

Package: duckdb.1.5.dev (via r-universe)

October 1, 2026

Title DBI Package for the DuckDB Database Management System

Version 1.5.6.9000.0

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 (>= 2.6.0), 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/gha/extra-packages adbcdrivermanager=?ignore-unavailable&ignore-build-errors,
arrow=?ignore-build-errors

Config/comment/gha/extra-packages arrow does not compile against Rtools45, and the aarch64 Windows runner has no binary to install instead. adbcdrivermanager needs ignore-unavailable as well, and needs it on every runner: CRAN archived it, and pak is called with dependencies = ``all", which resolves Enhances too, so the package is in the plan and resolves to nothing. ignore-build-errors does not cover that -- it demotes a failed build, not a failed lookup. ignore-build-errors stays for the day it returns to CRAN and Rtools45 has to build it again. Explained in handbook/operations/ci/matrix/README.md.

Config/testthat/edition 3
Encoding UTF-8
Roxygen list(markdown = TRUE)
SystemRequirements xz (for building from source)
Config/roxygen2/version 8.1.0.9000
Repository <https://krmlr.r-universe.dev>
Date/Publication 2026-10-01 08:04:54 UTC
RemoteUrl <https://github.com/krmlr/duckdb-r>
RemoteRef v1.5-variegata-fwd-green
RemoteSha 25132c3f9196f36eb9d42532fd90f60666635a4d

Contents

backend-duckdb	2
default_conn	4
duckdb	4
duckdb_error	9
duckdb_explain-class	10
duckdb_memory	10
duckdb_read_csv	12
duckdb_register	14
duckdb_register_arrow	15
duckdb_storage_status	16
duckdb_types	19
duckdb_types_arrow	22
sql_query	26
to_arrow_stream	27
Index	29

backend-duckdb	<i>DuckDB SQL backend for dbplyr</i>
----------------	--------------------------------------

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

`tbl_function()` is an experimental variant of `dbplyr::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 `dbplyr::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

<code>src</code>	A duckdb connection object, <code>default_conn()</code> if omitted.
<code>path</code>	Path to existing Parquet, CSV or JSON file
<code>...</code>	Any parameters to be forwarded
<code>cache</code>	Enable object cache for Parquet files
<code>query</code>	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	<i>Get the default connection</i>
--------------	-----------------------------------

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	<i>Connect to a DuckDB database instance</i>
--------	--

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

<code>dbdir</code>	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.
<code>read_only</code>	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 <code>drv</code> object or a <code>drv</code> object pointing to the same path) cannot apply it, and fail rather than ignoring it.
<code>bigint</code>	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 cannot apply them, and fail rather than ignoring them.
...	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 <i>not</i> 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 <code>duckdb()</code>
debug	Print additional debug information, such as queries.
timezone_out	The time zone in which plain <code>TIMESTAMP</code> columns (without time zone) are returned to R, defaults to "UTC". If you want to display datetime values in the local timezone, set to <code>Sys.timezone()</code> or "". <code>TIMESTAMPTZ</code> columns follow the session's <code>TimeZone</code> setting instead.
tz_out_convert	How to convert timestamp columns to the timezone specified in <code>timezone_out</code> . 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 <code>wk</code> <code>wk_wkb</code> vectors. Use <code>wk::wk_handle()</code> or <code>sf::st_as_sf()</code> 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 <code>vctrs::list_of()</code> whose <code>ptype</code> is a <code>data.frame(key = <K>, value = <V>)</code> that records the SQL key/value types. This enables MAP columns to round-trip through <code>dbWriteTable()</code> / <code>dbCreateTable()</code> without specifying <code>field.types</code> , and lets scans accept named-list cells as MAP entries.
conn	A <code>duckdb_connection</code> 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 fails with a lock error in another process, and, on Linux and macOS, is not prevented at all within the same one. 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.

The key is the path as the *engine* resolves it, not as `normalizePath()` does. DuckDB canonicalizes the longest part of the path that exists and appends the rest, so a database that does not exist yet gets the key it will keep once created, and two spellings of one database (a relative path, a symlink, a different separator) share an instance instead of each opening their own. A path that resolves no further is used as it stands rather than refused. Symbolic links are not supported on Windows, where creating one takes administrator rights or Developer Mode: there, a symlink to a database that does not exist yet fails to open.

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 cannot apply them, and fails rather than dropping them. Passing dbdir to dbConnect() fails too when the driver owns a database file of its own, because the connection would go to dbdir while the driver kept its own database open. 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()` closes one connection, and its shutdown argument is unused. Connections keep the instance alive, so it is released once the last connection to it closes, unless a result not yet cleared with `dbClearResult()` or an Arrow stream not yet released still uses it. The cache does not find an instance that only such a result or stream keeps open, so duckdb() with the same dbdir then opens a second instance of the file in the same session. A driver that was never connected to releases its instance when the driver is garbage-collected or the session ends. `dbIsValid()` reports whether a driver still holds an instance.

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_error

DuckDB error conditions

Description

[Experimental]

Every error the database engine raises reaches R as a condition of class `duckdb_error`, carrying DuckDB's own classification alongside the message. Catch it with [tryCatch\(\)](#) or `rlang::try_fetch()` and branch on the fields rather than on the message text, which is formatted for display and is not a stable interface.

