Package: OmicsLake (via r-universe)

July 15, 2026

Title OmicsLake: Versioned, On-Disk Omics Data Management for Bioconductor

Version 0.99.3

Author Yusuke Matsui [aut, cre]

Maintainer Yusuke Matsui <mail.to.matsui@gmail.com>

Description A lightweight framework for versioned, on-disk omics data management using DuckDB, Arrow, and Parquet. It provides read/write-style functions, snapshots, and version-aware dataset lineage for tracked or explicitly annotated workflows. Adapters preserve supported Bioconductor containers, including SummarizedExperiment and MultiAssayExperiment, while retaining queryable tabular components.

License MIT + file LICENSE

Encoding UTF-8

Depends R (>= 4.5.0)

Imports jsonlite, arrow, dplyr, dbplyr, DBI, duckdb, Matrix, SummarizedExperiment, S4Vectors, R6, rlang, tools, methods, withr

Suggests MultiAssayExperiment, HDF5Array, DelayedArray, knitr, rmarkdown, testthat (>= 3.0.0), igraph, yaml, bench, grDevices, graphics, stats, utils, SingleCellExperiment, GenomicRanges, Spectra, MsExperiment, QFeatures, xcms, Chromatograms, renv, tibble, pkgdown

VignetteBuilder knitr

biocViews Infrastructure, DataImport, DataRepresentation

URL https://github.com/matsui-lab/OmicsLake

BugReports https://github.com/matsui-lab/OmicsLake/issues

RoxygenNote 7.3.3

NeedsCompilation no

Config/pak/sysreqs cmake libicu-dev libssl-dev xz-utils zlib1g-dev

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

Date/Publication 2026-07-15 13:42:39 UTC

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

RemoteRef HEAD

RemoteSha 0d2b2fc7793e889f82a54deb6964649404a66bb6

Contents

`OmicsLake-package` *OmicsLake: OmicsLake: Versioned, On-Disk Omics Data Management for Bioconductor*

Description

A lightweight framework for versioned, on-disk omics data management using DuckDB + Arrow + Parquet. Provides simple read/write-style functions and Bioconductor-compatible readers (SummarizedExperiment, MultiAssayExperiment).

Author(s)

Maintainer: Yusuke Matsui <mail.to.matsui@gmail.com> (ORCID)

See Also

Useful links:

- https://github.com/matsui-lab/OmicsLake
- Report bugs at https://github.com/matsui-lab/OmicsLake/issues

---

`%between%` *BETWEEN operator for range filtering*

Description

BETWEEN operator for range filtering

Usage

```
x %between% range
```

Arguments

`x`	Numeric vector
`range`	Numeric vector of length 2 (min, max), inclusive

Value

Logical vector

Examples

```
values <- 1:20
values[values %between% c(5, 15)]
```

`%>>%` *Pipe operator that saves to lake*

Description

Use this operator at the end of a pipe to save results to a lake. Format: data

Usage

```
.data %>>% target
```

Arguments

`.data`	Data from pipe
`target`	Target specification ("name" or "project/name")

Value

Invisibly returns the data

Examples

```
if (FALSE) {
use_lake("my_project")

df |>
  dplyr::filter(x > 5) %>>%
  "filtered_data"
}
```

%ilike% *Case-insensitive LIKE operator*

Description

Case-insensitive LIKE operator

Usage

```
x %ilike% pattern
```

Arguments

x	Character vector to match against
pattern	Pattern string (SQL LIKE syntax)

Value

Logical vector

Examples

```
names <- c("John", "JOHN", "johnny", "Jane")
names[names %ilike% "john%"]
```

%iregex% *Case-insensitive regex match*

Description

Case-insensitive regex match

Usage

```
x %iregex% pattern
```

Arguments

x	Character vector
pattern	Regular expression pattern

Value

Logical vector

Examples

```
genes <- c("ENSG001", "ensg002", "MT-CO1")
genes[genes %iregex% "^ensg"]
```

`%like%` *LIKE operator for SQL-style pattern matching*

Description

Matches patterns using SQL LIKE syntax where: - ‘ - ‘_‘ matches any single character

Usage

```
x %like% pattern
```

Arguments

`x`	Character vector to match against
`pattern`	Pattern string (SQL LIKE syntax)

Value

Logical vector

Examples

```
genes <- c("MT-CO1", "MT-CO2", "ACTB", "GAPDH")
genes[genes %like% "MT-%"]
```

`%regex%` *Regex match operator*

Description

Regex match operator

Usage

```
x %regex% pattern
```

Arguments

`x`	Character vector
`pattern`	Regular expression pattern

Value

Logical vector

Examples

```
genes <- c("ENSG00000001", "ENSG00000002", "MT-CO1")
genes[genes %regex% "^ENSG"]
```

`ATACAdapter` *ATAC Adapter*

Description

Adapter for storing and retrieving ATAC-layer objects.

Details

Supports explicit ATAC marker classes/metadata and ChromatinAssay objects.

Value

An R6 generator for a `LakeAdapter` subclass that serializes and restores objects of this omics layer.

Super class

`OmicsLake::LakeAdapter` -> `ATACAdapter`

Methods

Public methods:

- `ATACAdapter$name()`
- `ATACAdapter$priority()`
- `ATACAdapter$put()`
- `ATACAdapter$get()`
- `ATACAdapter$components()`
- `ATACAdapter$exists()`
- `ATACAdapter$list_names()`
- `ATACAdapter$can_handle()`
- `ATACAdapter$clone()`

Method `name()`:

Usage:

```
ATACAdapter$name()
```

Method `priority()`:

Usage:

```
ATACAdapter$priority()
```

Method `put()`**:**

Usage:

```
ATACAdapter$put(lake, name, data)
```

Method `get()`**:**

Usage:

```
ATACAdapter$get(lake, name, ref = "@latest")
```

Method `components()`**:**

Usage:

```
ATACAdapter$components(lake, name)
```

Method `exists()`**:**

Usage:

```
ATACAdapter$exists(lake, name)
```

Method `list_names()`**:**

Usage:

```
ATACAdapter$list_names(lake)
```

Method `can_handle()`**:**

Usage:

```
ATACAdapter$can_handle(data)
```

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
ATACAdapter$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
adapter <- ATACAdapter$new()
class(adapter)
```

ChIPAdapter *ChIP Adapter*

Description

Adapter for storing and retrieving ChIP-layer objects.

Details

Supports explicit ChIP marker classes/metadata.

Value

An R6 generator for a `LakeAdapter` subclass that serializes and restores objects of this omics layer.

Super class

`OmicsLake::LakeAdapter` -> `ChIPAdapter`

Methods

Public methods:

- `ChIPAdapter$name()`
- `ChIPAdapter$priority()`
- `ChIPAdapter$put()`
- `ChIPAdapter$get()`
- `ChIPAdapter$components()`
- `ChIPAdapter$exists()`
- `ChIPAdapter$list_names()`
- `ChIPAdapter$can_handle()`
- `ChIPAdapter$clone()`

Method `name()`:

Usage:

```
ChIPAdapter$name()
```

Method `priority()`:

Usage:

```
ChIPAdapter$priority()
```

Method `put()`:

Usage:

```
ChIPAdapter$put(lake, name, data)
```

Method `get()`:

Usage:

```
ChIPAdapter$get(lake, name, ref = "@latest")
```

Method `components()`**:**

Usage:

```
ChIPAdapter$components(lake, name)
```

Method `exists()`**:**

Usage:

```
ChIPAdapter$exists(lake, name)
```

Method `list_names()`**:**

Usage:

```
ChIPAdapter$list_names(lake)
```

Method `can_handle()`**:**

Usage:

```
ChIPAdapter$can_handle(data)
```

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
ChIPAdapter$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
adapter <- ChIPAdapter$new()
class(adapter)
```

`ChromatogramsAdapter` *Chromatograms Adapter*

Description

Adapter for storing and retrieving `Chromatograms` objects.

Details

This adapter currently guarantees full-fidelity roundtrip by storing the complete object and manifest as internal components.

Value

An R6 generator for a `LakeAdapter` subclass that serializes and restores objects of this omics layer.

Super class

OmicsLake::LakeAdapter -> ChromatogramsAdapter

Methods

Public methods:

- ChromatogramsAdapter$name()
- ChromatogramsAdapter$can_handle()
- ChromatogramsAdapter$priority()
- ChromatogramsAdapter$put()
- ChromatogramsAdapter$get()
- ChromatogramsAdapter$components()
- ChromatogramsAdapter$exists()
- ChromatogramsAdapter$list_names()
- ChromatogramsAdapter$clone()

Method name():

Usage:

```
ChromatogramsAdapter$name()
```

Method can_handle():

Usage:

```
ChromatogramsAdapter$can_handle(data)
```

Method priority():

Usage:

```
ChromatogramsAdapter$priority()
```

Method put():

Usage:

```
ChromatogramsAdapter$put(lake, name, data)
```

Method get():

Usage:

```
ChromatogramsAdapter$get(lake, name, ref = "@latest")
```

Method components():

Usage:

```
ChromatogramsAdapter$components(lake, name)
```

Method exists():

Usage:

```
ChromatogramsAdapter$exists(lake, name)
```

Method list_names():

Usage:

```
ChromatogramsAdapter$list_names(lake)
```

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
ChromatogramsAdapter$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
adapter <- ChromatogramsAdapter$new()
class(adapter)
```

`coalesce` *Coalesce - return first non-NA value*

Description

Thin wrapper around `dplyr::coalesce()` re-exported for convenience. Note: this will mask `dplyr::coalesce` when both packages are loaded.

Usage

```
coalesce(...)
```

Arguments

`...`	Vectors to coalesce

Value

Vector with first non-NA values

Examples

```
x <- c(NA, 2, NA)
y <- c(1, NA, 3)
coalesce(x, y) # Returns c(1, 2, 3)
```

contains_str *Check if string contains a substring*

Description

Check if string contains a substring

Usage

```
contains_str(x, substring)
```

Arguments

x	Character vector
substring	Substring to search for

Value

Logical vector

Examples

```
text <- c("hello world", "goodbye", "hello there")
text[contains_str(text, "hello")]
```

create_pipeline *Create a pipeline of tracked operations*

Description

Defines a sequence of operations that will be tracked as a unit.

Usage

```
create_pipeline(lake, name)
```

Arguments

lake	Lake instance
name	Name for this pipeline

Value

A Pipeline object

Examples

```
if (FALSE) {
    lake <- Lake$new("project")

    pipeline <- create_pipeline(lake, "preprocessing")
    pipeline$
        step("load", function() read.csv("data.csv"))$
        step("clean", function(data) na.omit(data))$
        step("normalize", function(data) scale(data))$
        run()

    # All steps are recorded in lineage
    lake$tree("preprocessing.normalize")
}
```

`deps` *Get dependencies from the default lake*

Description

Get dependencies from the default lake

Usage

```
deps(name, direction = "up")
```

Arguments

`name`	Data name
`direction`	"up" or "down"

Value

Dependencies data frame

Examples

```
use_lake("ex_deps", root = tempfile())
put("raw", data.frame(x = 1:3))
deps("raw")
```

dplyr_compat *dplyr Compatibility Layer*

Description

Make Lake work seamlessly with dplyr pipelines. Automatically tracks dependencies through dplyr operations.

Value

See the individual function help pages for return values.

Examples

```
if (FALSE) {
lake <- Lake$new("my_project")

# Dependencies are automatically tracked through the pipe
lake$ref("counts") |>
  dplyr::filter(quality > 0.8) |>
  dplyr::left_join(lake$ref("metadata"), by = "sample_id") |>
  dplyr::group_by(condition) |>
  dplyr::summarize(mean_expr = mean(expression)) |>
  save_as("summary", lake)
}
```

drop *Drop data from the default lake*

Description

Drop data from the default lake

Usage

```
drop(name, ...)
```

Arguments

name	Data name
...	Additional arguments passed to Lake$drop()

Value

Invisible Lake object

Examples

```
use_lake("ex_drop", root = tempfile())
put("t", data.frame(x = 1:3))
drop("t")
```

`ends_with_str` *Check if string ends with a suffix*

Description

Check if string ends with a suffix

Usage

```
ends_with_str(x, suffix)
```

Arguments

`x`	Character vector
`suffix`	Suffix to check for

Value

Logical vector

Examples

```
files <- c("a.csv", "b.txt", "c.csv")
files[ends_with_str(files, ".csv")]
```

`EpigenomicsAdapter` *Epigenomics Adapter*

Description

Adapter for storing and retrieving epigenomics-layer objects.

Details

This umbrella adapter is lower priority than specific adapters (e.g. Methylation / ATAC / ChIP) and captures epigenomics-marked objects.

Value

An R6 generator for a `LakeAdapter` subclass that serializes and restores objects of this omics layer.

Super class

OmicsLake::LakeAdapter -> EpigenomicsAdapter

Methods

Public methods:

- EpigenomicsAdapter$name()
- EpigenomicsAdapter$priority()
- EpigenomicsAdapter$put()
- EpigenomicsAdapter$get()
- EpigenomicsAdapter$components()
- EpigenomicsAdapter$exists()
- EpigenomicsAdapter$list_names()
- EpigenomicsAdapter$can_handle()
- EpigenomicsAdapter$clone()

Method name():

Usage:

```
EpigenomicsAdapter$name()
```

Method priority():

Usage:

```
EpigenomicsAdapter$priority()
```

Method put():

Usage:

```
EpigenomicsAdapter$put(lake, name, data)
```

Method get():

Usage:

```
EpigenomicsAdapter$get(lake, name, ref = "@latest")
```

Method components():

Usage:

```
EpigenomicsAdapter$components(lake, name)
```

Method exists():

Usage:

```
EpigenomicsAdapter$exists(lake, name)
```

Method list_names():

Usage:

```
EpigenomicsAdapter$list_names(lake)
```

Method can_handle():

Usage:

```
EpigenomicsAdapter$can_handle(data)
```

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
EpigenomicsAdapter$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
adapter <- EpigenomicsAdapter$new()
class(adapter)
```

`eval_bench` *OmicsLake Evaluation Benchmarks*

Description

Benchmark workloads W0-W2 for OmicsLake evaluation

Value

'NULL'. This topic documents internal evaluation helpers.

`eval_case_study` *OmicsLake Case Study (W3)*

Description

RNA-seq case study demonstrating reproducibility features

Value

'NULL'. This topic documents internal evaluation helpers.

`eval_generate` *OmicsLake Evaluation Data Generators*

Description

Functions for generating synthetic datasets for benchmarking

Value

'NULL'. This topic documents internal evaluation helpers.

eval_metrics *OmicsLake Evaluation Metrics*

Description

Functions for configuration loading, metrics collection, and result output

Value

'NULL'. This topic documents internal evaluation helpers.

eval_plot *OmicsLake Evaluation Plotting*

Description

Functions for generating evaluation figures

Value

'NULL'. This topic documents internal evaluation helpers.

export_data *Export data from the default lake*

Description

Export data from the default lake

Usage

```
export_data(name, path, ...)
```

Arguments

name	Data name to export
path	Output file path
...	Additional arguments

Value

Invisible path

Examples

```
use_lake("ex_export", root = tempfile())
put("t", data.frame(x = 1:3))
export_data("t", file.path(tempdir(), "t.parquet"))
```

`fetch` *Read data from the default lake*

Description

Note: This function is named 'fetch' to avoid conflict with base::get()

Usage

```
fetch(name, ...)
```

Arguments

`name`	Name of the data to read
`...`	Additional arguments passed to Lake$get()

Value

The requested data

See Also

Lake

Examples

```
use_lake("ex_fetch", root = tempfile())
put("counts", data.frame(gene = c("A", "B"), value = c(1, 2)))
fetch("counts")
```

from *Start a query from a table on the default lake*

Description

Start a query from a table on the default lake

Usage

```
from(table)
```

Arguments

table	Table name

Value

A QueryBuilder instance

Examples

```
use_lake("ex_from", root = tempfile())
put("t", data.frame(x = 1:3))
qb <- from("t")
```

GenomicsAdapter *Genomics Adapter*

Description

Adapter for storing and retrieving genomics-layer objects.

