Package: duckplyr (via r-universe)

August 21, 2024

Type Package

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

Version 0.4.1.9003

Description A drop-in replacement for 'dplyr', powered by 'DuckDB' for performance. Also defines a set of generics that provide a low-level implementer's interface for the high-level user interface of 'dplyr'.

License MIT + file LICENSE

URL `https://tidyverse.github.io/duckplyr`, `https://github.com/tidyverse/duckplyr`

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

Depends R (>= 4.1.0)

Imports cli, collections, DBI, dplyr (>= 1.1.4), duckdb (>= 0.10.2), glue, jsonlite, lifecycle, rlang (>= 1.0.6), tibble, tidyselect, utils, vctrs (>= 0.6.3)

Suggests arrow, brio, constructive (>= 1.0.0), curl, dbplyr, hms, lobstr, lubridate, palmerpenguins, pillar, prettycode, purrr, qs, reprex, rstudioapi, testthat (>= 3.1.5), usethis, withr

Config/Needs/check anthonynorth/roxyglobals

Config/testthat/edition 3

Config/testthat/parallel false

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

Encoding UTF-8

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

RoxygenNote 7.3.2.9000

Config/Needs/website tidyverse/tidytemplate

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

RemoteUrl https://github.com/duckdblabs/duckplyr

RemoteRef HEAD

RemoteSha 129ff92bf2400b2c634ef770d8a0130e846af9d8

Contents

`as_duckplyr_df` *Convert to a duckplyr data frame*

Description

These functions convert a data-frame-like input to an object of class `"duckpylr_df"`. For such objects, dplyr verbs such as `mutate()`, `select()` or `filter()` will attempt to use DuckDB. If this is not possible, the original dplyr implementation is used.

`as_duckplyr_df()` requires the input to be a plain data frame or a tibble, and will fail for any other classes, including subclasses of `"data.frame"` or `"tbl_df"`. This behavior is likely to change, do not rely on it.

`as_duckplyr_tibble()` converts the input to a tibble and then to a duckplyr data frame.

Usage

```
as_duckplyr_df(.data)

as_duckplyr_tibble(.data)
```

Arguments

`.data`	data frame or tibble to transform

Details

Set the `DUCKPLYR_FALLBACK_INFO` and `DUCKPLYR_FORCE` environment variables for more control over the behavior, see config for more details.

Value

For `as_duckplyr_df()`, an object of class `"duckplyr_df"`, inheriting from the classes of the `.data` argument.

For `as_duckplyr_df()`, an object of class `c("duckplyr_df", class(tibble()))` .

Examples

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

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

`config` *Configuration options*

Description

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

Options

`duckdb.materialize_message`: Set to `FALSE` to turn off diagnostic output from duckdb on data frame materialization. Currenty set to `TRUE` when duckplyr is loaded.

Environment variables

`DUCKPLYR_OUTPUT_ORDER`: If `TRUE`, row output order is preserved. The default may change the row order where dplyr would keep it stable.

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

`DUCKPLYR_FALLBACK_INFO`: If `TRUE`, print a message when a fallback to dplyr occurs because duckdb cannot handle a request.

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

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

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

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

Examples

```
# options(duckdb.materialize_message = FALSE)
data.frame(a = 3:1) %>%
  as_duckplyr_df() %>%
  inner_join(data.frame(a = 1:4), by = "a")

rlang::with_options(duckdb.materialize_message = FALSE, {
  data.frame(a = 3:1) %>%
    as_duckplyr_df() %>%
    inner_join(data.frame(a = 1:4), by = "a") %>%
    print()
})

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

withr::with_envvar(c(DUCKPLYR_OUTPUT_ORDER = "TRUE"), {
  data.frame(a = 3:1) %>%
    as_duckplyr_df() %>%
    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_duckplyr_df() %>%
  mutate(b = add_one(a))

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

`df_from_file` *Read Parquet, CSV, and other files using DuckDB*

Description

df_from_file() uses arbitrary table functions 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,

duckplyr_df_from_file() is a thin wrapper around df_from_file() that calls as_duckplyr_df() on the output.

These functions ingest data from a file using a table function. The results are transparently converted to a data frame, but the data is only read when the resulting data frame is actually accessed.

df_from_csv() reads a CSV file using the read_csv_auto() table function.

duckplyr_df_from_csv() is a thin wrapper around df_from_csv() that calls as_duckplyr_df() on the output.

df_from_parquet() reads a Parquet file using the read_parquet() table function.

duckplyr_df_from_parquet() is a thin wrapper around df_from_parquet() that calls as_duckplyr_df() on the output.

df_to_parquet() writes a data frame to a Parquet file via DuckDB. If the data frame is a duckplyr_df, the materialization occurs outside of R. An existing file will be overwritten. This function requires duckdb >= 0.10.0.

Usage

