Package: Rducks (via r-universe)

May 26, 2026

Title Register R User-Defined Functions in DuckDB

Version 0.1.1

Description R package and 'DuckDB' extension bridge for registering R functions as 'DuckDB' user-defined functions (UDFs). The package is designed around a loaded 'DuckDB' extension, declarative type descriptors, 'nanoarrow' marshalling over 'Arrow C Data', and a calling-R-thread execution discipline for safe interaction with R from 'DuckDB' execution. Arrow IPC (inter-process communication) worker transport uses vendored 'NNG' (nanomsg next generation) for worker-process communication.

License GPL (>= 3)

Copyright See inst/LICENSE.note for bundled third-party copyright and licensing details.

SystemRequirements cmake for building vendored NNG/Mbed TLS support

Encoding UTF-8

Roxygen list(markdown = TRUE)

RoxygenNote 7.3.3

Depends R (>= 4.3)

Imports codetools, DBI, duckdb (>= 1.5.0), methods, mirai, nanoarrow, nanonext, S7

LinkingTo nanoarrow

Suggests bench, dplyr, duckplyr, globals (>= 0.18.0), knitr, mori, rmarkdown, tinytest

VignetteBuilder knitr

URL https://github.com/sounkou-bioinfo/Rducks, https://sounkou-bioinfo.github.io/Rducks/

BugReports https://github.com/sounkou-bioinfo/Rducks/issues

Config/pak/sysreqs cmake libzstd-dev xz-utils

Repository https://sounkou-bioinfo.r-universe.dev

Date/Publication 2026-05-26 09:41:52 UTC

RemoteUrl https://github.com/sounkou-bioinfo/Rducks

RemoteRef HEAD

RemoteSha 573ce8282218affa6e54e8830f0efe6aad190eda

Contents

`Ops.rducks_bits` *BIT logical operations*

Description

BIT logical operations

Usage

```
## S3 method for class 'rducks_bits'
Ops(e1, e2)

rducks_bits_xor(e1, e2)
```

Arguments

`e1`, `e2`	`rducks_bits` values, raw bytes, or 0/1 vectors.

Value

`rducks_bits` for bitwise operations or logical values for equality.

Examples

```
a <- rducks_bits("1010")
b <- rducks_bits("1100")
as.character(a & b)
as.character(a | b)
as.character(rducks_bits_xor(a, b))
a == b
```

rducks_argument_type_mapping *Describe how Rducks argument values are passed to R functions*

Description

`rducks_argument_type_mapping()` is the package-level source of truth for the R value shape used when DuckDB argument values are marshalled into an R function call. It is used by scalar-UDF registration checks and the nanoarrow scalar-UDF marshalling adapter.

Usage

```
rducks_argument_type_mapping(x = NULL)
```

Arguments

`x`	Optional scalar type tokens or constructed `rducks_type` descriptors. When `NULL`, all currently implemented DuckDB scalar-UDF scalar argument mappings are returned. Composite mappings should be requested with constructors such as `INTEGER[]`, `INTEGER[3]`, `STRUCT(a = INTEGER)`, and `MAP(VARCHAR, INTEGER)`.

Details

With `null_handling = "default"`, top-level SQL `NULL` inputs short-circuit to a SQL `NULL` result and the R function is not called. The `special_null_argument` column describes the value passed only when `null_handling = "special"`. This value is type-specific: ordinary R scalar types receive typed `NA` values, while exact Rducks value classes, binary values, and top-level composite values receive R `NULL`. Within homogeneous scalar lists/arrays, SQL `NULL` elements are represented as typed `NA` values where the child type has an R `NA` representation; nested composite `NULL` values are represented as R `NULL`.

The default table contains all scalar descriptors supported by the nanoarrow scalar-UDF marshalling adapter. `DECIMAL`, `ENUM`, `UNION`, and composite descriptors can be requested explicitly to inspect their recursive R function shapes.

Value

A data frame with one row per requested type descriptor.

Examples

```
rducks_argument_type_mapping()
rducks_argument_type_mapping(INTEGER)
rducks_argument_type_mapping(STRUCT(a = INTEGER, b = VARCHAR))
```

rducks_as_date *Convert R date/time values to Rducks scalar-UDF shapes*

Description

These helpers normalize common R date/time inputs to the R value shapes used by Rducks DuckDB scalar-UDF marshalling for DATE, TIME, TIMESTAMP, and INTERVAL values.

Usage

```
rducks_as_date(x, tz = "UTC", origin = "1970-01-01")

rducks_as_timestamp(x, tz = "UTC", origin = "1970-01-01")

rducks_as_time(x, tz = "UTC")

rducks_as_interval(
  x,
  units = c("secs", "mins", "hours", "days", "weeks"),
  months = 0L,
  days = 0L
)

rducks_interval_between(start, end, tz = "UTC")
```

Arguments

x, start, end	R date/time value. Supported inputs include Date, POSIXct/POSIXlt, difftime, numeric seconds where documented, and character strings accepted by base R or HH:MM:SS[.ffffff] for times. Numeric DATE/TIMESTAMP inputs must be finite or missing. DuckDB TIME inputs must be finite seconds in [0, 86400) or missing.
tz	Time zone used when converting date-times.
origin	Origin for numeric timestamp/date input.
units	Units for numeric interval input.
months, days	Extra interval month/day components for rducks_as_interval(). Numeric and difftime intervals are rounded to the nearest microsecond before constructing rducks_interval().

Value

rducks_as_date() returns a Date vector; rducks_as_time() returns numeric seconds since midnight; rducks_as_timestamp() returns POSIXct; rducks_as_interval() and rducks_interval_between() return rducks_interval.

Examples

```
rducks_as_date(as.Date("2024-01-15"))
rducks_as_date("2024-01-15")
rducks_as_timestamp(as.POSIXct("2024-01-15 12:00:00", tz = "UTC"))
rducks_as_time("08:30:00")
rducks_as_interval(3600, units = "secs")
rducks_interval_between(
  as.POSIXct("2024-01-01", tz = "UTC"),
  as.POSIXct("2024-01-02", tz = "UTC")
)
```

---

rducks_bigint *Construct exact DuckDB BIGINT values*

Description

Values are stored as canonical decimal strings so signed 64-bit values are not silently rounded through R double.

Usage

```
rducks_bigint(x = character())
```

Arguments

x	Numeric, integer, or character vector of whole numbers.

Value

Character vector with class `rducks_bigint`.

Examples

```
rducks_bigint(1:3)
rducks_bigint("9223372036854775807")
```

---

rducks_bits *Construct DuckDB BIT values*

Description

`rducks_bits()` stores bits as packed raw bytes plus an explicit bit length. Bits are packed left-to-right, with the first bit in the high bit of the first byte.

Usage

```
rducks_bits(x = raw(), length = NULL)

rducks_bits_raw(x)
```

Arguments

x	Character string of 0/1, logical/integer bit vector, raw bytes, or another rducks_bits object.
length	Optional bit length when x is raw.

Value

Object of class rducks_bits.

Examples

```
b <- rducks_bits("10110")
as.character(b)
rducks_bits_raw(b)
```

---

rducks_check_value *Check that an R value is compatible with a DuckDB type*

Description

This is a pre-marshalling guard for Rducks type descriptors. It checks the R value shape that the marshaller expects for scalar, decimal, enum, list, array, struct, map, and union descriptors.

Usage

```
rducks_check_value(type, x, size = NULL, what = "value")

rducks_check_argument(type, x, name = "argument")

rducks_check_return(type, x)
```

Arguments

type	A rducks_type descriptor such as INTEGER, INTEGER[], STRUCT(a = INTEGER), or a character scalar token accepted by rducks_type_normalize().
x	R value to check.
size	Optional exact length for scalar/vector checks.
what	Label used in error messages.
name	Argument label used by rducks_check_argument() in error messages.

Value

x, invisibly, on success.

Examples

```
rducks_check_value(INTEGER, 42L)
rducks_check_value(VARCHAR, "hello")
rducks_check_argument(DOUBLE, 3.14, name = "x")
rducks_check_return(BOOLEAN, TRUE)
```

---

rducks_current_execution_plan

Inspect the current Rducks execution plan

Description

Returns the R-side execution plan recorded for a DuckDB connection. If no plan has been recorded yet, this returns the reference plan `arrow_r + serial`.

