# Package: duckdb (via r-universe)

July 28, 2026

**Title** DBI Package for the DuckDB Database Management System

**Version** 1.5.5

**Description** The DuckDB project is an embedded analytical data management system with support for the Structured Query Language (SQL). This package includes all of DuckDB and an R Database Interface (DBI) connector.

**License** MIT + file LICENSE

**URL** https://r.duckdb.org/, https://github.com/duckdb/duckdb-r

**BugReports** https://github.com/duckdb/duckdb-r/issues

**Depends** DBI, R (>= 4.2.0)

**Imports** methods, utils

**Suggests** adbcdrivermanager, arrow (>= 13.0.0), bit64, callr, clock, DBItest, dbplyr, dplyr, nanoarrow, rlang, sf, testthat (>= 3.0.0), tibble, vctrs, wk, withr

**Biarch** true

**Config/build/compilation-database** false

**Config/build/never-clean** true

**Config/comment/compilation-database** Generate manually with pkgload:::generate_db() for faster pkgload::load_all()

**Config/testthat/edition** 3

**Encoding** UTF-8

**Roxygen** list(markdown = TRUE)

**SystemRequirements** xz (for building from source)

**Config/roxygen2/version** 8.0.0.9000

**Config/pak/sysreqs** xz-utils

**Repository** https://test.r-universe.dev

**Date/Publication** 2026-07-24 08:07:54 UTC

**RemoteUrl** https://github.com/duckdb/duckdb-r

**RemoteRef** v1.5.5

**RemoteSha** 7930f8602ef8cd37cd4903f4c92aeef0edb29eab

# Contents

backend-duckdb . . . . . 2
default_conn . . . . . 3
duckdb . . . . . 4
duckdb_explain-class . . . . . 9
duckdb_read_csv . . . . . 9
duckdb_register . . . . . 11
duckdb_register_arrow . . . . . 12
duckdb_storage_status . . . . . 12
sql_query . . . . . 16

**Index** **17**

---

`backend-duckdb` *DuckDB SQL backend for dbplyr*

---

## Description

This is a SQL backend for dbplyr tailored to take into account DuckDB's possibilities. This mainly follows the backend for PostgreSQL, but contains more mapped functions.

`tbl_file()` is an experimental variant of `dplyr::tbl()` to directly access files on disk. It is safer than `dplyr::tbl()` because there is no risk of misinterpreting the request, and paths with special characters are supported.

`tbl_function()` is an experimental variant of `dplyr::tbl()` to create a lazy table from a table-generating function, useful for reading nonstandard CSV files or other data sources. It is safer than `dplyr::tbl()` because there is no risk of misinterpreting the query. See `https://duckdb.org/docs/data/overview` for details on data importing functions.

As an alternative, use `dplyr::tbl(src, dplyr::sql("SELECT ... FROM ..."))` for custom SQL queries.

`tbl_query()` is deprecated in favor of `tbl_function()`.

Use `simulate_duckdb()` with `lazy_frame()` to see simulated SQL without opening a DuckDB connection.

## Usage

```
tbl_file(src = NULL, path, ..., cache = FALSE)

tbl_function(src, query, ..., cache = FALSE)

tbl_query(src, query, ...)

simulate_duckdb(...)
```

## Arguments

| | |
|---|---|
| src | A duckdb connection object, default_conn() if omitted. |
| path | Path to existing Parquet, CSV or JSON file |
| ... | Any parameters to be forwarded |
| cache | Enable object cache for Parquet files |
| query | SQL code, omitting the FROM clause |

## Examples

```
library(dplyr, warn.conflicts = FALSE)
con <- DBI::dbConnect(duckdb(), path = ":memory:")

db <- copy_to(con, data.frame(a = 1:3, b = letters[2:4]))

db %>%
  filter(a > 1) %>%
  select(b)

path <- tempfile(fileext = ".csv")
write.csv(data.frame(a = 1:3, b = letters[2:4]))

db_csv <- tbl_file(con, path)
db_csv %>%
  summarize(sum_a = sum(a))

db_csv_fun <- tbl_function(con, paste0("read_csv_auto('", path, "')"))
db_csv %>%
  count()

DBI::dbDisconnect(con, shutdown = TRUE)
```

---

# default_conn *Get the default connection*

---

## Description

**[Experimental]**

default_conn() returns a default, built-in connection.

