Package 'duckplyr'

February 7, 2025

Type Package

Title A 'DuckDB'-Backed Version of 'dplyr'

Version 1.0.0

Description A drop-in replacement for 'dplyr', powered by 'DuckDB' for performance. Offers convenient utilities for working with in-memory and larger-than-memory data while retaining full 'dplyr' compatibility.

License MIT + file LICENSE

URL https://duckplyr.tidyverse.org, https://github.com/tidyverse/duckplyr

BugReports https://github.com/tidyverse/duckplyr/issues

Depends R (>= 4.0.0), dplyr (>= 1.1.4)

Imports cli, collections, DBI, duckdb (>= 1.1.3-2), glue, jsonlite, lifecycle, magrittr, memoise, pillar (>= 1.10.1), rlang (>= 1.0.6), tibble, tidyselect, utils, vctrs (>= 0.6.3)

Suggests arrow, brio, conflicted, constructive (>= 1.0.0), curl, dbplyr, hms, knitr, lobstr, lubridate, nycflights13, palmerpenguins, prettycode, purrr, readr, rmarkdown, testthat (>= 3.1.5), usethis, withr

Enhances qs, reprex, rstudioapi

Config/Needs/check anthonynorth/roxyglobals

Config/Needs/website tidyverse/tidytemplate, dbplyr, rmarkdown

Config/testthat/edition 3

Config/testthat/parallel false

Config/testthat/start-first rel_api, tpch, as_duckplyr_df, dplyr-mutate, dplyr-filter, dplyr-count-tally

Encoding UTF-8

RoxygenNote 7.3.2.9000

VignetteBuilder knitr

NeedsCompilation no

Author Hannes Mühleisen [aut] (<https://orcid.org/0000-0001-8552-0029>),
Kirill Müller [aut, cre] (<https://orcid.org/0000-0002-1416-3412>),
Posit Software, PBC [cph, fnd]

Maintainer Kirill Müller <kirill@cynkra.com>

Repository CRAN

Date/Publication 2025-02-07 11:30:28 UTC

Contents

`anti_join.duckplyr_df` *Anti join*

Description

This is a method for the `dplyr::anti_join()` generic. `anti_join()` returns all rows from `x` with**out** a match in `y`.

Usage

```
## S3 method for class 'duckplyr_df'
anti_join(x, y, by = NULL, copy = FALSE, ..., na_matches = c("na", "never"))
```

Arguments

`x, y`	A pair of data frames, data frame extensions (e.g. a tibble), or lazy data frames (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details.
`by`	A join specification created with `join_by()`, or a character vector of variables to join by. If `NULL`, the default, `*_join()` will perform a natural join, using all variables in common across `x` and `y`. A message lists the variables so that you can check they're correct; suppress the message by supplying `by` explicitly. To join on different variables between `x` and `y`, use a `join_by()` specification. For example, `join_by(a == b)` will match `x$a` to `y$b`. To join by multiple variables, use a `join_by()` specification with multiple expressions. For example, `join_by(a == b, c == d)` will match `x$a` to `y$b` and `x$c` to `y$d`. If the column names are the same between `x` and `y`, you can shorten this by listing only the variable names, like `join_by(a, c)`. `join_by()` can also be used to perform inequality, rolling, and overlap joins. See the documentation at ?join_by for details on these types of joins. For simple equality joins, you can alternatively specify a character vector of variable names to join by. For example, `by = c("a", "b")` joins `x$a` to `y$a` and `x$b` to `y$b`. If variable names differ between `x` and `y`, use a named character vector like `by = c("x_a" = "y_a", "x_b" = "y_b")`. To perform a cross-join, generating all combinations of `x` and `y`, see `cross_join()`.
`copy`	If `x` and `y` are not from the same data source, and `copy` is `TRUE`, then `y` will be copied into the same src as `x`. This allows you to join tables across srcs, but it is a potentially expensive operation so you must opt into it.
`...`	Other parameters passed onto methods.
`na_matches`	Should two `NA` or two `NaN` values match?

- "na", the default, treats two NA or two NaN values as equal, like %in%, match(), and merge().
- "never" treats two NA or two NaN values as different, and will never match them together or to any other values. This is similar to joins for database sources and to base::merge(incomparables = NA).

See Also

dplyr::anti_join()

Examples

```
library(duckplyr)
band_members %>% anti_join(band_instruments)
```

---

arrange.duckplyr_df *Order rows using column values*

Description

This is a method for the dplyr::arrange() generic. See "Fallbacks" section for differences in implementation. arrange() orders the rows of a data frame by the values of selected columns.

Unlike other dplyr verbs, arrange() largely ignores grouping; you need to explicitly mention grouping variables (or use .by_group = TRUE) in order to group by them, and functions of variables are evaluated once per data frame, not once per group.

Usage

```
## S3 method for class 'duckplyr_df'
arrange(.data, ..., .by_group = FALSE, .locale = NULL)
```

Arguments

.data	A data frame, data frame extension (e.g. a tibble), or a lazy data frame (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details.
...	<data-masking> Variables, or functions of variables. Use desc() to sort a variable in descending order.
.by_group	If TRUE, will sort first by grouping variable. Applies to grouped data frames only.
.locale	The locale to sort character vectors in. • If NULL, the default, uses the "C" locale unless the dplyr.legacy_locale global option escape hatch is active. See the dplyr-locale help page for more details. • If a single string from stringi::stri_locale_list() is supplied, then this will be used as the locale to sort with. For example, "en" will sort with the American English locale. This requires the stringi package.

- If "C" is supplied, then character vectors will always be sorted in the C locale. This does not require stringi and is often much faster than supplying a locale identifier.

The C locale is not the same as English locales, such as "en", particularly when it comes to data containing a mix of upper and lower case letters. This is explained in more detail on the locale help page under the Default locale section.

Fallbacks

There is no DuckDB translation in arrange.duckplyr_df()

- with .by_group = TRUE,
- providing a value for the .locale argument,
- providing a value for the dplyr.legacy_locale option.

These features fall back to dplyr::arrange(), see vignette("fallback") for details.

See Also

dplyr::arrange()

Examples

```
library(duckplyr)
arrange(mtcars, cyl, disp)
arrange(mtcars, desc(disp))
```

---

collect.duckplyr_df *Force conversion to a data frame*

Description

This is a method for the dplyr::collect() generic. collect() converts the input to a tibble, materializing any lazy operations.

Usage

```
## S3 method for class 'duckplyr_df'
collect(x, ...)
```

Arguments

x	A data frame, data frame extension (e.g. a tibble), or a lazy data frame (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details.
...	Arguments passed on to methods

See Also

dplyr::collect()

Examples

```
library(duckplyr)
df <- duckdb_tibble(x = c(1, 2), .lazy = TRUE)
df
try(print(df$x))
df <- collect(df)
df
```

---

compute.duckplyr_df *Compute results*

Description

This is a method for the dplyr::compute() generic. For a duckplyr frame, compute() executes a query but stores it in a (temporary) table, or in a Parquet or CSV file. The result is a duckplyr frame that can be used with subsequent dplyr verbs.

Usage

```
## S3 method for class 'duckplyr_df'
compute(
  x,
  ...,
  prudence = NULL,
  name = NULL,
  schema_name = NULL,
  temporary = TRUE
)
```

Arguments

x	A duckplyr frame.
...	Arguments passed on to methods
prudence	Memory protection, controls if DuckDB may convert intermediate results in DuckDB-managed memory to data frames in R memory. • "lavish": regardless of size, • "stingy": never, • "thrifty": up to a maximum size of 1 million cells. The default is to inherit from the input. This argument is provided here only for convenience. The same effect can be achieved by forwarding the output to as_duckdb_tibble() with the desired prudence. See vignette("prudence") for more information.

name	The name of the table to store the result in.
schema_name	The schema to store the result in, defaults to the current schema.
temporary	Set to FALSE to store the result in a permanent table.

Value

A duckplyr frame.

See Also

dplyr::collect()

Examples

```
library(duckplyr)
df <- duckdb_tibble(x = c(1, 2))
df <- mutate(df, y = 2)
explain(df)
df <- compute(df)
explain(df)
```

---

compute_file *Compute results to a file*

Description

These functions apply to duckplyr frames. They executes a query and stores the results in a flat file. The result is a duckplyr frame that can be used with subsequent dplyr verbs.

`compute_parquet()` creates a Parquet file.

`compute_csv()` creates a CSV file.

Usage

```
compute_parquet(x, path, ..., prudence = NULL, options = NULL)

compute_csv(x, path, ..., prudence = NULL, options = NULL)
```

Arguments

x	A duckplyr frame.
path	The path to store the result in.
...	These dots are for future extensions and must be empty.
prudence	Memory protection, controls if DuckDB may convert intermediate results in DuckDB-managed memory to data frames in R memory. • "lavish": regardless of size, • "stingy": never,

- "thrifty": up to a maximum size of 1 million cells.

The default is to inherit from the input. This argument is provided here only for convenience. The same effect can be achieved by forwarding the output to as_duckdb_tibble() with the desired prudence. See vignette("prudence") for more information.

options — A list of additional options to pass to create the storage format, see https://duckdb.org/docs/data/parquet/overview#writing-to-parquet-files or https://duckdb.org/docs/data/csv/overview#writing-using-the-copy-statement for details.

Value

A duckplyr frame.

See Also

compute.duckplyr_df(), dplyr::collect()

Examples

```
library(duckplyr)
df <- data.frame(x = c(1, 2))
df <- mutate(df, y = 2)
path <- tempfile(fileext = ".parquet")
df <- compute_parquet(df, path)
explain(df)
```

---

config *Configuration options*

Description

The behavior of duckplyr can be fine-tuned with several environment variables, and one option.

