

Package: duckplyr (via r-universe)

May 24, 2026

Type Package

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

Version 1.2.1.9012

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 (>= 1.2.0)

Imports cli, collections, DBI, duckdb (>= 1.4.4), glue, jsonlite, lifecycle, magrittr, memoise, pillar (>= 1.10.2), rlang (>= 1.0.6), tibble, tidyselect, utils, vctrs (>= 0.6.3)

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

Enhances qs2

Config/Needs/check anthonymnorth/roxyglobals

Config/Needs/development devtools, qs2, reprex, r-lib/roxygen2, roxyglobals, rstudioapi, tidyverse

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

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

Roxygen list(markdown = TRUE, roclets = c(`\`collate", ``namespace",
 ``rd", ``roxyglobals::global_roclet"))

VignetteBuilder knitr

Config/roxygen2/version 8.0.0.9000

Config/pak/sysreqs xz-utils

Repository <https://fastverse.r-universe.dev>

Date/Publication 2026-05-24 04:56:01 UTC

RemoteUrl <https://github.com/tidyverse/duckplyr>

RemoteRef HEAD

RemoteSha fa9ae49cac0257b2f4d19f6577875c4470b0b6d8

Contents

| | |
|----------------------------------|----|
| anti_join.duckplyr_df | 3 |
| arrange.duckplyr_df | 4 |
| as_tbl | 6 |
| collect.duckplyr_df | 6 |
| compute.duckplyr_df | 7 |
| compute_csv | 8 |
| compute_parquet | 9 |
| config | 11 |
| count.duckplyr_df | 12 |
| db_exec | 13 |
| distinct.duckplyr_df | 14 |
| duckdb_tibble | 15 |
| explain.duckplyr_df | 16 |
| fallback | 17 |
| filter.duckplyr_df | 19 |
| flights_df | 20 |
| full_join.duckplyr_df | 21 |
| head.duckplyr_df | 23 |
| inner_join.duckplyr_df | 24 |
| intersect.duckplyr_df | 27 |
| last_rel | 28 |
| left_join.duckplyr_df | 28 |
| methods_overwrite | 31 |
| mutate.duckplyr_df | 32 |
| new_relational | 33 |
| new_relexpr | 37 |
| pull.duckplyr_df | 39 |
| read_csv_duckdb | 40 |
| read_file_duckdb | 41 |
| read_json_duckdb | 42 |
| read_parquet_duckdb | 43 |
| read_sql_duckdb | 44 |

| | |
|----------------------------------|----|
| read_tbl_duckdb | 45 |
| relocate.duckplyr_df | 46 |
| rename.duckplyr_df | 47 |
| right_join.duckplyr_df | 48 |
| select.duckplyr_df | 51 |
| semi_join.duckplyr_df | 52 |
| setdiff.duckplyr_df | 53 |
| slice_head.duckplyr_df | 54 |
| stats_show | 55 |
| summarise.duckplyr_df | 55 |
| symdiff.duckplyr_df | 57 |
| transmute.duckplyr_df | 58 |
| union.duckplyr_df | 59 |
| union_all.duckplyr_df | 59 |
| unsupported | 60 |

| | |
|--------------|-----------|
| Index | 62 |
|--------------|-----------|

anti_join.duckplyr_df *Anti join*

Description

This is a method for the `dplyr::anti_join()` generic. `anti_join()` returns all rows from `x` without 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

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

`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 <code>x</code> and <code>y</code> are not from the same data source, and <code>copy</code> is <code>TRUE</code> , then <code>y</code> will be copied into the same <code>src</code> as <code>x</code> . This allows you to join tables across <code>srcs</code> , 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? <ul style="list-style-type: none"> "na", the default, treats two NA or two NaN values as equal, like <code>%in%</code>, <code>match()</code>, and <code>merge()</code>. "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 <code>base::merge(incomparables = NA)</code>. |

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 deprecated `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(dis))
```

| | |
|--------|---|
| as_tbl | <i>Convert a duckplyr frame to a dbplyr table</i> |
|--------|---|

Description

[Experimental]

This function converts a lazy duckplyr frame or a data frame to a dbplyr table in duckplyr's internal connection. This allows using dbplyr functions on the data, including hand-written SQL queries. Use `as_duckdb_tibble()` to convert back to a lazy duckplyr frame.

Usage

```
as_tbl(.data)
```

Arguments

`.data` A lazy duckplyr frame or a data frame.

Value

A dbplyr table.

Examples

```
df <- duckdb_tibble(a = 1L)
df

tbl <- as_tbl(df)
tbl

tbl %>%
  mutate(b = sql("a + 1")) %>%
  as_duckdb_tibble()
```

| | |
|---------------------|---|
| collect.duckplyr_df | <i>Force conversion to a data frame</i> |
|---------------------|---|

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_csv

Compute results to a CSV file

Description

This is a generic function that executes a query and stores the results in a CSV file. For a duckplyr frame, the materialization occurs outside of R. The result is a duckplyr frame that can be used with subsequent dplyr verbs.

Usage

```
compute_csv(x, path, ...)

