Package 'querychat'

June 2, 2026

Title Filter and Query Data Frames in 'shiny' Using an LLM Chat Interface

Version 0.3.0

Description Adds an LLM-powered chatbot to your 'shiny' app, that can turn your users' natural language questions into 'SQL' queries that run against your data, and return the result as a reactive data frame. Use it to drive reactive calculations, visualizations, downloads, and more.

License MIT + file LICENSE

URL https://posit-dev.github.io/querychat/r/, https://posit-dev.github.io/querychat/, https://github.com/posit-dev/querychat

BugReports https://github.com/posit-dev/querychat/issues

Depends R (>= 4.1.0)

Imports bsicons, bslib (>= 0.11.0), cli, DBI, ellmer (>= 0.4.1), htmltools, lifecycle, promises, R6, rlang (>= 1.1.0), S7, shiny, shinychat (>= 0.4.0), utils, whisker

Suggests dbplyr, dplyr, DT, duckdb, ggsql, knitr, palmerpenguins, rmarkdown, RSQLite, rsvg, shinytest2, testthat (>= 3.0.0), V8, withr

VignetteBuilder knitr

Config/roxygen2/version 8.0.0

Config/testthat/edition 3

Config/testthat/parallel true

Encoding UTF-8

NeedsCompilation no

Author Garrick Aden-Buie [aut, cre] (ORCID: <https://orcid.org/0000-0002-7111-0077>), Joe Cheng [aut, ccp], Carson Sievert [aut] (ORCID: <https://orcid.org/0000-0002-4958-2844>), Posit Software, PBC [cph, fnd]

Maintainer Garrick Aden-Buie <garrick@posit.co>

Repository CRAN

Date/Publication 2026-06-01 22:00:02 UTC

Contents

`DataFrameSource` *Data Frame Source*

Description

A DataSource implementation that wraps a data frame using DuckDB or SQLite for SQL query execution.

Details

This class creates an in-memory database connection and registers the provided data frame as a table. All SQL queries are executed against this database table. See DBISource for the full description of available methods.

By default, DataFrameSource uses the first available engine from duckdb (checked first) or RSQLite. You can explicitly set the `engine` parameter to choose between `"duckdb"` or `"sqlite"`, or set the global option `querychat.DataFrameSource.engine` to choose the default engine for all DataFrameSource instances. At least one of these packages must be installed.

Super classes

`DataSource` -> `DBISource` -> `DataFrameSource`

Methods

Public methods:

- `DataFrameSource$new()`
- `DataFrameSource$clone()`

`DataFrameSource$new()`: Create a new DataFrameSource

Usage:

```
DataFrameSource$new(
  df,
  table_name,
  engine = getOption("querychat.DataFrameSource.engine", NULL)
)
```

Arguments:

`df` A data frame.

`table_name` Name to use for the table in SQL queries. Must be a valid table name (start with letter, contain only letters, numbers, and underscores)

`engine` Database engine to use: "duckdb" or "sqlite". Set the global option `querychat.DataFrameSource.engine` to specify the default engine for all instances. If NULL (default), uses the first available engine from duckdb or RSQLite (in that order).

Returns: A new DataFrameSource object

`DataFrameSource$clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
DataFrameSource$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# Create a data frame source (uses first available: duckdb or sqlite)
df_source <- DataFrameSource$new(mtcars, "mtcars")

# Get database type
df_source$get_db_type()  # Returns "DuckDB" or "SQLite"

# Execute a query
result <- df_source$execute_query("SELECT * FROM mtcars WHERE mpg > 25")

# Explicitly choose an engine
df_sqlite <- DataFrameSource$new(mtcars, "mtcars", engine = "sqlite")

# Clean up when done
df_source$cleanup()
df_sqlite$cleanup()
```

`DataSource` *Data Source Base Class*

Description

An abstract R6 class defining the interface that custom QueryChat data sources must implement. This class should not be instantiated directly; instead, use one of its concrete implementations like DataFrameSource or DBISource.

Public fields

`table_name` Name of the table to be used in SQL queries

Methods

Public methods:

- `DataSource$get_db_type()`
- `DataSource$get_schema()`
- `DataSource$execute_query()`
- `DataSource$test_query()`
- `DataSource$get_data()`
- `DataSource$cleanup()`
- `DataSource$clone()`

`DataSource$get_db_type()`: Get the database type

Usage:

```
DataSource$get_db_type()
```

Returns: A string describing the database type (e.g., "DuckDB", "SQLite")

`DataSource$get_schema()`: Get schema information about the table

Usage:

```
DataSource$get_schema(categorical_threshold = 20)
```

Arguments:

`categorical_threshold` Maximum number of unique values for a text column to be considered categorical

Returns: A string containing schema information formatted for LLM prompts

`DataSource$execute_query()`: Execute a SQL query and return results

Usage:

```
DataSource$execute_query(query)
```

Arguments:

`query` SQL query string to execute

Returns: A data frame containing query results

`DataSource$test_query()`: Test a SQL query by fetching only one row

Usage:

```
DataSource$test_query(query, require_all_columns = FALSE)
```

Arguments:

`query` SQL query string to test

`require_all_columns` If `TRUE`, validates that the result includes all original table columns (default: `FALSE`)

Returns: A data frame containing one row of results (or empty if no matches)

`DataSource$get_data()`: Get the unfiltered data as a data frame

Usage:

```
DataSource$get_data()
```

Returns: A data frame containing all data from the table

`DataSource$cleanup()`: Clean up resources (close connections, etc.)

Usage:

```
DataSource$cleanup()
```

Returns: NULL (invisibly)

`DataSource$clone()`: The objects of this class are cloneable with this method.

Usage:

```
DataSource$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

Examples

```
MyDataSource <- R6::R6Class(
  "MyDataSource",
  inherit = DataSource,
  public = list(
    initialize = function(table_name) {
      self$table_name <- table_name
    }
    # Implement abstract methods here...
  )
)
```

`DBISource` *DBI Source*

Description

A DataSource implementation for DBI database connections (SQLite, PostgreSQL, MySQL, etc.). This class wraps a DBI connection and provides SQL query execution against a single table in the database.

Super class

`DataSource` -> `DBISource`

Methods

Public methods:

- DBISource$new()
- DBISource$get_db_type()
- DBISource$get_schema()
- DBISource$get_semantic_views_description()
- DBISource$execute_query()
- DBISource$test_query()
- DBISource$get_data()
- DBISource$cleanup()
- DBISource$clone()

`DBISource$new()`: Create a new DBISource

Usage:

```
DBISource$new(conn, table_name)
```

Arguments:

`conn` A DBI connection object

`table_name` Name of the table in the database. Can be a character string or a `DBI::Id()` object for tables in catalogs/schemas

Returns: A new DBISource object

`DBISource$get_db_type()`: Get the database type

Usage:

```
DBISource$get_db_type()
```

Returns: A string identifying the database type

`DBISource$get_schema()`: Get schema information for the database table

Usage:

```
DBISource$get_schema(categorical_threshold = 20)
```

Arguments:

`categorical_threshold` Maximum number of unique values for a text column to be considered categorical (default: 20)

Returns: A string describing the schema

`DBISource$get_semantic_views_description()`: Get information about semantic views (if any) for the system prompt.

Usage:

```
DBISource$get_semantic_views_description()
```

Returns: A string with semantic view information, or empty string if none

`DBISource$execute_query()`: Execute a SQL query

Usage:

`DBISource$execute_query(query)`

Arguments:

`query` SQL query string. If NULL or empty, returns all data

Returns: A data frame with query results

`DBISource$test_query()`: Test a SQL query by fetching only one row

Usage:

```
DBISource$test_query(query, require_all_columns = FALSE)
```

Arguments:

`query` SQL query string

`require_all_columns` If `TRUE`, validates that the result includes all original table columns (default: `FALSE`)

Returns: A data frame with one row of results

`DBISource$get_data()`: Get all data from the table

Usage:

```
DBISource$get_data()
```

Returns: A data frame containing all data

`DBISource$cleanup()`: Disconnect from the database

Usage:

```
DBISource$cleanup()
```

Returns: NULL (invisibly)

`DBISource$clone()`: The objects of this class are cloneable with this method.