Environment variables

DUCKPLYR_TEMP_DIR: Set to a path where temporary files can be created. By default, tempdir() is used.

DUCKPLYR_OUTPUT_ORDER: If TRUE, row output order is preserved. The default may change the row order where dplyr would keep it stable. Preserving the order leads to more complicated execution plans with less potential for optimization, and thus may be slower.

DUCKPLYR_FORCE: If TRUE, fail if duckdb cannot handle a request.

DUCKPLYR_CHECK_ROUNDTRIP: If TRUE, check if all columns are roundtripped perfectly when creating a relational object from a data frame, This is slow, and mostly useful for debugging. The default is to check roundtrip of attributes.

DUCKPLYR_EXPERIMENTAL: If TRUE, pass experimental = TRUE to certain duckdb functions. Currently unused.

DUCKPLYR_METHODS_OVERWRITE: If TRUE, call methods_overwrite() when the package is loaded.

See fallback for more options related to printing, logging, and uploading of fallback events.

Examples

```
# Sys.setenv(DUCKPLYR_OUTPUT_ORDER = TRUE)
data.frame(a = 3:1) %>%
  as_duckdb_tibble() %>%
  inner_join(data.frame(a = 1:4), by = "a")

withr::with_envvar(c(DUCKPLYR_OUTPUT_ORDER = "TRUE"), {
  data.frame(a = 3:1) %>%
    as_duckdb_tibble() %>%
    inner_join(data.frame(a = 1:4), by = "a")
})

# Sys.setenv(DUCKPLYR_FORCE = TRUE)
add_one <- function(x) {
  x + 1
}

data.frame(a = 3:1) %>%
  as_duckdb_tibble() %>%
  mutate(b = add_one(a))

try(withr::with_envvar(c(DUCKPLYR_FORCE = "TRUE"), {
  data.frame(a = 3:1) %>%
    as_duckdb_tibble() %>%
    mutate(b = add_one(a))
}))

# Sys.setenv(DUCKPLYR_FALLBACK_INFO = TRUE)
withr::with_envvar(c(DUCKPLYR_FALLBACK_INFO = "TRUE"), {
  data.frame(a = 3:1) %>%
    as_duckdb_tibble() %>%
    mutate(b = add_one(a))
})
```

---

count.duckplyr_df *Count the observations in each group*

Description

This is a method for the dplyr::count() generic. See "Fallbacks" section for differences in implementation. count() lets you quickly count the unique values of one or more variables: df %>% count(a, b) is roughly equivalent to df %>% group_by(a, b) %>% summarise(n = n()). count() is paired with tally(), a lower-level helper that is equivalent to df %>% summarise(n = n()). Supply wt to perform weighted counts, switching the summary from n = n() to n = sum(wt).

Usage

```
## S3 method for class 'duckplyr_df'
count(
  x,
  ...,
  wt = NULL,
  sort = FALSE,
  name = NULL,
  .drop = group_by_drop_default(x)
)
```

Arguments

x	A data frame, data frame extension (e.g. a tibble), or a lazy data frame (e.g. from dbplyr or dtplyr).
...	<data-masking> Variables to group by.
wt	<data-masking> Frequency weights. Can be NULL or a variable: • If NULL (the default), counts the number of rows in each group. • If a variable, computes sum(wt) for each group.
sort	If TRUE, will show the largest groups at the top.
name	The name of the new column in the output. If omitted, it will default to n. If there's already a column called n, it will use nn. If there's a column called n and nn, it'll use nnn, and so on, adding ns until it gets a new name.
.drop	Handling of factor levels that don't appear in the data, passed on to group_by(). For count(): if FALSE will include counts for empty groups (i.e. for levels of factors that don't exist in the data). **[Deprecated]** For add_count(): deprecated since it can't actually affect the output.

Fallbacks

There is no DuckDB translation in `count.duckplyr_df()`

- with complex expressions in `...`,
- with `.drop = FALSE`,
- with `sort = TRUE`.

These features fall back to `dplyr::count()`, see `vignette("fallback")` for details.

See Also

`dplyr::count()`

Examples

```
library(duckplyr)
count(mtcars, am)
```

db_exec *Execute a statement for the default connection*

Description

The **duckplyr** package relies on a DBI connection to an in-memory database. The `db_exec()` function allows running SQL statements with side effects on this connection. It can be used to execute statements that start with `PRAGMA`, `SET`, or `ATTACH` to, e.g., set up credentials, change configuration options, or attach other databases. See https://duckdb.org/docs/configuration/overview.html for more information on the configuration options, and https://duckdb.org/docs/sql/statements/attach.html for attaching databases.

Usage

```
db_exec(sql, ..., con = NULL)
```

Arguments

`sql`	The statement to run.
`...`	These dots are for future extensions and must be empty.
`con`	The connection, defaults to the default connection.

Value

The return value of the `DBI::dbExecute()` call, invisibly.

See Also

`read_sql_duckdb()`

Examples

```
db_exec("SET threads TO 2")
```

distinct.duckplyr_df *Keep distinct/unique rows*

Description

This is a method for the `dplyr::distinct()` generic. Keep only unique/distinct rows from a data frame. This is similar to `unique.data.frame()` but considerably faster.

Usage

```
## S3 method for class 'duckplyr_df'
distinct(.data, ..., .keep_all = FALSE)
```

Arguments

.data	A data frame, data frame extension (e.g. a tibble), or a lazy data frame (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details.
...	<data-masking> Optional variables to use when determining uniqueness. If there are multiple rows for a given combination of inputs, only the first row will be preserved. If omitted, will use all variables in the data frame.
.keep_all	If TRUE, keep all variables in .data. If a combination of ... is not distinct, this keeps the first row of values.

See Also

dplyr::distinct()

Examples

```
df <- duckdb_tibble(
  x = sample(10, 100, rep = TRUE),
  y = sample(10, 100, rep = TRUE)
)
nrow(df)
nrow(distinct(df))
```

---

duckdb_tibble *duckplyr data frames*

Description

Data frames backed by duckplyr have a special class, "duckplyr_df", in addition to the default classes. This ensures that dplyr methods are dispatched correctly. For such objects, dplyr verbs such as dplyr::mutate(), dplyr::select() or dplyr::filter() will use DuckDB.

duckdb_tibble() works like tibble::tibble().

as_duckdb_tibble() converts a data frame or a dplyr lazy table to a duckplyr data frame. This is a generic function that can be overridden for custom classes.

is_duckdb_tibble() returns TRUE if x is a duckplyr data frame.

Usage

```
duckdb_tibble(..., .prudence = c("lavish", "thrifty", "stingy"))

as_duckdb_tibble(x, ..., prudence = c("lavish", "thrifty", "stingy"))

is_duckdb_tibble(x)
```

Arguments

`...`	For `duckdb_tibble()`, passed on to `tibble::tibble()`. For `as_duckdb_tibble()`, passed on to methods.
`x`	The object to convert or to test.
`prudence, .prudence`	Memory protection, controls if DuckDB may convert intermediate results in DuckDB-managed memory to data frames in R memory. • `"lavish"`: regardless of size, • `"stingy"`: never, • `"thrifty"`: up to a maximum size of 1 million cells. The default is `"lavish"` for `duckdb_tibble()` and `as_duckdb_tibble()`, and may be different for other functions. See `vignette("prudence")` for more information.

Value

For `duckdb_tibble()` and `as_duckdb_tibble()`, an object with the following classes:

- `"prudent_duckplyr_df"` if `prudence` is not `"lavish"`
- `"duckplyr_df"`
- Classes of a tibble::tibble

For `is_duckdb_tibble()`, a scalar logical.

Fine-tuning prudence

[Experimental]

The `prudence` argument can also be a named numeric vector with at least one of `cells` or `rows` to limit the cells (values) and rows in the resulting data frame after automatic materialization. If both limits are specified, both are enforced. The equivalent of `"thrifty"` is `c(cells = 1e6)`.

Examples

```
x <- duckdb_tibble(a = 1)
x

library(dplyr)
x %>%
  mutate(b = 2)

x$a

y <- duckdb_tibble(a = 1, .prudence = "stingy")
y
try(length(y$a))
length(collect(y)$a)
```

explain.duckplyr_df *Explain details of a tbl*

Description

This is a method for the `dplyr::explain()` generic. This is a generic function which gives more details about an object than `print()`, and is more focused on human readable output than `str()`.

Usage

```
## S3 method for class 'duckplyr_df'
explain(x, ...)
```

Arguments

`x`	An object to explain
`...`	Other parameters possibly used by generic

Value

The input, invisibly.

See Also

`dplyr::explain()`

Examples

```
library(duckplyr)
df <- duckdb_tibble(x = c(1, 2))
df <- mutate(df, y = 2)
explain(df)
```

fallback *Fallback to dplyr*

Description

The **duckplyr** package aims at providing a fully compatible drop-in replacement for **dplyr**. To achieve this, only a carefully selected subset of **dplyr**'s operations, R functions, and R data types are implemented. Whenever a request cannot be handled by DuckDB, **duckplyr** falls back to **dplyr**. See `vignette("fallback")`‘ for details.

To assist future development, the fallback situations can be logged to the console or to a local file and uploaded for analysis. By default, **duckplyr** will not log or upload anything. The functions and environment variables on this page control the process.