## Usage

```
default_conn()
```

## Details

Currently, the connection is established with `duckdb(environment_scan = TRUE)` and `dbConnect(timezone_out = "", array = "matrix")` so that data frames are automatically available as tables, timestamps are returned in the local timezone, and DuckDB's array type is returned as an R matrix. The details of how the connection is established are subject to change. In particular, returning the output as a tibble or other object may be supported in the future.

This connection is intended for interactive use. There is no way for this or other packages to comprehensively track the state of this connection, so scripts and packages should manage their own connections.

## Value

A DuckDB connection object

## Examples

```
conn <- default_conn()
sql_query("SELECT 42", conn = conn)
```

---

# duckdb *Connect to a DuckDB database instance*

---

## Description

`duckdb()` creates or reuses a database instance.

`duckdb_shutdown()` shuts down a database instance.

Return an `adbcdrivermanager::adbc_driver()` for use with Arrow Database Connectivity via the adbcdrivermanager package.

`dbConnect()` connects to a database instance.

`dbDisconnect()` closes a DuckDB database connection. The associated DuckDB database instance is shut down automatically, it is no longer necessary to set `shutdown = TRUE` or to call `duckdb_shutdown()`.

## Usage

```
duckdb(
  dbdir = DBDIR_MEMORY,
  read_only = FALSE,
  bigint = "numeric",
  config = list(),
  ...,
  home = NULL,
  shared_home = NULL,
  allow_extensions = NULL,
  environment_scan = FALSE
```

```
)

duckdb_shutdown(drv)

duckdb_adbc()

## S4 method for signature 'duckdb_driver'
dbConnect(
  drv,
  dbdir = DBDIR_MEMORY,
  ...,
  debug = getOption("duckdb.debug", FALSE),
  read_only = FALSE,
  timezone_out = "UTC",
  tz_out_convert = c("with", "force"),
  config = list(),
  bigint = "numeric",
  array = "none",
  geometry = "blob",
  map = "data.frame"
)

## S4 method for signature 'duckdb_connection'
dbDisconnect(conn, ..., shutdown = TRUE)
```

## Arguments

| | |
|---|---|
| dbdir | Location for database files. Should be a path to an existing directory in the file system. With the default (or `""`), all data is kept in RAM. |
| read_only | Set to `TRUE` for read-only operation. For file-based databases, this is only applied when the database file is opened for the first time. Subsequent connections (via the same `drv` object or a `drv` object pointing to the same path) will silently ignore this flag. |
| bigint | How 64-bit integers should be returned. There are two options: `"numeric"` and `"integer64"`. If `"numeric"` is selected, bigint integers will be treated as double/numeric. If `"integer64"` is selected, bigint integers will be set to bit64 encoding. |
| config | Named list with DuckDB configuration flags, see https://duckdb.org/docs/configuration/overview#configuration-reference for the possible options. These flags are only applied when the database object is instantiated. Subsequent connections will silently ignore these flags. |
| ... | These dots are for future extensions and must be empty. |
| home | Root directory for DuckDB's downloaded extensions and stored secrets. `NULL` (the default) resolves the location as described in duckdb_storage: an existing `~/.duckdb`, else a per-session temporary directory (with an offer to create `~/.duckdb` in interactive sessions). Pass a path to use it as the root explicitly, creating it if needed. Cannot be combined with `shared_home`. Applied only |

when the database instance is created; see the 'Database instances and driver reuse' section.

`shared_home` Opt in or out of the shared `~/.duckdb` location, overriding the automatic resolution. One of:

- `NULL` (the default) – resolve automatically (see duckdb_storage). This is the safe default.
- `TRUE` – store extensions and secrets under `~/.duckdb`, **creating that directory if it does not exist**. This is a good setting for permanent deployments (Posit Connect, Shiny, APIs). Do not use on CRAN or on other infrastructure where you don't own `~/.duckdb`.
  The setting is a durable, machine-level side effect that is *not* scoped to the current session: the directory persists after R exits, is reused by every future R session (and by the DuckDB CLI, Python and other clients that share `~/.duckdb`), and any secrets written there outlive this process. Applying this setting repeatedly is a fast no-op.
- `FALSE` – use a per-session temporary directory even if `~/.duckdb` already exists. Nothing persists beyond the session.