Usage:

```
DBISource$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# Connect to a database
con <- DBI::dbConnect(RSQLite::SQLite(), ":memory:")
DBI::dbWriteTable(con, "mtcars", mtcars)

# Create a DBI source
db_source <- DBISource$new(con, "mtcars")

# Get database type
db_source$get_db_type()  # Returns "SQLite"

# Execute a query
result <- db_source$execute_query("SELECT * FROM mtcars WHERE mpg > 25")

# Note: cleanup() will disconnect the connection
# If you want to keep the connection open, don't call cleanup()
db_source$cleanup()
```

QueryChat *QueryChat: Interactive Data Querying with Natural Language*

Description

QueryChat is an R6 class built on Shiny, shinychat, and ellmer to enable interactive querying of data using natural language. It leverages large language models (LLMs) to translate user questions into SQL queries, execute them against a data source (data frame or database), and various ways of accessing/displaying the results.

The QueryChat class takes your data (a data frame or database connection) as input and provides methods to:

- Generate a chat UI for natural language queries (e.g., $app(), $sidebar())
- Initialize server logic that returns session-specific reactive values (via $server())
- Access reactive data, SQL queries, and titles through the returned server values

Usage in Shiny Apps

```
library(querychat)

# Create a QueryChat object
qc <- QueryChat$new(mtcars)

# Quick start: run a complete app
qc$app()

# Or build a custom Shiny app
ui <- page_sidebar(
  qc$sidebar(),
  verbatimTextOutput("sql"),
  dataTableOutput("data")
)

server <- function(input, output, session) {
  qc_vals <- qc$server()

  output$sql <- renderText(qc_vals$sql())
  output$data <- renderDataTable(qc_vals$df())
}

shinyApp(ui, server)
```

Public fields

greeting The greeting message displayed to users.

id ID for the QueryChat instance.

tools The allowed tools for the chat client.

Active bindings

`system_prompt` Get the system prompt.

`data_source` Get or set the current data source. When setting, the value is normalized and the system prompt is rebuilt.

Methods

Public methods:

`QueryChat$new()`: Create a new QueryChat object.

Usage:

```
QueryChat$new(
  data_source,
  table_name = missing_arg(),
  ...,
  id = NULL,
  greeting = NULL,
  client = NULL,
  tools = c("filter", "query"),
  data_description = NULL,
  categorical_threshold = 20,
  extra_instructions = NULL,
  prompt_template = NULL,
  cleanup = NA
)
```

Arguments:

`data_source` Either a data.frame, a database connection (e.g., DBI connection), or `NULL` to defer setting the data source until later. When `NULL`, the data source must be set via the `$data_source` property or passed to `$server()` before calling methods that require data access.

`table_name` A string specifying the table name to use in SQL queries. If `data_source` is a data.frame, this is the name to refer to it by in queries (typically the variable name). If not provided, will be inferred from the variable name for data.frame inputs. For database connections or `NULL` data sources, this parameter is required.

`...` Additional arguments (currently unused).

`id` Optional module ID for the QueryChat instance. If not provided, will be auto-generated from `table_name`. The ID is used to namespace the Shiny module.

`greeting` Optional initial message to display to users. Can be a character string (in Markdown format) or a file path. If not provided, a greeting will be generated at the start of each conversation using the LLM, which adds latency and cost. Use `$generate_greeting()` to create a greeting to save and reuse.

`client` Optional chat client. Can be:

- An ellmer::Chat object
- A string to pass to `ellmer::chat()` (e.g., `"openai/gpt-4o"`)
- `NULL` (default): Uses the `querychat.client` option, the `QUERYCHAT_CLIENT` environment variable, or defaults to `ellmer::chat_openai()`

`tools` Which querychat tools to include in the chat client, by default. `"filter"` includes the tools for filtering and resetting the dashboard and `"query"` includes the tool for executing SQL queries. Use `tools = "filter"` when you only want the dashboard filtering tools, or when you want to disable the querying tool entirely to prevent the LLM from seeing any of the data in your dataset. The legacy name `"update"` is still accepted as an alias for `"filter"`.

`data_description` Optional description of the data in plain text or Markdown. Can be a string or a file path. This provides context to the LLM about what the data represents.

`categorical_threshold` For text columns, the maximum number of unique values to consider as a categorical variable. Default is 20.

`extra_instructions` Optional additional instructions for the chat model in plain text or Markdown. Can be a string or a file path.

`prompt_template` Optional path to or string of a custom prompt template file. If not provided, the default querychat template will be used. See the package prompts directory for the default template format.

`cleanup` Whether or not to automatically run `$cleanup()` when the Shiny session/app stops. By default, cleanup only occurs if `QueryChat` gets created within a Shiny session. Set to `TRUE` to always clean up, or `FALSE` to never clean up automatically.