`fallback_sitrep()` prints the current settings for fallback printing, logging, and uploading, the number of reports ready for upload, and the location of the logs.

`fallback_config()` configures the current settings for fallback printing, logging, and uploading. Only settings that do not affect computation results can be configured, this is by design. The configuration is stored in a file under `tools::R_user_dir("duckplyr", "config")`. When the **duckplyr** package is loaded, the configuration is read from this file, and the corresponding environment variables are set.

`fallback_review()` prints the available reports for review to the console.

`fallback_upload()` uploads the available reports to a central server for analysis. The server is hosted on AWS and the reports are stored in a private S3 bucket. Only authorized personnel have access to the reports.

`fallback_purge()` deletes some or all available reports.

Usage

```
fallback_sitrep()

fallback_config(
  ...,
  reset_all = FALSE,
  info = NULL,
  logging = NULL,
  autoupload = NULL,
  log_dir = NULL,
  verbose = NULL
)

fallback_review(oldest = NULL, newest = NULL, detail = TRUE)

fallback_upload(oldest = NULL, newest = NULL, strict = TRUE)

fallback_purge(oldest = NULL, newest = NULL)
```

Arguments

`...`	These dots are for future extensions and must be empty.
`reset_all`	Set to `TRUE` to reset all settings to their defaults. The R session must be restarted for the changes to take effect.
`info`	Set to `TRUE` to enable fallback printing.
`logging`	Set to `FALSE` to disable fallback logging, set to `TRUE` to explicitly enable it.
`autoupload`	Set to `TRUE` to enable automatic fallback uploading, set to `FALSE` to disable it.
`log_dir`	Set the location of the logs in the file system. The directory will be created if it does not exist.
`verbose`	Set to `TRUE` to enable verbose logging.
`oldest, newest`	The number of oldest or newest reports to review. If not specified, all reports are dispayed.

detail	Print the full content of the reports. Set to FALSE to only print the file names.
strict	If TRUE, the function aborts if any of the reports fail to upload. With FALSE, only a message is printed.

Details

Logging is on by default, but can be turned off. Uploading is opt-in.

The following environment variables control the logging and uploading:

- DUCKPLYR_FALLBACK_INFO controls human-friendly alerts for fallback events. If TRUE, a message is printed when a fallback to dplyr occurs because DuckDB cannot handle a request. These messages are never logged.
- DUCKPLYR_FALLBACK_COLLECT controls logging, set it to 1 or greater to enable logging. If the value is 0, logging is disabled. Future versions of **duckplyr** may start logging additional data and thus require a higher value to enable logging. Set to 99 to enable logging for all future versions. Use `usethis::edit_r_environ()` to edit the environment file.
- DUCKPLYR_FALLBACK_AUTOUPLOAD controls uploading, set it to 1 or greater to enable uploading. If the value is 0, uploading is disabled. Currently, uploading is active if the value is 1 or greater. Future versions of **duckplyr** may start logging additional data and thus require a higher value to enable uploading. Set to 99 to enable uploading for all future versions. Use `usethis::edit_r_environ()` to edit the environment file.
- DUCKPLYR_FALLBACK_LOG_DIR controls the location of the logs. It must point to a directory (existing or not) where the logs will be written. By default, logs are written to a directory in the user's cache directory as returned by `tools::R_user_dir("duckplyr", "cache")`.
- DUCKPLYR_FALLBACK_VERBOSE controls printing of log data, set it to TRUE or FALSE to enable or disable printing. If the value is TRUE, a message is printed to the console for each fallback situation. This setting is only relevant if logging is enabled, and mostly useful for **duckplyr**'s internal tests.

All code related to fallback logging and uploading is in the `fallback.R` and `telemetry.R` files.

Examples

```
fallback_sitrep()
```

---

filter.duckplyr_df *Keep rows that match a condition*

Description

This is a method for the `dplyr::select()` generic. See "Fallbacks" section for differences in implementation. The `filter()` function is used to subset a data frame, retaining all rows that satisfy your conditions. To be retained, the row must produce a value of TRUE for all conditions. Note that when a condition evaluates to NA the row will be dropped, unlike base subsetting with `[`.

Usage

```
## S3 method for class 'duckplyr_df'
filter(.data, ..., .by = NULL, .preserve = FALSE)
```

Arguments

.data	A data frame, data frame extension (e.g. a tibble), or a lazy data frame (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details.
...	<data-masking> Expressions that return a logical value, and are defined in terms of the variables in .data. If multiple expressions are included, they are combined with the & operator. Only rows for which all conditions evaluate to TRUE are kept.
.by	**[Experimental]** <tidy-select> Optionally, a selection of columns to group by for just this operation, functioning as an alternative to group_by(). For details and examples, see ?dplyr_by.
.preserve	Relevant when the .data input is grouped. If .preserve = FALSE (the default), the grouping structure is recalculated based on the resulting data, otherwise the grouping is kept as is.

Fallbacks

There is no DuckDB translation in `filter.duckplyr_df()`

- with no filter conditions,
- nor for a grouped operation (if .by is set).

These features fall back to `dplyr::filter()`, see `vignette("fallback")` for details.

See Also

`dplyr::filter()`

Examples

```
df <- duckdb_tibble(x = 1:3, y = 3:1)
filter(df, x >= 2)
```

---

flights_df *Flight data*

Description

Provides a variant of `nycflights13::flights` that is compatible with duckplyr, as a tibble: the timezone has been set to UTC to work around a current limitation of duckplyr, see `vignette("limits")`. Call `as_duckdb_tibble()` to enable duckplyr operations.

Usage

```
flights_df()
```

Examples

```
flights_df()
```

`full_join.duckplyr_df` *Full join*

Description

This is a method for the `dplyr::full_join()` generic. See "Fallbacks" section for differences in implementation. A `full_join()` keeps all observations in `x` and `y`.

Usage

```
## S3 method for class 'duckplyr_df'
full_join(
  x,
  y,
  by = NULL,
  copy = FALSE,
  suffix = c(".x", ".y"),
  ...,
  keep = NULL,
  na_matches = c("na", "never"),
  multiple = "all",
  relationship = NULL
)
```

Arguments

`x, y`	A pair of data frames, data frame extensions (e.g. a tibble), or lazy data frames (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details.
`by`	A join specification created with `join_by()`, or a character vector of variables to join by. If `NULL`, the default, `*_join()` will perform a natural join, using all variables in common across `x` and `y`. A message lists the variables so that you can check they're correct; suppress the message by supplying `by` explicitly. To join on different variables between `x` and `y`, use a `join_by()` specification. For example, `join_by(a == b)` will match `x$a` to `y$b`. To join by multiple variables, use a `join_by()` specification with multiple expressions. For example, `join_by(a == b, c == d)` will match `x$a` to `y$b` and

x$c to y$d. If the column names are the same between x and y, you can shorten this by listing only the variable names, like join_by(a, c).

join_by() can also be used to perform inequality, rolling, and overlap joins. See the documentation at ?join_by for details on these types of joins.

For simple equality joins, you can alternatively specify a character vector of variable names to join by. For example, by = c("a", "b") joins x$a to y$a and x$b to y$b. If variable names differ between x and y, use a named character vector like by = c("x_a" = "y_a", "x_b" = "y_b").

To perform a cross-join, generating all combinations of x and y, see cross_join().

copy — If x and y are not from the same data source, and copy is TRUE, then y will be copied into the same src as x. This allows you to join tables across srcs, but it is a potentially expensive operation so you must opt into it.

suffix — If there are non-joined duplicate variables in x and y, these suffixes will be added to the output to disambiguate them. Should be a character vector of length 2.

... — Other parameters passed onto methods.

keep — Should the join keys from both x and y be preserved in the output?

- If NULL, the default, joins on equality retain only the keys from x, while joins on inequality retain the keys from both inputs.
- If TRUE, all keys from both inputs are retained.
- If FALSE, only keys from x are retained. For right and full joins, the data in key columns corresponding to rows that only exist in y are merged into the key columns from x. Can't be used when joining on inequality conditions.

na_matches — Should two NA or two NaN values match?

- "na", the default, treats two NA or two NaN values as equal, like %in%, match(), and merge().
- "never" treats two NA or two NaN values as different, and will never match them together or to any other values. This is similar to joins for database sources and to base::merge(incomparables = NA).

multiple — Handling of rows in x with multiple matches in y. For each row of x:

- "all", the default, returns every match detected in y. This is the same behavior as SQL.
- "any" returns one match detected in y, with no guarantees on which match will be returned. It is often faster than "first" and "last" if you just need to detect if there is at least one match.
- "first" returns the first match detected in y.
- "last" returns the last match detected in y.

relationship — Handling of the expected relationship between the keys of x and y. If the expectations chosen from the list below are invalidated, an error is thrown.

- NULL, the default, doesn't expect there to be any relationship between x and y. However, for equality joins it will check for a many-to-many relationship (which is typically unexpected) and will warn if one occurs, encouraging you to either take a closer look at your inputs or make this relationship explicit by specifying "many-to-many".
  See the *Many-to-many relationships* section for more details.

- "one-to-one" expects:
  - Each row in x matches at most 1 row in y.
  - Each row in y matches at most 1 row in x.
- "one-to-many" expects:
  - Each row in y matches at most 1 row in x.
- "many-to-one" expects:
  - Each row in x matches at most 1 row in y.
- "many-to-many" doesn't perform any relationship checks, but is provided to allow you to be explicit about this relationship if you know it exists.

relationship doesn't handle cases where there are zero matches. For that, see unmatched.