Cannot be combined with `home`. Applied only when the database instance is created; see the 'Database instances and driver reuse' section.

`allow_extensions`

**[Experimental]** Whether this driver may load DuckDB extensions (`INSTALL` / `LOAD`). One of:

- `NULL` (the default) – decide automatically. Extensions are enabled, except on an affected Linux build (one not compiled with `libstdc++`), where they are disabled and a throttled advisory message is shown. See the 'DuckDB extensions on Linux' section.
- `TRUE` – force-enable extensions, attempting to load them even on an affected build (which may crash R). No message.
- `FALSE` – disable extensions and silence the advisory message.

The argument takes precedence over the `duckdb.allow_extensions` option (a scalar logical) and the `DUCKDB_R_ALLOW_EXTENSIONS` environment variable (a value R reads as `TRUE` enables extensions and `FALSE` disables them; unset, empty, or any other value is undecided). Applied only when the database instance is created; see the 'Database instances and driver reuse' section.

`environment_scan`

Set to `TRUE` to treat data frames from the calling environment as tables. If a database table with the same name exists, it takes precedence. The default of this setting may change in a future version.

`drv` Object returned by `duckdb()`

`debug` Print additional debug information, such as queries.

`timezone_out` The time zone returned to R, defaults to `"UTC"`, which is currently the only timezone supported by duckdb. If you want to display datetime values in the local timezone, set to `Sys.timezone()` or `""`.

`tz_out_convert` How to convert timestamp columns to the timezone specified in `timezone_out`. There are two options: `"with"`, and `"force"`. If `"with"` is chosen, the timestamp will be returned as it would appear in the specified time zone. If `"force"`

is chosen, the timestamp will have the same clock time as the timestamp in the database, but with the new time zone.

`array` How arrays should be returned. There are two options: `"none"` and `"matrix"`. If `"none"` is selected, arrays are not returned. Instead an error is generated. If `"matrix"` is selected, arrays are returned as a column matrix. Each array is one row in the matrix.

`geometry` How geometry columns should be returned. There are two options: `"blob"` and `"wk"`. If `"blob"` is selected, geometry columns are returned as a list of raw vectors containing WKB data. If `"wk"` is selected, geometry columns are returned as **wk** `wk_wkb` vectors. Use `wk::wk_handle()` or `sf::st_as_sfc()` to convert to other geometry formats.

`map` How `MAP` columns should be returned. There are two options: `"data.frame"` and `"list_of"`. If `"data.frame"` is selected (the default), `MAP` columns are returned as a list of data frames with `key` and `value` columns. If `"list_of"` is selected, `MAP` columns are returned as a `vctrs::list_of()` whose `ptype` is a `data.frame(key = <K>, value = <V>)` that records the SQL key/value types. This enables MAP columns to round-trip through `dbWriteTable()` / `dbCreateTable()` without specifying `field.types`, and lets scans accept named-list cells as MAP entries.

`conn` A `duckdb_connection` object

`shutdown` Unused. The database instance is shut down automatically.

## Details

The behavior of `with = "force"` at DST transitions depends on how R handles translation from the underlying time representation to a human-readable format. If the timestamp is invalid in the target timezone, the resulting value may be `NA` or an adjusted time.

## Value

`duckdb()` returns an object of class duckdb_driver.

`dbDisconnect()` and `duckdb_shutdown()` are called for their side effect.

An object of class "adbc_driver"

`dbConnect()` returns an object of class duckdb_connection.

## Database instances and driver reuse

`duckdb()` returns a driver object that owns a DuckDB *database instance*. `dbConnect()` opens connections to that instance, and many connections can share one instance.

For a file-based `dbdir`, the instance is cached, keyed by the (normalized) path: calling `duckdb()` again with the same `dbdir` returns the same driver and instance while it is still alive. This is deliberate. DuckDB allows only a single read-write handle to a database file at a time, so opening a second instance of the same file would fail with a lock error. Reusing one instance instead lets any number of `dbConnect(duckdb(dbdir = "my.db"))` calls share it. An in-memory database (`:memory:`, the default) has no file to lock and is never cached: every `duckdb()` call creates a fresh, isolated instance.