Returns: A new `QueryChat` object.

`QueryChat$client()`**:** Create a chat client, complete with registered tools, for the current data source.

Usage:

```
QueryChat$client(
  tools = NA,
  update_dashboard = function(query, title) {
 },
  reset_dashboard = function() {
 },
  visualize = function(data) {
 },
  session = NULL
)
```

Arguments:

`tools` Which querychat tools to include in the chat client. `"filter"` includes the tools for filtering and resetting the dashboard and `"query"` includes the tool for executing SQL queries. By default, when `tools = NA`, the values provided at initialization are used. The legacy name `"update"` is still accepted as an alias for `"filter"`.

`update_dashboard` Optional function to call with the `query` and `title` generated by the LLM for the `update_dashboard` tool.

`reset_dashboard` Optional function to call when the `reset_dashboard` tool is called.

`visualize` Optional function to call with a list containing `ggsql`, `title`, and `widget_id` when a visualization succeeds.

`session` A Shiny session object. Required when `"visualize"` is in `tools` and you want interactive chart rendering. When `NULL` (the default), visualizations still execute but are not rendered as Shiny outputs.

`QueryChat$console()`: Launch a console-based chat interface with the data source.

Usage:

```
QueryChat$console(new = FALSE, ..., tools = "query")
```

Arguments:

`new` Whether to create a new chat client instance or continue the conversation from the last console chat session (the default).

`...` Additional arguments passed to the `$client()` method.

`tools` Which querychat tools to include in the chat client. See `$client()` for details. Ignored when not creating a new chat client. By default, only the `"query"` tool is included, regardless of the `tools` set at initialization.

`QueryChat$app()`: Create and run a Shiny gadget for chatting with data

Runs a Shiny gadget (designed for interactive use) that provides a complete interface for chatting with your data using natural language. If you're looking to deploy this app or run it through some other means, see `$app_obj()`.

```
library(querychat)

qc <- QueryChat$new(mtcars)
qc$app()
```

Usage:

```
QueryChat$app(..., bookmark_store = "url")
```

Arguments:

`...` Arguments passed to `$app_obj()`.

`bookmark_store` The bookmarking storage method. Passed to `shiny::enableBookmarking()`. If `"url"` or `"server"`, the chat state (including current query) will be bookmarked. Default is `"url"`.

Returns: Invisibly returns a list of session-specific values:

- `df`: The final filtered data frame
- `sql`: The final SQL query string
- `title`: The final title
- `client`: The session-specific chat client instance

`QueryChat$app_obj()`: A streamlined Shiny app for chatting with data

Creates a Shiny app designed for chatting with data, with:

- A sidebar containing the chat interface
- A card displaying the current SQL query
- A card displaying the filtered data table
- A reset button to clear the query

```
library(querychat)

qc <- QueryChat$new(mtcars)
app <- qc$app_obj()
shiny::runApp(app)
```

Usage:

```
QueryChat$app_obj(..., bookmark_store = "url")
```

Arguments:

`...` Additional arguments (currently unused).

`bookmark_store` The bookmarking storage method. Passed to `shiny::enableBookmarking()`. If `"url"` or `"server"`, the chat state (including current query) will be bookmarked. Default is `"url"`.

Returns: A Shiny app object that can be run with `shiny::runApp()`.

`QueryChat$sidebar()`: Create a sidebar containing the querychat UI.

This method generates a `bslib::sidebar()` component containing the chat interface, suitable for use with `bslib::page_sidebar()` or similar layouts.

```
qc <- QueryChat$new(mtcars)

ui <- page_sidebar(
  qc$sidebar(),
  # Main content here
)
```

Usage:

```
QueryChat$sidebar(
  ...,
  width = 400,
  height = "100%",
  fillable = TRUE,
  id = NULL
)
```

Arguments:

`...` Additional arguments passed to `bslib::sidebar()`.

`width` Width of the sidebar in pixels. Default is 400.

`height` Height of the sidebar. Default is "100%".

`fillable` Whether the sidebar should be fillable. Default is `TRUE`.

id Optional ID for the QueryChat instance. If not provided, will use the ID provided at initialization. If using $sidebar() in a Shiny module, you'll need to provide id = ns("your_id") where ns is the namespacing function from shiny::NS().

Returns: A bslib::sidebar() UI component.

QueryChat$ui(): Create the UI for the querychat chat interface.