Fallbacks

There is no DuckDB translation in full_join.duckplyr_df()

- for an implicit cross join,
- for a value of the multiple argument that isn't the default "all".

These features fall back to dplyr::full_join(), see vignette("fallback") for details.

See Also

dplyr::full_join()

Examples

```
library(duckplyr)
full_join(band_members, band_instruments)
```

---

head.duckplyr_df *Return the First Parts of an Object*

Description

This is a method for the head() generic. See "Fallbacks" section for differences in implementation. Return the first rows of a data.frame

Usage

```
## S3 method for class 'duckplyr_df'
head(x, n = 6L, ...)
```

Arguments

x	A data.frame
n	A positive integer, how many rows to return.
...	Not used yet.

Fallbacks

There is no DuckDB translation in `head.duckplyr_df()`

- with a negative `n`.

These features fall back to `head()`, see `vignette("fallback")` for details.

See Also

`head()`

Examples

```
head(mtcars, 2)
```

`inner_join.duckplyr_df` *Inner join*

Description

This is a method for the `dplyr::inner_join()` generic. See "Fallbacks" section for differences in implementation. An `inner_join()` only keeps observations from `x` that have a matching key in `y`.

Usage

```
## S3 method for class 'duckplyr_df'
inner_join(
  x,
  y,
  by = NULL,
  copy = FALSE,
  suffix = c(".x", ".y"),
  ...,
  keep = NULL,
  na_matches = c("na", "never"),
  multiple = "all",
  unmatched = "drop",
  relationship = NULL
)
```

Arguments

`x`, `y`	A pair of data frames, data frame extensions (e.g. a tibble), or lazy data frames (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details.

by
: A join specification created with join_by(), or a character vector of variables to join by.

  If NULL, the default, *_join() will perform a natural join, using all variables in common across x and y. A message lists the variables so that you can check they're correct; suppress the message by supplying by explicitly.

  To join on different variables between x and y, use a join_by() specification. For example, join_by(a == b) will match x$a to y$b.

  To join by multiple variables, use a join_by() specification with multiple expressions. For example, join_by(a == b, c == d) will match x$a to y$b and x$c to y$d. If the column names are the same between x and y, you can shorten this by listing only the variable names, like join_by(a, c).

  join_by() can also be used to perform inequality, rolling, and overlap joins. See the documentation at ?join_by for details on these types of joins.

  For simple equality joins, you can alternatively specify a character vector of variable names to join by. For example, by = c("a", "b") joins x$a to y$a and x$b to y$b. If variable names differ between x and y, use a named character vector like by = c("x_a" = "y_a", "x_b" = "y_b").

  To perform a cross-join, generating all combinations of x and y, see cross_join().

copy
: If x and y are not from the same data source, and copy is TRUE, then y will be copied into the same src as x. This allows you to join tables across srcs, but it is a potentially expensive operation so you must opt into it.

suffix
: If there are non-joined duplicate variables in x and y, these suffixes will be added to the output to disambiguate them. Should be a character vector of length 2.

...
: Other parameters passed onto methods.

keep
: Should the join keys from both x and y be preserved in the output?

  - If NULL, the default, joins on equality retain only the keys from x, while joins on inequality retain the keys from both inputs.
  - If TRUE, all keys from both inputs are retained.
  - If FALSE, only keys from x are retained. For right and full joins, the data in key columns corresponding to rows that only exist in y are merged into the key columns from x. Can't be used when joining on inequality conditions.

na_matches
: Should two NA or two NaN values match?

  - "na", the default, treats two NA or two NaN values as equal, like %in%, match(), and merge().
  - "never" treats two NA or two NaN values as different, and will never match them together or to any other values. This is similar to joins for database sources and to base::merge(incomparables = NA).

multiple
: Handling of rows in x with multiple matches in y. For each row of x:

  - "all", the default, returns every match detected in y. This is the same behavior as SQL.
  - "any" returns one match detected in y, with no guarantees on which match will be returned. It is often faster than "first" and "last" if you just need to detect if there is at least one match.
  - "first" returns the first match detected in y.

- "last" returns the last match detected in y.

unmatched How should unmatched keys that would result in dropped rows be handled?

- "drop" drops unmatched keys from the result.
- "error" throws an error if unmatched keys are detected.

unmatched is intended to protect you from accidentally dropping rows during a join. It only checks for unmatched keys in the input that could potentially drop rows.

- For left joins, it checks y.
- For right joins, it checks x.
- For inner joins, it checks both x and y. In this case, unmatched is also allowed to be a character vector of length 2 to specify the behavior for x and y independently.

relationship Handling of the expected relationship between the keys of x and y. If the expectations chosen from the list below are invalidated, an error is thrown.

- NULL, the default, doesn't expect there to be any relationship between x and y. However, for equality joins it will check for a many-to-many relationship (which is typically unexpected) and will warn if one occurs, encouraging you to either take a closer look at your inputs or make this relationship explicit by specifying "many-to-many".
  See the *Many-to-many relationships* section for more details.
- "one-to-one" expects:
  - Each row in x matches at most 1 row in y.
  - Each row in y matches at most 1 row in x.
- "one-to-many" expects:
  - Each row in y matches at most 1 row in x.
- "many-to-one" expects:
  - Each row in x matches at most 1 row in y.
- "many-to-many" doesn't perform any relationship checks, but is provided to allow you to be explicit about this relationship if you know it exists.

relationship doesn't handle cases where there are zero matches. For that, see unmatched.

Fallbacks

There is no DuckDB translation in inner_join.duckplyr_df()

- for an implicit crossjoin,
- for a value of the multiple argument that isn't the default "all".
- for a value of the unmatched argument that isn't the default "drop".

These features fall back to dplyr::inner_join(), see vignette("fallback") for details.

See Also

dplyr::inner_join()

Examples

```
library(duckplyr)
inner_join(band_members, band_instruments)
```

`intersect.duckplyr_df` *Intersect*

Description

This is a method for the `dplyr::intersect()` generic. See "Fallbacks" section for differences in implementation. `intersect(x, y)` finds all rows in both `x` and `y`.

Usage

```
## S3 method for class 'duckplyr_df'
intersect(x, y, ...)
```

Arguments

`x, y`	Pair of compatible data frames. A pair of data frames is compatible if they have the same column names (possibly in different orders) and compatible types.
`...`	These dots are for future extensions and must be empty.

Fallbacks

There is no DuckDB translation in `intersect.duckplyr_df()`

- if column names are duplicated in one of the tables,
- if column names are different in both tables.

These features fall back to `dplyr::intersect()`, see `vignette("fallback")` for details.

See Also

`dplyr::intersect()`

Examples

```
df1 <- duckdb_tibble(x = 1:3)
df2 <- duckdb_tibble(x = 3:5)
intersect(df1, df2)
```

last_rel *Retrieve details about the most recent computation*

Description

Before a result is computed, it is specified as a "relation" object. This function retrieves this object for the last computation that led to the materialization of a data frame.

Usage

```
last_rel()
```

Value

A duckdb "relation" object, or `NULL` if no computation has been performed yet.

left_join.duckplyr_df *Left join*

Description

This is a method for the `dplyr::left_join()` generic. See "Fallbacks" section for differences in implementation. A `left_join()` keeps all observations in `x`.

Usage

```
## S3 method for class 'duckplyr_df'
left_join(
  x,
  y,
  by = NULL,
  copy = FALSE,
  suffix = c(".x", ".y"),
  ...,
  keep = NULL,
  na_matches = c("na", "never"),
  multiple = "all",
  unmatched = "drop",
  relationship = NULL
)
```

Arguments

x, y	A pair of data frames, data frame extensions (e.g. a tibble), or lazy data frames (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details.
by	A join specification created with `join_by()`, or a character vector of variables to join by. If `NULL`, the default, `*_join()` will perform a natural join, using all variables in common across `x` and `y`. A message lists the variables so that you can check they're correct; suppress the message by supplying `by` explicitly. To join on different variables between `x` and `y`, use a `join_by()` specification. For example, `join_by(a == b)` will match `x$a` to `y$b`. To join by multiple variables, use a `join_by()` specification with multiple expressions. For example, `join_by(a == b, c == d)` will match `x$a` to `y$b` and `x$c` to `y$d`. If the column names are the same between `x` and `y`, you can shorten this by listing only the variable names, like `join_by(a, c)`. `join_by()` can also be used to perform inequality, rolling, and overlap joins. See the documentation at ?join_by for details on these types of joins. For simple equality joins, you can alternatively specify a character vector of variable names to join by. For example, `by = c("a", "b")` joins `x$a` to `y$a` and `x$b` to `y$b`. If variable names differ between `x` and `y`, use a named character vector like `by = c("x_a" = "y_a", "x_b" = "y_b")`. To perform a cross-join, generating all combinations of `x` and `y`, see `cross_join()`.
copy	If `x` and `y` are not from the same data source, and `copy` is `TRUE`, then `y` will be copied into the same src as `x`. This allows you to join tables across srcs, but it is a potentially expensive operation so you must opt into it.
suffix	If there are non-joined duplicate variables in `x` and `y`, these suffixes will be added to the output to disambiguate them. Should be a character vector of length 2.
...	Other parameters passed onto methods.
keep	Should the join keys from both `x` and `y` be preserved in the output? • If `NULL`, the default, joins on equality retain only the keys from `x`, while joins on inequality retain the keys from both inputs. • If `TRUE`, all keys from both inputs are retained. • If `FALSE`, only keys from `x` are retained. For right and full joins, the data in key columns corresponding to rows that only exist in `y` are merged into the key columns from `x`. Can't be used when joining on inequality conditions.
na_matches	Should two `NA` or two `NaN` values match? • `"na"`, the default, treats two `NA` or two `NaN` values as equal, like `%in%`, `match()`, and `merge()`. • `"never"` treats two `NA` or two `NaN` values as different, and will never match them together or to any other values. This is similar to joins for database sources and to `base::merge(incomparables = NA)`.
multiple	Handling of rows in `x` with multiple matches in `y`. For each row of `x`: • `"all"`, the default, returns every match detected in `y`. This is the same behavior as SQL.