## S3 method for class 'duckplyr_df'
compute_csv(x, path, ..., prudence = NULL, options = NULL)

## S3 method for class 'data.frame'
compute_csv(x, path, ..., prudence = NULL, options = NULL)
```

Arguments

| | |
|----------|---|
| x | A data frame or lazy data frame. |
| path | The path of the CSV file to create. |
| ... | Additional arguments passed to methods. |
| prudence | Memory protection, controls if DuckDB may convert intermediate results in DuckDB-managed memory to data frames in R memory. <ul style="list-style-type: none"> • "lavish": regardless of size, • "stingy": never, • "thrifty": up to a maximum size of 1 million cells. <p>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 <code>as_duckdb_tibble()</code> with the desired prudence. See <code>vignette("prudence")</code> for more information.</p> |
| options | A list of additional options to pass to create the storage format, see https://duckdb.org/docs/sql/statements/copy.html#csv-options for details. |

Value

A data frame (the class may vary based on the input).

See Also

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

Examples

```
library(duckplyr)
df <- data.frame(x = c(1, 2))
df <- mutate(df, y = 2)
path <- tempfile(fileext = ".csv")
df <- compute_csv(df, path)
readLines(path)
```

compute_parquet

Compute results to a Parquet file

Description

This is a generic function that executes a query and stores the results in a Parquet file. For a duckplyr frame, the materialization occurs outside of R. The result is a duckplyr frame that can be used with subsequent dplyr verbs.

Usage

```
compute_parquet(x, path, ...)

