only non-zero elements are written, making it efficient for sparse data. When a grid is provided with multiple cells, the array is partitioned and each partition is written to a separate subdirectory. Array writes delegate to `writeCoordArray`; arguments `append`, `along`, `offset`, `group_offset`, `arrowtype`, and `max_dim` are forwarded via `...` and documented on the `writeCoordArray` help page.

Flat table append (`data.frame` / `DataFrame`)**:** Use `part`, `part_digits`, `append`, and `offset` to stream chunked sample or feature tables as `part-0.parquet`, `part-1.parquet`, ... without temporary directories. The global index column (when `indexcol` is set) uses `offset + seq_len(nrow(x))` on each chunk. The first call returns a Frictionless resource list (one element); later append parts return `NULL`. Build `datapackage.json` from the first return, for example:

```
res <- writeParquet(chunk1, samples_dir, indexcol = "__sample__",
                    part = 0L, name = "samples", dimension = "sample")
pkg <- list(resources = res)
writeParquet(chunk2, samples_dir, indexcol = "__sample__",
             offset = nrow(chunk1), part = 1L, append = TRUE, ...)
```

`data.frame` **objects**: Writes data frames directly with optional row names and dimension lookup tables for partitioning information.

`DataFrame` **objects:** Writes data frames with schema properties that describe column types and semantics. Column metadata (`mcols`) is not preserved; use object-level `metadata()` for annotations.

`TransposedDataFrame` **objects:** Writes transposed data frames with optional row names and dimension lookup tables for partitioning information.

`GenomicRanges` **objects:** Writes genomic coordinates with proper column naming and schema properties (genomicCoords) that describe the mapping of genomic coordinate columns. Range metadata columns are written as regular columns in the output.

`GenomicRangesList` **objects:** Separates list element metadata from ranges data, writing them to different paths with schema properties that describe how to split the ranges into list elements.

`SelfHits` **objects:** Writes graph edge lists as a data frame with `from`, `to`, and optional metadata columns. The node count (`nnode`) is stored in the schema properties to enable reconstruction as a DuckDBSelfHits object.

Automatic unnesting of nested DataFrames: When writing `SingleCellExperiment` objects, any DataFrame-class columns found in `rowData()` or `colData()` are automatically extracted and moved to `rowTables()` or `colTables()` respectively. This unnesting enables independent Parquet serialization of multi-valued properties (e.g., multiple diseases per patient, multiple isoforms per gene) while keeping the main metadata tables flat and SQL-queryable. The original nested columns

are removed from rowData()/colData() after extraction. This behavior is automatic and requires no user intervention.

SummarizedExperiment **objects:** Writes multi-assay experiments with separate paths for feature data, sample data, and assay data.

RangedSummarizedExperiment **objects:** Extends SummarizedExperiment functionality with genomic ranges for features.

SingleCellExperiment **objects:** Extends SummarizedExperiment with single-cell specific data. All collections are written as flat resource directories directly under path. The directory name encodes both axis and type: feature_embeddings/, sample_embeddings/, feature_table_<name>/, sample_table_<name>/, feature_graph_<name>/, sample_graph_<name>/, experiment_<name>/. The path prefix is derived automatically from layout. Alternative experiments are written directly to root (flattened, same pattern as MultiAssayExperiment experiments). See the automatic unnesting section for details about DataFrame columns in rowData()/colData().

MultiAssayExperiment **objects:** Writes multi-experiment studies with separate paths for sample data, sample mapping, and experiment data.

Value

For primitive methods (ANY, data.frame, DataFrame, TransposedDataFrame, GenomicRanges, GenomicRangesList, SelfHits, Assays, list, List, ExperimentList, PointsLayerList, ShapesLayerList): invisibly returns a list of Frictionless resource entries suitable for inclusion in a datapackage.json resources array.

For experiment methods (SummarizedExperiment, SingleCellExperiment, MultiAssayExperiment, MultiAssaySpatialExperiment): invisibly returns NULL; the datapackage.json is written to path as a side effect.