- `"any"` returns one match detected in `y`, with no guarantees on which match will be returned. It is often faster than `"first"` and `"last"` if you just need to detect if there is at least one match.
- `"first"` returns the first match detected in `y`.
- `"last"` returns the last match detected in `y`.

`unmatched` How should unmatched keys that would result in dropped rows be handled?

- `"drop"` drops unmatched keys from the result.
- `"error"` throws an error if unmatched keys are detected.

`unmatched` is intended to protect you from accidentally dropping rows during a join. It only checks for unmatched keys in the input that could potentially drop rows.

- For left joins, it checks `y`.
- For right joins, it checks `x`.
- For inner joins, it checks both `x` and `y`. In this case, `unmatched` is also allowed to be a character vector of length 2 to specify the behavior for `x` and `y` independently.

`relationship` Handling of the expected relationship between the keys of `x` and `y`. If the expectations chosen from the list below are invalidated, an error is thrown.

- `NULL`, the default, doesn't expect there to be any relationship between `x` and `y`. However, for equality joins it will check for a many-to-many relationship (which is typically unexpected) and will warn if one occurs, encouraging you to either take a closer look at your inputs or make this relationship explicit by specifying `"many-to-many"`.
  See the *Many-to-many relationships* section for more details.
- `"one-to-one"` expects:
  - Each row in `x` matches at most 1 row in `y`.
  - Each row in `y` matches at most 1 row in `x`.
- `"one-to-many"` expects:
  - Each row in `y` matches at most 1 row in `x`.
- `"many-to-one"` expects:
  - Each row in `x` matches at most 1 row in `y`.
- `"many-to-many"` doesn't perform any relationship checks, but is provided to allow you to be explicit about this relationship if you know it exists.

`relationship` doesn't handle cases where there are zero matches. For that, see `unmatched`.

Fallbacks

There is no DuckDB translation in `left_join.duckplyr_df()`

- for an implicit cross join,
- for a value of the `multiple` argument that isn't the default `"all"`.
- for a value of the `unmatched` argument that isn't the default `"drop"`.

These features fall back to `dplyr::left_join()`, see `vignette("fallback")` for details.

See Also

dplyr::left_join()

Examples

```
library(duckplyr)
left_join(band_members, band_instruments)
```

---

methods_overwrite *Forward all dplyr methods to duckplyr*

Description

After calling `methods_overwrite()`, all dplyr methods are redirected to duckplyr for the duraton of the session, or until a call to `methods_restore()`. The `methods_overwrite()` function is called automatically when the package is loaded if the environment variable `DUCKPLYR_METHODS_OVERWRITE` is set to `TRUE`.

Usage

```
methods_overwrite()

methods_restore()
```

Value

Called for their side effects.

Examples

```
tibble(a = 1:3) %>%
  mutate(b = a + 1)

methods_overwrite()

tibble(a = 1:3) %>%
  mutate(b = a + 1)

methods_restore()

tibble(a = 1:3) %>%
  mutate(b = a + 1)
```

`mutate.duckplyr_df` *Create, modify, and delete columns*

Description

This is a method for the `dplyr::mutate()` generic. `mutate()` creates new columns that are functions of existing variables. It can also modify (if the name is the same as an existing column) and delete columns (by setting their value to `NULL`).

Usage

```
## S3 method for class 'duckplyr_df'
mutate(
  .data,
  ...,
  .by = NULL,
  .keep = c("all", "used", "unused", "none"),
  .before = NULL,
  .after = NULL
)
```

Arguments

`.data`	A data frame, data frame extension (e.g. a tibble), or a lazy data frame (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details.
`...`	<`data-masking`> Name-value pairs. The name gives the name of the column in the output. The value can be: • A vector of length 1, which will be recycled to the correct length. • A vector the same length as the current group (or the whole data frame if ungrouped). • `NULL`, to remove the column. • A data frame or tibble, to create multiple columns in the output.
`.by`	**[Experimental]** <`tidy-select`> Optionally, a selection of columns to group by for just this operation, functioning as an alternative to `group_by()`. For details and examples, see ?dplyr_by.
`.keep`	Control which columns from `.data` are retained in the output. Grouping columns and columns created by `...` are always kept. • `"all"` retains all columns from `.data`. This is the default. • `"used"` retains only the columns used in `...` to create new columns. This is useful for checking your work, as it displays inputs and outputs side-by-side.

- `"unused"` retains only the columns *not* used in `...` to create new columns. This is useful if you generate new columns, but no longer need the columns used to generate them.
- `"none"` doesn't retain any extra columns from `.data`. Only the grouping variables and columns created by `...` are kept.

`.before, .after` <tidy-select> Optionally, control where new columns should appear (the default is to add to the right hand side). See `relocate()` for more details.

See Also

`dplyr::mutate()`

Examples

```
library(duckplyr)
df <- data.frame(x = c(1, 2))
df <- mutate(df, y = 2)
df
```

`new_relational` *Relational implementer's interface*

Description

The constructor and generics described here define a class that helps separating dplyr's user interface from the actual underlying operations. In the longer term, this will help packages that implement the dplyr interface (such as **dbplyr**, **dtplyr**, **arrow** and similar) to focus on the core details of their functionality, rather than on the intricacies of dplyr's user interface.

`new_relational()` constructs an object of class `"relational"`. Users are encouraged to provide the `class` argument. The typical use case will be to create a wrapper function.

`rel_to_df()` extracts a data frame representation from a relational object, to be used by `dplyr::collect()`.

`rel_filter()` keeps rows that match a predicate, to be used by `dplyr::filter()`.

`rel_project()` selects columns or creates new columns, to be used by `dplyr::select()`, `dplyr::rename()`, `dplyr::mutate()`, `dplyr::relocate()`, and others.

`rel_aggregate()` combines several rows into one, to be used by `dplyr::summarize()`.

`rel_order()` reorders rows by columns or expressions, to be used by `dplyr::arrange()`.

`rel_join()` joins or merges two tables, to be used by `dplyr::left_join()`, `dplyr::right_join()`, `dplyr::inner_join()`, `dplyr::full_join()`, `dplyr::cross_join()`, `dplyr::semi_join()`, and `dplyr::anti_join()`.

`rel_limit()` limits the number of rows in a table, to be used by `utils::head()`.

`rel_distinct()` only keeps the distinct rows in a table, to be used by `dplyr::distinct()`.

`rel_set_intersect()` returns rows present in both tables, to be used by `generics::intersect()`.

`rel_set_diff()` returns rows present in any of both tables, to be used by `generics::setdiff()`.

`rel_set_symdiff()` returns rows present in any of both tables, to be used by `dplyr::symdiff()`.

`rel_union_all()` returns rows present in any of both tables, to be used by `dplyr::union_all()`.

`rel_explain()` prints an explanation of the plan executed by the relational object.

`rel_alias()` returns the alias name for a relational object.

`rel_set_alias()` sets the alias name for a relational object.

`rel_names()` returns the column names as character vector, to be used by `colnames()`.

Usage

```
new_relational(..., class = NULL)

rel_to_df(rel, ...)

rel_filter(rel, exprs, ...)

rel_project(rel, exprs, ...)

rel_aggregate(rel, groups, aggregates, ...)

rel_order(rel, orders, ascending, ...)

rel_join(
  left,
  right,
  conds,
  join = c("inner", "left", "right", "outer", "cross", "semi", "anti"),
  join_ref_type = c("regular", "natural", "cross", "positional", "asof"),
  ...
)

rel_limit(rel, n, ...)

rel_distinct(rel, ...)

rel_set_intersect(rel_a, rel_b, ...)

rel_set_diff(rel_a, rel_b, ...)

rel_set_symdiff(rel_a, rel_b, ...)

rel_union_all(rel_a, rel_b, ...)

rel_explain(rel, ...)

rel_alias(rel, ...)

rel_set_alias(rel, alias, ...)
```

```
rel_names(rel, ...)
```

Arguments

`...`	Reserved for future extensions, must be empty.
`class`	Classes added in front of the `"relational"` base class.
`rel`, `rel_a`, `rel_b`, `left`, `right`	A relational object.
`exprs`	A list of `"relational_relexpr"` objects to filter by, created by `new_relexpr()`.
`groups`	A list of expressions to group by.
`aggregates`	A list of expressions with aggregates to compute.
`orders`	A list of expressions to order by.
`ascending`	A logical vector describing the sort order.
`conds`	A list of expressions to use for the join.
`join`	The type of join.
`join_ref_type`	The ref type of join.
`n`	The number of rows.
`alias`	the new alias

Value

- `new_relational()` returns a new relational object.
- `rel_to_df()` returns a data frame.
- `rel_names()` returns a character vector.
- All other generics return a modified relational object.

