

Package: hubData (via r-universe)

October 13, 2026

Title Tools for accessing and working with hubverse data

Version 2.2.2

Description A set of utility functions for accessing and working with
forecast and target data from Infectious Disease Modeling Hubs.

License MIT + file LICENSE

Encoding UTF-8

Roxygen list(markdown = TRUE)

RoxygenNote 7.3.2

Config/testthat/edition 3

URL <https://hubverse-org.github.io/hubData>,
<https://github.com/hubverse-org/hubData>

BugReports <https://github.com/hubverse-org/hubData/issues>

Imports arrow ($\geq 17.0.0$), checkmate, cli, dplyr ($\geq 1.1.0$), fs,
hubUtils ($\geq 1.2.0$), lifecycle, magrittr, purrr, rlang,
stringr, tibble, yaml, zoltr ($\geq 1.0.1$)

Suggests covr, curl, dbplyr, digest, duckdb, gert, gh, hubAdmin ($\geq$
1.9.0), jsonlite, knitr, mockery, rmarkdown, testthat ($\geq$
3.3.0), withr

Remotes hubverse-org/hubUtils, hubverse-org/hubAdmin

Config/Needs/website hubverse-org/hubStyle

Depends R (≥ 4.1)

VignetteBuilder knitr

LazyData true

Config/pak/sysreqs cmake git make libicu-dev libuv1-dev libssl-dev libx11-dev

Repository <https://hubverse-org.r-universe.dev>

Date/Publication 2026-07-15 09:22:09 UTC

RemoteUrl <https://github.com/hubverse-org/hubData>

RemoteRef v2.2.2

RemoteSha 842d47aa2c083e2dff3ee870e4957bd2d3dd1232

Contents

arrow_to_r_datatypes	2
as_r_schema	3
coerce_to_hub_schema	5
collect_hub	6
collect_zoltar	7
connect_hub	9
connect_target_oracle_output	12
connect_target_timeseries	15
create_hub_schema	17
create_model_out_submit_tmpl	18
create_oracle_output_schema	20
create_timeseries_schema	21
expand_model_out_val_grid	23
extract_hive_partitions	24
get_s3_bucket_name	25
get_target_data_colnames	26
get_target_file_ext	27
get_target_path	28
gs_bucket	29
is_hive_partitioned_path	29
load_model_metadata	30
print.hub_connection	31
r_to_arrow_datatypes	32
s3_bucket	33
Index	34

arrow_to_r_datatypes *Mapping of Arrow types to base R types*

Description

A named character vector mapping common [arrow::Schema](#) field types (as strings) to their corresponding base R types. This mapping is used to translate or validate column types when working with Parquet files or Arrow datasets, especially for schema inference and compatibility checks.

Usage

```
arrow_to_r_datatypes
```

Format

A named character vector with 8 entries.

Details

Only the safest and most portable Arrow types are supported in the hubverse. Types not present in this mapping should be treated as unsupported.

Arrow type	R type	Notes
bool	logical	
int32	integer	
int64	integer	R supports via Arrow
float	double	Promoted to double in R
double	double	
string	character	
date32[day]	Date	
timestamp[ms]	POSIXct	Safest timestamp format

See Also

[as_r_schema\(\)](#), [arrow_schema_to_string\(\)](#)

<code>as_r_schema</code>	<i>Convert or validate an Arrow schema for compatibility with base R column types</i>
--------------------------	---

Description

These functions help convert or validate an [arrow::Schema](#) object (typically from a Parquet file or Arrow dataset) by translating Arrow types to R equivalents, extracting type strings, or checking for compatibility.

Usage

```
as_r_schema(arrow_schema, call = rlang::caller_env())

arrow_schema_to_string(arrow_schema)

is_supported_arrow_type(arrow_schema)

validate_arrow_schema(arrow_schema, call = rlang::caller_env())
```

Arguments

<code>arrow_schema</code>	An arrow::Schema object, such as one returned by <code>arrow::read_parquet(..., as_data_frame = FALSE)\$schema</code> or <code>arrow::open_dataset(...)\$schema</code> .
<code>call</code>	The calling environment, used for error reporting in <code>validate_arrow_schema()</code> and <code>as_r_schema()</code> (default: caller's environment).

Details

- `as_r_schema()` maps Arrow types to base **R types** (e.g., `"int32" → "integer"`). It throws an error if unsupported column types are present.
- `arrow_schema_to_string()` returns a named character vector of **raw Arrow type strings** (e.g., `"int64"`, `"date32[day]"`) for schema field.
- `is_supported_arrow_type()` returns a named logical vector indicating whether each schema field type is supported.
- `validate_arrow_schema()` throws an error if any fields has an unsupported Arrow type.

For a full list of supported types and their R mappings, see [arrow_to_r_datatypes\(\)](#).

Value

- `as_r_schema()`: A named character vector mapping column names to base R type strings (e.g., `"integer"`, `"double"`, `"logical"`).
- `arrow_schema_to_string()`: A named character vector mapping column names to Arrow type strings.
- `is_supported_arrow_type()`: A named logical vector indicating whether each column is supported.
- `validate_arrow_schema()`: Returns the original schema (invisibly) if all column types are supported; otherwise throws an error.

Examples

```
# Path to a single Parquet file
file_path <- system.file(
  "testhubs/parquet/model-output/hub-baseline/2022-10-01-hub-baseline.parquet",
  package = "hubUtils"
)

# Get schema from the file
file_schema <- arrow::read_parquet(file_path, as_data_frame = FALSE)$schema

# Convert to R types
as_r_schema(file_schema)

# Get raw Arrow type strings
arrow_schema_to_string(file_schema)

# Check which columns are supported
is_supported_arrow_type(file_schema)

# Validate schema (throws error if any unsupported types are present)
validate_arrow_schema(file_schema)

# From a multi-file dataset
dataset_path <- system.file(
  "testhubs/parquet/model-output/hub-baseline",
```

```

    package = "hubUtils"
  )
  ds <- arrow::open_dataset(dataset_path)
  as_r_schema(ds$schema)
  arrow_schema_to_string(ds$schema)
  is_supported_arrow_type(ds$schema)
  validate_arrow_schema(ds$schema)

```

coerce_to_hub_schema *Coerce data.frame/tibble column data types to hub schema data types or character.*

Description

Coerce data.frame/tibble column data types to hub schema data types or character.

