Package 'arrow'

December 5, 2024

Title Integration to 'Apache' 'Arrow'

Version 18.1.0

Description 'Apache' 'Arrow' <https://arrow.apache.org/> is a cross-language development platform for in-memory data. It specifies a standardized language-independent columnar memory format for flat and hierarchical data, organized for efficient analytic operations on modern hardware. This package provides an interface to the 'Arrow C++' library.

Depends R (>= 4.0)

License Apache License (>= 2.0)

URL https://github.com/apache/arrow/, https://arrow.apache.org/docs/r/

BugReports https://github.com/apache/arrow/issues

Encoding UTF-8

Language en-US

SystemRequirements C++17; for AWS S3 support on Linux, libcurl and openssl (optional); cmake >= 3.16 (build-time only, and only for full source build)

Biarch true

Imports assertthat, bit64 (>= 0.9-7), glue, methods, purrr, R6, rlang (>= 1.0.0), stats, tidyselect (>= 1.0.0), utils, vctrs

RoxygenNote 7.3.2

Config/testthat/edition 3

Config/build/bootstrap TRUE

Suggests blob, curl, cli, DBI, dbplyr, decor, distro, dplyr, duckdb (>= 0.2.8), hms, jsonlite, knitr, lubridate, pillar, pkgload, reticulate, rmarkdown, stringi, stringr, sys, testthat (>= 3.1.0), tibble, tzdb, withr

LinkingTo cpp11 (>= 0.4.2)

Collate 'arrowExports.R' 'enums.R' 'arrow-object.R' 'type.R' 'array-data.R' 'arrow-datum.R' 'array.R' 'arrow-info.R' 'arrow-package.R' 'arrow-tabular.R' 'buffer.R'

'chunked-array.R' 'io.R' 'compression.R' 'scalar.R' 'compute.R'
'config.R' 'csv.R' 'dataset.R' 'dataset-factory.R'
'dataset-format.R' 'dataset-partition.R' 'dataset-scan.R'
'dataset-write.R' 'dictionary.R' 'dplyr-across.R'
'dplyr-arrange.R' 'dplyr-by.R' 'dplyr-collect.R'
'dplyr-count.R' 'dplyr-datetime-helpers.R' 'dplyr-distinct.R'
'dplyr-eval.R' 'dplyr-filter.R' 'dplyr-funcs-agg.R'
'dplyr-funcs-augmented.R' 'dplyr-funcs-conditional.R'
'dplyr-funcs-datetime.R' 'dplyr-funcs-doc.R'
'dplyr-funcs-math.R' 'dplyr-funcs-simple.R'
'dplyr-funcs-string.R' 'dplyr-funcs-type.R' 'expression.R'
'dplyr-funcs.R' 'dplyr-glimpse.R' 'dplyr-group-by.R'
'dplyr-join.R' 'dplyr-mutate.R' 'dplyr-select.R'
'dplyr-slice.R' 'dplyr-summarize.R' 'dplyr-union.R'
'record-batch.R' 'table.R' 'dplyr.R' 'duckdb.R' 'extension.R'
'feather.R' 'field.R' 'filesystem.R' 'flight.R'
'install-arrow.R' 'ipc-stream.R' 'json.R' 'memory-pool.R'
'message.R' 'metadata.R' 'parquet.R' 'python.R'
'query-engine.R' 'record-batch-reader.R'
'record-batch-writer.R' 'reexports-bit64.R'
'reexports-tidyselect.R' 'schema.R' 'udf.R' 'util.R'

NeedsCompilation yes

Author Neal Richardson [aut],
Ian Cook [aut],
Nic Crane [aut],
Dewey Dunnington [aut] (<https://orcid.org/0000-0002-9415-4582>),
Romain François [aut] (<https://orcid.org/0000-0002-2444-4226>),
Jonathan Keane [aut, cre],
Dragoș Moldovan-Grünfeld [aut],
Jeroen Ooms [aut],
Jacob Wujciak-Jens [aut],
Javier Luraschi [ctb],
Karl Dunkle Werner [ctb] (<https://orcid.org/0000-0003-0523-7309>),
Jeffrey Wong [ctb],
Apache Arrow [aut, cph]

Maintainer Jonathan Keane <jkeane@gmail.com>

Repository CRAN

Date/Publication 2024-12-05 06:20:02 UTC

Contents

`acero` *Functions available in Arrow dplyr queries*

Description

The `arrow` package contains methods for 37 `dplyr` table functions, many of which are "verbs" that do transformations to one or more tables. The package also has mappings of 212 R functions to the corresponding functions in the Arrow compute library. These allow you to write code inside of `dplyr` methods that call R functions, including many in packages like `stringr` and `lubridate`, and they will get translated to Arrow and run on the Arrow query engine (Acero). This document lists all of the mapped functions.

`dplyr` verbs

Most verb functions return an `arrow_dplyr_query` object, similar in spirit to a `dbplyr::tbl_lazy`. This means that the verbs do not eagerly evaluate the query on the data. To run the query, call either `compute()`, which returns an `arrow` Table, or `collect()`, which pulls the resulting Table into an R `tibble`.

- `anti_join()`: the `copy` argument is ignored
- `arrange()`
- `collapse()`
- `collect()`
- `compute()`
- `count()`
- `distinct()`: `.keep_all = TRUE` not supported
- `explain()`
- `filter()`

- full_join(): the copy argument is ignored
- glimpse()
- group_by()
- group_by_drop_default()
- group_vars()
- groups()
- inner_join(): the copy argument is ignored
- left_join(): the copy argument is ignored
- mutate()
- pull(): the name argument is not supported; returns an R vector by default but this behavior is deprecated and will return an Arrow ChunkedArray in a future release. Provide as_vector = TRUE/FALSE to control this behavior, or set options(arrow.pull_as_vector) globally.
- relocate()
- rename()
- rename_with()
- right_join(): the copy argument is ignored
- select()
- semi_join(): the copy argument is ignored
- show_query()
- slice_head(): slicing within groups not supported; Arrow datasets do not have row order, so head is non-deterministic; prop only supported on queries where nrow() is knowable without evaluating
- slice_max(): slicing within groups not supported; with_ties = TRUE (dplyr default) is not supported; prop only supported on queries where nrow() is knowable without evaluating
- slice_min(): slicing within groups not supported; with_ties = TRUE (dplyr default) is not supported; prop only supported on queries where nrow() is knowable without evaluating
- slice_sample(): slicing within groups not supported; replace = TRUE and the weight_by argument not supported; n only supported on queries where nrow() is knowable without evaluating
- slice_tail(): slicing within groups not supported; Arrow datasets do not have row order, so tail is non-deterministic; prop only supported on queries where nrow() is knowable without evaluating
- summarise(): window functions not currently supported; arguments .drop = FALSE and .groups = "rowwise" not supported
- tally()
- transmute()
- ungroup()
- union()
- union_all()

Function mappings

In the list below, any differences in behavior or support between Acero and the R function are listed. If no notes follow the function name, then you can assume that the function works in Acero just as it does in R.

Functions can be called either as `pkg::fun()` or just `fun()`, i.e. both `str_sub()` and `stringr::str_sub()` work.

In addition to these functions, you can call any of Arrow's 262 compute functions directly. Arrow has many functions that don't map to an existing R function. In other cases where there is an R function mapping, you can still call the Arrow function directly if you don't want the adaptations that the R mapping has that make Acero behave like R. These functions are listed in the C++ documentation, and in the function registry in R, they are named with an `arrow_` prefix, such as `arrow_ascii_is_decimal`.

arrow:

- `add_filename()`
- `cast()`

base:

- `!`
- `!=`
- `%%`
- `%/%`
- `%in%`
- `&`
- `*`
- `+`
- `-`
- `/`
- `<`
- `<=`
- `==`
- `>`
- `>=`
- `ISOdate()`
- `ISOdatetime()`
- `^`
- `abs()`
- `acos()`
- `all()`
- `any()`
- `as.Date()`: Multiple `tryFormats` not supported in Arrow. Consider using the lubridate specialised parsing functions `ymd()`, `ymd()`, etc.
- `as.character()`

- as.difftime(): only supports units = "secs" (the default)
- as.double()
- as.integer()
- as.logical()
- as.numeric()
- asin()
- ceiling()
- cos()
- data.frame(): row.names and check.rows arguments not supported; stringsAsFactors must be FALSE
- difftime(): only supports units = "secs" (the default); tz argument not supported
- endsWith()
- exp()
- floor()
- format()
- grepl()
- gsub()
- ifelse()
- is.character()
- is.double()
- is.factor()
- is.finite()
- is.infinite()
- is.integer()
- is.list()
- is.logical()
- is.na()
- is.nan()
- is.numeric()
- log()
- log10()
- log1p()
- log2()
- logb()
- max()
- mean()
- min()
- nchar(): allowNA = TRUE and keepNA = TRUE not supported
- paste(): the collapse argument is not yet supported
- paste0(): the collapse argument is not yet supported
- pmax()
- pmin()

- prod()
- round()
- sign()
- sin()
- sqrt()
- startsWith()
- strftime()
- strptime(): accepts a unit argument not present in the base function. Valid values are "s", "ms" (default), "us", "ns".
- strrep()
- strsplit()
- sub()
- substr(): start and stop must be length 1
- substring()
- sum()
- tan()
- tolower()
- toupper()
- trunc()
- |

bit64:

- as.integer64()
- is.integer64()

dplyr:

- across()
- between()
- case_when(): .ptype and .size arguments not supported
- coalesce()
- desc()
- if_all()
- if_any()
- if_else()
- n()
- n_distinct()

lubridate:

- am()
- as_date()
- as_datetime()
- ceiling_date()
- date()

- date_decimal()
- day()
- ddays()
- decimal_date()
- dhours()
- dmicroseconds()
- dmilliseconds()
- dminutes()
- dmonths()
- dmy(): locale argument not supported
- dmy_h(): locale argument not supported
- dmy_hm(): locale argument not supported
- dmy_hms(): locale argument not supported
- dnanoseconds()
- dpicoseconds(): not supported
- dseconds()
- dst()
- dweeks()
- dyears()
- dym(): locale argument not supported
- epiweek()
- epiyear()
- fast_strptime(): non-default values of lt and cutoff_2000 not supported
- floor_date()
- force_tz(): Timezone conversion from non-UTC timezone not supported; roll_dst values of 'error' and 'boundary' are supported for nonexistent times, roll_dst values of 'error', 'pre', and 'post' are supported for ambiguous times.
- format_ISO8601()
- hour()
- is.Date()
- is.POSIXct()
- is.instant()
- is.timepoint()
- isoweek()
- isoyear()
- leap_year()
- make_date()
- make_datetime(): only supports UTC (default) timezone
- make_difftime(): only supports units = "secs" (the default); providing both num and ... is not supported
- mday()
- mdy(): locale argument not supported

- mdy_h(): locale argument not supported
- mdy_hm(): locale argument not supported
- mdy_hms(): locale argument not supported
- minute()
- month()
- my(): locale argument not supported
- myd(): locale argument not supported
- parse_date_time(): quiet = FALSE is not supported Available formats are H, I, j, M, S, U, w, W, y, Y, R, T. On Linux and OS X additionally a, A, b, B, Om, p, r are available.
- pm()
- qday()
- quarter()
- round_date()
- second()
- semester()
- tz()
- wday()
- week()
- with_tz()
- yday()
- ydm(): locale argument not supported
- ydm_h(): locale argument not supported
- ydm_hm(): locale argument not supported
- ydm_hms(): locale argument not supported
- year()
- ym(): locale argument not supported
- ymd(): locale argument not supported
- ymd_h(): locale argument not supported
- ymd_hm(): locale argument not supported
- ymd_hms(): locale argument not supported
- yq(): locale argument not supported

methods:

- is()

rlang:

- is_character()
- is_double()
- is_integer()
- is_list()
- is_logical()

stats:

- median(): approximate median (t-digest) is computed
- quantile(): probs must be length 1; approximate quantile (t-digest) is computed
- sd()
- var()

stringi:

- stri_reverse()

stringr:

Pattern modifiers coll() and boundary() are not supported in any functions.

- str_c(): the collapse argument is not yet supported
- str_count(): pattern must be a length 1 character vector
- str_detect()
- str_dup()
- str_ends()
- str_length()
- str_like()
- str_pad()
- str_remove()
- str_remove_all()
- str_replace()
- str_replace_all()
- str_split(): Case-insensitive string splitting and splitting into 0 parts not supported
- str_starts()
- str_sub(): start and end must be length 1
- str_to_lower()
- str_to_title()
- str_to_upper()
- str_trim()

tibble:

- tibble()

tidyselect:

- all_of()
- contains()
- ends_with()
- everything()
- last_col()
- matches()
- num_range()
- one_of()
- starts_with()

`Array` *Array Classes*

Description

An `Array` is an immutable data array with some logical type and some length. Most logical types are contained in the base `Array` class; there are also subclasses for `DictionaryArray`, `ListArray`, and `StructArray`.

Factory

The `Array$create()` factory method instantiates an `Array` and takes the following arguments:

- `x`: an R vector, list, or `data.frame`
- `type`: an optional data type for `x`. If omitted, the type will be inferred from the data.

`Array$create()` will return the appropriate subclass of `Array`, such as `DictionaryArray` when given an R factor.

To compose a `DictionaryArray` directly, call `DictionaryArray$create()`, which takes two arguments:

- `x`: an R vector or `Array` of integers for the dictionary indices
- `dict`: an R vector or `Array` of dictionary values (like R factor levels but not limited to strings only)

Usage

```
a <- Array$create(x)
length(a)

print(a)
a == a
```

Methods

- `$IsNull(i)`: Return true if value at index is null. Does not boundscheck
- `$IsValid(i)`: Return true if value at index is valid. Does not boundscheck
- `$length()`: Size in the number of elements this array contains
- `$nbytes()`: Total number of bytes consumed by the elements of the array
- `$offset`: A relative position into another array's data, to enable zero-copy slicing
- `$null_count`: The number of null entries in the array
- `$type`: logical type of data
- `$type_id()`: type id
- `$Equals(other)` : is this array equal to `other`
- `$ApproxEquals(other)` :
- `$Diff(other)` : return a string expressing the difference between two arrays

- `$data()`: return the underlying ArrayData
- `$as_vector()`: convert to an R vector
- `$ToString()`: string representation of the array
- `$Slice(offset, length = NULL)`: Construct a zero-copy slice of the array with the indicated offset and length. If length is `NULL`, the slice goes until the end of the array.
- `$Take(i)`: return an `Array` with values at positions given by integers (R vector or Array Array) `i`.
- `$Filter(i, keep_na = TRUE)`: return an `Array` with values at positions where logical vector (or Arrow boolean Array) `i` is `TRUE`.
- `$SortIndices(descending = FALSE)`: return an `Array` of integer positions that can be used to rearrange the `Array` in ascending or descending order
- `$RangeEquals(other, start_idx, end_idx, other_start_idx)` :
- `$cast(target_type, safe = TRUE, options = cast_options(safe))`: Alter the data in the array to change its type.
- `$View(type)`: Construct a zero-copy view of this array with the given type.
- `$Validate()` : Perform any validation checks to determine obvious inconsistencies within the array's internal data. This can be an expensive check, potentially `O(length)`

Examples

```
my_array <- Array$create(1:10)
my_array$type
my_array$cast(int8())

# Check if value is null; zero-indexed
na_array <- Array$create(c(1:5, NA))
na_array$IsNull(0)
na_array$IsNull(5)
na_array$IsValid(5)
na_array$null_count

# zero-copy slicing; the offset of the new Array will be the same as the index passed to $Slice
new_array <- na_array$Slice(5)
new_array$offset

# Compare 2 arrays
na_array2 <- na_array
na_array2 == na_array # element-wise comparison
na_array2$Equals(na_array) # overall comparison
```

`ArrayData` *ArrayData class*

Description

The `ArrayData` class allows you to get and inspect the data inside an `arrow::Array`.

Usage

```
data <- Array$create(x)$data()

data$type
data$length
data$null_count
data$offset
data$buffers
```

Methods

...

`arrow_array` *Create an Arrow Array*

Description

Create an Arrow Array

Usage

```
arrow_array(x, type = NULL)
```

Arguments

`x`	An R object representable as an Arrow array, e.g. a vector, list, or `data.frame`.
`type`	An optional data type for `x`. If omitted, the type will be inferred from the data.

Examples

```
my_array <- arrow_array(1:10)

# Compare 2 arrays
na_array <- arrow_array(c(1:5, NA))
na_array2 <- na_array
na_array2 == na_array # element-wise comparison
```

arrow_info *Report information on the package's capabilities*

Description

This function summarizes a number of build-time configurations and run-time settings for the Arrow package. It may be useful for diagnostics.

Usage

```
arrow_info()

arrow_available()

arrow_with_acero()

arrow_with_dataset()

arrow_with_substrait()

arrow_with_parquet()

arrow_with_s3()

arrow_with_gcs()

arrow_with_json()
```

Value

`arrow_info()` returns a list including version information, boolean "capabilities", and statistics from Arrow's memory allocator, and also Arrow's run-time information. The `_available()` functions return a logical value whether or not the C++ library was built with support for them.

See Also

If any capabilities are `FALSE`, see the install guide for guidance on reinstalling the package.

arrow_table *Create an Arrow Table*

Description

Create an Arrow Table

Usage

```
arrow_table(..., schema = NULL)
```

Arguments

...	A data.frame or a named set of Arrays or vectors. If given a mixture of data.frames and named vectors, the inputs will be autospliced together (see examples). Alternatively, you can provide a single Arrow IPC InputStream, Message, Buffer, or R raw object containing a Buffer.
schema	a Schema, or NULL (the default) to infer the schema from the data in .... When providing an Arrow IPC buffer, schema is required.

See Also

Table

Examples

```
tbl <- arrow_table(name = rownames(mtcars), mtcars)
dim(tbl)
dim(head(tbl))
names(tbl)
tbl$mpg
tbl[["cyl"]]
as.data.frame(tbl[4:8, c("gear", "hp", "wt")])
```

---

as_arrow_array *Convert an object to an Arrow Array*

Description

The as_arrow_array() function is identical to Array$create() except that it is an S3 generic, which allows methods to be defined in other packages to convert objects to Array. Array$create() is slightly faster because it tries to convert in C++ before falling back on as_arrow_array().

Usage

```
as_arrow_array(x, ..., type = NULL)

## S3 method for class 'Array'
as_arrow_array(x, ..., type = NULL)

## S3 method for class 'Scalar'
as_arrow_array(x, ..., type = NULL)

## S3 method for class 'ChunkedArray'
as_arrow_array(x, ..., type = NULL)
```

Arguments

x	An object to convert to an Arrow Array
...	Passed to S3 methods
type	A type for the final Array. A value of NULL will default to the type guessed by infer_type().

Value

An Array with type type.

Examples

```
as_arrow_array(1:5)
```

---

as_arrow_table *Convert an object to an Arrow Table*

Description

Whereas arrow_table() constructs a table from one or more columns, as_arrow_table() converts a single object to an Arrow Table.

Usage

```
as_arrow_table(x, ..., schema = NULL)

## Default S3 method:
as_arrow_table(x, ...)

## S3 method for class 'Table'
as_arrow_table(x, ..., schema = NULL)

## S3 method for class 'RecordBatch'
as_arrow_table(x, ..., schema = NULL)

## S3 method for class 'data.frame'
as_arrow_table(x, ..., schema = NULL)

## S3 method for class 'RecordBatchReader'
as_arrow_table(x, ...)

## S3 method for class 'Dataset'
as_arrow_table(x, ...)

## S3 method for class 'arrow_dplyr_query'
```

```
as_arrow_table(x, ...)

## S3 method for class 'Schema'
as_arrow_table(x, ...)
```

Arguments

x	An object to convert to an Arrow Table
...	Passed to S3 methods
schema	a Schema, or NULL (the default) to infer the schema from the data in .... When providing an Arrow IPC buffer, schema is required.