Details

The condition is raised against the call that caused it – [DBI::dbGetQuery\(\)](#), [DBI::dbExecute\(\)](#), [DBI::dbBind\(\)](#), [DBI::dbFetch\(\)](#), and the relational API alike – and the fields survive that rethrow.

Fields

`error_type` DuckDB's exception type as a string, such as "BINDER", "PARSER", "CONSTRAINT", "CONVERSION", "IO" or "OUT_OF_MEMORY". This is the field to classify on. The set is the engine's and grows with it, so treat an unrecognized value as "some other error" rather than failing on it.

extra_info A named character vector of whatever else the engine attached to the error, for instance the position within the query. Which names appear depends on the error, and empty is a normal answer.

context The internal operation that failed, such as "rapi_prepare" or "rapi_execute". Useful to tell a failure at prepare time from one at execution time; the individual names are an implementation detail and may change.

raw_message The message as the engine phrased it, without the exception type prefix and without the display formatting.

A field the engine did not supply is absent from the condition, so reading it gives NULL. Classification code should therefore treat NULL as "unknown" and keep a fallback branch: an error raised before the engine is reached, by an R-level check or by a failing callback, is an ordinary error with none of these fields.

Errors are formatted with bullets when rlang is installed; without it the message is a single line, and the class and the fields are the same.

Examples

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

err <- tryCatch(dbGetQuery(con, "SELECT missing_column"), error = identity)
class(err)
err$error_type
err$context

dbDisconnect(con, shutdown = TRUE)
```

duckdb_explain-class *DuckDB EXPLAIN query tree*

Description

DuckDB EXPLAIN query tree

duckdb_memory *Memory-efficient reading and writing*

Description

A result that fits comfortably in the database can still be too large for R, and a data frame that fits in R can still be copied on its way into the database. This page shows memory-efficient ways for reading and writing:

- Memory limits
- Streamed reading via Arrow

- Registering a data frame for writing

The routes below keep R's share to one batch, or one data frame, at a time. Follow these recipes to process datasets larger than memory.

Set the limit where it takes effect

DuckDB and R allocate from two different budgets. The engine's memory is bounded by `memory_limit` and can spill to disk; R's vectors are unbounded and only spilled to the system swap space, if at all.

The engine's limit is set on the driver or on a live connection.

```
library(DBI)
con <- dbConnect(duckdb(config = list(memory_limit = "2GB")))
# or, on a live connection:
dbExecute(con, "SET memory_limit = '2GB'")
```

Reading: stream, and release each batch

Reading via Arrow stream never holds the result whole. Execution starts at `DBI::dbSendQueryArrow()` and pauses at a bounded buffer. Each `DBI::dbFetchArrowChunk()` hands over one batch that can be converted using `as.data.frame()`. For best results, release a batch immediately via `nanoarrow::nanoarrow_pointer_release()`. `duckdb_result_arrow` says what the release does and what happens without it.

```
rs <- dbSendQueryArrow(con, "SELECT * FROM huge")
repeat {
  batch <- dbFetchArrowChunk(rs, chunk_size = 1e6)
  if (batch$length == 0) break
  df <- as.data.frame(batch)
  # ... consume df ...
  nanoarrow::nanoarrow_pointer_release(batch)
}
dbClearResult(rs)
```

Writing: hand the data frame over, and let the engine scan it

`DBI::dbWriteTable()` copies nothing on the R side: it registers the data frame as a view that the engine scans in place, creates the table from that scan, and unregisters. `DBI::dbAppendTable()` does the same into an existing table, so data that arrives in pieces is appended piece by piece with the same cost.

```
dbWriteTable(con, "t", df)          # one frame
dbAppendTable(con, "t", next_df)    # or piece by piece
```

Where no table is needed at all, `duckdb_register()` alone lets queries scan the frame with negligible overhead.

```
duckdb_register(con, "df", df)
dbGetQuery(con, "SELECT count(*) FROM df WHERE a > 0.5")
```

Files: let the engine read them:

DuckDB supports readers for various formats, bypassing R memory entirely.

```
dbExecute(con, "CREATE TABLE t AS SELECT * FROM read_parquet('data.parquet')")
```

Arrow streams: append a batch at a time:

Data that arrives as an Arrow stream, from a pipe, a socket or another library's reader, goes in through `DBI::dbWriteTableArrow()`, or `DBI::dbAppendTableArrow()` into an existing table. Both pull one batch on R's thread and append it as a data frame.