Usage

```

coerce_to_hub_schema(
  tbl,
  config_tasks,
  skip_date_coercion = FALSE,
  as_arrow_table = FALSE,
  output_type_id_datatype = c("from_config", "auto", "character", "double", "integer",
    "logical", "Date")
)

coerce_to_character(tbl, as_arrow_table = FALSE)

```

Arguments

tbl a model output data.frame/tibble

config_tasks a list version of the content's of a hub's `tasks.json` config file created using function `hubUtils::read_config()`.

skip_date_coercion Logical. Whether to skip coercing dates. This can be faster, especially for larger `tbls`.

as_arrow_table Logical. Whether to return an arrow table. Defaults to `FALSE`.

output_type_id_datatype character string. One of "from_config", "auto", "character", "double", "integer", "logical", "Date". Defaults to "from_config" which uses the setting in the `output_type_id_datatype` property in the `tasks.json` config file if available. If the property is not set in the config, the argument falls back to "auto" which determines the `output_type_id` data type automatically from the `tasks.json` config file as the simplest data type required to represent all output type ID values across all output types in the

hub. When only point estimate output types (where `output_type_ids` are NA,) are being collected by a hub, the `output_type_id` column is assigned a `character` data type when auto-determined. Other data type values can be used to override automatic determination. Note that attempting to coerce `output_type_id` to a data type that is not valid for the data (e.g. trying to coerce "character" values to "double") will likely result in an error or potentially unexpected behaviour so use with care.

Value

`tbl` with column data types coerced to hub schema data types or character. if `as_arrow_table = TRUE`, output is also converted to arrow table.

Functions

- `coerce_to_hub_schema()`: coerce columns to hub schema data types.
- `coerce_to_character()`: coerce all columns to character

<code>collect_hub</code>	<i>Collect Hub model output data</i>
--------------------------	--------------------------------------

Description

`collect_hub` retrieves data from a `<hub_connection>/<mod_out_connection>` after executing any `<arrow_dplyr_query>` into a local tibble. The function also attempts to convert the output to a `model_out_tbl` class object before returning.

Usage

```
collect_hub(x, silent = FALSE, ...)
```

Arguments

<code>x</code>	a <code><hub_connection>/<mod_out_connection></code> or <code><arrow_dplyr_query></code> object.
<code>silent</code>	Logical. Whether to suppress message generated if conversion to <code>model_out_tbl</code> fails.
<code>...</code>	Further argument passed on to <code>as_model_out_tbl()</code> .

Value

A `model_out_tbl`, unless conversion to `model_out_tbl` fails in which case a tibble is returned.

Examples

```
hub_path <- system.file("testhubs/simple", package = "hubUtils")
hub_con <- connect_hub(hub_path)
# Collect all data in a hub
hub_con |> collect_hub()
# Filter data before collecting
hub_con |>
  dplyr::filter(is.na(output_type_id)) |>
  collect_hub()
# Pass arguments to as_model_out_tbl()
dplyr::filter(hub_con, is.na(output_type_id)) |>
  collect_hub(remove_empty = TRUE)
```

collect_zoltar	<i>Load forecasts from zoltardata.com in hubverse format</i>
----------------	--

Description

`collect_zoltar` retrieves data from a zoltardata.com project and transforms it from Zoltar's native download format into a hubverse one. Zoltar ([documentation here](#)) is a pre-hubverse research project that implements a repository of model forecast results, including tools to administer, query, and visualize uploaded data, along with R and Python APIs to access data programmatically ([zoltr](#) and [zoltpy](#), respectively.) (This `hubData` function is itself implemented using the `zoltr` package.)

Usage

```
collect_zoltar(
  project_name,
  models = NULL,
  timezeros = NULL,
  units = NULL,
  targets = NULL,
  types = NULL,
  as_of = NULL,
  point_output_type = "median"
)
```

Arguments

project_name	A string naming the Zoltar project to load forecasts from. Assumes the host is <code>zoltardata.com</code> .
models	A character vector that specifies the models to query. Must be model abbreviations. Defaults to <code>NULL</code> , which queries all models in the project.
timezeros	A character vector that specifies the timezeros to query. Must be <code>yyyy-mm-dd</code> format. Defaults to <code>NULL</code> , which queries all timezeros in the project.

units	A character vector that specifies the units to query. Must be unit abbreviations. Defaults to NULL, which queries all units in the project.
targets	A character vector that specifies the targets to query. Must be target names. Defaults to NULL, which queries all targets in the project.
types	A character vector that specifies the forecast types to query. Choices are "bin", "point", "sample", "quantile", "mean", and "median". Defaults to NULL, which queries all types in the project. Note: While Zoltar supports "named" and "mode" forecasts, this function ignores them.
as_of	A datetime string that specifies the forecast version. The datetime must include timezone information for disambiguation, without which the query will fail. The datetime parsing function used below (<code>base::strptime</code>) is extremely lenient when it comes to formatting, so please exercise caution. Defaults to NULL to load the latest version.
point_output_type	A string that specifies how to convert zoltar point forecast data to hubverse output type. Must be either "median" or "mean". Defaults to "median".

Details

Zoltar's data model differs from that of the hubverse in a few important ways. While Zoltar's model has the concepts of unit, target, and timezero, hubverse projects have hub-configurable columns, which makes the mapping from the former to the latter imperfect. In particular, Zoltar units translate roughly to hubverse task IDs, Zoltar targets include both the target outcome and numeric horizon in the target name, and Zoltar timezeros map to round ids. Finally, Zoltar's forecast types differ from those of the hubverse. Whereas Zoltar has seven types (bin, named, point, sample, quantile, mean, median, and mode), the hubverse has six (cdf, mean, median, pmf, quantile, sample), only some of which overlap.

Additional notes:

- Requires the user to have a Zoltar account (use the [Zoltar contact page](#) to request one).
- Requires `Z_USERNAME` and `Z_PASSWORD` environment vars to be set to those of the user's Zoltar account.
- While Zoltar supports "named" and "mode" forecasts, this function ignores them.
- Rows with non-numeric values are ignored.
- This function removes `numeric_horizon` mentions from zoltar target names. Target names can contain a maximum of one `numeric_horizon`. Example: "1 wk ahead inc case" -> "wk ahead inc case".
- Querying a large number of rows may cause errors, so we recommend providing one or more filtering arguments (e.g., `models`, `timezeros`, etc.) to limit the result.

Value

A hubverse `model_out_tbl` containing the following columns: "model_id", "timezero", "season", "unit", "horizon", "target", "output_type", "output_type_id", and "value".

Examples

```
## Not run:
df <- collect_zoltar("Docs Example Project")
df <-
  collect_zoltar("Docs Example Project", models = c("docs_mod"),
    timezeros = c("2011-10-16"), units = c("loc1", "loc3"),
    targets = c("pct next week", "cases next week"), types = c("point"),
    as_of = NULL, point_output_type = "mean")

## End(Not run)
```

connect_hub	<i>Connect to model output data.</i>
-------------	--------------------------------------

Description

Connect to data in a model output directory through a Modeling Hub or directly. Data can be stored in a local directory or in the cloud on AWS or GCS.

Usage

```
connect_hub(
  hub_path,
  file_format = c("csv", "parquet", "arrow"),
  output_type_id_datatype = c("from_config", "auto", "character", "double", "integer",
    "logical", "Date"),
  partitions = list(model_id = arrow::utf8()),
  skip_checks = TRUE,
  na = c("NA", ""),
  ignore_files = NULL
)

connect_model_output(
  model_output_dir,
  file_format = c("csv", "parquet", "arrow"),
  partition_names = "model_id",
  schema = NULL,
  skip_checks = FALSE,
  na = c("NA", ""),
  ignore_files = NULL
)
```

Arguments

hub_path	Either a character string path to a local Modeling Hub directory or an object of class <code><SubTreeFileSystem></code> created using functions <code>s3_bucket()</code> or <code>gs_bucket()</code> by providing a string S3 or GCS bucket name or path to
----------	---

a Modeling Hub directory stored in the cloud. For more details consult the [Using cloud storage \(S3, GCS\)](#) in the `arrow` package. The hub must be fully configured with valid `admin.json` and `tasks.json` files within the `hub-config` directory.

