Package: DuckDBSpatial (via r-universe)

July 23, 2026

Version 0.99.00

Date 2026-07-19

Title DuckDB-Backed Spatial Data Structures

Description Provides DuckDB-backed implementations of spatial data structures for efficient out-of-memory operations on large spatial datasets. The package includes 68 sf-compatible spatial methods (st_intersects, st_contains, st_within, etc.) and GeoParquet 1.0 I/O support via writeGeoParquet(). Ideal for spatial analysis, genomic interval analysis, and integration with BiocDuckDB for MultiAssaySpatialExperiment workflows requiring memory-efficient operations on large datasets.

License MIT + file LICENSE

Depends R (>= 4.5.0), methods, DuckDBDataFrame, sf

Imports BiocGenerics, S4Vectors, DBI, dbplyr, dplyr

Suggests knitr, rmarkdown, BiocStyle, testthat, jsonlite, lwgeom, nanoparquet

biocViews DataImport, DataRepresentation, Infrastructure, Software

VignetteBuilder knitr

Encoding UTF-8

Collate 'DuckDBTable-spatial.R' 'DuckDBColumn-spatial.R' 'DuckDBDataFrame-spatial.R' 'coordinate-transform-graph.R' 'coordinate-transforms.R' 'geoparquet-io.R' 'points-coord-index.R' 'writeGeoParquet.R' 'zzz.R'

Config/roxygen2/version 8.0.0

Config/pak/sysreqs libabsl-dev cmake libgdal-dev gdal-bin libgeos-dev libicu-dev libssl-dev libproj-dev libsqlite3-dev libudunits2-dev xz-utils zlib1g-dev

Repository https://biocstaging.r-universe.dev

Date/Publication 2026-07-21 15:15:41 UTC

RemoteUrl https://github.com/BiocStaging/DuckDBSpatial

RemoteRef HEAD

RemoteSha 1018d2fe81246b4a659df93fbb26ee6a724f3454

Contents

`asCoordinateTransform` *Coerce a transform dict into a CoordinateTransform*

Description

Turns an RFC-5 transform dict (a named list with a `type` field, as read from JSON) into a `coordinate-transforms` object, recursing through `sequence` / `byDimension` / `bijection`. A `CoordinateTransform` passes through unchanged.

Usage

```
asCoordinateTransform(x)
```

Arguments

`x`	A `CoordinateTransform` or a transform dict (named list).

Value

A `CoordinateTransform`.

coordinate-transforms *Coordinate-transform constructors (OME-NGFF RFC-5)*

Description

Build the RFC-5 affine-family coordinate transforms. Each returns a `CoordinateTransform` (a classed list mirroring the RFC-5 / SpatialData on-disk dict), consumed by `ctAffineMatrix` / `ctApply` and the coordinate-transform graph (`ctGraph`). Transforms are positional: they act on the coordinate vector in the coordinate system's axis order.

Usage

```
ctIdentity()

ctScale(scale)

ctTranslation(translation)

ctRotation(rotation)

ctAffine(affine)

ctSequence(transformations)

ctMapAxis(map)

ctByDimension(parts)

ctBijection(forward, inverse)
```

Arguments

`scale`, `translation`	Numeric per-axis vectors.
`rotation`, `affine`	A rotation (square) or affine ($n_{out} \times (n_{in}+1)$, RFC-5 non-homogeneous) matrix, as a matrix or list of rows.
`transformations`	A list of `CoordinateTransforms` (applied first to last) for `ctSequence`.
`map`	Integer 0-based axis permutation: output axis `i` is taken from input axis `map[i]`.
`parts`	For `ctByDimension`, a list of `list(transformation=, input_axes=, output_axes=)` (axes 0-based), each sub-transform acting on its own axis subset.
`forward`, `inverse`	`CoordinateTransforms` giving an explicit forward/inverse pair for `ctBijection`.

Value

A `CoordinateTransform`.

Examples

```
ctScale(c(10, 20))
ctSequence(list(ctScale(c(10, 20)), ctTranslation(c(1, 1))))
```

`coordinateSystem` *Define a named coordinate system*

Description

A coordinate system is a name plus an ordered set of axes (RFC-5 `{name, type, unit}`). Coordinate transforms map points between coordinate systems; see `ctGraph`.

Usage

```
coordinateSystem(name, axes)
```

Arguments

`name`	Coordinate-system name.
`axes`	A character vector of axis names, or a list of axis specs each with `name` (and optional `type`, `unit`).

Value

A `CoordinateSystem`.

Examples

```
coordinateSystem("global", c("y", "x"))
```

ctAffineMatrix *Lower a coordinate transform to a homogeneous affine matrix*

Description

Returns the homogeneous $(n_{out} + 1) \times (n_{in} + 1)$ matrix of a coordinate-transforms transform, so that $[out, 1]^T = M\,[in, 1]^T$. Every RFC-5 type reduces to such a matrix; `sequence` composes by matrix multiplication and `byDimension` scatters its per-axis sub-transforms into the block. `bijection` uses its forward transform.

