Package 'mlr3db'

November 4, 2026

Title Data Base Backend for 'mlr3'

Version 0.7.3

Description Extends the 'mlr3' package with a backend to transparently work with databases such as 'SQLite', 'DuckDB', 'MySQL', 'MariaDB', or 'PostgreSQL'. The package provides three additional backends: 'DataBackendDplyr' relies on the abstraction of package 'dbplyr' to interact with most DBMS. 'DataBackendDuckDB' operates on 'DuckDB' data bases and also on Apache Parquet files. 'DataBackendPolars' operates on 'Polars' data frames.

License LGPL-3

URL https://mlr3db.mlr-org.com, https://github.com/mlr-org/mlr3db

BugReports https://github.com/mlr-org/mlr3db/issues

Depends mlr3 (>= 1.5.0), R (>= 3.1.0)

Imports backports, checkmate, data.table, mlr3misc (>= 0.10.0), R6

Suggests DBI, dbplyr, dplyr, duckdb (>= 0.4.0), future, future.apply, future.callr, hms, lgr, polars (>= 1.16.0), RSQLite, rlang, testthat (>= 3.0.0), tibble

Additional_repositories https://community.r-multiverse.org

Config/testthat/edition 3

Encoding UTF-8

Config/roxygen2/version 8.1.0

NeedsCompilation no

Author Michel Lang [aut] (ORCID: <https://orcid.org/0000-0001-9754-0393>), Lona Koers [aut], Marc Becker [cre, aut] (ORCID: <https://orcid.org/0000-0002-8115-0400>)

Maintainer Marc Becker <marcbecker@posteo.de>

Repository CRAN

Date/Publication 2026-11-03 23:10:02 UTC

Contents

mlr3db-package 2
as_duckdb_backend 3
as_polars_backend 4
as_sqlite_backend 4
DataBackendDplyr 5
DataBackendDuckDB 9
DataBackendPolars 12

Index **16**

`mlr3db-package` *mlr3db: Data Base Backend for 'mlr3'*

Description

Extends the 'mlr3' package with a backend to transparently work with databases such as 'SQLite', 'DuckDB', 'MySQL', 'MariaDB', or 'PostgreSQL'. The package provides three additional backends: 'DataBackendDplyr' relies on the abstraction of package 'dbplyr' to interact with most DBMS. 'DataBackendDuckDB' operates on 'DuckDB' data bases and also on Apache Parquet files. 'DataBackendPolars' operates on 'Polars' data frames.

Options

- `mlr3db.sqlite_dir`: Default directory to store SQLite databases constructed with `as_sqlite_backend()`..
- `mlr3db.sqlite_dir`: Default directory to store DuckDB databases constructed with `as_duckdb_backend()`..

Author(s)

Maintainer: Marc Becker <marcbecker@posteo.de> (ORCID)

Authors:

- Marc Becker <marcbecker@posteo.de> (ORCID)
- Michel Lang <michellang@gmail.com> (ORCID)
- Lona Koers <lona.koers@gmail.com>

See Also

Useful links:

- https://mlr3db.mlr-org.com
- https://github.com/mlr-org/mlr3db
- Report bugs at https://github.com/mlr-org/mlr3db/issues

as_duckdb_backend *Convert to DuckDB Backend*

Description

Converts to a DataBackendDuckDB using the **duckdb** database, depending on the input type:

- `data.frame`: Creates a new mlr3::DataBackendDataTable first using `mlr3::as_data_backend()`, then proceeds with the conversion from mlr3::DataBackendDataTable to DataBackendDuckDB.
- mlr3::DataBackend: Creates a new DuckDB data base in the specified path. The filename is determined by the hash of the mlr3::DataBackend. If the file already exists, a connection to the existing database is established and the existing files are reused.

The created backend automatically reconnects to the database if the connection was lost, e.g. because the object was serialized to the filesystem and restored in a different R session. The only requirement is that the path does not change and that the path is accessible on all workers.