<code>file_format</code>	The file format model output files are stored in. For connection to a fully configured hub, accessed through <code>hub_path</code> , <code>file_format</code> is inferred from the hub's <code>file_format</code> configuration in <code>admin.json</code> and is ignored by default. If supplied, it will override hub configuration setting. Multiple formats can be supplied to <code>connect_hub</code> but only a single file format can be supplied to <code>connect_model_output</code> .
<code>output_type_id_datatype</code>	character string. One of "from_config", "auto", "character", "double", "integer", "logical", "Date". Defaults to "from_config" which uses the setting in the <code>output_type_id_datatype</code> property in the <code>tasks.json</code> config file if available. If the property is not set in the config, the argument falls back to "auto" which determines the <code>output_type_id</code> data type automatically from the <code>tasks.json</code> config file as the simplest data type required to represent all output type ID values across all output types in the hub. When only point estimate output types (where <code>output_type_ids</code> are NA,) are being collected by a hub, the <code>output_type_id</code> column is assigned a <code>character</code> data type when auto-determined. Other data type values can be used to override automatic determination. Note that attempting to coerce <code>output_type_id</code> to a data type that is not valid for the data (e.g. trying to coerce "character" values to "double") will likely result in an error or potentially unexpected behaviour so use with care.
<code>partitions</code>	a named list specifying the arrow data types of any partitioning column.
<code>skip_checks</code>	Logical. If <code>TRUE</code> (default), skip validation checks when opening hub datasets, providing optimal performance especially for large cloud hubs (AWS S3, GCS) by minimizing I/O operations. However, this will result in an error if the model output directory contains files that cannot be opened as part of the dataset. Setting to <code>FALSE</code> will attempt to open and exclude any invalid files that cannot be read as part of the dataset. This results in slower performance due to increased I/O operations but provides more robustness when working with directories that may contain invalid files. Note that hubs validated through the <code>hubValidations</code> package should not require these additional checks. If invalid (non-model output) files are present in the model output directory, use the <code>ignore_files</code> argument to exclude them.
<code>na</code>	A character vector of strings to interpret as missing values. Only applies to CSV files. The default is <code>c("NA", "")</code> . Useful when actual character string "NA" values are used in the data. In such a case, use empty cells to indicate missing values in your files and set <code>na = ""</code> .
<code>ignore_files</code>	A character vector of file names (not paths) or file prefixes to ignore when discovering model output files to include in dataset connections. Parent directory names should not be included. Common non-data files such as "README" and ".DS_Store" are ignored automatically, but additional files can be excluded by specifying them here.

<code>model_output_dir</code>	Either a character string path to a local directory containing model output data or an object of class <code><SubTreeFileSystem></code> created using functions <code>s3_bucket()</code> or <code>gs_bucket()</code> by providing a string S3 or GCS bucket name or path to a directory containing model output data stored in the cloud. For more details consult the Using cloud storage (S3, GCS) in the <code>arrow</code> package.
<code>partition_names</code>	character vector that defines the field names to which recursive directory names correspond to. Defaults to a single <code>model_id</code> field which reflects the standard expected structure of a <code>model-output</code> directory.
<code>schema</code>	An <code>arrow::Schema</code> object for the Dataset. If NULL (the default), the schema will be inferred from the data sources.

Details

By default, common non-data files that may be present in model output directories (e.g. "README", ".DS_Store") are excluded automatically to prevent errors when connecting via Arrow. Additional files can be excluded using the `ignore_files` parameter.

Value

- `connect_hub` returns an S3 object of class `<hub_connection>`.
- `connect_model_output` returns an S3 object of class `<mod_out_connection>`.

Both objects are connected to the data in the model-output directory via an Apache arrow `FileSystemDataset` connection. The connection can be used to extract data using `dplyr` custom queries. The `<hub_connection>` class also contains modeling hub metadata.

Functions

- `connect_hub()`: connect to a fully configured Modeling Hub directory.
- `connect_model_output()`: connect directly to a `model-output` directory. This function can be used to access data directly from an appropriately set up model output directory which is not part of a fully configured hub.

Examples

```
# Connect to a local simple forecasting Hub.
hub_path <- system.file("testhubs/simple", package = "hubUtils")
hub_con <- connect_hub(hub_path)
hub_con
hub_con <- connect_hub(hub_path, output_type_id_datatype = "character")
hub_con
# Connect directly to a local `model-output` directory
mod_out_path <- system.file("testhubs/simple/model-output", package = "hubUtils")
mod_out_con <- connect_model_output(mod_out_path)
mod_out_con
# Query hub_connection for data
library(dplyr)
hub_con |>
```

```

    filter(
      origin_date == "2022-10-08",
      horizon == 2
    ) |>
    collect_hub()
mod_out_con |>
  filter(
    origin_date == "2022-10-08",
    horizon == 2
  ) |>
  collect_hub()
# Ignore a file
connect_hub(hub_path, ignore_files = c("README", "2022-10-08-team1-goodmodel.csv"))
# Connect to a simple forecasting Hub stored in an AWS S3 bucket.
## Not run:
hub_path <- s3_bucket("hubverse/hubutils/testhubs/simple/")
hub_con <- connect_hub(hub_path)
hub_con

## End(Not run)

```

connect_target_oracle_output

Open connection to oracle-output target data

Description

[Experimental] Open the oracle-output target data file(s) in a hub as an arrow dataset.

Usage

```

connect_target_oracle_output(
  hub_path = ".",
  date_col = NULL,
  na = c("NA", ""),
  ignore_files = NULL,
  output_type_id_datatype = c("from_config", "auto", "character", "double", "integer",
    "logical", "Date")
)

```

Arguments

hub_path	Either a character string path to a local Modeling Hub directory or an object of class <code><SubTreeFileSystem></code> created using functions <code>s3_bucket()</code> or <code>gs_bucket()</code> by providing a string S3 or GCS bucket name or path to a Modeling Hub directory stored in the cloud. For more details consult the Using cloud storage (S3, GCS) in the <code>arrow</code> package. The hub must be fully configured with valid <code>admin.json</code> and <code>tasks.json</code> files within the <code>hub-config</code> directory.
----------	---