```
stream <- nanoarrow::read_nanoarrow(file("data.arrows", "rb"))
dbWriteTableArrow(con, "t", stream)
```

Measurements

The method, every other route, and the same measurements through the Python, Node, Go and Rust clients are recorded in the repository: [experiments/2026-09-14-memory-clients/](#) for reading, [experiments/2026-09-19-memory-ingest/](#) for writing.

See Also

[duckdb_result_arrow](#) for the Arrow result and what releases a batch, [duckdb_register\(\)](#) and [duckdb_register_arrow\(\)](#) for scanning R data in place, [duckdb_storage](#) for where the engine spills.

duckdb_read_csv	<i>Reads a CSV file into DuckDB</i>
-----------------	-------------------------------------

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

<code>conn</code>	A DuckDB connection, created by <code>dbConnect()</code> .
<code>name</code>	The name for the virtual table that is registered or unregistered
<code>files</code>	One or more CSV file names, should all have the same structure though
<code>...</code>	These dots are for future extensions and must be empty.
<code>header</code>	Whether or not the CSV files have a separate header in the first line
<code>na.strings</code>	Which strings in the CSV files should be considered to be NULL
<code>nrow.check</code>	How many rows should be read from the CSV file to figure out data types
<code>delim</code>	Which field separator should be used
<code>quote</code>	Which quote character is used for columns in the CSV file
<code>col.names</code>	Override the detected or generated column names
<code>col.types</code>	Character vector of column types in the same order as <code>col.names</code> , 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.
<code>lower.case.names</code>	Transform column names to lower case
<code>sep</code>	Alias for <code>delim</code> for compatibility
<code>transaction</code>	Should a transaction be used for the entire operation
<code>temporary</code>	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`. Temporary storage is on by default, with the DuckDB CLI's semantics: the directory is created only when a query actually spills, and removed again when the database instance shuts down. An on-disk database keeps the

engine's own default, `<dbdir>.tmp` next to the database file. For an in-memory (`:memory:`) database the engine's own default would spill to `.tmp` in the current working directory, so the package points it at a fresh per-instance sub-directory below `tempdir()` instead. Every in-memory instance gets its own spill directory: instances must not share one, because the engine's spill file names are deterministic and an instance cleans up its directory when it shuts down. 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	<code>home_directory</code>	–	left untouched
Extensions	<code>extension_directory</code>	<code>home arg / duckdb.home / DUCKDB_R_HOME</code> (as <code><home>/extensions</code>)	<code>tempdir()</code>
Stored secrets	<code>secret_directory</code>	like extensions (<code><home>/stored_secrets</code>)	<code>tempdir()</code>
Temp/spill	<code>temp_directory</code>	<code>duckdb.temp_directory / DUCKDB_R_TEMP_DIRECTORY</code>	<code>memory: t</code>
Logs	<code>log_query_path</code>	DuckDB setting	disabled (o

"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. The temp/spill setting is left unset for

an on-disk database: the engine's own <dbdir>.tmp default already matches the DuckDB CLI, no matter whether the database is opened through `duckdb()` or through the `dbdir` argument of `DBI::dbConnect()`. 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(home = tempdir_for_tests))
```

See Also

[duckdb\(\)](#) for the `home` and `shared_home` arguments.

Examples

```
duckdb_storage_status()
```

duckdb_types

DuckDB data types in R

Description

This page documents every DuckDB type as it crosses to R and back: what a value of the type becomes when read, which R value writes it again, and what to do where neither works. See [duckdb_types_arrow](#) for a description of the conversion via Arrow.

The routes

Reading. [dbGetQuery\(\)](#) converts each column by its type, and four [dbConnect\(\)](#) arguments change the shape, with defaults `bigint = "numeric"`, `array = "none"`, `map = "data.frame"` and `geometry = "blob"`. [dbGetQueryArrow\(\)](#) hands out the engine's own Arrow export instead, and what each type becomes there, and in the R readers that convert the stream, is documented in [duckdb_types_arrow](#). A cast to `VARCHAR` in the query reads any type as text.

Writing. [dbWriteTable\(\)](#) and [duckdb_register\(\)](#) take a column's type from its R class, and `field.types` casts that column to the type it names, from any value that casts. [dbAppendTable\(\)](#) casts to the type of the existing column, and a parameter (`params =`) binds by its R class, cast by the query. A character column holding a value's text form writes every scalar type through `field.types`, because DuckDB parses the text it prints. Arrow, registered with [duckdb_register_arrow\(\)](#), writes the types no R class does.

Numbers

The **numeric** and **boolean** types:

- `BOOLEAN` (`BOOL`, `LOGICAL`) reads as `logical`, and `logical` writes it.
- `TINYINT`, `SMALLINT`, `UTINYINT`, `USMALLINT` read as `integer`, exactly. `integer` writes `INTEGER`, and `field.types` names the narrower type.

- INTEGER (INT4, INT, SIGNED) reads as integer, exactly but for the minimum, and integer writes it.
- UINTEGER reads as numeric, exactly; numeric writes DOUBLE, and `field.types` names UINTEGER.
- BIGINT (INT8, LONG) reads as numeric, exact up to 2^{53} , and its rounding past that is a **limitation**. With `bigint = "integer64"` it reads as `bit64::integer64`, exact but for the minimum. An integer64 column or parameter writes BIGINT whatever bigint says.
- UBIGINT reads as numeric, exact up to 2^{53} , and its rounding past that is a **limitation**. With `bigint = "integer64"` it reads as integer64, which holds the values below 2^{63} . Below 2^{63} , the integer64 it reads as writes it back through `field.types`, and its text writes any value.
- HUGEINT, UHUGEINT read as numeric, and bigint does not change that; their rounding is a **limitation**. Their text is exact both ways.
- BIGNUM (VARINT) reads and writes through its text, and Arrow writes it (see [duckdb_types_arrow](#)).
- DECIMAL(width, scale) (NUMERIC) reads as numeric at every width; its rounding is a **limitation**. Its text is exact both ways, and Arrow writes it exactly.
- FLOAT (REAL) and DOUBLE read as numeric, and numeric writes DOUBLE. NaN reads and writes as NaN, never as NA, and NA is NULL in both directions.