## S3 method for class 'duckplyr_df'
compute_parquet(x, path, ..., prudence = NULL, options = NULL)

## S3 method for class 'data.frame'
compute_parquet(x, path, ..., prudence = NULL, options = NULL)
```

Arguments

| | |
|----------|---|
| x | A data frame or lazy data frame. |
| path | The path of the Parquet file to create. |
| ... | Additional arguments passed to methods. |
| prudence | Memory protection, controls if DuckDB may convert intermediate results in DuckDB-managed memory to data frames in R memory. <ul style="list-style-type: none"> • "lavish": regardless of size, • "stingy": never, • "thrifty": up to a maximum size of 1 million cells. <p>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 <code>as_duckdb_tibble()</code> with the desired prudence. See <code>vignette("prudence")</code> for more information.</p> |
| options | A list of additional options to pass to create the Parquet file, see https://duckdb.org/docs/sql/statements/copy.html#parquet-options for details. |

Value

A data frame (the class may vary based on the input).

See Also

`compute_csv()`, `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 | <i>Configuration options</i> |
|--------|------------------------------|

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_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

| | |
|-------------------|---|
| <code>x</code> | A data frame, data frame extension (e.g. a tibble), or a lazy data frame (e.g. from <code>dbplyr</code> or <code>dtplyr</code>). |
| <code>...</code> | <data-masking> Variables to group by. |
| <code>wt</code> | <data-masking> Frequency weights. Can be <code>NULL</code> or a variable: <ul style="list-style-type: none"> • If <code>NULL</code> (the default), counts the number of rows in each group. • If a variable, computes <code>sum(wt)</code> for each group. |
| <code>sort</code> | If <code>TRUE</code> , will show the largest groups at the top. |
| <code>name</code> | The name of the new column in the output. If omitted, it will default to <code>n</code> . If there's already a column called <code>n</code> , it will use <code>nn</code> . If there's a column called <code>n</code> and <code>nn</code> , it'll use <code>nnn</code> , and so on, adding <code>ns</code> 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).
[Defunct] For `add_count()`: defunct 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

| | |
|------------------------|---|
| <code>.data</code> | A data frame, data frame extension (e.g. a tibble), or a lazy data frame (e.g. from <code>dbplyr</code> or <code>dtplyr</code>). See <i>Methods</i> , below, for more details. |
| <code>...</code> | <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. |
| <code>.keep_all</code> | If TRUE, keep all variables in <code>.data</code> . If a combination of <code>...</code> 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 | <i>duckplyr data frames</i> |
|---------------|-----------------------------|

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 `mutate()`, `select()` or `filter()` will use DuckDB.

`duckdb_tibble()` works like `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()`. 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`

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

| | |
|------------------|---|
| <code>x</code> | An object to explain |
| <code>...</code> | 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 displayed. |
| 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::filter()` 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)

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

Arguments

| | |
|------------------------|--|
| <code>.data</code> | A data frame, data frame extension (e.g. a tibble), or a lazy data frame (e.g. from <code>dbplyr</code> or <code>dtplyr</code>). See <i>Methods</i> , below, for more details. |
| <code>...</code> | <data-masking> Expressions that return a logical vector, defined in terms of the variables in <code>.data</code> . If multiple expressions are included, they are combined with the <code>&</code> operator. To combine expressions using <code> </code> instead, wrap them in <code>when_any()</code> . Only rows for which all expressions evaluate to TRUE are kept (for <code>filter()</code>) or dropped (for <code>filter_out()</code>). |
| <code>.by</code> | <tidy-select> Optionally, a selection of columns to group by for just this operation, functioning as an alternative to <code>group_by()</code> . For details and examples, see <code>?dplyr_by</code> . |
| <code>.preserve</code> | Relevant when the <code>.data</code> input is grouped. If <code>.preserve = FALSE</code> (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.

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

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

These features fall back to `dplyr::filter_out()`, 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

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

| | |
|--------------|---|
| copy | If <code>x</code> and <code>y</code> are not from the same data source, and <code>copy</code> is <code>TRUE</code> , then <code>y</code> will be copied into the same <code>src</code> as <code>x</code> . This allows you to join tables across <code>srcs</code> , but it is a potentially expensive operation so you must opt into it. |
| suffix | If there are non-joined duplicate variables in <code>x</code> and <code>y</code> , 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 <code>x</code> and <code>y</code> be preserved in the output? <ul style="list-style-type: none"> • If <code>NULL</code>, the default, joins on equality retain only the keys from <code>x</code>, while joins on inequality retain the keys from both inputs. • If <code>TRUE</code>, all keys from both inputs are retained. • If <code>FALSE</code>, only keys from <code>x</code> are retained. For right and full joins, the data in key columns corresponding to rows that only exist in <code>y</code> are merged into the key columns from <code>x</code>. Can't be used when joining on inequality conditions. |
| na_matches | Should two NA or two NaN values match? <ul style="list-style-type: none"> • "na", the default, treats two NA or two NaN values as equal, like <code>%in%</code>, <code>match()</code>, and <code>merge()</code>. • "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 <code>base::merge(incomparables = NA)</code>. |
| multiple | Handling of rows in <code>x</code> with multiple matches in <code>y</code> . For each row of <code>x</code> : <ul style="list-style-type: none"> • "all", the default, returns every match detected in <code>y</code>. This is the same behavior as SQL. • "any" returns one match detected in <code>y</code>, 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 <code>y</code>. • "last" returns the last match detected in <code>y</code>. |
| relationship | Handling of the expected relationship between the keys of <code>x</code> and <code>y</code> . If the expectations chosen from the list below are invalidated, an error is thrown. <ul style="list-style-type: none"> • <code>NULL</code>, the default, doesn't expect there to be any relationship between <code>x</code> and <code>y</code>. 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 <i>Many-to-many relationships</i> section for more details. • "one-to-one" expects: <ul style="list-style-type: none"> – Each row in <code>x</code> matches at most 1 row in <code>y</code>. – Each row in <code>y</code> matches at most 1 row in <code>x</code>. • "one-to-many" expects: <ul style="list-style-type: none"> – Each row in <code>y</code> matches at most 1 row in <code>x</code>. • "many-to-one" expects: <ul style="list-style-type: none"> – Each row in <code>x</code> matches at most 1 row in <code>y</code>. • "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

| | |
|------------------|--|
| <code>x</code> | A <code>data.frame</code> |
| <code>n</code> | A positive integer, how many rows to return. |
| <code>...</code> | 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

- | | |
|-------------------|---|
| <code>x, y</code> | A pair of data frames, data frame extensions (e.g. a tibble), or lazy data frames (e.g. from <code>dbplyr</code> or <code>dtplyr</code>). See <i>Methods</i> , below, for more details. |
| <code>by</code> | A join specification created with <code>join_by()</code> , or a character vector of variables to join by. If <code>NULL</code> , the default, <code>*_join()</code> will perform a natural join, using all variables in common across <code>x</code> and <code>y</code> . A message lists the variables so that you can check they're correct; suppress the message by supplying <code>by</code> explicitly. To join on different variables between <code>x</code> and <code>y</code> , use a <code>join_by()</code> specification. For example, <code>join_by(a == b)</code> will match <code>x\$a</code> to <code>y\$b</code> . To join by multiple variables, use a <code>join_by()</code> specification with multiple expressions. For example, <code>join_by(a == b, c == d)</code> will match <code>x\$a</code> to <code>y\$b</code> 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 <code>copy</code> 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? <ul style="list-style-type: none"> • 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? <ul style="list-style-type: none"> • "na", the default, treats two NA or two NaN values as equal, like <code>%in%</code>, <code>match()</code>, and <code>merge()</code>. • "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 <code>base::merge(incomparables = NA)</code>. |
| multiple | Handling of rows in x with multiple matches in y . For each row of x : <ul style="list-style-type: none"> • "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? <ul style="list-style-type: none"> • "drop" drops unmatched keys from the result. • "error" throws an error if unmatched keys are detected. <p><code>unmatched</code> 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.</p> <ul style="list-style-type: none"> • 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

| | |
|-------------------|---|
| <code>x, y</code> | 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. |
| <code>...</code> | 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 | <i>Retrieve details about the most recent computation</i> |
|----------|---|

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 | <i>Left join</i> |
|-----------------------|------------------|

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 <i>Methods</i> , below, for more details. |
| by | <p>A join specification created with <code>join_by()</code>, or a character vector of variables to join by.</p> <p>If NULL, the default, <code>*_join()</code> 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.</p> <p>To join on different variables between x and y, use a <code>join_by()</code> specification. For example, <code>join_by(a == b)</code> will match x\$a to y\$b.</p> <p>To join by multiple variables, use a <code>join_by()</code> specification with multiple expressions. For example, <code>join_by(a == b, c == d)</code> 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 <code>join_by(a, c)</code>.</p> <p><code>join_by()</code> can also be used to perform inequality, rolling, and overlap joins. See the documentation at ?join_by for details on these types of joins.</p> <p>For simple equality joins, you can alternatively specify a character vector of variable names to join by. For example, <code>by = c("a", "b")</code> 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 <code>by = c("x_a" = "y_a", "x_b" = "y_b")</code>.</p> <p>To perform a cross-join, generating all combinations of x and y, see <code>cross_join()</code>.</p> |
| 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 | <p>Should the join keys from both x and y be preserved in the output?</p> <ul style="list-style-type: none"> • 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 | <p>Should two NA or two NaN values match?</p> <ul style="list-style-type: none"> • "na", the default, treats two NA or two NaN values as equal, like <code>%in%</code>, <code>match()</code>, and <code>merge()</code>. • "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 <code>base::merge(incomparables = NA)</code>. |
| multiple | <p>Handling of rows in x with multiple matches in y. For each row of x:</p> <ul style="list-style-type: none"> • "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 duration 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

| | |
|--------------------|--|
| <code>.data</code> | A data frame, data frame extension (e.g. a tibble), or a lazy data frame (e.g. from <code>dbplyr</code> or <code>dtplyr</code>). See <i>Methods</i> , below, for more details. |
| <code>...</code> | <p><data-masking> Name-value pairs. The name gives the name of the column in the output.</p> <p>The value can be:</p> <ul style="list-style-type: none"> • 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). • <code>NULL</code>, to remove the column. • A data frame or tibble, to create multiple columns in the output. |
| <code>.by</code> | <tidy-select> Optionally, a selection of columns to group by for just this operation, functioning as an alternative to <code>group_by()</code> . For details and examples, see <code>?dplyr_by</code> . |
| <code>.keep</code> | <p>Control which columns from <code>.data</code> are retained in the output. Grouping columns and columns created by <code>...</code> are always kept.</p> <ul style="list-style-type: none"> • "all" retains all columns from <code>.data</code>. This is the default. • "used" retains only the columns used in <code>...</code> 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 <i>not</i> used in <code>...</code> 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 `intersect()`.

`rel_set_diff()` returns rows present in any of both tables, to be used by `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 <code>new_relexpr()</code> . |
| 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, order_bys = NULL)
```