Usage

```
ctAffineMatrix(transform, input_axes, output_axes)
```

Arguments

`transform`	A `CoordinateTransform` (or a transform dict).
`input_axes, output_axes`	Character vectors of the source / target axis names (only their lengths, the dimensionalities, are used).

Value

A numeric homogeneous matrix.

Examples

```
ctAffineMatrix(ctScale(c(10, 20)), c("y", "x"), c("y", "x"))
```

ctApply *Apply a coordinate transform to a matrix of points*

Description

Transforms an $n \times d_{in}$ matrix of coordinates to $n \times d_{out}$ via the transform's homogeneous affine matrix (`ctAffineMatrix`). This is the RFC-5 conformance oracle.

Usage

```
ctApply(transform, points, input_axes, output_axes)
```

Arguments

`transform`	A `CoordinateTransform` (or a transform dict).
`points`	A numeric matrix (rows = points) or a vector (one point).
`input_axes, output_axes`	Character vectors of source / target axis names.

Value

A numeric matrix of transformed points ($n \times d_{out}$).

Examples

```
ctApply(ctTranslation(c(10, 20)), rbind(c(1, 2)), c("y", "x"), c("y", "x"))
```

`ctBetween` *Transform points between two coordinate systems*

Description

Resolves the transform with `ctPath` and applies it to a matrix of points with `ctApply`, using the axis orders recorded for the two coordinate systems in the graph.

Usage

```
ctBetween(graph, from, to, points)
```

Arguments

`graph`	A `CTgraph` built with `systems` (so axis orders are known).
`from`, `to`	Coordinate-system names.
`points`	A numeric matrix (rows = points) or a vector (one point).

Value

A numeric matrix of transformed points.

Examples

```
g <- ctGraph(list(list(input = "a", output = "b",
             transform = ctScale(c(2, 3)))),
             systems = list(a = c("y", "x"), b = c("y", "x")))
ctBetween(g, "a", "b", rbind(c(1, 1)))
```

ctGraph *Build a coordinate-transform graph*

Description

Assembles the directed graph of coordinate systems and the transforms between them. Each invertible edge gets an automatically added reverse edge (via `ctInvert`), so a target reachable by inversion is found by `ctPath`. Mirrors SpatialData's transformation graph.

Usage

```
ctGraph(edges, systems = NULL)
```

Arguments

`edges`	A list of edges, each `list(input=, output=, transform=)` where `input`/`output` are coordinate-system names and `transform` is a `coordinate-transforms` object or dict.
`systems`	Optional named list of coordinate systems (`coordinateSystem` objects or axis-name vectors), keyed by name. Supplies axis orders for `ctBetween` and the set of valid nodes (so a path to/from an undefined system errors).

Value

A `CTgraph`.

Examples

```
g <- ctGraph(list(list(input = "a", output = "b",
             transform = ctScale(c(2, 2)))),
             systems = list(a = c("y", "x"), b = c("y", "x")))
ctPath(g, "a", "b")
```

ctInvert *Invert a coordinate transform*

Description

Returns the inverse `coordinate-transforms` transform: identity is its own inverse; scale inverts elementwise; translation negates; rotation / affine invert by `solve` (errors for a non-invertible or dimensionality-changing affine); mapAxis inverts the permutation; sequence reverses a list of inverses; bijection swaps its declared forward / inverse. Used by `ctGraph` to add reverse edges.

Usage

```
ctInvert(transform)
```

Arguments

transform	A CoordinateTransform (or a transform dict).

Value

The inverse CoordinateTransform.

Examples

```
ctInvert(ctScale(c(10, 20)))
```

---

ctPath *Resolve the transform between two coordinate systems*

Description

Finds the shortest path in a ctGraph from from to to and returns the composed coordinate-transforms transform (a ctSequence of the edge transforms, or the single edge transform for a one-hop path, or ctIdentity() when from == to). Errors if either coordinate system is unknown, if no path exists, or if two distinct shortest paths tie (an ambiguous resolution).

Usage

```
ctPath(graph, from, to)
```

Arguments

graph	A CTgraph.
from, to	Coordinate-system names.

Value

A CoordinateTransform.

Examples

```
g <- ctGraph(list(list(input = "a", output = "b",
             transform = ctScale(c(2, 2)))),
             systems = list(a = c("y", "x"), b = c("y", "x")))
ctPath(g, "b", "a")  # traverses the auto-added inverse edge
```

DuckDBColumn-spatial *Spatial operations on DuckDBColumn objects*

Description

Spatial operations on DuckDBColumn objects.

Value

Method return types are documented in the sections above.

Geometry Creation

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

`st_as_sfc(x, ..., crs = NA_integer_, GeoJSON = FALSE, WKB = FALSE)`: Parses WKT, GeoJSON, or WKB into GEOMETRY.

Geometry Coercion

`st_as_binary(x, hex = FALSE)`: Serialises to WKB or hex-encoded WKB.

`st_as_text(x, geojson = FALSE)`: Serialises to WKT or GeoJSON.

Geometry Accessors

Scalar properties of each geometry.