For flat multi-part table writes (data.frame with append = TRUE and part > 0): invisibly returns NULL.

Frictionless Data Package Metadata

This function writes a datapackage.json file (Frictionless Data Package v2.0) alongside the Parquet files. The file contains two levels of metadata:

Package-level fields (top of datapackage.json):

- model - Overall schema identifier. Determines which readParquet reader is used to reconstruct the container object (e.g. "single_cell_experiment", "multi_assay_experiment"). When absent, readParquet returns a SimpleList of resources with no imposed schema.
- annotations - Non-relational elements from metadata(x) (scalars, vectors, 1-D arrays, JSON-safe nested lists). Tabular / relational items (DataFrame, matrix, 2-D arrays) are written as unbound Parquet sidecars and referenced here via parquet_ref stubs; mixed nested lists use a nested_mapping wrapper.

Per-resource fields (each entry in the resources array):

- name - Resource identifier (typically the object name)
- path - Relative directory path to the Parquet dataset
- dimension - Biological axis: "feature", "sample", "crossed", or "unbound"
- layout - Physical storage layout (BiocDuckDB extension); together with dimension, uniquely determines the R accessor and AnnData slot used during reconstruction
- format - File format ("parquet")

- mediatype - MIME type ("application/vnd.apache.parquet")
- schema - Field definitions including types, primary key, sort order, and foreign key references

See the BiocDuckDB storage patterns documentation for the full model table and dimension + layout dispatch table.

Author(s)

Patrick Aboyoun

See Also

- readParquet for reading Bioconductor objects from parquet
- writeCoordArray for coord-array layout, hive partitioning, and array append (append, along, offset, group_offset)
- parquet-io for shared Parquet I/O: flat append validation (setupFlatParquetWrite, checkAppendPart, validateAppendOffset), SQL COPY TO helpers (buildParquetCopySQL), and lazy table export (writeDuckDBTableParquet)
- createDimTables for creating dimension lookup tables
- write_dataset and write_parquet
- ArrayGrid for grid partitioning
- BiocParallelParam for parallel processing options

Examples

```
# Write the Titanic dataset to a single Parquet file
tf1 <- tempfile()
writeParquet(Titanic, tf1)
list.files(tf1, full.names = TRUE, recursive = TRUE)

# Write the state.x77 matrix to multiple Parquet files using grid partitioning
tf3 <- tempfile()
state_grid <- RegularArrayGrid(dim(state.x77), c(10, 4))
writeParquet(state.x77, file.path(tf3, "state"), grid = state_grid)
dimtbls <- createDimTables(state.x77, grid = state_grid)
list.files(tf3, full.names = TRUE, recursive = TRUE)
```

---

writeStreamingResource

Stream a large table to a single multi-part Parquet resource

Description

Write a table that does not fit in memory to one flat, multi-part Parquet resource by pulling it block-by-block from a producer callback. This owns the streaming bookkeeping that every large producer would otherwise hand-roll on top of writeParquet: the running row offset, the part index and zero-padded part_digits, the append flag (part 0 creates the directory, later parts append), the positional dimtbl slice per block, the index_max threading that keeps the __index__ column one consistent (possibly >2^31) integer type across parts, and the post-write integrity checks. The single descriptor it returns is ready to hand to writeDatapackage.

Usage

```
writeStreamingResource(
  blocks,
  path,
  dimension,
  indexcol = "__index__",
  keycol = "__name__",
  layout = "data_frame",
  refs = NULL,
  dimtbl = NULL,
  cluster_by = NULL,
  index_max = NULL,
  expected_rows = NULL,
  part_digits = 6L,
  name = basename(path),
  ...
)
```

