Package 'qryflow'

July 22, 2025

Title Execute Multi-Step 'SQL' Workflows

Version 0.1.0

Description Execute multi-step 'SQL' workflows by leveraging specially formatted comments to define and control execution. This enables users to mix queries, commands, and metadata within a single script. Results are returned as named objects for use in downstream workflows.

License MIT + file LICENSE

Encoding UTF-8

RoxygenNote 7.3.2

Imports DBI

Suggests knitr, rmarkdown, RSQLite, testthat (>= 3.0.0)

Config/testthat/edition 3

VignetteBuilder knitr

URL https://christian-million.github.io/qryflow/, https://github.com/christian-million/qryflow

BugReports https://github.com/christian-million/qryflow/issues

NeedsCompilation no

Author Christian Million [aut, cre, cph]

Maintainer Christian Million <christianmillion93@gmail.com>

Repository CRAN

Date/Publication 2025-07-18 15:20:18 UTC

Contents

`collapse_sql_lines` *Collapse SQL lines into single character*

Description

A thin wrapper around `paste0(x, collapse = '\\n')` to standardize the way qryflow collapses SQL lines.

Usage

```
collapse_sql_lines(x)
```

Arguments

`x`	character vector of SQL lines

Value

a character vector of length 1

Examples

```
path <- example_sql_path()

lines <- read_sql_lines(path)

sql <- collapse_sql_lines(lines)
```

example_db_connect *Create an example in-memory database*

Description

This function creates a connection to an in-memory SQLite database, with the option to add a table to the database. This function is intended to facilitate examples, vignettes, and package tests.

Usage

```
example_db_connect(df = NULL)
```

Arguments

df	Optional data.frame to add to the database.

Value

connection from `DBI::dbConnect()`

Examples

```
con <- example_db_connect(mtcars)

x <- DBI::dbGetQuery(con, "SELECT * FROM mtcars;")

head(x)

DBI::dbDisconnect(con)
```

example_sql_path *Get path to qryflow example SQL scripts*

Description

qryflow provides example SQL scripts in its `inst/sql` directory. Use this function to retrieve the path to an example script. This function is intended to facilitate examples, vignettes, and package tests.

Usage

```
example_sql_path(path = "mtcars.sql")
```

Arguments

path	filename of the example script.

Value

path to example SQL script

Examples

```
path <- example_sql_path("mtcars.sql")

file.exists(path)
```

`extract_all_tags` *Extract tagged metadata from a SQL chunk*

Description

`extract_all_tags()` scans SQL for specially formatted comment tags (e.g., `-- @tag: value`) and returns them as a named list. This is exported with the intent to be useful for users extending `qryflow`. It's typically used against a single SQL chunk, such as one parsed from a `.sql` file.

Additional helpers like `extract_tag()`, `extract_name()`, and `extract_type()` provide convenient access to specific tag values. `subset_tags()` lets you filter or exclude tags by name.

Usage

```
extract_all_tags(text, tag_pattern = "^\\s*--\\s*@([^:]+):\\s*(.*)$")

extract_tag(text, tag)

extract_name(text)

extract_type(text)

subset_tags(tags, keep, negate = FALSE)
```

Arguments

`text`	A character vector of SQL lines or a file path to a SQL script.
`tag_pattern`	A regular expression for extracting tags. Defaults to lines in the form `-- @tag: value`.
`tag`	A character string naming the tag to extract (used in `extract_tag()`).
`tags`	A named list of tags, typically from `extract_all_tags()`. Used in `subset_tags()`.
`keep`	A character vector of tag names to keep or exclude in `subset_tags()`.
`negate`	Logical; if `TRUE`, `subset_tags()` returns all tags except those listed in `keep`.

Details

The formal type of a qryflow SQL chunk is determined by `extract_type()` using a prioritized approach:

1. If the chunk includes an explicit `-- @type:` tag, its value is used directly as the chunk type.
2. If the `@type:` tag is absent, `qryflow` searches for other tags (e.g., `@query:`, `@exec:`) that correspond to registered chunk types through `ls_qryflow_types()`. The first matching tag found defines the chunk type.
3. If neither an explicit `@type:` tag nor any recognized tag is present, the chunk type falls back to the default type returned by `qryflow_default_type()`.