`st_coordinates(x)`: Returns a DuckDBDataFrame with X, Y (and Z, M when present) coordinate columns.

`st_dimension(x)`: Topological dimension (0, 1, or 2).

`st_end_point(x)`: End point of a linestring.

`st_geometry_type(x)`: Geometry type name as character.

`st_is_closed(x)`: Logical: is the linestring closed (first point = last point)?

`st_is_empty(x)`: Logical: is the geometry empty?

`st_is_ring(x)`: Logical: is the linestring both closed and simple?

`st_is_simple(x)`: Logical: is the geometry simple (e.g. not self-intersecting)?

`st_is_valid(x)`: Logical: is the geometry valid?

`st_num_geometries(x)`: Number of geometries in a geometry collection.

`st_num_interior_rings(x)`: Number of interior rings in a polygon.

`st_num_points(x)`: Number of points within a geometry.

`st_start_point(x)`: Start point of a linestring.

Measurement

`st_area(x)`: Area of each geometry.

`st_distance(x, y)`: Euclidean distance from each geometry to `y`.

`st_length(x)`: Length of linestring or multilinestring; zero for points and polygons.

`st_perimeter(x)`: Perimeter of polygon or multipolygon; zero for points and lines.

Unary Geometry Operations

Each returns a DuckDBColumn with a transformed geometry per row.

`st_boundary(x)`: Geometry boundary.

`st_buffer(x, dist)`: Buffer by `dist` units.

`st_build_area(x)`: Build area of a geometry.

`st_centroid(x)`: Centroid point.

`st_collection_extract(x, type)`: Extract geometries of given type from collections.

`st_concave_hull(x, ratio, allow_holes = FALSE)`: Concave hull.

`st_convex_hull(x)`: Convex hull polygon.

`st_envelope(x)`: Axis-aligned bounding-box polygon.

`st_exterior_ring(x)`: Exterior ring of a polygon.

`st_flip_coordinates(x)`: Flip the coordinates of a geometry.

`st_inscribed_circle(x, dTolerance, ..., nQuadSegs = 30)`: Maximum inscribed circle of polygon.

`st_line_interpolate(line, dist, normalized = FALSE)`: Point at distance `dist` along `line`.

`st_line_merge(x, directed = FALSE)`: Merge connected line segments.

`st_line_project(line, point, normalized = FALSE)`: Distance or fraction along `line` to nearest point.

`st_line_substring(line, start, end)`: Substring of `line` between `start` and `end` fractions.

`st_make_valid(x)`: Repair invalid geometry.

`st_minimum_rotated_rectangle(x)`: Minimum-area rotated bounding rectangle.

`st_node(x)`: Add nodes at intersection points for line geometry.

`st_normalize(x)`: Normalise vertex order.

`st_point_on_surface(x)`: A point guaranteed to lie on the surface.

`st_reverse(x)`: Reverse vertex order.

`st_reduce_precision(x, precision)`: Reduce precision of a geometry.

`st_remove_repeated_points(x)`: Remove repeated points from a geometry.

`st_simplify(x, preserveTopology = FALSE, dTolerance = 0.0)`: Simplify geometry.

`st_voronoi(x)`: Voronoi diagram.

Binary Spatial Predicates

Each takes a second geometry argument y and returns a BOOLEAN DuckDBColumn. See DuckDBTable-spatial for accepted y types.

`st_contains(x, y)`: Does x contain y?

`st_contains_properly(x, y)`: Does x properly contain y (boundary excluded)?

`st_covered_by(x, y)`: Is x covered by y?

`st_covers(x, y)`: Does x cover y?

`st_crosses(x, y)`: Does x cross y?

`st_disjoint(x, y)`: Are x and y disjoint?

`st_equals(x, y)`: Are x and y geometrically equal?

`st_intersects(x, y)`: Do x and y intersect?

`st_is_within_distance(x, y, dist)`: Is x within `dist` of y?

`st_overlaps(x, y)`: Does x overlap y?

`st_touches(x, y)`: Does x touch y?

`st_within(x, y)`: Is x within y?

`st_within_properly(x, y)`: Is x properly within y (boundary excluded)?

Binary Set Operations

`st_difference(x, y)`: Portion of x that does not intersect y.

`st_intersection(x, y)`: Portion of x that intersects y.

`st_nearest_points(x, y)`: Shortest line between x and y.

`st_union(x, y)`: Binary: returns a DuckDBColumn. Aggregate (no y): Apply `ST_Union_Agg` and materialise to `sfc`.

Aggregate Operations

These methods materialise and return concrete R objects.

`st_bbox(obj)`: Returns a `bbox` named numeric vector `c(xmin, ymin, xmax, ymax)` via `MIN/MAX` of `ST_XMin/ST_YMin/ST_XMax/ST_YMax`.

`st_collect(x)`: Returns an `sfc` GEOMETRYCOLLECTION via `ST_Collect`.

`st_union(x)` **(no** y**)**: Returns an `sfc` via `ST_Union_Agg`.