Value

A Table

Examples

```
# use as_arrow_table() for a single object
as_arrow_table(data.frame(col1 = 1, col2 = "two"))

# use arrow_table() to create from columns
arrow_table(col1 = 1, col2 = "two")
```

---

as_chunked_array *Convert an object to an Arrow ChunkedArray*

Description

Whereas chunked_array() constructs a ChunkedArray from zero or more Arrays or R vectors, as_chunked_array() converts a single object to a ChunkedArray.

Usage

```
as_chunked_array(x, ..., type = NULL)

## S3 method for class 'ChunkedArray'
as_chunked_array(x, ..., type = NULL)

## S3 method for class 'Array'
as_chunked_array(x, ..., type = NULL)
```

Arguments

x	An object to convert to an Arrow Chunked Array
...	Passed to S3 methods
type	A type for the final Array. A value of NULL will default to the type guessed by infer_type().

Value

A ChunkedArray.

Examples

```
as_chunked_array(1:5)
```

`as_data_type` *Convert an object to an Arrow DataType*

Description

Convert an object to an Arrow DataType

Usage

```
as_data_type(x, ...)

## S3 method for class 'DataType'
as_data_type(x, ...)

## S3 method for class 'Field'
as_data_type(x, ...)

## S3 method for class 'Schema'
as_data_type(x, ...)
```

Arguments

`x`	An object to convert to an Arrow DataType
`...`	Passed to S3 methods.

Value

A DataType object.

Examples

```
as_data_type(int32())
```

as_record_batch *Convert an object to an Arrow RecordBatch*

Description

Whereas record_batch() constructs a RecordBatch from one or more columns, as_record_batch() converts a single object to an Arrow RecordBatch.

Usage

```
as_record_batch(x, ..., schema = NULL)

## S3 method for class 'RecordBatch'
as_record_batch(x, ..., schema = NULL)

## S3 method for class 'Table'
as_record_batch(x, ..., schema = NULL)

## S3 method for class 'arrow_dplyr_query'
as_record_batch(x, ...)

## S3 method for class 'data.frame'
as_record_batch(x, ..., schema = NULL)
```

Arguments

x	An object to convert to an Arrow RecordBatch
...	Passed to S3 methods
schema	a Schema, or NULL (the default) to infer the schema from the data in .... When providing an Arrow IPC buffer, schema is required.

Value

A RecordBatch

Examples

```
# use as_record_batch() for a single object
as_record_batch(data.frame(col1 = 1, col2 = "two"))

# use record_batch() to create from columns
record_batch(col1 = 1, col2 = "two")
```

`as_record_batch_reader` *Convert an object to an Arrow RecordBatchReader*

Description

Convert an object to an Arrow RecordBatchReader

Usage

```
as_record_batch_reader(x, ...)

## S3 method for class 'RecordBatchReader'
as_record_batch_reader(x, ...)

## S3 method for class 'Table'
as_record_batch_reader(x, ...)

## S3 method for class 'RecordBatch'
as_record_batch_reader(x, ...)

## S3 method for class 'data.frame'
as_record_batch_reader(x, ...)

## S3 method for class 'Dataset'
as_record_batch_reader(x, ...)

## S3 method for class '`function`'
as_record_batch_reader(x, ..., schema)

## S3 method for class 'arrow_dplyr_query'
as_record_batch_reader(x, ...)

## S3 method for class 'Scanner'
as_record_batch_reader(x, ...)
```

Arguments

`x`	An object to convert to a RecordBatchReader
`...`	Passed to S3 methods
`schema`	The `schema()` that must match the schema returned by each call to `x` when `x` is a function.

Value

A RecordBatchReader

Examples

```
reader <- as_record_batch_reader(data.frame(col1 = 1, col2 = "two"))
reader$read_next_batch()
```

---

as_schema *Convert an object to an Arrow Schema*

Description

Convert an object to an Arrow Schema

Usage

```
as_schema(x, ...)

## S3 method for class 'Schema'
as_schema(x, ...)

## S3 method for class 'StructType'
as_schema(x, ...)
```

Arguments

x	An object to convert to a schema()
...	Passed to S3 methods.

Value

A Schema object.

Examples

```
as_schema(schema(col1 = int32()))
```

Buffer *Buffer class*

Description

A Buffer is an object containing a pointer to a piece of contiguous memory with a particular size.

Factory

`buffer()` lets you create an `arrow::Buffer` from an R object

Methods

- `$is_mutable` : is this buffer mutable?
- `$ZeroPadding()` : zero bytes in padding, i.e. bytes between size and capacity
- `$size` : size in memory, in bytes
- `$capacity`: possible capacity, in bytes

Examples

```
my_buffer <- buffer(c(1, 2, 3, 4))
my_buffer$is_mutable
my_buffer$ZeroPadding()
my_buffer$size
my_buffer$capacity
```

buffer *Create a Buffer*

Description

Create a Buffer

Usage

```
buffer(x)
```

Arguments

x	R object. Only raw, numeric and integer vectors are currently supported

Value

an instance of `Buffer` that borrows memory from `x`

See Also

Buffer

call_function *Call an Arrow compute function*

Description

This function provides a lower-level API for calling Arrow functions by their string function name. You won't use it directly for most applications. Many Arrow compute functions are mapped to R methods, and in a `dplyr` evaluation context, all Arrow functions are callable with an `arrow_` prefix.

Usage

```
call_function(
  function_name,
  ...,
  args = list(...),
  options = empty_named_list()
)
```

Arguments

`function_name`	string Arrow compute function name
`...`	Function arguments, which may include `Array`, `ChunkedArray`, `Scalar`, `RecordBatch`, or `Table`.
`args`	list arguments as an alternative to specifying in `...`
`options`	named list of C++ function options.

Details

When passing indices in `...`, `args`, or `options`, express them as 0-based integers (consistent with C++).

Value

An `Array`, `ChunkedArray`, `Scalar`, `RecordBatch`, or `Table`, whatever the compute function results in.

See Also

Arrow C++ documentation for the functions and their respective options.

Examples

```
a <- Array$create(c(1L, 2L, 3L, NA, 5L))
s <- Scalar$create(4L)
call_function("coalesce", a, s)

a <- Array$create(rnorm(10000))
call_function("quantile", a, options = list(q = seq(0, 1, 0.25)))
```

ChunkedArray *ChunkedArray class*

Description

A ChunkedArray is a data structure managing a list of primitive Arrow Arrays logically as one large array. Chunked arrays may be grouped together in a Table.

Factory

The ChunkedArray$create() factory method instantiates the object from various Arrays or R vectors. chunked_array() is an alias for it.

Methods

- $length(): Size in the number of elements this array contains
- $chunk(i): Extract an Array chunk by integer position
- '$nbytes() : Total number of bytes consumed by the elements of the array
- $as_vector(): convert to an R vector
- $Slice(offset, length = NULL): Construct a zero-copy slice of the array with the indicated offset and length. If length is NULL, the slice goes until the end of the array.
- $Take(i): return a ChunkedArray with values at positions given by integers i. If i is an Arrow Array or ChunkedArray, it will be coerced to an R vector before taking.
- $Filter(i, keep_na = TRUE): return a ChunkedArray with values at positions where logical vector or Arrow boolean-type (Chunked)Array i is TRUE.
- $SortIndices(descending = FALSE): return an Array of integer positions that can be used to rearrange the ChunkedArray in ascending or descending order
- $cast(target_type, safe = TRUE, options = cast_options(safe)): Alter the data in the array to change its type.
- $null_count: The number of null entries in the array
- $chunks: return a list of Arrays
- $num_chunks: integer number of chunks in the ChunkedArray
- $type: logical type of data
- $View(type): Construct a zero-copy view of this ChunkedArray with the given type.
- $Validate(): Perform any validation checks to determine obvious inconsistencies within the array's internal data. This can be an expensive check, potentially O(length)

See Also

Array

Examples

```
# Pass items into chunked_array as separate objects to create chunks
class_scores <- chunked_array(c(87, 88, 89), c(94, 93, 92), c(71, 72, 73))
class_scores$num_chunks

# When taking a Slice from a chunked_array, chunks are preserved
class_scores$Slice(2, length = 5)

# You can combine Take and SortIndices to return a ChunkedArray with 1 chunk
# containing all values, ordered.
class_scores$Take(class_scores$SortIndices(descending = TRUE))

# If you pass a list into chunked_array, you get a list of length 1
list_scores <- chunked_array(list(c(9.9, 9.6, 9.5), c(8.2, 8.3, 8.4), c(10.0, 9.9, 9.8)))
list_scores$num_chunks

# When constructing a ChunkedArray, the first chunk is used to infer type.
doubles <- chunked_array(c(1, 2, 3), c(5L, 6L, 7L))
doubles$type

# Concatenating chunked arrays returns a new chunked array containing all chunks
a <- chunked_array(c(1, 2), 3)
b <- chunked_array(c(4, 5), 6)
c(a, b)
```

---

chunked_array *Create a Chunked Array*

Description

Create a Chunked Array

Usage

```
chunked_array(..., type = NULL)
```

Arguments

...	R objects to coerce into a ChunkedArray. They must be of the same type.
type	An optional data type. If omitted, the type will be inferred from the data.

See Also

ChunkedArray

Examples

```
# Pass items into chunked_array as separate objects to create chunks
class_scores <- chunked_array(c(87, 88, 89), c(94, 93, 92), c(71, 72, 73))

# If you pass a list into chunked_array, you get a list of length 1
list_scores <- chunked_array(list(c(9.9, 9.6, 9.5), c(8.2, 8.3, 8.4), c(10.0, 9.9, 9.8)))

# When constructing a ChunkedArray, the first chunk is used to infer type.
infer_type(chunked_array(c(1, 2, 3), c(5L, 6L, 7L)))

# Concatenating chunked arrays returns a new chunked array containing all chunks
a <- chunked_array(c(1, 2), 3)
b <- chunked_array(c(4, 5), 6)
c(a, b)
```

Codec *Compression Codec class*

Description

Codecs allow you to create compressed input and output streams.

Factory

The `Codec$create()` factory method takes the following arguments:

- `type`: string name of the compression method. Possible values are "uncompressed", "snappy", "gzip", "brotli", "zstd", "lz4", "lzo", or "bz2". `type` may be upper- or lower-cased. Not all methods may be available; support depends on build-time flags for the C++ library. See `codec_is_available()`. Most builds support at least "snappy" and "gzip". All support "uncompressed".
- `compression_level`: compression level, the default value (`NA`) uses the default compression level for the selected compression `type`.

---

codec_is_available *Check whether a compression codec is available*

Description

Support for compression libraries depends on the build-time settings of the Arrow C++ library. This function lets you know which are available for use.

Usage

```
codec_is_available(type)
```

Arguments

type	A string, one of "uncompressed", "snappy", "gzip", "brotli", "zstd", "lz4", "lzo", or "bz2", case-insensitive.

Value

Logical: is type available?

Examples

```
codec_is_available("gzip")
```

---

compression *Compressed stream classes*

Description

CompressedInputStream and CompressedOutputStream allow you to apply a compression Codec to an input or output stream.

Factory

The CompressedInputStream$create() and CompressedOutputStream$create() factory methods instantiate the object and take the following arguments:

- stream An InputStream or OutputStream, respectively
- codec A Codec, either a Codec instance or a string
- compression_level compression level for when the codec argument is given as a string

Methods

Methods are inherited from InputStream and OutputStream, respectively

---

concat_arrays *Concatenate zero or more Arrays*

Description

Concatenates zero or more Array objects into a single array. This operation will make a copy of its input; if you need the behavior of a single Array but don't need a single object, use ChunkedArray.

Usage

```
concat_arrays(..., type = NULL)

## S3 method for class 'Array'
c(...)
```

Arguments

...	zero or more Array objects to concatenate
type	An optional type describing the desired type for the final Array.

Value

A single Array

Examples

```
concat_arrays(Array$create(1:3), Array$create(4:5))
```

---

concat_tables *Concatenate one or more Tables*

Description

Concatenate one or more Table objects into a single table. This operation does not copy array data, but instead creates new chunked arrays for each column that point at existing array data.

Usage

```
concat_tables(..., unify_schemas = TRUE)
```

Arguments

...	A Table
unify_schemas	If TRUE, the schemas of the tables will be first unified with fields of the same name being merged, then each table will be promoted to the unified schema before being concatenated. Otherwise, all tables should have the same schema.

Examples

```
tbl <- arrow_table(name = rownames(mtcars), mtcars)
prius <- arrow_table(name = "Prius", mpg = 58, cyl = 4, disp = 1.8)
combined <- concat_tables(tbl, prius)
tail(combined)$to_data_frame()
```

copy_files *Copy files between FileSystems*

Description

Copy files between FileSystems

Usage

```
copy_files(from, to, chunk_size = 1024L * 1024L)
```

Arguments

from	A string path to a local directory or file, a URI, or a SubTreeFileSystem. Files will be copied recursively from this path.
to	A string path to a local directory or file, a URI, or a SubTreeFileSystem. Directories will be created as necessary
chunk_size	The maximum size of block to read before flushing to the destination file. A larger chunk_size will use more memory while copying but may help accommodate high latency FileSystems.

Value

Nothing: called for side effects in the file system

Examples

```
# Copy an S3 bucket's files to a local directory:
copy_files("s3://your-bucket-name", "local-directory")
# Using a FileSystem object
copy_files(s3_bucket("your-bucket-name"), "local-directory")
# Or go the other way, from local to S3
copy_files("local-directory", s3_bucket("your-bucket-name"))
```

cpu_count *Manage the global CPU thread pool in libarrow*

Description

Manage the global CPU thread pool in libarrow

Usage

```
cpu_count()

set_cpu_count(num_threads)
```

Arguments

num_threads	integer: New number of threads for thread pool

---

create_package_with_all_dependencies

Create a source bundle that includes all thirdparty dependencies

---

Description

Create a source bundle that includes all thirdparty dependencies

Usage

```
create_package_with_all_dependencies(dest_file = NULL, source_file = NULL)
```

Arguments

dest_file	File path for the new tar.gz package. Defaults to arrow_V.V.V_with_deps.tar.gz in the current directory (V.V.V is the version)
source_file	File path for the input tar.gz package. Defaults to downloading the package from CRAN (or whatever you have set as the first in getOption("repos"))

Value

The full path to dest_file, invisibly

This function is used for setting up an offline build. If it's possible to download at build time, don't use this function. Instead, let cmake download the required dependencies for you. These downloaded dependencies are only used in the build if ARROW_DEPENDENCY_SOURCE is unset, BUNDLED, or AUTO. https://arrow.apache.org/docs/developers/cpp/building.html#offline-builds

If you're using binary packages you shouldn't need to use this function. You should download the appropriate binary from your package repository, transfer that to the offline computer, and install that. Any OS can create the source bundle, but it cannot be installed on Windows. (Instead, use a standard Windows binary package.)

Note if you're using RStudio Package Manager on Linux: If you still want to make a source bundle with this function, make sure to set the first repo in options("repos") to be a mirror that contains source packages (that is: something other than the RSPM binary mirror URLs).

Steps for an offline install with optional dependencies::

Using a computer with internet access, pre-download the dependencies::

- Install the arrow package *or* run source("https://raw.githubusercontent.com/apache/arrow/main/r/R/ins
- Run create_package_with_all_dependencies("my_arrow_pkg.tar.gz")
- Copy the newly created my_arrow_pkg.tar.gz to the computer without internet access

On the computer without internet access, install the prepared package::

- Install the arrow package from the copied file

  - install.packages("my_arrow_pkg.tar.gz", dependencies = c("Depends", "Imports", "LinkingTo"))
  - This installation will build from source, so cmake must be available
- Run arrow_info() to check installed capabilities

Examples

```
## Not run:
new_pkg <- create_package_with_all_dependencies()
# Note: this works when run in the same R session, but it's meant to be
# copied to a different computer.
install.packages(new_pkg, dependencies = c("Depends", "Imports", "LinkingTo"))

## End(Not run)
```

CsvFileFormat *CSV dataset file format*

Description

A CSVFileFormat is a FileFormat subclass which holds information about how to read and parse the files included in a CSV Dataset.

Value

A CsvFileFormat object

Factory

CSVFileFormat$create() can take options in the form of lists passed through as parse_options, read_options, or convert_options parameters. Alternatively, readr-style options can be passed through individually. While it is possible to pass in CSVReadOptions, CSVConvertOptions, and CSVParseOptions objects, this is not recommended as options set in these objects are not validated for compatibility.

See Also

FileFormat

Examples

```
# Set up directory for examples
tf <- tempfile()
dir.create(tf)
on.exit(unlink(tf))
df <- data.frame(x = c("1", "2", "NULL"))
write.table(df, file.path(tf, "file1.txt"), sep = ",", row.names = FALSE)

# Create CsvFileFormat object with Arrow-style null_values option
```

```
format <- CsvFileFormat$create(convert_options = list(null_values = c("", "NA", "NULL")))
open_dataset(tf, format = format)

# Use readr-style options
format <- CsvFileFormat$create(na = c("", "NA", "NULL"))
open_dataset(tf, format = format)
```

`CsvReadOptions` *File reader options*

Description

`CsvReadOptions`, `CsvParseOptions`, `CsvConvertOptions`, `JsonReadOptions`, `JsonParseOptions`, and `TimestampParser` are containers for various file reading options. See their usage in `read_csv_arrow()` and `read_json_arrow()`, respectively.

Factory

The `CsvReadOptions$create()` and `JsonReadOptions$create()` factory methods take the following arguments:

- `use_threads` Whether to use the global CPU thread pool
- `block_size` Block size we request from the IO layer; also determines the size of chunks when use_threads is `TRUE`. NB: if `FALSE`, JSON input must end with an empty line.

`CsvReadOptions$create()` further accepts these additional arguments:

- `skip_rows` Number of lines to skip before reading data (default 0).
- `column_names` Character vector to supply column names. If length-0 (the default), the first non-skipped row will be parsed to generate column names, unless `autogenerate_column_names` is `TRUE`.
- `autogenerate_column_names` Logical: generate column names instead of using the first non-skipped row (the default)? If `TRUE`, column names will be "f0", "f1", ..., "fN".
- `encoding` The file encoding. (default `"UTF-8"`)
- `skip_rows_after_names` Number of lines to skip after the column names (default 0). This number can be larger than the number of rows in one block, and empty rows are counted. The order of application is as follows:
  - `skip_rows` is applied (if non-zero);
  - column names are read (unless `column_names` is set);
  - `skip_rows_after_names` is applied (if non-zero).

`CsvParseOptions$create()` takes the following arguments:

- `delimiter` Field delimiting character (default `","`)
- `quoting` Logical: are strings quoted? (default `TRUE`)

- quote_char Quoting character, if quoting is TRUE (default '"')
- double_quote Logical: are quotes inside values double-quoted? (default TRUE)
- escaping Logical: whether escaping is used (default FALSE)
- escape_char Escaping character, if escaping is TRUE (default "\\")
- newlines_in_values Logical: are values allowed to contain CR (0x0d) and LF (0x0a) characters? (default FALSE)
- ignore_empty_lines Logical: should empty lines be ignored (default) or generate a row of missing values (if FALSE)?

JsonParseOptions$create() accepts only the newlines_in_values argument.