<code>date_col</code>	Optional column name to be interpreted as date. Default is NULL. Useful when the required date column is a partitioning column in the target data and does not have the same name as a date typed task ID variable in the config. Note: Ignored when <code>target-data.json</code> exists (v6+); date column is read from config.
<code>na</code>	A character vector of strings to interpret as missing values. Only applies to CSV files. The default is <code>c("NA", "")</code> . Useful when actual character string "NA" values are used in the data. In such a case, use empty cells to indicate missing values in your files and set <code>na = ""</code> .
<code>ignore_files</code>	A character vector of file names (not paths) or file prefixes to ignore when discovering model output files to include in dataset connections. Parent directory names should not be included. Common non-data files such as "README" and ".DS_Store" are ignored automatically, but additional files can be excluded by specifying them here.
<code>output_type_id_datatype</code>	character string. One of "from_config", "auto", "character", "double", "integer", "logical", "Date". Defaults to "from_config" which uses the setting in the <code>output_type_id_datatype</code> property in the <code>tasks.json</code> config file if available. If the property is not set in the config, the argument falls back to "auto" which determines the <code>output_type_id</code> data type automatically from the <code>tasks.json</code> config file as the simplest data type required to represent all output type ID values across all output types in the hub. When only point estimate output types (where <code>output_type_ids</code> are NA,) are being collected by a hub, the <code>output_type_id</code> column is assigned a character data type when auto-determined. Other data type values can be used to override automatic determination. Note that attempting to coerce <code>output_type_id</code> to a data type that is not valid for the data (e.g. trying to coerce "character" values to "double") will likely result in an error or potentially unexpected behaviour so use with care.

Details

If the target data is split across multiple files in a `oracle-output` directory, all files must share the same file format, either csv or parquet. No other types of files are currently allowed in a `oracle-output` directory.

Schema Creation:

This function uses different methods to create the Arrow schema depending on the hub configuration version:

v6+ hubs (with `target-data.json`): Schema is created directly from the `target-data.json` configuration file using `create_oracle_output_schema()`. This config-based approach is fast and deterministic, requiring no filesystem I/O to scan data files. It's especially beneficial for cloud storage where file scanning can be slow.

Hubs (without `target-data.json`): Schema is inferred by scanning the actual data files. This inference-based approach examines file structure and content to determine column types.

The function automatically detects which method to use based on the presence of `target-data.json` in the hub configuration.

Schema Ordering:

Column ordering in the resulting dataset depends on configuration version and file format:

v6+ hubs (with target-data.json):

- **Parquet:** Columns are reordered to the standard hubverse convention (see [get_target_data_colnames\(\)](#)). Parquet's column-by-name matching enables safe reordering.
- **CSV:** Original file ordering is preserved to avoid column name/position mismatches during collection.

Hubs (without target-data.json): Original file ordering is preserved regardless of format.

Value

An arrow dataset object of subclass `<target_oracle_output>`.

Examples

```
# Column Ordering: CSV vs Parquet in v6+ hubs
# For v6+ hubs with target-data.json, ordering differs by file format

# Example 1: CSV format (single file) - preserves original file ordering
hub_path_csv <- system.file("testhubs/v6/target_file", package = "hubUtils")
oo_con_csv <- connect_target_oracle_output(hub_path_csv)

# CSV columns are in their original file order
names(oo_con_csv)

# Collect and filter as usual
oo_con_csv |> dplyr::collect()
oo_con_csv |>
  dplyr::filter(location == "US") |>
  dplyr::collect()

# Example 2: Parquet format (directory) - reordered to hubverse convention
hub_path_parquet <- system.file("testhubs/v6/target_dir", package = "hubUtils")
oo_con_parquet <- connect_target_oracle_output(hub_path_parquet)

# Parquet columns follow hubverse convention (date first, then alphabetical)
names(oo_con_parquet)

# Reordering is safe for Parquet because it matches columns by name
# rather than position during collection
oo_con_parquet |> dplyr::collect()

# Both formats support the same filtering operations
oo_con_parquet |>
  dplyr::filter(target_end_date == "2022-12-31") |>
  dplyr::collect()

# Get distinct target_end_date values
oo_con_parquet |>
  dplyr::distinct(target_end_date) |>
```

```

dplyr::pull(as_vector = TRUE)

## Not run:
# Access Target oracle-output data from a cloud hub
s3_hub_path <- s3_bucket("example-complex-forecast-hub")
s3_con <- connect_target_oracle_output(s3_hub_path)
s3_con
s3_con |> dplyr::collect()

## End(Not run)

```

```
connect_target_timeseries
```

Open connection to time-series target data

Description

[Experimental] Open the time-series target data file(s) in a hub as an arrow dataset.

Usage

```

connect_target_timeseries(
  hub_path = ".",
  date_col = NULL,
  na = c("NA", ""),
  ignore_files = NULL
)

```

Arguments

- | | |
|-----------------|---|
| hub_path | Either a character string path to a local Modeling Hub directory or an object of class <code><SubTreeFileSystem></code> created using functions <code>s3_bucket()</code> or <code>gs_bucket()</code> by providing a string S3 or GCS bucket name or path to a Modeling Hub directory stored in the cloud. For more details consult the Using cloud storage (S3, GCS) in the <code>arrow</code> package. The hub must be fully configured with valid <code>admin.json</code> and <code>tasks.json</code> files within the <code>hub-config</code> directory. |
| date_col | Optional column name to be interpreted as date. Default is <code>NULL</code> . Useful when the required date column is a partitioning column in the target data and does not have the same name as a date typed task ID variable in the config. |
| na | A character vector of strings to interpret as missing values. Only applies to CSV files. The default is <code>c("NA", "")</code> . Useful when actual character string "NA" values are used in the data. In such a case, use empty cells to indicate missing values in your files and set <code>na = ""</code> . |

ignore_files A character vector of file **names** (not paths) or file **prefixes** to ignore when discovering model output files to include in dataset connections. Parent directory names should not be included. Common non-data files such as "README" and ".DS_Store" are ignored automatically, but additional files can be excluded by specifying them here.

Details

If the target data is split across multiple files in a **time-series** directory, all files must share the same file format, either csv or parquet. No other types of files are currently allowed in a **time-series** directory.

Schema Creation:

This function uses different methods to create the Arrow schema depending on the hub configuration version:

v6+ hubs (with target-data.json): Schema is created directly from the **target-data.json** configuration file using **create_timeseries_schema()**. This config-based approach is fast and deterministic, requiring no filesystem I/O to scan data files. It's especially beneficial for cloud storage where file scanning can be slow.

Hubs (without target-data.json): Schema is inferred by scanning the actual data files. This inference-based approach examines file structure and content to determine column types.

The function automatically detects which method to use based on the presence of **target-data.json** in the hub configuration.

Schema Ordering:

Column ordering in the resulting dataset depends on configuration version and file format:

v6+ hubs (with target-data.json):

- **Parquet:** Columns are reordered to the standard hubverse convention (see **get_target_data_colnames()**). Parquet's column-by-name matching enables safe reordering.
- **CSV:** Original file ordering is preserved to avoid column name/position mismatches during collection.

Hubs (without target-data.json): Original file ordering is preserved regardless of format.

Value

An arrow dataset object of subclass `<target_timeseries>`.