Author(s)

Patrick Aboyoun

See Also

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

Examples

```
spatial_path <- system.file("extdata", "spatial", package = "DuckDBSpatial")
df <- DuckDBDataFrame(spatial_path)
df <- df[which(!is.na(df$type)), ]
geom <- df[["geometry"]]
st_geometry_type(geom)[1:3]
st_area(geom)[1:3]
st_intersects(geom, sf::st_sfc(sf::st_point(c(30, 10))))[1:3]
```

DuckDBTable-spatial *Spatial operations on DuckDBTable objects*

Description

Spatial operations on DuckDBTable objects.

Usage

```
st_make_envelope_sql(xmin, ymin, xmax, ymax)

st_geomfromtext_sql(wkt)

st_point_sql(x_col, y_col)

st_intersects_table(x, y, sparse = TRUE, ...)

## S3 method for class 'DuckDBTable'
st_filter(x, y, ..., .predicate = st_intersects)

## S3 method for class 'DuckDBTable'
st_join(x, y, join = st_intersects, ...)

st_union_agg(x, by = NULL, ...)

st_collect_agg(x, by = NULL, ...)

st_envelope_agg(x, by = NULL, ...)
```

Arguments

xmin, ymin, xmax, ymax	Bounding box limits.
wkt	A WKT character string.
x_col, y_col	Column names for x and y coordinates.
x	A DuckDBColumn or DuckDBTable geometry column.

y	A DuckDBTable or DuckDBDataFrame to join against.
sparse	Logical; return a lazy table when TRUE.
...	Ignored.
.predicate	Spatial predicate function (default st_intersects).
join	Spatial predicate function (default st_intersects).
by	Optional grouping column name(s).

Value

Method return types are documented in the sections above.

Geometry Creation

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

st_as_sfc(x, ..., crs = NA_integer_, GeoJSON = FALSE, WKB = FALSE): Parses WKT (default), GeoJSON, or WKB into GEOMETRY via ST_GeomFromText, ST_GeomFromGeoJSON, or ST_GeomFromWKB.

Geometry Coercion

st_as_binary(x, hex = FALSE): Converts to WKB (ST_AsWKB) or hex-encoded WKB (ST_AsHEXWKB).

st_as_text(x, geojson = FALSE): Converts to WKT (ST_AsText) or GeoJSON (ST_AsGeoJSON).

Geometry Accessors

Scalar properties of each geometry.

st_dimension(x): Topological dimension: 0 (point), 1 (line), or 2 (polygon).

st_end_point(x): End point of a linestring.

st_geometry_type(x): Geometry type name as character (e.g. "POINT", "POLYGON").

st_is_closed(x): Logical: is the linestring closed (first point = last point)?

st_is_empty(x): Logical: is the geometry empty?

st_is_ring(x): Logical: is the linestring both closed and simple?

st_is_simple(x): Logical: is the geometry simple (e.g. not self-intersecting)?

st_is_valid(x): Logical: is the geometry valid?

st_num_geometries(x): Number of geometries in a geometry collection.

st_num_interior_rings(x): Number of interior rings in a polygon.

st_num_points(x): Number of points within a geometry.

st_start_point(x): Start point of a linestring.

Measurement

st_area(x): Area of each geometry.

st_distance(x, y): Euclidean distance from each geometry in x to y. Argument y accepts WKT character, sfg, sfc (length 1), or call.

st_length(x): Length of linestring or multilinestring; zero for points and polygons.

st_perimeter(x): Perimeter of polygon or multipolygon; zero for points and lines.

Unary Geometry Operations

Each returns a DuckDBTable with a transformed geometry per row.

`st_boundary(x)`: Geometry boundary.

`st_buffer(x, dist)`: Buffer by `dist` units.

`st_build_area(x)`: Build area of a geometry.

`st_centroid(x)`: Centroid point.

`st_collection_extract(x, type = c("POLYGON", "POINT", "LINESTRING"))`: Extract geometries of given type from collections.

`st_concave_hull(x, ratio, allow_holes = FALSE)`: Concave hull; `ratio` in [0,1] (1 = convex).

`st_convex_hull(x)`: Convex hull polygon.

`st_envelope(x)`: Axis-aligned bounding-box polygon.

`st_exterior_ring(x)`: Exterior ring of a polygon.

`st_flip_coordinates(x)`: Flip the coordinates of a geometry.

`st_inscribed_circle(x, dTolerance, ..., nQuadSegs = 30)`: Maximum inscribed circle of polygon; `dTolerance` required.

`st_line_interpolate(line, dist, normalized = FALSE)`: Point at distance `dist` along `line`; when `normalized = FALSE`, `dist` is in line units; when `TRUE`, `dist` is fraction in [0,1].

`st_line_merge(x, directed = FALSE)`: Merge connected line segments.

`st_line_project(line, point, normalized = FALSE)`: Distance or fraction along `line` to nearest point to `point`; `normalized` controls units vs fraction.

`st_line_substring(line, start, end)`: Substring of `line` between `start` and `end` fractions.

`st_make_valid(x)`: Repair invalid geometry.

`st_minimum_rotated_rectangle(x)`: Minimum-area rotated bounding rectangle.