CsvConvertOptions$create() takes the following arguments:

- check_utf8 Logical: check UTF8 validity of string columns? (default TRUE)
- null_values character vector of recognized spellings for null values. Analogous to the na.strings argument to read.csv() or na in readr::read_csv().
- strings_can_be_null Logical: can string / binary columns have null values? Similar to the quoted_na argument to readr::read_csv(). (default FALSE)
- true_values character vector of recognized spellings for TRUE values
- false_values character vector of recognized spellings for FALSE values
- col_types A Schema or NULL to infer types
- auto_dict_encode Logical: Whether to try to automatically dictionary-encode string / binary data (think stringsAsFactors). Default FALSE. This setting is ignored for non-inferred columns (those in col_types).
- auto_dict_max_cardinality If auto_dict_encode, string/binary columns are dictionary-encoded up to this number of unique values (default 50), after which it switches to regular encoding.
- include_columns If non-empty, indicates the names of columns from the CSV file that should be actually read and converted (in the vector's order).
- include_missing_columns Logical: if include_columns is provided, should columns named in it but not found in the data be included as a column of type null()? The default (FALSE) means that the reader will instead raise an error.
- timestamp_parsers User-defined timestamp parsers. If more than one parser is specified, the CSV conversion logic will try parsing values starting from the beginning of this vector. Possible values are (a) NULL, the default, which uses the ISO-8601 parser; (b) a character vector of strptime parse strings; or (c) a list of TimestampParser objects.
- decimal_point Character to use for decimal point in floating point numbers. Default: "."

TimestampParser$create() takes an optional format string argument. See strptime() for example syntax. The default is to use an ISO-8601 format parser.

The CsvWriteOptions$create() factory method takes the following arguments:

- include_header Whether to write an initial header line with column names
- batch_size Maximum number of rows processed at a time. Default is 1024.

- `null_string` The string to be written for null values. Must not contain quotation marks. Default is an empty string (`""`).
- `eol` The end of line character to use for ending rows.
- `delimiter` Field delimiter
- `quoting_style` Quoting style: "Needed" (Only enclose values in quotes which need them, because their CSV rendering can contain quotes itself (e.g. strings or binary values)), "AllValid" (Enclose all valid values in quotes), or "None" (Do not enclose any values in quotes).

Active bindings

- `column_names`: from `CsvReadOptions`

`CsvTableReader` *Arrow CSV and JSON table reader classes*

Description

`CsvTableReader` and `JsonTableReader` wrap the Arrow C++ CSV and JSON table readers. See their usage in `read_csv_arrow()` and `read_json_arrow()`, respectively.

Factory

The `CsvTableReader$create()` and `JsonTableReader$create()` factory methods take the following arguments:

- `file` An Arrow InputStream
- `convert_options` (CSV only), `parse_options`, `read_options`: see CsvReadOptions
- `...` additional parameters.

Methods

- `$Read()`: returns an Arrow Table.

`csv_convert_options` *CSV Convert Options*

Description

CSV Convert Options

Usage

```
csv_convert_options(
  check_utf8 = TRUE,
  null_values = c("", "NA"),
  true_values = c("T", "true", "TRUE"),
  false_values = c("F", "false", "FALSE"),
  strings_can_be_null = FALSE,
  col_types = NULL,
  auto_dict_encode = FALSE,
  auto_dict_max_cardinality = 50L,
  include_columns = character(),
  include_missing_columns = FALSE,
  timestamp_parsers = NULL,
  decimal_point = "."
)
```

Arguments

check_utf8	Logical: check UTF8 validity of string columns?
null_values	Character vector of recognized spellings for null values. Analogous to the na.strings argument to read.csv() or na in readr::read_csv().
true_values	Character vector of recognized spellings for TRUE values
false_values	Character vector of recognized spellings for FALSE values
strings_can_be_null	Logical: can string / binary columns have null values? Similar to the quoted_na argument to readr::read_csv()
col_types	A Schema or NULL to infer types
auto_dict_encode	Logical: Whether to try to automatically dictionary-encode string / binary data (think stringsAsFactors). This setting is ignored for non-inferred columns (those in col_types).
auto_dict_max_cardinality	If auto_dict_encode, string/binary columns are dictionary-encoded up to this number of unique values (default 50), after which it switches to regular encoding.
include_columns	If non-empty, indicates the names of columns from the CSV file that should be actually read and converted (in the vector's order).
include_missing_columns	Logical: if include_columns is provided, should columns named in it but not found in the data be included as a column of type null()? The default (FALSE) means that the reader will instead raise an error.
timestamp_parsers	User-defined timestamp parsers. If more than one parser is specified, the CSV conversion logic will try parsing values starting from the beginning of this vector. Possible values are (a) NULL, the default, which uses the ISO-8601 parser;

(b) a character vector of strptime parse strings; or (c) a list of TimestampParser objects.

`decimal_point` Character to use for decimal point in floating point numbers.

Examples

```
tf <- tempfile()
on.exit(unlink(tf))
writeLines("x\n1\nNULL\n2\nNA", tf)
read_csv_arrow(tf, convert_options = csv_convert_options(null_values = c("", "NA", "NULL")))
open_csv_dataset(tf, convert_options = csv_convert_options(null_values = c("", "NA", "NULL")))
```

---

csv_parse_options *CSV Parsing Options*

Description

CSV Parsing Options

Usage

```
csv_parse_options(
  delimiter = ",",
  quoting = TRUE,
  quote_char = "\"",
  double_quote = TRUE,
  escaping = FALSE,
  escape_char = "\\",
  newlines_in_values = FALSE,
  ignore_empty_lines = TRUE
)
```

Arguments

`delimiter` Field delimiting character

`quoting` Logical: are strings quoted?

`quote_char` Quoting character, if `quoting` is `TRUE`

`double_quote` Logical: are quotes inside values double-quoted?

`escaping` Logical: whether escaping is used

`escape_char` Escaping character, if `escaping` is `TRUE`

`newlines_in_values`
Logical: are values allowed to contain CR (`0x0d`) and LF (`0x0a`) characters?

`ignore_empty_lines`
Logical: should empty lines be ignored (default) or generate a row of missing values (if `FALSE`)?

Examples

```
tf <- tempfile()
on.exit(unlink(tf))
writeLines("x\n1\n\n2", tf)
read_csv_arrow(tf, parse_options = csv_parse_options(ignore_empty_lines = FALSE))
open_csv_dataset(tf, parse_options = csv_parse_options(ignore_empty_lines = FALSE))
```

---

csv_read_options *CSV Reading Options*

Description

CSV Reading Options

Usage

```
csv_read_options(
  use_threads = option_use_threads(),
  block_size = 1048576L,
  skip_rows = 0L,
  column_names = character(0),
  autogenerate_column_names = FALSE,
  encoding = "UTF-8",
  skip_rows_after_names = 0L
)
```

Arguments

use_threads	Whether to use the global CPU thread pool
block_size	Block size we request from the IO layer; also determines the size of chunks when use_threads is TRUE.
skip_rows	Number of lines to skip before reading data (default 0).
column_names	Character vector to supply column names. If length-0 (the default), the first non-skipped row will be parsed to generate column names, unless autogenerate_column_names is TRUE.
autogenerate_column_names	Logical: generate column names instead of using the first non-skipped row (the default)? If TRUE, column names will be "f0", "f1", ..., "fN".
encoding	The file encoding. (default "UTF-8")
skip_rows_after_names	Number of lines to skip after the column names (default 0). This number can be larger than the number of rows in one block, and empty rows are counted. The order of application is as follows: - skip_rows is applied (if non-zero); - column names are read (unless column_names is set); - skip_rows_after_names is applied (if non-zero).

Examples

```
tf <- tempfile()
on.exit(unlink(tf))
writeLines("my file has a non-data header\nx\n1\n2", tf)
read_csv_arrow(tf, read_options = csv_read_options(skip_rows = 1))
open_csv_dataset(tf, read_options = csv_read_options(skip_rows = 1))
```

`csv_write_options` *CSV Writing Options*

Description

CSV Writing Options

Usage

```
csv_write_options(
  include_header = TRUE,
  batch_size = 1024L,
  null_string = "",
  delimiter = ",",
  eol = "\n",
  quoting_style = c("Needed", "AllValid", "None")
)
```

Arguments

`include_header`	Whether to write an initial header line with column names
`batch_size`	Maximum number of rows processed at a time.
`null_string`	The string to be written for null values. Must not contain quotation marks.
`delimiter`	Field delimiter
`eol`	The end of line character to use for ending rows
`quoting_style`	How to handle quotes. "Needed" (Only enclose values in quotes which need them, because their CSV rendering can contain quotes itself (e.g. strings or binary values)), "AllValid" (Enclose all valid values in quotes), or "None" (Do not enclose any values in quotes).

Examples

```
tf <- tempfile()
on.exit(unlink(tf))
write_csv_arrow(airquality, tf, write_options = csv_write_options(null_string = "-99"))
```

data-type *Create Arrow data types*

Description

These functions create type objects corresponding to Arrow types. Use them when defining a `schema()` or as inputs to other types, like `struct`. Most of these functions don't take arguments, but a few do.

Usage

```
int8()

int16()

int32()

int64()

uint8()

uint16()

uint32()

uint64()

float16()

halffloat()

float32()

float()

float64()

boolean()

bool()

utf8()

large_utf8()

binary()
```

```
large_binary()

fixed_size_binary(byte_width)

string()

date32()

date64()

time32(unit = c("ms", "s"))

time64(unit = c("ns", "us"))

duration(unit = c("s", "ms", "us", "ns"))

null()

timestamp(unit = c("s", "ms", "us", "ns"), timezone = "")

decimal(precision, scale)

decimal128(precision, scale)

decimal256(precision, scale)

struct(...)

list_of(type)

large_list_of(type)

fixed_size_list_of(type, list_size)

map_of(key_type, item_type, .keys_sorted = FALSE)
```

Arguments

byte_width	byte width for FixedSizeBinary type.
unit	For time/timestamp types, the time unit. time32() can take either "s" or "ms", while time64() can be "us" or "ns". timestamp() can take any of those four values.
timezone	For timestamp(), an optional time zone string.
precision	For decimal(), decimal128(), and decimal256() the number of significant digits the arrow decimal type can represent. The maximum precision for decimal128() is 38 significant digits, while for decimal256() it is 76 digits. decimal() will use it to choose which type of decimal to return.

scale	For decimal(), decimal128(), and decimal256() the number of digits after the decimal point. It can be negative.
...	For struct(), a named list of types to define the struct columns
type	For list_of(), a data type to make a list-of-type
list_size	list size for FixedSizeList type.
key_type, item_type	For MapType, the key and item types.
.keys_sorted	Use TRUE to assert that keys of a MapType are sorted.

Details

A few functions have aliases:

- utf8() and string()
- float16() and halffloat()
- float32() and float()
- bool() and boolean()
- When called inside an arrow function, such as schema() or cast(), double() also is supported as a way of creating a float64()

date32() creates a datetime type with a "day" unit, like the R Date class. date64() has a "ms" unit.

uint32 (32 bit unsigned integer), uint64 (64 bit unsigned integer), and int64 (64-bit signed integer) types may contain values that exceed the range of R's integer type (32-bit signed integer). When these arrow objects are translated to R objects, uint32 and uint64 are converted to double ("numeric") and int64 is converted to bit64::integer64. For int64 types, this conversion can be disabled (so that int64 always yields a bit64::integer64 object) by setting options(arrow.int64_downcast = FALSE).

decimal128() creates a Decimal128Type. Arrow decimals are fixed-point decimal numbers encoded as a scalar integer. The precision is the number of significant digits that the decimal type can represent; the scale is the number of digits after the decimal point. For example, the number 1234.567 has a precision of 7 and a scale of 3. Note that scale can be negative.

As an example, decimal128(7, 3) can exactly represent the numbers 1234.567 and -1234.567 (encoded internally as the 128-bit integers 1234567 and -1234567, respectively), but neither 12345.67 nor 123.4567.

decimal128(5, -3) can exactly represent the number 12345000 (encoded internally as the 128-bit integer 12345), but neither 123450000 nor 1234500. The scale can be thought of as an argument that controls rounding. When negative, scale causes the number to be expressed using scientific notation and power of 10.

decimal256() creates a Decimal256Type, which allows for higher maximum precision. For most use cases, the maximum precision offered by Decimal128Type is sufficient, and it will result in a more compact and more efficient encoding.

decimal() creates either a Decimal128Type or a Decimal256Type depending on the value for precision. If precision is greater than 38 a Decimal256Type is returned, otherwise a Decimal128Type.

Use decimal128() or decimal256() as the names are more informative than decimal().

Value

An Arrow type object inheriting from DataType.

See Also

`dictionary()` for creating a dictionary (factor-like) type.

Examples

```
bool()
struct(a = int32(), b = double())
timestamp("ms", timezone = "CEST")
time64("ns")

# Use the cast method to change the type of data contained in Arrow objects.
# Please check the documentation of each data object class for details.
my_scalar <- Scalar$create(0L, type = int64()) # int64
my_scalar$cast(timestamp("ns")) # timestamp[ns]

my_array <- Array$create(0L, type = int64()) # int64
my_array$cast(timestamp("s", timezone = "UTC")) # timestamp[s, tz=UTC]

my_chunked_array <- chunked_array(0L, 1L) # int32
my_chunked_array$cast(date32()) # date32[day]

# You can also use `cast()` in an Arrow dplyr query.
if (requireNamespace("dplyr", quietly = TRUE)) {
  library(dplyr, warn.conflicts = FALSE)
  arrow_table(mtcars) %>%
    transmute(
      col1 = cast(cyl, string()),
      col2 = cast(cyl, int8())
    ) %>%
    compute()
}
```

Dataset *Multi-file datasets*

Description

Arrow Datasets allow you to query against data that has been split across multiple files. This sharding of data may indicate partitioning, which can accelerate queries that only touch some partitions (files).

A `Dataset` contains one or more `Fragments`, such as files, of potentially differing type and partitioning.

For `Dataset$create()`, see `open_dataset()`, which is an alias for it.

`DatasetFactory` is used to provide finer control over the creation of `Datasets`.

Factory

DatasetFactory is used to create a Dataset, inspect the Schema of the fragments contained in it, and declare a partitioning. FileSystemDatasetFactory is a subclass of DatasetFactory for discovering files in the local file system, the only currently supported file system.

For the DatasetFactory$create() factory method, see dataset_factory(), an alias for it. A DatasetFactory has:

- $Inspect(unify_schemas): If unify_schemas is TRUE, all fragments will be scanned and a unified Schema will be created from them; if FALSE (default), only the first fragment will be inspected for its schema. Use this fast path when you know and trust that all fragments have an identical schema.
- $Finish(schema, unify_schemas): Returns a Dataset. If schema is provided, it will be used for the Dataset; if omitted, a Schema will be created from inspecting the fragments (files) in the dataset, following unify_schemas as described above.

FileSystemDatasetFactory$create() is a lower-level factory method and takes the following arguments:

- filesystem: A FileSystem
- selector: Either a FileSelector or NULL
- paths: Either a character vector of file paths or NULL
- format: A FileFormat
- partitioning: Either Partitioning, PartitioningFactory, or NULL

Methods

A Dataset has the following methods:

- $NewScan(): Returns a ScannerBuilder for building a query
- $WithSchema(): Returns a new Dataset with the specified schema. This method currently supports only adding, removing, or reordering fields in the schema: you cannot alter or cast the field types.
- $schema: Active binding that returns the Schema of the Dataset; you may also replace the dataset's schema by using ds$schema <- new_schema.

FileSystemDataset has the following methods:

- $files: Active binding, returns the files of the FileSystemDataset
- $format: Active binding, returns the FileFormat of the FileSystemDataset

UnionDataset has the following methods:

- $children: Active binding, returns all child Datasets.

See Also

open_dataset() for a simple interface to creating a Dataset

`dataset_factory` *Create a DatasetFactory*

Description

A Dataset can constructed using one or more DatasetFactorys. This function helps you construct a `DatasetFactory` that you can pass to `open_dataset()`.

Usage

```
dataset_factory(
  x,
  filesystem = NULL,
  format = c("parquet", "arrow", "ipc", "feather", "csv", "tsv", "text", "json"),
  partitioning = NULL,
  hive_style = NA,
  factory_options = list(),
  ...
)
```

Arguments

`x`	A string path to a directory containing data files, a vector of one one or more string paths to data files, or a list of `DatasetFactory` objects whose datasets should be combined. If this argument is specified it will be used to construct a `UnionDatasetFactory` and other arguments will be ignored.
`filesystem`	A FileSystem object; if omitted, the `FileSystem` will be detected from `x`
`format`	A FileFormat object, or a string identifier of the format of the files in `x`. Currently supported values: • "parquet" • "ipc"/"arrow"/"feather", all aliases for each other; for Feather, note that only version 2 files are supported • "csv"/"text", aliases for the same thing (because comma is the default delimiter for text files • "tsv", equivalent to passing `format = "text", delimiter = "\t"` Default is "parquet", unless a `delimiter` is also specified, in which case it is assumed to be "text".
`partitioning`	One of • A `Schema`, in which case the file paths relative to `sources` will be parsed, and path segments will be matched with the schema fields. For example, `schema(year = int16(), month = int8())` would create partitions for file paths like "2019/01/file.parquet", "2019/02/file.parquet", etc. • A character vector that defines the field names corresponding to those path segments (that is, you're providing the names that would correspond to a `Schema` but the types will be autodetected)

- A HivePartitioning or HivePartitioningFactory, as returned by hive_partition() which parses explicit or autodetected fields from Hive-style path segments
- NULL for no partitioning

hive_style Logical: if partitioning is a character vector or a Schema, should it be interpreted as specifying Hive-style partitioning? Default is NA, which means to inspect the file paths for Hive-style partitioning and behave accordingly.

factory_options

list of optional FileSystemFactoryOptions:

- partition_base_dir: string path segment prefix to ignore when discovering partition information with DirectoryPartitioning. Not meaningful (ignored with a warning) for HivePartitioning, nor is it valid when providing a vector of file paths.
- exclude_invalid_files: logical: should files that are not valid data files be excluded? Default is FALSE because checking all files up front incurs I/O and thus will be slower, especially on remote filesystems. If false and there are invalid files, there will be an error at scan time. This is the only FileSystemFactoryOption that is valid for both when providing a directory path in which to discover files and when providing a vector of file paths.
- selector_ignore_prefixes: character vector of file prefixes to ignore when discovering files in a directory. If invalid files can be excluded by a common filename prefix this way, you can avoid the I/O cost of exclude_invalid_files. Not valid when providing a vector of file paths (but if you're providing the file list, you can filter invalid files yourself).

... Additional format-specific options, passed to FileFormat$create(). For CSV options, note that you can specify them either with the Arrow C++ library naming ("delimiter", "quoting", etc.) or the readr-style naming used in read_csv_arrow() ("delim", "quote", etc.). Not all readr options are currently supported; please file an issue if you encounter one that arrow should support.

Details

If you would only have a single DatasetFactory (for example, you have a single directory containing Parquet files), you can call open_dataset() directly. Use dataset_factory() when you want to combine different directories, file systems, or file formats.

Value

A DatasetFactory object. Pass this to open_dataset(), in a list potentially with other DatasetFactory objects, to create a Dataset.

DataType *DataType class*

Description

DataType class

R6 Methods

- `$ToString()`: String representation of the DataType
- `$Equals(other)`: Is the DataType equal to `other`
- `$fields()`: The children fields associated with this type
- `$code(namespace)`: Produces an R call of the data type. Use `namespace=TRUE` to call with `arrow::`.