Usage

```
as_duckdb_backend(data, path = getOption("mlr3db.duckdb_dir", ":temp:"), ...)
```

Arguments

`data`	(`data.frame()` | mlr3::DataBackend) See description.
`path`	(`character(1)`) Path for the DuckDB databases. Either a valid path to a directory which will be created if it not exists, or one of the special strings: • `":temp:"` (default): Temporary directory of the R session is used, see `tempdir()`. Note that this directory will be removed during the shutdown of the R session. Also note that this usually does not work for parallelization on remote workers. Set to a custom path instead or use special string `":user:"` instead. • `":user:"`: User cache directory as returned by `R_user_dir()` is used. The default for this argument can be configured via option `"mlr3db.sqlite_dir"` or `"mlr3db.duckdb_dir"`, respectively. The database files will use the hash of the mlr3::DataBackend as filename with file extension `".duckdb"` or `".sqlite"`. If the database already exists on the file system, the converters will just established a new read-only connection.
`...`	(any) Additional arguments, passed to DataBackendDuckDB.

Value

DataBackendDuckDB or mlr3::Task.

as_polars_backend *Convert to Polars Backend*

Description

Converts to a DataBackendPolars using the **polars** database, depending on the input type:

- `data.frame`: Creates a new mlr3::DataBackendDataTable first using `mlr3::as_data_backend()`, then proceeds with the conversion from mlr3::DataBackendDataTable to DataBackendPolars.
- mlr3::DataBackend: Creates a new DataBackendPolars.

There is no automatic connection to the origin file set. If the data is obtained using scanning and the data is streamed, a `connector` can be set manually but is not required.

Usage

```
as_polars_backend(data, streaming = FALSE, ...)
```

Arguments

`data`	(`data.frame()` | mlr3::DataBackend) See description.
`streaming`	(`logical(1)`) Whether the data should be only scanned (recommended for large data sets) and streamed with every DataBackendPolars operation or loaded into memory completely.
`...`	(any) Additional arguments, passed to DataBackendPolars.

Value

DataBackendPolars or mlr3::Task.

as_sqlite_backend *Convert to SQLite Backend*

Description

Converts to a DataBackendDplyr using a **RSQLite** database, depending on the input type:

- `data.frame`: Creates a new mlr3::DataBackendDataTable first using `mlr3::as_data_backend()`, then proceeds with the conversion from mlr3::DataBackendDataTable to DataBackendDplyr.
- mlr3::DataBackend: Creates a new SQLite data base in the specified path. The filename is determined by the hash of the mlr3::DataBackend. If the file already exists, a connection to the existing database is established and the existing files are reused.

The created backend automatically reconnects to the database if the connection was lost, e.g. because the object was serialized to the filesystem and restored in a different R session. The only requirement is that the path does not change and that the path is accessible on all workers.

Usage

```
as_sqlite_backend(data, path = getOption("mlr3db.sqlite_dir", ":temp:"), ...)
```

Arguments

`data`	(`data.frame()` | mlr3::DataBackend See description.
`path`	(`character(1)`) Path for the DuckDB databases. Either a valid path to a directory which will be created if it not exists, or one of the special strings: • `":temp:"` (default): Temporary directory of the R session is used, see `tempdir()`. Note that this directory will be removed during the shutdown of the R session. Also note that this usually does not work for parallelization on remote workers. Set to a custom path instead or use special string `":user:"` instead. • `":user:"`: User cache directory as returned by `R_user_dir()` is used. The default for this argument can be configured via option `"mlr3db.sqlite_dir"` or `"mlr3db.duckdb_dir"`, respectively. The database files will use the hash of the mlr3::DataBackend as filename with file extension `".duckdb"` or `".sqlite"`. If the database already exists on the file system, the converters will just established a new read-only connection.
`...`	(any) Additional arguments, passed to DataBackendDplyr.

Value

DataBackendDplyr or mlr3::Task.

DataBackendDplyr *DataBackend for dplyr/dbplyr*

Description

A mlr3::DataBackend using `dplyr::tbl()` from packages **dplyr**/**dbplyr**. This includes `tibbles` and abstract database connections interfaced by **dbplyr**. The latter allows mlr3::Tasks to interface an out-of-memory database.

Super class

`mlr3::DataBackend` -> `DataBackendDplyr`

Public fields

`levels` (named `list()`)
: List (named with column names) of factor levels as `character()`. Used to auto-convert character columns to factor variables.

`connector` (`function()`)
: Function which is called to re-connect in case the connection became invalid.

Active bindings

`rownames (integer())`
: Returns vector of all distinct row identifiers, i.e. the contents of the primary key column.