`st_node(x)`: Add nodes at intersection points (noding) for line geometry.

`st_normalize(x)`: Normalise vertex order.

`st_point_on_surface(x)`: A point guaranteed to lie on the surface.

`st_reduce_precision(x, precision)`: Reduce precision of a geometry.

`st_remove_repeated_points(x)`: Remove repeated points from a geometry.

`st_reverse(x)`: Reverse vertex order.

`st_simplify(x, preserveTopology = FALSE, dTolerance = 0.0)`: Simplify geometry. Uses `ST_SimplifyPreserveTopo` when `preserveTopology = TRUE`.

`st_voronoi(x)`: Voronoi diagram of points (DuckDB: no envelope/tolerance).

Binary Spatial Predicates

Each takes a second geometry argument `y` and returns a BOOLEAN DuckDBTable. Argument `y` accepts a WKT character string, an `sfc`/`sfg` object from **sf**, or a `call` (raw SQL expression). Results are usable for row filtering via `extractROWS` or `subset`.

`st_contains(x, y)`: Does `x` contain `y`?

`st_contains_properly(x, y)`: Does `x` properly contain `y` (boundary excluded)?

`st_covered_by(x, y)`: Is `x` covered by `y`?

`st_covers(x, y)`: Does `x` cover `y`?

`st_crosses(x, y)`: Does `x` cross `y`?

`st_disjoint(x, y)`: Are `x` and `y` disjoint?

`st_equals(x, y)`: Are `x` and `y` geometrically equal?

`st_intersects(x, y)`: Do `x` and `y` intersect?

`st_is_within_distance(x, y, dist)`: Is `x` within `dist` of `y`?

`st_overlaps(x, y)`: Does `x` overlap `y`?

`st_touches(x, y)`: Does `x` touch `y`?

`st_within(x, y)`: Is `x` within `y`?

`st_within_properly(x, y)`: Is `x` properly within `y` (boundary excluded)?

Binary Set Operations

Each takes a second geometry argument `y` (same accepted types as predicates) and returns a GEOMETRY DuckDBTable.

`st_difference(x, y)`: Portion of `x` that does not intersect `y`.

`st_intersection(x, y)`: Portion of `x` that intersects `y`.

`st_nearest_points(x, y)`: Shortest line between `x` and `y` (DuckDB `ST_ShortestLine`).

`st_union(x, y)`: Union of `x` and `y`. Binary form only at the DuckDBTable level; aggregate union is available on DuckDBColumn.

Aggregate Operations

`st_collect(x)`: Combine all geometries into a single collection.

`st_union_agg(x, by = NULL)`: Aggregate union over a geometry column via `ST_Union_Agg`.

`st_collect_agg(x, by = NULL)`: Aggregate collect over a geometry column via `ST_Collect`.

`st_envelope_agg(x, by = NULL)`: Aggregate envelope over a geometry column via `ST_Envelope_Agg`.

Table Filter, Join, and Sparse Intersects

`st_filter(x, y, .predicate = st_intersects)`: Return rows of `x` that spatially match `y`.

`st_join(x, y, join = st_intersects)`: Spatial inner join of two lazy tables on `geometry`.

`st_intersects_table(x, y, sparse = TRUE)`: Return a sparse index table of intersecting row pairs.

SQL Geometry Helpers

Low-level helpers that return `dplyr::sql` expressions for use in lazy queries.

`st_make_envelope_sql(xmin, ymin, xmax, ymax)`: Bounding-box envelope expression via `ST_MakeEnvelope`.

`st_point_sql(x_col, y_col)`: Point expression from column names via `ST_Point`.

`st_geomfromtext_sql(wkt)`: Geometry expression from WKT via `ST_GeomFromText`.

Author(s)

Patrick Aboyoun

See Also

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

Examples

```
spatial_path <- system.file("extdata", "spatial", package = "DuckDBSpatial")
df <- DuckDBDataFrame(spatial_path)
df <- df[which(!is.na(df$type)), ]
query_pt <- sf::st_sfc(sf::st_point(c(30, 10)))
filtered <- st_filter(df, query_pt)
nrow(filtered)
st_make_envelope_sql(0, 0, 100, 100)
```

---

enableGeoParquetConversion

Enable DuckDB GeoParquet conversion on a connection

Description

Sets `enable_geoparquet_conversion = true` so geometry columns in Parquet files are read as native `GEOMETRY` types.

Usage

```
enableGeoParquetConversion(conn = NULL)
```

Arguments

conn	A DuckDB DBI connection. When `NULL`, uses `acquireDuckDBConn()`.

Value

The connection, invisibly.

Examples

```
enableGeoParquetConversion()
```

`layerBboxRange` *Subset a points layer to a bounding box with a prunable range predicate*

Description

Returns the rows of point layer `x` inside `[xmin, xmax]` x `[ymin, ymax]` using a plain coordinate range predicate (`x_col >= xmin & x_col <= xmax & y_col >= ymin & y_col <= ymax`). Unlike `layerSubsetByBbox` (which builds a per-row `ST_Point` and tests `ST_Intersects` against an envelope, and returns row indices), the range predicate pushes into DuckDB's Parquet reader, so on a Morton-sorted (coord-indexed) points file (see `writeSpatialPointsParquet`) it prunes row groups outside the box. The result is a **lazy** `DuckDBDataFrame` (nothing is materialized until collected), the R counterpart of scibis' `query_points`.