This method generates the chat UI component. Typically you'll use $sidebar() instead, which wraps this in a sidebar layout.

```
qc <- QueryChat$new(mtcars)

ui <- fluidPage(
  qc$ui()
)
```

Usage:

```
QueryChat$ui(..., id = NULL)
```

Arguments:

... Additional arguments passed to shinychat::chat_ui().

id Optional ID for the QueryChat instance. If not provided, will use the ID provided at initialization. If using $ui() in a Shiny module, you'll need to provide id = ns("your_id") where ns is the namespacing function from shiny::NS().

Returns: A UI component containing the chat interface.

QueryChat$server(): Initialize the querychat server logic.

This method must be called within a Shiny server function. It sets up the reactive logic for the chat interface and returns session-specific reactive values.

```
qc <- QueryChat$new(mtcars)

server <- function(input, output, session) {
  qc_vals <- qc$server(enable_bookmarking = TRUE)

  output$data <- renderDataTable(qc_vals$df())
  output$query <- renderText(qc_vals$sql())
  output$title <- renderText(qc_vals$title() %||% "No Query")
}
```

Usage:

```
QueryChat$server(
  data_source = NULL,
  client = NULL,
  enable_bookmarking = FALSE,
  ...,
  id = NULL,
  session = shiny::getDefaultReactiveDomain()
)
```

Arguments:

`data_source` Optional data source to use. If provided, sets the data_source property before initializing server logic. This is useful for the deferred pattern where data_source is not known at initialization time (e.g., when the data source depends on session- specific authentication).

`client` Optional chat client override for this session. Can be an ellmer::Chat object or a string (e.g., `"openai/gpt-4o"`). If provided, overrides the client set at initialization for this session only — other sessions are unaffected. This is useful when the client must be created within a session scope (e.g., Posit Connect managed credentials).

`enable_bookmarking` Whether to enable bookmarking for the chat state. Default is `FALSE`. When enabled, the chat state (including current query, title, and chat history) will be saved and restored with Shiny bookmarks. This requires that the Shiny app has bookmarking enabled via `shiny::enableBookmarking()` or the `enableBookmarking` parameter of `shiny::shinyApp()`.

`...` Ignored.

`id` Optional module ID for the QueryChat instance. If not provided, will use the ID provided at initialization. When used in Shiny modules, this `id` should match the `id` used in the corresponding UI function (i.e., `qc$ui(id = ns("your_id"))` pairs with `qc$server(id = "your_id")`).

`session` The Shiny session object.

Returns: A list containing session-specific reactive values and the chat client with the following elements:

- `df`: Reactive expression returning the current filtered data frame
- `sql`: Reactive value for the current SQL query string
- `title`: Reactive value for the current title
- `client`: The session-specific chat client instance

`QueryChat$generate_greeting()`: Generate a welcome greeting for the chat.

By default, `QueryChat$new()` generates a greeting at the start of every new conversation, which is convenient for getting started and development, but also might add unnecessary latency and cost. Use this method to generate a greeting once and save it for reuse.

```
# Create QueryChat object
qc <- QueryChat$new(mtcars)

# Generate a greeting and save it
greeting <- qc$generate_greeting()
writeLines(greeting, "mtcars_greeting.md")

# Later, use the saved greeting
qc2 <- QueryChat$new(mtcars, greeting = "mtcars_greeting.md")
```

Usage:

```
QueryChat$generate_greeting(echo = c("none", "output"))
```

Arguments:

`echo` Whether to print the greeting to the console. Options are `"none"` (default, no output) or `"output"` (print to console).

Returns: The greeting string in Markdown format.

`QueryChat$cleanup()`: Clean up resources associated with the data source.

This method releases any resources (e.g., database connections) associated with the data source. Call this when you are done using the QueryChat object to avoid resource leaks.

Note: If `auto_cleanup` was set to `TRUE` in the constructor, this will be called automatically when the Shiny app stops.

Usage:

```
QueryChat$cleanup()
```

Returns: Invisibly returns `NULL`. Resources are cleaned up internally.

`QueryChat$clone()`: The objects of this class are cloneable with this method.