Arguments

`blocks`	A function of one argument, the 1-based part index `i`. It must return the `i`-th block as a `data.frame` (or `DataFrame` / anything `as.data.frame`-able), or `NULL` when the stream is exhausted. Blocks are consumed in ascending index order; how a block maps to source rows (a row range, a sample id, a query page) is the producer's business. Empty (zero-row) blocks are skipped without consuming a `part`.
`path`	Directory for the resource's Parquet parts.
`dimension`	Biological axis of the resource: one of `"unbound"`, `"sample"`, `"feature"`, `"crossed"`.
`indexcol, keycol`	Passed to `writeParquet`. `indexcol` is the streamed row index (default `"__index__"`); `keycol` is the row-name column (`"__name__"`), or `NULL` for a row-number key.
`layout`	Physical layout string (default `"data_frame"`); also accepts other flat layouts `writeParquet` supports for a `data.frame` (e.g. `"spatial_points"`).
`refs, cluster_by`	Passed to `writeParquet` (foreign-key references; on-write clustering key).
`dimtbl`	An optional full, index-ordered dimension-table `DataFrameList` for the resource. It is sliced positionally per block (`dimtbl[offset + seq_len(nrow(block)), ]`), so it must have exactly one row per streamed row in the same order. A drift is fatal (see Details).
`index_max`	Optional upper bound on the row index (the resource's total row count, or `Inf` when unknown ahead of streaming). Passing it types `__index__` wide enough on part 0 for the whole resource, which is **required** for a > 2^31-row resource and removes any dependence on block size (see Details). `NULL` keeps per-block index narrowing.
`expected_rows`	Optional expected total row count. When given and the streamed total differs, a *warning* is emitted (an unbound junction table can legitimately differ; a bound one usually indicates an off-by-one or a dropped block).
`part_digits`	Zero-padding width for the part index in filenames. The reader globs `*.parquet`, so this is cosmetic; defaults to `6L`.

name	Resource name recorded in the descriptor (default basename(path)).
...	Further arguments forwarded to writeParquet.

Details

Index typing / the narrowing floor. writeParquet types part 0's __index__ column by the index range it sees. Without index_max, part 0 is typed to its own [0, nrow] range, so a small part 0 can pick a narrow integer type (e.g. uint16) that a later part's larger index overflows on append. Passing index_max (the total row count, or Inf) types the index for the whole resource up front and removes this hazard entirely. When index_max is not given and part 0 has fewer than 65536 rows in a multi-part stream, a warning is emitted.

Partition alignment. The per-block dimtbl slice assumes the stream is dense and contiguous with exactly nrow(dimtbl) rows in index order. If the streamed total does not equal nrow(dimtbl) the partition column would be misaligned (silently corrupting directory pruning), so this is a fatal error rather than a warning.

Commit boundary. A multi-part resource is complete only once every part is written. To make "readable only when complete" hold by construction, an _INCOMPLETE marker file is dropped into path immediately before part 0 is written and removed once the stream finishes, so a crash during part 0 is covered too. A finished resource is therefore a clean directory of only parquet parts that the reader loads normally, while an interrupted write leaves the marker behind; DuckDBDataFrame refuses to open a directory carrying the marker, so a read of an incomplete resource fails loudly instead of silently returning the partial parts. A fatal partition-alignment error likewise leaves the marker in place.

Value

Invisibly, the single Frictionless resource descriptor (a list) captured from part 0, with an added n_rows count, ready for writeDatapackage. NULL if the stream yielded no rows.

Author(s)

Patrick Aboyoun

See Also

writeParquet for the single-block writer; writeDatapackage to assemble the resources into a datapackage.json.

Examples

```
tf <- tempfile()
on.exit(unlink(tf, recursive = TRUE))
chunks <- list(data.frame(v = 1:3), data.frame(v = 4:5))
res <- writeStreamingResource(
    function(i) if (i <= length(chunks)) chunks[[i]] else NULL,
    path = file.path(tf, "samples"), dimension = "unbound",
    keycol = NULL, index_max = 5, expected_rows = 5)
res[[1]][["n_rows"]]                  # 5
list.files(file.path(tf, "samples"))  # part-000000.parquet, part-000001.parquet
```

Index