Usage

```
layerBboxRange(
  x,
  xmin,
  xmax,
  ymin,
  ymax,
  x_col = "x",
  y_col = "y",
  zmin = NULL,
  zmax = NULL,
  z_col = "z"
)
```

Arguments

`x`	A DuckDBDataFrame point layer.
`xmin, xmax, ymin, ymax`	Bounding-box limits (inclusive).
`x_col, y_col`	Coordinate column names (default `"x"/"y"`).
`zmin, zmax`	Optional inclusive depth range for a 3-D viewport; when both are non-`NULL` a `z_col BETWEEN zmin AND zmax` term is ANDed on (and pushes into the scan too, so a 3-D layout prunes on `z` as well).
`z_col`	Depth column name (default `"z"`), used only with `zmin`/`zmax`.

Value

A lazy `DuckDBDataFrame` filtered to the box.

Examples

```
p <- tempfile(fileext = ".parquet")
writeSpatialPointsParquet(
    data.frame(x = c(1, 5, 50), y = c(1, 5, 50), gene = c("A", "B", "C")), p)
pts <- DuckDBDataFrame::DuckDBDataFrame(p)
as.data.frame(layerBboxRange(pts, 0, 10, 0, 10))
unlink(p)
```

`layerSpatialMatch` *Match points in a lazy table layer to a geometry table*

Description

Match points in a lazy table layer to a geometry table

Usage

```
layerSpatialMatch(x, table, coords, geom = "geometry", join = NULL)
```

Arguments

`x`	A `DuckDBDataFrame` point layer.
`table`	A `DuckDBDataFrame` or in-memory `DataFrame` with geometries.
`coords`	Length-two character vector of x/y column names in `x`.
`geom`	Geometry column name in `table`.
`join`	Optional `sf` predicate (default `sf::st_intersects`).

Value

Integer vector of match indices (NA where unmatched).

Examples

```
spatial_path <- system.file("extdata", "spatial", package = "DuckDBSpatial")
shapes <- DuckDBDataFrame(spatial_path)
shapes <- shapes[which(!is.na(shapes$type)), ]
pts_path <- tempfile(fileext = ".csv")
on.exit(unlink(pts_path), add = TRUE)
write.csv(data.frame(x = c(1, 5, 30), y = c(1, 5, 10)), pts_path, row.names = FALSE)
pts <- DuckDBDataFrame(pts_path, datacols = c("x", "y"))
layerSpatialMatch(pts, shapes, coords = c("x", "y"))
```

layerSpatialOverlaps *Test spatial intersection for a lazy table layer*

Description

Layer-level intersection predicate for DuckDBDataFrame objects. Accepts coordinate columns or a geometry column.

Usage

```
layerSpatialOverlaps(x, y, coords = NULL, geom = "geometry")
```

Arguments

x	A DuckDBDataFrame layer.
y	A geometry (sfc, sfg), WKT character, or dplyr::sql expression.
coords	Optional length-two character vector of x/y column names.
geom	Geometry column name when coords is NULL.

Value

Logical vector of length nrow(x).

Examples

```
pts_path <- tempfile(fileext = ".csv")
on.exit(unlink(pts_path), add = TRUE)
write.csv(data.frame(x = c(1, 5, 10), y = c(1, 5, 10)), pts_path, row.names = FALSE)
pts <- DuckDBDataFrame(pts_path, datacols = c("x", "y"))
poly <- sf::st_as_sfc("POLYGON((0 0, 6 0, 6 6, 0 6, 0 0))")
layerSpatialOverlaps(pts, poly, coords = c("x", "y"))
```

layerSubsetByBbox *Row indices of a layer inside a bounding box*

Description

Row indices of a layer inside a bounding box

Usage

```
layerSubsetByBbox(
  x,
  xmin,
  xmax,
  ymin,
  ymax,
  x_col = "x",
  y_col = "y",
  geom = "geometry"
)
```

Arguments

x	A DuckDBDataFrame or in-memory tabular layer.
xmin, xmax, ymin, ymax	Bounding box limits.
x_col, y_col	Coordinate column names for point layers.
geom	Geometry column name for polygon layers.

Value

Integer row indices.

Examples

```
pts_path <- tempfile(fileext = ".csv")
on.exit(unlink(pts_path), add = TRUE)
write.csv(data.frame(x = c(1, 5, 10), y = c(1, 5, 10)), pts_path, row.names = FALSE)
pts <- DuckDBDataFrame(pts_path, datacols = c("x", "y"))
layerSubsetByBbox(pts, 0, 10, 0, 10, x_col = "x", y_col = "y")
```

---

layerSubsetByGeometry *Row indices of a layer intersecting a geometry*

Description

Row indices of a layer intersecting a geometry

Usage

```
layerSubsetByGeometry(x, y, coords = NULL, geom = "geometry")
```

Arguments

x	A DuckDBDataFrame or in-memory tabular layer.
y	A geometry to test against.
coords	Optional x/y column names for point layers.
geom	Geometry column name for polygon layers.