Usage

```
rducks_current_execution_plan(con)
```

Arguments

con	A duckdb_connection.

Value

An object of class `rducks_execution_plan`.

Examples

```
db <- duckdb::dbConnect(duckdb::duckdb(config = list(allow_unsigned_extensions = "true")))
rducks_enable(db)
rducks_current_execution_plan(db)
rducks_release(db)
DBI::dbDisconnect(db)
```

rducks_decimal *Construct exact DuckDB DECIMAL values*

Description

Values are stored as fixed-point character data plus a declared width and scale. This avoids silently rounding exact decimal values through R double.

Usage

```
rducks_decimal(x = character(), width, scale = 0L)
```

Arguments

x	Numeric, integer, or character vector of fixed-point decimal values.
width	DuckDB decimal width, from 1 to 38.
scale	DuckDB decimal scale, from 0 to width.

Value

Object of class rducks_decimal.

Examples

```
rducks_decimal(c(1.5, 2.25, NA), width = 10, scale = 2)
```

rducks_disable_inproc *Disable in-process queued scalar-UDF execution*

Description

Switches a Rducks-enabled DuckDB connection back to the direct serial backend. Optionally updates DuckDB thread settings at the same time.

Usage

```
rducks_disable_inproc(con, threads = NULL, external_threads = NULL)
```

Arguments

con	A duckdb_connection.
threads	Optional positive integer to set with PRAGMA threads.
external_threads	Optional positive integer to set with SET external_threads. Use NULL to leave unchanged.

Value

con, invisibly.

Examples

```
db <- duckdb::dbConnect(duckdb::duckdb(config = list(allow_unsigned_extensions = "true")))
rducks_enable(db)
rducks_enable_inproc(db)
rducks_disable_inproc(db)
rducks_release(db)
DBI::dbDisconnect(db)
```

---

rducks_duckdb_signature

Format a DuckDB scalar function signature

Description

Format a DuckDB scalar function signature

Usage

```
rducks_duckdb_signature(name, args, returns)
```

Arguments

name	SQL function name.
args	Argument type descriptors.
returns	Return type descriptor.

Value

Character scalar signature such as `f(INTEGER) -> DOUBLE`.

Examples

```
rducks_duckdb_signature("my_udf", c(INTEGER, VARCHAR), DOUBLE)
```

rducks_duckdb_types *Convert Rducks type descriptors to DuckDB SQL types*

Description

Convert Rducks type descriptors to DuckDB SQL types

Usage

```
rducks_duckdb_types(x)
```

Arguments

x	Character scalar tokens, rducks_type descriptors, or a list of descriptors.

Value

Character vector of DuckDB SQL type names.

Examples

```
rducks_duckdb_types(INTEGER)
rducks_duckdb_types(c(INTEGER, DOUBLE, VARCHAR))
rducks_duckdb_types(STRUCT(a = INTEGER, b = VARCHAR))
```

rducks_enable *Enable Rducks on a DuckDB connection*

Description

Loads the bundled Rducks DuckDB extension. The registration-safe R UDF path requires R API work to happen on the recorded main R thread; pass `threads = "single"` to set `external_threads=1` and `PRAGMA threads=1` explicitly. `rducks_enable()` also sets DuckDB's `arrow_lossless_conversion=true` option on the user connection; the extension applies the same setting to its internal connections so DuckDB-specific Arrow metadata is preserved for typed scalar-UDF, table, and query-stream marshalling. Use `rducks_set_execution_plan()` before scalar-UDF registration to select a non-reference marshalling or concurrency plan.

Usage

```
rducks_enable(
  con,
  extension_path = rducks_extension_path(),
  threads = c("unchanged", "single")
)
```

Arguments

con	A duckdb_connection.
extension_path	Extension path. Defaults to rducks_extension_path().
threads	Either "unchanged" or "single".

Value

con, invisibly.

Examples

```
db <- duckdb::dbConnect(duckdb::duckdb(config = list(allow_unsigned_extensions = "true")))
rducks_enable(db)
rducks_release(db)
DBI::dbDisconnect(db)
```

---

rducks_enable_inproc *Enable in-process queued scalar-UDF execution*

Description

Switches a Rducks-enabled DuckDB connection to an inproc_concurrent execution plan for subsequent scalar-UDF registrations and updates the native runtime backend. This backend preserves R's thread discipline: DuckDB worker-side scalar-UDF callbacks submit chunk requests to an extension-owned queue, and the recorded main R thread drains the queue and performs all R API work. This is a same-process scheduling mode, not a performance promise; R function calls are still serialized on the main R thread.

Usage

```
rducks_enable_inproc(con, threads = NULL, external_threads = NULL)
```

Arguments

con	A duckdb_connection already enabled with rducks_enable().
threads	Optional positive integer to set with PRAGMA threads before enabling the in-process backend. Use NULL to leave unchanged.
external_threads	Optional positive integer to set with SET external_threads before enabling the in-process backend. Use NULL to leave unchanged. For actual DuckDB worker concurrency, keep this smaller than threads (for example threads = 4, external_threads = 1)

Details

This is a compatibility helper for the `arrow_r`/`arrow_c` in-process queue. New code can call `rducks_set_execution_plan()` directly with `rducks_execution_plan("arrow_r", "inproc_concurrent")` or `rducks_execution_plan("arrow_c", "inproc_concurrent")`. Select the plan before registering scalar UDFs whose reported execution plan should be the queued in-process path.

Value

`con`, invisibly.

Examples

```
db <- duckdb::dbConnect(duckdb::duckdb(config = list(allow_unsigned_extensions = "true")))
rducks_enable(db)
rducks_enable_inproc(db)
rducks_release(db)
DBI::dbDisconnect(db)
```

---

rducks_enum *Construct DuckDB ENUM values*

Description

`rducks_enum()` stores values as a factor with an additional class so the DuckDB enum dictionary is explicit.

Usage

```
rducks_enum(x, levels = NULL)
```

Arguments

`x`	Character vector or factor of enum values.
`levels`	Character vector of allowed enum dictionary values. If `x` is a factor and `levels` is omitted, the factor levels are used.

Value

Factor with class `rducks_enum`.

Examples

```
rducks_enum(c("a", "b", NA), levels = c("a", "b", "c"))
```

`rducks_execution_plan` *Define an Rducks execution plan*

Description

An execution plan describes how Rducks should marshal DuckDB chunks and what concurrency model is allowed. When stored on a connection it is the default for future `rducks_register_scalar_udf()` calls and updates the native runtime backend used for matching concurrent execution; the selected evaluator/marshalling is frozen into each registered scalar UDF's database-catalog metadata. It is separate from DuckDB function kind and from scalar-UDF registration semantics such as Rducks evaluation mode (`"scalar"` row calls versus `"vectorized"` chunk calls), argument/return types, NULL handling, error handling, and side effects.

Usage

```
rducks_execution_plan(
  marshalling = c("arrow_r", "arrow_c", "arrow_ipc"),
  concurrency = c("serial", "inproc_concurrent", "multiprocess_parallel"),
  ipc_globals = "auto",
  ipc_packages = NULL,
  ipc_timeout = NULL,
  ipc_endpoints = NULL,
  ipc_transport = NULL,
  ipc_globals_share = "none",
  ipc_provider = "nng",
  ipc_workers = 1L,
  ipc_max_pending = 64L
)
```

Arguments

`marshalling` Chunk marshalling implementation. `"arrow_r"` uses Arrow C Data plus nanoarrow/R materialization and is the reference implementation. `"arrow_c"` uses native C/DuckDB-vector materialization for supported scalar-UDF evaluation modes. `"arrow_ipc"` uses Arrow IPC bytes as the explicit task/result payload for the NNG multiprocess path.

`concurrency` Concurrency contract. `"serial"` evaluates one chunk at a time in the calling process. `"inproc_concurrent"` allows in-process DuckDB callback concurrency while keeping R API work serialized on the recorded main R thread. `"multiprocess_parallel"` uses persistent NNG/nanonext workers for process-isolated chunk work and requires `marshalling = "arrow_ipc"`. When `ipc_endpoints` is `NULL`, Rducks starts local worker loops with mirai daemons; otherwise the endpoint URLs are passed through unchanged.