Text and binary

The **text**, **blob** and **bitstring** types, and UUID:

- VARCHAR (CHAR, BPCHAR, TEXT, STRING) reads as character, and character writes it; non-UTF-8 text is a **limitation**.
- BLOB (BYTEA, BINARY, VARBINARY) reads as a list of raw vectors. A `blob::blob` or a list of raw vectors writes it.
- BIT (BITSTRING) reads and writes through its text.
- UUID reads as character, lowercase and hyphenated. character writes VARCHAR, and `field.types` makes it a UUID.

Dates and times

The **date**, **time**, **timestamp** and **interval** types:

- DATE reads as Date, and a Date writes it, stored as double or as integer.
- TIME reads as `difftime` in seconds. Its text writes it through `field.types`, and so does Arrow (see [duckdb_types_arrow](#)).
- TIME_NS reads through Arrow, and to the microsecond through a cast to TIME in the query. Its text writes it, and so does Arrow.
- TIMETZ (TIME WITH TIME ZONE) reads as the `difftime` of its local time. The offset it drops is a **limitation**. Its text writes it.
- TIMESTAMP_S, TIMESTAMP_MS, TIMESTAMP (DATETIME) read as POSIXct. POSIXct writes TIMESTAMP, the instant in UTC, and `field.types` names the other precisions.
- TIMESTAMP_NS reads as POSIXct. POSIXct writes it to the microsecond through `field.types`, and Arrow writes it directly.

- `TIMESTAMPZ` (`TIMESTAMP WITH TIME ZONE`) reads as `POSIXct`. `POSIXct` writes the plain `TIMESTAMP` of the same instant; `field.types` makes it `TIMESTAMPZ`, and Arrow writes it directly.
- `INTERVAL` reads as `difftime` in seconds, counting a month as 30 days and a day as 24 hours. A `difftime` in any unit, or an `hms`, writes `INTERVAL`.

Enums and nested types

The `enum` type, and the `nested` ones:

- `ENUM` reads as `factor`, with every value of the type as a level. A `factor` or ordered column writes `ENUM` of its levels; a `factor` parameter binds as `VARCHAR`.
- `ARRAY` (`INTEGER[3]`) reads with `array = "matrix"`, as a matrix with a row per value. A matrix column writes it.
- `LIST` (`INTEGER[]`) reads as a list of vectors, `NULL` for a `NULL` row, and a list column whose elements share a type writes it.
- `MAP` reads as a list of `data.frame(key, value)`, which writes a list of structs unless `field.types` names the map. With `map = "list_of"`, the `vctrs::list_of()` it reads as writes back as `MAP` without `field.types` (#200), and a list column of named lists writes a list of structs, an entry per name, valued by the first element of the name's value, or by `NULL` where that value is `NULL` or empty. Its text casts to `MAP` in the query and as a parameter.
- `STRUCT` (`ROW`) reads as a data frame column. A data frame column writes it, and a data frame parameter binds a struct per row.
- `UNION` reads in the query through `union_tag()`, `union_extract()` or a cast to `VARCHAR`, and an Arrow result carries it. A column of a member's type writes it through `field.types`, which picks that member; text picks the `VARCHAR` member.
- `VARIANT` reads as a list, each value converted by its own type. A column of the value's type writes it through `field.types`.

Geometry

The `GEOMETRY` type, its coordinate reference system (CRS), and the `spatial` extension's own types:

- `GEOMETRY` reads as `WKB`. With `geometry = "blob"`, the default, it reads as a list of raw vectors; with `geometry = "wk"`, as `wk_wkb`, carrying the column's CRS as an attribute, which `sf::st_as_sf()` converts onward, CRS included. The type is core since DuckDB 1.5, so reading one needs no extension; the geometry functions are the `spatial extension's`. Arrow carries the column as `GeoArrow WKB` with its CRS, in both directions (see [duckdb_types_arrow](#)).
- **WKT writes** `GEOMETRY`. A character column of `WKT`, as `sf::st_as_text()` makes it, writes a `GEOMETRY` column with `field.types = c(geom = "GEOMETRY")` and appends to one with `dbAppendTable()`, because the cast from `VARCHAR` parses `WKT`. Naming the CRS in the type, as `"GEOMETRY('EPSG:4267')"`, gives the column its CRS. Writing `WKB` as raw vectors, an `sf` object or an `sfc` column is a **limitation**.
- **The spatial extension's own types are aliases, and read as what they alias.** `POINT_2D`, `POINT_3D`, `POINT_4D`, `BOX_2D` and `BOX_2DF` are structs, and read as data frame columns; `LINESTRING_2D` and `LINESTRING_3D` are lists of point structs, and read as lists of data frames; `POLYGON_2D` and `POLYGON_3D` are lists of those rings, and read as lists of lists; `WKB_BLOB`

is a BLOB, and reads as raw vectors. The same shapes write the plain struct or list, and `field.types` naming the alias casts back to it. They cast to GEOMETRY in the query, and the point, linestring, polygon and WKB types cast from it, as `'POINT (1 2)': :GEOMETRY::POINT_2D`.