Examples

```
# Column Ordering: CSV vs Parquet in v6+ hubs
# For v6+ hubs with target-data.json, ordering differs by file format

# Example 1: CSV format (single file) - preserves original file ordering
hub_path_csv <- system.file("testhubs/v6/target_file", package = "hubUtils")
ts_con_csv <- connect_target_timeseries(hub_path_csv)

# CSV columns are in their original file order
```

```

names(ts_con_csv)
# Note: columns appear in the order they are in the CSV file

# Collect and filter as usual
ts_con_csv |> dplyr::collect()
ts_con_csv |>
  dplyr::filter(location == "US") |>
  dplyr::collect()

# Example 2: Parquet format (directory) - reordered to hubverse convention
hub_path_parquet <- system.file("testhubs/v6/target_dir", package = "hubUtils")
ts_con_parquet <- connect_target_timeseries(hub_path_parquet)

# Parquet columns follow hubverse convention
names(ts_con_parquet)

# Reordering is safe for Parquet because it matches columns by name
# rather than position during collection
ts_con_parquet |> dplyr::collect()

# Both formats support the same filtering operations
ts_con_parquet |>
  dplyr::filter(target_end_date == "2022-12-31") |>
  dplyr::collect()

## Not run:
# Access Target time-series data from a cloud hub
s3_hub_path <- s3_bucket("example-complex-forecast-hub")
s3_con <- connect_target_timeseries(s3_hub_path)
s3_con
s3_con |> dplyr::collect()

## End(Not run)

```

create_hub_schema	<i>Create a Hub arrow schema</i>
-------------------	----------------------------------

Description

Create an arrow schema from a `tasks.json` config file. For use when opening an arrow dataset.

Usage

```

create_hub_schema(
  config_tasks,
  partitions = list(model_id = arrow::utf8()),
  output_type_id_datatype = c("from_config", "auto", "character", "double", "integer",
    "logical", "Date"),
  r_schema = FALSE
)

```

Arguments

- config_tasks** a list version of the content's of a hub's `tasks.json` config file created using function `hubUtils::read_config()`.
- partitions** a named list specifying the arrow data types of any partitioning column.
- output_type_id_datatype** character string. One of "from_config", "auto", "character", "double", "integer", "logical", "Date". Defaults to "from_config" which uses the setting in the `output_type_id_datatype` property in the `tasks.json` config file if available. If the property is not set in the config, the argument falls back to "auto" which determines the `output_type_id` data type automatically from the `tasks.json` config file as the simplest data type required to represent all output type ID values across all output types in the hub. When only point estimate output types (where `output_type_ids` are NA,) are being collected by a hub, the `output_type_id` column is assigned a `character` data type when auto-determined. Other data type values can be used to override automatic determination. Note that attempting to coerce `output_type_id` to a data type that is not valid for the data (e.g. trying to coerce "character" values to "double") will likely result in an error or potentially unexpected behaviour so use with care.
- r_schema** Logical. If `FALSE` (default), return an `arrow::schema()` object. If `TRUE`, return a character vector of R data types.

Value

an arrow schema object that can be used to define column datatypes when opening model output data. If `r_schema = TRUE`, a character vector of R data types.

Examples

```
hub_path <- system.file("testhubs/simple", package = "hubUtils")
config_tasks <- hubUtils::read_config(hub_path, "tasks")
schema <- create_hub_schema(config_tasks)
```

```
create_model_out_submit_tmpl
```

Create a model output submission file template

Description

[Defunct] This function has been moved to the `hubValidations` package and renamed to `submission_tmpl()`.

Usage

```
create_model_out_submit_tmpl(
  hub_con,
  config_tasks,
  round_id,
  required_vals_only = FALSE,
  complete_cases_only = TRUE
)
```

Arguments

<code>hub_con</code>	A <code><hub_connection></code> class object.
<code>config_tasks</code>	a list version of the content's of a hub's <code>tasks.json</code> config file, accessed through the "config_tasks" attribute of a <code><hub_connection></code> object or function <code>hubUtils::read_config()</code> .
<code>round_id</code>	Character string. Round identifier. If the round is set to <code>round_id_from_variable: true</code> , IDs are values of the task ID defined in the round's <code>round_id</code> property of <code>config_tasks</code> . Otherwise should match round's <code>round_id</code> value in config. Ignored if hub contains only a single round.
<code>required_vals_only</code>	Logical. Whether to return only combinations of Task ID and related output type ID required values.
<code>complete_cases_only</code>	Logical. If <code>TRUE</code> (default) and <code>required_vals_only = TRUE</code> , only rows with complete cases of combinations of required values are returned. If <code>FALSE</code> , rows with incomplete cases of combinations of required values are included in the output.

Details

For task IDs or `output_type_ids` where all values are optional, by default, columns are included as columns of NAs when `required_vals_only = TRUE`. When such columns exist, the function returns a tibble with zero rows, as no complete cases of required value combinations exists. *(Note that determination of complete cases does excludes valid NA output_type_id values in "mean" and "median" output types)*. To return a template of incomplete required cases, which includes NA columns, use `complete_cases_only = FALSE`.

When sample output types are included in the output, the `output_type_id` column contains example sample indexes which are useful for identifying the compound task ID structure of multivariate sampling distributions in particular, i.e. which combinations of task ID values represent individual samples.

When a round is set to `round_id_from_variable: true`, the value of the task ID from which round IDs are derived (i.e. the task ID specified in `round_id` property of `config_tasks`) is set to the value of the `round_id` argument in the returned output.

Value

a tibble template containing an expanded grid of valid task ID and output type ID value combinations for a given submission round and output type. If `required_vals_only =`

TRUE, values are limited to the combination of required values only.

```
create_oracle_output_schema
```

Create oracle-output target data file schema

Description

Create oracle-output target data file schema

Usage

```
create_oracle_output_schema(
  hub_path,
  date_col = NULL,
  na = c("NA", ""),
  ignore_files = NULL,
  r_schema = FALSE,
  output_type_id_datatype = c("from_config", "auto", "character", "double", "integer",
    "logical", "Date")
)
```

Arguments

hub_path	Either a character string path to a local Modeling Hub directory or an object of class <code><SubTreeFileSystem></code> created using functions <code>s3_bucket()</code> or <code>gs_bucket()</code> by providing a string S3 or GCS bucket name or path to a Modeling Hub directory stored in the cloud. For more details consult the Using cloud storage (S3, GCS) in the <code>arrow</code> package. The hub must be fully configured with valid <code>admin.json</code> and <code>tasks.json</code> files within the <code>hub-config</code> directory.
date_col	Optional column name to be interpreted as date. Default is <code>NULL</code> . Useful when the required date column is a partitioning column in the target data and does not have the same name as a date typed task ID variable in the config. Note: Ignored when <code>target-data.json</code> exists (v6+); date column is read from config.
na	A character vector of strings to interpret as missing values. Only applies to CSV files. The default is <code>c("NA", "")</code> . Useful when actual character string "NA" values are used in the data. In such a case, use empty cells to indicate missing values in your files and set <code>na = ""</code> .
ignore_files	A character vector of file names (not paths) or file prefixes to ignore when discovering model output files to include in dataset connections. Parent directory names should not be included. Common non-data files such as "README" and ".DS_Store" are ignored automatically, but additional files can be excluded by specifying them here.
r_schema	Logical. If <code>FALSE</code> (default), return an <code>arrow::schema()</code> object. If <code>TRUE</code> , return a character vector of R data types.