Value

Integer row indices.

Examples

```
pts_path <- tempfile(fileext = ".csv")
on.exit(unlink(pts_path), add = TRUE)
write.csv(data.frame(x = c(1, 5, 10), y = c(1, 5, 10)), pts_path, row.names = FALSE)
pts <- DuckDBDataFrame(pts_path, datacols = c("x", "y"))
poly <- sf::st_as_sfc("POLYGON((0 0, 6 0, 6 6, 0 6, 0 0))")
layerSubsetByGeometry(pts, poly, coords = c("x", "y"))
```

---

readGeoParquet *Read a GeoParquet file as a lazy* DuckDBDataFrame

---

Description

Read a GeoParquet file as a lazy DuckDBDataFrame

Usage

```
readGeoParquet(path, ...)
```

Arguments

path	Path to a GeoParquet file or directory.
...	Passed to DuckDBDataFrame.

Value

A DuckDBDataFrame.

Examples

```
spatial_path <- system.file("extdata", "spatial", package = "DuckDBSpatial")
ddb <- readGeoParquet(spatial_path)
nrow(ddb)
colnames(ddb)
```

`spatialSortPoints` *Reorder a points table by a Morton (Z-order) spatial key*

Description

Returns `df` with its rows reordered by the Morton code of `(x_col, y_col)` (or `(x_col, y_col, z_col)` when `z_col` is given and present), so that spatially neighboring points become contiguous. Writing the reordered table to Parquet gives each row group a tight coordinate zonemap, which lets DuckDB prune groups outside a viewport query (see `writeSpatialPointsParquet` and `layerBboxRange`). A pure row permutation: no column is added, so it round-trips identically. A no-op (returns `df` unchanged) when `df` is empty or lacks a required coordinate column.

Usage

```
spatialSortPoints(df, x_col = "x", y_col = "y", z_col = NULL, bits = 16L)
```

Arguments

`df`	A `data.frame` or DataFrame of points.
`x_col, y_col`	Coordinate column names (default `"x"/"y"`).
`z_col`	Optional third coordinate column for 3-D clustering; used only when non-`NULL` and present in `df` (default `NULL`, 2-D).
`bits`	Grid resolution per axis (default 16, a 2^{16} grid).

Details

This is the low-dimensional spatial convenience wrapper over `clusterSort(df, zorder(c(x_col, y_col)))`; the Morton math lives in DuckDBDataFrame and is already N-D, so `(x, y, z)` is a genuine 3-D Morton curve, not a 2-D curve with `z` appended.

Value

`df` reordered by Morton code (same class, same columns).

Examples

```
df <- data.frame(x = c(9, 1, 5), y = c(9, 1, 5), gene = c("C", "A", "B"))
spatialSortPoints(df)
```

st_affine *Apply a 2-D affine coordinate transform to geometries*

Description

Applies a 2-D affine map to a geometry column via DuckDB `ST_Affine`: $x' = ax+by+xoff, y' = dx+ey+yoff$. The affine may be a `coordinate-transforms` object (e.g. `ctScale`, `ctRotation`, `ctSequence`), lowered to its 2-D coefficients, or a numeric matrix (`2x2` linear, `2x3` linear+offset, or `3x3` homogeneous). Errors if the transform is not 2-D-expressible (e.g. a dimensionality change). For point layers stored as `x`/`y` columns rather than a geometry column, use `transformLayer`.

Usage

```
st_affine(x, affine, ...)
```

Arguments

`x`	A `DuckDBTable` / `DuckDBColumn` geometry, or an `sf` object (default method).
`affine`	A `CoordinateTransform` or affine matrix.
`...`	Ignored.

Value

An object of the same class as `x` with transformed geometries.

See Also

`coordinate-transforms`, `transformLayer`

transformLayer *Apply a coordinate transform to a spatial layer*

Description

Transforms a spatial layer by a 2-D affine coordinate transform, in place and lazily (nothing is materialized until collected). `transform` is a `coordinate-transforms` object (`ctScale`, `ctTranslation`, `ctRotation`, `ctAffine`, `ctSequence`, ...) or an affine matrix, lowered to a 2-D affine ($x' = ax + by + xoff$, $y' = dx + ey + yoff$). A point layer (columns `x_col`/ `y_col`) is transformed by arithmetic on those columns; a geometry layer (column `geom`) by `st_affine` (DuckDB `ST_Affine`). To move a layer between named coordinate systems, resolve the transform with `ctPath` first and pass it here.

Usage

```
transformLayer(x, transform, x_col = "x", y_col = "y", geom = "geometry")
```

Arguments

x	A DuckDBDataFrame layer.
transform	A CoordinateTransform or affine matrix (must be 2-D-expressible; a dimensionality change errors).
x_col, y_col	Coordinate column names for point layers (default "x"/"y"); set x_col = NULL to force the geometry path.
geom	Geometry column name for geometry layers (default "geometry").