Examples

```
new_dfrel <- function(x) {
  stopifnot(is.data.frame(x))
  new_relational(list(x), class = "dfrel")
}
mtcars_rel <- new_dfrel(mtcars[1:5, 1:4])

rel_to_df.dfrel <- function(rel, ...) {
  unclass(rel)[[1]]
}
rel_to_df(mtcars_rel)

rel_filter.dfrel <- function(rel, exprs, ...) {
  df <- unclass(rel)[[1]]

  # A real implementation would evaluate the predicates defined
  # by the exprs argument
  new_dfrel(df[seq_len(min(3, nrow(df))), ])
```

```
}

rel_filter(
  mtcars_rel,
  list(
    relexpr_function(
      "gt",
      list(relexpr_reference("cyl"), relexpr_constant("6"))
    )
  )
)

rel_project.dfrel <- function(rel, exprs, ...) {
  df <- unclass(rel)[[1]]

  # A real implementation would evaluate the expressions defined
  # by the exprs argument
  new_dfrel(df[seq_len(min(3, ncol(df)))])
}

rel_project(
  mtcars_rel,
  list(relexpr_reference("cyl"), relexpr_reference("disp"))
)

rel_order.dfrel <- function(rel, exprs, ...) {
  df <- unclass(rel)[[1]]

  # A real implementation would evaluate the expressions defined
  # by the exprs argument
  new_dfrel(df[order(df[[1]]), ])
}

rel_order(
  mtcars_rel,
  list(relexpr_reference("mpg"))
)

rel_join.dfrel <- function(left, right, conds, join, ...) {
  left_df <- unclass(left)[[1]]
  right_df <- unclass(right)[[1]]

  # A real implementation would evaluate the expressions
  # defined by the conds argument,
  # use different join types based on the join argument,
  # and implement the join itself instead of relaying to left_join().
  new_dfrel(dplyr::left_join(left_df, right_df))
}

rel_join(new_dfrel(data.frame(mpg = 21)), mtcars_rel)


rel_limit.dfrel <- function(rel, n, ...) {
```

```
  df <- unclass(rel)[[1]]

  new_dfrel(df[seq_len(n), ])
}

rel_limit(mtcars_rel, 3)

rel_distinct.dfrel <- function(rel, ...) {
  df <- unclass(rel)[[1]]

  new_dfrel(df[!duplicated(df), ])
}

rel_distinct(new_dfrel(mtcars[1:3, 1:4]))

rel_names.dfrel <- function(rel, ...) {
  df <- unclass(rel)[[1]]

  names(df)
}

rel_names(mtcars_rel)
```

`new_relexpr` *Relational expressions*

Description

These functions provide a backend-agnostic way to construct expression trees built of column references, constants, and functions. All subexpressions in an expression tree can have an alias.

`new_relexpr()` constructs an object of class `"relational_relexpr"`. It is used by the higher-level constructors, users should rarely need to call it directly.

`relexpr_reference()` constructs a reference to a column.

`relexpr_constant()` wraps a constant value.

`relexpr_function()` applies a function. The arguments to this function are a list of other expression objects.

`relexpr_comparison()` wraps a comparison expression.

`relexpr_window()` applies a function over a window, similarly to the SQL `OVER` clause.

`relexpr_set_alias()` assigns an alias to an expression.

Usage

```
new_relexpr(x, class = NULL)

relexpr_reference(name, rel = NULL, alias = NULL)
```

```
relexpr_constant(val, alias = NULL)

relexpr_function(name, args, alias = NULL)

relexpr_comparison(cmp_op, exprs)

relexpr_window(
  expr,
  partitions,
  order_bys = list(),
  offset_expr = NULL,
  default_expr = NULL,
  alias = NULL
)

relexpr_set_alias(expr, alias = NULL)
```

Arguments

x	An object.
class	Classes added in front of the "relational_relexpr" base class.
name	The name of the column or function to reference.
rel	The name of the relation to reference.
alias	An alias for the new expression.
val	The value to use in the constant expression.
args	Function arguments, a list of expr objects.
cmp_op	Comparison operator, e.g., "<" or "==".
exprs	Expressions to compare, a list of expr objects.
expr	An expr object.
partitions	Partitions, a list of expr objects.
order_bys	which variables to order results by (list).
offset_expr	offset relational expression.
default_expr	default relational expression.

Value

an object of class "relational_relexpr"

an object of class "relational_relexpr"

an object of class "relational_relexpr"

an object of class "relational_relexpr"

an object of class "relational_relexpr"

an object of class "relational_relexpr"

Examples

```
relexpr_set_alias(
  alias = "my_predicate",
  relexpr_function(
    "<",
    list(
      relexpr_reference("my_number"),
      relexpr_constant(42)
    )
  )
)
```

`pull.duckplyr_df` *Extract a single column*

Description

This is a method for the `dplyr::pull()` generic. See "Fallbacks" section for differences in implementation. `pull()` is similar to `$`. It's mostly useful because it looks a little nicer in pipes, it also works with remote data frames, and it can optionally name the output.

Usage

```
## S3 method for class 'duckplyr_df'
pull(.data, var = -1, name = NULL, ...)
```

Arguments

`.data`	A data frame, data frame extension (e.g. a tibble), or a lazy data frame (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details.
`var`	A variable specified as: • a literal variable name • a positive integer, giving the position counting from the left • a negative integer, giving the position counting from the right. The default returns the last column (on the assumption that's the column you've created most recently). This argument is taken by expression and supports quasiquotation (you can unquote column names and column locations).
`name`	An optional parameter that specifies the column to be used as names for a named vector. Specified in a similar manner as `var`.
`...`	For use by methods.

Fallbacks

There is no DuckDB translation in `pull.duckplyr_df()`

- with a selection that returns no columns.

These features fall back to `dplyr::pull()`, see `vignette("fallback")` for details.

See Also

dplyr::pull()

Examples

```
library(duckplyr)
pull(mtcars, cyl)
pull(mtcars, 1)
```

---

read_file_duckdb *Read Parquet, CSV, and other files using DuckDB*

Description

These functions ingest data from a file. In many cases, these functions return immediately because they only read the metadata. The actual data is only read when it is actually processed.

`read_parquet_duckdb()` reads a CSV file using DuckDB's `read_parquet()` table function.

`read_csv_duckdb()` reads a CSV file using DuckDB's `read_csv_auto()` table function.

`read_json_duckdb()` reads a JSON file using DuckDB's `read_json()` table function.

`read_file_duckdb()` uses arbitrary readers to read data. See https://duckdb.org/docs/data/overview for a documentation of the available functions and their options. To read multiple files with the same schema, pass a wildcard or a character vector to the `path` argument,

Usage

```
read_parquet_duckdb(
  path,
  ...,
  prudence = c("thrifty", "lavish", "stingy"),
  options = list()
)

read_csv_duckdb(
  path,
  ...,
  prudence = c("thrifty", "lavish", "stingy"),
  options = list()
)

read_json_duckdb(
  path,
  ...,
  prudence = c("thrifty", "lavish", "stingy"),
  options = list()
)
```

```
read_file_duckdb(
  path,
  table_function,
  ...,
  prudence = c("thrifty", "lavish", "stingy"),
  options = list()
)
```

Arguments

path	Path to files, glob patterns * and ? are supported.
...	These dots are for future extensions and must be empty.
prudence	Memory protection, controls if DuckDB may convert intermediate results in DuckDB-managed memory to data frames in R memory. • "thrifty": up to a maximum size of 1 million cells, • "lavish": regardless of size, • "stingy": never. The default is "thrifty" for the ingestion functions, and may be different for other functions. See vignette("prudence") for more information.
options	Arguments to the DuckDB function indicated by table_function.
table_function	The name of a table-valued DuckDB function such as "read_parquet", "read_csv", "read_csv_auto" or "read_json".

Value

A duckplyr frame, see as_duckdb_tibble() for details.

Fine-tuning prudence

[Experimental]

The prudence argument can also be a named numeric vector with at least one of cells or rows to limit the cells (values) and rows in the resulting data frame after automatic materialization. If both limits are specified, both are enforced. The equivalent of "thrifty" is c(cells = 1e6).

Examples

```
# Create simple CSV file
path <- tempfile("duckplyr_test_", fileext = ".csv")
write.csv(data.frame(a = 1:3, b = letters[4:6]), path, row.names = FALSE)

# Reading is immediate
df <- read_csv_duckdb(path)

# Names are always available
names(df)

# Materialization upon access is turned off by default
```

```
try(print(df$a))

# Materialize explicitly
collect(df)$a

# Automatic materialization with prudence = "lavish"
df <- read_csv_duckdb(path, prudence = "lavish")
df$a

# Specify column types
read_csv_duckdb(
  path,
  options = list(delim = ",", types = list(c("DOUBLE", "VARCHAR")))
)

# Create and read a simple JSON file
path <- tempfile("duckplyr_test_", fileext = ".json")
writeLines('[{"a": 1, "b": "x"}, {"a": 2, "b": "y"}]', path)

# Reading needs the json extension
db_exec("INSTALL json")
db_exec("LOAD json")
read_json_duckdb(path)
```

`read_sql_duckdb` *Return SQL query as duckdb_tibble*

Description

[Experimental]

Runs a query and returns it as a duckplyr frame.

Usage

```
read_sql_duckdb(
  sql,
  ...,
  prudence = c("thrifty", "lavish", "stingy"),
  con = NULL
)
```

Arguments

`sql`	The SQL to run.
`...`	These dots are for future extensions and must be empty.
`prudence`	Memory protection, controls if DuckDB may convert intermediate results in DuckDB-managed memory to data frames in R memory.

- `"thrifty"`: up to a maximum size of 1 million cells,

- `"lavish"`: regardless of size,
- `"stingy"`: never.

The default is `"thrifty"` for the ingestion functions, and may be different for other functions. See `vignette("prudence")` for more information.