Everything else

- An **untyped NULL comes back as NA_integer_**, matching the engine's own `SELECT NULL`; mapping it to logical NA instead was declined ([#155](#)). A typed NULL, as a scanned logical column or a bound NA parameter, round-trips as logical NA.
- JSON, the **json extension's** alias of VARCHAR, reads as character. Its text writes it through `field.types`.
- INET, the **inet extension's** address type, reads as a data frame column whose address is a HUGEINT read as a double, exact for IPv4; an IPv6 address is a **limitation**. Its text reads and writes it exactly.

Limitations and reference

The limitations are listed in the handbook, in [usage/types/](#).

The mapping is implemented in [src/types.cpp](#) (R vector to LogicalType) and [src/transform.cpp](#) (the way back). The list of types is DuckDB's own [documentation](#) for the release vendored here, and every entry on this page was measured on DuckDB 1.5.5, in [experiments/2026-09-26-type-catalog/](#), [experiments/2026-09-27-review-limits/](#), [experiments/2026-09-28-type-rereview/](#) or, for geometry route by route, [experiments/2026-08-09-spatial-interop/](#). Which zone labels a timestamp is documented in the handbook's [timestamps/](#).

What `expr_constant(NA)` builds in the relational API is documented in the handbook's [relational/](#).

duckdb_types_arrow

DuckDB data types through Arrow

Description

This page documents every DuckDB type as it crosses to R through Arrow and back: the Arrow type the engine exports it as, what the two R readers make of that, which Arrow type writes it again, and which R functions keep Arrow's types on the way in. See [duckdb_types](#) for a description of the direct conversion to R vectors.

The routes

Reading. Every function that returns Arrow hands out the engine's own export, under the connection's settings: `dbGetQueryArrow()`, `dbFetchArrow()` and `dbFetchArrowChunk()` after `dbSendQueryArrow()`, `dbReadTableArrow()`, `duckdb_fetch_arrow()` and `duckdb_fetch_record_batch()` after `dbSendQuery` (arrow = TRUE), and `arrow::to_arrow()`. No R vector exists until a reader converts the stream, and the two readers differ, so each entry below names both: nanoarrow's `as.data.frame()`, and arrow's `as.data.frame()` of `arrow::as_arrow_table()`.

The export settings are DuckDB's, set with SET, and each changes the Arrow type of some columns:

- `arrow_lossless_conversion = true` exports each type that has no exact Arrow counterpart as an extension type that names it, so that DuckDB, or another Arrow consumer that knows the name, gets the type back. Where nanoarrow falls back to the storage of one, the entries below say so.
- `arrow_large_buffer_size = true` gives strings, binary data and lists 64-bit offsets, as `large_string`, `large_binary` and `large_list`, which both readers convert as they convert the others.
- `arrow_output_version`, '1.0' by default, gates the newer layouts. From '1.4', binary data exports as `binary_view`, and `produce_arrow_string_view = true` and `arrow_output_list_view = true` take effect, as `string_view` and `list_view`; with an older version those two change nothing. From '1.5', a DECIMAL up to width 9 exports as `decimal32` and up to width 18 as `decimal64`. nanoarrow converts `string_view` and `binary_view`.

Writing. Only the routes that let DuckDB scan the Arrow data keep its types: `duckdb_register_arrow()`, and `arrow::to_duckdb()`, which calls it. `duckdb_register_arrow()` takes whatever `arrow::Scanner$create()` scans. Nothing is copied: the result is a view, and `CREATE TABLE ... AS SELECT * FROM` it writes a table. A registered arrow Table is scanned by every query, and a registered RecordBatchReader is a **limitation**. Every other route converts through an R data frame, so a column lands as the type its R vector writes (see `duckdb_types`). `dbWriteTableArrow()`, `dbCreateTableArrow()` and `dbAppendTableArrow()` are DBI's defaults, which do that batch by batch. `dbBindArrow()` converts the same way, then binds by position.

Reading

Numbers:

- `BOOLEAN` exports as `bool` and reads as `logical`; with `arrow_lossless_conversion`, as `arrow.bool8`, which nanoarrow reads as the integer of its storage.
- `TINYINT`, `SMALLINT`, `UTINYINT`, `USMALLINT` export as `int8`, `int16`, `uint8` and `uint16`, and read as `integer`, exactly.
- `INTEGER` exports as `int32` and reads as `integer`, exactly but for the minimum.
- `UINTEGER` exports as `uint32`. nanoarrow reads it as `numeric`; arrow reads it as `integer` when every value fits, and as `numeric` otherwise.
- `BIGINT` exports as `int64`. nanoarrow reads it as `numeric`, exact up to 2^{53} . arrow reads it as `integer` when every value fits and as `bit64::integer64` otherwise, exactly but for the minimum, or as `integer64` always under `options(arrow.int64_downcast = FALSE)`.
- `UBIGINT` exports as `uint64`, and both read it as `numeric`, exact up to 2^{53} ; its rounding past that is a **limitation**.
- `HUGEINT`, `UHUGEINT` export as `decimal128(38, 0)`, and both read them as `numeric`; their rounding is a **limitation**, and so is a `UHUGEINT` of 2^{127} or more reading as negative. With `arrow_lossless_conversion` they export as `arrow.opaque`, which carries every value. Their text reads them exactly (see `duckdb_types`).
- `BIGNUM` exports as `arrow.opaque` under either setting; nanoarrow reads its storage bytes as a blob.
- `DECIMAL(width, scale)` exports as `decimal128(width, scale)`, or narrower from output version 1.5, and both read it as `numeric`; its rounding is a **limitation**.
- `FLOAT`, `DOUBLE` export as `float` and `double`, and read as `numeric`.

Text and binary:

- VARCHAR exports as string and reads as character.
- BLOB exports as binary; nanoarrow reads it as a blob::blob, and arrow as an arrow_binary list of raw vectors.
- BIT exports as binary, the bytes DuckDB stores for the bit string, and reads as a blob or arrow_binary of those bytes. With arrow_lossless_conversion it exports as arrow.opaque, which nanoarrow reads as the same bytes. Its text reads it as 0 and 1 (see [duckdb_types](#)).
- UUID exports as string, lowercase and hyphenated, and reads as character. With arrow_lossless_conversion it exports as arrow.uuid.

Dates and times:

- DATE exports as date32 and reads as Date.
- TIME exports as time64('us') and reads as hms.
- TIME_NS exports as time64('ns') and reads as hms.
- TIMETZ exports as the time64('us') of its local time and reads as hms; the offset it drops is a **limitation**. With arrow_lossless_conversion it exports as arrow.opaque, which keeps the offset.
- TIMESTAMP_S, TIMESTAMP_MS, TIMESTAMP, TIMESTAMP_NS export as timestamp in their own unit, without a zone, and read as POSIXct, whose double cannot hold a TIMESTAMP_NS's nanoseconds, a **limitation**. The two readers label the same instant differently: nanoarrow gives it the zone UTC, so it prints the stored clock, and arrow gives it none, so it prints in R's session zone, a different clock outside UTC.
- TIMESTAMPTZ exports as timestamp('us', zone), where the zone is DuckDB's TimeZone setting, and both readers read it as a POSIXct labelled with that zone.
- INTERVAL exports as interval_month_day_nano.

Enums and nested types:

- ENUM exports as a dictionary of its values; nanoarrow reads it as character, and arrow as factor.
- ARRAY exports as a fixed_size_list and reads as a list of vectors (vctrs::list_of() or arrow_fixed_size_list), without the array = "matrix" that dbGetQuery() needs.
- LIST exports as list and reads as a list of vectors (vctrs::list_of() or arrow_list).
- MAP exports as map and reads as a list of key and value data frames.
- STRUCT exports as struct and reads as a data frame column, a tibble in arrow's case.
- UNION exports as a sparse_union. nanoarrow reads it as a data frame with a column per member, NA where the value is another member's.

Everything else:

- NULL, untyped, exports as int32 and reads as NA_integer_, as it does through dbGetQuery().
- JSON exports as string and reads as character; with arrow_lossless_conversion it exports as arrow.json, which nanoarrow reads as character.
- INET exports as a struct whose address is the HUGEINT the engine stores, as a decimal128(38, 0), which both read as a double, the same as dbGetQuery() does, and its text reads the address (see [duckdb_types](#)). With arrow_lossless_conversion that field becomes arrow.opaque.

Writing

Arrow types:

Each Arrow type lands as one DuckDB type when DuckDB scans it:

- `bool`, `int8` **to** `int64`, `uint8` **to** `uint64`, `float`, `double` land as `BOOLEAN`, the integer type of the same width and sign, `FLOAT` and `DOUBLE`.
- `decimal32`, `decimal64`, `decimal128` land as `DECIMAL` of the same width and scale.
- `string`, `large_string`, `string_view` land as `VARCHAR`, and `binary`, `large_binary`, `binary_view`, `fixed_size_binary` as `BLOB`.
- `date32`, `date64` land as `DATE`.
- `time32` **and** `time64('us')` land as `TIME`, and `time64('ns')` as `TIME_NS`.
- `timestamp` without a zone lands as `TIMESTAMP_S`, `TIMESTAMP_MS`, `TIMESTAMP` or `TIMESTAMP_NS` by its unit, and with a zone as `TIMESTAMPTZ`, the same instant.
- `duration` lands as `INTERVAL` in any unit.
- `interval_months` and `interval_month_day_nano` land as `INTERVAL`, each part kept.
- `list`, `large_list`, `list_view` land as `LIST`, and `fixed_size_list` as `ARRAY`.
- `struct` lands as `STRUCT`, `map` as `MAP`, and `sparse_union` as `UNION`.
- **A dictionary** lands as `VARCHAR`.
- `na` lands as a column of type `NULL`.
- **The extension types** land as the DuckDB type they name: `arrow.uuid` as `UUID`, `arrow.json` as `JSON`, `arrow.bool8` as `BOOLEAN`, and `arrow.opaque` as the DuckDB type in its metadata. What GeoArrow WKB lands as is under `Geometry`, and the other GeoArrow encodings are a **limitation**.