There are also some active bindings:

- `$id`: integer Arrow type id.
- `$name`: string Arrow type name.
- `$num_fields`: number of child fields.

See Also

`infer_type()`

`data-type`

`dictionary` *Create a dictionary type*

Description

Create a dictionary type

Usage

```
dictionary(index_type = int32(), value_type = utf8(), ordered = FALSE)
```

Arguments

`index_type`	A DataType for the indices (default `int32()`)
`value_type`	A DataType for the values (default `utf8()`)
`ordered`	Is this an ordered dictionary (default `FALSE`)?

Value

A DictionaryType

See Also

Other Arrow data types

DictionaryType *class DictionaryType*

Description

class DictionaryType

Methods

TODO

Expression *Arrow expressions*

Description

`Expressions` are used to define filter logic for passing to a Dataset Scanner.

`Expression$scalar(x)` constructs an `Expression` which always evaluates to the provided scalar (length-1) R value.

`Expression$field_ref(name)` is used to construct an `Expression` which evaluates to the named column in the `Dataset` against which it is evaluated.

`Expression$create(function_name, ..., options)` builds a function-call `Expression` containing one or more `Expressions`. Anything in `...` that is not already an expression will be wrapped in `Expression$scalar()`.

`Expression$op(FUN, ...)` is for logical and arithmetic operators. Scalar inputs in `...` will be attempted to be cast to the common type of the `Expressions` in the call so that the types of the columns in the `Dataset` are preserved and not unnecessarily upcast, which may be expensive.

ExtensionArray *ExtensionArray class*

Description

ExtensionArray class

Methods

The `ExtensionArray` class inherits from `Array`, but also provides access to the underlying storage of the extension.

- `$storage()`: Returns the underlying Array used to store values.

The `ExtensionArray` is not intended to be subclassed for extension types.

`ExtensionType` *ExtensionType class*

Description

ExtensionType class

Methods

The `ExtensionType` class inherits from `DataType`, but also defines extra methods specific to extension types:

- `$storage_type()`: Returns the underlying DataType used to store values.
- `$storage_id()`: Returns the Type identifier corresponding to the `$storage_type()`.
- `$extension_name()`: Returns the extension name.
- `$extension_metadata()`: Returns the serialized version of the extension metadata as a `raw()` vector.
- `$extension_metadata_utf8()`: Returns the serialized version of the extension metadata as a UTF-8 encoded string.
- `$WrapArray(array)`: Wraps a storage Array into an ExtensionArray with this extension type.

In addition, subclasses may override the following methods to customize the behaviour of extension classes.

- `$deserialize_instance()`: This method is called when a new ExtensionType is initialized and is responsible for parsing and validating the serialized extension_metadata (a `raw()` vector) such that its contents can be inspected by fields and/or methods of the R6 ExtensionType subclass. Implementations must also check the `storage_type` to make sure it is compatible with the extension type.
- `$as_vector(extension_array)`: Convert an Array or ChunkedArray to an R vector. This method is called by `as.vector()` on ExtensionArray objects, when a RecordBatch containing an ExtensionArray is converted to a `data.frame()`, or when a ChunkedArray (e.g., a column in a Table) is converted to an R vector. The default method returns the converted storage array.
- `$ToString()` Return a string representation that will be printed to the console when this type or an Array of this type is printed.

`FeatherReader` *FeatherReader class*

Description

This class enables you to interact with Feather files. Create one to connect to a file or other InputStream, and call `Read()` on it to make an `arrow::Table`. See its usage in `read_feather()`.

Factory

The `FeatherReader$create()` factory method instantiates the object and takes the following argument:

- `file` an Arrow file connection object inheriting from `RandomAccessFile`.

Methods

- `$Read(columns)`: Returns a `Table` of the selected columns, a vector of integer indices
- `$column_names`: Active binding, returns the column names in the Feather file
- `$schema`: Active binding, returns the schema of the Feather file
- `$version`: Active binding, returns `1` or `2`, according to the Feather file version

---

`Field` *Field class*

Description

`field()` lets you create an `arrow::Field` that maps a DataType to a column name. Fields are contained in Schemas.

Methods

- `f$ToString()`: convert to a string
- `f$Equals(other)`: test for equality. More naturally called as `f == other`

`field` *Create a Field*

Description

Create a Field

Usage

```
field(name, type, metadata, nullable = TRUE)
```

Arguments

`name`	field name
`type`	logical type, instance of DataType
`metadata`	currently ignored
`nullable`	TRUE if field is nullable

See Also

Field

Examples

```
field("x", int32())
```

`FileFormat` *Dataset file formats*

Description

A `FileFormat` holds information about how to read and parse the files included in a `Dataset`. There are subclasses corresponding to the supported file formats (`ParquetFileFormat` and `IpcFileFormat`).

Factory

`FileFormat$create()` takes the following arguments:

- `format`: A string identifier of the file format. Currently supported values:
  - "parquet"
  - "ipc"/"arrow"/"feather", all aliases for each other; for Feather, note that only version 2 files are supported
  - "csv"/"text", aliases for the same thing (because comma is the default delimiter for text files
  - "tsv", equivalent to passing `format = "text", delimiter = "\t"`
- `...`: Additional format-specific options

  `format = "parquet"`:
  - `dict_columns`: Names of columns which should be read as dictionaries.
  - Any Parquet options from FragmentScanOptions.

  `format = "text"`: see CsvParseOptions. Note that you can specify them either with the Arrow C++ library naming ("delimiter", "quoting", etc.) or the `readr`-style naming used in `read_csv_arrow()` ("delim", "quote", etc.). Not all `readr` options are currently supported; please file an issue if you encounter one that `arrow` should support. Also, the following options are supported. From CsvReadOptions:
  - `skip_rows`
  - `column_names`. Note that if a Schema is specified, `column_names` must match those specified in the schema.
  - `autogenerate_column_names` From CsvFragmentScanOptions (these values can be overridden at scan time):
  - `convert_options`: a CsvConvertOptions
  - `block_size`

It returns the appropriate subclass of `FileFormat` (e.g. `ParquetFileFormat`)

Examples

```
## Semi-colon delimited files
# Set up directory for examples
tf <- tempfile()
dir.create(tf)
on.exit(unlink(tf))
write.table(mtcars, file.path(tf, "file1.txt"), sep = ";", row.names = FALSE)

# Create FileFormat object
format <- FileFormat$create(format = "text", delimiter = ";")

open_dataset(tf, format = format)
```

FileInfo *FileSystem entry info*

Description

FileSystem entry info

Methods

- `base_name()` : The file base name (component after the last directory separator).
- `extension()` : The file extension

Active bindings

- `$type`: The file type
- `$path`: The full file path in the filesystem
- `$size`: The size in bytes, if available. Only regular files are guaranteed to have a size.
- `$mtime`: The time of last modification, if available.

FileSelector *file selector*

Description

file selector

Factory

The `$create()` factory method instantiates a `FileSelector` given the 3 fields described below.

Fields

- base_dir: The directory in which to select files. If the path exists but doesn't point to a directory, this should be an error.
- allow_not_found: The behavior if base_dir doesn't exist in the filesystem. If FALSE, an error is returned. If TRUE, an empty selection is returned
- recursive: Whether to recurse into subdirectories.

FileSystem *FileSystem classes*

Description

FileSystem is an abstract file system API, LocalFileSystem is an implementation accessing files on the local machine. SubTreeFileSystem is an implementation that delegates to another implementation after prepending a fixed base path

Factory

LocalFileSystem$create() returns the object and takes no arguments.

SubTreeFileSystem$create() takes the following arguments:

- base_path, a string path
- base_fs, a FileSystem object

S3FileSystem$create() optionally takes arguments:

- anonymous: logical, default FALSE. If true, will not attempt to look up credentials using standard AWS configuration methods.
- access_key, secret_key: authentication credentials. If one is provided, the other must be as well. If both are provided, they will override any AWS configuration set at the environment level.
- session_token: optional string for authentication along with access_key and secret_key
- role_arn: string AWS ARN of an AccessRole. If provided instead of access_key and secret_key, temporary credentials will be fetched by assuming this role.
- session_name: optional string identifier for the assumed role session.
- external_id: optional unique string identifier that might be required when you assume a role in another account.
- load_frequency: integer, frequency (in seconds) with which temporary credentials from an assumed role session will be refreshed. Default is 900 (i.e. 15 minutes)
- region: AWS region to connect to. If omitted, the AWS library will provide a sensible default based on client configuration, falling back to "us-east-1" if no other alternatives are found.
- endpoint_override: If non-empty, override region with a connect string such as "localhost:9000". This is useful for connecting to file systems that emulate S3.

- scheme: S3 connection transport (default "https")
- proxy_options: optional string, URI of a proxy to use when connecting to S3
- background_writes: logical, whether OutputStream writes will be issued in the background, without blocking (default TRUE)
- allow_bucket_creation: logical, if TRUE, the filesystem will create buckets if $CreateDir() is called on the bucket level (default FALSE).
- allow_bucket_deletion: logical, if TRUE, the filesystem will delete buckets if$DeleteDir() is called on the bucket level (default FALSE).
- request_timeout: Socket read time on Windows and macOS in seconds. If negative, the AWS SDK default (typically 3 seconds).
- connect_timeout: Socket connection timeout in seconds. If negative, AWS SDK default is used (typically 1 second).

GcsFileSystem$create() optionally takes arguments:

- anonymous: logical, default FALSE. If true, will not attempt to look up credentials using standard GCS configuration methods.
- access_token: optional string for authentication. Should be provided along with expiration
- expiration: POSIXct. optional datetime representing point at which access_token will expire.
- json_credentials: optional string for authentication. Either a string containing JSON credentials or a path to their location on the filesystem. If a path to credentials is given, the file should be UTF-8 encoded.
- endpoint_override: if non-empty, will connect to provided host name / port, such as "localhost:9001", instead of default GCS ones. This is primarily useful for testing purposes.
- scheme: connection transport (default "https")
- default_bucket_location: the default location (or "region") to create new buckets in.
- retry_limit_seconds: the maximum amount of time to spend retrying if the filesystem encounters errors. Default is 15 seconds.
- default_metadata: default metadata to write in new objects.
- project_id: the project to use for creating buckets.

Methods

- path(x): Create a SubTreeFileSystem from the current FileSystem rooted at the specified path x.
- cd(x): Create a SubTreeFileSystem from the current FileSystem rooted at the specified path x.
- ls(path, ...): List files or objects at the given path or from the root of the FileSystem if path is not provided. Additional arguments passed to FileSelector$create, see FileSelector.
- $GetFileInfo(x): x may be a FileSelector or a character vector of paths. Returns a list of FileInfo
- $CreateDir(path, recursive = TRUE): Create a directory and subdirectories.

- `$DeleteDir(path)`: Delete a directory and its contents, recursively.
- `$DeleteDirContents(path)`: Delete a directory's contents, recursively. Like `$DeleteDir()`, but doesn't delete the directory itself. Passing an empty path (`""`) will wipe the entire filesystem tree.
- `$DeleteFile(path)` : Delete a file.
- `$DeleteFiles(paths)` : Delete many files. The default implementation issues individual delete operations in sequence.
- `$Move(src, dest)`: Move / rename a file or directory. If the destination exists: if it is a non-empty directory, an error is returned otherwise, if it has the same type as the source, it is replaced otherwise, behavior is unspecified (implementation-dependent).
- `$CopyFile(src, dest)`: Copy a file. If the destination exists and is a directory, an error is returned. Otherwise, it is replaced.
- `$OpenInputStream(path)`: Open an input stream for sequential reading.
- `$OpenInputFile(path)`: Open an input file for random access reading.
- `$OpenOutputStream(path)`: Open an output stream for sequential writing.
- `$OpenAppendStream(path)`: Open an output stream for appending.

Active bindings

- `$type_name`: string filesystem type name, such as "local", "s3", etc.
- `$region`: string AWS region, for `S3FileSystem` and `SubTreeFileSystem` containing a `S3FileSystem`
- `$base_fs`: for `SubTreeFileSystem`, the `FileSystem` it contains
- `$base_path`: for `SubTreeFileSystem`, the path in `$base_fs` which is considered root in this `SubTreeFileSystem`.
- `$options`: for `GcsFileSystem`, the options used to create the `GcsFileSystem` instance as a `list`

Notes

On S3FileSystem, `$CreateDir()` on a top-level directory creates a new bucket. When S3FileSystem creates new buckets (assuming allow_bucket_creation is TRUE), it does not pass any non-default settings. In AWS S3, the bucket and all objects will be not publicly visible, and will have no bucket policies and no resource tags. To have more control over how buckets are created, use a different API to create them.

On S3FileSystem, output is only produced for fatal errors or when printing return values. For troubleshooting, the log level can be set using the environment variable `ARROW_S3_LOG_LEVEL` (e.g., `Sys.setenv("ARROW_S3_LOG_LEVEL"="DEBUG")`). The log level must be set prior to running any code that interacts with S3. Possible values include 'FATAL' (the default), 'ERROR', 'WARN', 'INFO', 'DEBUG' (recommended), 'TRACE', and 'OFF'.

`FileWriteOptions` *Format-specific write options*

Description

A `FileWriteOptions` holds write options specific to a `FileFormat`.

FixedWidthType *FixedWidthType class*

Description

FixedWidthType class

Methods

TODO

flight_connect *Connect to a Flight server*

Description

Connect to a Flight server

Usage

```
flight_connect(host = "localhost", port, scheme = "grpc+tcp")
```

Arguments

host	string hostname to connect to
port	integer port to connect on
scheme	URL scheme, default is "grpc+tcp"

Value

A `pyarrow.flight.FlightClient`.

flight_disconnect *Explicitly close a Flight client*

Description

Explicitly close a Flight client

Usage

```
flight_disconnect(client)
```

Arguments

client	The client to disconnect

flight_get *Get data from a Flight server*

Description

Get data from a Flight server

Usage

```
flight_get(client, path)
```

Arguments

client	pyarrow.flight.FlightClient, as returned by flight_connect()
path	string identifier under which data is stored

Value

A Table

flight_put *Send data to a Flight server*

Description

Send data to a Flight server

Usage

```
flight_put(client, data, path, overwrite = TRUE, max_chunksize = NULL)
```

Arguments

client	pyarrow.flight.FlightClient, as returned by flight_connect()
data	data.frame, RecordBatch, or Table to upload
path	string identifier to store the data under
overwrite	logical: if path exists on client already, should we replace it with the contents of data? Default is TRUE; if FALSE and path exists, the function will error.
max_chunksize	integer: Maximum number of rows for RecordBatch chunks when a data.frame is sent. Individual chunks may be smaller depending on the chunk layout of individual columns.

Value

client, invisibly.

FragmentScanOptions *Format-specific scan options*

Description

A FragmentScanOptions holds options specific to a FileFormat and a scan operation.

Factory

FragmentScanOptions$create() takes the following arguments:

- format: A string identifier of the file format. Currently supported values:
  - "parquet"
  - "csv"/"text", aliases for the same format.
- ...: Additional format-specific options

  format = "parquet":
  - use_buffered_stream: Read files through buffered input streams rather than loading entire row groups at once. This may be enabled to reduce memory overhead. Disabled by default.
  - buffer_size: Size of buffered stream, if enabled. Default is 8KB.
  - pre_buffer: Pre-buffer the raw Parquet data. This can improve performance on high-latency filesystems. Disabled by default.
  - thrift_string_size_limit: Maximum string size allocated for decoding thrift strings. May need to be increased in order to read files with especially large headers. Default value 100000000.
  - thrift_container_size_limit: Maximum size of thrift containers. May need to be increased in order to read files with especially large headers. Default value 1000000. format = "text": see CsvConvertOptions. Note that options can only be specified with the Arrow C++ library naming. Also, "block_size" from CsvReadOptions may be given.

It returns the appropriate subclass of FragmentScanOptions (e.g. CsvFragmentScanOptions).

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

Description

gs_bucket() is a convenience function to create an GcsFileSystem object that holds onto its relative path

Usage

```
gs_bucket(bucket, ...)
```

Arguments

bucket	string GCS bucket name or path
...	Additional connection options, passed to GcsFileSystem$create()

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")
```

---

hive_partition *Construct Hive partitioning*

Description

Hive partitioning embeds field names and values in path segments, such as "/year=2019/month=2/data.parquet".

Usage

```
hive_partition(..., null_fallback = NULL, segment_encoding = "uri")
```

Arguments

...	named list of data types, passed to schema()
null_fallback	character to be used in place of missing values (NA or NULL) in partition columns. Default is "__HIVE_DEFAULT_PARTITION__", which is what Hive uses.
segment_encoding	Decode partition segments after splitting paths. Default is "uri" (URI-decode segments). May also be "none" (leave as-is).

Details

Because fields are named in the path segments, order of fields passed to hive_partition() does not matter.

Value

A HivePartitioning, or a HivePartitioningFactory if calling hive_partition() with no arguments.

Examples

```
hive_partition(year = int16(), month = int8())
```

infer_schema *Extract a schema from an object*

Description

Extract a schema from an object

Usage

```
infer_schema(x)
```

Arguments

x	An object which has a schema, e.g. a Dataset

infer_type *Infer the arrow Array type from an R object*

Description

type() is deprecated in favor of infer_type().

Usage

```
infer_type(x, ...)

type(x)
```

Arguments

x	an R object (usually a vector) to be converted to an Array or ChunkedArray.
...	Passed to S3 methods

Value

An arrow data type

Examples

```
infer_type(1:10)
infer_type(1L:10L)
infer_type(c(1, 1.5, 2))
infer_type(c("A", "B", "C"))
infer_type(mtcars)
infer_type(Sys.Date())
infer_type(as.POSIXlt(Sys.Date()))
infer_type(vctrs::new_vctr(1:5, class = "my_custom_vctr_class"))
```

InputStream *InputStream classes*

Description

RandomAccessFile inherits from InputStream and is a base class for: ReadableFile for reading from a file; MemoryMappedFile for the same but with memory mapping; and BufferReader for reading from a buffer. Use these with the various table readers.

Factory

The $create() factory methods instantiate the InputStream object and take the following arguments, depending on the subclass:

- path For ReadableFile, a character file name
- x For BufferReader, a Buffer or an object that can be made into a buffer via buffer().

To instantiate a MemoryMappedFile, call mmap_open().

Methods

- $GetSize():
- $supports_zero_copy(): Logical
- $seek(position): go to that position in the stream
- $tell(): return the position in the stream
- $close(): close the stream
- $Read(nbytes): read data from the stream, either a specified nbytes or all, if nbytes is not provided
- $ReadAt(position, nbytes): similar to $seek(position)$Read(nbytes)
- $Resize(size): for a MemoryMappedFile that is writeable

install_arrow *Install or upgrade the Arrow library*

Description

Use this function to install the latest release of arrow, to switch to or from a nightly development version, or on Linux to try reinstalling with all necessary C++ dependencies.

Usage