`ipc_globals`, `ipc_packages`, `ipc_timeout`, `ipc_endpoints`, `ipc_transport` Arrow IPC worker options. By default (`ipc_globals = "auto"`), Rducks discovers scalar-UDF globals once at registration-wrapper creation and broadcasts

them to each NNG worker when the scalar UDF is registered with the shared provider pool. Automatic capture estimates the serialized globals payload and warns when it exceeds option `rducks.ipc_globals.warn_bytes` (8 MiB by default); option `rducks.ipc_globals.max_bytes` can set a hard byte limit. Set `ipc_globals_share = "mori"` to pass selected globals through mori shared memory references for same-host workers; Rducks keeps the shared objects anchored for the registered scalar UDF lifetime. Use `ipc_packages` for packages that workers should attach, `ipc_globals = FALSE` to rely only on the serialized UDF closure and explicit task state, or a character vector / named list for explicit extra globals. `ipc_timeout` is the positive finite provider wait timeout in seconds; `NULL` uses a finite default of 30 seconds. `ipc_endpoints` optionally supplies NNG endpoint URLs for worker processes that the caller starts and stops; those processes must run the Rducks NNG worker loop. Any NNG URL transport supported by both endpoints is allowed. When endpoints are not supplied, `ipc_transport` selects the transport used for the mirai-launched local worker endpoints and must be left as `NULL` when explicit `ipc_endpoints` are supplied. Rducks retries local TCP/WebSocket startup with fresh endpoint bundles after startup-ping failure; caller-supplied endpoints remain caller-owned and fail fast. `"abstract"` means Linux abstract IPC, `"ipc"` means NNG IPC (Unix-domain sockets on POSIX and named pipes on Windows), `"unix"` means the POSIX Unix-domain alias, and `"tcp"` / `"ws"` use loopback TCP / WebSocket endpoints. The default is `"abstract"` on Linux and `"ipc"` elsewhere.

`ipc_globals_share`
: How selected IPC globals are represented before worker broadcast. `"none"` serializes them into the registration payload. `"mori"` applies `mori::share()` to each selected global before serialization, which can turn large atomic vectors, lists, and data frames into same-host shared-memory references. This requires the optional mori package and workers on the same machine.

`ipc_provider`
: Worker provider for `arrow_ipc + multiprocess_parallel`. Only `"nng"` is supported. The NNG provider broadcasts each registered scalar UDF closure plus discovered globals/packages to every worker in the shared database-runtime provider pool, so avoid capturing large objects in UDF environments unless that memory cost is intended or `ipc_globals_share = "mori"` is appropriate.

`ipc_workers`
: Number of persistent NNG workers.

`ipc_max_pending`
: Maximum simultaneous native NNG requests admitted per registered scalar-UDF client pool. `NULL` uses the provider default of 64. Non-IPC plans store `NA_integer_` for this field. The current provider still uses synchronous request/reply per callback rather than collect-many batching, but this value is enforced as a bounded pending/in-flight guard before a callback enters the native request path.

Details

`arrow_r + serial` is the reference implementation used for conformance. Other plans must be explicitly implemented and validated against that reference; Rducks does not silently switch from one plan to another. `arrow_ipc + multiprocess_parallel` uses the native NNG path with ven-

dored nanoarrow C/IPC encoding. Each valid pair maps to a concrete internal `engine_id` such as `"arrow_c_direct_serial"` or `"ipc_nng_pool"`.

Value

An object of class `rducks_execution_plan`.

Examples

```
rducks_execution_plan("arrow_r", "serial")
rducks_execution_plan("arrow_c", "inproc_concurrent")
```

`rducks_explain_udf` *Explain a registered Rducks scalar UDF*

Description

Returns the R-side registration metadata together with native execution counters for a DuckDB scalar UDF registered by `rducks_register_scalar_udf()`. The `mode` column is the Rducks scalar-UDF evaluation mode, while `plan_id`, `marshalling`, and `concurrency` describe the plan recorded at registration time. The `r_side_record` column is `FALSE` when native catalog metadata is still present but the connection-local R registry view was detached or is otherwise unavailable. The native counters are useful for checking that a plan executed through its requested evaluator instead of silently switching engines: for example, an `arrow_c` scalar UDF should increment `arrow_c_chunks` and leave `arrow_r_chunks` unchanged.

Usage

```
rducks_explain_udf(con, name)
```

Arguments

`con`	A `duckdb_connection` with Rducks enabled.
`name`	SQL scalar-UDF function name registered with `rducks_register_scalar_udf()`.

Value

A one-row data frame with scalar-UDF registration metadata and native counters.

Examples

```
db <- duckdb::dbConnect(duckdb::duckdb(config = list(allow_unsigned_extensions = "true")))
rducks_enable(db, threads = "single")
rducks_register_scalar_udf(db, "my_fn", function(x) x + 1L,
  args = list(INTEGER), returns = INTEGER)
rducks_explain_udf(db, "my_fn")
rducks_release(db)
DBI::dbDisconnect(db)
```

rducks_extension_path *Locate the built Rducks DuckDB extension*

Description

Locate the built Rducks DuckDB extension

Usage

```
rducks_extension_path()
```

Value

Character scalar path to `rducks.duckdb_extension`.

Examples

```
rducks_extension_path()
```

rducks_hugeint *Construct exact DuckDB HUGEINT values*

Description

Values are stored as canonical decimal strings so values outside R's exact numeric range are not silently rounded.

Usage

```
rducks_hugeint(x = character())
```

Arguments

x	Numeric, integer, or character vector of whole numbers.

Value

Character vector with class `rducks_hugeint`.

Examples

```
rducks_hugeint(1:3)
rducks_hugeint("170141183460469231731687303715884105727")
```

rducks_inproc_self_test *Exercise the in-process queue*

Description

Runs a native self-test that submits `n` requests from worker threads to the extension-owned main-thread queue and drains them on the recorded main R thread. This validates the queue/condition-variable path without calling an R UDF. This diagnostic SQL surface is dev/test-only; set `RDUCKS_DEV_SURFACES=true` before `rducks_enable()` if you need it.

Usage

```
rducks_inproc_self_test(con, n = 1000L)
```

Arguments

`con`	A `duckdb_connection`.
`n`	Number of queue round trips to run.

Value

Integer-like numeric scalar: number of requests completed.

Examples

```
# Requires RDUCKS_DEV_SURFACES=true set before rducks_enable()
db <- duckdb::dbConnect(duckdb::duckdb(config = list(allow_unsigned_extensions = "true")))
rducks_enable(db)
rducks_inproc_self_test(db, n = 10L)
rducks_release(db)
DBI::dbDisconnect(db)
```

rducks_inproc_stats *Inspect in-process queue counters*

Description

Returns diagnostic counters for the extension-owned in-process queue. `submitted` counts requests submitted to the recorded main R thread, `executed` counts requests drained by that thread, and `timeouts` counts requests that were abandoned rather than waiting indefinitely. The `pending_*` and `running_*` columns expose current and maximum queue pressure: pending requests are waiting to be drained by the main R thread, while running requests have been popped by that thread and are executing or collecting. `main_drains`, `main_drain_batches`, and `main_drain_max_batch`

count how often the recorded main R thread attempted queue drains and how many queued requests were handled in non-empty drain waves. `pending_timeout_ms` is the configured native pending-request timeout. Running requests borrow DuckDB callback-frame input/output storage, so running-timeout cancellation is intentionally not supported and is reported via `running_timeout_supported = FALSE`. This is a runtime queue summary; for per-scalar-UDF execution detail such as selected evaluator, Arrow IPC waves, direct `arrow_c` input snapshots, and owned result-chunk counters, use `rducks_explain_udf()`.

Usage

```
rducks_inproc_stats(con)
```

Arguments

`con`	A `duckdb_connection`.

Value

A one-row data frame with queue diagnostic columns.

Examples

```
db <- duckdb::dbConnect(duckdb::duckdb(config = list(allow_unsigned_extensions = "true")))
rducks_enable(db)
rducks_inproc_stats(db)
rducks_release(db)
DBI::dbDisconnect(db)
```

---

rducks_interval *Construct DuckDB INTERVAL values*

Description

DuckDB intervals have three independent components: months, days, and microseconds. This class preserves those components instead of collapsing an interval to a single duration.

Usage

```
rducks_interval(months = 0L, days = 0L, micros = 0L)
```

Arguments

`months`	Integer month components.
`days`	Integer day components.
`micros`	Integer microsecond components. Values outside R's exact numeric range should be supplied as character strings.