output_type_id_datatype

character string. One of "from_config", "auto", "character", "double", "integer", "logical", "Date". Defaults to "from_config" which uses the setting in the `output_type_id_datatype` property in the `tasks.json` config file if available. If the property is not set in the config, the argument falls back to "auto" which determines the `output_type_id` data type automatically from the `tasks.json` config file as the simplest data type required to represent all output type ID values across all output types in the hub. When only point estimate output types (where `output_type_ids` are NA,) are being collected by a hub, the `output_type_id` column is assigned a `character` data type when auto-determined. Other data type values can be used to override automatic determination. Note that attempting to coerce `output_type_id` to a data type that is not valid for the data (e.g. trying to coerce "character" values to "double") will likely result in an error or potentially unexpected behaviour so use with care.

Details

When `target-data.json` (v6.0.0+) is present, schema is created directly from config without reading target data files. Otherwise, schema is inferred by reading the dataset. Config-based approach avoids file I/O (especially beneficial for cloud storage) and provides deterministic schema creation.

Value

an arrow `<schema>` class object

Examples

```
hub_path <- system.file("testhubs/v5/target_file", package = "hubUtils")
# Create target oracle-output schema
create_oracle_output_schema(hub_path)
# target oracle-output schema from a cloud hub
s3_hub_path <- s3_bucket("example-complex-forecast-hub")
create_oracle_output_schema(s3_hub_path)
```

```
create_timeseries_schema
```

Create time-series target data file schema

Description

Create time-series target data file schema

Usage

```
create_timeseries_schema(
  hub_path,
  date_col = NULL,
  na = c("NA", ""),
  ignore_files = NULL,
  r_schema = FALSE
)
```

Arguments

- | | |
|---------------------|---|
| hub_path | Either a character string path to a local Modeling Hub directory or an object of class <code><SubTreeFileSystem></code> created using functions <code>s3_bucket()</code> or <code>gs_bucket()</code> by providing a string S3 or GCS bucket name or path to a Modeling Hub directory stored in the cloud. For more details consult the Using cloud storage (S3, GCS) in the <code>arrow</code> package. The hub must be fully configured with valid <code>admin.json</code> and <code>tasks.json</code> files within the <code>hub-config</code> directory. |
| date_col | Optional column name to be interpreted as date. Default is <code>NULL</code> . Useful when the required date column is a partitioning column in the target data and does not have the same name as a date typed task ID variable in the config. Note: Ignored when <code>target-data.json</code> exists (v6+); date column is read from config. |
| na | A character vector of strings to interpret as missing values. Only applies to CSV files. The default is <code>c("NA", "")</code> . Useful when actual character string "NA" values are used in the data. In such a case, use empty cells to indicate missing values in your files and set <code>na = ""</code> . |
| ignore_files | A character vector of file names (not paths) or file prefixes to ignore when discovering model output files to include in dataset connections. Parent directory names should not be included. Common non-data files such as "README" and ".DS_Store" are ignored automatically, but additional files can be excluded by specifying them here. |
| r_schema | Logical. If <code>FALSE</code> (default), return an <code>arrow::schema()</code> object. If <code>TRUE</code> , return a character vector of R data types. |

Details

When `target-data.json` (v6.0.0+) is present, schema is created directly from config without reading target data files. Otherwise, schema is inferred by reading the dataset. Config-based approach avoids file I/O (especially beneficial for cloud storage) and provides deterministic schema creation.

Value

an arrow `<schema>` class object

Examples

```
hub_path <- system.file("testhubs/v5/target_file", package = "hubUtils")
# Create target time-series schema
create_timeseries_schema(hub_path)
# target time-series schema from a cloud hub
s3_hub_path <- s3_bucket("example-complex-forecast-hub")
create_timeseries_schema(s3_hub_path)
```

```
expand_model_out_val_grid
```

Create expanded grid of valid task ID and output type value combinations

Description

[Defunct] This function has been moved to the `hubValidations` package and renamed to `expand_model_out_grid()`.

Usage

```
expand_model_out_val_grid(
  config_tasks,
  round_id,
  required_vals_only = FALSE,
  all_character = FALSE,
  as_arrow_table = FALSE,
  bind_model_tasks = TRUE,
  include_sample_ids = FALSE
)
```

Arguments

- | | |
|---------------------------------|--|
| <code>config_tasks</code> | a list version of the content's of a hub's <code>tasks.json</code> config file, accessed through the "config_tasks" attribute of a <hub_connection> object or function <code>hubUtils::read_config()</code> . |
| <code>round_id</code> | Character string. Round identifier. If the round is set to <code>round_id_from_variable: true</code> , IDs are values of the task ID defined in the round's <code>round_id</code> property of <code>config_tasks</code> . Otherwise should match round's <code>round_id</code> value in config. Ignored if hub contains only a single round. |
| <code>required_vals_only</code> | Logical. Whether to return only combinations of Task ID and related output type ID required values. |
| <code>all_character</code> | Logical. Whether to return all character column. |
| <code>as_arrow_table</code> | Logical. Whether to return an arrow table. Defaults to <code>FALSE</code> . |

bind_model_tasks

Logical. Whether to bind expanded grids of values from multiple modeling tasks into a single tibble/arrow table or return a list.

include_sample_ids

Logical. Whether to include sample identifiers in the `output_type_id` column.

Details

When a round is set to `round_id_from_variable: true`, the value of the task ID from which round IDs are derived (i.e. the task ID specified in `round_id` property of `config_tasks`) is set to the value of the `round_id` argument in the returned output.

When sample output types are included in the output and `include_sample_ids = TRUE`, the `output_type_id` column contains example sample indexes which are useful for identifying the compound task ID structure of multivariate sampling distributions in particular, i.e. which combinations of task ID values represent individual samples.

Value

If `bind_model_tasks = TRUE` (default) a tibble or arrow table containing all possible task ID and related output type ID value combinations. If `bind_model_tasks = FALSE`, a list containing a tibble or arrow table for each round modeling task.

Columns are coerced to data types according to the hub schema, unless `all_character = TRUE`. If `all_character = TRUE`, all columns are returned as character which can be faster when large expanded grids are expected. If `required_vals_only = TRUE`, values are limited to the combinations of required values only.

extract_hive_partitions

Extract Hive-style partition key-value pairs from a path

Description

Given a filesystem path, this function extracts Hive-style partition key-value pairs (i.e., path components formatted as `key=value`). It supports decoding URL-encoded values (e.g., `"wk%20flu"` → `"wk flu"`), and handles empty values (e.g., `"key="`) as `NA`, consistent with Hive and Arrow semantics.

