# Package 'mlr3db'

November 4, 2023

**Title** Data Base Backend for 'mlr3'

**Version** 0.5.2

**Description** Extends the 'mlr3' package with a backend to
transparently work with databases such as 'SQLite', 'DuckDB', 'MySQL',
'MariaDB', or 'PostgreSQL'. The package provides two 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.

**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 (>= 0.13.0), R (>= 3.1.0)

**Imports** R6 (>= 2.4.0), backports, checkmate, data.table, mlr3misc (>=
0.10.0)

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

**Encoding** UTF-8

**Config/testthat/edition** 3

**RoxygenNote** 7.2.3

**NeedsCompilation** no

**Author** Michel Lang [cre, aut] (<https://orcid.org/0000-0001-9754-0393>)

**Maintainer** Michel Lang <michellang@gmail.com>

**Repository** CRAN

**Date/Publication** 2023-11-04 09:40:02 UTC

## R topics documented:

---

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 two 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.

### 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: Michel Lang <michellang@gmail.com> ([ORCID](#))

### 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 DataBackendDataTable first using as_data_backend(), then proceeds with the conversion from DataBackendDataTable to DataBackendDuckDB.
- mlr3::DataBackend: Creates a new DuckDB data base in the specified path. The filename is determined by the hash of the 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 paralleliza-
  tion 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 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 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 DataBackendDataTable first using as_data_backend(), then
  proceeds with the conversion from DataBackendDataTable to DataBackendDplyr.
- mlr3::DataBackend: Creates a new SQLite data base in the specified path. The filename is
  determined by the hash of the 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. be-
cause 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 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 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()](#)

**Method** 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 [DataBack-endDplyr](#) (this includes all connectors from **dbplyr**). Local "tbl" objects such as [tibbles](#)
> will converted to a [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.

**Method** 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, data_format = "data.table")

*Arguments:*
rows integer()
    Row indices.
cols character()
    Column names.
data_format (character(1))
    Desired data format, e.g. "data.table" or "Matrix".

**Method** 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.

**Method** 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.

**Method** 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 disinct 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 char-
    acter 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()

**Method** 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.

**Method** 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, data_format = "data.table")
```
*Arguments:*

rows integer()
    Row indices.

cols character()
    Column names.

data_format (character(1))
    Desired data format, e.g. "data.table" or "Matrix".

**Method** 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.

**Method** 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.

**Method** 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/

# Index