Value

Object of class `rducks_interval`.

Examples

```
rducks_interval(months = 1L, days = 15L, micros = 0L)
rducks_interval(days = c(1L, 2L, NA_integer_))
```

`rducks_ipc_workers` *List Rducks-managed IPC workers*

Description

Lists the local Rducks NNG providers currently known to this R process. These are the managed workers used by `arrow_ipc` + `multiprocess_parallel` scalar-UDF execution plans when `ipc_endpoints` is not supplied. Caller-supplied external endpoints are shown as external providers.

Usage

```
rducks_ipc_workers(con = NULL, ping = FALSE, timeout = 1)
```

Arguments

`con`	Optional DuckDB connection. When supplied, only providers attached to that connection's Rducks runtime token are listed. With `NULL`, all providers known to this R process are listed.
`ping`	Logical scalar. If `TRUE`, send a lightweight NNG ping to every listed endpoint and report ok or the ping error.
`timeout`	Positive timeout in seconds used for `ping = TRUE`.

Value

A data frame with one row per configured worker endpoint.

Examples

```
db <- duckdb::dbConnect(duckdb::duckdb(config = list(allow_unsigned_extensions = "true")))
rducks_enable(db)
rducks_ipc_workers(db)
rducks_release(db)
DBI::dbDisconnect(db)
```

rducks_list_udfs *List registered Rducks scalar UDFs*

Description

Returns one row per DuckDB scalar UDF registered through `rducks_register_scalar_udf()` in the current DuckDB database runtime, including the same registration metadata and native counters as `rducks_explain_udf()`. This is an Rducks scalar-UDF registry view, not a complete DuckDB catalog listing: aggregate functions, table functions, functions registered by other extensions, and raw SQL functions are not included. Because DuckDB's function catalog is database scoped, sibling DBI connections to the same database runtime share this view.

Usage

```
rducks_list_udfs(con)
```

Arguments

`con`	A `duckdb_connection` with Rducks enabled.

Value

A data frame with one row per Rducks scalar UDF registered on `con`.

Examples

```
db <- duckdb::dbConnect(duckdb::duckdb(config = list(allow_unsigned_extensions = "true")))
rducks_enable(db, threads = "single")
rducks_register_scalar_udf(db, "my_fn", function(x) x + 1L,
  args = list(INTEGER), returns = INTEGER)
rducks_list_udfs(db)
rducks_release(db)
DBI::dbDisconnect(db)
```

rducks_mode_semantics *Describe Rducks scalar-UDF evaluation mode semantics*

Description

`rducks_mode_semantics()` is the package-level schema for Rducks evaluation modes used by DuckDB scalar UDFs registered with `rducks_register_scalar_udf()`. This is distinct from DuckDB function kind (scalar, aggregate, or table) and from Rducks execution plans. `mode = "scalar"` calls the R function once for each DuckDB row. `mode = "vectorized"` calls the R function once per DuckDB chunk with one R vector/list-column per declared or dynamically bound argument. Vectorized mode is exposed for `arrow_r`, direct `arrow_c`, and worker-provider `arrow_ipc` plans.

Usage

```
rducks_mode_semantics(mode = NULL)
```

Arguments

mode	Optional character vector of scalar-UDF evaluation mode names. When NULL, all known modes are returned.

Value

A data frame describing status, call granularity, input and return shape, NULL handling, length checks, error behavior, threading, and copy semantics for each scalar-UDF evaluation mode.

Examples

```
rducks_mode_semantics()
rducks_mode_semantics("scalar")
rducks_mode_semantics("vectorized")
```

---

rducks_native_execution_backend

Inspect the native Rducks execution backend

Description

Returns the backend currently recorded in the native database-scoped runtime. This is a diagnostic cross-check for rducks_current_execution_plan(), whose value is the R-side default plan for future registrations through this connection.

Usage

```
rducks_native_execution_backend(con)
```

Arguments

con	A duckdb_connection already enabled with rducks_enable().

Value

Character scalar backend name: "single", "concurrent_inproc", or "multiprocess_parallel".

Examples

```
db <- duckdb::dbConnect(duckdb::duckdb(config = list(allow_unsigned_extensions = "true")))
rducks_enable(db)
rducks_native_execution_backend(db)
rducks_release(db)
DBI::dbDisconnect(db)
```

rducks_query_stream *Stream a DuckDB query in batches*

Description

Opens a connection-bound query stream with explicit `next_batch()` and `close()` methods. The query itself is executed by the Rducks DuckDB extension using DuckDB's native streaming result and data-chunk APIs; each fetched DuckDB chunk is exported through DuckDB Arrow C Data. Rducks can either return the owned nanoarrow record-batch object directly or materialize it with the package's Rducks/nanoarrow helpers. This is an R-side result/session API; it is not inferred from scalar UDF IPC behavior and does not use the R-backed SQL table function path. Because execution uses a dedicated extension-owned DuckDB connection, database-scoped objects are visible but temporary tables/views that exist only on the caller's DBI connection are not part of the stream query scope. That dedicated stream connection is separate from the extension connection used for dynamic scalar/table/aggregate registration; a caller connection currently supports one active native query stream at a time. Delivery into R runs on the recorded R thread: even record-batch mode creates R external-pointer objects and installs nanoarrow finalizers, so Rducks does not call R/nanoarrow code from arbitrary DuckDB worker threads.

Usage

```
rducks_query_stream(
  con,
  sql,
  batch_size = 1024L,
  format = c("data.frame", "record_batch", "nanoarrow")
)
```

Arguments

`con`	A `duckdb_connection` with Rducks enabled.
`sql`	SQL query string.
`batch_size`	Maximum number of rows returned by `next_batch()` when its `n` argument is `NULL`. DuckDB may fetch a larger native chunk internally; Rducks buffers any remainder for later `next_batch()` calls.
`format`	Default batch representation. `"data.frame"` materializes batches to base R data frames. `"record_batch"` returns the owned `nanoarrow_array` record batch directly. `"nanoarrow"` is accepted as an alias for `"record_batch"`.

Details

`next_batch()` returns the next batch or `NULL` at end-of-stream. With `format = "data.frame"` it returns a base R data-frame batch. With `format = "record_batch"` it returns a `nanoarrow_array` struct array with an attached `nanoarrow_schema`; nanoarrow's R finalizer owns the Arrow C Data release callbacks, so callers can materialize later without Rducks copying the batch to R vectors first. Returned batches carry the stream's DuckDB/nanoarrow schema as the `"rducks_nanoarrow_schema"`

attribute. close() clears the native streaming result; it is safe to call more than once. A finalizer also closes unclosed streams, and rducks_release(con) closes streams registered on that connection before detaching connection-local state.

Value

Object of class rducks_query_stream with next_batch(n = NULL, format = NULL), close(), is_closed(), schema, and prototype fields.

Examples

```
db <- duckdb::dbConnect(duckdb::duckdb(config = list(allow_unsigned_extensions = "true")))
rducks_enable(db)
stream <- rducks_query_stream(db, "SELECT 1 AS n UNION ALL SELECT 2")
stream$next_batch()
stream$close()
rducks_release(db)
DBI::dbDisconnect(db)
```

---

rducks_register_aggregate

Register an R aggregate function in DuckDB

Description

Registers an R-backed DuckDB aggregate. The aggregate state is an arbitrary R object, not a serialized raw vector. Rducks stores a preserved reference to the state object inside the native DuckDB aggregate state and passes that same object back to later R callbacks. Returning NULL means "empty/no state"; use a wrapper such as list(value = NULL) if NULL itself must be represented as a non-empty state.