```
df_from_file(path, table_function, ..., options = list(), class = NULL)

duckplyr_df_from_file(
  path,
  table_function,
  ...,
  options = list(),
  class = NULL
)

df_from_csv(path, ..., options = list(), class = NULL)

duckplyr_df_from_csv(path, ..., options = list(), class = NULL)

df_from_parquet(path, ..., options = list(), class = NULL)

duckplyr_df_from_parquet(path, ..., options = list(), class = NULL)

df_to_parquet(data, path)
```

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.

`options`	Arguments to the DuckDB function indicated by `table_function`.
`class`	The class of the output. By default, a tibble is created. The returned object will always be a data frame. Use `class = "data.frame"` or `class = character()` to create a plain data frame.
`data`	A data frame to be written to disk.

Value

A data frame for `df_from_file()`, or a `duckplyr_df` for `duckplyr_df_from_file()`, extended by the provided `class`.

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 <- df_from_csv(path)

# Materialization only upon access
names(df)
df$a

# Return as tibble, specify column types:
df_from_file(
  path,
  "read_csv",
  options = list(delim = ",", types = list(c("DOUBLE", "VARCHAR"))),
  class = class(tibble())
)

# Read multiple file at once
path2 <- tempfile("duckplyr_test_", fileext = ".csv")
write.csv(data.frame(a = 4:6, b = letters[7:9]), path2, row.names = FALSE)

duckplyr_df_from_csv(file.path(tempdir(), "duckplyr_test_*.csv"))

unlink(c(path, path2))

# Write a Parquet file:
path_parquet <- tempfile(fileext = ".parquet")
df_to_parquet(df, path_parquet)

# With a duckplyr_df, the materialization occurs outside of R:
df %>%
  as_duckplyr_df() %>%
  mutate(b = a + 1) %>%
  df_to_parquet(path_parquet)

duckplyr_df_from_parquet(path_parquet)
```

```
unlink(path_parquet)
```

`duckplyr_execute` *Execute a statement for the default connection*

Description

The **duckplyr** package relies on a DBI connection to an in-memory database. The `duckplyr_execute()` function allows running SQL statements with this connection to, e.g., set up credentials or attach other databases.

Usage

```
duckplyr_execute(sql)
```

Arguments

`sql`	The statement to run.

Value

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

Examples

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

`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 duckplyr encounters an incompatibility, it falls back to dplyr.

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 logging and uploading, the number of reports ready for upload, and the location of the logs.

`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_review(oldest = NULL, newest = NULL, detail = TRUE)

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

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

Arguments

`oldest, newest`	The number of oldest or newest reports to review. If not specified, all reports are dispayed.
`detail`	Print the full content of the reports. Set to `FALSE` to only print the file names.
`strict`	If `TRUE`, the function aborts if any of the reports fail to upload. With `FALSE`, only a message is printed.

Details

Logging and uploading are both opt-in. By default, for logging, a message is printed to the console for the first time in a session and then once every 8 hours.

The following environment variables control the logging and uploading:

- `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_VERBOSE` controls printing, 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.
- `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")`.

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

Examples

```
fallback_sitrep()
```

`is_duckplyr_df` *Class predicate for duckplyr data frames*

Description

Tests if the input object is of class `"duckplyr_df"`.

Usage

```
is_duckplyr_df(.data)
```

Arguments

`.data`	The object to test

Value

`TRUE` if the input object is of class `"duckplyr_df"`, otherwise `FALSE`.

Examples

```
tibble(a = 1:3) %>%
  is_duckplyr_df()

tibble(a = 1:3) %>%
  as_duckplyr_df() %>%
  is_duckplyr_df()
```

`methods_overwrite` *Forward all dplyr methods to duckplyr*

Description

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

Usage

```
methods_overwrite()

methods_restore()
```

Value

Called for their side effects.

Examples

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

methods_overwrite()

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

methods_restore()

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

`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 `new_relexpr()`.
`groups`	A list of expressions to group by.
`aggregates`	A list of expressions with aggregates to compute.
`orders`	A list of expressions to order by.
`ascending`	A logical vector describing the sort order.
`conds`	A list of expressions to use for the join.
`join`	The type of join.
`join_ref_type`	The ref type of join.
`n`	The number of rows.
`alias`	the new alias

Value

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

Examples

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

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

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

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

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

```
)

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

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

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

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

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

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

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

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

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


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

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

rel_limit(mtcars_rel, 3)

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

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

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

  names(df)
}

rel_names(mtcars_rel)
```

new_relexpr *Relational expressions*

Description

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

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

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

`relexpr_constant()` wraps a constant value.

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

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

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

Usage

```
new_relexpr(x, class = NULL)

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

relexpr_constant(val, alias = NULL)

relexpr_function(name, args, alias = NULL)

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

relexpr_set_alias(expr, alias = NULL)
```

Arguments

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

Value

an object of class `"relational_relexpr"`

an object of class `"relational_relexpr"`

an object of class `"relational_relexpr"`

an object of class `"relational_relexpr"`

an object of class `"relational_relexpr"`

Examples

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

`stats_show` *Show stats*

Description

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

Usage

```
stats_show()
```

Value

Called for its side effect.

Examples

```
stats_show()

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

stats_show()
```

Index