Usage

```
extract_hive_partitions(path, strict = TRUE)
```

Arguments**path**

A character string of length 1: the path to a file or directory.

strict

Logical. If `TRUE`, invalid partition segments (e.g., `=value`, or just `=`) will trigger an error. If `FALSE`, only valid `key=value` components are returned.

Details

If `strict = TRUE`, the function will abort with a detailed error message if any malformed partition-like segments are found.

Value

A named character vector where the names are partition keys and the values are decoded values. Returns `NULL` if no valid partitions are found.

See Also

[is_hive_partitioned_path\(\)](#)

Examples

```
extract_hive_partitions("data/country=US/year=2024/file.parquet")
extract_hive_partitions("data/country=/year=2024/", strict = TRUE)
# extract_hive_partitions("data/=US/year=2024/", strict = TRUE) # This will error
```

<code>get_s3_bucket_name</code>	<i>Get the bucket name for the cloud storage location.</i>
---------------------------------	--

Description

Get the bucket name for the cloud storage location.

Usage

```
get_s3_bucket_name(hub_path = ".")
```

Arguments

`hub_path` Path to a hub directory.

Value

The bucket name for the cloud storage location.

Examples

```
hub_path <- system.file("testhubs/v5/target_file", package = "hubUtils")
get_s3_bucket_name(hub_path)
# Get config info from GitHub
get_s3_bucket_name(
  "https://github.com/hubverse-org/example-complex-forecast-hub"
)
```

```
get_target_data_colnames
```

Get expected target data column names from config

Description

Extracts the expected column names for target data from a hub's configuration files in the correct order. This is useful for validation and schema generation without needing to inspect the actual dataset.

Usage

```
get_target_data_colnames(
  config_target_data,
  target_type = c("time-series", "oracle-output")
)
```

Arguments

<code>config_target_data</code>	A <code>config_target_data</code> object (from <code>hubUtils::read_config(hub_path, "target-data")</code>)
<code>target_type</code>	Character string specifying the target data type. Must be either <code>"time-series"</code> or <code>"oracle-output"</code> .

Details

The function builds the column name vector directly from the configuration objects without requiring dataset inspection. This makes it lightweight, efficient, and suitable for validation purposes.

For **time-series** data, columns are ordered as:

1. Task ID columns from `observable_unit`
2. Date column (if not in `observable_unit`)
3. Non-task ID columns from `target-data.json` (if present)
4. `observation` column (target value)
5. `as_of` column (if `versioned = TRUE`)

For **oracle-output** data, columns are ordered as:

1. Task ID columns from `observable_unit`
2. Date column (if not in `observable_unit`)
3. `output_type` and `output_type_id` columns (if `has_output_type_ids = TRUE`)
4. `oracle_value` column (target value)
5. `as_of` column (if `versioned = TRUE`)

Value

A character vector of expected column names in the correct order:

- Date column
- Task ID columns (from `observable_unit`)
- Non-task ID columns (time-series only, if specified in config)
- Output type columns (`output_type` and `output_type_id`, oracle-output only if specified in config)
- Target value column (`observation` for time-series, `oracle_value` for oracle-output)
- `as_of` column (if data is versioned)

Examples

```
# Note: These examples require test data
hub_path <- system.file("testhubs/v6/target_file", package = "hubUtils")
config_target_data <- hubUtils::read_config(hub_path, "target-data")

# Get time-series column names
get_target_data_colnames(
  config_target_data,
  target_type = "time-series"
)

# Get oracle-output column names
get_target_data_colnames(
  config_target_data,
  target_type = "oracle-output"
)
```

`get_target_file_ext` *Get target data file unique file extensions.*

Description

Get the unique file extension(s) of the target data file(s) in `target_path`. If `target_path` is a directory, the function will return the unique file extensions of all files in the directory. If `target_path` is a file, the function will return the file extension of that file.

Usage

```
get_target_file_ext(hub_path = NULL, target_path)
```

Arguments

<code>hub_path</code>	If not <code>NULL</code> , must be a <code>SubTreeFileSystem</code> class object of the root to a cloud hosted hub. Required to trigger the <code>SubTreeFileSystem</code> method.
<code>target_path</code>	character string. The path to the target data file or directory. Usually the output of <code>get_target_path()</code> .

Examples

```
hub_path <- system.file("testhubs/v5/target_file", package = "hubUtils")
target_path <- get_target_path(hub_path, "time-series")
get_target_file_ext(hub_path, target_path)
```

get_target_path	<i>Get the path(s) to the target data file(s) in the hub directory.</i>
-----------------	---

Description

Get the path(s) to the target data file(s) in the hub directory.

Usage

```
get_target_path(hub_path, target_type = c("time-series", "oracle-output"))
```

Arguments

hub_path	Either a character string path to a local Modeling Hub directory or an object of class <code><SubTreeFileSystem></code> created using functions <code>s3_bucket()</code> or <code>gs_bucket()</code> by providing a string S3 or GCS bucket name or path to a Modeling Hub directory stored in the cloud. For more details consult the Using cloud storage (S3, GCS) in the <code>arrow</code> package. The hub must be fully configured with valid <code>admin.json</code> and <code>tasks.json</code> files within the <code>hub-config</code> directory.
target_type	Type of target data to retrieve matching files. One of "time-series" or "oracle-output". Defaults to "time-series".

Value

a character vector of path(s) to target data file(s) (in the `target-data` directory) that make the `target_type` requested.

Examples

```
hub_path <- system.file("testhubs/v5/target_file", package = "hubUtils")
get_target_path(hub_path)
get_target_path(hub_path, "time-series")
get_target_path(hub_path, "oracle-output")
# Access cloud data
s3_bucket_name <- get_s3_bucket_name(hub_path)
s3_hub_path <- s3_bucket(s3_bucket_name)
get_target_path(s3_hub_path)
get_target_path(s3_hub_path, "oracle-output")
```

`gs_bucket`*Connect to a Google Cloud Storage (GCS) bucket*

Description

See `arrow::gs_bucket()` for details.

Value

A `SubTreeFileSystem` containing an `GcsFileSystem` and the bucket's relative path. Note that this function's success does not guarantee that you are authorized to access the bucket's contents.

Examples

```
bucket <- gs_bucket("voltrondata-labs-datasets")
```

`is_hive_partitioned_path`*Check whether a path contains Hive-style partitioning*

Description

This function checks if a given file or directory path includes one or more Hive-style partition segments (i.e., subdirectories formatted as `key=value`). This function can operate in a strict or lenient mode, depending on whether you want to catch malformed partition-like segments.

Usage

```
is_hive_partitioned_path(path, strict = TRUE)
```

Arguments

<code>path</code>	Character string. Path to a file or directory.
<code>strict</code>	Logical. If <code>TRUE</code> , the function will throw an error if any malformed partition segments are found (e.g., <code>=value</code> , missing key, or malformed <code>=</code> without a value). If <code>FALSE</code> , it simply returns <code>TRUE</code> if any valid <code>key=value</code> segments are found.

Details

A valid partition segment must:

- Contain an equals sign (=)
- Have a non-empty key before the equals sign
- May have an empty value (interpreted as NA in most Hive/Arrow contexts)

In strict mode, the function validates that all **key=value** segments are well-formed and will abort if any are not.