Details

This umbrella adapter is lower priority than specific adapters (e.g. VCF / RaggedExperiment) and captures genomics-marked objects.

Value

An R6 generator for a `LakeAdapter` subclass that serializes and restores objects of this omics layer.

Super class

`OmicsLake::LakeAdapter` -> `GenomicsAdapter`

Methods

Public methods:

- GenomicsAdapter$name()
- GenomicsAdapter$priority()
- GenomicsAdapter$put()
- GenomicsAdapter$get()
- GenomicsAdapter$components()
- GenomicsAdapter$exists()
- GenomicsAdapter$list_names()
- GenomicsAdapter$can_handle()
- GenomicsAdapter$clone()

Method `name()`**:**

Usage:

```
GenomicsAdapter$name()
```

Method `priority()`**:**

Usage:

```
GenomicsAdapter$priority()
```

Method `put()`**:**

Usage:

```
GenomicsAdapter$put(lake, name, data)
```

Method `get()`**:**

Usage:

```
GenomicsAdapter$get(lake, name, ref = "@latest")
```

Method `components()`**:**

Usage:

```
GenomicsAdapter$components(lake, name)
```

Method `exists()`**:**

Usage:

```
GenomicsAdapter$exists(lake, name)
```

Method `list_names()`**:**

Usage:

```
GenomicsAdapter$list_names(lake)
```

Method `can_handle()`**:**

Usage:

```
GenomicsAdapter$can_handle(data)
```

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
GenomicsAdapter$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

Examples

```
adapter <- GenomicsAdapter$new()
class(adapter)
```

`get_adapters` *Get registered adapters*

Description

Get registered adapters

Usage

```
get_adapters()
```

Value

List of registered adapters

Examples

```
adapters <- get_adapters()
length(adapters)
```

`GlycomicsAdapter` *Glycomics Adapter*

Description

Adapter for storing and retrieving glycomics-layer objects.

Details

Supports explicit glycomics marker classes/metadata.

Value

An R6 generator for a `LakeAdapter` subclass that serializes and restores objects of this omics layer.

Super class

`OmicsLake::LakeAdapter` -> `GlycomicsAdapter`

Methods

Public methods:

- GlycomicsAdapter$name()
- GlycomicsAdapter$priority()
- GlycomicsAdapter$put()
- GlycomicsAdapter$get()
- GlycomicsAdapter$components()
- GlycomicsAdapter$exists()
- GlycomicsAdapter$list_names()
- GlycomicsAdapter$can_handle()
- GlycomicsAdapter$clone()

Method `name()`**:**

Usage:

```
GlycomicsAdapter$name()
```

Method `priority()`**:**

Usage:

```
GlycomicsAdapter$priority()
```

Method `put()`**:**

Usage:

```
GlycomicsAdapter$put(lake, name, data)
```

Method `get()`**:**

Usage:

```
GlycomicsAdapter$get(lake, name, ref = "@latest")
```

Method `components()`**:**

Usage:

```
GlycomicsAdapter$components(lake, name)
```

Method `exists()`**:**

Usage:

```
GlycomicsAdapter$exists(lake, name)
```

Method `list_names()`**:**

Usage:

```
GlycomicsAdapter$list_names(lake)
```

Method `can_handle()`**:**

Usage:

```
GlycomicsAdapter$can_handle(data)
```

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
GlycomicsAdapter$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

Examples

```
adapter <- GlycomicsAdapter$new()
class(adapter)
```

---

history *Show history from the default lake*

Description

Show history from the default lake

Usage

```
history(name = NULL, ...)
```

Arguments

name	Optional data name
...	Additional arguments passed to Lake$history()

Value

History data frame

Examples

```
use_lake("ex_history", root = tempfile())
put("t", data.frame(x = 1:3))
history()
```

---

if_else_na *If-else with NA handling*

Description

If-else with NA handling

Usage

```
if_else_na(condition, true, false, na = NA)
```

Arguments

condition	Logical vector
true	Value when TRUE
false	Value when FALSE
na	Value when NA (default: NA)

Value

Vector with conditional values

Examples

```
cond <- c(TRUE, FALSE, NA)
if_else_na(cond, "yes", "no", na = "missing")
```

`import_data` *Import data into the default lake*

Description

Import data into the default lake

Usage

```
import_data(path, name, ...)
```

Arguments

`path`	File path to import
`name`	Name to store as
`...`	Additional arguments

Value

Invisible Lake object

Examples

```
use_lake("ex_import", root = tempfile())
f <- file.path(tempdir(), "imp.csv")
write.csv(data.frame(x = 1:3), f, row.names = FALSE)
import_data(f, "imported")
```

into *Create a pipe-compatible lake write function*

Description

Creates a function that can be used at the end of a pipe to save data.

Usage

```
into(lake)
```

Arguments

lake	Lake instance

Value

A function that accepts data and name, saves to lake

Examples

```
if (FALSE) {
write_to <- into(lake)

df |>
  dplyr::filter(x > 5) |>
  write_to("filtered_data")
}
```

is_not_null *Check for non-NULL/non-NA values*

Description

Check for non-NULL/non-NA values

Usage

```
is_not_null(x)
```

Arguments

x	Vector to check

Value

Logical vector indicating non-NA values

Examples

```
x <- c(1, NA, 3, NA, 5)
x[is_not_null(x)]
```

---

is_null *Check for NULL/NA values*

Description

Check for NULL/NA values

Usage

```
is_null(x)
```

Arguments

x	Vector to check

Value

Logical vector indicating NA values

Examples

```
x <- c(1, NA, 3, NA, 5)
x[is_null(x)]
```

---

lake *Get the current default lake*

Description

Get the current default lake

Usage

```
lake()
```

Value

The default Lake object

Examples

```
use_lake("ex_lake", root = tempfile())
l <- lake()
l$put("data", data.frame(x = 1:3))
```

Lake *Lake - OmicsLake Core Class*

Description

R6 class for versioned, lineage-tracked data management. Provides a modern, fluent API for data operations with automatic dependency tracking.

Value

A Lake R6 generator; call Lake$new() to create a lake.

SummarizedExperiment Support

SummarizedExperiment objects are stored via the SEAdapter pattern, which decomposes the object into queryable components:

- Assays are stored as tables (long format for efficient querying)
- colData and rowData are stored as separate tables
- Metadata is stored as a serialized R object

When you put() a SummarizedExperiment, all components are stored atomically within a transaction. When you tag() an SE object, all components are tagged together to maintain version consistency.

Legacy objects stored before the adapter registry are still readable via fallback detection.

Storage Types

- **Tables**: data.frames stored in DuckDB (queryable with SQL/dplyr)
- **Objects**: Arbitrary R objects serialized with qs/RDS
- **SummarizedExperiment**: Decomposed storage via SEAdapter

Methods

Public methods:

- Lake$new()
- Lake$put()
- Lake$get()
- Lake$ref()
- Lake$snap()
- Lake$tag()
- Lake$restore()
- Lake$diff()
- Lake$tree()
- Lake$plot()

- Lake$deps()
- Lake$impact()
- Lake$query()
- Lake$from()
- Lake$join()
- Lake$count()
- Lake$mean()
- Lake$tables()
- Lake$objects()
- Lake$ls()
- Lake$snaps()
- Lake$log()
- Lake$history()
- Lake$drop()
- Lake$rm()
- Lake$export()
- Lake$import()
- Lake$sql()
- Lake$q()
- Lake$[()
- Lake$[<-()
- Lake$print()
- Lake$exists()
- Lake$find()
- Lake$status()
- Lake$doctor()
- Lake$repair()
- Lake$finalize()

Method `new()`**:** Initialize a Lake project

Usage:

```
Lake$new(project = NULL, backend = "duckdb", auto_track = TRUE, root = NULL)
```

Arguments:

`project` Project name or path. If NULL, auto-generates a name.
`backend` Storage backend (currently only "duckdb" supported)
`auto_track` Enable automatic dependency tracking (default: TRUE)
`root` Root directory for lake storage (default: ~/.omicslake)

Returns: A new Lake object

Method `put()`**:** Write data to the lake

Usage:

```
Lake$put(name, data, depends_on = NULL, tags = NULL)
```

Arguments:

`name` Name for the data

`data` Data to store (data.frame, matrix, list, or any R object)

`depends_on` Optional explicit dependencies. Can be: - Character vector of names (uses @latest for all) - List of lists with 'name' and 'ref' elements for version-aware deps
Auto-tracking note: When 'auto_track=TRUE' (default), dependencies are automatically detected from: 1. 'lake_deps' attribute on data (set by 'save_as()' from dplyr pipes) 2. Tracked reads within a 'with_tracking()' or 'wrap_fn()' context
A simple 'lake$get()' followed by 'lake$put()' does NOT auto-capture dependencies. Use 'save_as()', 'with_tracking()', or specify 'depends_on'.

`tags` Optional tags for this version

Returns: Invisible self for chaining

Method `get()`**:** Read data from the lake

Usage:

```
Lake$get(name, ref = "@latest", where = NULL, select = NULL, collect = TRUE)
```

Arguments:

`name` Name of the data to read

`ref` Version reference ("@latest", "@first", "@tag(name)", or timestamp)

`where` Filter condition (formula, e.g., ~ col > 5)

`select` Columns to select (character vector)

`collect` Whether to collect results immediately (FALSE returns lazy reference)

Returns: The requested data

Method `ref()`**:** Get a lazy reference for dplyr operations

Usage:

```
Lake$ref(name, ref = "@latest")
```

Arguments:

`name` Table name

`ref` Version reference (default: "@latest")

Returns: A lazy table reference (tbl_lazy)

Method `snap()`**:** Create a project-wide snapshot

Usage:

```
Lake$snap(label, note = "", params = list())
```

Arguments:

`label` Label for this snapshot

`note` Optional description

`params` Optional parameters to store with the snapshot

Returns: Invisible self for chaining

Method `tag()`**:** Tag a specific data version

Usage:

```
Lake$tag(name, tag)
```

Arguments:

`name` Data name

`tag` Tag to apply

Returns: Invisible self for chaining

Method `restore()`**:** Restore project to a labeled snapshot

Usage:

```
Lake$restore(label)
```

Arguments:

`label` Snapshot label to restore

Returns: Invisible self for chaining

Method `diff()`**:** Compare two versions of data

Usage:

```
Lake$diff(name, ref1 = "@latest", ref2 = "@first")
```

Arguments:

`name` Data name

`ref1` First version reference (default: "@latest")

`ref2` Second version reference (default: "@first")

Returns: Comparison summary with row counts, column differences, and sample data

Method `tree()`**:** Show lineage tree

Usage:

```
Lake$tree(name = NULL, direction = "up", depth = 10)
```

Arguments:

`name` Starting node (NULL for all)

`direction` "up" (ancestors), "down" (descendants), or "both"

`depth` Maximum depth to traverse

Returns: Data frame of lineage relationships

Method `plot()`**:** Plot lineage graph

Usage:

```
Lake$plot(name = NULL, direction = "both")
```

Arguments:

`name` Starting node (NULL for full graph)

`direction` "up", "down", or "both"

Returns: Lineage plot (requires igraph)

Method `deps()`**:** Get direct dependencies

Usage:

```
Lake$deps(name, direction = "up")
```

Arguments:

`name` Data name

`direction` "up" (parents) or "down" (children)

Returns: Data frame of dependencies

Method `impact()`**:** Analyze impact of changing a data source

Usage:

```
Lake$impact(name)
```

Arguments:

`name` Data name to analyze

Returns: Data frame of affected downstream data

Method `query()`**:** Start a query builder chain

Usage:

```
Lake$query()
```

Returns: A new QueryBuilder instance

Method `from()`**:** Shortcut to start query from a table

Usage:

```
Lake$from(table)
```

Arguments:

`table` Table name

Returns: A QueryBuilder instance with FROM clause set

Method `join()`**:** Join two tables

Usage:

```
Lake$join(left, right, by = NULL, type = "left")
```

Arguments:

`left` Left table name

`right` Right table name

`by` Join columns (character vector or named vector for different column names)

`type` Join type ("left", "inner", "right", "full")

Returns: Joined data

Method `count()`**:** Count rows, optionally grouped

Usage:

```
Lake$count(table, ...)
```

Arguments:

`table` Table name

`...` Grouping variables (unquoted)

Returns: Data frame with counts

Method `mean()`**:** Calculate mean of a column

Usage:

```
Lake$mean(table, col, ...)
```

Arguments:

`table` Table name

`col` Column to average (unquoted)

`...` Grouping variables (unquoted)

Returns: Data frame with means

Method `tables()`**:** List all tables in the lake

Usage:

```
Lake$tables()
```

Returns: Data frame of table names

Method `objects()`**:** List all objects in the lake

Usage:

```
Lake$objects()
```

Returns: Data frame of object names

Method `ls()`**:** List all data (tables and objects)

Usage:

```
Lake$ls()
```

Returns: List with tables and objects data frames

Method `snaps()`**:** List all snapshots/labels

Usage:

```
Lake$snaps()
```

Returns: Data frame of snapshots

Method `log()`**:** Show history/log

Usage:

```
Lake$log(name = NULL, n = 20)
```

Arguments:

`name` Optional data name (NULL for project history)

`n` Maximum number of entries to return

Returns: Data frame of history entries

Method `history()`**:** Alias for log

Usage:

```
Lake$history(name = NULL, n = 20)
```

Arguments:

`name` Optional data name

`n` Maximum number of entries

Returns: Data frame of history entries

Method `drop()`**:** Remove data from the lake

Usage:

```
Lake$drop(name, force = FALSE)
```

Arguments:

`name` Data name

`force` Force removal even if has dependents

Returns: Invisible self for chaining

Method `rm()`**:** Alias for drop

Usage:

```
Lake$rm(name, force = FALSE)
```

Arguments:

`name` Data name

`force` Force removal

Returns: Invisible self for chaining

Method `export()`**:** Export data to a file

Usage:

```
Lake$export(name, path, format = NULL)
```

Arguments:

`name` Data name

`path` Output file path

`format` Output format ("parquet", "csv", "rds"). Auto-detected from extension if NULL.