Usage

```
rducks_register_aggregate(
  con,
  name,
  update = NULL,
  finalize = NULL,
  args,
  returns,
  combine = NULL,
  null_handling = c("default", "special"),
  copy = NULL,
  copy_chunk = NULL,
  update_chunk = NULL,
  combine_chunk = NULL,
  finalize_chunk = NULL
)
```

Arguments

con	A duckdb_connection.
name	SQL aggregate function name.
update	Optional row-wise R function called as update(state, ...); may return any R object state or NULL.
finalize	Optional row-wise R function called as finalize(state); must return a scalar compatible with returns or NULL for SQL NULL.
args	Input type specification. Use exported DuckDB-style descriptors such as INTEGER, DOUBLE, or VARCHAR.
returns	Return type specification.
combine	Optional R function called as combine(left, right) when two non-NULL partial states must be merged. It may return any R object state or NULL.
null_handling	Either "default" to skip rows with top-level NULL inputs, or "special" to pass missing values to update callbacks.
copy	Optional R function called as copy(state) when DuckDB needs to place a non-NULL partial state into an empty target state during combine. When omitted, Rducks preserves another reference to the same R object.
copy_chunk	Optional vectorized R function called as copy_chunk(states) with a list of states to copy. It must return a list of replacement states of the same length. It takes precedence over copy().
update_chunk	Optional vectorized R function called as update_chunk(states, group_id, ...), where states is a list of current R state objects, group_id maps each input row to an element of states, and the remaining arguments are full R input vectors. It must return a list of replacement states with the same length as states.
combine_chunk	Optional vectorized R function called as combine_chunk(left_states, right_states), where both arguments are lists of R state objects or NULL. It must return a list of states of the same length.
finalize_chunk	Optional vectorized R function called as finalize_chunk(states), where states is a list of R state objects or NULL. It must return one result per state as either a vector or list.

Details

The row-wise API calls update(state, ...) for each selected input row and finalize(state) for each output state. The vectorized update API calls update_chunk(states, group_id, ...) once per DuckDB input chunk. states is a list of the distinct aggregate-state objects referenced by that chunk, and group_id is an integer vector with one entry per input row: 0L means the row was skipped by default NULL handling, otherwise the value is a one-based index into states. The remaining arguments are full, unsliced R vectors for the aggregate inputs. update_chunk() must return a list of replacement states with the same length as states. combine_chunk(left, right) receives lists of state objects for partial-state merging and must return a list with one merged state per pair. finalize_chunk(states) must return a vector or list with one scalar result per output state. Chunk callbacks take precedence over row-wise callbacks.

This API is deliberately serialized. Registration requires `rducks_enable(con, threads = "single")` or equivalent `external_threads=1` plus `PRAGMA threads=1`, and execution rejects attempts to call R from non-calling DuckDB worker threads. If DuckDB combines partial states and the target state is empty, Rducks preserves another reference to the source R object rather than serializing or deep-copying it. Use `copy` or `copy_chunk` when empty-target combine must create independent mutable state. Merging two non-`NULL` states requires either `combine(left, right)` or `combine_chunk(left, right)` and must still run on the recorded R thread.

With `null_handling = "default"`, rows with any top-level SQL `NULL` input do not call `update()` or appear in a positive `group_id` entry for `update_chunk()`. Groups with no non-NULL rows therefore pass `NULL` to `finalize()` or `finalize_chunk()`. With `null_handling = "special"`, update callbacks receive the declared type's R missing-value shape for NULL inputs.

Value

Object of class `rducks_aggregate_registration` containing the connection and normalized aggregate signature. The aggregate remains registered in DuckDB even if this object is discarded.

Examples

```
db <- duckdb::dbConnect(duckdb::duckdb(config = list(allow_unsigned_extensions = "true")))
rducks_enable(db, threads = "single")
rducks_register_aggregate(
  db, "my_sum",
  update = function(state, x) if (is.null(state)) x else state + x,
  finalize = function(state) if (is.null(state)) 0L else state,
  args = list(INTEGER), returns = INTEGER
)
DBI::dbGetQuery(db, "SELECT my_sum(x) FROM (VALUES (1), (2), (3)) t(x)")
rducks_release(db)
DBI::dbDisconnect(db)
```

---

rducks_register_scalar_udf

Register an R-backed DuckDB scalar UDF

Description

Registers an R function as a DuckDB scalar SQL function using the loaded Rducks extension. In DuckDB terminology this is a scalar UDF: it returns one SQL value for each logical input row. The `mode` argument is Rducks' evaluation mode for that scalar UDF, not a DuckDB function kind: `"scalar"` calls the R function once per logical row, while `"vectorized"` calls the R function once per DuckDB chunk with vector/list-column inputs.

Usage

```
rducks_register_scalar_udf(
  con,
  name,
  fun,
  args,
  returns,
  mode = "scalar",
  null_handling = c("default", "special"),
  exception_handling = c("rethrow", "return_null"),
  side_effects = FALSE
)
```

Arguments

con	A duckdb_connection.
name	SQL function name.
fun	R function.
args	Optional argument type specification. If omitted, Rducks registers a dynamic-varargs DuckDB scalar function. DuckDB resolves the concrete argument logical types at bind time, and Rducks materializes those inputs with the same typed semantics used for an explicit args = ... signature across scalar/vectorized evaluation and supported arrow_r, arrow_c, and arrow_ipc execution plans. Use explicit NULL for a zero-argument scalar UDF. Otherwise use exported DuckDB-style type descriptors such as INTEGER, DOUBLE, GEOMETRY, VARIANT, INTEGER[], INTEGER[3], STRUCT(a = INTEGER), or MAP(VARCHAR, INTEGER). VARIANT signatures require a DuckDB runtime whose C API exposes VARIANT logical types, and are not supported by the direct arrow_c marshalling path yet.
returns	Return type specification.
mode	Rducks evaluation mode for this DuckDB scalar UDF. "scalar" calls the R function once per DuckDB row. "vectorized" calls the R function once per DuckDB chunk with one R vector/list-column per declared or dynamically bound argument.
null_handling	Either "default" for NULL-in/NULL-out without calling the R function, or "special" to call the R function with the declared type's missing-value shape for NULL inputs (for example typed NA for ordinary scalar types and NULL for exact/exotic, binary, and composite values).
exception_handling	Either "rethrow" to report user R function errors to DuckDB, or "return_null" to turn user R function errors into SQL NULL values. Return type-checking and marshalling errors still abort the query.
side_effects	Logical scalar. Use TRUE for functions with randomness, counters, I/O, mutation, or other side effects so DuckDB does not treat the function as pure.

Details

Registration requires `external_threads=1` plus `PRAGMA threads=1` so native registration and the default scalar evaluation path stay on the calling R thread. The active `rducks_execution_plan()` selects and freezes the marshalling/concurrency implementation for this registration; unsupported plan/evaluation-mode/type combinations fail instead of switching engines. If a later call registers the same SQL name/signature, the callable implementation is replaced in the shared DuckDB database catalog rather than being tied to the registering DBI connection. Choose the desired execution plan before registration with `rducks_set_execution_plan()`; the selected evaluator/marshalling metadata is then stored with the native catalog entry. R-backed UDF registrations are live DuckDB-runtime catalog entries, not durable schema objects: they are visible to sibling connections while the same DuckDB database runtime remains open, but a file-backed database must be enabled and registered again after it is fully closed and reopened. For `arrow_ipc` plans, the UDF closure and discovered globals are copied once to each NNG worker in the shared provider pool and retained for that pool's lifetime.

Value

Object of class `rducks_scalar_udf_registration` containing the connection, normalized signature, and registration options. The scalar UDF remains registered in DuckDB even if this object is discarded.

Examples

```
db <- duckdb::dbConnect(duckdb::duckdb(config = list(allow_unsigned_extensions = "true")))
rducks_enable(db, threads = "single")
rducks_register_scalar_udf(db, "my_double", function(x) x * 2L,
  args = list(INTEGER), returns = INTEGER)
DBI::dbGetQuery(db, "SELECT my_double(3)")
rducks_release(db)
DBI::dbDisconnect(db)
```

---

rducks_register_table *Register an R table function in DuckDB*

Description

Registers an R-backed DuckDB table function. The registered SQL table function infers its positional SQL argument count from `formals(fun)` and registers those arguments with DuckDB's dynamic `ANY` type. During DuckDB's bind phase, Rducks converts the actual SQL argument values to R scalars/lists and calls `fun(...)` on the recorded calling R thread. `fun()` may return either a finite data frame/named list or a `rducks_table_stream()` object.

Usage

```
rducks_register_table(con, name, fun, chunk_size = 1024L)
```

Arguments

con	A duckdb_connection.
name	SQL table function name.
fun	R function returning a data frame, named list of columns, or rducks_table_stream(). Its finite formal argument count defines the SQL positional argument count; each positional argument is registered as DuckDB ANY and converted at bind time from the actual SQL value.
chunk_size	Maximum number of rows emitted per DuckDB output chunk. Must be an integer from 1 to 1024.