```
install_arrow(
  nightly = FALSE,
  binary = Sys.getenv("LIBARROW_BINARY", TRUE),
  use_system = Sys.getenv("ARROW_USE_PKG_CONFIG", FALSE),
  minimal = Sys.getenv("LIBARROW_MINIMAL", FALSE),
  verbose = Sys.getenv("ARROW_R_DEV", FALSE),
  repos = getOption("repos"),
  ...
)
```

Arguments

nightly	logical: Should we install a development version of the package, or should we install from CRAN (the default).
binary	On Linux, value to set for the environment variable LIBARROW_BINARY, which governs how C++ binaries are used, if at all. The default value, TRUE, tells the installation script to detect the Linux distribution and version and find an appropriate C++ library. FALSE would tell the script not to retrieve a binary and instead build Arrow C++ from source. Other valid values are strings corresponding to a Linux distribution-version, to override the value that would be detected. See the install guide for further details.
use_system	logical: Should we use pkg-config to look for Arrow system packages? Default is FALSE. If TRUE, source installation may be faster, but there is a risk of version mismatch. This sets the ARROW_USE_PKG_CONFIG environment variable.
minimal	logical: If building from source, should we build without optional dependencies (compression libraries, for example)? Default is FALSE. This sets the LIBARROW_MINIMAL environment variable.
verbose	logical: Print more debugging output when installing? Default is FALSE. This sets the ARROW_R_DEV environment variable.
repos	character vector of base URLs of the repositories to install from (passed to install.packages())
...	Additional arguments passed to install.packages()

Details

Note that, unlike packages like tensorflow, blogdown, and others that require external dependencies, you do not need to run install_arrow() after a successful arrow installation.

See Also

arrow_info() to see if the package was configured with necessary C++ dependencies. install guide for more ways to tune installation on Linux.

install_pyarrow *Install pyarrow for use with reticulate*

Description

pyarrow is the Python package for Apache Arrow. This function helps with installing it for use with reticulate.

Usage

```
install_pyarrow(envname = NULL, nightly = FALSE, ...)
```

Arguments

envname	The name or full path of the Python environment to install into. This can be a virtualenv or conda environment created by reticulate. See reticulate::py_install().
nightly	logical: Should we install a development version of the package? Default is to use the official release version.
...	additional arguments passed to reticulate::py_install().

io_thread_count *Manage the global I/O thread pool in libarrow*

Description

Manage the global I/O thread pool in libarrow

Usage

```
io_thread_count()

set_io_thread_count(num_threads)
```

Arguments

num_threads	integer: New number of threads for thread pool. At least two threads are recommended to support all operations in the arrow package.

JsonFileFormat *JSON dataset file format*

Description

A JsonFileFormat is a FileFormat subclass which holds information about how to read and parse the files included in a JSON Dataset.

Value

A JsonFileFormat object

Factory

JsonFileFormat$create() can take options in the form of lists passed through as parse_options, or read_options parameters.

Available read_options parameters:

- use_threads: Whether to use the global CPU thread pool. Default TRUE. If FALSE, JSON input must end with an empty line.
- block_size: Block size we request from the IO layer; also determines size of chunks when use_threads is TRUE.

Available parse_options parameters:

- newlines_in_values:Logical: are values allowed to contain CR (0x0d or \r) and LF (0x0a or \n) characters? (default FALSE)

See Also

FileFormat

Examples

list_compute_functions *List available Arrow C++ compute functions*

Description

This function lists the names of all available Arrow C++ library compute functions. These can be called by passing to call_function(), or they can be called by name with an arrow_ prefix inside a dplyr verb.

Usage

```
list_compute_functions(pattern = NULL, ...)
```

Arguments

pattern	Optional regular expression to filter the function list
...	Additional parameters passed to grep()

Details

The resulting list describes the capabilities of your arrow build. Some functions, such as string and regular expression functions, require optional build-time C++ dependencies. If your arrow package was not compiled with those features enabled, those functions will not appear in this list.

Some functions take options that need to be passed when calling them (in a list called options). These options require custom handling in C++; many functions already have that handling set up but not all do. If you encounter one that needs special handling for options, please report an issue.

Note that this list does *not* enumerate all of the R bindings for these functions. The package includes Arrow methods for many base R functions that can be called directly on Arrow objects, as well as some tidyverse-flavored versions available inside dplyr verbs.

Value

A character vector of available Arrow C++ function names

See Also

acero for R bindings for Arrow functions

Examples

```
available_funcs <- list_compute_functions()
utf8_funcs <- list_compute_functions(pattern = "^UTF8", ignore.case = TRUE)
```

---

list_flights *See available resources on a Flight server*

Description

See available resources on a Flight server

Usage

```
list_flights(client)

flight_path_exists(client, path)
```

Arguments

client	pyarrow.flight.FlightClient, as returned by flight_connect()
path	string identifier under which data is stored

Value

list_flights() returns a character vector of paths. flight_path_exists() returns a logical value, the equivalent of path %in% list_flights()

---

load_flight_server *Load a Python Flight server*

Description

Load a Python Flight server

Usage

```
load_flight_server(name, path = system.file(package = "arrow"))
```

Arguments

name	string Python module name
path	file system path where the Python module is found. Default is to look in the inst/ directory for included modules.

Examples

```
load_flight_server("demo_flight_server")
```

---

map_batches *Apply a function to a stream of RecordBatches*

Description

As an alternative to calling collect() on a Dataset query, you can use this function to access the stream of RecordBatches in the Dataset. This lets you do more complex operations in R that operate on chunks of data without having to hold the entire Dataset in memory at once. You can include map_batches() in a dplyr pipeline and do additional dplyr methods on the stream of data in Arrow after it.

Usage

```
map_batches(X, FUN, ..., .schema = NULL, .lazy = TRUE, .data.frame = NULL)
```

Arguments

X	A Dataset or arrow_dplyr_query object, as returned by the dplyr methods on Dataset.
FUN	A function or purrr-style lambda expression to apply to each batch. It must return a RecordBatch or something coercible to one via 'as_record_batch()'.
...	Additional arguments passed to FUN
.schema	An optional schema(). If NULL, the schema will be inferred from the first batch.
.lazy	Use TRUE to evaluate FUN lazily as batches are read from the result; use FALSE to evaluate FUN on all batches before returning the reader.
.data.frame	Deprecated argument, ignored

Details

This is experimental and not recommended for production use. It is also single-threaded and runs in R not C++, so it won't be as fast as core Arrow methods.

Value

An arrow_dplyr_query.

---

match_arrow *Value matching for Arrow objects*

Description

base::match() and base::%in% are not generics, so we can't just define Arrow methods for them. These functions expose the analogous functions in the Arrow C++ library.

Usage

```
match_arrow(x, table, ...)

is_in(x, table, ...)
```

Arguments

x	Scalar, Array or ChunkedArray
table	Scalar, Array, ChunkedArray', or R vector lookup table.
...	additional arguments, ignored

Value

match_arrow() returns an int32-type Arrow object of the same length and type as x with the (0-based) indexes into table. is_in() returns a boolean-type Arrow object of the same length and type as x with values indicating per element of x it it is present in table.

Examples

```
# note that the returned value is 0-indexed
cars_tbl <- arrow_table(name = rownames(mtcars), mtcars)
match_arrow(Scalar$create("Mazda RX4 Wag"), cars_tbl$name)

is_in(Array$create("Mazda RX4 Wag"), cars_tbl$name)

# Although there are multiple matches, you are returned the index of the first
# match, as with the base R equivalent
match(4, mtcars$cyl) # 1-indexed
match_arrow(Scalar$create(4), cars_tbl$cyl) # 0-indexed

# If `x` contains multiple values, you are returned the indices of the first
# match for each value.
match(c(4, 6, 8), mtcars$cyl)
match_arrow(Array$create(c(4, 6, 8)), cars_tbl$cyl)

# Return type matches type of `x`
is_in(c(4, 6, 8), mtcars$cyl) # returns vector
is_in(Scalar$create(4), mtcars$cyl) # returns Scalar
is_in(Array$create(c(4, 6, 8)), cars_tbl$cyl) # returns Array
is_in(ChunkedArray$create(c(4, 6), 8), cars_tbl$cyl) # returns ChunkedArray
```

Message *Message class*

Description

Message class

Methods

TODO

MessageReader *MessageReader class*

Description

MessageReader class

Methods

TODO

mmap_create *Create a new read/write memory mapped file of a given size*

Description

Create a new read/write memory mapped file of a given size

Usage

```
mmap_create(path, size)
```

Arguments

path	file path
size	size in bytes

Value

a arrow::io::MemoryMappedFile

mmap_open *Open a memory mapped file*

Description

Open a memory mapped file

Usage

```
mmap_open(path, mode = c("read", "write", "readwrite"))
```

Arguments

path	file path
mode	file mode (read/write/readwrite)

new_extension_type *Extension types*

Description

Extension arrays are wrappers around regular Arrow Array objects that provide some customized behaviour and/or storage. A common use-case for extension types is to define a customized conversion between an an Arrow Array and an R object when the default conversion is slow or loses metadata important to the interpretation of values in the array. For most types, the built-in vctrs extension type is probably sufficient.

Usage

```
new_extension_type(
  storage_type,
  extension_name,
  extension_metadata = raw(),
  type_class = ExtensionType
)

new_extension_array(storage_array, extension_type)

register_extension_type(extension_type)

reregister_extension_type(extension_type)

unregister_extension_type(extension_name)
```

Arguments

storage_type	The data type of the underlying storage array.
extension_name	The extension name. This should be namespaced using "dot" syntax (i.e., "some_package.some_type"). The namespace "arrow" is reserved for extension types defined by the Apache Arrow libraries.
extension_metadata	A raw() or character() vector containing the serialized version of the type. Character vectors must be length 1 and are converted to UTF-8 before converting to raw().
type_class	An R6::R6Class whose $new() class method will be used to construct a new instance of the type.
storage_array	An Array object of the underlying storage.
extension_type	An ExtensionType instance.

Details

These functions create, register, and unregister ExtensionType and ExtensionArray objects. To use an extension type you will have to:

- Define an R6::R6Class that inherits from ExtensionType and reimplement one or more methods (e.g., `deserialize_instance()`).
- Make a type constructor function (e.g., `my_extension_type()`) that calls `new_extension_type()` to create an R6 instance that can be used as a data type elsewhere in the package.
- Make an array constructor function (e.g., `my_extension_array()`) that calls `new_extension_array()` to create an Array instance of your extension type.
- Register a dummy instance of your extension type created using you constructor function using `register_extension_type()`.

If defining an extension type in an R package, you will probably want to use `reregister_extension_type()` in that package's `.onLoad()` hook since your package will probably get reloaded in the same R session during its development and `register_extension_type()` will error if called twice for the same `extension_name`. For an example of an extension type that uses most of these features, see `vctrs_extension_type()`.

Value

- `new_extension_type()` returns an ExtensionType instance according to the `type_class` specified.
- `new_extension_array()` returns an ExtensionArray whose `$type` corresponds to `extension_type`.
- `register_extension_type()`, `unregister_extension_type()` and `reregister_extension_type()` return `NULL`, invisibly.

Examples

```
# Create the R6 type whose methods control how Array objects are
# converted to R objects, how equality between types is computed,
# and how types are printed.
QuantizedType <- R6::R6Class(
  "QuantizedType",
  inherit = ExtensionType,
  public = list(
    # methods to access the custom metadata fields
    center = function() private$.center,
    scale = function() private$.scale,

    # called when an Array of this type is converted to an R vector
    as_vector = function(extension_array) {
      if (inherits(extension_array, "ExtensionArray")) {
        unquantized_array <-
          (extension_array$storage()$cast(float64()) / private$.scale) +
          private$.center

        as.vector(unquantized_array)
      } else {
```

```
        super$as_vector(extension_array)
      }
    },

    # populate the custom metadata fields from the serialized metadata
    deserialize_instance = function() {
      vals <- as.numeric(strsplit(self$extension_metadata_utf8(), ";")[[1]])
      private$.center <- vals[1]
      private$.scale <- vals[2]
    }
  ),
  private = list(
    .center = NULL,
    .scale = NULL
  )
)

# Create a helper type constructor that calls new_extension_type()
quantized <- function(center = 0, scale = 1, storage_type = int32()) {
  new_extension_type(
    storage_type = storage_type,
    extension_name = "arrow.example.quantized",
    extension_metadata = paste(center, scale, sep = ";"),
    type_class = QuantizedType
  )
}

# Create a helper array constructor that calls new_extension_array()
quantized_array <- function(x, center = 0, scale = 1,
                            storage_type = int32()) {
  type <- quantized(center, scale, storage_type)
  new_extension_array(
    Array$create((x - center) * scale, type = storage_type),
    type
  )
}

# Register the extension type so that Arrow knows what to do when
# it encounters this extension type
reregister_extension_type(quantized())

# Create Array objects and use them!
(vals <- runif(5, min = 19, max = 21))

(array <- quantized_array(
  vals,
  center = 20,
  scale = 2^15 - 1,
  storage_type = int16()
)
)

array$type$center()
```

```
array$type$scale()

as.vector(array)
```

`open_dataset` *Open a multi-file dataset*

Description

Arrow Datasets allow you to query against data that has been split across multiple files. This sharding of data may indicate partitioning, which can accelerate queries that only touch some partitions (files). Call `open_dataset()` to point to a directory of data files and return a `Dataset`, then use `dplyr` methods to query it.

Usage

```
open_dataset(
  sources,
  schema = NULL,
  partitioning = hive_partition(),
  hive_style = NA,
  unify_schemas = NULL,
  format = c("parquet", "arrow", "ipc", "feather", "csv", "tsv", "text", "json"),
  factory_options = list(),
  ...
)
```

Arguments

`sources` One of:

- a string path or URI to a directory containing data files
- a FileSystem that references a directory containing data files (such as what is returned by `s3_bucket()`)
- a string path or URI to a single file
- a character vector of paths or URIs to individual data files
- a list of `Dataset` objects as created by this function
- a list of `DatasetFactory` objects as created by `dataset_factory()`.

When `sources` is a vector of file URIs, they must all use the same protocol and point to files located in the same file system and having the same format.

`schema` Schema for the `Dataset`. If `NULL` (the default), the schema will be inferred from the data sources.

`partitioning` When `sources` is a directory path/URI, one of:

- a `Schema`, in which case the file paths relative to `sources` will be parsed, and path segments will be matched with the schema fields.

- a character vector that defines the field names corresponding to those path segments (that is, you're providing the names that would correspond to a `Schema` but the types will be autodetected)
- a `Partitioning` or `PartitioningFactory`, such as returned by `hive_partition()`
- `NULL` for no partitioning

The default is to autodetect Hive-style partitions unless `hive_style = FALSE`. See the "Partitioning" section for details. When `sources` is not a directory path/URI, `partitioning` is ignored.

`hive_style` Logical: should `partitioning` be interpreted as Hive-style? Default is `NA`, which means to inspect the file paths for Hive-style partitioning and behave accordingly.

`unify_schemas` logical: should all data fragments (files, `Datasets`) be scanned in order to create a unified schema from them? If `FALSE`, only the first fragment will be inspected for its schema. Use this fast path when you know and trust that all fragments have an identical schema. The default is `FALSE` when creating a dataset from a directory path/URI or vector of file paths/URIs (because there may be many files and scanning may be slow) but `TRUE` when `sources` is a list of `Datasets` (because there should be few `Datasets` in the list and their `Schemas` are already in memory).

`format` A FileFormat object, or a string identifier of the format of the files in `x`. This argument is ignored when `sources` is a list of `Dataset` objects. Currently supported values:

- "parquet"
- "ipc"/"arrow"/"feather", all aliases for each other; for Feather, note that only version 2 files are supported
- "csv"/"text", aliases for the same thing (because comma is the default delimiter for text files
- "tsv", equivalent to passing `format = "text", delimiter = "\t"`
- "json", for JSON format datasets Note: only newline-delimited JSON (aka ND-JSON) datasets are currently supported Default is "parquet", unless a `delimiter` is also specified, in which case it is assumed to be "text".

`factory_options`

list of optional FileSystemFactoryOptions:

- `partition_base_dir`: string path segment prefix to ignore when discovering partition information with DirectoryPartitioning. Not meaningful (ignored with a warning) for HivePartitioning, nor is it valid when providing a vector of file paths.
- `exclude_invalid_files`: logical: should files that are not valid data files be excluded? Default is `FALSE` because checking all files up front incurs I/O and thus will be slower, especially on remote filesystems. If false and there are invalid files, there will be an error at scan time. This is the only FileSystemFactoryOption that is valid for both when providing a directory path in which to discover files and when providing a vector of file paths.
- `selector_ignore_prefixes`: character vector of file prefixes to ignore when discovering files in a directory. If invalid files can be excluded by a

common filename prefix this way, you can avoid the I/O cost of `exclude_invalid_files`. Not valid when providing a vector of file paths (but if you're providing the file list, you can filter invalid files yourself).

`...` additional arguments passed to `dataset_factory()` when `sources` is a directory path/URI or vector of file paths/URIs, otherwise ignored. These may include `format` to indicate the file format, or other format-specific options (see `read_csv_arrow()`, `read_parquet()` and `read_feather()` on how to specify these).

Value

A Dataset R6 object. Use `dplyr` methods on it to query the data, or call `$NewScan()` to construct a query directly.

Partitioning

Data is often split into multiple files and nested in subdirectories based on the value of one or more columns in the data. It may be a column that is commonly referenced in queries, or it may be time-based, for some examples. Data that is divided this way is "partitioned," and the values for those partitioning columns are encoded into the file path segments. These path segments are effectively virtual columns in the dataset, and because their values are known prior to reading the files themselves, we can greatly speed up filtered queries by skipping some files entirely.

Arrow supports reading partition information from file paths in two forms:

- "Hive-style", deriving from the Apache Hive project and common to some database systems. Partitions are encoded as "key=value" in path segments, such as `"year=2019/month=1/file.parquet"`. While they may be awkward as file names, they have the advantage of being self-describing.
- "Directory" partitioning, which is Hive without the key names, like `"2019/01/file.parquet"`. In order to use these, we need know at least what names to give the virtual columns that come from the path segments.

The default behavior in `open_dataset()` is to inspect the file paths contained in the provided directory, and if they look like Hive-style, parse them as Hive. If your dataset has Hive-style partitioning in the file paths, you do not need to provide anything in the `partitioning` argument to `open_dataset()` to use them. If you do provide a character vector of partition column names, they will be ignored if they match what is detected, and if they don't match, you'll get an error. (If you want to rename partition columns, do that using `select()` or `rename()` after opening the dataset.). If you provide a `Schema` and the names match what is detected, it will use the types defined by the Schema. In the example file path above, you could provide a Schema to specify that "month" should be `int8()` instead of the `int32()` it will be parsed as by default.

If your file paths do not appear to be Hive-style, or if you pass `hive_style = FALSE`, the `partitioning` argument will be used to create Directory partitioning. A character vector of names is required to create partitions; you may instead provide a `Schema` to map those names to desired column types, as described above. If neither are provided, no partitioning information will be taken from the file paths.

See Also

datasets article

Examples

```
# Set up directory for examples
tf <- tempfile()
dir.create(tf)
on.exit(unlink(tf))

write_dataset(mtcars, tf, partitioning = "cyl")

# You can specify a directory containing the files for your dataset and
# open_dataset will scan all files in your directory.
open_dataset(tf)

# You can also supply a vector of paths
open_dataset(c(file.path(tf, "cyl=4/part-0.parquet"), file.path(tf, "cyl=8/part-0.parquet")))

## You must specify the file format if using a format other than parquet.
tf2 <- tempfile()
dir.create(tf2)
on.exit(unlink(tf2))
write_dataset(mtcars, tf2, format = "ipc")
# This line will results in errors when you try to work with the data
## Not run:
open_dataset(tf2)

## End(Not run)
# This line will work
open_dataset(tf2, format = "ipc")

## You can specify file partitioning to include it as a field in your dataset
# Create a temporary directory and write example dataset
tf3 <- tempfile()
dir.create(tf3)
on.exit(unlink(tf3))
write_dataset(airquality, tf3, partitioning = c("Month", "Day"), hive_style = FALSE)

# View files - you can see the partitioning means that files have been written
# to folders based on Month/Day values
tf3_files <- list.files(tf3, recursive = TRUE)