Because the instance is created once per database file, `config`, `read_only`, `home`, and `shared_home` take effect only at creation. A call that reuses an existing instance ignores them. To apply different values to a file-based database – for example to reopen it read-only, or to send extensions and secrets elsewhere – first release the instance with `duckdb_shutdown()`, which also drops it from the cache, then create it again. `dbDisconnect()` only closes a connection, it does not release the instance, and its `shutdown` argument is unused. Instances are shut down automatically when the driver is garbage-collected or the session ends.

## DuckDB extensions on Linux

DuckDB's prebuilt extensions for Linux are compiled with the GNU C++ standard library (`libstdc++`). Loading one into a `duckdb` package that was itself built with a *different* C++ standard library – most commonly `libc++` (clang's `-stdlib=libc++`) – is an ABI mismatch that crashes R (https://github.com/duckdb/duckdb-r/issues/1107). Almost all Linux builds (CRAN binaries and most source installs) use `libstdc++` and are unaffected; macOS and Windows are unaffected.

Each `duckdb()` call decides whether the driver it returns may load extensions, via the `allow_extensions` argument, the `duckdb.allow_extensions` option, the `DUCKDB_R_ALLOW_EXTENSIONS` environment variable, or automatic detection. On the automatic path a build that was not compiled with `libstdc++` on Linux disables extensions: `INSTALL` / `LOAD` raise a clear error instead of crashing, automatic extension install/load is turned off, and a throttled advisory message is shown when `duckdb()` is called. Pass `allow_extensions = FALSE` to disable extensions and silence that message, or `allow_extensions = TRUE` to attempt loading anyway (which may still crash R).

The decision is carried on the returned driver as the experimental `allow_extensions` slot (see duckdb_driver).

## Examples

```
library(adbcdrivermanager)
with_adbc(db <- adbc_database_init(duckdb_adbc()), {
  as.data.frame(read_adbc(db, "SELECT 1 as one;"))
})


drv <- duckdb()
con <- dbConnect(drv)

dbGetQuery(con, "SELECT 'Hello, world!'")

dbDisconnect(con)
duckdb_shutdown(drv)

# Shorter:
con <- dbConnect(duckdb())
dbGetQuery(con, "SELECT 'Hello, world!'")
dbDisconnect(con, shutdown = TRUE)
```

## duckdb_explain-class *DuckDB EXPLAIN query tree*

### Description

DuckDB EXPLAIN query tree

## duckdb_read_csv *Reads a CSV file into DuckDB*

### Description

Directly reads a CSV file into DuckDB, tries to detect and create the correct schema for it. This usually is much faster than reading the data into R and writing it to DuckDB.

### Usage

```
duckdb_read_csv(
  conn,
  name,
  files,
  ...,
  header = TRUE,
  na.strings = "",
  nrow.check = 500,
  delim = ",",
  quote = "\"",
  col.names = NULL,
  col.types = NULL,
  lower.case.names = FALSE,
  sep = delim,
  transaction = TRUE,
  temporary = FALSE
)
```

### Arguments

| | |
|---|---|
| conn | A DuckDB connection, created by dbConnect(). |
| name | The name for the virtual table that is registered or unregistered |
| files | One or more CSV file names, should all have the same structure though |
| ... | These dots are for future extensions and must be empty. |
| header | Whether or not the CSV files have a separate header in the first line |
| na.strings | Which strings in the CSV files should be considered to be NULL |

| | |
|---|---|
| nrow.check | How many rows should be read from the CSV file to figure out data types |
| delim | Which field separator should be used |
| quote | Which quote character is used for columns in the CSV file |
| col.names | Override the detected or generated column names |
| col.types | Character vector of column types in the same order as col.names, or a named character vector where names are column names and types pairs. Valid types are DuckDB data types, e.g. VARCHAR, DOUBLE, DATE, BIGINT, BOOLEAN, etc. |
| lower.case.names | Transform column names to lower case |
| sep | Alias for delim for compatibility |
| transaction | Should a transaction be used for the entire operation |
| temporary | Set to TRUE to create a temporary table |

## Details

If the table already exists in the database, the csv is appended to it. Otherwise the table is created.

## Value

The number of rows in the resulted table, invisibly.