Details

For finite results, Rducks imports the full result into one DuckDB data chunk during bind and then emits row batches during scan. For streaming results, bind uses only the stream prototype to define the DuckDB schema; scan calls next_batch() repeatedly and imports one returned batch at a time. Both paths honor DuckDB projection pushdown, so unreferenced columns are not copied from imported chunks into DuckDB output chunks.

This is intentionally separate from DuckDB scalar-UDF registration through rducks_register_scalar_udf(): table functions have their own bind/init/scan state, bind-time dynamic schemas, and positional SQL arguments fixed by the R function's finite formal argument count. Variadic ... arguments are not supported. If you already have a static R data frame to expose as a virtual table, prefer duckdb::duckdb_register(); DuckDB's R package routes that through its native data-frame scan path. Use rducks_enable(con, threads = "single") or otherwise set external_threads=1 plus PRAGMA threads=1 before registration and execution; worker-thread calls into R are rejected.

Value

Object of class rducks_table_registration containing the connection and normalized table signature. The table function remains registered in DuckDB even if this object is discarded.

Examples

```
db <- duckdb::dbConnect(duckdb::duckdb(config = list(allow_unsigned_extensions = "true")))
rducks_enable(db, threads = "single")
rducks_register_table(db, "my_table", function() data.frame(x = 1:3))
DBI::dbGetQuery(db, "SELECT * FROM my_table()")
rducks_release(db)
DBI::dbDisconnect(db)
```

---

rducks_release *Detach Rducks connection-local state*

Description

Detaches Rducks' connection-local R state for `con`. This clears the current default execution plan and releases this connection's R-side runtime anchor. It does not drop DuckDB catalog functions, unregister scalar UDFs, or release native-owned R closures that are still referenced by database-scoped catalog metadata. If sibling DBI connections are attached to the same DuckDB database runtime, their database-scoped Rducks registration metadata remains visible. For `arrow_ipc` + `multiprocess_parallel`, releasing the last Rducks attachment to a runtime also closes native client pools for Rducks-launched local workers and stops those local mirai/NNG workers. If `ipc_endpoints` was supplied, those URLs name user-owned worker processes; Rducks does not send stop requests to them during release. For file-backed databases, releasing the last attachment also closes Rducks' extension-owned DuckDB connections, which lets the DuckDB file be closed and reopened in the same R process on platforms with strict file locking.

Usage

```
rducks_release(con)

rducks_detach(con)
```

Arguments

`con`	A `duckdb_connection`.

Details

Rducks deliberately keeps the plain `duckdb_connection` object and does not override DBI's `dbDisconnect()` method. Call `rducks_release(con)` explicitly before `DBI::dbDisconnect(con)` when you want deterministic connection-local Rducks cleanup; weak-reference finalizers provide only best-effort cleanup if the connection object is garbage-collected.

Call `rducks_enable()` again before using `con` for further Rducks registrations or connection-local plan changes.

Value

`con`, invisibly.

Examples

```
db <- duckdb::dbConnect(duckdb::duckdb(config = list(allow_unsigned_extensions = "true")))
rducks_enable(db)
rducks_release(db)
DBI::dbDisconnect(db)
```

rducks_release_stats *Inspect preserved-object release counters*

Description

Returns process-local diagnostics for preserved R objects that native DuckDB catalog metadata could not release immediately because destruction happened off the recorded main R thread. Safe main-thread drain points include rducks_enable(), rducks_release(), rducks_register_scalar_udf(), scalar-UDF execution, and metadata/stat queries.

Usage

```
rducks_release_stats(con)
```

Arguments

con	A duckdb_connection.

Value

A one-row data frame with queued, released, failed, and pending counters.

Examples

```
db <- duckdb::dbConnect(duckdb::duckdb(config = list(allow_unsigned_extensions = "true")))
rducks_enable(db)
rducks_release_stats(db)
rducks_release(db)
DBI::dbDisconnect(db)
```

rducks_reset_udf_counters *Reset Rducks scalar-UDF counters*

Description

Resets native per-scalar-UDF diagnostic counters without unregistering any DuckDB catalog function. Current liveness gauges such as pending/in-flight counts are preserved; their max fields are reset to the current values.

Usage

```
rducks_reset_udf_counters(con, name = NULL)
```

Arguments

con	A duckdb_connection with Rducks enabled.
name	Optional SQL scalar-UDF function name registered with rducks_register_scalar_udf(). If NULL, reset counters for all native Rducks scalar UDFs in the database runtime.

Value

Invisibly TRUE on success.

Examples

```
db <- duckdb::dbConnect(duckdb::duckdb(config = list(allow_unsigned_extensions = "true")))
rducks_enable(db, threads = "single")
rducks_register_scalar_udf(db, "my_fn", function(x) x + 1L,
  args = list(INTEGER), returns = INTEGER)
rducks_reset_udf_counters(db, "my_fn")
rducks_release(db)
DBI::dbDisconnect(db)
```

---

rducks_runtime_stats *Inspect native runtime registry counters*

Description

Returns process-local diagnostics for database-scoped native runtime entries and their extension-owned DuckDB connections. The con argument is only the enabled DuckDB connection used to reach the diagnostic SQL functions; the counters are process-global, not scoped only to con. active_entries means entries whose stored database handle has not been marked as a stale registry alias, and stale_entries means entries retained only to avoid reusing an old raw database address. DuckDB's C extension API does not currently provide a clean database-close callback for this package, so these counters are accounting diagnostics rather than deterministic lifetime guarantees. connections_current and native_release_supported are derived R-side summary fields. For file-backed databases, Rducks closes extension-owned DuckDB connections when the last Rducks attachment to a runtime is released; the process-local runtime entry itself is retained as inert metadata so catalog destructors and stale database-address detection remain safe.

Usage

```
rducks_runtime_stats(con)
```

Arguments

con	An enabled duckdb_connection.

Value

A one-row data frame with runtime registry and connection counters.

Examples

```
db <- duckdb::dbConnect(duckdb::duckdb(config = list(allow_unsigned_extensions = "true")))
rducks_enable(db)
rducks_runtime_stats(db)
rducks_release(db)
DBI::dbDisconnect(db)
```

---

rducks_set_execution_plan

Set the Rducks execution plan for a connection

Description

Stores the R-side default execution plan used by subsequent `rducks_register_scalar_udf()` calls through this connection and updates the native runtime backend needed by that plan. Scalar-UDF registration still defines Rducks evaluation semantics such as scalar row calls versus vectorized chunk calls, declared types, NULL handling, error handling, and side effects. The selected evaluator/marshalling for an already-registered scalar UDF remains frozen in its database-catalog metadata.

Usage

```
rducks_set_execution_plan(
  con,
  plan = rducks_execution_plan(),
  threads = NULL,
  external_threads = NULL
)
```

Arguments

`con`	A `duckdb_connection` already enabled with `rducks_enable()`.
`plan`	An `rducks_execution_plan()` object.
`threads`	Optional positive integer to set with `PRAGMA threads`.
`external_threads`	Optional positive integer to set with `SET external_threads`. Use `NULL` to restore/keep the previous setting after Rducks briefly forces single-thread SQL execution to update its native backend on the recorded main R thread. For actual DuckDB worker concurrency, keep this smaller than `threads`.

Value

con, invisibly.

Examples

```
db <- duckdb::dbConnect(duckdb::duckdb(config = list(allow_unsigned_extensions = "true")))
rducks_enable(db)
rducks_set_execution_plan(db, rducks_execution_plan("arrow_c", "serial"))
rducks_release(db)
DBI::dbDisconnect(db)
```

---

rducks_table_stream *Create a streaming result for an Rducks table function*

Description

Return this object from a function registered with rducks_register_table() to expose a finite table without materializing all rows during DuckDB bind. The prototype supplies the output column names and types. During scan, Rducks repeatedly calls next_batch(n) and imports each returned data frame, named list, nanoarrow_array, or one-batch nanoarrow_array_stream. Return NULL from next_batch() to signal end-of-stream.

Usage

```
rducks_table_stream(
  prototype,
  next_batch,
  close = NULL,
  cardinality = NA_real_,
  exact = FALSE
)
```

Arguments

prototype	Data frame or named list whose column names and R types define the stream schema. A zero-row prototype is usually appropriate.
next_batch	Function called as next_batch(n) or next_batch() if it has no formal arguments. It must return the next batch or NULL for EOF.
close	Optional cleanup function.
cardinality	Optional non-negative row count, or NA when unknown.
exact	Whether cardinality is exact rather than an estimate.