# With no partitioning specified, dataset contains all files but doesn't include
# directory names as field names
open_dataset(tf3)

# Now that partitioning has been specified, your dataset contains columns for Month and Day
open_dataset(tf3, partitioning = c("Month", "Day"))

# If you want to specify the data types for your fields, you can pass in a Schema
open_dataset(tf3, partitioning = schema(Month = int8(), Day = int8()))
```

`open_delim_dataset` *Open a multi-file dataset of CSV or other delimiter-separated format*

Description

A wrapper around open_dataset which explicitly includes parameters mirroring `read_csv_arrow()`, `read_delim_arrow()`, and `read_tsv_arrow()` to allow for easy switching between functions for opening single files and functions for opening datasets.

Usage

```
open_delim_dataset(
  sources,
  schema = NULL,
  partitioning = hive_partition(),
  hive_style = NA,
  unify_schemas = NULL,
  factory_options = list(),
  delim = ",",
  quote = "\"",
  escape_double = TRUE,
  escape_backslash = FALSE,
  col_names = TRUE,
  col_types = NULL,
  na = c("", "NA"),
  skip_empty_rows = TRUE,
  skip = 0L,
  convert_options = NULL,
  read_options = NULL,
  timestamp_parsers = NULL,
  quoted_na = TRUE,
  parse_options = NULL
)

open_csv_dataset(
  sources,
  schema = NULL,
  partitioning = hive_partition(),
  hive_style = NA,
  unify_schemas = NULL,
  factory_options = list(),
  quote = "\"",
  escape_double = TRUE,
  escape_backslash = FALSE,
  col_names = TRUE,
  col_types = NULL,
  na = c("", "NA"),
```

```
  skip_empty_rows = TRUE,
  skip = 0L,
  convert_options = NULL,
  read_options = NULL,
  timestamp_parsers = NULL,
  quoted_na = TRUE,
  parse_options = NULL
)

open_tsv_dataset(
  sources,
  schema = NULL,
  partitioning = hive_partition(),
  hive_style = NA,
  unify_schemas = NULL,
  factory_options = list(),
  quote = "\"",
  escape_double = TRUE,
  escape_backslash = FALSE,
  col_names = TRUE,
  col_types = NULL,
  na = c("", "NA"),
  skip_empty_rows = TRUE,
  skip = 0L,
  convert_options = NULL,
  read_options = NULL,
  timestamp_parsers = NULL,
  quoted_na = TRUE,
  parse_options = NULL
)
```

Arguments

sources	One of: • a string path or URI to a directory containing data files • a FileSystem that references a directory containing data files (such as what is returned by s3_bucket()) • a string path or URI to a single file • a character vector of paths or URIs to individual data files • a list of Dataset objects as created by this function • a list of DatasetFactory objects as created by dataset_factory(). When sources is a vector of file URIs, they must all use the same protocol and point to files located in the same file system and having the same format.
schema	Schema for the Dataset. If NULL (the default), the schema will be inferred from the data sources.
partitioning	When sources is a directory path/URI, one of:

- a Schema, in which case the file paths relative to sources will be parsed, and path segments will be matched with the schema fields.
- a character vector that defines the field names corresponding to those path segments (that is, you're providing the names that would correspond to a Schema but the types will be autodetected)
- a Partitioning or PartitioningFactory, such as returned by hive_partition()
- NULL for no partitioning

The default is to autodetect Hive-style partitions unless hive_style = FALSE. See the "Partitioning" section for details. When sources is not a directory path/URI, partitioning is ignored.

hive_style — Logical: should partitioning be interpreted as Hive-style? Default is NA, which means to inspect the file paths for Hive-style partitioning and behave accordingly.

unify_schemas — logical: should all data fragments (files, Datasets) be scanned in order to create a unified schema from them? If FALSE, only the first fragment will be inspected for its schema. Use this fast path when you know and trust that all fragments have an identical schema. The default is FALSE when creating a dataset from a directory path/URI or vector of file paths/URIs (because there may be many files and scanning may be slow) but TRUE when sources is a list of Datasets (because there should be few Datasets in the list and their Schemas are already in memory).

factory_options

list of optional FileSystemFactoryOptions:

- partition_base_dir: string path segment prefix to ignore when discovering partition information with DirectoryPartitioning. Not meaningful (ignored with a warning) for HivePartitioning, nor is it valid when providing a vector of file paths.
- exclude_invalid_files: logical: should files that are not valid data files be excluded? Default is FALSE because checking all files up front incurs I/O and thus will be slower, especially on remote filesystems. If false and there are invalid files, there will be an error at scan time. This is the only FileSystemFactoryOption that is valid for both when providing a directory path in which to discover files and when providing a vector of file paths.
- selector_ignore_prefixes: character vector of file prefixes to ignore when discovering files in a directory. If invalid files can be excluded by a common filename prefix this way, you can avoid the I/O cost of exclude_invalid_files. Not valid when providing a vector of file paths (but if you're providing the file list, you can filter invalid files yourself).

delim — Single character used to separate fields within a record.

quote — Single character used to quote strings.

escape_double — Does the file escape quotes by doubling them? i.e. If this option is TRUE, the value """" represents a single quote, \".

escape_backslash

Does the file use backslashes to escape special characters? This is more general than escape_double as backslashes can be used to escape the delimiter character, the quote character, or to add special characters like \\n.

col_names — If TRUE, the first row of the input will be used as the column names and will not be included in the data frame. If FALSE, column names will be generated by Arrow, starting with "f0", "f1", ..., "fN". Alternatively, you can specify a character vector of column names.

col_types — A compact string representation of the column types, an Arrow Schema, or NULL (the default) to infer types from the data.

na — A character vector of strings to interpret as missing values.

skip_empty_rows — Should blank rows be ignored altogether? If TRUE, blank rows will not be represented at all. If FALSE, they will be filled with missings.

skip — Number of lines to skip before reading data.

convert_options — see CSV conversion options

read_options — see CSV reading options

timestamp_parsers — User-defined timestamp parsers. If more than one parser is specified, the CSV conversion logic will try parsing values starting from the beginning of this vector. Possible values are:

- NULL: the default, which uses the ISO-8601 parser
- a character vector of strptime parse strings
- a list of TimestampParser objects

quoted_na — Should missing values inside quotes be treated as missing values (the default) or strings. (Note that this is different from the the Arrow C++ default for the corresponding convert option, strings_can_be_null.)

parse_options — see CSV parsing options. If given, this overrides any parsing options provided in other arguments (e.g. delim, quote, etc.).

Options currently supported by read_delim_arrow() which are not supported here

- file (instead, please specify files in sources)
- col_select (instead, subset columns after dataset creation)
- as_data_frame (instead, convert to data frame after dataset creation)
- parse_options

See Also

open_dataset()

Examples

```
# Set up directory for examples
tf <- tempfile()
dir.create(tf)
df <- data.frame(x = c("1", "2", "NULL"))

file_path <- file.path(tf, "file1.txt")
```

```
write.table(df, file_path, sep = ",", row.names = FALSE)

read_csv_arrow(file_path, na = c("", "NA", "NULL"), col_names = "y", skip = 1)
open_csv_dataset(file_path, na = c("", "NA", "NULL"), col_names = "y", skip = 1)

unlink(tf)
```

`OutputStream` *OutputStream classes*

Description

`FileOutputStream` is for writing to a file; `BufferOutputStream` writes to a buffer; You can create one and pass it to any of the table writers, for example.

Factory

The `$create()` factory methods instantiate the `OutputStream` object and take the following arguments, depending on the subclass:

- `path` For `FileOutputStream`, a character file name
- `initial_capacity` For `BufferOutputStream`, the size in bytes of the buffer.

Methods

- `$tell()`: return the position in the stream
- `$close()`: close the stream
- `$write(x)`: send `x` to the stream
- `$capacity()`: for `BufferOutputStream`
- `$finish()`: for `BufferOutputStream`
- `$GetExtentBytesWritten()`: for `MockOutputStream`, report how many bytes were sent.

`ParquetArrowReaderProperties`
ParquetArrowReaderProperties class

Description

This class holds settings to control how a Parquet file is read by ParquetFileReader.

Factory

The `ParquetArrowReaderProperties$create()` factory method instantiates the object and takes the following arguments:

- `use_threads` Logical: whether to use multithreading (default `TRUE`)

Methods

- `$read_dictionary(column_index)`
- `$set_read_dictionary(column_index, read_dict)`
- `$use_threads(use_threads)`

`ParquetFileReader` *ParquetFileReader class*

Description

This class enables you to interact with Parquet files.

Factory

The `ParquetFileReader$create()` factory method instantiates the object and takes the following arguments:

- `file` A character file name, raw vector, or Arrow file connection object (e.g. `RandomAccessFile`).
- `props` Optional ParquetArrowReaderProperties
- `mmap` Logical: whether to memory-map the file (default `TRUE`)
- `reader_props` Optional ParquetReaderProperties
- `...` Additional arguments, currently ignored

Methods

- `$ReadTable(column_indices)`: get an `arrow::Table` from the file. The optional `column_indices=` argument is a 0-based integer vector indicating which columns to retain.
- `$ReadRowGroup(i, column_indices)`: get an `arrow::Table` by reading the `i`th row group (0-based). The optional `column_indices=` argument is a 0-based integer vector indicating which columns to retain.
- `$ReadRowGroups(row_groups, column_indices)`: get an `arrow::Table` by reading several row groups (0-based integers). The optional `column_indices=` argument is a 0-based integer vector indicating which columns to retain.
- `$GetSchema()`: get the `arrow::Schema` of the data in the file
- `$ReadColumn(i)`: read the `i`th column (0-based) as a ChunkedArray.

Active bindings

- `$num_rows`: number of rows.
- `$num_columns`: number of columns.
- `$num_row_groups`: number of row groups.

Examples

```
f <- system.file("v0.7.1.parquet", package = "arrow")
pq <- ParquetFileReader$create(f)
pq$GetSchema()
if (codec_is_available("snappy")) {
  # This file has compressed data columns
  tab <- pq$ReadTable()
  tab$schema
}
```

`ParquetFileWriter` *ParquetFileWriter class*

Description

This class enables you to interact with Parquet files.

Factory

The `ParquetFileWriter$create()` factory method instantiates the object and takes the following arguments:

- `schema` A Schema
- `sink` An arrow::io::OutputStream
- `properties` An instance of ParquetWriterProperties
- `arrow_properties` An instance of `ParquetArrowWriterProperties`

Methods

- `WriteTable` Write a Table to `sink`
- `WriteBatch` Write a RecordBatch to `sink`
- `Close` Close the writer. Note: does not close the `sink`. arrow::io::OutputStream has its own `close()` method.

`ParquetReaderProperties`

ParquetReaderProperties class

Description

This class holds settings to control how a Parquet file is read by ParquetFileReader.

Factory

The `ParquetReaderProperties$create()` factory method instantiates the object and takes no arguments.

Methods

- `$thrift_string_size_limit()`
- `$set_thrift_string_size_limit()`
- `$thrift_container_size_limit()`
- `$set_thrift_container_size_limit()`

`ParquetWriterProperties`

ParquetWriterProperties class

Description

This class holds settings to control how a Parquet file is read by ParquetFileWriter.

Details

The parameters `compression`, `compression_level`, `use_dictionary` and write_statistics‘ support various patterns:

- The default `NULL` leaves the parameter unspecified, and the C++ library uses an appropriate default for each column (defaults listed above)
- A single, unnamed, value (e.g. a single string for `compression`) applies to all columns
- An unnamed vector, of the same size as the number of columns, to specify a value for each column, in positional order
- A named vector, to specify the value for the named columns, the default value for the setting is used when not supplied

Unlike the high-level write_parquet, `ParquetWriterProperties` arguments use the C++ defaults. Currently this means "uncompressed" rather than "snappy" for the `compression` argument.

Factory

The ParquetWriterProperties$create() factory method instantiates the object and takes the following arguments:

- table: table to write (required)
- version: Parquet version, "1.0" or "2.0". Default "1.0"
- compression: Compression type, algorithm "uncompressed"
- compression_level: Compression level; meaning depends on compression algorithm
- use_dictionary: Specify if we should use dictionary encoding. Default TRUE
- write_statistics: Specify if we should write statistics. Default TRUE
- data_page_size: Set a target threshold for the approximate encoded size of data pages within a column chunk (in bytes). Default 1 MiB.

See Also

write_parquet

Schema for information about schemas and metadata handling.

Partitioning *Define Partitioning for a Dataset*

Description

Pass a Partitioning object to a FileSystemDatasetFactory's $create() method to indicate how the file's paths should be interpreted to define partitioning.

DirectoryPartitioning describes how to interpret raw path segments, in order. For example, schema(year = int16(), month = int8()) would define partitions for file paths like "2019/01/file.parquet", "2019/02/file.parquet", etc. In this scheme NULL values will be skipped. In the previous example: when writing a dataset if the month was NA (or NULL), the files would be placed in "2019/file.parquet". When reading, the rows in "2019/file.parquet" would return an NA for the month column. An error will be raised if an outer directory is NULL and an inner directory is not.

HivePartitioning is for Hive-style partitioning, which embeds field names and values in path segments, such as "/year=2019/month=2/data.parquet". Because fields are named in the path segments, order does not matter. This partitioning scheme allows NULL values. They will be replaced by a configurable null_fallback which defaults to the string "__HIVE_DEFAULT_PARTITION__" when writing. When reading, the null_fallback string will be replaced with NAs as appropriate.

PartitioningFactory subclasses instruct the DatasetFactory to detect partition features from the file paths.

Factory

Both `DirectoryPartitioning$create()` and `HivePartitioning$create()` methods take a Schema as a single input argument. The helper function `hive_partition(...)` is shorthand for `HivePartitioning$create(schema`

With `DirectoryPartitioningFactory$create()`, you can provide just the names of the path segments (in our example, `c("year", "month")`), and the `DatasetFactory` will infer the data types for those partition variables. `HivePartitioningFactory$create()` takes no arguments: both variable names and their types can be inferred from the file paths. `hive_partition()` with no arguments returns a `HivePartitioningFactory`.

`read_delim_arrow` *Read a CSV or other delimited file with Arrow*

Description

These functions uses the Arrow C++ CSV reader to read into a `tibble`. Arrow C++ options have been mapped to argument names that follow those of `readr::read_delim()`, and `col_select` was inspired by `vroom::vroom()`.

Usage

```
read_delim_arrow(
  file,
  delim = ",",
  quote = "\"",
  escape_double = TRUE,
  escape_backslash = FALSE,
  schema = NULL,
  col_names = TRUE,
  col_types = NULL,
  col_select = NULL,
  na = c("", "NA"),
  quoted_na = TRUE,
  skip_empty_rows = TRUE,
  skip = 0L,
  parse_options = NULL,
  convert_options = NULL,
  read_options = NULL,
  as_data_frame = TRUE,
  timestamp_parsers = NULL,
  decimal_point = "."
)

read_csv_arrow(
  file,
  quote = "\"",
  escape_double = TRUE,
```

```
  escape_backslash = FALSE,
  schema = NULL,
  col_names = TRUE,
  col_types = NULL,
  col_select = NULL,
  na = c("", "NA"),
  quoted_na = TRUE,
  skip_empty_rows = TRUE,
  skip = 0L,
  parse_options = NULL,
  convert_options = NULL,
  read_options = NULL,
  as_data_frame = TRUE,
  timestamp_parsers = NULL
)

read_csv2_arrow(
  file,
  quote = "\"",
  escape_double = TRUE,
  escape_backslash = FALSE,
  schema = NULL,
  col_names = TRUE,
  col_types = NULL,
  col_select = NULL,
  na = c("", "NA"),
  quoted_na = TRUE,
  skip_empty_rows = TRUE,
  skip = 0L,
  parse_options = NULL,
  convert_options = NULL,
  read_options = NULL,
  as_data_frame = TRUE,
  timestamp_parsers = NULL
)

read_tsv_arrow(
  file,
  quote = "\"",
  escape_double = TRUE,
  escape_backslash = FALSE,
  schema = NULL,
  col_names = TRUE,
  col_types = NULL,
  col_select = NULL,
  na = c("", "NA"),
  quoted_na = TRUE,
  skip_empty_rows = TRUE,
```

```
  skip = 0L,
  parse_options = NULL,
  convert_options = NULL,
  read_options = NULL,
  as_data_frame = TRUE,
  timestamp_parsers = NULL
)
```

Arguments

file	A character file name or URI, connection, literal data (either a single string or a raw vector), an Arrow input stream, or a FileSystem with path (SubTreeFileSystem). If a file name, a memory-mapped Arrow InputStream will be opened and closed when finished; compression will be detected from the file extension and handled automatically. If an input stream is provided, it will be left open. To be recognised as literal data, the input must be wrapped with I().
delim	Single character used to separate fields within a record.
quote	Single character used to quote strings.
escape_double	Does the file escape quotes by doubling them? i.e. If this option is TRUE, the value """" represents a single quote, \".
escape_backslash	Does the file use backslashes to escape special characters? This is more general than escape_double as backslashes can be used to escape the delimiter character, the quote character, or to add special characters like \\n.
schema	Schema that describes the table. If provided, it will be used to satisfy both col_names and col_types.
col_names	If TRUE, the first row of the input will be used as the column names and will not be included in the data frame. If FALSE, column names will be generated by Arrow, starting with "f0", "f1", ..., "fN". Alternatively, you can specify a character vector of column names.
col_types	A compact string representation of the column types, an Arrow Schema, or NULL (the default) to infer types from the data.
col_select	A character vector of column names to keep, as in the "select" argument to data.table::fread(), or a tidy selection specification of columns, as used in dplyr::select().
na	A character vector of strings to interpret as missing values.
quoted_na	Should missing values inside quotes be treated as missing values (the default) or strings. (Note that this is different from the the Arrow C++ default for the corresponding convert option, strings_can_be_null.)
skip_empty_rows	Should blank rows be ignored altogether? If TRUE, blank rows will not be represented at all. If FALSE, they will be filled with missings.
skip	Number of lines to skip before reading data.
parse_options	see CSV parsing options. If given, this overrides any parsing options provided in other arguments (e.g. delim, quote, etc.).

convert_options
: see CSV conversion options

read_options
: see CSV reading options

as_data_frame
: Should the function return a tibble (default) or an Arrow Table?

timestamp_parsers
: User-defined timestamp parsers. If more than one parser is specified, the CSV conversion logic will try parsing values starting from the beginning of this vector. Possible values are:

  - NULL: the default, which uses the ISO-8601 parser
  - a character vector of strptime parse strings
  - a list of TimestampParser objects

decimal_point
: Character to use for decimal point in floating point numbers.

Details

read_csv_arrow() and read_tsv_arrow() are wrappers around read_delim_arrow() that specify a delimiter. read_csv2_arrow() uses ; for the delimiter and , for the decimal point.

Note that not all readr options are currently implemented here. Please file an issue if you encounter one that arrow should support.

If you need to control Arrow-specific reader parameters that don't have an equivalent in readr::read_csv(), you can either provide them in the parse_options, convert_options, or read_options arguments, or you can use CsvTableReader directly for lower-level access.

Value

A tibble, or a Table if as_data_frame = FALSE.

Specifying column types and names

By default, the CSV reader will infer the column names and data types from the file, but there are a few ways you can specify them directly.

One way is to provide an Arrow Schema in the schema argument, which is an ordered map of column name to type. When provided, it satisfies both the col_names and col_types arguments. This is good if you know all of this information up front.

You can also pass a Schema to the col_types argument. If you do this, column names will still be inferred from the file unless you also specify col_names. In either case, the column names in the Schema must match the data's column names, whether they are explicitly provided or inferred. That said, this Schema does not have to reference all columns: those omitted will have their types inferred.

Alternatively, you can declare column types by providing the compact string representation that readr uses to the col_types argument. This means you provide a single string, one character per column, where the characters map to Arrow types analogously to the readr type mapping:

- "c": utf8()
- "i": int32()
- "n": float64()

- "d": `float64()`
- "l": `bool()`
- "f": `dictionary()`
- "D": `date32()`
- "T": `timestamp(unit = "ns")`
- "t": `time32()` (The `unit` arg is set to the default value `"ms"`)
- "_": `null()`
- "-": `null()`
- "?": infer the type from the data

If you use the compact string representation for `col_types`, you must also specify `col_names`.

Regardless of how types are specified, all columns with a `null()` type will be dropped.

Note that if you are specifying column names, whether by `schema` or `col_names`, and the CSV file has a header row that would otherwise be used to identify column names, you'll need to add `skip = 1` to skip that row.

Examples