Returns: Invisible path

Method `import()`**:** Import external data into the lake

Usage:

```
Lake$import(path, name, format = NULL)
```

Arguments:

`path` Input file path

`name` Name to store as

`format` Input format (auto-detected from extension if NULL)

Returns: Invisible self for chaining

Method `sql()`**:** Execute raw SQL query

Usage:

```
Lake$sql(query, collect = TRUE)
```

Arguments:

`query` SQL query string

`collect` Whether to collect results immediately

Returns: Query results

Method `q()`**:** Alias for sql

Usage:

```
Lake$q(query, collect = TRUE)
```

Arguments:

`query` SQL query string

`collect` Whether to collect results

Returns: Query results

Method `[()`**:** Bracket access for reading data

Usage:

```
Lake$[(name, i, j)
```

Arguments:

`name` Data name

`i` Row filter expression (optional)

`j` Column selection (optional)

Returns: Filtered/selected data

Method `[<-()`**:** Bracket assignment for writing data

Usage:

```
Lake$[<-(name, value)
```

Arguments:

`name` Data name

`value` Data to store

Method `print()`**:** Print lake summary

Usage:

```
Lake$print()
```

Method `exists()`**:** Check whether a data name exists in the lake

Usage:

```
Lake$exists(name, type = c("any", "table", "object"))
```

Arguments:

`name` Data name

`type` One of "any", "table", or "object"

Returns: TRUE/FALSE

Method `find()`**:** Find data names matching a pattern

Usage:

```
Lake$find(
  pattern = NULL,
  type = c("any", "table", "object"),
  ignore.case = TRUE,
  fixed = FALSE,
  fuzzy = TRUE,
  max_distance = 3L,
  min_score = -Inf,
  prefer_type = c("none", "table", "object"),
  limit = Inf
)
```

Arguments:

`pattern` Optional regex pattern; NULL returns all names

`type` One of "any", "table", or "object"

`ignore.case` Whether matching should ignore case

Returns: Data frame with columns name, type

Method `status()`**:** One-line status summary for this lake

Usage:

```
Lake$status()
```

Returns: A one-row data frame with status fields

Method `doctor()`**:** Run diagnostic checks on this lake

Usage:

```
Lake$doctor(verbose = TRUE)
```

Arguments:

`verbose` If TRUE, print a readable report

Returns: Data frame with columns check, ok, detail

Method `repair()`**:** Run the repair workflow on this lake

Usage:

```
Lake$repair(...)
```

Arguments:

`...` Arguments forwarded to the repair implementation

Returns: Repair report (invisibly)

Method `finalize()`**:** Clean up connections when object is garbage collected

Usage:

```
Lake$finalize()
```

Examples

```
if (FALSE) {
# Initialize a lake
lake <- Lake$new("my_project")

# Store and retrieve data
lake$put("counts", counts_df)
data <- lake$get("counts")

# Filter with formula syntax
filtered <- lake$get("counts", where = ~ expression > 100)

# Use with dplyr
lake$ref("counts") |>
  dplyr::filter(quality > 0.8) |>
  dplyr::collect()

# Version control
lake$snap("v1.0")
lake$tag("counts", "raw")
lake$restore("v1.0")

# View lineage
lake$tree("results")

# SummarizedExperiment storage
library(SummarizedExperiment)
se <- SummarizedExperiment(assays = list(counts = matrix(1:100, 10, 10)))
lake$put("my_experiment", se)
lake$tag("my_experiment", "v1")
se_retrieved <- lake$get("my_experiment", ref = "@tag(v1)")
}
```

---

lake_aliases *Simple 'lake_*' Aliases*

Description

Consistent, memorable aliases that follow one naming rule: use 'lake_<verb>' for common operations.

Usage

```
lake_use(project = NULL, ...)

lake_put(name, data, ...)

lake_get(name, ...)
```

```
lake_ref(name)

lake_snap(label, ...)

lake_tag(name, tag)

lake_tree(name = NULL, ...)

lake_has(name, type = c("any", "table", "object"))

lake_track(expr, ...)

lake_track_script(path, ...)

lake_auto_on(...)

lake_auto_off(commit = TRUE)

lake_strict_on(...)
```

Arguments

project	Project name for the default lake
...	Additional arguments forwarded to the corresponding function
name	Data/table/node name (meaning depends on each function)
data	Data to store
label	Snapshot label
tag	Tag name
type	One of "any", "table", or "object"
expr	Expression to track
path	Script file path
commit	If TRUE, write tracked lineage before disabling transparent mode

Value

Each alias returns the value of the function it forwards to.

Examples

```
lake_use("ex_aliases", root = tempfile())
lake_put("counts", data.frame(x = 1:3))
lake_get("counts")
lake_snap("v1")
```

lake_doctor *Run diagnostics for the default lake (or a specified project)*

Description

Run diagnostics for the default lake (or a specified project)

Usage

```
lake_doctor(project = NULL, ..., verbose = TRUE)
```

Arguments

project	Optional project name. If provided, checks that project directly.
...	Additional arguments passed to Lake$new() when project is provided.
verbose	If TRUE, print a readable report

Value

Data frame with diagnostic checks

Examples

```
use_lake("ex_doctor", root = tempfile())
put("t", data.frame(x = 1:3))
lake_doctor(verbose = FALSE)
```

lake_exists *Check whether a data name exists in the default lake*

Description

Check whether a data name exists in the default lake

Usage

```
lake_exists(name, type = c("any", "table", "object"))
```

Arguments

name	Data name
type	One of "any", "table", or "object"

Value

TRUE/FALSE

Examples

```
use_lake("ex_exists", root = tempfile())
put("t", data.frame(x = 1:3))
lake_exists("t")
```

`lake_find` *Find data names in the default lake*

Description

Find data names in the default lake

Usage

```
lake_find(
  pattern = NULL,
  type = c("any", "table", "object"),
  ignore.case = TRUE,
  fixed = FALSE,
  fuzzy = TRUE,
  max_distance = 3L,
  min_score = -Inf,
  prefer_type = c("none", "table", "object"),
  limit = Inf
)
```

Arguments

`pattern`	Optional regex/fixed pattern; NULL returns all names
`type`	One of "any", "table", or "object"
`ignore.case`	Whether matching should ignore case
`fixed`	Whether to treat pattern as fixed string
`fuzzy`	Whether to include fuzzy matches when exact/regex matching misses
`max_distance`	Maximum edit distance for fuzzy matches
`min_score`	Minimum score threshold for returned candidates
`prefer_type`	Optional type priority in tie-breaks: "none", "table", or "object"
`limit`	Maximum number of rows to return (Inf for all)

Value

Data frame with columns name, type, score, distance, and match_type

Examples

```
use_lake("ex_find", root = tempfile())
put("counts", data.frame(x = 1:3))
lake_find("count")
```

lake_repair *Repair workflow for the default lake*

Description

Runs a five-step workflow: 1) situation summary, 2) cause identification, 3) fix proposals, 4) optional auto-execution, and 5) before/after comparison.

Usage

```
lake_repair(
  target = NULL,
  restore_label = NULL,
  auto = FALSE,
  enable_strict = FALSE,
  strict_path = getOption("ol.repro.path", getwd()),
  renv_restore = FALSE,
  numeric_tolerance = 1e-08,
  ignore_row_order = TRUE,
  verbose = TRUE
)
```

Arguments

target	Optional data name to focus lineage diagnostics
restore_label	Optional snapshot label used for rollback proposals
auto	If TRUE, execute auto-supported proposals
enable_strict	If TRUE, enable strict reproducibility mode during auto execution
strict_path	Path used when enabling strict reproducibility mode
renv_restore	If TRUE, run renv::restore() during auto execution
numeric_tolerance	Numeric tolerance used for semantic value comparison
ignore_row_order	If TRUE, compare tabular targets ignoring row order
verbose	If TRUE, print the five-step report

Value

A lake_repair_report object

Examples

```
use_lake("ex_lake_repair", root = tempfile())
put("t", data.frame(x = 1:3))
lake_repair()
```

lake_status *Show one-line status for the default lake (or a specified project)*

Description

Show one-line status for the default lake (or a specified project)

Usage

```
lake_status(project = NULL, ...)
```

Arguments

project	Optional project name. If provided, checks that project directly.
...	Additional arguments passed to Lake$new() when project is provided.

Value

One-row data frame with status fields

Examples

```
use_lake("ex_status", root = tempfile())
put("t", data.frame(x = 1:3))
lake_status()
```

LakeAdapter *Lake Adapter Base Class*

Description

Base class for type-specific data adapters. Adapters provide specialized storage and retrieval for complex object types.

Details

Adapters allow OmicsLake to store complex objects (like SummarizedExperiment or Seurat objects) in a structured way that preserves all metadata and allows partial queries.

Value

A LakeAdapter R6 generator (base class for type adapters).

Methods

Public methods:

- LakeAdapter$name()
- LakeAdapter$can_handle()
- LakeAdapter$priority()
- LakeAdapter$put()
- LakeAdapter$get()
- LakeAdapter$components()
- LakeAdapter$exists()
- LakeAdapter$list_names()
- LakeAdapter$clone()

Method `name()`**:** Get the adapter name

Usage:

```
LakeAdapter$name()
```

Returns: Character string

Method `can_handle()`**:** Check if this adapter can handle the given data

Usage:

```
LakeAdapter$can_handle(data)
```

Arguments:

`data` Data to check

Returns: Logical indicating if adapter can handle this type

Method `priority()`**:** Get the priority of this adapter (higher = checked first)

Usage:

```
LakeAdapter$priority()
```

Returns: Numeric priority value

Method `put()`**:** Store data using this adapter

Usage:

```
LakeAdapter$put(lake, name, data)
```

Arguments:

`lake` Lake instance
`name` Data name
`data` Data to store

Returns: Invisible TRUE on success

Method `get()`**:** Retrieve data using this adapter

Usage:

```
LakeAdapter$get(lake, name, ref = "@latest")
```

Arguments:

`lake` Lake instance

`name` Data name

`ref` Version reference

Returns: The retrieved data

Method `components()`**:** List components stored for this data

Usage:

```
LakeAdapter$components(lake, name)
```

Arguments:

`lake` Lake instance

`name` Data name

Returns: Data frame of components

Method `exists()`**:** Check if data exists

Usage:

```
LakeAdapter$exists(lake, name)
```

Arguments:

`lake` Lake instance

`name` Data name

Returns: Logical

Method `list_names()`**:** List root names managed by this adapter

Usage:

```
LakeAdapter$list_names(lake)
```

Arguments:

`lake` Lake instance

Returns: Character vector of names

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
LakeAdapter$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
if (FALSE) {
    # Register a custom adapter
    my_adapter <- MyAdapter$new()
    register_adapter(my_adapter)

    # Now Lake will use this adapter for matching types
    lake$put("my_data", my_special_object)
}
```

`link` *Create an explicit dependency link*

Description

Manually creates a dependency relationship between two nodes in the lineage graph.

Usage

```
link(from, to, lake = NULL, relationship = "linked")
```

Arguments

`from`	Source node name (parent)
`to`	Target node name (child)
`lake`	Lake instance (uses default if NULL)
`relationship`	Type of relationship (default: "linked")

Value

Invisible TRUE on success

Examples

```
if (FALSE) {
    lake <- Lake$new("project")

    # Store some data
    lake$put("source_data", df1)
    lake$put("derived_data", df2)

    # Manually link them
    link("source_data", "derived_data", lake)

    # Now lineage shows the connection
    lake$tree("derived_data")
}
```

`LipidomicsAdapter` *Lipidomics Adapter*

Description

Adapter for storing and retrieving lipidomics-layer objects.

Details

Supports explicit lipidomics marker classes/metadata.

Value

An R6 generator for a `LakeAdapter` subclass that serializes and restores objects of this omics layer.

Super class

`OmicsLake::LakeAdapter` -> `LipidomicsAdapter`

Methods

Public methods:

- `LipidomicsAdapter$name()`
- `LipidomicsAdapter$priority()`
- `LipidomicsAdapter$put()`
- `LipidomicsAdapter$get()`
- `LipidomicsAdapter$components()`
- `LipidomicsAdapter$exists()`
- `LipidomicsAdapter$list_names()`
- `LipidomicsAdapter$can_handle()`
- `LipidomicsAdapter$clone()`

Method `name()`:

Usage:

```
LipidomicsAdapter$name()
```

Method `priority()`:

Usage:

```
LipidomicsAdapter$priority()
```

Method `put()`:

Usage:

```
LipidomicsAdapter$put(lake, name, data)
```

Method `get()`:

Usage:

```
LipidomicsAdapter$get(lake, name, ref = "@latest")
```

Method `components()`**:**

Usage:

```
LipidomicsAdapter$components(lake, name)
```

Method `exists()`**:**

Usage:

```
LipidomicsAdapter$exists(lake, name)
```

Method `list_names()`**:**

Usage:

```
LipidomicsAdapter$list_names(lake)
```

Method `can_handle()`**:**

Usage:

```
LipidomicsAdapter$can_handle(data)
```

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
LipidomicsAdapter$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
adapter <- LipidomicsAdapter$new()
class(adapter)
```

`MAEAdapter` *MultiAssayExperiment Adapter*

Description

Adapter for storing and retrieving `MultiAssayExperiment` objects.

Details

This adapter preserves `MultiAssayExperiment` components:

- experiments (`experiments`)
- sample map (`sampleMap`)
- participant metadata (`colData`)
- dropped samples (`drops`)
- object metadata (`metadata`)

Value

An R6 generator for a `LakeAdapter` subclass that serializes and restores objects of this omics layer.

Super class

`OmicsLake::LakeAdapter` -> `MAEAdapter`

Methods

Public methods:

- `MAEAdapter$name()`
- `MAEAdapter$can_handle()`
- `MAEAdapter$priority()`
- `MAEAdapter$put()`
- `MAEAdapter$get()`
- `MAEAdapter$components()`
- `MAEAdapter$exists()`
- `MAEAdapter$list_names()`
- `MAEAdapter$clone()`

Method `name()`:

Usage:

```
MAEAdapter$name()
```

Method `can_handle()`:

Usage:

```
MAEAdapter$can_handle(data)
```

Method `priority()`:

Usage:

```
MAEAdapter$priority()
```

Method `put()`:

Usage:

```
MAEAdapter$put(lake, name, data)
```

Method `get()`:

Usage:

```
MAEAdapter$get(lake, name, ref = "@latest")
```

Method `components()`:

Usage:

```
MAEAdapter$components(lake, name)
```

Method `exists()`:

Usage:

```
MAEAdapter$exists(lake, name)
```

Method `list_names()`**:**

Usage:

```
MAEAdapter$list_names(lake)
```

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
MAEAdapter$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
adapter <- MAEAdapter$new()
class(adapter)
```

`mark` *Mark data in lineage without storing*

Description

Creates a node in the lineage graph without actually storing any data. Useful for tracking external files or intermediate calculations.

Usage

```
mark(name, data = NULL, lake = NULL)
```

Arguments

`name`	Node name in the lineage graph
`data`	Optional data to extract metadata from (not stored). Use a character scalar for file path/URL, or a named list with fields like `source_kind`, `source_id`, `etag`, `last_modified`, `status_code`, `content_md5`, `response_md5`.
`lake`	Lake instance (uses default if NULL)

Value

Invisibly returns the data (for piping)

Examples

```
if (FALSE) {
    lake <- Lake$new("project")

    # Mark an external file in the lineage
    mark("external_data.csv", lake = lake)

    # Mark with metadata extraction
    large_matrix <- matrix(1:1000000, 1000, 1000)
    mark("large_computation", large_matrix, lake)
    # Only metadata is recorded, not the actual data
}
```

MetabolomicsAdapter *Metabolomics Adapter*

Description

Adapter for storing and retrieving metabolomics-layer objects.

Details

Supports explicit metabolomics marker classes/metadata.

Value

An R6 generator for a `LakeAdapter` subclass that serializes and restores objects of this omics layer.

Super class

`OmicsLake::LakeAdapter` -> `MetabolomicsAdapter`

Methods

Public methods:

- `MetabolomicsAdapter$name()`
- `MetabolomicsAdapter$priority()`
- `MetabolomicsAdapter$put()`
- `MetabolomicsAdapter$get()`
- `MetabolomicsAdapter$components()`
- `MetabolomicsAdapter$exists()`
- `MetabolomicsAdapter$list_names()`
- `MetabolomicsAdapter$can_handle()`
- `MetabolomicsAdapter$clone()`

Method `name()`:

Usage:

```
MetabolomicsAdapter$name()
```

Method `priority()`**:**

Usage:

```
MetabolomicsAdapter$priority()
```

Method `put()`**:**

Usage:

```
MetabolomicsAdapter$put(lake, name, data)
```

Method `get()`**:**

Usage:

```
MetabolomicsAdapter$get(lake, name, ref = "@latest")
```

Method `components()`**:**

Usage:

```
MetabolomicsAdapter$components(lake, name)
```

Method `exists()`**:**

Usage:

```
MetabolomicsAdapter$exists(lake, name)
```

Method `list_names()`**:**

Usage:

```
MetabolomicsAdapter$list_names(lake)
```

Method `can_handle()`**:**

Usage:

```
MetabolomicsAdapter$can_handle(data)
```

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
MetabolomicsAdapter$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
adapter <- MetabolomicsAdapter$new()
class(adapter)
```

`MethylationAdapter` *Methylation Adapter*

Description

Adapter for storing and retrieving methylation-layer objects.

Details

Supports common Bioconductor methylation classes (e.g. minfi family) by storing a full serialized object and manifest.

Value

An R6 generator for a `LakeAdapter` subclass that serializes and restores objects of this omics layer.