Details

close, when supplied, is called at most once when the stream reaches EOF. Rducks also tries to close unreached EOF streams when DuckDB releases the native bind state on the recorded R thread, and a finalizer provides eventual best-effort cleanup if the stream object is later garbage-collected. Use it to release file handles, sockets, iterators, or other producer-side resources. cardinality is optional scan metadata; set exact = TRUE only when the stream will emit exactly that many rows.

Value

Object of class rducks_table_stream.

Examples

```
rows <- data.frame(x = 1:3)
i <- 0L
stream <- rducks_table_stream(
  prototype = rows[0, , drop = FALSE],
  next_batch = function(n) { i <<- i + 1L; if (i > 1L) NULL else rows }
)
stream
```

---

rducks_type_normalize *Normalize an Rducks type token*

Description

Character input is limited to canonical scalar tokens such as i32, f64, and varchar. Composite, DECIMAL, ENUM, and UNION types are represented by constructed rducks_type descriptors rather than quoted type strings.

Usage

```
rducks_type_normalize(x)
```

Arguments

x	Character scalar type token or a rducks_type descriptor.

Value

Canonical scalar token for character input, or the descriptor's wire token for a rducks_type.

Examples

```
rducks_type_normalize("i32")
rducks_type_normalize(INTEGER)
rducks_type_normalize(LIST(VARCHAR))
```

rducks_type_objects *Rducks DuckDB type descriptors and constructors*

Description

Use these descriptors and constructors in rducks_register_scalar_udf() and rducks_register_aggregate() to avoid quoted type specifications. Examples include args = INTEGER, args = c(INTEGER, DOUBLE), args = INTEGER[], args = INTEGER[3], args = STRUCT(a = INTEGER, b = VARCHAR), and args = MAP(VARCHAR, INTEGER).

Usage

```
rducks_is_type(x)

BOOLEAN

TINYINT

UTINYINT

SMALLINT

USMALLINT

INTEGER

UINTEGER

BIGINT

UBIGINT

FLOAT

DOUBLE

VARCHAR

BLOB

GEOMETRY

VARIANT

DATE

TIME
```

```
TIMESTAMP

HUGEINT

UHUGEINT

UUID

INTERVAL

BIT

DECIMAL(width, scale = 0L)

ENUM(levels)

UNION(...)

LIST(type)

ARRAY(type, size)

MAP(key, value)

STRUCT(...)
```

Arguments

x	Object to test with rducks_is_type().
width, scale	DuckDB decimal width and scale for DECIMAL().
levels	Character vector of enum dictionary values for ENUM().
...	Named field types for STRUCT()/UNION() or descriptors for c().
type	Child type for LIST() or ARRAY().
size	Fixed array size for ARRAY().
key, value	Key and value types for MAP().

Format

An object of class Rducks::rducks_bool_type (inherits from Rducks::rducks_logical_scalar_type, Rducks::rducks_scalar_type, Rducks::rducks_type, list, S7_object) of length 7.

An object of class Rducks::rducks_i8_type (inherits from Rducks::rducks_r_integer_scalar_type, Rducks::rducks_scalar_type, Rducks::rducks_type, list, S7_object) of length 7.

An object of class Rducks::rducks_u8_type (inherits from Rducks::rducks_r_integer_scalar_type, Rducks::rducks_scalar_type, Rducks::rducks_type, list, S7_object) of length 7.

An object of class Rducks::rducks_i16_type (inherits from Rducks::rducks_r_integer_scalar_type, Rducks::rducks_scalar_type, Rducks::rducks_type, list, S7_object) of length 7.

An object of class Rducks::rducks_u16_type (inherits from Rducks::rducks_r_integer_scalar_type, Rducks::rducks_scalar_type, Rducks::rducks_type, list, S7_object) of length 7.

An object of class Rducks::rducks_i32_type (inherits from Rducks::rducks_r_integer_scalar_type, Rducks::rducks_scalar_type, Rducks::rducks_type, list, S7_object) of length 7.

An object of class Rducks::rducks_u32_type (inherits from Rducks::rducks_r_numeric_scalar_type, Rducks::rducks_scalar_type, Rducks::rducks_type, list, S7_object) of length 7.

An object of class Rducks::rducks_i64_type (inherits from Rducks::rducks_exact_integer_scalar_type, Rducks::rducks_scalar_type, Rducks::rducks_type, list, S7_object) of length 7.

An object of class Rducks::rducks_u64_type (inherits from Rducks::rducks_exact_integer_scalar_type, Rducks::rducks_scalar_type, Rducks::rducks_type, list, S7_object) of length 7.

An object of class Rducks::rducks_f32_type (inherits from Rducks::rducks_floating_scalar_type, Rducks::rducks_r_numeric_scalar_type, Rducks::rducks_scalar_type, Rducks::rducks_type, list, S7_object) of length 7.

An object of class Rducks::rducks_f64_type (inherits from Rducks::rducks_floating_scalar_type, Rducks::rducks_r_numeric_scalar_type, Rducks::rducks_scalar_type, Rducks::rducks_type, list, S7_object) of length 7.

An object of class Rducks::rducks_varchar_type (inherits from Rducks::rducks_character_scalar_type, Rducks::rducks_scalar_type, Rducks::rducks_type, list, S7_object) of length 7.

An object of class Rducks::rducks_blob_type (inherits from Rducks::rducks_binary_scalar_type, Rducks::rducks_scalar_type, Rducks::rducks_type, list, S7_object) of length 7.

An object of class Rducks::rducks_geometry_type (inherits from Rducks::rducks_binary_scalar_type, Rducks::rducks_scalar_type, Rducks::rducks_type, list, S7_object) of length 7.

An object of class Rducks::rducks_variant_type (inherits from Rducks::rducks_scalar_type, Rducks::rducks_type, list, S7_object) of length 7.

An object of class Rducks::rducks_date_type (inherits from Rducks::rducks_temporal_scalar_type, Rducks::rducks_scalar_type, Rducks::rducks_type, list, S7_object) of length 7.

An object of class Rducks::rducks_time_type (inherits from Rducks::rducks_temporal_scalar_type, Rducks::rducks_scalar_type, Rducks::rducks_type, list, S7_object) of length 7.

An object of class Rducks::rducks_timestamp_type (inherits from Rducks::rducks_temporal_scalar_type, Rducks::rducks_scalar_type, Rducks::rducks_type, list, S7_object) of length 7.

An object of class Rducks::rducks_hugeint_type (inherits from Rducks::rducks_exact_integer_scalar_type, Rducks::rducks_scalar_type, Rducks::rducks_type, list, S7_object) of length 7.

An object of class Rducks::rducks_uhugeint_type (inherits from Rducks::rducks_exact_integer_scalar_type, Rducks::rducks_scalar_type, Rducks::rducks_type, list, S7_object) of length 7.

An object of class Rducks::rducks_uuid_type (inherits from Rducks::rducks_uuid_scalar_type, Rducks::rducks_scalar_type, Rducks::rducks_type, list, S7_object) of length 7.

An object of class Rducks::rducks_interval_type (inherits from Rducks::rducks_interval_scalar_type, Rducks::rducks_scalar_type, Rducks::rducks_type, list, S7_object) of length 7.

An object of class Rducks::rducks_bit_type (inherits from Rducks::rducks_binary_scalar_type, Rducks::rducks_scalar_type, Rducks::rducks_type, list, S7_object) of length 7.

Value

A formal S7 rducks_type descriptor, or a rducks_type_list from c().

Examples

```
INTEGER
rducks_is_type(INTEGER)
rducks_is_type("not a type")
LIST(VARCHAR)
ARRAY(INTEGER, 3L)
STRUCT(a = INTEGER, b = VARCHAR)
MAP(VARCHAR, DOUBLE)
DECIMAL(10L, 2L)
ENUM(c("small", "medium", "large"))
UNION(i = INTEGER, s = VARCHAR)
c(INTEGER, DOUBLE, VARCHAR)
```

`rducks_type_token` *Rducks type descriptor helpers*

Description

These generic helpers expose the formal DuckDB type descriptor carried by `rducks_type` descriptors such as `INTEGER`, `INTEGER[]`, `STRUCT(...)`, `DECIMAL(...)`, `ENUM(...)`, and `UNION(...)`.