```
tf <- tempfile()
on.exit(unlink(tf))
write.csv(mtcars, file = tf)
df <- read_csv_arrow(tf)
dim(df)
# Can select columns
df <- read_csv_arrow(tf, col_select = starts_with("d"))

# Specifying column types and names
write.csv(data.frame(x = c(1, 3), y = c(2, 4)), file = tf, row.names = FALSE)
read_csv_arrow(tf, schema = schema(x = int32(), y = utf8()), skip = 1)
read_csv_arrow(tf, col_types = schema(y = utf8()))
read_csv_arrow(tf, col_types = "ic", col_names = c("x", "y"), skip = 1)

# Note that if a timestamp column contains time zones,
# the string "T" `col_types` specification won't work.
# To parse timestamps with time zones, provide a [Schema] to `col_types`
# and specify the time zone in the type object:
tf <- tempfile()
write.csv(data.frame(x = "1970-01-01T12:00:00+12:00"), file = tf, row.names = FALSE)
read_csv_arrow(
  tf,
  col_types = schema(x = timestamp(unit = "us", timezone = "UTC"))
)

# Read directly from strings with `I()`
read_csv_arrow(I("x,y\n1,2\n3,4"))
read_delim_arrow(I(c("x y", "1 2", "3 4")), delim = " ")
```

read_feather *Read a Feather file (an Arrow IPC file)*

Description

Feather provides binary columnar serialization for data frames. It is designed to make reading and writing data frames efficient, and to make sharing data across data analysis languages easy. `read_feather()` can read both the Feather Version 1 (V1), a legacy version available starting in 2016, and the Version 2 (V2), which is the Apache Arrow IPC file format. `read_ipc_file()` is an alias of `read_feather()`.

Usage

```
read_feather(file, col_select = NULL, as_data_frame = TRUE, mmap = TRUE)

read_ipc_file(file, col_select = NULL, as_data_frame = TRUE, mmap = TRUE)
```

Arguments

`file`	A character file name or URI, connection, `raw` vector, an Arrow input stream, or a `FileSystem` with path (`SubTreeFileSystem`). If a file name or URI, an Arrow InputStream will be opened and closed when finished. If an input stream is provided, it will be left open.
`col_select`	A character vector of column names to keep, as in the "select" argument to `data.table::fread()`, or a tidy selection specification of columns, as used in `dplyr::select()`.
`as_data_frame`	Should the function return a `tibble` (default) or an Arrow Table?
`mmap`	Logical: whether to memory-map the file (default `TRUE`)

Value

A `tibble` if `as_data_frame` is `TRUE` (the default), or an Arrow Table otherwise

See Also

FeatherReader and RecordBatchReader for lower-level access to reading Arrow IPC data.

Examples

```
# We recommend the ".arrow" extension for Arrow IPC files (Feather V2).
tf <- tempfile(fileext = ".arrow")
on.exit(unlink(tf))
write_feather(mtcars, tf)
df <- read_feather(tf)
dim(df)
# Can select columns
df <- read_feather(tf, col_select = starts_with("d"))
```

read_ipc_stream *Read Arrow IPC stream format*

Description

Apache Arrow defines two formats for serializing data for interprocess communication (IPC): a "stream" format and a "file" format, known as Feather. `read_ipc_stream()` and `read_feather()` read those formats, respectively.

Usage

```
read_ipc_stream(file, as_data_frame = TRUE, ...)
```

Arguments

`file`	A character file name or URI, connection, `raw` vector, an Arrow input stream, or a `FileSystem` with path (`SubTreeFileSystem`). If a file name or URI, an Arrow InputStream will be opened and closed when finished. If an input stream is provided, it will be left open.
`as_data_frame`	Should the function return a `tibble` (default) or an Arrow Table?
`...`	extra parameters passed to `read_feather()`.

Value

A `tibble` if `as_data_frame` is `TRUE` (the default), or an Arrow Table otherwise

See Also

`write_feather()` for writing IPC files. RecordBatchReader for a lower-level interface.

read_json_arrow *Read a JSON file*

Description

Wrapper around JsonTableReader to read a newline-delimited JSON (ndjson) file into a data frame or Arrow Table.

Usage

```
read_json_arrow(
  file,
  col_select = NULL,
  as_data_frame = TRUE,
  schema = NULL,
  ...
)
```

Arguments

file	A character file name or URI, connection, literal data (either a single string or a raw vector), an Arrow input stream, or a FileSystem with path (SubTreeFileSystem). If a file name, a memory-mapped Arrow InputStream will be opened and closed when finished; compression will be detected from the file extension and handled automatically. If an input stream is provided, it will be left open. To be recognised as literal data, the input must be wrapped with I().
col_select	A character vector of column names to keep, as in the "select" argument to data.table::fread(), or a tidy selection specification of columns, as used in dplyr::select().
as_data_frame	Should the function return a tibble (default) or an Arrow Table?
schema	Schema that describes the table.
...	Additional options passed to JsonTableReader$create()

Details

If passed a path, will detect and handle compression from the file extension (e.g. .json.gz).

If schema is not provided, Arrow data types are inferred from the data:

- JSON null values convert to the null() type, but can fall back to any other type.
- JSON booleans convert to boolean().
- JSON numbers convert to int64(), falling back to float64() if a non-integer is encountered.
- JSON strings of the kind "YYYY-MM-DD" and "YYYY-MM-DD hh:mm:ss" convert to timestamp(unit = "s"), falling back to utf8() if a conversion error occurs.
- JSON arrays convert to a list_of() type, and inference proceeds recursively on the JSON arrays' values.
- Nested JSON objects convert to a struct() type, and inference proceeds recursively on the JSON objects' values.

When as_data_frame = TRUE, Arrow types are further converted to R types.

Value

A tibble, or a Table if as_data_frame = FALSE.

Examples

```
tf <- tempfile()
on.exit(unlink(tf))
writeLines('
    { "hello": 3.5, "world": false, "yo": "thing" }
    { "hello": 3.25, "world": null }
    { "hello": 0.0, "world": true, "yo": null }
  ', tf, useBytes = TRUE)

read_json_arrow(tf)
```

```
# Read directly from strings with `I()`
read_json_arrow(I(c('{"x": 1, "y": 2}', '{"x": 3, "y": 4}')))
```

`read_message` *Read a Message from a stream*

Description

Read a Message from a stream

Usage

```
read_message(stream)
```

Arguments

`stream`	an InputStream

`read_parquet` *Read a Parquet file*

Description

'Parquet' is a columnar storage file format. This function enables you to read Parquet files into R.

Usage

```
read_parquet(
  file,
  col_select = NULL,
  as_data_frame = TRUE,
  props = ParquetArrowReaderProperties$create(),
  mmap = TRUE,
  ...
)
```

Arguments

`file`	A character file name or URI, connection, `raw` vector, an Arrow input stream, or a `FileSystem` with path (`SubTreeFileSystem`). If a file name or URI, an Arrow InputStream will be opened and closed when finished. If an input stream is provided, it will be left open.
`col_select`	A character vector of column names to keep, as in the "select" argument to `data.table::fread()`, or a tidy selection specification of columns, as used in `dplyr::select()`.

as_data_frame	Should the function return a tibble (default) or an Arrow Table?
props	ParquetArrowReaderProperties
mmap	Use TRUE to use memory mapping where possible
...	Additional arguments passed to ParquetFileReader$create()

Value

A tibble if as_data_frame is TRUE (the default), or an Arrow Table otherwise.

Examples

```
tf <- tempfile()
on.exit(unlink(tf))
write_parquet(mtcars, tf)
df <- read_parquet(tf, col_select = starts_with("d"))
head(df)
```

---

read_schema *Read a Schema from a stream*

Description

Read a Schema from a stream

Usage

```
read_schema(stream, ...)
```

Arguments

stream	a Message, InputStream, or Buffer
...	currently ignored

Value

A Schema

RecordBatch *RecordBatch class*

Description

A record batch is a collection of equal-length arrays matching a particular Schema. It is a table-like data structure that is semantically a sequence of fields, each a contiguous Arrow Array.

S3 Methods and Usage

Record batches are data-frame-like, and many methods you expect to work on a `data.frame` are implemented for `RecordBatch`. This includes `[`, `[[`, `$`, `names`, `dim`, `nrow`, `ncol`, `head`, and `tail`. You can also pull the data from an Arrow record batch into R with `as.data.frame()`. See the examples.

A caveat about the `$` method: because `RecordBatch` is an `R6` object, `$` is also used to access the object's methods (see below). Methods take precedence over the table's columns. So, `batch$Slice` would return the "Slice" method function even if there were a column in the table called "Slice".

R6 Methods

In addition to the more R-friendly S3 methods, a `RecordBatch` object has the following R6 methods that map onto the underlying C++ methods:

- `$Equals(other)`: Returns `TRUE` if the `other` record batch is equal
- `$column(i)`: Extract an `Array` by integer position from the batch
- `$column_name(i)`: Get a column's name by integer position
- `$names()`: Get all column names (called by `names(batch)`)
- `$nbytes()`: Total number of bytes consumed by the elements of the record batch
- `$RenameColumns(value)`: Set all column names (called by `names(batch) <- value`)
- `$GetColumnByName(name)`: Extract an `Array` by string name
- `$RemoveColumn(i)`: Drops a column from the batch by integer position
- `$SelectColumns(indices)`: Return a new record batch with a selection of columns, expressed as 0-based integers.
- `$Slice(offset, length = NULL)`: Create a zero-copy view starting at the indicated integer offset and going for the given length, or to the end of the table if `NULL`, the default.
- `$Take(i)`: return an `RecordBatch` with rows at positions given by integers (R vector or Array Array) `i`.
- `$Filter(i, keep_na = TRUE)`: return an `RecordBatch` with rows at positions where logical vector (or Arrow boolean Array) `i` is `TRUE`.
- `$SortIndices(names, descending = FALSE)`: return an `Array` of integer row positions that can be used to rearrange the `RecordBatch` in ascending or descending order by the first named column, breaking ties with further named columns. `descending` can be a logical vector of length one or of the same length as `names`.
- `$serialize()`: Returns a raw vector suitable for interprocess communication

- `$cast(target_schema, safe = TRUE, options = cast_options(safe))`: Alter the schema of the record batch.

There are also some active bindings

- `$num_columns`
- `$num_rows`
- `$schema`
- `$metadata`: Returns the key-value metadata of the `Schema` as a named list. Modify or replace by assigning in (`batch$metadata <- new_metadata`). All list elements are coerced to string. See `schema()` for more information.
- `$columns`: Returns a list of `Array`s

`RecordBatchReader` *RecordBatchReader classes*

Description

Apache Arrow defines two formats for serializing data for interprocess communication (IPC): a "stream" format and a "file" format, known as Feather. `RecordBatchStreamReader` and `RecordBatchFileReader` are interfaces for accessing record batches from input sources in those formats, respectively.

For guidance on how to use these classes, see the examples section.

Factory

The `RecordBatchFileReader$create()` and `RecordBatchStreamReader$create()` factory methods instantiate the object and take a single argument, named according to the class:

- `file` A character file name, raw vector, or Arrow file connection object (e.g. RandomAccessFile).
- `stream` A raw vector, Buffer, or InputStream.

Methods

- `$read_next_batch()`: Returns a `RecordBatch`, iterating through the Reader. If there are no further batches in the Reader, it returns `NULL`.
- `$schema`: Returns a Schema (active binding)
- `$batches()`: Returns a list of `RecordBatch`es
- `$read_table()`: Collects the reader's `RecordBatch`es into a Table
- `$get_batch(i)`: For `RecordBatchFileReader`, return a particular batch by an integer index.
- `$num_record_batches()`: For `RecordBatchFileReader`, see how many batches are in the file.

See Also

`read_ipc_stream()` and `read_feather()` provide a much simpler interface for reading data from these formats and are sufficient for many use cases.

Examples

```
tf <- tempfile()
on.exit(unlink(tf))

batch <- record_batch(chickwts)

# This opens a connection to the file in Arrow
file_obj <- FileOutputStream$create(tf)
# Pass that to a RecordBatchWriter to write data conforming to a schema
writer <- RecordBatchFileWriter$create(file_obj, batch$schema)
writer$write(batch)
# You may write additional batches to the stream, provided that they have
# the same schema.
# Call "close" on the writer to indicate end-of-file/stream
writer$close()
# Then, close the connection--closing the IPC message does not close the file
file_obj$close()

# Now, we have a file we can read from. Same pattern: open file connection,
# then pass it to a RecordBatchReader
read_file_obj <- ReadableFile$create(tf)
reader <- RecordBatchFileReader$create(read_file_obj)
# RecordBatchFileReader knows how many batches it has (StreamReader does not)
reader$num_record_batches
# We could consume the Reader by calling $read_next_batch() until all are,
# consumed, or we can call $read_table() to pull them all into a Table
tab <- reader$read_table()
# Call as.data.frame to turn that Table into an R data.frame
df <- as.data.frame(tab)
# This should be the same data we sent
all.equal(df, chickwts, check.attributes = FALSE)
# Unlike the Writers, we don't have to close RecordBatchReaders,
# but we do still need to close the file connection
read_file_obj$close()
```

`RecordBatchWriter` *RecordBatchWriter classes*

Description

Apache Arrow defines two formats for serializing data for interprocess communication (IPC): a "stream" format and a "file" format, known as Feather. `RecordBatchStreamWriter` and `RecordBatchFileWriter` are interfaces for writing record batches to those formats, respectively.

For guidance on how to use these classes, see the examples section.

Factory

The `RecordBatchFileWriter$create()` and `RecordBatchStreamWriter$create()` factory methods instantiate the object and take the following arguments:

- sink An OutputStream
- schema A Schema for the data to be written
- use_legacy_format logical: write data formatted so that Arrow libraries versions 0.14 and lower can read it. Default is FALSE. You can also enable this by setting the environment variable ARROW_PRE_0_15_IPC_FORMAT=1.
- metadata_version: A string like "V5" or the equivalent integer indicating the Arrow IPC MetadataVersion. Default (NULL) will use the latest version, unless the environment variable ARROW_PRE_1_0_METADATA_VERSION=1, in which case it will be V4.

Methods

- $write(x): Write a RecordBatch, Table, or data.frame, dispatching to the methods below appropriately
- $write_batch(batch): Write a RecordBatch to stream
- $write_table(table): Write a Table to stream
- $close(): close stream. Note that this indicates end-of-file or end-of-stream–it does not close the connection to the sink. That needs to be closed separately.

See Also

write_ipc_stream() and write_feather() provide a much simpler interface for writing data to these formats and are sufficient for many use cases. write_to_raw() is a version that serializes data to a buffer.

Examples

```
tf <- tempfile()
on.exit(unlink(tf))

batch <- record_batch(chickwts)

# This opens a connection to the file in Arrow
file_obj <- FileOutputStream$create(tf)
# Pass that to a RecordBatchWriter to write data conforming to a schema
writer <- RecordBatchFileWriter$create(file_obj, batch$schema)
writer$write(batch)
# You may write additional batches to the stream, provided that they have
# the same schema.
# Call "close" on the writer to indicate end-of-file/stream
writer$close()
# Then, close the connection--closing the IPC message does not close the file
file_obj$close()

# Now, we have a file we can read from. Same pattern: open file connection,
# then pass it to a RecordBatchReader
read_file_obj <- ReadableFile$create(tf)
reader <- RecordBatchFileReader$create(read_file_obj)
# RecordBatchFileReader knows how many batches it has (StreamReader does not)
reader$num_record_batches
# We could consume the Reader by calling $read_next_batch() until all are,
```

```
# consumed, or we can call $read_table() to pull them all into a Table
tab <- reader$read_table()
# Call as.data.frame to turn that Table into an R data.frame
df <- as.data.frame(tab)
# This should be the same data we sent
all.equal(df, chickwts, check.attributes = FALSE)
# Unlike the Writers, we don't have to close RecordBatchReaders,
# but we do still need to close the file connection
read_file_obj$close()
```

`record_batch` *Create a RecordBatch*

Description

Create a RecordBatch

Usage

```
record_batch(..., schema = NULL)
```

Arguments

`...`	A `data.frame` or a named set of Arrays or vectors. If given a mixture of data.frames and vectors, the inputs will be autospliced together (see examples). Alternatively, you can provide a single Arrow IPC `InputStream`, `Message`, `Buffer`, or R `raw` object containing a `Buffer`.
`schema`	a Schema, or `NULL` (the default) to infer the schema from the data in `...`. When providing an Arrow IPC buffer, `schema` is required.

Examples

```
batch <- record_batch(name = rownames(mtcars), mtcars)
dim(batch)
dim(head(batch))
names(batch)
batch$mpg
batch[["cyl"]]
as.data.frame(batch[4:8, c("gear", "hp", "wt")])
```

register_scalar_function *Register user-defined functions*

Description

These functions support calling R code from query engine execution (i.e., a dplyr::mutate() or dplyr::filter() on a Table or Dataset). Use register_scalar_function() attach Arrow input and output types to an R function and make it available for use in the dplyr interface and/or call_function(). Scalar functions are currently the only type of user-defined function supported. In Arrow, scalar functions must be stateless and return output with the same shape (i.e., the same number of rows) as the input.

Usage

```
register_scalar_function(name, fun, in_type, out_type, auto_convert = FALSE)
```

Arguments

name	The function name to be used in the dplyr bindings
fun	An R function or rlang-style lambda expression. The function will be called with a first argument context which is a list() with elements batch_size (the expected length of the output) and output_type (the required DataType of the output) that may be used to ensure that the output has the correct type and length. Subsequent arguments are passed by position as specified by in_types. If auto_convert is TRUE, subsequent arguments are converted to R vectors before being passed to fun and the output is automatically constructed with the expected output type via as_arrow_array().
in_type	A DataType of the input type or a schema() for functions with more than one argument. This signature will be used to determine if this function is appropriate for a given set of arguments. If this function is appropriate for more than one signature, pass a list() of the above.
out_type	A DataType of the output type or a function accepting a single argument (types), which is a list() of DataTypes. If a function it must return a DataType.
auto_convert	Use TRUE to convert inputs before passing to fun and construct an Array of the correct type from the output. Use this option to write functions of R objects as opposed to functions of Arrow R6 objects.

Value

NULL, invisibly

Examples

```
library(dplyr, warn.conflicts = FALSE)

some_model <- lm(mpg ~ disp + cyl, data = mtcars)
register_scalar_function(
  "mtcars_predict_mpg",
  function(context, disp, cyl) {
    predict(some_model, newdata = data.frame(disp, cyl))
  },
  in_type = schema(disp = float64(), cyl = float64()),
  out_type = float64(),
  auto_convert = TRUE
)

as_arrow_table(mtcars) %>%
  transmute(mpg, mpg_predicted = mtcars_predict_mpg(disp, cyl)) %>%
  collect() %>%
  head()
```

---

s3_bucket *Connect to an AWS S3 bucket*

Description

`s3_bucket()` is a convenience function to create an `S3FileSystem` object that automatically detects the bucket's AWS region and holding onto the its relative path.

Usage

```
s3_bucket(bucket, ...)
```

Arguments

`bucket`	string S3 bucket name or path
`...`	Additional connection options, passed to `S3FileSystem$create()`

Details

By default, `s3_bucket` and other `S3FileSystem` functions only produce output for fatal errors or when printing their return values. When troubleshooting problems, it may be useful to increase the log level. See the Notes section in `S3FileSystem` for more information or see Examples below.

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")
```

Scalar *Arrow scalars*

Description

A `Scalar` holds a single value of an Arrow type.