Value

- `extract_all_tags()`: A named list of all tags found in the SQL chunk.
- `extract_tag()`, `extract_name()`, `extract_type()`: A single tag value (character or `NULL`).
- `subset_tags()`: A filtered named list of tags or `NULL` if none remain.

See Also

`qryflow_parse()`, `ls_qryflow_types()`, `qryflow_default_type()`

Examples

```
filepath <- example_sql_path('mtcars.sql')
parsed <- qryflow_parse(filepath)

chunk <- parsed$chunks[[1]]
tags <- extract_all_tags(chunk$sql)

extract_name(chunk$sql)
extract_type(chunk$sql)
subset_tags(tags, keep = c("query"))
```

`is_tag_line` *Detect the presence of a properly structured tagline*

Description

Checks whether a specially structured comment line if formatted in the way that qryflow expects.

Usage

```
is_tag_line(line)
```

Arguments

`line`	A character vector to check. It is a vectorized function.

Details

Tag lines should look like this: `-- @key: value`

- Begins with an inline comment (`--`)
- An `@` precedes a tag type (e.g., `type`, `name`, `query`, `exec`) and is followed by a colon (`:`)
- A value is provided

Value

Logical. Indicating whether each line matches tag specification.

Examples

```
a <- "-- @query: df_mtcars"
b <- "-- @exec: prep_tbl"
c <- "-- @type: query"

lines <- c(a, b, c)

is_tag_line(lines)
```

`ls_qryflow_handlers` *List currently registered chunk types*

Description

Helper function to access the names of the currently registered chunk types. Functions available for accessing just the parsers or just the handlers.

Usage

```
ls_qryflow_handlers()

ls_qryflow_parsers()

ls_qryflow_types()
```

Details

`ls_qryflow_types` is implemented to return the union of the results of `ls_qryflow_parsers` and `ls_qryflow_handlers`. It's expected that a both a parser and a handler exist for each type. If this assumption is violated, the `ls_qryflow_types` may suggest otherwise.

Value

Character vector of registered chunk types

Examples

```
ls_qryflow_types()
```

`new_qryflow_chunk` *Create an instance of the* `qryflow_chunk` *class*

Description

Create an instance of the `qryflow_chunk` class

Usage

```
new_qryflow_chunk(
  type = character(),
  name = character(),
  sql = character(),
  tags = NULL,
  results = NULL
)
```

Arguments

`type`	Character indicating the type of chunk (e.g., "query", "exec")
`name`	Name of the chunk
`sql`	SQL statement associated with chunk
`tags`	Optional, additional tags included in chunk
`results`	Optional, filled in after chunk execution

Details

Exported for users intending to extend qryflow. Subsequent processes rely on the structure of a qryflow_chunk.

Value

An list-like object of class `qryflow_chunk`

Examples

```
chunk <- new_qryflow_chunk("query", "df_name", "SELECT * FROM mtcars;")
```

qryflow *Run a multi-step SQL workflow and return query results*

Description

`qryflow()` is the main entry point to the `qryflow` package. It executes a SQL workflow defined in a tagged `.sql` script or character string and returns query results as R objects.

The SQL script can contain multiple steps tagged with `@query` or `@exec`. Query results are captured and returned as a named list, where names correspond to the `@query` tags.

Usage

```
qryflow(sql, con, ..., simplify = TRUE)
```

Arguments

`sql`	A file path to a `.sql` workflow or a character string containing SQL code.
`con`	A database connection from `DBI::dbConnect()`
`...`	Additional arguments passed to `qryflow_run()` or `qryflow_results()`.
`simplify`	Logical; if `TRUE` (default), a list of length 1 is simplified to the single result object.

Details

This is a wrapper around the combination of `qryflow_run()`, which always provides a list of results and metadata, and `qryflow_results()`, which filters the output of `qryflow_run()` to only include the results of the SQL.

Value

A named list of query results, or a single result if `simplify = TRUE` and only one chunk exists.

See Also

`qryflow_run()`, `qryflow_results()`

Examples

```
con <- example_db_connect(mtcars)

filepath <- example_sql_path("mtcars.sql")

results <- qryflow(filepath, con)

head(results$df_mtcars)

DBI::dbDisconnect(con)
```

qryflow_default_type *Access the default qryflow chunk type*

Description

Retrieves the value from the option `qryflow.default.type`, if set. Otherwise returns "query", which is the officially supported default type. If any value is supplied to the function, it returns that value.