R classes, through Arrow:

An R vector reaches DuckDB through Arrow as the Arrow type its package infers, which differs from what `dbWriteTable()` gives for some classes. Where nothing is said below, `nanoarrow` and `arrow` infer the type `dbWriteTable()` writes.

- `factor` infers a dictionary and lands as `VARCHAR`, where `dbWriteTable()` writes `ENUM`. `arrow` lands an ordered one as `VARCHAR` too.
- `POSIXct` infers a timestamp with its zone, or R's session zone where it has none, and lands as `TIMESTAMPTZ`, where `dbWriteTable()` writes a plain `TIMESTAMP`.
- `difftime` infers a duration and lands as `INTERVAL` in hours and below, so 2 days land as `48:00:00`, where `dbWriteTable()` keeps the days.
- `hms` infers `time32` and lands as `TIME`, the one R route to that type; its truncation is a **limitation**.
- **A plain list of vectors** lands as `LIST` through `arrow`.
- **A matrix column** lands as `ARRAY` through `nanoarrow`.

Geometry

`GEOMETRY` and the spatial extension's own types through Arrow, and where the `geoarrow` and `sf` packages meet them:

- `GEOMETRY` **exports as** `geoarrow.wkb`, **with the column's CRS in the field's metadata**, as `PROJJSON` where the core or spatial knows the CRS, and as its identifier otherwise: `OGC:CRS84` exports as `PROJJSON` without spatial, and `EPSG:4267` only with it. It stays `geoarrow.wkb`

under every export setting, `arrow_lossless_conversion` included; `arrow_large_buffer_size` makes its storage `large_binary`, and an `arrow_output_version` from '1.4' makes it `binary_view`. With the `gearrow` package loaded, both readers convert it to a `gearrow_vctr` in each of those layouts, arrow the view one included.

- **sf reads a result through GeoArrow in one call.** With `gearrow` loaded, `sf::st_as_sf(dbGetQueryArrow(con, sql))` gives an `sf` whose geometries and CRS equal the source's, in the large and view layouts too, and so do `sf::st_as_sf()` of the result's `arrow::as_arrow_table()`, and `sf::st_as_sf()` of the `gearrow_vctr` column of `as.data.frame()`.
- **The spatial extension's own types cross as their storage.** `POINT_2D` and the other point and box types export as a struct, `LINESTRING_2D` and `LINESTRING_3D` as a list of point structs, `POLYGON_2D` and `POLYGON_3D` as a list of those lists, and `WKB_BLOB` as binary, with `arrow_lossless_conversion` too, and each reader converts them as it converts those Arrow types.
- **GeoArrow WKB writes GEOMETRY with its CRS.** Encode the geometry column as WKB, as `gearrow::as_gearrow_vctr(sf::st_geometry(x), schema = gearrow::gearrow_wkb(crs = sf::st_crs(x)))`, or as a `wk::as_wkb()` column, which `nanoarrow` infers as `gearrow.wkb` once `gearrow` is loaded. Register `arrow::as_arrow_table(nanoarrow::as_nanoarrow_array_stream(df))` with `duckdb_register_arrow()`, and a `CREATE TABLE ... AS SELECT` from the view writes a `GEOMETRY` column with the CRS, which reads back into an `sf` equal to the one written. With `spatial` loaded, the type names the CRS by its identifier, as `GEOMETRY('EPSG:4267')`; without it, it keeps the PROJJSON it came as, and `sf::st_crs()` reads the same CRS from both. WKB without a CRS lands as plain `GEOMETRY`, and so do the large and view layouts DuckDB's own export makes.
- **A `gearrow_vctr` column holds indices.** The `gearrow_vctr` that `as.data.frame()` gives for a geometry in an Arrow result is an integer vector of indices into the Arrow data it holds, and writing it back is a **limitation**.

Limitations and reference

The limitations are listed in the handbook, in [usage/arrow-types/](#).

The routes through R vectors are documented in [duckdb_types](#), and how a stream behaves, when it drains and what invalidates it, is documented in the handbook's [integrations/](#). Every entry on this page was measured on DuckDB 1.5.5, `nanoarrow` 0.9.0 and `arrow` 25.0.1 in [experiments/2026-09-27-arrow-types/](#) and [experiments/2026-09-28-type-rereview/](#), and geometry also with `gearrow` 0.4.4 and `sf` 1.1-3 in [experiments/2026-09-27-gearrow/](#).