Super class

`OmicsLake::LakeAdapter` -> `MethylationAdapter`

Methods

Public methods:

- `MethylationAdapter$name()`
- `MethylationAdapter$priority()`
- `MethylationAdapter$put()`
- `MethylationAdapter$get()`
- `MethylationAdapter$components()`
- `MethylationAdapter$exists()`
- `MethylationAdapter$list_names()`
- `MethylationAdapter$can_handle()`
- `MethylationAdapter$clone()`

Method `name()`:

Usage:

```
MethylationAdapter$name()
```

Method `priority()`:

Usage:

```
MethylationAdapter$priority()
```

Method `put()`:

Usage:

```
MethylationAdapter$put(lake, name, data)
```

Method `get()`**:**

Usage:

```
MethylationAdapter$get(lake, name, ref = "@latest")
```

Method `components()`**:**

Usage:

```
MethylationAdapter$components(lake, name)
```

Method `exists()`**:**

Usage:

```
MethylationAdapter$exists(lake, name)
```

Method `list_names()`**:**

Usage:

```
MethylationAdapter$list_names(lake)
```

Method `can_handle()`**:**

Usage:

```
MethylationAdapter$can_handle(data)
```

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
MethylationAdapter$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
adapter <- MethylationAdapter$new()
class(adapter)
```

`MsExperimentAdapter` *MsExperiment Adapter*

Description

Adapter for storing and retrieving `MsExperiment` objects.

Details

This adapter preserves `MsExperiment` components:

- sample metadata (`sampleData`)
- spectra (`spectra`)
- quantitative/assay data (`qdata`)
- experiment files (`experimentFiles`)
- auxiliary data (`otherData`)

Value

An R6 generator for a `LakeAdapter` subclass that serializes and restores objects of this omics layer.

Super class

`OmicsLake::LakeAdapter` -> `MsExperimentAdapter`

Methods

Public methods:

- `MsExperimentAdapter$name()`
- `MsExperimentAdapter$can_handle()`
- `MsExperimentAdapter$priority()`
- `MsExperimentAdapter$put()`
- `MsExperimentAdapter$get()`
- `MsExperimentAdapter$components()`
- `MsExperimentAdapter$exists()`
- `MsExperimentAdapter$list_names()`
- `MsExperimentAdapter$clone()`

Method `name()`:

Usage:

```
MsExperimentAdapter$name()
```

Method `can_handle()`:

Usage:

```
MsExperimentAdapter$can_handle(data)
```

Method `priority()`:

Usage:

```
MsExperimentAdapter$priority()
```

Method `put()`:

Usage:

```
MsExperimentAdapter$put(lake, name, data)
```

Method `get()`:

Usage:

```
MsExperimentAdapter$get(lake, name, ref = "@latest")
```

Method `components()`:

Usage:

```
MsExperimentAdapter$components(lake, name)
```

Method `exists()`:

Usage:

```
MsExperimentAdapter$exists(lake, name)
```

Method `list_names()`**:**

Usage:

```
MsExperimentAdapter$list_names(lake)
```

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
MsExperimentAdapter$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
adapter <- MsExperimentAdapter$new()
class(adapter)
```

`not_between_operator` *NOT BETWEEN operator*

Description

NOT BETWEEN operator

Usage

```
x %!between% range
```

Arguments

`x`	Numeric vector
`range`	Numeric vector of length 2 (min, max)

Value

Logical vector

Examples

```
values <- 1:20
values[values %!between% c(5, 15)]
```

not_in_operator *NOT IN operator*

Description

NOT IN operator

Usage

```
x %!in% table
```

Arguments

x	Vector
table	Values to exclude

Value

Logical vector

Examples

```
letters[letters %!in% c("a", "e", "i", "o", "u")]
```

objects *List objects in the default lake*

Description

List objects in the default lake

Usage

```
objects()
```

Value

Data frame of object names

Examples

```
use_lake("ex_objects", root = tempfile())
put("model", list(a = 1, b = 2))
objects()
```

`observe` *Observation Mode*

Description

Track file I/O operations without modifying existing code. Provides a non-invasive way to build lineage from existing workflows.

Executes code while tracking file reads and writes to build a lineage graph. This is useful for understanding data flow in existing scripts without modifying them.

Usage

```
observe(expr, track_functions = NULL)
```

Arguments

`expr`	Expression or code block to observe
`track_functions`	Character vector of function names to intercept. 'NULL' uses common I/O defaults; 'character(0)' disables auto interception.

Details

Observation mode provides a framework for tracking file I/O operations. By default, `observe()` now auto-tracks common unqualified I/O calls (e.g. `read.csv()`, `write.csv()`, `readRDS()`, `saveRDS()`).

Limitations

Automatic interception is best-effort: - Namespaced calls (e.g. `utils::read.csv`) are not intercepted - I/O executed outside the observed R expression cannot be intercepted - For unsupported functions, use `record_read()` / `record_write()`

Recommended Approach

For reliable lineage tracking, use OmicsLake's native functions: - 'ol_write()' / 'ol_read()' for tables - 'ol_save()' / 'ol_read_object()' for R objects

These automatically track dependencies when 'depends_on' is specified.

Manual Recording

For legacy code, use 'record_read()' and 'record_write()' within observe() to manually record file operations.

Value

A list containing:

`result`	The result of evaluating expr
`reads`	Character vector of files read
`writes`	Character vector of files written
`lineage`	Data frame of inferred dependencies

Examples

```
if (FALSE) {
    # Manual recording within observe()
    lineage <- observe({
        record_read("input.csv")
        data <- read.csv("input.csv")
        result <- transform(data)
        record_write("output.csv")
        write.csv(result, "output.csv")
    })

    # View what was tracked
    print(lineage)
}

if (FALSE) {
    result <- observe({
        data <- read.csv("data.csv")
        processed <- data[data$value > 0, ]
        write.csv(processed, "processed.csv")
    })

    print(result$reads) # "data.csv"
    print(result$writes) # "processed.csv"
    print(result$lineage) # data.csv -> processed.csv
}
```

`observe_session` *Create an observed session*

Description

Starts an observation session that tracks all Lake operations until stopped.

Usage

```
observe_session(lake)
```

Arguments

`lake`	Lake instance to observe

Value

An ObserveSession object

Examples

```
if (FALSE) {
    lake <- Lake$new("project")
    session <- observe_session(lake)

    # Do work...
    lake$put("data1", df1)
    lake$put("data2", df2, depends_on = "data1")

    # Stop and get summary
    summary <- session$stop()
    print(summary)
}
```

`observe_to_lake` *Observe and record to a Lake*

Description

Executes code while tracking I/O, then records the lineage to a Lake.

Usage

```
observe_to_lake(expr, lake, prefix = "file:", track_functions = NULL)
```

Arguments

`expr`	Expression to observe
`lake`	Lake instance to record lineage to
`prefix`	Prefix for file-based node names in the lake
`track_functions`	Character vector forwarded to ‘observe()‘

Value

The result of evaluating expr

Examples

```
if (FALSE) {
    lake <- Lake$new("my_project")

    observe_to_lake(
        {
            source("analysis_pipeline.R")
        },
        lake = lake
    )
```

```
    # View the recorded lineage
    lake$tree()
}
```

`ol_add_rank` *Add ranking column to table*

Description

Add a ranking column using window functions (ROW_NUMBER, RANK, DENSE_RANK).

Usage

```
ol_add_rank(
  table,
  rank_by,
  partition_by = NULL,
  method = "row_number",
  descending = TRUE,
  as_column = "rank",
  project = getOption("ol.project"),
  collect = TRUE
)
```

Arguments

`table`	Character string, name of the table
`rank_by`	Character string, column name to rank by
`partition_by`	Character vector of columns to partition by (default: NULL)
`method`	Character, ranking method: "row_number", "rank", or "dense_rank" (default: "row_number")
`descending`	Logical; if TRUE, rank in descending order (default: TRUE)
`as_column`	Character, name of the ranking column to add (default: "rank")
`project`	Project name (default: current project)
`collect`	Logical; if TRUE returns data.frame, if FALSE returns lazy dplyr table

Value

Original table with added ranking column

Examples

```
if (FALSE) {
    ol_add_rank("genes", rank_by = "expression", method = "dense_rank")

    ol_add_rank("genes",
        rank_by = "expression",
        partition_by = "sample",
        as_column = "expression_rank"
    )
}
```

`ol_aggregate` *Aggregate data with multiple statistics*

Description

Calculate multiple aggregate statistics, optionally grouped by columns. Supports common aggregates: count, sum, avg, min, max, stddev, median, etc.

Usage

```
ol_aggregate(
  table,
  group_by = NULL,
  ...,
  project = getOption("ol.project"),
  collect = TRUE
)
```

Arguments

`table`	Character string, name of the table to aggregate
`group_by`	Character vector of column names to group by (default: NULL for overall aggregates)
`...`	Named arguments specifying aggregates. Each argument should be a list with 'func' (aggregate function name) and 'col' (column name). The argument name becomes the output column name.
`project`	Project name (default: current project)
`collect`	Logical; if TRUE returns data.frame, if FALSE returns lazy dplyr table

Value

Aggregated results as data.frame (if collect=TRUE) or lazy table (if collect=FALSE)

Examples

```
ol_init("ex_aggregate", root = tempfile())
ol_write("t", data.frame(g = c("a", "a", "b"), v = c(1, 2, 3)))
ol_aggregate("t", group_by = "g", mean_v = list(func = "mean", col = "v"))
```

---

ol_checkout *Restore entire project to a labeled state*

Description

Restore entire project to a labeled state

Usage

```
ol_checkout(label, project = getOption("ol.project"))
```

Arguments

label	The label name to restore to
project	The project name

Value

Invisible TRUE on success, FALSE if no tables found

Examples

```
ol_init("ex_ol_checkout", root = tempfile())
ol_write("t", data.frame(x = 1:3))
ol_label("v1")
ol_checkout("v1")
```

---

ol_commit *Commit with metadata (note and parameters)*

Description

Commit with metadata (note and parameters)

Usage

```
ol_commit(note = "", params = list(), project = getOption("ol.project"))
```

Arguments

note	Commit message describing the changes
params	Named list of parameters to store with the commit
project	Project name

Details

By default, commits automatically embed reproducibility context under 'params$omicslake_repro' (Git state, 'renv.lock', and session/system metadata when available). Control this with: 'options(ol.repro.capture = TRUE/FALSE)', 'options(ol.repro.path = "/path/to/analysis")', 'options(ol.repro.include = c("git","renv","session","system 'options(ol.repro.strict = TRUE/FALSE)', 'options(ol.repro.require_clean_git = TRUE/FALSE)', and AI metadata options such as 'options(ol.agent.prompt_id = "...")'.

Value

Invisible commit identifier string

Examples

```
ol_init("ex_ol_commit", root = tempfile())
ol_write("t", data.frame(x = 1:3))
ol_commit("initial load")
```

---

ol_compare_versions *Compare versions of an object*

Description

Shows a side-by-side comparison of multiple versions of an object, including timestamps, tags, size changes, and dependency changes.

Usage

```
ol_compare_versions(name, versions = NULL, project = getOption("ol.project"))
```

Arguments

name	Name of the object to compare versions for
versions	Optional vector of version identifiers (timestamps or tags). If NULL, compares all versions.
project	Project name

Value

A data frame comparing the specified versions

Examples

```
ol_init("ex_compare_versions", root = tempfile())
ol_save("m", list(a = 1))
ol_save("m", list(a = 2))
ol_compare_versions("m")
```

`ol_create_view` *Create a database view*

Description

Creates a virtual table (view) based on a SQL query. Views are useful for creating reusable queries, especially for comparing multiple versions of data. Views do not store data but execute their query each time they are referenced.

Usage

```
ol_create_view(
  name,
  sql,
  project = getOption("ol.project"),
  replace = TRUE,
  depends_on = NULL
)
```

Arguments

`name`	Name for the view to create
`sql`	SQL query defining the view (must be a SELECT statement)
`project`	Project name (default: current project from options)
`replace`	Whether to replace the view if it already exists (default: TRUE)
`depends_on`	Optional character vector of table/object names that this view depends on

Value

Invisible qualified view name

Examples

```
if (FALSE) {
    ol_init("myproject")
    ol_write("genes", data.frame(gene_id = 1:100, expr = rnorm(100)))

    ol_create_view("high_expr", "SELECT * FROM genes WHERE expr > 0")

    ol_write("genes_v2", data.frame(gene_id = 1:100, expr = rnorm(100)))
    ol_create_view("gene_comparison",
```

```
        "SELECT g1.gene_id, g1.expr as expr_v1, g2.expr as expr_v2,
    (g2.expr - g1.expr) as change
    FROM genes g1
    JOIN genes_v2 g2 ON g1.gene_id = g2.gene_id",
        depends_on = c("genes", "genes_v2")
     )

     ol_read("gene_comparison")
}
```

`ol_cumulative_sum` *Calculate cumulative sum*

Description

Add a cumulative sum column using window functions.

Usage

```
ol_cumulative_sum(
  table,
  column,
  partition_by = NULL,
  order_by,
  as_column = NULL,
  project = getOption("ol.project"),
  collect = TRUE
)
```

Arguments

`table`	Character string, name of the table
`column`	Character string, column name to calculate cumulative sum on
`partition_by`	Character vector of columns to partition by (default: NULL)
`order_by`	Character string, column to order by (required)
`as_column`	Character, name of the output column (default: paste0(column, "_cumsum"))
`project`	Project name (default: current project)
`collect`	Logical; if TRUE returns data.frame, if FALSE returns lazy dplyr table

Value

Original table with added cumulative sum column

Examples

```
if (FALSE) {
    ol_cumulative_sum("counts", "value", order_by = "time")

    ol_cumulative_sum("counts", "value",
        partition_by = "sample",
        order_by = "gene_id"
    )
}
```

`ol_disable_transparent_tracking` *Disable transparent background tracking and optionally commit results*

Description

Restores wrapped I/O functions in '.GlobalEnv' and optionally records tracked reads/writes to Lake before stopping observation mode.

Usage

```
ol_disable_transparent_tracking(commit = TRUE)
```

Arguments

`commit`	If TRUE, write tracked lineage and optional observation object to Lake

Value

A 'lake_observation' object (invisible) with tracked reads/writes/lineage

Examples

```
if (FALSE) {
ol_enable_transparent_tracking(project = "rna_project", auto_disable =
FALSE)
    # ... run normal analysis code ...
    ol_disable_transparent_tracking(commit = TRUE)
}
```

ol_drop — *Drop (delete) a table from the project*

Description

Drop (delete) a table from the project

Usage

```
ol_drop(name, project = getOption("ol.project"))
```

Arguments

name	Name of the table to drop
project	Project name

Value

Invisible TRUE on success

Examples

```
ol_init("ex_ol_drop", root = tempfile())
ol_write("t", data.frame(x = 1:3))
ol_drop("t")
```

ol_drop_object — *Drop (delete) an object from the project*

Description

Drop (delete) an object from the project

Usage

```
ol_drop_object(name, project = getOption("ol.project"))
```

Arguments

name	Name of the object to drop
project	Project name

Value

Invisible TRUE on success

Examples

```
ol_init("ex_ol_drop_object", root = tempfile())
ol_save("m", list(a = 1))
ol_drop_object("m")
```

`ol_drop_view` *Drop a database view*

Description

Drops a view from the project. Also removes any dependency records associated with the view.

Usage

```
ol_drop_view(name, project = getOption("ol.project"))
```

Arguments

`name`	Name of the view to drop
`project`	Project name (default: current project from options)

Value

Invisible TRUE if successful

Examples

```
if (FALSE) {
    ol_init("myproject")
    ol_create_view("my_view", "SELECT * FROM genes WHERE expr > 0")

    ol_drop_view("my_view")
}
```

`ol_enable_strict_repro_mode`
Enable Strict Reproducibility Mode

Description

Applies an opinionated, audit-first option preset for agent-mediated analysis: reproducibility metadata capture ON, clean-git requirement ON, and snapshot validation ON.