Examples

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

| | |
|------------------|--------------------------------|
| pull.duckplyr_df | <i>Extract a single column</i> |
|------------------|--------------------------------|

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

| | |
|--------------------|---|
| <code>.data</code> | A data frame, data frame extension (e.g. a tibble), or a lazy data frame (e.g. from <code>dbplyr</code> or <code>dtplyr</code>). See <i>Methods</i> , below, for more details. |
| <code>var</code> | <p>A variable specified as:</p> <ul style="list-style-type: none"> • a literal variable name • a positive integer, giving the position counting from the left • a negative integer, giving the position counting from the right. <p>The default returns the last column (on the assumption that's the column you've created most recently).</p> <p>This argument is taken by expression and supports quasiquotation (you can unquote column names and column locations).</p> |
| <code>name</code> | An optional parameter that specifies the column to be used as names for a named vector. Specified in a similar manner as <code>var</code> . |
| <code>...</code> | 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_csv_duckdb | <i>Read CSV files using DuckDB</i> |
|-----------------|------------------------------------|

Description

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

Usage

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

Arguments

| | |
|-----------------------|---|
| <code>path</code> | Path to files, glob patterns <code>*</code> and <code>?</code> are supported. |
| <code>...</code> | These dots are for future extensions and must be empty. |
| <code>prudence</code> | Memory protection, controls if DuckDB may convert intermediate results in DuckDB-managed memory to data frames in R memory. <ul style="list-style-type: none"> "thrifty": up to a maximum size of 1 million cells, "lavish": regardless of size, "stingy": never. <p>The default is "thrifty" for the ingestion functions, and may be different for other functions. See <code>vignette("prudence")</code> for more information.</p> |
| <code>options</code> | Arguments to the DuckDB <code>read_csv_auto</code> table function. |

See Also

[read_parquet_duckdb\(\)](#), [read_json_duckdb\(\)](#)