`colnames (character())`
: Returns vector of all column names, including the primary key column.

`nrow (integer(1))`
: Number of rows (observations).

`ncol (integer(1))`
: Number of columns (variables), including the primary key column.

`valid (logical(1))`
: Returns `NA` if the data does not inherits from `"tbl_sql"` (i.e., it is not a real SQL data base). Returns the result of `DBI::dbIsValid()` otherwise.

Methods

Public methods:

- `DataBackendDplyr$new()`
- `DataBackendDplyr$data()`
- `DataBackendDplyr$head()`
- `DataBackendDplyr$distinct()`
- `DataBackendDplyr$missings()`

`DataBackendDplyr$new()`**:** Creates a backend for a `dplyr::tbl()` object.

Usage:

```
DataBackendDplyr$new(
  data,
  primary_key,
  strings_as_factors = TRUE,
  connector = NULL
)
```

Arguments:

`data (dplyr::tbl())`
: The data object.
Instead of calling the constructor yourself, you can call `mlr3::as_data_backend()` on a `dplyr::tbl()`. Note that only objects of class `"tbl_lazy"` will be converted to a DataBackendDplyr (this includes all connectors from **dbplyr**). Local `"tbl"` objects such as `tibbles` will converted to a mlr3::DataBackendDataTable.

`primary_key (character(1))`
: Name of the primary key column.

`strings_as_factors (logical(1) || character())`
: Either a character vector of column names to convert to factors, or a single logical flag: if `FALSE`, no column will be converted, if `TRUE` all string columns (except the primary key). For conversion, the backend is queried for distinct values of the respective columns on construction and their levels are stored in `$levels`.

`connector` (function())\cr If not NULL', a function which re-connects to the database in case the connection has become invalid. Database connections can become invalid due to timeouts or if the backend is serialized to the file system and then de-serialized again. This round trip is often performed for parallelization, e.g. to send the objects to remote workers. `DBI::dbIsValid()` is called to validate the connection. The function must return just the connection, not a `dplyr::tbl()` object! Note that this this function is serialized together with the backend, including possible sensitive information such as login credentials. These can be retrieved from the stored mlr3::DataBackend/mlr3::Task. To protect your credentials, it is recommended to use the **secret** package.

`DataBackendDplyr$data()`: Returns a slice of the data. Calls `dplyr::filter()` and `dplyr::select()` on the table and converts it to a `data.table::data.table()`.

The rows must be addressed as vector of primary key values, columns must be referred to via column names. Queries for rows with no matching row id and queries for columns with no matching column name are silently ignored. Rows are guaranteed to be returned in the same order as `rows`, columns may be returned in an arbitrary order. Duplicated row ids result in duplicated rows, duplicated column names lead to an exception.

Usage:

```
DataBackendDplyr$data(rows, cols)
```

Arguments:

`rows integer()`
Row indices.

`cols character()`
Column names.

`DataBackendDplyr$head()`: Retrieve the first `n` rows.

Usage:

```
DataBackendDplyr$head(n = 6L)
```

Arguments:

`n (integer(1))`
Number of rows.

Returns: `data.table::data.table()` of the first `n` rows.

`DataBackendDplyr$distinct()`: Returns a named list of vectors of distinct values for each column specified. If `na_rm` is `TRUE`, missing values are removed from the returned vectors of distinct values. Non-existing rows and columns are silently ignored.

Usage:

```
DataBackendDplyr$distinct(rows, cols, na_rm = TRUE)
```

Arguments:

`rows integer()`
Row indices.

`cols character()`
Column names.

`na_rm logical(1)`
Whether to remove NAs or not.

Returns: Named `list()` of distinct values.

`DataBackendDplyr$missings()`: Returns the number of missing values per column in the specified slice of data. Non-existing rows and columns are silently ignored.

Usage:

```
DataBackendDplyr$missings(rows, cols)
```

Arguments:

`rows integer()`
: Row indices.

`cols character()`
: Column names.

Returns: Total of missing values per column (named `numeric()`).