Usage

```
ol_enable_strict_repro_mode(
  path = getwd(),
  prompt_id = NULL,
  run_id = NULL,
  agent_name = NULL,
  include = c("git", "renv", "session", "system"),
  snapshot_validate_mode = c("error", "warn", "off"),
  max_tables = 200L
)
```

Arguments

path	Base path used for Git/renv detection (usually repository root)
prompt_id	Optional prompt/work-item identifier
run_id	Optional run/session identifier
agent_name	Optional software-agent name
include	Metadata components to capture
snapshot_validate_mode	Snapshot validation behavior: "error", "warn", or "off"
max_tables	Maximum tables stored in snapshot validation details

Value

Named list of previous option values (invisible)

Examples

```
# Minimal runnable example for BiocCheck
x <- 1
x

if (FALSE) {
ol_enable_strict_repro_mode(
  path = getwd(),
  prompt_id = "prompt-2026-02-21-001"
)
}
```

---

ol_enable_transparent_tracking

Enable transparent background tracking for a session

Description

Installs temporary wrappers for common unqualified I/O functions in '.GlobalEnv', so existing analysis code can run unchanged while reads/writes are tracked and recorded to Lake.

Usage

```
ol_enable_transparent_tracking(
  project = NULL,
  prefix = "file:",
  snapshot = NULL,
  track_functions = NULL,
  store_observation = TRUE,
  observation_name = NULL,
  observation_depends_on = c("writes", "reads", "both", "none"),
  auto_disable = TRUE,
  ...
)
```

Arguments

project	Optional project name. If set, calls ‘use_lake(project, ...)‘.
prefix	Prefix for file-based node names in the lake
snapshot	Optional snapshot label created when tracking is disabled with commit
track_functions	Character vector forwarded to tracking wrappers
store_observation	If TRUE, store observed reads/writes/lineage as an object
observation_name	Optional target name when ‘store_observation = TRUE‘
observation_depends_on	Dependencies used for observation record: ‘"writes"‘, ‘"reads"‘, ‘"both"‘, or ‘"none"‘
auto_disable	If TRUE, installs a ‘.Last()‘ hook to auto-commit on session end
...	Additional arguments passed to ‘use_lake()‘ when ‘project‘ is provided

Value

Invisible Lake object

Examples

```
use_lake("ex_transparent", root = tempfile())
ol_enable_transparent_tracking()
ol_disable_transparent_tracking(commit = FALSE)
```

`ol_export_parquet` *Export a table or object to a Parquet file*

Description

Export a table or object to a Parquet file

Usage

```
ol_export_parquet(
  name,
  path,
  ref = "@latest",
  project = getOption("ol.project"),
  compression = c("snappy", "zstd", "lz4", "brotli", "uncompressed"),
  compression_level = NULL,
  row_group_size = 1e+05,
  overwrite = FALSE
)
```

Arguments

`name`	Name of the table or object to export
`path`	Output file path for the Parquet file
`ref`	Version reference (default: "@latest")
`project`	Project name
`compression`	Compression algorithm: "snappy" (default), "zstd", "lz4", "brotli", or "uncompressed"
`compression_level`	Compression level (codec-specific, NULL for default)
`row_group_size`	Number of rows per row group (default: 100000)
`overwrite`	Whether to overwrite existing file (default: FALSE)

Value

Invisible normalized output path

Examples

```
ol_init("ex_export_parquet", root = tempfile())
ol_write("t", data.frame(x = 1:3))
ol_export_parquet("t", file.path(tempdir(), "t.parquet"), overwrite = TRUE)
```

`ol_fread` *fread-like reader for Parquet tables*

Description

fread-like reader for Parquet tables

Usage

```
ol_fread(
  name,
  ref = "@latest",
  select = NULL,
  drop = NULL,
  nrows = Inf,
  filter = NULL,
  project = getOption("ol.project"),
  as_tibble = FALSE
)
```

Arguments

`name`	Table name
`ref`	Reference string
`select`	Optional character vector of columns to keep
`drop`	Optional character vector of columns to drop
`nrows`	Maximum rows to return
`filter`	Optional filter expression (string parsed by rlang)
`project`	Project name
`as_tibble`	If TRUE, return tibble when available

Value

Data frame or tibble

Examples

```
ol_init("ex_ol_fread", root = tempfile())
ol_write("t", data.frame(x = 1:3, y = 4:6))
ol_fread("t", select = "x")
```

ol_get_dependencies *Get dependencies for a table or object*

Description

Get dependencies for a table or object

Usage

```
ol_get_dependencies(
  name,
  direction = c("upstream", "downstream", "both"),
  project = getOption("ol.project")
)
```

Arguments

name	Name of the table or object
direction	Direction to query: "upstream" (parents), "downstream" (children), or "both"
project	Project name

Value

Data frame with dependency info including version references (parent_ref, parent_version_id)

Examples

```
ol_init("ex_ol_get_dependencies", root = tempfile())
ol_write("raw", data.frame(x = 1:3))
ol_write("filtered", data.frame(x = 1:2), depends_on = "raw")
ol_get_dependencies("filtered")
```

ol_import_parquet *Import a Parquet file into the project*

Description

Import a Parquet file into the project

Usage

```
ol_import_parquet(
  path,
  name,
  project = getOption("ol.project"),
  mode = c("create", "overwrite", "append"),
  depends_on = NULL,
  hive_partitioning = NULL,
  union_by_name = FALSE
)
```

Arguments

path	Input file path(s) for Parquet file(s). Can be a single file, list of files, or glob pattern.
name	Destination table name in the project
project	Project name
mode	Import mode: "create", "overwrite", or "append"
depends_on	Optional character vector of table/object names that this import depends on
hive_partitioning	Whether to interpret path as Hive partitioned (NULL for auto-detect, TRUE/FALSE to force)
union_by_name	Whether to unify columns by name rather than position when reading multiple files

Value

Invisible TRUE on success

Examples

```
ol_init("ex_import_parquet", root = tempfile())
ol_write("t", data.frame(x = 1:3))
p <- file.path(tempdir(), "t.parquet")
ol_export_parquet("t", p, overwrite = TRUE)
ol_import_parquet(p, "t_imported")
```

---

ol_init *Initialize an OmicsLake project*

Description

Initialize an OmicsLake project

Usage

```
ol_init(project, root = NULL, ...)
```

Arguments

project	Project name
root	Optional storage root directory. Defaults to the OmicsLake root.
...	Additional arguments passed to the backend initializer

Value

Invisible normalized project root path

Examples

```
ol_init("ex_ol_init", root = tempfile())
```

---

ol_label *Label current state with a human-friendly alias*

Description

Label current state with a human-friendly alias

Usage

```
ol_label(
  label,
  state_id = NULL,
  project = getOption("ol.project"),
  .in_transaction = FALSE
)
```

Arguments

label	Label name to assign to the current state
state_id	Optional state identifier; defaults to the current state
project	Project name
.in_transaction	Internal parameter - if TRUE, skip transaction in child calls (caller handles it)

Value

Invisible label name

Examples

```
ol_init("ex_ol_label", root = tempfile())
ol_write("t", data.frame(x = 1:3))
ol_label("v1")
```

ol_list_labels *List all project-level labels*

Description

List all project-level labels

Usage

```
ol_list_labels(project = getOption("ol.project"))
```

Arguments

project	Project name

Value

Data frame of project labels

Examples

```
ol_init("ex_ol_list_labels", root = tempfile())
ol_write("t", data.frame(x = 1:3))
ol_list_labels()
```

ol_list_object_versions *List all versions of an object*

Description

Returns a data frame with all saved versions of an object, including timestamps, tags, size, and dependencies for each version.

Usage

```
ol_list_object_versions(name, project = getOption("ol.project"))
```

Arguments

name	Name of the object to list versions for
project	Project name

Value

A data frame with columns: version_ts, tags, size_bytes, dependencies

Examples

```
ol_init("ex_list_object_versions", root = tempfile())
ol_save("m", list(a = 1))
ol_list_object_versions("m")
```

`ol_list_objects` *List all saved objects in the project*

Description

List all saved objects in the project

Usage

```
ol_list_objects(project = getOption("ol.project"))
```

Arguments

`project`	Project name

Value

Data frame of object names

Examples

```
ol_init("ex_ol_list_objects", root = tempfile())
ol_save("m", list(a = 1))
ol_list_objects()
```

`ol_list_tables` *List all tables in the project*

Description

List all tables in the project

Usage

```
ol_list_tables(project = getOption("ol.project"))
```

Arguments

`project`	Project name

Value

Data frame of table names

Examples

```
ol_init("ex_ol_list_tables", root = tempfile())
ol_write("t", data.frame(x = 1:3))
ol_list_tables()
```

`ol_list_tags` *List all tags for a table or all project labels*

Description

List all tags for a table or all project labels

Usage

```
ol_list_tags(name = NULL, project = getOption("ol.project"))
```

Arguments

`name`	Optional table name to list tags for. If NULL, lists all tags.
`project`	Project name

Value

Data frame of tags

Examples

```
ol_init("ex_ol_list_tags", root = tempfile())
ol_write("t", data.frame(x = 1:3))
ol_list_tags()
```

ol_list_views *List all views in the project*

Description

Returns a data frame with information about all views in the project, including their SQL definitions and dependencies.

Usage

```
ol_list_views(project = getOption("ol.project"))
```

Arguments

project	Project name (default: current project from options)

Value

A data frame with columns: view_name, definition, dependencies

Examples

```
ol_init("ex_list_views", root = tempfile())
ol_write("t", data.frame(x = 1:3))
ol_create_view("v", "SELECT * FROM t")
ol_list_views()
```

ol_load *Alias of* ol_read()

Description

Alias of ol_read()

Usage

```
ol_load(name, ref = "@latest", project = getOption("ol.project"))
```

Arguments

name	Table or object name
ref	Reference string (e.g. "@latest", "@tag(v1)")
project	Project name

Value

Data frame, lazy table, or stored object

Examples

```
ol_init("ex_ol_load", root = tempfile())
ol_save("model", list(a = 1))
ol_load("model")
```

---

ol_log *Return version log for a table*

Description

Return version log for a table

Usage

```
ol_log(name = NULL, project = getOption("ol.project"))
```

Arguments

name	Optional table name. If NULL, returns commit log.
project	Project name

Value

Data frame of version history

Examples

```
ol_init("ex_ol_log", root = tempfile())
ol_write("t", data.frame(x = 1:3))
ol_log()
```

---

ol_log_commits *View commit history*

Description

View commit history

Usage

```
ol_log_commits(project = getOption("ol.project"), n = 20)
```

Arguments

project	Project name
n	Maximum number of commits to return

Value

Data frame of commits with reproducibility and agent metadata columns

Examples

```
ol_init("ex_ol_log_commits", root = tempfile())
ol_write("t", data.frame(x = 1:3))
ol_commit("note")
ol_log_commits()
```

`ol_moving_avg` *Calculate moving average*

Description

Add a moving average column using window functions.

Usage

```
ol_moving_avg(
  table,
  column,
  window_size = 3,
  partition_by = NULL,
  order_by,
  as_column = NULL,
  project = getOption("ol.project"),
  collect = TRUE
)
```

Arguments

`table`	Character string, name of the table
`column`	Character string, column name to calculate moving average on
`window_size`	Integer, size of the moving window (default: 3)
`partition_by`	Character vector of columns to partition by (default: NULL)
`order_by`	Character string, column to order by (required)
`as_column`	Character, name of the output column (default: paste0(column, "_ma", window_size))
`project`	Project name (default: current project)
`collect`	Logical; if TRUE returns data.frame, if FALSE returns lazy dplyr table

Value

Original table with added moving average column

Examples

```
ol_init("ex_moving_avg", root = tempfile())
ol_write("t", data.frame(x = 1:5))
ol_moving_avg("t", "x", window_size = 2, order_by = "x")
```

`ol_plot_lineage` *Visualize dependency lineage as a graph*

Description

Creates a visualization of the dependency relationships between tables and objects in the OmicsLake project. Requires the igraph package to be installed.

Usage

```
ol_plot_lineage(
  name,
  direction = c("upstream", "downstream", "both"),
  layout = c("tree", "sugiyama", "circle", "auto"),
  max_depth = 10,
  project = getOption("ol.project"),
  vertex.size = 20,
  vertex.label.cex = 0.8,
  edge.arrow.size = 0.5,
  main = NULL,
  ...
)
```

Arguments

`name`	Name of the table or object to visualize
`direction`	Direction to traverse: "upstream" (dependencies), "downstream" (dependents), or "both"
`layout`	Graph layout algorithm: "tree", "sugiyama" (hierarchical), "circle", or "auto"
`max_depth`	Maximum depth to traverse (default: 10)
`project`	Project name
`vertex.size`	Size of graph nodes (default: 20)
`vertex.label.cex`	Size of node labels (default: 0.8)
`edge.arrow.size`	Size of edge arrows (default: 0.5)
`main`	Plot title (auto-generated if NULL)
`...`	Additional arguments passed to plot.igraph()

Value

An igraph object (invisibly)

Examples

```
if (requireNamespace("igraph", quietly = TRUE)) {
    ol_init("ex_plot_lineage", root = tempfile())
    ol_write("raw", data.frame(x = 1:3))
    ol_write("f", data.frame(x = 1:2), depends_on = "raw")
    ol_plot_lineage("f")
}
```

---

ol_query *Execute a custom SQL query on the OmicsLake database*

Description

This function allows you to run arbitrary SQL queries against the tables in your OmicsLake project. It provides full access to DuckDB's SQL capabilities including JOINs, aggregations, window functions, and more. Table names can be referenced without the schema prefix (e.g., 'genes' instead of 'ol.genes').

Usage

```
ol_query(sql, project = getOption("ol.project"), collect = TRUE, params = NULL)
```

Arguments

sql	Character string containing the SQL query to execute
project	Project name (default: current project from options)
collect	Logical; if TRUE (default), returns a data.frame. If FALSE, returns a lazy dplyr table for further manipulation
params	Optional named list of parameters for parameterized queries

Value

If collect=TRUE, returns a data.frame with query results. If collect=FALSE, returns a lazy dplyr tbl for further manipulation.