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 positionally
read_csv_duckdb(
  path,
  options = list(delim = ",", types = list(c("DOUBLE", "VARCHAR")))
)

# Specify column types by name
read_csv_duckdb(
  path,
  options = list(types = list(c(a = "DOUBLE", b = "VARCHAR")))
)

```

read_file_duckdb

Read files using DuckDB

Description

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_file_duckdb(
  path,
  table_function,
  ...,
  prudence = c("thrifty", "lavish", "stingy"),

```

```

  options = list()
)

```

Arguments

| | |
|----------------|---|
| path | Path to files, glob patterns * and ? are supported. |
| table_function | The name of a table-valued DuckDB function such as "read_parquet", "read_csv", "read_csv_auto" or "read_json". |
| ... | 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. <ul style="list-style-type: none"> "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. |

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)`.

See Also

[read_csv_duckdb\(\)](#), [read_parquet_duckdb\(\)](#), [read_json_duckdb\(\)](#)

| | |
|------------------|-------------------------------------|
| read_json_duckdb | <i>Read JSON files using DuckDB</i> |
|------------------|-------------------------------------|

Description

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

Usage

```

read_json_duckdb(
  path,
  ...,
  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. <ul style="list-style-type: none"> • "thrifty": up to a maximum size of 1 million cells, • "lavish": regardless of size, • "stingy": never. <p>The default is "thrifty" for the ingestion functions, and may be different for other functions. See vignette("prudence") for more information.</p> |
| options | Arguments to the DuckDB read_json table function. |

See Also

[read_csv_duckdb\(\)](#), [read_parquet_duckdb\(\)](#)

Examples

```
# 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_parquet_duckdb *Read Parquet files using DuckDB*

Description

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

Usage

```
read_parquet_duckdb(
  path,
  ...,
  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. <ul style="list-style-type: none"> • "thrifty": up to a maximum size of 1 million cells, • "lavish": regardless of size, • "stingy": never. <p>The default is "thrifty" for the ingestion functions, and may be different for other functions. See vignette("prudence") for more information.</p> |
| options | Arguments to the DuckDB read_parquet table function. |

See Also

[read_csv_duckdb\(\)](#), [read_json_duckdb\(\)](#)

| | |
|-----------------|--|
| read_sql_duckdb | <i>Return SQL query as duckdb_tibble</i> |
|-----------------|--|

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. <ul style="list-style-type: none"> • "thrifty": up to a maximum size of 1 million cells, • "lavish": regardless of size, • "stingy": never. <p>The default is "thrifty" for the ingestion functions, and may be different for other functions. See vignette("prudence") for more information.</p> |
| 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()")
```

| | |
|-----------------|---|
| read_tbl_duckdb | <i>Read a table from a DuckDB database file</i> |
|-----------------|---|

Description**[Experimental]**

`read_tbl_duckdb()` reads a table from a DuckDB database file by attaching the database file and reading the specified table.

The database file is attached to the default duckplyr connection and remains attached for the duration of the R session to support lazy evaluation.

Usage

```
read_tbl_duckdb(
  path,
  table_name,
  ...,
  schema = "main",
  prudence = c("thrifty", "lavish", "stingy")
)
```

Arguments

| | |
|------------|--|
| path | Path to the DuckDB database file. |
| table_name | The name of the table to read. |
| ... | These dots are for future extensions and must be empty. |
| schema | The schema name where the table is located. Defaults to "main". |
| prudence | Memory protection, controls if DuckDB may convert intermediate results in DuckDB-managed memory to data frames in R memory. <ul style="list-style-type: none"> • "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 <code>vignette("prudence")</code> for more information. |

Value

A duckplyr frame, see [as_duckdb_tibble\(\)](#) for details.

See Also

[read_sql_duckdb\(\)](#), [db_exec\(\)](#)

Examples

```
# Create a temporary DuckDB database with a table
db_path <- tempfile(fileext = ".duckdb")
con <- DBI::dbConnect(duckdb::duckdb(), db_path)
DBI::dbWriteTable(con, "test_table", data.frame(a = 1:3, b = c("x", "y", "z")))
DBI::dbDisconnect(con)

# Read the table from the database file
df <- read_tbl_duckdb(db_path, "test_table")
print(df)

# Clean up
unlink(db_path)
```