Usage:

```
QueryChat$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# Basic usage with a data frame
qc <- QueryChat$new(mtcars)
## Not run:
app <- qc$app()

## End(Not run)

# With a custom greeting
greeting <- "Welcome! Ask me about the mtcars dataset."
qc <- QueryChat$new(mtcars, greeting = greeting)

# With a specific LLM provider
qc <- QueryChat$new(mtcars, client = "anthropic/claude-sonnet-4-5")

# Generate a greeting for reuse (requires internet/API access)
## Not run:
qc <- QueryChat$new(mtcars)
greeting <- qc$generate_greeting(echo = "text")
# Save greeting for next time
writeLines(greeting, "mtcars_greeting.md")

## End(Not run)

# Or specify greeting and additional options at initialization
qc <- QueryChat$new(
  mtcars,
  greeting = "Welcome to the mtcars explorer!",
  client = "openai/gpt-4o",
  data_description = "Motor Trend car road tests dataset"
)
```

```
# Create a QueryChat object from a database connection
# 1. Set up the database connection
con <- DBI::dbConnect(RSQLite::SQLite(), ":memory:")

# 2. (For this demo) Create a table in the database
DBI::dbWriteTable(con, "mtcars", mtcars)

# 3. Pass the connection and table name to `QueryChat`
qc <- QueryChat$new(con, "mtcars")
```

---

querychat *QueryChat convenience functions*

Description

Convenience functions for wrapping QueryChat creation (i.e., `querychat()`) and app launching (i.e., `querychat_app()`).

Usage

```
querychat(
  data_source,
  table_name = missing_arg(),
  ...,
  id = NULL,
  greeting = NULL,
  client = NULL,
  tools = c("filter", "query"),
  data_description = NULL,
  categorical_threshold = 20,
  extra_instructions = NULL,
  prompt_template = NULL,
  cleanup = NA
)

querychat_app(
  data_source,
  table_name = missing_arg(),
  ...,
  id = NULL,
  greeting = NULL,
  client = NULL,
  tools = c("filter", "query"),
  data_description = NULL,
  categorical_threshold = 20,
  extra_instructions = NULL,
  prompt_template = NULL,
```

```
  cleanup = NA,
  bookmark_store = "url"
)
```

Arguments

`data_source`	Either a data.frame or a database connection (e.g., DBI connection).
`table_name`	A string specifying the table name to use in SQL queries. If `data_source` is a data.frame, this is the name to refer to it by in queries (typically the variable name). If not provided, will be inferred from the variable name for data.frame inputs. For database connections, this parameter is required.
`...`	Additional arguments (currently unused).
`id`	Optional module ID for the QueryChat instance. If not provided, will be auto-generated from `table_name`. The ID is used to namespace the Shiny module.
`greeting`	Optional initial message to display to users. Can be a character string (in Markdown format) or a file path. If not provided, a greeting will be generated at the start of each conversation using the LLM, which adds latency and cost. Use `$generate_greeting()` to create a greeting to save and reuse.
`client`	Optional chat client. Can be: • An ellmer::Chat object • A string to pass to `ellmer::chat()` (e.g., `"openai/gpt-4o"`) • `NULL` (default): Uses the `querychat.client` option, the `QUERYCHAT_CLIENT` environment variable, or defaults to `ellmer::chat_openai()`
`tools`	Which querychat tools to include in the chat client, by default. `"filter"` includes the tools for filtering and resetting the dashboard and `"query"` includes the tool for executing SQL queries. Use `tools = "filter"` when you only want the dashboard filtering tools, or when you want to disable the querying tool entirely to prevent the LLM from seeing any of the data in your dataset. The legacy name `"update"` is still accepted as an alias for `"filter"`.
`data_description`	Optional description of the data in plain text or Markdown. Can be a string or a file path. This provides context to the LLM about what the data represents.
`categorical_threshold`	For text columns, the maximum number of unique values to consider as a categorical variable. Default is 20.
`extra_instructions`	Optional additional instructions for the chat model in plain text or Markdown. Can be a string or a file path.
`prompt_template`	Optional path to or string of a custom prompt template file. If not provided, the default querychat template will be used. See the package prompts directory for the default template format.
`cleanup`	Whether or not to automatically run `$cleanup()` when the Shiny session/app stops. By default, cleanup only occurs if `QueryChat` is created within a Shiny app. Set to `TRUE` to always clean up, or `FALSE` to never clean up automatically.

In `querychat_app()`, in-memory databases created for data frames are always cleaned up.

`bookmark_store` The bookmarking storage method. Passed to `shiny::enableBookmarking()`. If `"url"` or `"server"`, the chat state (including current query) will be bookmarked. Default is `"url"`.