Examples

```
ol_init("ex_query", root = tempfile())
ol_write("t", data.frame(x = 1:3))
ol_query("SELECT COUNT(*) AS n FROM t")
```

ol_read *Read a table by name and reference*

Description

Read a table by name and reference

Usage

```
ol_read(
  name,
  ref = "@latest",
  project = getOption("ol.project"),
  collect = TRUE
)
```

Arguments

name	Table or object name
ref	Reference string (e.g. "@latest", "@tag(v1)")
project	Project name
collect	If TRUE, return data.frame; if FALSE, return lazy table

Value

Data frame, lazy table, or stored object

Examples

```
ol_init("ex_ol_read", root = tempfile())
ol_write("counts", data.frame(x = 1:3))
ol_read("counts")
```

ol_read_mae *Compose a MultiAssayExperiment*

Description

Compose a MultiAssayExperiment

Usage

```
ol_read_mae(
  assays,
  ref = "@latest",
  project = getOption("ol.project"),
  backing = c("hdf5", "memory")
)
```

Arguments

assays	Named list of assay specifications
ref	Reference string
project	Project name
backing	Backing mode: "hdf5" or "memory"

Value

MultiAssayExperiment object

Examples

```
if (requireNamespace("MultiAssayExperiment", quietly = TRUE)) {
    ol_init("ex_ol_read_mae", root = tempfile())
    long <- data.frame(
        feature = rep(c("g1", "g2"), 2),
        sample = rep(c("s1", "s2"), each = 2),
        value = c(5, 8, 1, 3)
    )
    ol_write("rna_long", long)
    ol_read_mae(list(rna = list(name = "rna_long")), backing = "memory")
}
```

---

ol_read_object *Read a stored object*

Description

Read a stored object

Usage

```
ol_read_object(
  name,
  ref = "@latest",
  when = NULL,
  project = getOption("ol.project")
)
```

Arguments

name	Name of the object to read
ref	Reference to read from (e.g., "@latest", "@tag", "v1.0"). Defaults to "@latest"
when	Deprecated. Use ref parameter instead. If provided, "latest" or "first"
project	Project name

Value

The stored R object

Examples

```
ol_init("ex_ol_read_object", root = tempfile())
ol_save("model", list(a = 1))
ol_read_object("model")
```

---

ol_read_se *Build a SummarizedExperiment from long table*

Description

Build a SummarizedExperiment from long table

Usage

```
ol_read_se(
  name,
  ref = "@latest",
  feature_col = "feature",
  sample_col = "sample",
  value_col = "value",
  project = getOption("ol.project"),
  backing = c("hdf5", "memory")
)
```

Arguments

name	Table name
ref	Reference string
feature_col	Column name containing feature ids
sample_col	Column name containing sample ids
value_col	Column name containing measurement values
project	Project name
backing	Backing mode: "hdf5" or "memory"

Value

SummarizedExperiment object

Examples

```
ol_init("ex_ol_read_se", root = tempfile())
long <- data.frame(
    feature = rep(c("g1", "g2"), each = 2),
    sample = rep(c("s1", "s2"), 2),
    value = c(5, 8, 1, 3)
)
ol_write("expr_long", long)
ol_read_se("expr_long", backing = "memory")
```

---

ol_save *Save an R object via the backend metadata table*

Description

Save an R object via the backend metadata table

Usage

```
ol_save(
  name,
  object,
  project = getOption("ol.project"),
  mime = NULL,
  depends_on = NULL
)
```

Arguments

name	Object name
object	R object to save
project	Project name
mime	MIME type for object payload
depends_on	Optional character vector of table/object names that this object depends on

Value

Invisible TRUE on success

Examples

```
ol_init("ex_ol_save", root = tempfile())
ol_save("model", list(coef = 1:3))
```

ol_show_lineage *Show lineage (full dependency tree) for a table or object*

Description

Show lineage (full dependency tree) for a table or object

Usage

```
ol_show_lineage(
  name,
  direction = c("upstream", "downstream"),
  max_depth = 10,
  project = getOption("ol.project")
)
```

Arguments

name	Name of the table or object
direction	Direction to traverse: "upstream" (all ancestors) or "downstream" (all descendants)
max_depth	Maximum depth to traverse (default: 10)
project	Project name

Value

Data frame describing the lineage tree

Examples

```
ol_init("ex_ol_show_lineage", root = tempfile())
ol_write("raw", data.frame(x = 1:3))
ol_write("filtered", data.frame(x = 1:2), depends_on = "raw")
ol_show_lineage("filtered")
```

ol_tag *Tag a table by creating a backup*

Description

Tag a table by creating a backup

Usage

```
ol_tag(
  name,
  tag,
  ref = "@latest",
  project = getOption("ol.project"),
  .in_transaction = FALSE
)
```

Arguments

name	Table name to tag
tag	Tag name to assign
ref	Reference to tag (e.g. "@latest")
project	Project name
.in_transaction	Internal parameter - if TRUE, skip transaction management (caller handles it)

Value

Invisible backup table name

Examples

```
ol_init("ex_ol_tag", root = tempfile())
ol_write("t", data.frame(x = 1:3))
ol_tag("t", "raw")
```

---

ol_tag_object *Tag a stored object version*

Description

Tag a stored object version

Usage

```
ol_tag_object(
  name,
  tag,
  when = "latest",
  project = getOption("ol.project"),
  .in_transaction = FALSE
)
```

Arguments

name	Name of the object to tag
tag	Tag name to assign
when	Which version to tag: "latest" (default) or "first", or a specific version_ts timestamp
project	Project name
.in_transaction	Internal parameter - if TRUE, skip transaction management (caller handles it)

Value

Invisible TRUE on success

Examples

```
ol_init("ex_ol_tag_object", root = tempfile())
ol_save("m", list(a = 1))
ol_tag_object("m", "v1")
```

---

ol_top_n *Get top N rows*

Description

Get top N rows from a table, optionally partitioned.

Usage

```
ol_top_n(
  table,
  n = 10,
  order_by,
  partition_by = NULL,
  descending = TRUE,
  project = getOption("ol.project"),
  collect = TRUE
)
```

Arguments

table	Character string, name of the table
n	Integer, number of top rows to return per partition (default: 10)
order_by	Character string, column to order by (required)
partition_by	Character vector of columns to partition by (default: NULL for overall top N)
descending	Logical; if TRUE, order in descending order (default: TRUE)
project	Project name (default: current project)
collect	Logical; if TRUE returns data.frame, if FALSE returns lazy dplyr table

Value

Top N rows

Examples

```
if (FALSE) {
    ol_top_n("genes", n = 10, order_by = "expression")

    ol_top_n("genes",
        n = 5,
        order_by = "expression",
        partition_by = "sample"
    )
}
```

`ol_write` *Write a table using the DuckDB backend*

Description

Write a table using the DuckDB backend

Usage

```
ol_write(
  name,
  data,
  project = getOption("ol.project"),
  mode = c("create", "overwrite", "append"),
  depends_on = NULL
)
```

Arguments

`name`	Table name
`data`	Data frame to store
`project`	Project name
`mode`	Write mode: "create", "overwrite", or "append"
`depends_on`	Optional character vector of table/object names that this table depends on

Value

Invisible qualified table name

Examples

```
ol_init("ex_ol_write", root = tempfile())
ol_write("counts", data.frame(gene = c("A", "B"), value = c(1, 2)))
```

operators *Custom Query Operators*

Description

SQL-like operators for use in Lake queries. These operators provide familiar SQL semantics for filtering data.

Value

See the individual operator help pages; each returns a logical vector.

Examples

```
df <- data.frame(
    gene = c("MT-CO1", "MT-CO2", "ACTB"),
    value = c(50, 80, 5)
)
# Pattern matching
df[df$gene %like% "MT-%", ]
# Range filtering
df[df$value %between% c(10, 100), ]
# Case-insensitive matching
df[df$gene %ilike% "mt-%", ]
```

PhosphoproteomicsAdapter *Phosphoproteomics Adapter*

Description

Adapter for storing and retrieving phosphoproteomics-layer objects.

Details

Supports explicit phosphoproteomics marker classes/metadata.

Value

An R6 generator for a `LakeAdapter` subclass that serializes and restores objects of this omics layer.

Super class

`OmicsLake::LakeAdapter` -> `PhosphoproteomicsAdapter`

Methods

Public methods:

- PhosphoproteomicsAdapter$name()
- PhosphoproteomicsAdapter$priority()
- PhosphoproteomicsAdapter$put()
- PhosphoproteomicsAdapter$get()
- PhosphoproteomicsAdapter$components()
- PhosphoproteomicsAdapter$exists()
- PhosphoproteomicsAdapter$list_names()
- PhosphoproteomicsAdapter$can_handle()
- PhosphoproteomicsAdapter$clone()

Method `name()`**:**

Usage:

```
PhosphoproteomicsAdapter$name()
```

Method `priority()`**:**

Usage:

```
PhosphoproteomicsAdapter$priority()
```

Method `put()`**:**

Usage:

```
PhosphoproteomicsAdapter$put(lake, name, data)
```

Method `get()`**:**

Usage:

```
PhosphoproteomicsAdapter$get(lake, name, ref = "@latest")
```

Method `components()`**:**

Usage:

```
PhosphoproteomicsAdapter$components(lake, name)
```

Method `exists()`**:**

Usage:

```
PhosphoproteomicsAdapter$exists(lake, name)
```

Method `list_names()`**:**

Usage:

```
PhosphoproteomicsAdapter$list_names(lake)
```

Method `can_handle()`**:**

Usage:

```
PhosphoproteomicsAdapter$can_handle(data)
```

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
PhosphoproteomicsAdapter$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

Examples

```
adapter <- PhosphoproteomicsAdapter$new()
class(adapter)
```

---

print.lake_observation *Print method for lake_observation*

Description

Print method for lake_observation

Usage

```
## S3 method for class 'lake_observation'
print(x, ...)
```

Arguments

x	A lake_observation object
...	Additional arguments (ignored)

Value

Invisibly returns x.

---

print.lake_repair_report *Print method for lake_repair_report*

Description

Print method for lake_repair_report

Usage

```
## S3 method for class 'lake_repair_report'
print(x, ...)
```

Arguments

x	A lake_repair_report object
...	Additional arguments (ignored)

Value

The input object, invisibly

`ProteomicsAdapter` *Proteomics Adapter*

Description

Adapter for storing and retrieving proteomics-layer objects.

Details

This umbrella adapter is lower priority than specific adapters (e.g. QFeatures / MsExperiment / Spectra) and captures proteomics-marked objects.

Value

An R6 generator for a `LakeAdapter` subclass that serializes and restores objects of this omics layer.

Super class

`OmicsLake::LakeAdapter` -> `ProteomicsAdapter`

Methods

Public methods:

- `ProteomicsAdapter$name()`
- `ProteomicsAdapter$priority()`
- `ProteomicsAdapter$put()`
- `ProteomicsAdapter$get()`
- `ProteomicsAdapter$components()`
- `ProteomicsAdapter$exists()`
- `ProteomicsAdapter$list_names()`
- `ProteomicsAdapter$can_handle()`
- `ProteomicsAdapter$clone()`

Method `name()`:

Usage:

```
ProteomicsAdapter$name()
```

Method `priority()`:

Usage:

```
ProteomicsAdapter$priority()
```

Method `put()`:

Usage:

```
ProteomicsAdapter$put(lake, name, data)
```

Method `get()`**:**

Usage:

```
ProteomicsAdapter$get(lake, name, ref = "@latest")
```

Method `components()`**:**

Usage:

```
ProteomicsAdapter$components(lake, name)
```

Method `exists()`**:**

Usage:

```
ProteomicsAdapter$exists(lake, name)
```

Method `list_names()`**:**

Usage:

```
ProteomicsAdapter$list_names(lake)
```

Method `can_handle()`**:**

Usage:

```
ProteomicsAdapter$can_handle(data)
```

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
ProteomicsAdapter$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
adapter <- ProteomicsAdapter$new()
class(adapter)
```

`put` *Write data to the default lake*

Description

Write data to the default lake

Usage

```
put(name, data, ...)
```

Arguments

`name`	Name for the data
`data`	Data to store
`...`	Additional arguments passed to Lake$put()

Value

Invisible Lake object

See Also

Lake

Examples

```
use_lake("ex_put", root = tempfile())
put("counts", data.frame(gene = c("A", "B"), value = c(1, 2)))
```

QFeaturesAdapter *QFeatures Adapter*

Description

Adapter for storing and retrieving QFeatures objects.

Details

This adapter preserves QFeatures components:

- underlying experiments/sample maps via MultiAssayExperiment adapter
- assay links (assayLinks)

Value

An R6 generator for a LakeAdapter subclass that serializes and restores objects of this omics layer.

Super class

OmicsLake::LakeAdapter -> QFeaturesAdapter

Methods

Public methods:

- QFeaturesAdapter$name()
- QFeaturesAdapter$can_handle()
- QFeaturesAdapter$priority()
- QFeaturesAdapter$put()
- QFeaturesAdapter$get()
- QFeaturesAdapter$components()
- QFeaturesAdapter$exists()
- QFeaturesAdapter$list_names()
- QFeaturesAdapter$clone()

Method `name()`**:**

Usage:

```
QFeaturesAdapter$name()
```

Method `can_handle()`**:**

Usage:

```
QFeaturesAdapter$can_handle(data)
```

Method `priority()`**:**

Usage:

```
QFeaturesAdapter$priority()
```

Method `put()`**:**

Usage:

```
QFeaturesAdapter$put(lake, name, data)
```

Method `get()`**:**

Usage:

```
QFeaturesAdapter$get(lake, name, ref = "@latest")
```

Method `components()`**:**

Usage:

```
QFeaturesAdapter$components(lake, name)
```

Method `exists()`**:**

Usage:

```
QFeaturesAdapter$exists(lake, name)
```

Method `list_names()`**:**

Usage:

```
QFeaturesAdapter$list_names(lake)
```

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
QFeaturesAdapter$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
adapter <- QFeaturesAdapter$new()
class(adapter)
```

query *Start a query builder on the default lake*

Description

Start a query builder on the default lake

Usage

```
query()
```

Value

A QueryBuilder instance

Examples

```
use_lake("ex_query", root = tempfile())
put("t", data.frame(x = 1:3))
qb <- query()
```

QueryBuilder *QueryBuilder - Fluent Query Interface*

Description

Build complex queries using a fluent, chainable API. Provides an intuitive way to construct queries without writing SQL.

Value

A `QueryBuilder` R6 generator; call `$new()` to create one.

Methods

Public methods:

- `QueryBuilder$new()`
- `QueryBuilder$from()`
- `QueryBuilder$join()`
- `QueryBuilder$left_join()`
- `QueryBuilder$inner_join()`
- `QueryBuilder$right_join()`
- `QueryBuilder$full_join()`
- `QueryBuilder$where()`

- QueryBuilder$filter()
- QueryBuilder$select()
- QueryBuilder$pick()
- QueryBuilder$mutate()
- QueryBuilder$group_by()
- QueryBuilder$summarize()
- QueryBuilder$summarise()
- QueryBuilder$having()
- QueryBuilder$order_by()
- QueryBuilder$arrange()
- QueryBuilder$limit()
- QueryBuilder$take()
- QueryBuilder$top()
- QueryBuilder$offset()
- QueryBuilder$distinct()
- QueryBuilder$run()
- QueryBuilder$collect()
- QueryBuilder$as()
- QueryBuilder$save_as()
- QueryBuilder$show_sql()
- QueryBuilder$explain()
- QueryBuilder$print()
- QueryBuilder$clone()

Method `new()`**:** Initialize a QueryBuilder

Usage:

```
QueryBuilder$new(lake)
```

Arguments:

`lake` Lake instance to query from

Method `from()`**:** Specify the source table

Usage:

```
QueryBuilder$from(table, alias = NULL)
```

Arguments:

`table` Table name

`alias` Optional alias for the table

Returns: Self for chaining

Method `join()`**:** Add a join clause

Usage:

```
QueryBuilder$join(table, on = NULL, type = "left", alias = NULL)
```

Arguments:

`table` Table to join

`on` Join condition (character vector of column names, or named vector for different column names)

`type` Join type ("left", "inner", "right", "full")

`alias` Optional alias for the joined table

Returns: Self for chaining

Method `left_join()`**:** Add a LEFT JOIN

Usage:

```
QueryBuilder$left_join(table, on = NULL, alias = NULL)
```

Arguments:

`table` Table to join

`on` Join condition

`alias` Optional alias

Returns: Self for chaining

Method `inner_join()`**:** Add an INNER JOIN

Usage:

```
QueryBuilder$inner_join(table, on = NULL, alias = NULL)
```

Arguments:

`table` Table to join

`on` Join condition

`alias` Optional alias

Returns: Self for chaining

Method `right_join()`**:** Add a RIGHT JOIN

Usage:

```
QueryBuilder$right_join(table, on = NULL, alias = NULL)
```

Arguments:

`table` Table to join

`on` Join condition

`alias` Optional alias

Returns: Self for chaining

Method `full_join()`**:** Add a FULL JOIN

Usage:

```
QueryBuilder$full_join(table, on = NULL, alias = NULL)
```

Arguments:

`table` Table to join

`on` Join condition

`alias` Optional alias

Returns: Self for chaining

Method `where()`**:** Add a filter condition (WHERE clause)

Usage:

```
QueryBuilder$where(...)
```

Arguments:

`...` Filter expressions (combined with AND)

Returns: Self for chaining

Method `filter()`**:** Alias for where

Usage:

```
QueryBuilder$filter(...)
```

Arguments:

`...` Filter expressions

Returns: Self for chaining

Method `select()`**:** Select columns

Usage:

```
QueryBuilder$select(...)
```

Arguments:

`...` Column names or expressions

Returns: Self for chaining

Method `pick()`**:** Alias for select

Usage:

```
QueryBuilder$pick(...)
```

Arguments:

`...` Column names or expressions

Returns: Self for chaining

Method `mutate()`**:** Add computed columns

Usage:

```
QueryBuilder$mutate(...)
```

Arguments:

`...` Name-value pairs for new columns

Returns: Self for chaining

Method `group_by()`**:** Group by columns

Usage:

```
QueryBuilder$group_by(...)
```

Arguments:

... Grouping columns

Returns: Self for chaining

Method `summarize()`**:** Summarize after grouping

Usage:

```
QueryBuilder$summarize(...)
```

Arguments:

... Aggregation expressions

Returns: Self for chaining

Method `summarise()`**:** Alias for summarize

Usage:

```
QueryBuilder$summarise(...)
```

Arguments:

... Aggregation expressions

Returns: Self for chaining

Method `having()`**:** Add HAVING clause (filter after grouping)

Usage:

```
QueryBuilder$having(...)
```

Arguments:

... Filter conditions

Returns: Self for chaining

Method `order_by()`**:** Order results

Usage:

```
QueryBuilder$order_by(...)
```

Arguments:

... Columns to order by (use desc() for descending)

Returns: Self for chaining

Method `arrange()`**:** Alias for order_by

Usage:

```
QueryBuilder$arrange(...)
```

Arguments:

... Columns to order by

Returns: Self for chaining

Method `limit()`**:** Limit the number of results

Usage:

```
QueryBuilder$limit(n)
```

Arguments:

`n` Maximum number of rows

Returns: Self for chaining

Method `take()`**:** Alias for limit

Usage:

```
QueryBuilder$take(n)
```

Arguments:

`n` Maximum number of rows

Returns: Self for chaining

Method `top()`**:** Get top N rows by a column

Usage:

```
QueryBuilder$top(n, by, desc = TRUE)
```

Arguments:

`n` Number of rows

`by` Column to order by

`desc` Use descending order (default: TRUE)

Returns: Self for chaining

Method `offset()`**:** Skip rows

Usage:

```
QueryBuilder$offset(n)
```

Arguments:

`n` Number of rows to skip

Returns: Self for chaining

Method `distinct()`**:** Return distinct rows only

Usage:

```
QueryBuilder$distinct(...)
```

Arguments:

`...` Optional columns to check for distinctness

Returns: Self for chaining

Method `run()`**:** Execute the query and return results

Usage:

```
QueryBuilder$run()
```

Returns: Data frame of results

Method `collect()`**:** Alias for run

Usage:

```
QueryBuilder$collect()
```

Returns: Data frame of results

Method `as()`**:** Execute and save to lake

Usage:

```
QueryBuilder$as(name)
```

Arguments:

`name` Name to save as

Returns: Invisible Lake object

Method `save_as()`**:** Save alias for as

Usage:

```
QueryBuilder$save_as(name)
```

Arguments:

`name` Name to save as

Returns: Invisible Lake object

Method `show_sql()`**:** Get the generated SQL (for debugging)

Usage:

```
QueryBuilder$show_sql()
```

Returns: SQL string

Method `explain()`**:** Explain the query plan

Usage:

```
QueryBuilder$explain()
```

Returns: Explanation data frame

Method `print()`**:** Print query builder state

Usage:

```
QueryBuilder$print()
```

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
QueryBuilder$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
if (FALSE) {
    lake <- Lake$new("my_project")

    # Basic query
    lake$from("users")$
        where(age > 30)$
```

```
        select(name, age)$
        run()

    # Join and aggregate
    lake$from("orders")$
        join("customers", on = "customer_id")$
        where(status == "completed")$
        group_by(region)$
        summarize(total = sum(amount))$
        order_by(desc(total))$
        run()

    # Save results to lake
    lake$from("data")$
        filter(quality > 0.8)$
        as("filtered_data")
}
```

`RaggedExperimentAdapter` *RaggedExperiment Adapter*

Description

Adapter for storing and retrieving RaggedExperiment objects.

Details

This adapter stores a full-fidelity serialized RaggedExperiment object and manifest.

Value

An R6 generator for a `LakeAdapter` subclass that serializes and restores objects of this omics layer.

Super class

`OmicsLake::LakeAdapter` -> `RaggedExperimentAdapter`

Methods

Public methods:

- `RaggedExperimentAdapter$name()`
- `RaggedExperimentAdapter$priority()`
- `RaggedExperimentAdapter$put()`
- `RaggedExperimentAdapter$get()`
- `RaggedExperimentAdapter$components()`
- `RaggedExperimentAdapter$exists()`

- RaggedExperimentAdapter$list_names()
- RaggedExperimentAdapter$can_handle()
- RaggedExperimentAdapter$clone()

Method `name()`**:**

Usage:

```
RaggedExperimentAdapter$name()
```

Method `priority()`**:**

Usage:

```
RaggedExperimentAdapter$priority()
```

Method `put()`**:**

Usage:

```
RaggedExperimentAdapter$put(lake, name, data)
```

Method `get()`**:**

Usage:

```
RaggedExperimentAdapter$get(lake, name, ref = "@latest")
```

Method `components()`**:**

Usage:

```
RaggedExperimentAdapter$components(lake, name)
```

Method `exists()`**:**

Usage:

```
RaggedExperimentAdapter$exists(lake, name)
```

Method `list_names()`**:**

Usage:

```
RaggedExperimentAdapter$list_names(lake)
```

Method `can_handle()`**:**

Usage:

```
RaggedExperimentAdapter$can_handle(data)
```

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
RaggedExperimentAdapter$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

Examples

```
adapter <- RaggedExperimentAdapter$new()
class(adapter)
```

record_read *Manually record a file read for observation*

Description

Use this to explicitly record file reads when automatic tracking is not available. Call this within an observe() block.

Usage

```
record_read(path)
```

Arguments

path	File path that was read

Value

Invisibly returns the path

Examples

```
if (FALSE) {
    observe({
        record_read("input.csv")
        data <- read.csv("input.csv")
        record_write("output.csv")
        write.csv(data, "output.csv")
    })
}
```

record_write *Manually record a file write for observation*

Description

Use this to explicitly record file writes when automatic tracking is not available. Call this within an observe() block.

Usage

```
record_write(path)
```

Arguments

path	File path that was written

Value

Invisibly returns the path

Examples

```
use_lake("ex_record_write", root = tempfile())
f <- file.path(tempdir(), "result.csv")
write.csv(data.frame(x = 1:3), f, row.names = FALSE)
record_write(f)
```

`ref` *Get a lazy reference from the default lake*

Description

Get a lazy reference from the default lake

Usage

```
ref(name)
```

Arguments

`name`	Table name

Value

A lazy table reference for use with dplyr

Examples

```
use_lake("ex_ref", root = tempfile())
put("t", data.frame(x = 1:5))
dplyr::collect(dplyr::filter(ref("t"), x > 2))
```

register_adapter *Register a data adapter*

Description

Register a data adapter

Usage

```
register_adapter(adapter)
```

Arguments

adapter	An LakeAdapter instance

Value

Invisible TRUE on success

Examples

```
register_adapter(SEAdapter$new())
```

restore *Restore the default lake to a snapshot*

Description

Restore the default lake to a snapshot

Usage

```
restore(label)
```

Arguments

label	Snapshot label

Value

Invisible Lake object

Examples

```
use_lake("ex_restore", root = tempfile())
put("t", data.frame(x = 1:3))
snap("v1")
restore("v1")
```

save_as *Save pipe result to lake*

Description

This function is designed to be used at the end of a dplyr pipe to save results to a Lake with automatic dependency tracking.

Usage

```
save_as(.data, name, lake = NULL)
```

Arguments

.data	Data from pipe (data.frame or tbl_lazy)
name	Name to save as in the lake
lake	Lake instance. If NULL, uses the default lake from use_lake()

Value

Invisibly returns the data (for potential further piping)

Examples

```
if (FALSE) {
lake$ref("raw_data") |>
  dplyr::filter(quality > 0.5) |>
  dplyr::mutate(normalized = value / mean(value)) |>
  save_as("processed_data", lake)
}
```

SCEAdapter *SingleCellExperiment Adapter*

Description

Adapter for storing and retrieving SingleCellExperiment objects.

Details

This adapter extends SummarizedExperiment support by preserving SingleCellExperiment-specific components:

- reduced dimensions (`reducedDims`)
- alternative experiments (`altExps`)
- row/column pairings (`rowPairs`, `colPairs`)
- size factors (`sizeFactors`)
- column labels (`colLabels`)
- main experiment label (`mainExpName`)

Value

An R6 generator for a `LakeAdapter` subclass that serializes and restores objects of this omics layer.

Super class

`OmicsLake::LakeAdapter` -> `SCEAdapter`

Methods

Public methods:

- `SCEAdapter$name()`
- `SCEAdapter$can_handle()`
- `SCEAdapter$priority()`
- `SCEAdapter$put()`
- `SCEAdapter$get()`
- `SCEAdapter$components()`
- `SCEAdapter$exists()`
- `SCEAdapter$list_names()`
- `SCEAdapter$clone()`

Method `name()`:

Usage:

```
SCEAdapter$name()
```

Method `can_handle()`:

Usage:

```
SCEAdapter$can_handle(data)
```

Method `priority()`:

Usage:

```
SCEAdapter$priority()
```

Method `put()`:

Usage:

```
SCEAdapter$put(lake, name, data)
```

Method `get()`**:**

Usage:

```
SCEAdapter$get(lake, name, ref = "@latest")
```

Method `components()`**:**

Usage:

```
SCEAdapter$components(lake, name)
```

Method `exists()`**:**

Usage:

```
SCEAdapter$exists(lake, name)
```

Method `list_names()`**:**

Usage:

```
SCEAdapter$list_names(lake)
```

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
SCEAdapter$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
adapter <- SCEAdapter$new()
class(adapter)
```

`SEAdapter` *SummarizedExperiment Adapter*

Description

Adapter for storing and retrieving Bioconductor SummarizedExperiment objects. Stores all components (assays, colData, rowData, metadata) with full fidelity.

Details

This adapter decomposes a SummarizedExperiment into its components:

- Assays are stored as tables (sparse matrix -> long format)
- colData is stored as a table
- rowData is stored as a table
- metadata is stored as an R object

Value

An SEAdapter R6 generator for SummarizedExperiment objects.

Super class

OmicsLake::LakeAdapter -> SEAdapter

Methods

Public methods:

- SEAdapter$name()
- SEAdapter$can_handle()
- SEAdapter$priority()
- SEAdapter$put()
- SEAdapter$get()
- SEAdapter$components()
- SEAdapter$exists()
- SEAdapter$list_names()
- SEAdapter$clone()

Method name():

Usage:

```
SEAdapter$name()
```

Method can_handle():

Usage:

```
SEAdapter$can_handle(data)
```

Method priority():

Usage:

```
SEAdapter$priority()
```

Method put():

Usage:

```
SEAdapter$put(lake, name, data)
```

Method get():

Usage:

```
SEAdapter$get(lake, name, ref = "@latest")
```

Method components():

Usage:

```
SEAdapter$components(lake, name)
```

Method exists():

Usage:

```
SEAdapter$exists(lake, name)
```

Method `list_names()`**:**

Usage:

```
SEAdapter$list_names(lake)
```

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
SEAdapter$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
if (FALSE) {
    library(SummarizedExperiment)

    # Create a SE object
    se <- SummarizedExperiment(
        assays = list(counts = matrix(1:100, 10, 10)),
        colData = DataFrame(sample = paste0("S", 1:10)),
        rowData = DataFrame(gene = paste0("G", 1:10))
    )

    # Store in lake
    lake$put("my_se", se)

    # Retrieve
    se2 <- lake$get("my_se")
}
```

`SeuratAdapter` *Seurat Adapter*

Description

Adapter for storing and retrieving Seurat objects.

Details

This adapter stores a full-fidelity serialized Seurat object and manifest.

Value

An R6 generator for a `LakeAdapter` subclass that serializes and restores objects of this omics layer.

Super class

OmicsLake::LakeAdapter -> SeuratAdapter

Methods

Public methods:

- SeuratAdapter$name()
- SeuratAdapter$priority()
- SeuratAdapter$put()
- SeuratAdapter$get()
- SeuratAdapter$components()
- SeuratAdapter$exists()
- SeuratAdapter$list_names()
- SeuratAdapter$can_handle()
- SeuratAdapter$clone()

Method name():

Usage:

```
SeuratAdapter$name()
```

Method priority():

Usage:

```
SeuratAdapter$priority()
```

Method put():

Usage:

```
SeuratAdapter$put(lake, name, data)
```

Method get():

Usage:

```
SeuratAdapter$get(lake, name, ref = "@latest")
```

Method components():

Usage:

```
SeuratAdapter$components(lake, name)
```

Method exists():

Usage:

```
SeuratAdapter$exists(lake, name)
```

Method list_names():

Usage:

```
SeuratAdapter$list_names(lake)
```

Method can_handle():

Usage:

```
SeuratAdapter$can_handle(data)
```

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
SeuratAdapter$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
adapter <- SeuratAdapter$new()
class(adapter)
```

`shortcuts` *Global Lake Shortcuts*

Description

Convenience functions for working with a default lake. These functions provide a simpler API when working with a single project.

Value

Each shortcut returns the value of the underlying `Lake` method it wraps (see the individual help pages).

Examples

```
use_lake("ex_shortcuts", root = tempfile())
put("counts", data.frame(gene = c("A", "B"), value = c(1, 2)))
fetch("counts")
snap("v1.0")
tree()
```

show_migration_guide *Show migration guide for ol_* to Lake API*

Description

Show migration guide for ol_* to Lake API

Usage

```
show_migration_guide()
```

Value

Invisible NULL; prints the migration guide to the console.

Examples

```
show_migration_guide()
```

snap *Create a snapshot of the default lake*

Description

Create a snapshot of the default lake

Usage

```
snap(label, ...)
```

Arguments

label	Label for the snapshot
...	Additional arguments passed to Lake$snap()

Value

Invisible Lake object

Examples

```
use_lake("ex_snap", root = tempfile())
put("t", data.frame(x = 1:3))
snap("v1.0")
```

`SpatialExperimentAdapter` *SpatialExperiment Adapter*

Description

Adapter for storing and retrieving SpatialExperiment objects.

Details

This adapter stores a full-fidelity serialized SpatialExperiment object and manifest.

Value

An R6 generator for a `LakeAdapter` subclass that serializes and restores objects of this omics layer.