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

| | |
|--|---|
| <code>.data</code> | A data frame, data frame extension (e.g. a tibble), or a lazy data frame (e.g. from <code>dbplyr</code> or <code>dtplyr</code>). See <i>Methods</i> , below, for more details. |
| <code>...</code> | <tidy-select> Columns to move. |
| <code>.before</code> , <code>.after</code> | <tidy-select> Destination of columns selected by <code>...</code> . 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)
```

| | |
|---------------------------------|-----------------------|
| <code>rename.duckplyr_df</code> | <i>Rename columns</i> |
|---------------------------------|-----------------------|

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

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

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

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

`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 <code>x</code> and <code>y</code> are not from the same data source, and <code>copy</code> is <code>TRUE</code> , then <code>y</code> will be copied into the same <code>src</code> as <code>x</code> . This allows you to join tables across <code>srcs</code> , but it is a potentially expensive operation so you must opt into it. |
| suffix | If there are non-joined duplicate variables in <code>x</code> and <code>y</code> , 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 <code>x</code> and <code>y</code> be preserved in the output? <ul style="list-style-type: none"> • If <code>NULL</code>, the default, joins on equality retain only the keys from <code>x</code>, while joins on inequality retain the keys from both inputs. • If <code>TRUE</code>, all keys from both inputs are retained. • If <code>FALSE</code>, only keys from <code>x</code> are retained. For right and full joins, the data in key columns corresponding to rows that only exist in <code>y</code> are merged into the key columns from <code>x</code>. Can't be used when joining on inequality conditions. |
| na_matches | Should two NA or two NaN values match? <ul style="list-style-type: none"> • <code>"na"</code>, the default, treats two NA or two NaN values as equal, like <code>%in%</code>, <code>match()</code>, and <code>merge()</code>. • <code>"never"</code> 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 <code>base::merge(incomparables = NA)</code>. |
| multiple | Handling of rows in <code>x</code> with multiple matches in <code>y</code> . For each row of <code>x</code> : <ul style="list-style-type: none"> • <code>"all"</code>, the default, returns every match detected in <code>y</code>. This is the same behavior as SQL. • <code>"any"</code> returns one match detected in <code>y</code>, with no guarantees on which match will be returned. It is often faster than <code>"first"</code> and <code>"last"</code> if you just need to detect if there is at least one match. • <code>"first"</code> returns the first match detected in <code>y</code>. • <code>"last"</code> returns the last match detected in <code>y</code>. |
| unmatched | How should unmatched keys that would result in dropped rows be handled? <ul style="list-style-type: none"> • <code>"drop"</code> drops unmatched keys from the result. • <code>"error"</code> throws an error if unmatched keys are detected. <p><code>unmatched</code> 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.</p> <ul style="list-style-type: none"> • For left joins, it checks <code>y</code>. • For right joins, it checks <code>x</code>. |

- 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

| | |
|--------------------|--|
| <code>.data</code> | A data frame, data frame extension (e.g. a tibble), or a lazy data frame (e.g. from <code>dbplyr</code> or <code>dtplyr</code>). See <i>Methods</i> , below, for more details. |
| <code>...</code> | <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 <code>x:y</code> 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

| | |
|-------------------------|---|
| <code>x, y</code> | A pair of data frames, data frame extensions (e.g. a tibble), or lazy data frames (e.g. from <code>dbplyr</code> or <code>dtplyr</code>). See <i>Methods</i> , below, for more details. |
| <code>by</code> | A join specification created with <code>join_by()</code> , or a character vector of variables to join by. If <code>NULL</code> , the default, <code>*_join()</code> will perform a natural join, using all variables in common across <code>x</code> and <code>y</code> . A message lists the variables so that you can check they're correct; suppress the message by supplying <code>by</code> explicitly. To join on different variables between <code>x</code> and <code>y</code> , use a <code>join_by()</code> specification. For example, <code>join_by(a == b)</code> will match <code>x\$a</code> to <code>y\$b</code> . To join by multiple variables, use a <code>join_by()</code> specification with multiple expressions. For example, <code>join_by(a == b, c == d)</code> will match <code>x\$a</code> to <code>y\$b</code> and <code>x\$c</code> to <code>y\$d</code> . If the column names are the same between <code>x</code> and <code>y</code> , you can shorten this by listing only the variable names, like <code>join_by(a, c)</code> . <code>join_by()</code> 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, <code>by = c("a", "b")</code> joins <code>x\$a</code> to <code>y\$a</code> and <code>x\$b</code> to <code>y\$b</code> . If variable names differ between <code>x</code> and <code>y</code> , use a named character vector like <code>by = c("x_a" = "y_a", "x_b" = "y_b")</code> . To perform a cross-join, generating all combinations of <code>x</code> and <code>y</code> , see <code>cross_join()</code> . |
| <code>copy</code> | If <code>x</code> and <code>y</code> are not from the same data source, and <code>copy</code> is <code>TRUE</code> , then <code>y</code> will be copied into the same <code>src</code> as <code>x</code> . This allows you to join tables across <code>srcs</code> , but it is a potentially expensive operation so you must opt into it. |
| <code>...</code> | Other parameters passed onto methods. |
| <code>na_matches</code> | Should two NA or two NaN values match? <ul style="list-style-type: none"> "na", the default, treats two NA or two NaN values as equal, like <code>%in%</code>, <code>match()</code>, and <code>merge()</code>. "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 <code>base::merge(incomparables = NA)</code>. |

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)
```

 slice_head.duckplyr_df

Subset rows using their positions

Description

This is a method for the `dplyr::slice_head()` generic. `slice_head()` selects the first rows.