Usage

```
rducks_type_token(x, ...)

rducks_type_sql(x, ...)

rducks_type_kind(x, ...)

rducks_type_children(x, ...)

rducks_type_child_names(x, ...)

rducks_type_size(x, ...)

rducks_type_parameters(x, ...)
```

Arguments

`x`	A `rducks_type` descriptor.
`...`	Reserved for methods.

Value

`rducks_type_token()` returns the internal wire token; `rducks_type_sql()` returns the DuckDB SQL spelling; `rducks_type_kind()` returns the descriptor kind; child and parameter helpers return descriptor metadata.

rducks_ubigint *Construct exact DuckDB UBIGINT values*

Description

Values are stored as canonical unsigned decimal strings.

Usage

```
rducks_ubigint(x = character())
```

Arguments

x	Numeric, integer, or character vector of whole unsigned numbers.

Value

Character vector with class `rducks_ubigint`.

Examples

```
rducks_ubigint(0:2)
rducks_ubigint("18446744073709551615")
```

rducks_uhugeint *Construct exact DuckDB UHUGEINT values*

Description

Values are stored as canonical unsigned decimal strings.

Usage

```
rducks_uhugeint(x = character())
```

Arguments

x	Numeric, integer, or character vector of whole unsigned numbers.

Value

Character vector with class `rducks_uhugeint`.

Examples

```
rducks_uhugeint(0:2)
rducks_uhugeint("340282366920938463463374607431768211455")
```

rducks_union *Construct DuckDB UNION values*

Description

`rducks_union()` represents one tagged union value. The tag should match a DuckDB union member name; `value` is the corresponding R value.

Usage

```
rducks_union(tag, value)
```

Arguments

`tag`	Character scalar union member name.
`value`	R value for that member.

Value

Object of class `rducks_union`.

Examples

```
rducks_union("num", 42L)
rducks_union("str", "hello")
```

rducks_uuid *Construct DuckDB UUID values*

Description

`rducks_uuid()` stores canonical UUID text in a dedicated class. Native UDF marshalling for DuckDB `UUID` is implemented separately from this value class.

Usage

```
rducks_uuid(x = character())
```

Arguments

`x`	Character vector of UUID strings.

Value

Character vector with class `rducks_uuid`.

Examples

```
rducks_uuid("550e8400-e29b-41d4-a716-446655440000")
```

rducks_value_semantics *Describe Rducks NULL, NA, NaN, and Inf semantics*

Description

`rducks_value_semantics()` is the package-level schema for DuckDB scalar-UDF missing and non-finite value handling. It is intended to be rendered directly in README and pkgdown documentation, and to keep the documented NULL/NA/NaN/Inf contract close to the type descriptors used by the marshaller.

Usage

```
rducks_value_semantics(x = NULL)
```

Arguments

`x`	Optional scalar type tokens or constructed `rducks_type` descriptors. When `NULL`, all currently implemented DuckDB scalar-UDF scalar type semantics are returned. Constructed descriptors such as `DECIMAL(10, 2)`, `ENUM(c("a", "b"))`, `UNION(i = INTEGER, s = VARCHAR)`, `INTEGER[]`, `INTEGER[3]`, `STRUCT(a = INTEGER)`, and `MAP(VARCHAR, INTEGER)` can be requested explicitly.

Details

With `null_handling = "default"`, top-level SQL `NULL` inputs short-circuit to SQL `NULL` and the R function is not called. The `special_null_argument` column describes what the R function receives with `null_handling = "special"`.

Return semantics are stated from R back to DuckDB. For DuckDB scalar UDFs, top-level `NULL` returns map to SQL `NULL`; type-specific R `NA` values also map to SQL `NULL` where a missing representation exists. `NaN` and `Inf` are values only for `FLOAT` and `DOUBLE`; integer, date, time, timestamp, and exact Rducks value classes reject non-finite values.

Value

A data frame with one row per requested type descriptor and columns describing SQL NULL input handling, R missing/non-finite return handling, Rducks value-class binary operation behavior, and error semantics.

Examples

```
rducks_value_semantics()
rducks_value_semantics(INTEGER)
rducks_value_semantics(DECIMAL(10L, 2L))
```

rducks_value_type *Generic helpers for Rducks value classes*

Description

These helpers provide a small common interface for Rducks' exact value classes used to represent DuckDB-specific values.

Usage

```
rducks_value_type(x, ...)

rducks_duckdb_literal(x, ...)
```

Arguments

x	A value object.
...	Reserved for methods.

Value

`rducks_value_type()` returns a DuckDB type string.

Examples

```
rducks_value_type(rducks_bigint(1L))
rducks_value_type(rducks_decimal(1.5, width = 10, scale = 2))
rducks_duckdb_literal(rducks_bigint("42"))
rducks_duckdb_literal(rducks_uuid("550e8400-e29b-41d4-a716-446655440000"))
```

rducks_variant *Construct a DuckDB VARIANT storage object*

Description

`VARIANT` values cross the Rducks boundary as DuckDB's typed storage object: a named list with `keys`, `children`, `values`, and `data` fields. Most code receives this object from a VARIANT argument and returns it unchanged or after using DuckDB SQL functions such as `variant_extract()` before crossing into R.

Usage

```
rducks_variant(x)
```

Arguments

x	Named list in DuckDB VARIANT storage shape.

Value

x with class rducks_variant after validation.

Examples

```
# VARIANT storage objects are normally produced by DuckDB at the R boundary.
# rducks_variant() validates the storage shape; constructing one by hand
# requires the full internal DuckDB VARIANT storage layout.
```

---

rducks_with_duckplyr *Evaluate a duckplyr pipeline with dynamic Rducks scalar UDFs*

Description

Registers selected R functions as dynamic-argument Rducks scalar UDFs on a DuckDB connection, rewrites matching calls in a captured duckplyr expression to duckplyr's DuckDB-function escape hatch, and evaluates the rewritten expression. This lets a duckplyr pipeline stay in DuckDB for those calls instead of falling back to dplyr, provided every registered function has an explicit return type.

Usage

```
rducks_with_duckplyr(
  con,
  expr,
  returns,
  env = parent.frame(),
  null_handling = c("default", "special"),
  exception_handling = c("rethrow", "return_null"),
  side_effects = FALSE,
  mode = "scalar"
)

## S3 method for class 'duckdb_connection'
with(
  data,
  expr,
  ...,
  rducks_returns,
  rducks_env = parent.frame(),
  rducks_mode = "scalar"
)
```

Arguments

con	A duckdb_connection with Rducks enabled.
expr	A duckplyr expression or pipeline to evaluate.
returns	Named list or named character vector of return types. Names must be R function names visible from env; values are Rducks type descriptors or scalar type tokens, e.g. list(score_fun = DOUBLE).
env	Evaluation environment for expr and function lookup.
null_handling, exception_handling, side_effects	Passed to rducks_register_scalar_udf().
mode	Rducks scalar-UDF evaluation mode for registered helpers. "scalar" calls the R helper once per row; "vectorized" calls it once per DuckDB chunk and requires a vectorized helper.
data	A duckdb_connection with Rducks enabled.
...	Reserved for future extensions; must be empty.
rducks_returns	Named return-type list for dynamic Rducks UDFs.
rducks_env	Evaluation environment for expr and function lookup.
rducks_mode	Rducks scalar-UDF evaluation mode for helpers registered through with.duckdb_connection().

Details

This helper intentionally requires return-type declarations: DuckDB needs a scalar function's return type during planning even when its input arguments are accepted dynamically. Dynamic arguments are a duckplyr-oriented convenience path that uses nanoarrow's default input conversion. The duckplyr bridge defaults to mode = "scalar" because ordinary R calls in duckplyr SQL expressions are written as row-wise scalar functions. Set mode = "vectorized" only for helpers that accept full vectors/chunks and return a vector of the same length. The selected Rducks execution plan is still taken from con, so arrow_c and arrow_ipc plans can be selected with rducks_set_execution_plan() before evaluating the duckplyr expression. Use explicit args in rducks_register_scalar_udf() when you need Rducks' declared composite, exotic, or special-NULL input semantics.

Value

The value of the evaluated expression.

Index