## Examples

```
con <- dbConnect(duckdb())

data <- data.frame(a = 1:3, b = letters[1:3])
path <- tempfile(fileext = ".csv")

write.csv(data, path, row.names = FALSE)

duckdb_read_csv(con, "data", path)
dbReadTable(con, "data")

dbDisconnect(con)


# Providing data types for columns
path <- tempfile(fileext = ".csv")
write.csv(iris, path, row.names = FALSE)

con <- dbConnect(duckdb())
duckdb_read_csv(con, "iris", path,
  col.types = c(
    Sepal.Length = "DOUBLE",
    Sepal.Width = "DOUBLE",
    Petal.Length = "DOUBLE",
    Petal.Width = "DOUBLE",
    Species = "VARCHAR"
```

```
  )
)
dbReadTable(con, "iris")
dbDisconnect(con)
```

---

`duckdb_register` *Register a data frame as a virtual table*

---

## Description

`duckdb_register()` registers a data frame as a virtual table (view) in a DuckDB connection. No data is copied.

## Usage

```
duckdb_register(conn, name, df, overwrite = FALSE, experimental = FALSE)

duckdb_unregister(conn, name)
```

## Arguments

| | |
|---|---|
| `conn` | A DuckDB connection, created by `dbConnect()`. |
| `name` | The name for the virtual table that is registered or unregistered |
| `df` | A `data.frame` with the data for the virtual table |
| `overwrite` | Should an existing registration be overwritten? |
| `experimental` | Enable experimental optimizations |

## Details

`duckdb_unregister()` unregisters a previously registered data frame.

## Value

These functions are called for their side effect.

## Examples

```
con <- dbConnect(duckdb())

data <- data.frame(a = 1:3, b = letters[1:3])

duckdb_register(con, "data", data)
dbReadTable(con, "data")

duckdb_unregister(con, "data")

dbDisconnect(con)
```

## duckdb_register_arrow *Register an Arrow data source as a virtual table*

### Description

duckdb_register_arrow() registers an Arrow data source as a virtual table (view) in a DuckDB connection. No data is copied.

### Usage

```
duckdb_register_arrow(conn, name, arrow_scannable, use_async = NULL)

duckdb_unregister_arrow(conn, name)

duckdb_list_arrow(conn)
```

### Arguments

| | |
|---|---|
| conn | A DuckDB connection, created by dbConnect(). |
| name | The name for the virtual table that is registered or unregistered |
| arrow_scannable | A scannable Arrow-object |
| use_async | Switched to the asynchronous scanner. (deprecated) |

### Details

duckdb_unregister_arrow() unregisters a previously registered data frame.

### Value

These functions are called for their side effect.

## duckdb_storage_status *DuckDB file-system usage: storage locations and how they are resolved*

### Description

**[Experimental]**

DuckDB writes several distinct kinds of data to the file system. This page catalogs every such location and documents the policy the duckdb R package uses to choose them. By default the package never creates anything in your home directory on its own: downloaded extensions and stored secrets go under the R session's temporary directory unless a ~/.duckdb directory already exists (or you point the package somewhere explicitly).

duckdb_storage_status() reports where each location currently resolves.

## Usage

```
duckdb_storage_status()
```

## Details

`duckdb_storage_status()` reports the directory the package would currently use for downloaded extensions and for persisted secrets, and which tier of the resolution above chose it. It has no side effects: it never prompts and never creates a directory, so an as-yet-uncreated `~/.duckdb` is reported as the per-session temporary default.

## Value

`duckdb_storage_status()` returns a data frame (class `"duckdb_storage_status"`) with one row per kind of state and columns `kind`, `source`, and `directory`; its print method renders a readable summary when the result is auto-printed.

## Kinds of on-disk state

**Home directory** The base DuckDB uses to expand a leading `~` and to derive default sub-locations. DuckDB setting: `home_directory`. The package does not set this: doing so would also redirect `~` in user SQL (e.g. `COPY ... TO '~/out.csv'`). The extension and secret locations below are pointed at the resolved home root directly instead.

**Extension binaries** Downloaded `*.duckdb_extension` files (e.g. `spatial`, `httpfs`, `h3`). DuckDB setting: `extension_directory`. A re-usable cache placed at `<home>/extensions`, where `<home>` is resolved as described below.