Usage

```
## S3 method for class 'duckplyr_df'
slice_head(.data, ..., n, prop, by = NULL)
```

Arguments

| | |
|----------------------|---|
| <code>.data</code> | A data frame, data frame extension (e.g. a tibble), or a lazy data frame (e.g. from <code>dbplyr</code> or <code>dtplyr</code>). See <i>Methods</i> , below, for more details. |
| <code>...</code> | For <code>slice()</code> : <code><data-masking></code> Integer row values. Provide either positive values to keep, or negative values to drop. The values provided must be either all positive or all negative. Indices beyond the number of rows in the input are silently ignored. For <code>slice_*</code> (), these arguments are passed on to methods. |
| <code>n, prop</code> | Provide either <code>n</code> , the number of rows, or <code>prop</code> , the proportion of rows to select. If neither are supplied, <code>n = 1</code> will be used. If <code>n</code> is greater than the number of rows in the group (or <code>prop > 1</code>), the result will be silently truncated to the group size. <code>prop</code> will be rounded towards zero to generate an integer number of rows. A negative value of <code>n</code> or <code>prop</code> will be subtracted from the group size. For example, <code>n = -2</code> with a group of 5 rows will select $5 - 2 = 3$ rows; <code>prop = -0.25</code> with 8 rows will select $8 * (1 - 0.25) = 6$ rows. |
| <code>by</code> | <code><tidy-select></code> Optionally, a selection of columns to group by for just this operation, functioning as an alternative to <code>group_by()</code> . For details and examples, see <code>?dplyr_by</code> . |

Fallbacks

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

- if `by` or `prop` is provided,
- with a negative `n`.

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

See Also

`dplyr::slice_head()`

Examples

```
library(duckplyr)
df <- data.frame(x = 1:3)
df <- slice_head(df, n = 2)
df
```

| | |
|------------|-------------------|
| stats_show | <i>Show stats</i> |
|------------|-------------------|

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 | <i>Summarise each group down to one row</i> |
|-----------------------|---|

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 with 1 row, to add multiple columns from a single expression.
- `.by` [<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": drops 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 set to "drop_last" for a grouped data frame, and "keep" for a rowwise data frame. In addition, a message informs you of how the result will be grouped 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(displacement), 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

| | |
|-------------------|---|
| <code>x, y</code> | 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. |
| <code>...</code> | 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

[Superseded]

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

| | |
|--------------------|--|
| <code>.data</code> | A data frame, data frame extension (e.g. a tibble), or a lazy data frame (e.g. from <code>dbplyr</code> or <code>dtplyr</code>). See <i>Methods</i> , below, for more details. |
| <code>...</code> | <code><data-masking></code> Name-value pairs. The name gives the name of the column in the output. The value can be: <ul style="list-style-type: none"> • 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.

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

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

Index

?dplyr_by, [19](#), [32](#), [54](#), [56](#)
?join_by, [4](#), [21](#), [25](#), [29](#), [49](#), [52](#)

add_count(), [60](#)
anti_join.duckplyr_df, [3](#)
arrange.duckplyr_df, [4](#)
as_duckdb_tibble (duckdb_tibble), [15](#)
as_duckdb_tibble(), [6](#), [8–10](#), [20](#), [42](#), [46](#)
as_tbl, [6](#)

collect.duckplyr_df, [6](#)
colnames(), [34](#)
compute.duckplyr_df, [7](#)
compute.duckplyr_df(), [9](#), [10](#)
compute_csv, [8](#)
compute_csv(), [10](#)
compute_parquet, [9](#)
compute_parquet(), [9](#)
config, [11](#)
count.duckplyr_df, [12](#)
cross_join(), [4](#), [21](#), [25](#), [29](#), [49](#), [52](#), [60](#)

db_exec, [13](#)
db_exec(), [45](#), [46](#)
DBI::dbExecute(), [14](#)
desc(), [5](#)
distinct.duckplyr_df, [14](#)
do(), [60](#)
dplyr-locale, [5](#)
dplyr::anti_join(), [3](#), [4](#), [33](#)
dplyr::arrange(), [4](#), [5](#), [33](#)
dplyr::collect(), [6–10](#), [33](#)
dplyr::compute(), [7](#)
dplyr::count(), [12](#), [13](#)
dplyr::cross_join(), [33](#)
dplyr::distinct(), [14](#), [33](#)
dplyr::explain(), [16](#)
dplyr::filter(), [19](#), [20](#), [33](#)
dplyr::filter_out(), [20](#)
dplyr::full_join(), [21](#), [23](#), [33](#)
dplyr::inner_join(), [24](#), [26](#), [33](#)
dplyr::intersect(), [27](#)
dplyr::left_join(), [28](#), [30](#), [31](#), [33](#)
dplyr::mutate(), [32](#), [33](#)
dplyr::pull(), [39](#), [40](#)
dplyr::relocate(), [33](#), [46](#), [47](#)
dplyr::rename(), [33](#), [47](#)
dplyr::right_join(), [33](#), [48](#), [50](#)
dplyr::select(), [33](#), [51](#)
dplyr::semi_join(), [33](#), [52](#), [53](#)
dplyr::setdiff(), [53](#)
dplyr::slice_head(), [54](#)
dplyr::summarise(), [55](#), [56](#)
dplyr::summarize(), [33](#)
dplyr::syndiff(), [33](#), [57](#)
dplyr::transmute(), [58](#)
dplyr::union(), [59](#)
dplyr::union_all(), [33](#), [59](#), [60](#)
duckdb_tibble, [15](#)