Usage

```
qryflow_default_type(type = getOption("qryflow.default.type", "query"))
```

Arguments

`type`	Optional. The type you want to return.

Value

Character. If set, result from `qryflow.default.type` option, otherwise "query" or value passed to `type`

Examples

```
x <- getOption("qryflow.default.type", "query")

y <- qryflow_default_type()

identical(x, y)
```

qryflow_execute *Execute a parsed qryflow SQL workflow*

Description

`qryflow_execute()` takes a parsed workflow object (as returned by `qryflow_parse()`), executes each chunk (e.g., `@query`, `@exec`), and collects the results and timing metadata.

This function is used internally by `qryflow_run()`, but can be called directly in concert with `qryflow_parse()` if you want to manually control parsing and execution.

Usage

```
qryflow_execute(x, con, ..., source = NULL)
```

Arguments

x	A parsed qryflow workflow object, typically created by qryflow_parse()
con	A database connection from DBI::dbConnect()
...	Reserved for future use.
source	Optional; a character string indicating the source SQL to include in metadata.

Value

An object of class qryflow_result, containing executed chunks with results and a meta field that includes timing and source information.

See Also

qryflow_run(), qryflow_parse()

Examples

```
con <- example_db_connect(mtcars)

filepath <- example_sql_path("mtcars.sql")

parsed <- qryflow_parse(filepath)

executed <- qryflow_execute(parsed, con, source = filepath)

DBI::dbDisconnect(con)
```

---

qryflow_handler_exists

Check existence of a given handler in the registry

Description

Checks whether the specified handler exists in the handler registry environment.

Usage

```
qryflow_handler_exists(type)
```

Arguments

type	chunk type to check (e.g., "query", "exec")

Value

Logical. Does type exist in the handler registry?

See Also

qryflow_parser_exists() for the parser equivalent.

Examples

```
qryflow_handler_exists("query")
```

---

qryflow_parse *Parse a SQL workflow into tagged chunks*

Description

qryflow_parse() reads a SQL workflow file or character vector and parses it into discrete tagged chunks based on @query, @exec, and other custom markers.

Usage

```
qryflow_parse(sql)
```

Arguments

sql	A file path to a SQL workflow file, or a character vector containing SQL lines.

Details

This function is used internally by qryflow_run(), but can also be used directly to preprocess or inspect the structure of a SQL workflow.

Value

An object of class qryflow_workflow, which is a structured list of SQL chunks and metadata.

See Also

qryflow(), qryflow_run(), qryflow_execute()

Examples

```
filepath <- example_sql_path("mtcars.sql")

parsed <- qryflow_parse(filepath)
```

qryflow_parser_exists *Check existence of a given parser in the registry*

Description

Checks whether the specified parser exists in the parser registry environment.

Usage

```
qryflow_parser_exists(type)
```

Arguments

type	chunk type to check (e.g., "query", "exec")

Value

Logical. Does type exist in the parser registry?

See Also

qryflow_handler_exists() for the handler equivalent.

Examples

```
qryflow_parser_exists("query")
```

qryflow_results *Extract results from a* qryflow_workflow *object*

Description

qryflow_results() retrieves the query results from a list returned by qryflow_run(), typically one that includes parsed and executed SQL chunks.

Usage

```
qryflow_results(x, ..., simplify = FALSE)
```

Arguments

x	Results from qryflow_run(), usually containing a mixture of qryflow_chunk objects.
...	Reserved for future use.
simplify	Logical; if TRUE, simplifies the result to a single object if only one query chunk is present. Defaults to FALSE.

Value

A named list of query results, or a single result object if `simplify = TRUE` and only one result is present.

See Also

`qryflow()`, `qryflow_run()`

Examples

```
con <- example_db_connect(mtcars)

filepath <- example_sql_path("mtcars.sql")

obj <- qryflow_run(filepath, con)

results <- qryflow_results(obj)

DBI::dbDisconnect(con)
```

---

qryflow_run *Parse and execute a tagged SQL workflow*

Description

`qryflow_run()` reads a SQL workflow from a file path or character string, parses it into tagged statements, and executes those statements against a database connection.

This function is typically used internally by `qryflow()`, but can also be called directly for more control over workflow execution.