Examples

```
if (mlr3misc::require_namespaces(c("tibble", "RSQLite", "dbplyr"), quietly = TRUE)) {
  # Backend using a in-memory tibble
  data = tibble::as_tibble(iris)
  data$Sepal.Length[1:30] = NA
  data$row_id = 1:150
  b = DataBackendDplyr$new(data, primary_key = "row_id")

  # Object supports all accessors of DataBackend
  print(b)
  b$nrow
  b$ncol
  b$colnames
  b$data(rows = 100:101, cols = "Species")
  b$distinct(b$rownames, "Species")

  # Classification task using this backend
  task = mlr3::TaskClassif$new(id = "iris_tibble", backend = b, target = "Species")
  print(task)
  head(task)

  # Create a temporary SQLite database
  con = DBI::dbConnect(RSQLite::SQLite(), ":memory:")
  dplyr::copy_to(con, data)
  tbl = dplyr::tbl(con, "data")

  # Define a backend on a subset of the database: do not use column "Sepal.Width"
  tbl = dplyr::select_at(tbl, setdiff(colnames(tbl), "Sepal.Width"))
  tbl = dplyr::filter(tbl, row_id %in% 1:120) # Use only first 120 rows
  b = DataBackendDplyr$new(tbl, primary_key = "row_id")
  print(b)

  # Query distinct values
  b$distinct(b$rownames, "Species")

  # Query number of missing values
```

```
  b$missings(b$rownames, b$colnames)

 # Note that SQLite does not support factors, column Species has been converted to character
  lapply(b$head(), class)

  # Cleanup
  rm(tbl)
  DBI::dbDisconnect(con)
}
```

DataBackendDuckDB *DataBackend for DuckDB*

Description

A mlr3::DataBackend for **duckdb**. Can be easily constructed with `as_duckdb_backend()`.

Super class

`mlr3::DataBackend` -> `DataBackendDuckDB`

Public fields

`levels` (named `list()`)
: List (named with column names) of factor levels as `character()`. Used to auto-convert character columns to factor variables.

`connector` (`function()`)
: Function which is called to re-connect in case the connection became invalid.

`table` (`character(1)`)
: Data base table or view to operate on.

Active bindings

`table_info` (`data.frame()`)
: Data frame as returned by pragma `table_info()`.

`rownames` (`integer()`)
: Returns vector of all distinct row identifiers, i.e. the contents of the primary key column.

`colnames` (`character()`)
: Returns vector of all column names, including the primary key column.

`nrow` (`integer(1)`)
: Number of rows (observations).

`ncol` (`integer(1)`)
: Number of columns (variables), including the primary key column.

`valid` (`logical(1)`)
: Returns `NA` if the data does not inherits from `"tbl_sql"` (i.e., it is not a real SQL data base). Returns the result of `DBI::dbIsValid()` otherwise.

Methods

Public methods:

- DataBackendDuckDB$new()
- DataBackendDuckDB$data()
- DataBackendDuckDB$head()
- DataBackendDuckDB$distinct()
- DataBackendDuckDB$missings()

`DataBackendDuckDB$new()`: Creates a backend for a `duckdb::duckdb()` database.

Usage:

```
DataBackendDuckDB$new(
  data,
  table,
  primary_key,
  strings_as_factors = TRUE,
  connector = NULL
)
```

Arguments:

`data` (connection)
: A connection created with `DBI::dbConnect()`. If constructed manually (and not via the helper function `as_duckdb_backend()`, make sure that there exists an (unique) index for the key column.

`table` (`character(1)`)
: Table or view to operate on.

`primary_key` (`character(1)`)
: Name of the primary key column.

`strings_as_factors` (`logical(1)` || `character()`)
: Either a character vector of column names to convert to factors, or a single logical flag: if `FALSE`, no column will be converted, if `TRUE` all string columns (except the primary key). For conversion, the backend is queried for distinct values of the respective columns on construction and their levels are stored in `$levels`.

`connector` (function())\cr If not NULL‘, a function which re-connects to the database in case the connection has become invalid. Database connections can become invalid due to timeouts or if the backend is serialized to the file system and then de-serialized again. This round trip is often performed for parallelization, e.g. to send the objects to remote workers. `DBI::dbIsValid()` is called to validate the connection. The function must return just the connection, not a `dplyr::tbl()` object! Note that this this function is serialized together with the backend, including possible sensitive information such as login credentials. These can be retrieved from the stored mlr3::DataBackend/mlr3::Task. To protect your credentials, it is recommended to use the **secret** package.