Value

A logical value: TRUE if the path contains one or more valid Hive-style partition segments, FALSE otherwise.

See Also

`extract_hive_partitions()` to extract key-value pairs from Hive-style paths.

Examples

```
is_hive_partitioned_path("data/country=US/year=2024/file.parquet")
is_hive_partitioned_path("data/country=/year=2024/", strict = TRUE)
# is_hive_partitioned_path("data/=US/year=2024/", strict = TRUE) # This will error
```

`load_model_metadata` *Compile hub model metadata*

Description

Loads in hub model metadata for all models or a specified subset of models and compiles it into a tibble with one row per model.

Usage

```
load_model_metadata(hub_path, model_ids = NULL)
```

Arguments

- | | |
|------------------------|---|
| <code>hub_path</code> | Either a character string path to a local Modeling Hub directory or an object of class <code><SubTreeFileSystem></code> created using functions <code>s3_bucket()</code> or <code>gs_bucket()</code> by providing a string S3 or GCS bucket name or path to a Modeling Hub directory stored in the cloud. For more details consult the Using cloud storage (S3, GCS) in the <code>arrow</code> package. |
| <code>model_ids</code> | A vector of character strings of models for which to load metadata. Defaults to NULL, in which case metadata for all models is loaded. |

Value

tibble with model metadata. One row for each model, one column for each top-level field in the metadata file. For metadata files with nested structures, this tibble may contain list-columns where the entries are lists containing the nested metadata values.

Examples

```
# Load in model metadata from local hub
hub_path <- system.file("testhubs/simple", package = "hubUtils")
load_model_metadata(hub_path)
load_model_metadata(hub_path, model_ids = c("hub-baseline"))
```

```
print.hub_connection Print a <hub_connection> or <mod_out_connection> S3 class object
```

Description

Print a <hub_connection> or <mod_out_connection> S3 class object

Usage

```
## S3 method for class 'hub_connection'
print(x, verbose = FALSE, ...)

## S3 method for class 'mod_out_connection'
print(x, verbose = FALSE, ...)
```

Arguments

x	A <hub_connection> or <mod_out_connection> S3 class object.
verbose	Logical. Whether to print the full structure of the object. Defaults to FALSE.
...	Further arguments passed to or from other methods.

Functions

- `print(hub_connection)`: print a <hub_connection> object.
- `print(mod_out_connection)`: print a <mod_out_connection> object.

Examples

```
hub_path <- system.file("testhubs/simple", package = "hubUtils")
hub_con <- connect_hub(hub_path)
hub_con
print(hub_con)
print(hub_con, verbose = TRUE)
mod_out_path <- system.file("testhubs/simple/model-output", package = "hubUtils")
```

```
mod_out_con <- connect_model_output(mod_out_path)
print(mod_out_con)
```

r_to_arrow_datatypes *Create R type to Arrow DataType mapping*

Description

Returns a named list mapping base R type strings (e.g., "character", "integer") to their corresponding Arrow [arrow::DataType](#) objects. This is the inverse of [arrow_to_r_datatypes](#) and is useful when creating Arrow schemas programmatically from R type specifications.

Usage

```
r_to_arrow_datatypes()
```

Details

This function generates the mapping dynamically. The R type strings match those used in the `non_task_id_schema` field of `target-data.json` configuration files.

This is particularly useful for:

- Creating custom Arrow schemas from R type specifications
- Converting configuration-based type information to Arrow schemas
- Programmatic schema generation

Value

A named list with 6 entries mapping R types to Arrow DataType objects:

```
logical arrow::bool()
integer arrow::int32() (uses int32 as default)
double  arrow::float64()
character arrow::utf8()
Date    arrow::date32()
POSIXct arrow::timestamp(unit = "ms")
```

See Also

[arrow_to_r_datatypes](#), [create_timeseries_schema\(\)](#), [create_oracle_output_schema\(\)](#)

Examples

```
# Get the mapping
type_map <- r_to_arrow_datatypes()

# Use it to create Arrow types from R type strings
r_types <- c("character", "integer", "double")
arrow_types <- type_map[r_types]

# Create a simple Arrow schema
my_schema <- arrow::schema(
  name = type_map[["character"]],
  age = type_map[["integer"]],
  score = type_map[["double"]]
)
my_schema
```

s3_bucket

Connect to an AWS S3 bucket

Description

See `arrow::s3_bucket()` for details.

Value

A `SubTreeFileSystem` containing an `S3FileSystem` and the bucket's relative path. Note that this function's success does not guarantee that you are authorized to access the bucket's contents.

Examples

```
bucket <- s3_bucket("voltrondata-labs-datasets")

# Turn on debug logging. The following line of code should be run in a fresh
# R session prior to any calls to `s3_bucket()` (or other S3 functions)
Sys.setenv("ARROW_S3_LOG_LEVEL" = "DEBUG")
bucket <- s3_bucket("voltrondata-labs-datasets")
```

Index

- * datasets
 - arrow_to_r_datatypes, 2
- arrow::DataType, 32
- arrow::gs_bucket(), 29
- arrow::s3_bucket(), 33
- arrow::Schema, 2, 3, 11
- arrow::schema(), 18, 20, 22
- arrow_schema_to_string (*as_r_schema*), 3
- arrow_schema_to_string(), 3
- arrow_to_r_datatypes, 2, 32
- arrow_to_r_datatypes(), 4
- as_model_out_tbl(), 6
- as_r_schema, 3
- as_r_schema(), 3
- coerce_to_character
 - (*coerce_to_hub_schema*), 5
- coerce_to_hub_schema, 5
- collect_hub, 6
- collect_zoltar, 7
- connect_hub, 9
- connect_model_output (*connect_hub*), 9
- connect_target_oracle_output, 12
- connect_target_timeseries, 15
- create_hub_schema, 17
- create_model_out_submit_tmpl, 18
- create_oracle_output_schema, 20
- create_oracle_output_schema(), 13, 32
- create_timeseries_schema, 21
- create_timeseries_schema(), 16, 32
- expand_model_out_val_grid, 23
- extract_hive_partitions, 24
- extract_hive_partitions(), 30
- get_s3_bucket_name, 25
- get_target_data_colnames, 26
- get_target_data_colnames(), 14, 16
- get_target_file_ext, 27
- get_target_path, 28
- get_target_path(), 27
- gs_bucket, 29
- gs_bucket(), 9, 11, 12, 15, 20, 22, 28, 30
- hubUtils::read_config(), 5, 18, 19, 23
- is_hive_partitioned_path, 29
- is_hive_partitioned_path(), 25
- is_supported_arrow_type
 - (*as_r_schema*), 3
- load_model_metadata, 30
- print.hub_connection, 31
- print.mod_out_connection
 - (*print.hub_connection*), 31
- r_to_arrow_datatypes, 32
- s3_bucket, 33
- s3_bucket(), 9, 11, 12, 15, 20, 22, 28, 30
- validate_arrow_schema (*as_r_schema*), 3