Usage

```
qryflow_run(sql, con, ...)
```

Arguments

`sql`	A character string representing either the path to a `.sql` file or raw SQL content.
`con`	A database connection from `DBI::dbConnect()`
`...`	Additional arguments passed to `qryflow_execute()`.

Value

A list representing the evaluated workflow, containing query results, execution metadata, or both, depending on the contents of the SQL script.

See Also

`qryflow()`, `qryflow_results()`, `qryflow_execute()`, `qryflow_parse()`

Examples

```
con <- example_db_connect(mtcars)

filepath <- example_sql_path("mtcars.sql")

obj <- qryflow_run(filepath, con)

obj$df_mtcars$sql
obj$df_mtcars$results

results <- qryflow_results(obj)

head(results$df_mtcars$results)

DBI::dbDisconnect(con)
```

`read_sql_lines` *Standardizes lines read from string, character vector, or file*

Description

This is a generic function to ensure lines read from a file, a single character vector, or already parsed lines return the same format. This helps avoid re-reading entire texts by enabling already read lines to pass easily.

This is useful for folks who may want to extend qryflow.

Usage

```
read_sql_lines(x)
```

Arguments

x	a filepath or character vector containing SQL

Value

A `qryflow_sql` object (inherits from character) with a length equal to the number of lines read

Examples

```
# From a file #####
path <- example_sql_path()
read_sql_lines(path)

# From a single string #####
sql <- "SELECT *
FROM mtcars;"
read_sql_lines(sql)
```

```
# From a character #####
lines <- c("SELECT *", "FROM mtcars;")
read_sql_lines(lines)
```

`register_qryflow_type` *Register custom chunk types*

Description

Use these functions to register the parsers and handlers associated with custom types. `register_qryflow_type` is a wrapper around both `register_qryflow_parser` and `register_qryflow_handler`.

Usage

```
register_qryflow_type(type, parser, handler, overwrite = FALSE)

register_qryflow_parser(type, parser, overwrite = FALSE)

register_qryflow_handler(type, handler, overwrite = FALSE)
```

Arguments

`type`	Character indicating the chunk type (e.g., "exec", "query")
`parser`	A function to parse the SQL associated with the type. Must accept arguments "x" and "..." and return a `qryflow_chunk` object.
`handler`	A function to execute the SQL associated with the type. Must accept arguments "chunk", "con", and "...".
`overwrite`	Logical. Overwrite existing parser and handler, if exists?

Details

To avoid manually registering your custom type each session, consider adding the registration code to your `.Rprofile` or creating a package that leverages `.onLoad()`

Value

Logical. Indicating whether types were successfully registered.

Examples

```
# Create custom parser #####
custom_parser <- function(x, ...){
  # Custom parsing code will go here

  # new_qryflow_chunk(type = "custom", name = name, sql = sql_txt, tags = tags)
}

# Create custom handler #####
```

```
custom_handler <- function(chunk, con, ...){
  # Custom execution code will go here...
  # return(result)
}

# Register Separately #####
register_qryflow_parser("custom", custom_parser, overwrite = TRUE)

register_qryflow_handler("custom", custom_handler, overwrite = TRUE)


# Register Simultaneously #####
register_qryflow_type("query-send", custom_parser, custom_handler, overwrite = TRUE)
```

`validate_qryflow_handler` *Ensure correct handler structure*

Description

This function checks that the passed object is a function and contains the arguments "chunk", "con, and "..." - in that order. This is to help ensure users only register valid handlers.

Usage

```
validate_qryflow_handler(handler)
```

Arguments

`handler`	object to check

Value

Logical. Generates an error if the object does not pass all the criteria.

See Also

`validate_qryflow_parser()` for the parser equivalent.

Examples

```
custom_func <- function(chunk, con, ...){

  # Parsing Code Goes Here

}

validate_qryflow_handler(custom_func)
```

validate_qryflow_parser *Ensure correct parser structure*

Description

This function checks that the passed object is a function and contains the arguments "x" and "..." - in that order. This is to help ensure users only register valid parsers.

Usage

```
validate_qryflow_parser(parser)
```

Arguments

parser	object to check

Value

Logical. Generates an error if the object does not pass all the criteria.

See Also

validate_qryflow_handler() for the handler equivalent.

Examples

```
custom_func <- function(x, ...){

  # Parsing Code Goes Here

}
validate_qryflow_parser(custom_func)
```

Index