`con` The connection, defaults to the default connection.

Details

Using data frames from the calling environment is not supported yet, see https://github.com/duckdb/duckdb-r/issues/645 for details.

See Also

`db_exec()`

Examples

```
read_sql_duckdb("FROM duckdb_settings()")
```

---

relocate.duckplyr_df *Change column order*

Description

This is a method for the `dplyr::relocate()` generic. See "Fallbacks" section for differences in implementation. Use `relocate()` to change column positions, using the same syntax as `select()` to make it easy to move blocks of columns at once.

Usage

```
## S3 method for class 'duckplyr_df'
relocate(.data, ..., .before = NULL, .after = NULL)
```

Arguments

`.data`	A data frame, data frame extension (e.g. a tibble), or a lazy data frame (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details.
`...`	<`tidy-select`> Columns to move.
`.before, .after`	<`tidy-select`> Destination of columns selected by `...`. Supplying neither will move columns to the left-hand side; specifying both is an error.

Fallbacks

There is no DuckDB translation in `relocate.duckplyr_df()`

- with a selection that returns no columns.

These features fall back to `dplyr::relocate()`, see `vignette("fallback")` for details.

See Also

dplyr::relocate()

Examples

```
df <- duckdb_tibble(a = 1, b = 1, c = 1, d = "a", e = "a", f = "a")
relocate(df, f)
```

---

rename.duckplyr_df *Rename columns*

Description

This is a method for the dplyr::rename() generic. See "Fallbacks" section for differences in implementation. rename() changes the names of individual variables using new_name = old_name syntax.

Usage

```
## S3 method for class 'duckplyr_df'
rename(.data, ...)
```

Arguments

.data	A data frame, data frame extension (e.g. a tibble), or a lazy data frame (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details.
...	For rename(): <tidy-select> Use new_name = old_name to rename selected variables. For rename_with(): additional arguments passed onto .fn.

Fallbacks

There is no DuckDB translation in rename.duckplyr_df()

- with a selection that returns no columns.

These features fall back to dplyr::rename(), see vignette("fallback") for details.

See Also

dplyr::rename()

Examples

```
library(duckplyr)
rename(mtcars, thing = mpg)
```

`right_join.duckplyr_df` *Right join*

Description

This is a method for the `dplyr::right_join()` generic. See "Fallbacks" section for differences in implementation. A `right_join()` keeps all observations in `y`.

Usage

```
## S3 method for class 'duckplyr_df'
right_join(
  x,
  y,
  by = NULL,
  copy = FALSE,
  suffix = c(".x", ".y"),
  ...,
  keep = NULL,
  na_matches = c("na", "never"),
  multiple = "all",
  unmatched = "drop",
  relationship = NULL
)
```

Arguments

`x, y`	A pair of data frames, data frame extensions (e.g. a tibble), or lazy data frames (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details.
`by`	A join specification created with `join_by()`, or a character vector of variables to join by. If `NULL`, the default, `*_join()` will perform a natural join, using all variables in common across `x` and `y`. A message lists the variables so that you can check they're correct; suppress the message by supplying `by` explicitly. To join on different variables between `x` and `y`, use a `join_by()` specification. For example, `join_by(a == b)` will match `x$a` to `y$b`. To join by multiple variables, use a `join_by()` specification with multiple expressions. For example, `join_by(a == b, c == d)` will match `x$a` to `y$b` and `x$c` to `y$d`. If the column names are the same between `x` and `y`, you can shorten this by listing only the variable names, like `join_by(a, c)`. `join_by()` can also be used to perform inequality, rolling, and overlap joins. See the documentation at ?join_by for details on these types of joins. For simple equality joins, you can alternatively specify a character vector of variable names to join by. For example, `by = c("a", "b")` joins `x$a` to `y$a` and

x$b to y$b. If variable names differ between x and y, use a named character vector like by = c("x_a" = "y_a", "x_b" = "y_b").

To perform a cross-join, generating all combinations of x and y, see cross_join().

copy — If x and y are not from the same data source, and copy is TRUE, then y will be copied into the same src as x. This allows you to join tables across srcs, but it is a potentially expensive operation so you must opt into it.

suffix — If there are non-joined duplicate variables in x and y, these suffixes will be added to the output to disambiguate them. Should be a character vector of length 2.

... — Other parameters passed onto methods.

keep — Should the join keys from both x and y be preserved in the output?

- If NULL, the default, joins on equality retain only the keys from x, while joins on inequality retain the keys from both inputs.
- If TRUE, all keys from both inputs are retained.
- If FALSE, only keys from x are retained. For right and full joins, the data in key columns corresponding to rows that only exist in y are merged into the key columns from x. Can't be used when joining on inequality conditions.

na_matches — Should two NA or two NaN values match?

- "na", the default, treats two NA or two NaN values as equal, like %in%, match(), and merge().
- "never" treats two NA or two NaN values as different, and will never match them together or to any other values. This is similar to joins for database sources and to base::merge(incomparables = NA).

multiple — Handling of rows in x with multiple matches in y. For each row of x:

- "all", the default, returns every match detected in y. This is the same behavior as SQL.
- "any" returns one match detected in y, with no guarantees on which match will be returned. It is often faster than "first" and "last" if you just need to detect if there is at least one match.
- "first" returns the first match detected in y.
- "last" returns the last match detected in y.

unmatched — How should unmatched keys that would result in dropped rows be handled?

- "drop" drops unmatched keys from the result.
- "error" throws an error if unmatched keys are detected.

unmatched is intended to protect you from accidentally dropping rows during a join. It only checks for unmatched keys in the input that could potentially drop rows.

- For left joins, it checks y.
- For right joins, it checks x.
- For inner joins, it checks both x and y. In this case, unmatched is also allowed to be a character vector of length 2 to specify the behavior for x and y independently.

relationship — Handling of the expected relationship between the keys of x and y. If the expectations chosen from the list below are invalidated, an error is thrown.

- NULL, the default, doesn't expect there to be any relationship between x and y. However, for equality joins it will check for a many-to-many relationship (which is typically unexpected) and will warn if one occurs, encouraging you to either take a closer look at your inputs or make this relationship explicit by specifying "many-to-many".
  See the *Many-to-many relationships* section for more details.
- "one-to-one" expects:
  - Each row in x matches at most 1 row in y.
  - Each row in y matches at most 1 row in x.
- "one-to-many" expects:
  - Each row in y matches at most 1 row in x.
- "many-to-one" expects:
  - Each row in x matches at most 1 row in y.
- "many-to-many" doesn't perform any relationship checks, but is provided to allow you to be explicit about this relationship if you know it exists.

relationship doesn't handle cases where there are zero matches. For that, see unmatched.

Fallbacks

There is no DuckDB translation in right_join.duckplyr_df()

- for an implicit cross join,
- for a value of the multiple argument that isn't the default "all".
- for a value of the unmatched argument that isn't the default "drop".

These features fall back to dplyr::right_join(), see vignette("fallback") for details.

See Also

dplyr::right_join()

Examples

```
library(duckplyr)
right_join(band_members, band_instruments)
```

---

select.duckplyr_df *Keep or drop columns using their names and types*

Description

This is a method for the dplyr::select() generic. See "Fallbacks" section for differences in implementation. Select (and optionally rename) variables in a data frame, using a concise mini-language that makes it easy to refer to variables based on their name (e.g. a:f selects all columns from a on the left to f on the right) or type (e.g. where(is.numeric) selects all numeric columns).

Usage

```
## S3 method for class 'duckplyr_df'
select(.data, ...)
```

Arguments

`.data`	A data frame, data frame extension (e.g. a tibble), or a lazy data frame (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details.
`...`	<tidy-select> One or more unquoted expressions separated by commas. Variable names can be used as if they were positions in the data frame, so expressions like `x:y` can be used to select a range of variables.

Fallbacks

There is no DuckDB translation in `select.duckplyr_df()`

- with no expression,
- nor with a selection that returns no columns.

These features fall back to `dplyr::select()`, see `vignette("fallback")` for details.

See Also

`dplyr::select()`

Examples

```
library(duckplyr)
select(mtcars, mpg)
```

`semi_join.duckplyr_df` *Semi join*

Description

This is a method for the `dplyr::semi_join()` generic. `semi_join()` returns all rows from x with a match in y.

Usage

```
## S3 method for class 'duckplyr_df'
semi_join(x, y, by = NULL, copy = FALSE, ..., na_matches = c("na", "never"))
```

Arguments

x, y	A pair of data frames, data frame extensions (e.g. a tibble), or lazy data frames (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details.
by	A join specification created with `join_by()`, or a character vector of variables to join by. If `NULL`, the default, `*_join()` will perform a natural join, using all variables in common across `x` and `y`. A message lists the variables so that you can check they're correct; suppress the message by supplying `by` explicitly. To join on different variables between `x` and `y`, use a `join_by()` specification. For example, `join_by(a == b)` will match `x$a` to `y$b`. To join by multiple variables, use a `join_by()` specification with multiple expressions. For example, `join_by(a == b, c == d)` will match `x$a` to `y$b` and `x$c` to `y$d`. If the column names are the same between `x` and `y`, you can shorten this by listing only the variable names, like `join_by(a, c)`. `join_by()` can also be used to perform inequality, rolling, and overlap joins. See the documentation at ?join_by for details on these types of joins. For simple equality joins, you can alternatively specify a character vector of variable names to join by. For example, `by = c("a", "b")` joins `x$a` to `y$a` and `x$b` to `y$b`. If variable names differ between `x` and `y`, use a named character vector like `by = c("x_a" = "y_a", "x_b" = "y_b")`. To perform a cross-join, generating all combinations of `x` and `y`, see `cross_join()`.
copy	If `x` and `y` are not from the same data source, and `copy` is `TRUE`, then `y` will be copied into the same src as `x`. This allows you to join tables across srcs, but it is a potentially expensive operation so you must opt into it.
...	Other parameters passed onto methods.
na_matches	Should two `NA` or two `NaN` values match? • `"na"`, the default, treats two `NA` or two `NaN` values as equal, like `%in%`, `match()`, and `merge()`. • `"never"` treats two `NA` or two `NaN` values as different, and will never match them together or to any other values. This is similar to joins for database sources and to `base::merge(incomparables = NA)`.