Super class

`OmicsLake::LakeAdapter` -> `SpatialExperimentAdapter`

Methods

Public methods:

- `SpatialExperimentAdapter$name()`
- `SpatialExperimentAdapter$priority()`
- `SpatialExperimentAdapter$put()`
- `SpatialExperimentAdapter$get()`
- `SpatialExperimentAdapter$components()`
- `SpatialExperimentAdapter$exists()`
- `SpatialExperimentAdapter$list_names()`
- `SpatialExperimentAdapter$can_handle()`
- `SpatialExperimentAdapter$clone()`

Method `name()`:

Usage:

```
SpatialExperimentAdapter$name()
```

Method `priority()`:

Usage:

```
SpatialExperimentAdapter$priority()
```

Method `put()`:

Usage:

```
SpatialExperimentAdapter$put(lake, name, data)
```

Method `get()`**:**

Usage:

```
SpatialExperimentAdapter$get(lake, name, ref = "@latest")
```

Method `components()`**:**

Usage:

```
SpatialExperimentAdapter$components(lake, name)
```

Method `exists()`**:**

Usage:

```
SpatialExperimentAdapter$exists(lake, name)
```

Method `list_names()`**:**

Usage:

```
SpatialExperimentAdapter$list_names(lake)
```

Method `can_handle()`**:**

Usage:

```
SpatialExperimentAdapter$can_handle(data)
```

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
SpatialExperimentAdapter$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
adapter <- SpatialExperimentAdapter$new()
class(adapter)
```

`SpectraAdapter` *Spectra Adapter*

Description

Adapter for storing and retrieving `Spectra` objects.

Details

This adapter preserves `Spectra` components:

- peak lists (`peaksData`)
- per-spectrum metadata (`spectraData`)

Value

An R6 generator for a `LakeAdapter` subclass that serializes and restores objects of this omics layer.

Super class

`OmicsLake::LakeAdapter` -> `SpectraAdapter`

Methods

Public methods:

- `SpectraAdapter$name()`
- `SpectraAdapter$can_handle()`
- `SpectraAdapter$priority()`
- `SpectraAdapter$put()`
- `SpectraAdapter$get()`
- `SpectraAdapter$components()`
- `SpectraAdapter$exists()`
- `SpectraAdapter$list_names()`
- `SpectraAdapter$clone()`

Method `name()`**:**

Usage:

```
SpectraAdapter$name()
```

Method `can_handle()`**:**

Usage:

```
SpectraAdapter$can_handle(data)
```

Method `priority()`**:**

Usage:

```
SpectraAdapter$priority()
```

Method `put()`**:**

Usage:

```
SpectraAdapter$put(lake, name, data)
```

Method `get()`**:**

Usage:

```
SpectraAdapter$get(lake, name, ref = "@latest")
```

Method `components()`**:**

Usage:

```
SpectraAdapter$components(lake, name)
```

Method `exists()`**:**

Usage:

```
SpectraAdapter$exists(lake, name)
```

Method `list_names()`**:**

Usage:

```
SpectraAdapter$list_names(lake)
```

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
SpectraAdapter$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
adapter <- SpectraAdapter$new()
class(adapter)
```

`sql` *Execute SQL on the default lake*

Description

Execute SQL on the default lake

Usage

```
sql(query, ...)
```

Arguments

`query`	SQL query string
`...`	Additional arguments passed to Lake$sql()

Value

Query results

Examples

```
use_lake("ex_sql", root = tempfile())
put("t", data.frame(x = 1:3))
sql("SELECT COUNT(*) AS n FROM t")
```

starts_with_str *Check if string starts with a prefix*

Description

Check if string starts with a prefix

Usage

```
starts_with_str(x, prefix)
```

Arguments

x	Character vector
prefix	Prefix to check for

Value

Logical vector

Examples

```
genes <- c("MT-CO1", "MT-CO2", "ACTB")
genes[starts_with_str(genes, "MT-")]
```

tables *List tables in the default lake*

Description

List tables in the default lake

Usage

```
tables()
```

Value

Data frame of table names

Examples

```
use_lake("ex_tables", root = tempfile())
put("t", data.frame(x = 1:3))
tables()
```

tag *Tag data in the default lake*

Description

Tag data in the default lake

Usage

```
tag(name, tag)
```

Arguments

name	Data name
tag	Tag to apply

Value

Invisible Lake object

Examples

```
use_lake("ex_tag", root = tempfile())
put("counts", data.frame(x = 1:3))
tag("counts", "raw")
```

trace_calls *Trace function calls for lineage*

Description

Temporarily hooks into specified functions to track their calls. More invasive than observe() but provides finer-grained control.

Usage

```
trace_calls(functions, expr, lake = NULL)
```

Arguments

functions	Named list of functions to trace
expr	Expression to evaluate while tracing
lake	Optional lake to record to

Value

List with result and call trace

Examples

```
lake <- Lake$new("ex_trace_calls", root = tempfile())
trace_calls(character(0), { x <- 1 }, lake = lake)
```

`track_pipeline` *Track a pipeline block into a Lake with minimal boilerplate*

Description

Uses ‘observe_to_lake()‘ under the hood and optionally creates a snapshot after successful execution.

Usage

```
track_pipeline(
  expr,
  project = NULL,
  prefix = "file:",
  snapshot = NULL,
  track_functions = NULL,
  save_result = FALSE,
  result_name = NULL,
  result_depends_on = c("writes", "reads", "both", "none"),
  store_observation = FALSE,
  observation_name = NULL,
  observation_depends_on = c("writes", "reads", "both", "none"),
  ...
)
```

Arguments

`expr`	Expression to execute and track
`project`	Optional project name. If set, calls ‘use_lake(project, ...)‘.
`prefix`	Prefix for file-based node names in the lake
`snapshot`	Optional snapshot label created after successful execution
`track_functions`	Character vector forwarded to ‘observe()‘
`save_result`	If TRUE, store the expression result in the Lake
`result_name`	Optional target name when ‘save_result = TRUE‘
`result_depends_on`	Dependencies used for saved result: ‘"writes"‘, ‘"reads"‘, ‘"both"‘, or ‘"none"‘

store_observation
: If TRUE, store observed reads/writes/lineage as an object

observation_name
: Optional target name when ‘store_observation = TRUE‘

observation_depends_on
: Dependencies used for observation record: ‘"writes"‘, ‘"reads"‘, ‘"both"‘, or ‘"none"‘

...
: Additional arguments passed to ‘use_lake()‘ when ‘project‘ is provided

Value

The result of evaluating ‘expr‘

Examples

```
if (FALSE) {
    # Track an existing block with the current default lake
    use_lake("rna_seq_project")
    track_pipeline(
        {
            x <- read.csv("counts.csv")
            write.csv(x, "counts_copy.csv", row.names = FALSE)
        },
        snapshot = "ingest_v1",
        save_result = TRUE,
        result_name = "counts_copy_summary",
        store_observation = TRUE,
        observation_name = "obs_ingest_v1"
    )
}
```

---

track_script *Track an existing analysis script with one function call*

Description

Sources a script with ‘local = TRUE‘ inside observation mode so common unqualified I/O calls in the script can be intercepted.

Usage

```
track_script(
  path,
  project = NULL,
  prefix = "file:",
  snapshot = NULL,
  track_functions = NULL,
  save_result = FALSE,
```

```
  result_name = NULL,
  result_depends_on = c("writes", "reads", "both", "none"),
  store_observation = FALSE,
  observation_name = NULL,
  observation_depends_on = c("writes", "reads", "both", "none"),
  chdir = FALSE,
  echo = FALSE,
  source_args = list(),
  ...
)
```

Arguments

path	Script path passed to 'source()'
project	Optional project name. If set, calls 'use_lake(project, ...)'.
prefix	Prefix for file-based node names in the lake
snapshot	Optional snapshot label created after successful execution
track_functions	Character vector forwarded to 'observe()'
save_result	If TRUE, store the script return value in the Lake
result_name	Optional target name when 'save_result = TRUE'
result_depends_on	Dependencies used for saved result: '"writes"', '"reads"', '"both"', or '"none"'
store_observation	If TRUE, store observed reads/writes/lineage as an object
observation_name	Optional target name when 'store_observation = TRUE'
observation_depends_on	Dependencies used for observation record: '"writes"', '"reads"', '"both"', or '"none"'
chdir	Passed to 'source()'
echo	Passed to 'source()'
source_args	Named list of additional arguments passed to 'source()'. Reserved names ('file', 'local', 'chdir', 'echo') are not allowed.
...	Additional arguments passed to 'use_lake()' when 'project' is provided

Value

The return value of the sourced script ('source(...)$value')

Examples

```
if (FALSE) {
track_script("analysis_pipeline.R", project = "rna_seq_project", snapshot =
"run_2026_02_21")
}
```

`TranscriptomicsAdapter` *Transcriptomics Adapter*

Description

Adapter for storing and retrieving transcriptomics-layer objects.

Details

This umbrella adapter is lower priority than specific adapters (e.g. SingleCellExperiment / SummarizedExperiment) and captures transcriptomics-marked objects.

Value

An R6 generator for a `LakeAdapter` subclass that serializes and restores objects of this omics layer.

Super class

`OmicsLake::LakeAdapter` -> `TranscriptomicsAdapter`

Methods

Public methods:

- `TranscriptomicsAdapter$name()`
- `TranscriptomicsAdapter$priority()`
- `TranscriptomicsAdapter$put()`
- `TranscriptomicsAdapter$get()`
- `TranscriptomicsAdapter$components()`
- `TranscriptomicsAdapter$exists()`
- `TranscriptomicsAdapter$list_names()`
- `TranscriptomicsAdapter$can_handle()`
- `TranscriptomicsAdapter$clone()`

Method `name()`:

Usage:

```
TranscriptomicsAdapter$name()
```

Method `priority()`:

Usage:

```
TranscriptomicsAdapter$priority()
```

Method `put()`:

Usage:

```
TranscriptomicsAdapter$put(lake, name, data)
```

Method `get()`**:**

Usage:

```
TranscriptomicsAdapter$get(lake, name, ref = "@latest")
```

Method `components()`**:**

Usage:

```
TranscriptomicsAdapter$components(lake, name)
```

Method `exists()`**:**

Usage:

```
TranscriptomicsAdapter$exists(lake, name)
```

Method `list_names()`**:**

Usage:

```
TranscriptomicsAdapter$list_names(lake)
```

Method `can_handle()`**:**

Usage:

```
TranscriptomicsAdapter$can_handle(data)
```

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
TranscriptomicsAdapter$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
adapter <- TranscriptomicsAdapter$new()
class(adapter)
```

`tree` *Show lineage tree from the default lake*

Description

Show lineage tree from the default lake

Usage

```
tree(name = NULL, ...)
```

Arguments

`name`	Starting node (optional)
`...`	Additional arguments passed to Lake$tree()

Value

Lineage data frame

Examples

```
use_lake("ex_tree", root = tempfile())
put("raw", data.frame(x = 1:3))
tree()
```

`unlink_dep` *Unlink a dependency*

Description

Removes a dependency relationship between two nodes.

Usage

```
unlink_dep(from, to, lake = NULL)
```

Arguments

`from`	Source node name
`to`	Target node name
`lake`	Lake instance (uses default if NULL)

Value

Invisible TRUE on success

Examples

```
use_lake("ex_unlink_dep", root = tempfile())
put("raw", data.frame(x = 1:3))
put("filtered", data.frame(x = 1:2), depends_on = "raw")
unlink_dep("raw", "filtered")
```

use_lake *Set or get the default lake*

Description

Set or get the default lake

Usage

```
use_lake(project = NULL, ...)
```

Arguments

project	Project name. If provided, creates/opens a lake and sets it as default.
...	Additional arguments passed to Lake$new()

Value

The default Lake object (invisibly when setting)

Examples

```
use_lake("ex_use_lake", root = tempfile())
current <- use_lake() # Get current lake
```

VCFAdapter *VCF Adapter*

Description

Adapter for storing and retrieving VariantAnnotation VCF objects.

Details

This adapter stores a full-fidelity serialized VCF object and manifest.

Value

An R6 generator for a `LakeAdapter` subclass that serializes and restores objects of this omics layer.

Super class

`OmicsLake::LakeAdapter` -> `VCFAdapter`

Methods

Public methods:

Method `name()`**:**

Usage:

```
VCFAdapter$name()
```

Method `priority()`**:**

Usage:

```
VCFAdapter$priority()
```

Method `put()`**:**

Usage:

```
VCFAdapter$put(lake, name, data)
```

Method `get()`**:**

Usage:

```
VCFAdapter$get(lake, name, ref = "@latest")
```

Method `components()`**:**

Usage:

```
VCFAdapter$components(lake, name)
```

Method `exists()`**:**

Usage:

```
VCFAdapter$exists(lake, name)
```

Method `list_names()`**:**

Usage:

```
VCFAdapter$list_names(lake)
```

Method `can_handle()`**:**

Usage:

```
VCFAdapter$can_handle(data)
```

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
VCFAdapter$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

Examples

```
adapter <- VCFAdapter$new()
class(adapter)
```

`with_tracking` *Track a code block with explicit lake integration*

Description

A simpler alternative to observe() that directly integrates with Lake. Wraps code execution and records any lake operations.

Usage

```
with_tracking(lake, name, expr)
```

Arguments

`lake`	Lake instance
`name`	Name for the output in the lineage
`expr`	Expression to track

Value

The result of the expression

Examples

```
if (FALSE) {
    lake <- Lake$new("project")

    result <- with_tracking(lake, "analysis_result", {
        data <- lake$get("raw_data")
        processed <- transform(data)
        processed
    })

    # Result is automatically saved with tracked dependencies
    lake$tree("analysis_result")
}
```

wrap *Function Wrapping for Lineage Tracking*

Description

Wrap existing functions to automatically track data lineage. Enables non-invasive integration with existing analysis pipelines.

Details

Function wrapping provides a way to add lineage tracking to existing code without modifying the original functions. Wrapped functions automatically record their inputs and outputs to the lineage graph.

Value

See the individual function help pages for return values.

Examples

```
if (FALSE) {
    lake <- Lake$new("project")

    # Wrap a function
    tracked_normalize <- wrap_fn(normalize_data, lake, "normalized")

    # Use the wrapped function
    result <- tracked_normalize(raw_data)
    # Dependencies are automatically recorded

    # Or wrap and use inline
    result <- lake |>
        wrap_call(normalize_data, raw_data, output = "normalized")
}
```

wrap_call *Wrap a function call inline*

Description

Execute a function call with lineage tracking, without creating a persistent wrapper.

Usage

```
wrap_call(lake, fn, ..., output = NULL, save = TRUE)
```

Arguments

lake	Lake instance
fn	Function to call
...	Arguments to pass to fn
output	Name for the output in the lineage
save	If TRUE, save the result to the lake

Value

The result of calling fn

Examples

```
if (FALSE) {
    lake <- Lake$new("project")

    result <- wrap_call(lake, mean, c(1, 2, 3, NA),
        na.rm = TRUE,
        output = "mean_result", save = FALSE
    )
}
```

---

wrap_fn *Wrap a function with lineage tracking*

Description

Creates a wrapped version of a function that automatically records its execution in the lake's lineage graph.

Usage

```
wrap_fn(fn, lake, output_name, input_names = NULL, save_output = TRUE)
```

Arguments

fn	Function to wrap
lake	Lake instance for lineage recording
output_name	Name for the output in the lineage graph
input_names	Optional character vector specifying which arguments should be recorded as dependencies. If NULL, attempts to detect lake-sourced data automatically.
save_output	If TRUE, automatically saves the output to the lake

Value

A wrapped function with the same signature as fn

Examples

```
lake <- Lake$new("ex_wrap_fn", root = tempfile())
lake$put("raw", data.frame(x = 1:3))
tracked <- wrap_fn(function(d) d, lake, "out")
invisible(tracked(lake$get("raw")))
```

`XCMSAdapter` *XCMS Adapter*

Description

Adapter for storing and retrieving xcms objects (`XCMSnExp` and `XcmsExperiment`).

Details

This adapter currently guarantees full-fidelity roundtrip by storing the complete object and manifest as internal components.

Value

An R6 generator for a `LakeAdapter` subclass that serializes and restores objects of this omics layer.

Super class

`OmicsLake::LakeAdapter` -> `XCMSAdapter`

Methods

Public methods:

- `XCMSAdapter$name()`
- `XCMSAdapter$can_handle()`
- `XCMSAdapter$priority()`
- `XCMSAdapter$put()`
- `XCMSAdapter$get()`
- `XCMSAdapter$components()`
- `XCMSAdapter$exists()`
- `XCMSAdapter$list_names()`
- `XCMSAdapter$clone()`

Method `name()`:

Usage:

```
XCMSAdapter$name()
```

Method `can_handle()`:

Usage:

```
XCMSAdapter$can_handle(data)
```

Method `priority()`**:**

Usage:

```
XCMSAdapter$priority()
```

Method `put()`**:**

Usage:

```
XCMSAdapter$put(lake, name, data)
```

Method `get()`**:**

Usage:

```
XCMSAdapter$get(lake, name, ref = "@latest")
```

Method `components()`**:**

Usage:

```
XCMSAdapter$components(lake, name)
```

Method `exists()`**:**

Usage:

```
XCMSAdapter$exists(lake, name)
```

Method `list_names()`**:**

Usage:

```
XCMSAdapter$list_names(lake)
```

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
XCMSAdapter$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
adapter <- XCMSAdapter$new()
class(adapter)
```

Index