`DataBackendDuckDB$data()`: Returns a slice of the data.

The rows must be addressed as vector of primary key values, columns must be referred to via column names. Queries for rows with no matching row id and queries for columns with no matching column name are silently ignored. Rows are guaranteed to be returned in the same order as `rows`, columns may be returned in an arbitrary order. Duplicated row ids result in duplicated rows, duplicated column names lead to an exception.

Usage:

```
DataBackendDuckDB$data(rows, cols)
```

Arguments:

`rows integer()`
: Row indices.

`cols character()`
: Column names.

`DataBackendDuckDB$head()`: Retrieve the first `n` rows.

Usage:

```
DataBackendDuckDB$head(n = 6L)
```

Arguments:

`n (integer(1))`
: Number of rows.

Returns: `data.table::data.table()` of the first `n` rows.

`DataBackendDuckDB$distinct()`: Returns a named list of vectors of distinct values for each column specified. If `na_rm` is `TRUE`, missing values are removed from the returned vectors of distinct values. Non-existing rows and columns are silently ignored.

Usage:

```
DataBackendDuckDB$distinct(rows, cols, na_rm = TRUE)
```

Arguments:

`rows integer()`
: Row indices.

`cols character()`
: Column names.

`na_rm logical(1)`
: Whether to remove NAs or not.

Returns: Named `list()` of distinct values.

`DataBackendDuckDB$missings()`: Returns the number of missing values per column in the specified slice of data. Non-existing rows and columns are silently ignored.

Usage:

```
DataBackendDuckDB$missings(rows, cols)
```

Arguments:

`rows integer()`
: Row indices.

`cols character()`
: Column names.

Returns: Total of missing values per column (named `numeric()`).

See Also

https://duckdb.org/

DataBackendPolars *DataBackend for Polars*

Description

A mlr3::DataBackend using LazyFrame from package polars. Can be easily constructed with as_polars_backend(). mlr3::Tasks can interface out-of-memory files if the polars::polars_lazy_frame was imported using a polars::scan_x function. Streaming, a polars alpha feature, is always enabled, but only used when applicable. A connector is not required but can be useful e.g. for scanning larger than memory files

Super class

mlr3::DataBackend -> DataBackendPolars

Public fields

levels (named list())
: List (named with column names) of factor levels as character(). Used to auto-convert character columns to factor variables.

connector (function())
: Function which is called to re-connect in case the connection became invalid.

Active bindings

rownames (integer())
: Returns vector of all distinct row identifiers, i.e. the contents of the primary key column.

colnames (character())
: Returns vector of all column names, including the primary key column.

nrow (integer(1))
: Number of rows (observations).

ncol (integer(1))
: Number of columns (variables), including the primary key column.

Methods

Public methods:

- DataBackendPolars$new()
- DataBackendPolars$data()
- DataBackendPolars$head()
- DataBackendPolars$distinct()
- DataBackendPolars$missings()

DataBackendPolars$new(): Creates a backend for a polars::polars_data_frame object.

Usage:

```
DataBackendPolars$new(
  data,
  primary_key,
  strings_as_factors = TRUE,
  connector = NULL
)
```

Arguments:

`data` (polars::polars_lazy_frame)
: The data object.
Instead of calling the constructor itself, please call `mlr3::as_data_backend()` on a polars::polars_lazy_frame or polars::polars_data_frame. Note that only polars::polars_lazy_frames will be converted to a DataBackendPolars. polars::polars_data_frame objects without lazy execution will be converted to a mlr3::DataBackendDataTable.

`primary_key` (`character(1)`)
: Name of the primary key column. Because `polars` does not natively support primary keys, uniqueness of the primary key column is expected but not enforced.

`strings_as_factors` (`logical(1)` | `character()`)
: Either a character vector of column names to convert to factors, or a single logical flag: if `FALSE`, no column will be converted, if `TRUE` all string columns (except the primary key). For conversion, the backend is queried for distinct values of the respective columns on construction and their levels are stored in `$levels`.

`connector` (`function()`)
: Optional function which is called to re-connect to e.g. a source file in case the connection became invalid.

`DataBackendPolars$data()`: Returns a slice of the data.

The rows must be addressed as vector of primary key values, columns must be referred to via column names. Queries for rows with no matching row id and queries for columns with no matching column name are silently ignored.