See Also

`dplyr::semi_join()`

Examples

```
library(duckplyr)
band_members %>% semi_join(band_instruments)
```

`setdiff.duckplyr_df` *Set difference*

Description

This is a method for the `dplyr::setdiff()` generic. See "Fallbacks" section for differences in implementation. `setdiff(x, y)` finds all rows in `x` that aren't in `y`.

Usage

```
## S3 method for class 'duckplyr_df'
setdiff(x, y, ...)
```

Arguments

`x`, `y`	Pair of compatible data frames. A pair of data frames is compatible if they have the same column names (possibly in different orders) and compatible types.
`...`	These dots are for future extensions and must be empty.

Fallbacks

There is no DuckDB translation in `setdiff.duckplyr_df()`

- if column names are duplicated in one of the tables,
- if column names are different in both tables.

These features fall back to `dplyr::setdiff()`, see `vignette("fallback")` for details.

See Also

`dplyr::setdiff()`

Examples

```
df1 <- duckdb_tibble(x = 1:3)
df2 <- duckdb_tibble(x = 3:5)
setdiff(df1, df2)
setdiff(df2, df1)
```

stats_show *Show stats*

Description

Prints statistics on how many calls were handled by DuckDB. The output shows the total number of requests in the current session, split by fallbacks to dplyr and requests handled by duckdb.

Usage

```
stats_show()
```

Value

Called for its side effect.

Examples

```
stats_show()

tibble(a = 1:3) %>%
  as_duckplyr_tibble() %>%
  mutate(b = a + 1)

stats_show()
```

summarise.duckplyr_df *Summarise each group down to one row*

Description

This is a method for the `dplyr::summarise()` generic. See "Fallbacks" section for differences in implementation. `summarise()` creates a new data frame. It returns one row for each combination of grouping variables; if there are no grouping variables, the output will have a single row summarising all observations in the input. It will contain one column for each grouping variable and one column for each of the summary statistics that you have specified.

Usage

```
## S3 method for class 'duckplyr_df'
summarise(.data, ..., .by = NULL, .groups = NULL)
```

Arguments

`.data`	A data frame, data frame extension (e.g. a tibble), or a lazy data frame (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details.
`...`	<data-masking> Name-value pairs of summary functions. The name will be the name of the variable in the result. The value can be: • A vector of length 1, e.g. `min(x)`, `n()`, or `sum(is.na(y))`. • A data frame, to add multiple columns from a single expression. **[Deprecated]** Returning values with size 0 or >1 was deprecated as of 1.1.0. Please use `reframe()` for this instead.
`.by`	**[Experimental]** <tidy-select> Optionally, a selection of columns to group by for just this operation, functioning as an alternative to `group_by()`. For details and examples, see ?dplyr_by.
`.groups`	**[Experimental]** Grouping structure of the result. • "drop_last": dropping the last level of grouping. This was the only supported option before version 1.0.0. • "drop": All levels of grouping are dropped. • "keep": Same grouping structure as `.data`. • "rowwise": Each row is its own group. When `.groups` is not specified, it is chosen based on the number of rows of the results: • If all the results have 1 row, you get "drop_last". • If the number of rows varies, you get "keep" (note that returning a variable number of rows was deprecated in favor of `reframe()`, which also unconditionally drops all levels of grouping). In addition, a message informs you of that choice, unless the result is ungrouped, the option "dplyr.summarise.inform" is set to `FALSE`, or when `summarise()` is called from a function in a package.

Fallbacks

There is no DuckDB translation in `summarise.duckplyr_df()`

- with `.groups = "rowwise"`.

These features fall back to `dplyr::summarise()`, see `vignette("fallback")` for details.

See Also

`dplyr::summarise()`

Examples

```
library(duckplyr)
summarise(mtcars, mean = mean(disp), n = n())
```

`symdiff.duckplyr_df` *Symmetric difference*

Description

This is a method for the `dplyr::symdiff()` generic. See "Fallbacks" section for differences in implementation. `symdiff(x, y)` computes the symmetric difference, i.e. all rows in `x` that aren't in `y` and all rows in `y` that aren't in `x`.

Usage

```
## S3 method for class 'duckplyr_df'
symdiff(x, y, ...)
```

Arguments

`x`, `y`	Pair of compatible data frames. A pair of data frames is compatible if they have the same column names (possibly in different orders) and compatible types.
`...`	These dots are for future extensions and must be empty.

Fallbacks

There is no DuckDB translation in `symdiff.duckplyr_df()`

- if column names are duplicated in one of the tables,
- if column names are different in both tables.

These features fall back to `dplyr::symdiff()`, see `vignette("fallback")` for details.

See Also

`dplyr::symdiff()`

Examples

```
df1 <- duckdb_tibble(x = 1:3)
df2 <- duckdb_tibble(x = 3:5)
symdiff(df1, df2)
```

`transmute.duckplyr_df` *Create, modify, and delete columns*

Description

This is a method for the `dplyr::transmute()` generic. See "Fallbacks" section for differences in implementation. `transmute()` creates a new data frame containing only the specified computations. It's superseded because you can perform the same job with `mutate(.keep = "none")`.

Usage

```
## S3 method for class 'duckplyr_df'
transmute(.data, ...)
```

Arguments

`.data`	A data frame, data frame extension (e.g. a tibble), or a lazy data frame (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details.
`...`	<`data-masking`> Name-value pairs. The name gives the name of the column in the output. The value can be: • A vector of length 1, which will be recycled to the correct length. • A vector the same length as the current group (or the whole data frame if ungrouped). • `NULL`, to remove the column. • A data frame or tibble, to create multiple columns in the output.

Fallbacks

There is no DuckDB translation in `transmute.duckplyr_df()`

- with a selection that returns no columns:

These features fall back to `dplyr::transmute()`, see `vignette("fallback")` for details.

See Also

`dplyr::transmute()`

Examples

```
library(duckplyr)
transmute(mtcars, mpg2 = mpg*2)
```

union.duckplyr_df *Union*

Description

This is a method for the dplyr::union() generic. union(x, y) finds all rows in either x or y, excluding duplicates. The implementation forwards to distinct(union_all(x, y)).

Usage

```
## S3 method for class 'duckplyr_df'
union(x, y, ...)
```

Arguments

x, y	Pair of compatible data frames. A pair of data frames is compatible if they have the same column names (possibly in different orders) and compatible types.
...	These dots are for future extensions and must be empty.

See Also

dplyr::union()

Examples

```
df1 <- duckdb_tibble(x = 1:3)
df2 <- duckdb_tibble(x = 3:5)
union(df1, df2)
```

union_all.duckplyr_df *Union of all*

Description

This is a method for the dplyr::union_all() generic. See "Fallbacks" section for differences in implementation. union_all(x, y) finds all rows in either x or y, including duplicates.

Usage

```
## S3 method for class 'duckplyr_df'
union_all(x, y, ...)
```

Arguments

x, y	Pair of compatible data frames. A pair of data frames is compatible if they have the same column names (possibly in different orders) and compatible types.
...	These dots are for future extensions and must be empty.

Fallbacks

There is no DuckDB translation in `union_all.duckplyr_df()`

- if column names are duplicated in one of the tables,
- if column names are different in both tables.

These features fall back to `dplyr::union_all()`, see `vignette("fallback")` for details.

See Also

`dplyr::union_all()`

Examples

```
df1 <- duckdb_tibble(x = 1:3)
df2 <- duckdb_tibble(x = 3:5)
union_all(df1, df2)
```

---

unsupported *Verbs not implemented in duckplyr*

Description

The following dplyr generics have no counterpart method in duckplyr. If you want to help add a new verb, please refer to our contributing guide https://duckplyr.tidyverse.org/CONTRIBUTING.html#support-new-verbs

Unsupported verbs

For these verbs, duckplyr will fall back to dplyr.

- `dplyr::add_count()`
- `dplyr::cross_join()`
- `dplyr::do()`
- `dplyr::group_by()`
- `dplyr::group_indices()`
- `dplyr::group_keys()`
- `dplyr::group_map()`
- `dplyr::group_modify()`
- `dplyr::group_nest()`
- `dplyr::group_size()`
- `dplyr::group_split()`
- `dplyr::group_trim()`
- `dplyr::groups()`

- dplyr::n_groups()
- dplyr::nest_by()
- dplyr::nest_join()
- dplyr::reframe()
- dplyr::rename_with()
- dplyr::rows_append()
- dplyr::rows_delete()
- dplyr::rows_insert()
- dplyr::rows_patch()
- dplyr::rows_update()
- dplyr::rows_upsert()
- dplyr::rowwise()
- generics::setequal()
- dplyr::slice_head()
- dplyr::slice_sample()
- dplyr::slice_tail()
- dplyr::slice()
- dplyr::ungroup()

Index