explain.duckplyr_df, [16](#)

fallback, [11](#), [17](#)
fallback_config (fallback), [17](#)
fallback_purge (fallback), [17](#)
fallback_review (fallback), [17](#)
fallback_sitrep (fallback), [17](#)
fallback_upload (fallback), [17](#)
filter(), [15](#)
filter.duckplyr_df, [19](#)
filter_out.duckplyr_df
 (filter.duckplyr_df), [19](#)
flights_df, [20](#)
full_join.duckplyr_df, [21](#)

group_by(), [13](#), [19](#), [32](#), [54](#), [56](#), [60](#)
group_indices(), [60](#)
group_keys(), [60](#)
group_map(), [60](#)
group_modify(), [60](#)

group_nest(), 60
 group_size(), 60
 group_split(), 60
 group_trim(), 60
 groups(), 60

 head(), 23, 24
 head.duckplyr_df, 23

 inner_join.duckplyr_df, 24
 intersect(), 33
 intersect.duckplyr_df, 27
 is_duckdb_tibble (duckdb_tibble), 15

 join_by(), 3, 4, 21, 24, 25, 29, 48, 49, 52

 last_rel, 28
 left_join.duckplyr_df, 28
 locale, 5

 match(), 4, 22, 25, 29, 49, 52
 merge(), 4, 22, 25, 29, 49, 52
 methods_overwrite, 31
 methods_restore (methods_overwrite), 31
 mutate(), 15
 mutate.duckplyr_df, 32

 n_groups(), 61
 nest_by(), 61
 nest_join(), 61
 new_relational, 33
 new_relexpr, 37
 new_relexpr(), 35

 pull.duckplyr_df, 39

 quasiquotation, 39

 read_csv_duckdb, 40
 read_csv_duckdb(), 42–44
 read_file_duckdb, 41
 read_json_duckdb, 42
 read_json_duckdb(), 40, 42, 44
 read_parquet_duckdb, 43
 read_parquet_duckdb(), 40, 42, 43
 read_sql_duckdb, 44
 read_sql_duckdb(), 14, 46
 read_tbl_duckdb, 45
 reframe(), 61
 rel_aggregate (new_relational), 33
 rel_alias (new_relational), 33
 rel_distinct (new_relational), 33
 rel_explain (new_relational), 33
 rel_filter (new_relational), 33
 rel_join (new_relational), 33
 rel_limit (new_relational), 33
 rel_names (new_relational), 33
 rel_order (new_relational), 33
 rel_project (new_relational), 33
 rel_set_alias (new_relational), 33
 rel_set_diff (new_relational), 33
 rel_set_intersect (new_relational), 33
 rel_set_syndiff (new_relational), 33
 rel_to_df (new_relational), 33
 rel_union_all (new_relational), 33
 relexpr_comparison (new_relexpr), 37
 relexpr_constant (new_relexpr), 37
 relexpr_function (new_relexpr), 37
 relexpr_reference (new_relexpr), 37
 relexpr_set_alias (new_relexpr), 37
 relexpr_window (new_relexpr), 37
 relocate(), 33
 relocate.duckplyr_df, 46
 rename.duckplyr_df, 47
 rename_with(), 61
 right_join.duckplyr_df, 48
 rows_append(), 61
 rows_delete(), 61
 rows_insert(), 61
 rows_patch(), 61
 rows_update(), 61
 rows_upsert(), 61
 rowwise(), 61

 select(), 15
 select.duckplyr_df, 51
 semi_join.duckplyr_df, 52
 setdiff(), 33
 setdiff.duckplyr_df, 53
 setequal(), 61
 slice(), 61
 slice_head(), 54
 slice_head.duckplyr_df, 54
 slice_sample(), 61
 slice_tail(), 61
 stats_show, 55
 stringi::stri_locale_list(), 5
 summarise.duckplyr_df, 55
 syndiff.duckplyr_df, 57

`tempdir()`, [11](#)
`tibble`, [15](#)
`tibble()`, [15](#)
`transmute.duckplyr_df`, [58](#)

`ungroup()`, [61](#)
`union.duckplyr_df`, [59](#)
`union_all.duckplyr_df`, [59](#)
`unsupported`, [60](#)
`usethis::edit_r_envirion()`, [18](#)
`utils::head()`, [33](#)

`when_any()`, [19](#)