Value

A lazy DuckDBDataFrame with transformed coordinates.

See Also

coordinate-transforms, ctGraph, st_affine

Examples

```
p <- tempfile(fileext = ".parquet")
writeSpatialPointsParquet(data.frame(x = c(1, 2, 3), y = c(0, 0, 0)), p)
pts <- DuckDBDataFrame::DuckDBDataFrame(p)
# rotate 90 degrees about the origin: (x, y) -> (-y, x)
as.data.frame(transformLayer(pts, ctRotation(rbind(c(0, -1), c(1, 0)))))
unlink(p)
```

---

writeGeoParquet *Write spatial data as GeoParquet*

Description

Writes sf objects or data frames with geometry columns as GeoParquet files conforming to the GeoParquet 1.0.0 specification. The geometry column is encoded as Well-Known Binary (WKB) and appropriate metadata is added to ensure compatibility with DuckDB's spatial extension, GeoPandas, and other GeoParquet readers.

Usage

```
writeGeoParquet(
  x,
  path,
  geom = "geometry",
  compression = "zstd",
  options = nanoparquet::parquet_options(compression_level = 3L),
  ...
)
```

Arguments

x	An sf object or a data frame with a geometry column (either sfc or a list of raw WKB vectors).
path	Character string specifying the output file path. Should end in .parquet.
geom	Character string specifying the name of the geometry column. Defaults to "geometry". For sf objects, this is automatically detected if not provided.
compression	Character string specifying the compression algorithm. Defaults to "zstd". Other options include "snappy", "gzip", "brotli", or "uncompressed".
options	Optional list of parquet options created by nanoparquet::parquet_options(). Defaults to compression level 3.
...	Additional arguments (currently unused).

Details

The function performs the following steps:

1. Converts the geometry column to Well-Known Binary (WKB) format
2. Generates GeoParquet 1.0.0 metadata including:
   - Geometry type(s) (Point, LineString, Polygon, etc.)
   - Bounding box (xmin, ymin, xmax, ymax)
   - Encoding format (WKB)
   - Primary geometry column name
3. Writes the data to a Parquet file using **nanoparquet**

The resulting files can be read by:

- DuckDB: `SELECT ST_AsText(geometry) FROM 'file.parquet'`
- GeoPandas: `gpd.read_parquet('file.parquet')`
- GDAL/OGR: `ogr2ogr -f GeoJSON output.json file.parquet`
- **sf**: `sf::st_read('file.parquet')`

Value

Invisibly returns NULL. Called for its side effect of writing a GeoParquet file to disk.

See Also

- https://geoparquet.org/ - GeoParquet specification
- write_parquet - Underlying Parquet writer
- st_write - Alternative sf-based writer

Examples

```
## Not run:
library(sf)

# Write a simple features object
nc <- st_read(system.file("shape/nc.shp", package="sf"))
writeGeoParquet(nc, "nc.parquet")

# Write with custom geometry column name
pts <- st_sf(id = 1:3, geom = st_sfc(st_point(c(0,0)),
                                     st_point(c(1,1)),
                                     st_point(c(2,2))))
writeGeoParquet(pts, "points.parquet", geom = "geom")

# Read back in DuckDB
library(DuckDBDataFrame)
ddb <- DuckDBDataFrame("nc.parquet")
ddb$geometry  # Native GEOMETRY column

## End(Not run)
```

`writeSpatialPointsParquet` *Write a coord-indexed (Morton-sorted) points Parquet*

Description

Writes a points table to a single Parquet file, Morton-sorting it first (by default) so viewport queries prune row groups. The write goes through the shared DuckDBDataFrame connection and `buildParquetCopySQL` with a bounded `ROW_GROUP_SIZE` (so there are groups to prune). With `spatial_sort = FALSE` the acquisition-order baseline is written instead.

Usage

```
writeSpatialPointsParquet(
  df,
  path,
  x_col = "x",
  y_col = "y",
  z_col = NULL,
  spatial_sort = TRUE,
  row_group_size = 131072L,
  bits = 16L
)
```

Arguments

df	A data.frame / DataFrame of points (must contain x_col and y_col; any other columns, e.g. gene, are carried through).
path	Output Parquet file path.
x_col, y_col	Coordinate column names (default "x"/"y").
z_col	Optional third coordinate column for 3-D Morton clustering. Default NULL auto-detects a "z" column (so 3-D points cluster in 3-D automatically, matching scibis); pass a name to override or NA to force 2-D.
spatial_sort	If TRUE (default), Morton-sort before writing (coord-indexed layout); if FALSE, write in input order (baseline).
row_group_size	Parquet ROW_GROUP_SIZE (default 131072); smaller groups give finer pruning at a small metadata cost.
bits	Morton grid resolution per axis (default 16).

Value

path, invisibly.

Examples

```
df <- data.frame(x = runif(1000, 0, 100), y = runif(1000, 0, 100))
p <- tempfile(fileext = ".parquet")
writeSpatialPointsParquet(df, p)
unlink(p)
```

Index