Usage:

```
DataBackendPolars$data(rows, cols)
```

Arguments:

`rows` (`integer()`)
: Row indices.

`cols` (`character()`)
: Column names.

`DataBackendPolars$head()`: Retrieve the first `n` rows.

Usage:

```
DataBackendPolars$head(n = 6L)
```

Arguments:

`n` (`integer(1)`)
: Number of rows.

Returns: `data.table::data.table()` of the first `n` rows.

`DataBackendPolars$distinct()`: Returns a named list of vectors of distinct values for each column specified. If `na_rm` is `TRUE`, missing values are removed from the returned vectors of distinct values. Non-existing rows and columns are silently ignored.

Usage:

```
DataBackendPolars$distinct(rows, cols, na_rm = TRUE)
```

Arguments:

`rows (integer())`
Row indices.

`cols (character())`
Column names.

`na_rm (logical(1))`
Whether to remove NAs or not.

Returns: Named `list()` of distinct values.

`DataBackendPolars$missings()`: Returns the number of missing values per column in the specified slice of data. Non-existing rows and columns are silently ignored.

Usage:

```
DataBackendPolars$missings(rows, cols)
```

Arguments:

`rows (integer())`
Row indices.

`cols (character())`
Column names.

Returns: Total of missing values per column (named `numeric()`).

See Also

https://pola-rs.github.io/r-polars/

Examples

```
if (mlr3misc::require_namespaces("polars", quietly = TRUE)) {
  # Backend using a in-memory data set
  data = iris
  data$Sepal.Length[1:30] = NA
  data$row_id = 1:150
  data = polars::as_polars_lf(data)
  b = DataBackendPolars$new(data, primary_key = "row_id")

  # Object supports all accessors of DataBackend
  print(b)
  b$nrow
  b$ncol
  b$colnames
  b$data(rows = 100:101, cols = "Species")
  b$distinct(b$rownames, "Species")
```

```
  # Classification task using this backend
  task = mlr3::TaskClassif$new(id = "iris_polars", backend = b, target = "Species")
  print(task)
  head(task)

  # Write a parquet file to scan
  data$collect()$write_parquet("iris.parquet")
  data = polars::pl$scan_parquet("iris.parquet")

  # Backend that re-reads the parquet file if the connection fails
  b = DataBackendPolars$new(data, "row_id",
    connector = function() polars::pl$scan_parquet("iris.parquet"))
  print(b)

  # Define a backend on a subset of the database: do not use column "Sepal.Width"
  data = data$select(
    polars::pl$col(names = setdiff(colnames(data), "Sepal.Width"))
  )$head(120) # Use only first 120 rows

  # Backend with only scanned data
  b = DataBackendPolars$new(data, "row_id", strings_as_factors = TRUE)
  print(b)

  # Query distinct values
  b$distinct(b$rownames, "Species")

  # Query number of missing values
  b$missings(b$rownames, b$colnames)

  # Cleanup
  if (file.exists("iris.parquet")) {
    file.remove("iris.parquet")
  }
}
```

Index

as_duckdb_backend, 3
as_duckdb_backend(), *2*, *9, 10*
as_polars_backend, 4
as_polars_backend(), *12*
as_sqlite_backend, 4
as_sqlite_backend(), *2*

data.table::data.table(), *7*, *11*, *13*
DataBackendDplyr, *4, 5*, 5, *6*
DataBackendDuckDB, *3*, 9
DataBackendPolars, *4*, 12, *13*
DBI::dbConnect(), *10*
DBI::dbIsValid(), *6, 7*, *9, 10*
dplyr::filter(), *7*
dplyr::select(), *7*
dplyr::tbl(), *5–7*, *10*
duckdb::duckdb(), *10*

mlr3::as_data_backend(), *3, 4*, *6*, *13*
mlr3::DataBackend, *3–5*, *7*, *9, 10*, *12*
mlr3::DataBackendDataTable, *3, 4*, *6*, *13*
mlr3::Task, *3–5*, *7*, *10*, *12*
mlr3db *(*mlr3db-package*)*, 2
mlr3db-package, 2

polars::polars_data_frame, *12, 13*
polars::polars_lazy_frame, *13*

R_user_dir(), *3*, *5*

tempdir(), *3*, *5*
tibbles, *5, 6*