Value

A `QueryChat` object. See QueryChat for available methods.

Invisibly returns the chat object after the app stops.

Examples

```
# Quick start - chat with mtcars dataset in one line
querychat_app(mtcars)

# Add options
querychat_app(
  mtcars,
  greeting = "Welcome to the mtcars explorer!",
  client = "openai/gpt-4o"
)

# Chat with a database table (table_name required)
con <- DBI::dbConnect(RSQLite::SQLite(), ":memory:")
DBI::dbWriteTable(con, "mtcars", mtcars)
querychat_app(con, "mtcars")

# Create QueryChat class object
qc <- querychat(mtcars, greeting = "Welcome to the mtcars explorer!")

# Run the app later
qc$app()
```

`TblSqlSource` *Data Source: SQL Tibble*

Description

A DataSource implementation for lazy SQL tibbles connected to databases via `dbplyr::tbl_sql()` or `dplyr::sql()`.

Super classes

`DataSource` -> `DBISource` -> `TblSqlSource`

Public fields

`table_name` Name of the table to be used in SQL queries

Methods

Public methods:

- TblSqlSource$new()
- TblSqlSource$get_db_type()
- TblSqlSource$get_schema()
- TblSqlSource$execute_query()
- TblSqlSource$test_query()
- TblSqlSource$prep_query()
- TblSqlSource$get_data()
- TblSqlSource$cleanup()
- TblSqlSource$clone()

`TblSqlSource$new()`: Create a new TblSqlSource

Usage:

```
TblSqlSource$new(tbl, table_name = missing_arg())
```

Arguments:

`tbl` A dbplyr::tbl_sql() (or SQL tibble via dplyr::tbl()).

`table_name` Name of the table in the database. Can be a character string, or will be inferred from the `tbl` argument, if possible.

Returns: A new TblSqlSource object

`TblSqlSource$get_db_type()`: Get the database type

Usage:

```
TblSqlSource$get_db_type()
```

Returns: A string describing the database type (e.g., "DuckDB", "SQLite")

`TblSqlSource$get_schema()`: Get schema information about the table

Usage:

```
TblSqlSource$get_schema(categorical_threshold = 20)
```

Arguments:

`categorical_threshold` Maximum number of unique values for a text column to be considered categorical

Returns: A string containing schema information formatted for LLM prompts

`TblSqlSource$execute_query()`: Execute a SQL query and return results

Usage:

```
TblSqlSource$execute_query(query)
```

Arguments:

`query` SQL query string to execute

Returns: A data frame containing query results

`TblSqlSource$test_query()`: Test a SQL query by fetching only one row

Usage:

```
TblSqlSource$test_query(query, require_all_columns = FALSE)
```

Arguments:

`query` SQL query string to test

`require_all_columns` If `TRUE`, validates that the result includes all original table columns (default: `FALSE`)

Returns: A data frame containing one row of results (or empty if no matches)

`TblSqlSource$prep_query()`**:** Prepare a generic `SELECT * FROM ____` query to work with the SQL tibble

Usage:

```
TblSqlSource$prep_query(query)
```

Arguments:

`query` SQL query as a string

Returns: A complete SQL query string

`TblSqlSource$get_data()`**:** Get the unfiltered data as a SQL tibble

Usage:

```
TblSqlSource$get_data()
```

Returns: A `dbplyr::tbl_sql()` containing the original, unfiltered data

`TblSqlSource$cleanup()`**:** Clean up resources (close connections, etc.)

Usage:

```
TblSqlSource$cleanup()
```

Returns: NULL (invisibly)

`TblSqlSource$clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
TblSqlSource$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
con <- DBI::dbConnect(duckdb::duckdb())
DBI::dbWriteTable(con, "mtcars", mtcars)

mtcars_source <- TblSqlSource$new(dplyr::tbl(con, "mtcars"))
mtcars_source$get_db_type()  # "DuckDB"

result <- mtcars_source$execute_query("SELECT * FROM mtcars WHERE cyl > 4")

# Note, the result is not the *full* data frame, but a lazy SQL tibble
result
```

```
# You can chain this result into a dplyr pipeline
dplyr::count(result, cyl, gear)

# Or collect the entire data frame into local memory
dplyr::collect(result)

# Finally, clean up when done with the database (closes the DB connection)
mtcars_source$cleanup()
```

Index