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")
```

to_arrow_stream	<i>Stream a dbplyr table on DuckDB into Arrow</i>
-----------------	---

Description

[Experimental]

to_arrow_stream() is a streaming counterpart of arrow::to_arrow() for dbplyr tables on a DuckDB connection. It sends the query with [DBI::dbGetQueryArrow\(\)](#) and hands the stream to arrow::as_record_batch_reader(), so the rows arrive batch by batch and the result is not held twice. arrow::to_arrow() materializes the whole result first, through dbSendQuery(arrow = TRUE).

The streaming comes with hard limits, listed in the handbook, in [usage/integrations/](#). Use to_arrow_stream() where a large result goes straight into Arrow and nothing else runs on its connection until the reader has been read to the end. Where that cannot be arranged, use arrow::to_arrow(), or run everything else on a second connection.

Usage

```
to_arrow_stream(.data)
```

Arguments

<code>.data</code>	A dbplyr table on a DuckDB connection, or an Arrow object, which is returned unchanged.
--------------------	---

Value

An Arrow RecordBatchReader, or an `arrow_dplyr_query` over it if `.data` is grouped.

Examples

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

# Read the reader to the end before anything else runs on `con`.
reader <- to_arrow_stream(dplyr::filter(dplyr::tbl(con, "mtcars"), cyl == 4))
as.data.frame(reader$read_table())

# Another statement on `con` breaks a reader that has not been read yet.
reader <- to_arrow_stream(dplyr::tbl(con, "mtcars"))
dbGetQuery(con, "SELECT 1")
try(reader$read_table())

# A second connection to the same database leaves the reader alone.
other <- dbConnect(con@driver)
reader <- to_arrow_stream(dplyr::tbl(con, "mtcars"))
dbGetQuery(other, "SELECT count(*) FROM mtcars")
reader$read_table()$num_rows

dbDisconnect(other)
dbDisconnect(con)
```

Index

adbcdrivermanager::adbc_driver(), 4
arrow::as_arrow_table(), 22
arrow::to_arrow(), 22
arrow::to_duckdb(), 23

backend-duckdb, 2

data.frame, 26
dbAppendTable(), 19
dbAppendTableArrow(), 23
dbBindArrow(), 23
dbClearResult(), 8
dbConnect(), 19
dbConnect, duckdb_driver-method
 (duckdb), 4
dbConnect__duckdb_driver (duckdb), 4
dbCreateTable(), 7
dbCreateTableArrow(), 23
dbDisconnect(), 8
dbDisconnect, duckdb_connection-method
 (duckdb), 4
dbDisconnect__duckdb_connection
 (duckdb), 4
dbFetchArrow(), 22
dbFetchArrowChunk(), 22
dbGetQuery(), 19, 24
dbGetQueryArrow(), 19, 22
DBI::dbAppendTable(), 11
DBI::dbAppendTableArrow(), 12
DBI::dbBind(), 9
DBI::dbConnect(), 18
DBI::dbExecute(), 9, 26
DBI::dbFetch(), 9
DBI::dbFetchArrowChunk(), 11
DBI::dbGetQuery(), 9, 26
DBI::dbGetQueryArrow(), 27
DBI::dbSendQueryArrow(), 11
DBI::dbWriteTable(), 11
DBI::dbWriteTableArrow(), 12
dbIsValid(), 8

dbReadTableArrow(), 22
dbSendQueryArrow(), 22
dbWriteTable(), 7, 19, 25
dbWriteTableArrow(), 23
default_conn, 4
default_conn(), 3, 27
dplyr::tbl(), 2
duckdb, 4
duckdb(), 17–19
duckdb_adbc (duckdb), 4
duckdb_connection, 7
duckdb_driver, 7, 9
duckdb_error, 9
duckdb_explain (duckdb_explain-class),
 10
duckdb_explain-class, 10
duckdb_fetch_arrow(), 22
duckdb_fetch_record_batch(), 22
duckdb_list_arrow
 (duckdb_register_arrow), 15
duckdb_memory, 10
duckdb_read_csv, 12
duckdb_register, 14
duckdb_register(), 11, 12, 19
duckdb_register_arrow, 15
duckdb_register_arrow(), 12, 19, 23
duckdb_result_arrow, 11, 12
duckdb_shutdown (duckdb), 4
duckdb_shutdown(), 8
duckdb_storage, 6, 12
duckdb_storage (duckdb_storage_status),
 16
duckdb_storage_status, 16
duckdb_storage_status(), 16
duckdb_types, 19, 22–24, 26
duckdb_types_arrow, 19–21, 22
duckdb_unregister (duckdb_register), 14
duckdb_unregister_arrow
 (duckdb_register_arrow), 15

`normalizePath()`, 8

`print.duckdb_explain`
 (`duckdb_explain-class`), 10

`sf::st_as_sf()`, 26

`sf::st_as_sfc()`, 7, 21, 26

`sf::st_as_text()`, 21

`sf::st_crs()`, 26

`simulate_duckdb` (`backend-duckdb`), 2

`sql_exec` (`sql_query`), 26

`sql_query`, 26

`suppressMessages()`, 18

`Sys.timezone()`, 7

`tbl_file` (`backend-duckdb`), 2

`tbl_function` (`backend-duckdb`), 2

`tbl_query` (`backend-duckdb`), 2

`tempdir()`, 17, 18

`to_arrow_stream`, 27

`tryCatch()`, 9

`vctrs::list_of()`, 7, 21, 24

`wk::as_wkb()`, 26

`wk::wk_handle()`, 7