**Stored secrets** Persisted credentials under `stored_secrets`. DuckDB setting: `secret_directory`. Placed at `<home>/stored_secrets`, the same `<home>`.

**Temporary / spill files** Out-of-core intermediates for sorts, hash joins, and similar operations. DuckDB settings: `temp_directory`, `max_temp_directory_size`. For an in-memory (`:memory:`) database DuckDB's own default spills to `.tmp` in the current working directory, so the package overrides it with a `tempdir()` sub-directory by default. This is a separate setting from the extension/secret home (see below).

**Logs and profiling output** Written only when a path is explicitly configured (DuckDB settings `log_query_path`, `http_logging_output`, profiling output). They default to *off*, so nothing is written without the user asking, and the user chooses where it goes.

**Database file, WAL, and checkpoints** Chosen by the user through the `dbdir` argument of `duckdb()`. The package does not manage these.

## Resolving the home directory

Extensions and secrets share one *home* root, resolved fresh on every call to `duckdb()` that creates a new database driver object. The first source that yields a value wins:

1. the `home` argument to `duckdb()`;
2. the `duckdb.home` R option, e.g. `options(duckdb.home = "/path/to/duckdb")`;
3. the `DUCKDB_R_HOME` environment variable;

4. ~/.duckdb, if that directory already exists – the location shared with the DuckDB CLI and other clients;
5. In interactive sessions only, the package offers to create ~/.duckdb once: answer "yes" to create and use it, "no" to fall through to the temporary directory below, or cancel the prompt to abort with an error.
6. Otherwise a per-session sub-directory of tempdir().

The extension cache is then <home>/extensions and the secret store is <home>/stored_secrets.

Because the decision is remade on every new driver object, creating ~/.duckdb (or setting the option/variable) takes effect immediately for drivers created afterwards. Existing drivers are unaffected.

The shared_home argument of duckdb() overrides this resolution: shared_home = TRUE uses (and creates) ~/.duckdb, and shared_home = FALSE forces a per-session tempdir() even if ~/.duckdb already exists.

## Per-location reference

| Kind | DuckDB setting | How to set it | Default |
|---|---|---|---|
| Home | home_directory | – | left untouc |
| Extensions | extension_directory | home arg / duckdb.home / DUCKDB_R_HOME (as <home>/extensions) | tempdir() |
| Stored secrets | secret_directory | like extensions (<home>/stored_secrets) | tempdir() |
| Temp/spill | temp_directory | duckdb.temp_directory / DUCKDB_R_TEMP_DIRECTORY | tempdir() |
| Logs | log_query_path | DuckDB setting | disabled (c |

"set" means duckdb() sets the value explicitly in the database config. The home directory is left untouched so that ~ in user SQL keeps its usual meaning. An extension_directory / secret_directory / temp_directory passed directly in the config list is always honored and takes precedence over the resolution above.

## Messages

**Storage-location message** When the package picked the location itself (a per-session tempdir(), or an existing ~/.duckdb), duckdb() emits an informational message describing where extensions and secrets are going and how to change it. It is throttled by session type: in an **interactive** session at most once every eight hours (a human can act on it); in a **non-interactive** session up to 60 times, after which it goes silent for good, so a long-running or automated process is not reminded forever. The message is suppressed entirely when you chose the location yourself – the home or shared_home argument, the duckdb.home option, or the DUCKDB_R_HOME environment variable. Non-interactively it covers both the temporary directory and an existing ~/.duckdb; interactively it is issued only when the user opts out of creating ~/.duckdb. It is also suppressed once you have made any explicit home or shared_home choice earlier in the session: having set the location explicitly once, you have seen how, so later auto-resolved calls stay quiet.

### Silencing the message:

Make the choice explicit and it is no longer announced. Pass shared_home to duckdb() – TRUE to keep extensions and secrets under ~/.duckdb, FALSE to accept a per-session temporary directory.

Alternatively, point home (or the duckdb.home option / DUCKDB_R_HOME variable) at a location of your choice. As a last resort, use suppressMessages():

```
# Explicit arguments:
con <- dbConnect(duckdb(shared_home = FALSE))
con <- dbConnect(duckdb(home = "/path/to/duckdb"))

# As a fallback:
con <- suppressMessages(dbConnect(duckdb()))

# With configuration:
Sys.setenv(DUCKDB_R_HOME = "/path/to/duckdb")
con <- dbConnect(duckdb())
options(duckdb.home = "/path/to/duckdb")
con <- dbConnect(duckdb())
```

## Use by other packages

Packages that use duckdb inherit this policy:

- duckdb never writes outside tempdir() on its own during checks. In a non-interactive session (which all R CMD check runs are) it uses tempdir() by default unless a ~/.duckdb already exists, and it never *creates* ~/.duckdb unless requested. So a package that merely opens a database needs no special handling.
- Downloading and installing an extension is the caller's responsibility. Ensure that all tests involving extensions are skipped if the download fails. For robust testing on CRAN and other platforms, ensure that the extensions your package uses can be downloaded and installed. Run the check in a subprocess to avoid crashing the main R process if the extension is incompatible with the platform. To force a throwaway cache in your own tests, connect with an explicit home:

```
tempdir_for_tests <- withr::local_tempdir()
con <- DBI::dbConnect(duckdb::duckdb(home = tempdir_for_tests))
```

## See Also

duckdb() for the home and shared_home arguments.

## Examples

```
duckdb_storage_status()
```

sql_query *Run an SQL query or statement*

## Description

**[Experimental]**

`sql_query()` runs an arbitrary SQL query using `DBI::dbGetQuery()` and returns a data.frame with the query results. `sql_exec()` runs an arbitrary SQL statement using `DBI::dbExecute()` and returns the number of affected rows.

These functions are intended as an easy way to interactively run DuckDB without having to manage connections. By default, data frame objects are available as views.

Scripts and packages should manage their own connections and prefer the DBI methods for more control.

## Usage

```
sql_query(sql, conn = default_conn())

sql_exec(sql, conn = default_conn())
```

## Arguments

| | |
|---|---|
| sql | A SQL string |
| conn | An optional connection, defaults to `default_conn()` |

## Value

A data frame with the query result

## Examples

```
# Queries
sql_query("SELECT 42")

# Statements with side effects
sql_exec("CREATE TABLE test (a INTEGER, b VARCHAR)")
sql_exec("INSERT INTO test VALUES (1, 'one'), (2, 'two')")
sql_query("FROM test")

# Data frames available as views
sql_query("FROM mtcars")
```

# Index


adbcdrivermanager::adbc_driver(), *4*

backend-duckdb, 2

data.frame, *16*
dbConnect,duckdb_driver-method
(duckdb), 4
dbConnect__duckdb_driver (duckdb), 4
dbCreateTable(), *7*
dbDisconnect(), *8*
dbDisconnect,duckdb_connection-method
(duckdb), 4
dbDisconnect__duckdb_connection
(duckdb), 4
DBI::dbExecute(), *16*
DBI::dbGetQuery(), *16*
dbWriteTable(), *7*
default_conn, 3
default_conn(), *3*, *16*
dplyr::tbl(), *2*
duckdb, 4
duckdb(), *13–15*
duckdb_adbc (duckdb), 4
duckdb_connection, *7*
duckdb_driver, *7*, *8*
duckdb_explain (duckdb_explain-class), 9
duckdb_explain-class, 9
duckdb_list_arrow
(duckdb_register_arrow), 12
duckdb_read_csv, 9
duckdb_register, 11
duckdb_register_arrow, 12
duckdb_shutdown (duckdb), 4
duckdb_shutdown(), *8*
duckdb_storage, *5*, *6*
duckdb_storage (duckdb_storage_status),
12
duckdb_storage_status, 12
duckdb_storage_status(), *12*
duckdb_unregister (duckdb_register), 11
duckdb_unregister_arrow
(duckdb_register_arrow), 12

print.duckdb_explain
(duckdb_explain-class), 9

sf::st_as_sfc(), *7*
simulate_duckdb (backend-duckdb), 2
sql_exec (sql_query), 16
sql_query, 16
suppressMessages(), *15*
Sys.timezone(), *6*

tbl_file (backend-duckdb), 2
tbl_function (backend-duckdb), 2
tbl_query (backend-duckdb), 2
tempdir(), *13–15*

vctrs::list_of(), *7*

wk::wk_handle(), *7*