Factory

The `Scalar$create()` factory method instantiates a `Scalar` and takes the following arguments:

- `x`: an R vector, list, or `data.frame`
- `type`: an optional data type for `x`. If omitted, the type will be inferred from the data.

Usage

```
a <- Scalar$create(x)
length(a)

print(a)
a == a
```

Methods

- `$ToString()`: convert to a string
- `$as_vector()`: convert to an R vector
- `$as_array()`: convert to an Arrow `Array`
- `$Equals(other)`: is this Scalar equal to `other`
- `$ApproxEquals(other)`: is this Scalar approximately equal to `other`
- `$is_valid`: is this Scalar valid
- `$null_count`: number of invalid values - 1 or 0
- `$type`: Scalar type
- `$cast(target_type, safe = TRUE, options = cast_options(safe))`: cast value to a different type

Examples

```
Scalar$create(pi)
Scalar$create(404)
# If you pass a vector into Scalar$create, you get a list containing your items
Scalar$create(c(1, 2, 3))

# Comparisons
my_scalar <- Scalar$create(99)
my_scalar$ApproxEquals(Scalar$create(99.00001)) # FALSE
my_scalar$ApproxEquals(Scalar$create(99.000009)) # TRUE
my_scalar$Equals(Scalar$create(99.000009)) # FALSE
my_scalar$Equals(Scalar$create(99L)) # FALSE (types don't match)

my_scalar$ToString()
```

`scalar` *Create an Arrow Scalar*

Description

Create an Arrow Scalar

Usage

```
scalar(x, type = NULL)
```

Arguments

`x`	An R vector, list, or `data.frame`
`type`	An optional data type for `x`. If omitted, the type will be inferred from the data.

Examples

```
scalar(pi)
scalar(404)
# If you pass a vector into scalar(), you get a list containing your items
scalar(c(1, 2, 3))

scalar(9) == scalar(10)
```

`Scanner` *Scan the contents of a dataset*

Description

A `Scanner` iterates over a Dataset's fragments and returns data according to given row filtering and column projection. A `ScannerBuilder` can help create one.

Factory

`Scanner$create()` wraps the `ScannerBuilder` interface to make a `Scanner`. It takes the following arguments:

- `dataset`: A `Dataset` or `arrow_dplyr_query` object, as returned by the `dplyr` methods on `Dataset`.
- `projection`: A character vector of column names to select columns or a named list of expressions
- `filter`: A `Expression` to filter the scanned rows by, or `TRUE` (default) to keep all rows.
- `use_threads`: logical: should scanning use multithreading? Default `TRUE`
- `...`: Additional arguments, currently ignored

Methods

`ScannerBuilder` has the following methods:

- `$Project(cols)`: Indicate that the scan should only return columns given by `cols`, a character vector of column names or a named list of Expression.
- `$Filter(expr)`: Filter rows by an Expression.
- `$UseThreads(threads)`: logical: should the scan use multithreading? The method's default input is `TRUE`, but you must call the method to enable multithreading because the scanner default is `FALSE`.
- `$BatchSize(batch_size)`: integer: Maximum row count of scanned record batches, default is 32K. If scanned record batches are overflowing memory then this method can be called to reduce their size.
- `$schema`: Active binding, returns the Schema of the Dataset
- `$Finish()`: Returns a `Scanner`

`Scanner` currently has a single method, `$ToTable()`, which evaluates the query and returns an Arrow Table.

Examples

```
# Set up directory for examples
tf <- tempfile()
dir.create(tf)
on.exit(unlink(tf))
```

```
write_dataset(mtcars, tf, partitioning="cyl")

ds <- open_dataset(tf)

scan_builder <- ds$NewScan()
scan_builder$Filter(Expression$field_ref("hp") > 100)
scan_builder$Project(list(hp_times_ten = 10 * Expression$field_ref("hp")))

# Once configured, call $Finish()
scanner <- scan_builder$Finish()

# Can get results as a table
as.data.frame(scanner$ToTable())

# Or as a RecordBatchReader
scanner$ToRecordBatchReader()
```

Schema *Schema class*

Description

A Schema is an Arrow object containing Fields, which map names to Arrow data types. Create a Schema when you want to convert an R data.frame to Arrow but don't want to rely on the default mapping of R types to Arrow types, such as when you want to choose a specific numeric precision, or when creating a Dataset and you want to ensure a specific schema rather than inferring it from the various files.

Many Arrow objects, including Table and Dataset, have a $schema method (active binding) that lets you access their schema.

Methods

- $ToString(): convert to a string
- $field(i): returns the field at index i (0-based)
- $GetFieldByName(x): returns the field with name x
- $WithMetadata(metadata): returns a new Schema with the key-value metadata set. Note that all list elements in metadata will be coerced to character.
- $code(namespace): returns the R code needed to generate this schema. Use namespace=TRUE to call with arrow::.

Active bindings

- $names: returns the field names (called in names(Schema))
- $num_fields: returns the number of fields (called in length(Schema))
- $fields: returns the list of Fields in the Schema, suitable for iterating over
- $HasMetadata: logical: does this Schema have extra metadata?
- $metadata: returns the key-value metadata as a named list. Modify or replace by assigning in (sch$metadata <- new_metadata). All list elements are coerced to string.

R Metadata

When converting a data.frame to an Arrow Table or RecordBatch, attributes from the `data.frame` are saved alongside tables so that the object can be reconstructed faithfully in R (e.g. with `as.data.frame()`). This metadata can be both at the top-level of the `data.frame` (e.g. `attributes(df)`) or at the column (e.g. `attributes(df$col_a)`) or for list columns only: element level (e.g. `attributes(df[1, "col_a"])`). For example, this allows for storing `haven` columns in a table and being able to faithfully re-create them when pulled back into R. This metadata is separate from the schema (column names and types) which is compatible with other Arrow clients. The R metadata is only read by R and is ignored by other clients (e.g. Pandas has its own custom metadata). This metadata is stored in `$metadata$r`.

Since Schema metadata keys and values must be strings, this metadata is saved by serializing R's attribute list structure to a string. If the serialized metadata exceeds 100Kb in size, by default it is compressed starting in version 3.0.0. To disable this compression (e.g. for tables that are compatible with Arrow versions before 3.0.0 and include large amounts of metadata), set the option `arrow.compress_metadata` to `FALSE`. Files with compressed metadata are readable by older versions of arrow, but the metadata is dropped.

---

schema *Create a schema or extract one from an object.*

Description

Create a schema or extract one from an object.

Usage

```
schema(...)
```

Arguments

...	fields, field name/data type pairs (or a list of), or object from which to extract a schema

See Also

Schema for detailed documentation of the Schema R6 object

Examples

```
# Create schema using pairs of field names and data types
schema(a = int32(), b = float64())

# Create a schema using a list of pairs of field names and data types
schema(list(a = int8(), b = string()))

# Create schema using fields
schema(
```

```
  field("b", double()),
  field("c", bool(), nullable = FALSE),
  field("d", string())
)

# Extract schemas from objects
df <- data.frame(col1 = 2:4, col2 = c(0.1, 0.3, 0.5))
tab1 <- arrow_table(df)
schema(tab1)
tab2 <- arrow_table(df, schema = schema(col1 = int8(), col2 = float32()))
schema(tab2)
```

---

show_exec_plan *Show the details of an Arrow Execution Plan*

Description

This is a function which gives more details about the logical query plan that will be executed when evaluating an arrow_dplyr_query object. It calls the C++ ExecPlan object's print method. Functionally, it is similar to dplyr::explain(). This function is used as the dplyr::explain() and dplyr::show_query() methods.

Usage

```
show_exec_plan(x)
```

Arguments

x	an arrow_dplyr_query to print the ExecPlan for.

Value

x, invisibly.

Examples

```
library(dplyr)
mtcars %>%
  arrow_table() %>%
  filter(mpg > 20) %>%
  mutate(x = gear / carb) %>%
  show_exec_plan()
```

`Table` *Table class*

Description

A Table is a sequence of chunked arrays. They have a similar interface to record batches, but they can be composed from multiple record batches or chunked arrays.

S3 Methods and Usage

Tables are data-frame-like, and many methods you expect to work on a `data.frame` are implemented for `Table`. This includes `[`, `[[`, `$`, `names`, `dim`, `nrow`, `ncol`, `head`, and `tail`. You can also pull the data from an Arrow table into R with `as.data.frame()`. See the examples.

A caveat about the `$` method: because `Table` is an `R6` object, `$` is also used to access the object's methods (see below). Methods take precedence over the table's columns. So, `tab$Slice` would return the "Slice" method function even if there were a column in the table called "Slice".

R6 Methods

In addition to the more R-friendly S3 methods, a `Table` object has the following R6 methods that map onto the underlying C++ methods:

- `$column(i)`: Extract a `ChunkedArray` by integer position from the table
- `$ColumnNames()`: Get all column names (called by `names(tab)`)
- `$nbytes()`: Total number of bytes consumed by the elements of the table
- `$RenameColumns(value)`: Set all column names (called by `names(tab) <- value`)
- `$GetColumnByName(name)`: Extract a `ChunkedArray` by string name
- `$field(i)`: Extract a `Field` from the table schema by integer position
- `$SelectColumns(indices)`: Return new `Table` with specified columns, expressed as 0-based integers.
- `$Slice(offset, length = NULL)`: Create a zero-copy view starting at the indicated integer offset and going for the given length, or to the end of the table if `NULL`, the default.
- `$Take(i)`: return an `Table` with rows at positions given by integers `i`. If `i` is an Arrow `Array` or `ChunkedArray`, it will be coerced to an R vector before taking.
- `$Filter(i, keep_na = TRUE)`: return an `Table` with rows at positions where logical vector or Arrow boolean-type `(Chunked)Array` `i` is `TRUE`.
- `$SortIndices(names, descending = FALSE)`: return an `Array` of integer row positions that can be used to rearrange the `Table` in ascending or descending order by the first named column, breaking ties with further named columns. `descending` can be a logical vector of length one or of the same length as `names`.
- `$serialize(output_stream, ...)`: Write the table to the given OutputStream
- `$cast(target_schema, safe = TRUE, options = cast_options(safe))`: Alter the schema of the record batch.

There are also some active bindings:

- $num_columns
- $num_rows
- $schema
- $metadata: Returns the key-value metadata of the Schema as a named list. Modify or replace by assigning in (tab$metadata <- new_metadata). All list elements are coerced to string. See schema() for more information.
- $columns: Returns a list of ChunkedArrays

---

to_arrow *Create an Arrow object from a DuckDB connection*

Description

This can be used in pipelines that pass data back and forth between Arrow and DuckDB.

Usage

```
to_arrow(.data)
```

Arguments

.data	the object to be converted

Details

Note that you can only call collect() or compute() on the result of this function once. To work around this limitation, you should either only call collect() as the final step in a pipeline or call as_arrow_table() on the result to materialize the entire Table in-memory.

Value

A RecordBatchReader.

Examples

```
library(dplyr)

ds <- InMemoryDataset$create(mtcars)

ds %>%
  filter(mpg < 30) %>%
  to_duckdb() %>%
  group_by(cyl) %>%
  summarize(mean_mpg = mean(mpg, na.rm = TRUE)) %>%
  to_arrow() %>%
  collect()
```

to_duckdb *Create a (virtual) DuckDB table from an Arrow object*

Description

This will do the necessary configuration to create a (virtual) table in DuckDB that is backed by the Arrow object given. No data is copied or modified until `collect()` or `compute()` are called or a query is run against the table.

Usage

```
to_duckdb(
  .data,
  con = arrow_duck_connection(),
  table_name = unique_arrow_tablename(),
  auto_disconnect = TRUE
)
```

Arguments

`.data`	the Arrow object (e.g. Dataset, Table) to use for the DuckDB table
`con`	a DuckDB connection to use (default will create one and store it in `options("arrow_duck_con")`)
`table_name`	a name to use in DuckDB for this object. The default is a unique string `"arrow_"` followed by numbers.
`auto_disconnect`	should the table be automatically cleaned up when the resulting object is removed (and garbage collected)? Default: `TRUE`

Details

The result is a dbplyr-compatible object that can be used in d(b)plyr pipelines.

If `auto_disconnect = TRUE`, the DuckDB table that is created will be configured to be unregistered when the `tbl` object is garbage collected. This is helpful if you don't want to have extra table objects in DuckDB after you've finished using them.

Value

A `tbl` of the new table in DuckDB

Examples

```
library(dplyr)

ds <- InMemoryDataset$create(mtcars)

ds %>%
  filter(mpg < 30) %>%
```

```
  group_by(cyl) %>%
  to_duckdb() %>%
  slice_min(disp)
```

---

unify_schemas *Combine and harmonize schemas*

Description

Combine and harmonize schemas

Usage

```
unify_schemas(..., schemas = list(...))
```

Arguments

...	Schemas to unify
schemas	Alternatively, a list of schemas

Value

A Schema with the union of fields contained in the inputs, or NULL if any of schemas is NULL

Examples

```
a <- schema(b = double(), c = bool())
z <- schema(b = double(), k = utf8())
unify_schemas(a, z)
```

---

value_counts *table for Arrow objects*

Description

This function tabulates the values in the array and returns a table of counts.

Usage

```
value_counts(x)
```

Arguments

x	Array or ChunkedArray

Value

A StructArray containing "values" (same type as x) and "counts" Int64.

Examples

```
cyl_vals <- Array$create(mtcars$cyl)
counts <- value_counts(cyl_vals)
```

---

vctrs_extension_array *Extension type for generic typed vectors*

Description

Most common R vector types are converted automatically to a suitable Arrow data type without the need for an extension type. For vector types whose conversion is not suitably handled by default, you can create a vctrs_extension_array(), which passes vctrs::vec_data() to Array$create() and calls vctrs::vec_restore() when the Array is converted back into an R vector.

Usage

```
vctrs_extension_array(x, ptype = vctrs::vec_ptype(x), storage_type = NULL)

vctrs_extension_type(x, storage_type = infer_type(vctrs::vec_data(x)))
```

Arguments

x	A vctr (i.e., vctrs::vec_is() returns TRUE).
ptype	A vctrs::vec_ptype(), which is usually a zero-length version of the object with the appropriate attributes set. This value will be serialized using serialize(), so it should not refer to any R object that can't be saved/reloaded.
storage_type	The data type of the underlying storage array.

Value

- vctrs_extension_array() returns an ExtensionArray instance with a vctrs_extension_type().
- vctrs_extension_type() returns an ExtensionType instance for the extension name "arrow.r.vctrs".

Examples

```
(array <- vctrs_extension_array(as.POSIXlt("2022-01-02 03:45", tz = "UTC")))
array$type
as.vector(array)

temp_feather <- tempfile()
write_feather(arrow_table(col = array), temp_feather)
read_feather(temp_feather)
unlink(temp_feather)
```

write_csv_arrow *Write CSV file to disk*

Description

Write CSV file to disk

Usage

```
write_csv_arrow(
  x,
  sink,
  file = NULL,
  include_header = TRUE,
  col_names = NULL,
  batch_size = 1024L,
  na = "",
  write_options = NULL,
  ...
)
```

Arguments

x	data.frame, RecordBatch, or Table
sink	A string file path, connection, URI, or OutputStream, or path in a file system (SubTreeFileSystem)
file	file name. Specify this or sink, not both.
include_header	Whether to write an initial header line with column names
col_names	identical to include_header. Specify this or include_headers, not both.
batch_size	Maximum number of rows processed at a time. Default is 1024.
na	value to write for NA values. Must not contain quote marks. Default is "".
write_options	see CSV write options
...	additional parameters

Value

The input x, invisibly. Note that if sink is an OutputStream, the stream will be left open.

Examples

```
tf <- tempfile()
on.exit(unlink(tf))
write_csv_arrow(mtcars, tf)
```

`write_dataset` *Write a dataset*

Description

This function allows you to write a dataset. By writing to more efficient binary storage formats, and by specifying relevant partitioning, you can make it much faster to read and query.

Usage

```
write_dataset(
  dataset,
  path,
  format = c("parquet", "feather", "arrow", "ipc", "csv", "tsv", "txt", "text"),
  partitioning = dplyr::group_vars(dataset),
  basename_template = paste0("part-{i}.", as.character(format)),
  hive_style = TRUE,
  existing_data_behavior = c("overwrite", "error", "delete_matching"),
  max_partitions = 1024L,
  max_open_files = 900L,
  max_rows_per_file = 0L,
  min_rows_per_group = 0L,
  max_rows_per_group = bitwShiftL(1, 20),
  ...
)
```

Arguments

`dataset`	Dataset, RecordBatch, Table, `arrow_dplyr_query`, or `data.frame`. If an `arrow_dplyr_query`, the query will be evaluated and the result will be written. This means that you can `select()`, `filter()`, `mutate()`, etc. to transform the data before it is written if you need to.
`path`	string path, URI, or `SubTreeFileSystem` referencing a directory to write to (directory will be created if it does not exist)
`format`	a string identifier of the file format. Default is to use "parquet" (see FileFormat)
`partitioning`	`Partitioning` or a character vector of columns to use as partition keys (to be written as path segments). Default is to use the current `group_by()` columns.
`basename_template`	string template for the names of files to be written. Must contain `"{i}"`, which will be replaced with an autoincremented integer to generate basenames of datafiles. For example, `"part-{i}.arrow"` will yield `"part-0.arrow"`, ... If not specified, it defaults to `"part-{i}.<default extension>"`.
`hive_style`	logical: write partition segments as Hive-style (`key1=value1/key2=value2/file.ext`) or as just bare values. Default is `TRUE`.

`existing_data_behavior`
: The behavior to use when there is already data in the destination directory. Must be one of "overwrite", "error", or "delete_matching".

  - "overwrite" (the default) then any new files created will overwrite existing files
  - "error" then the operation will fail if the destination directory is not empty
  - "delete_matching" then the writer will delete any existing partitions if data is going to be written to those partitions and will leave alone partitions which data is not written to.

`max_partitions`
: maximum number of partitions any batch may be written into. Default is 1024L.

`max_open_files`
: maximum number of files that can be left opened during a write operation. If greater than 0 then this will limit the maximum number of files that can be left open. If an attempt is made to open too many files then the least recently used file will be closed. If this setting is set too low you may end up fragmenting your data into many small files. The default is 900 which also allows some # of files to be open by the scanner before hitting the default Linux limit of 1024.

`max_rows_per_file`
: maximum number of rows per file. If greater than 0 then this will limit how many rows are placed in any single file. Default is 0L.

`min_rows_per_group`
: write the row groups to the disk when this number of rows have accumulated. Default is 0L.

`max_rows_per_group`
: maximum rows allowed in a single group and when this number of rows is exceeded, it is split and the next set of rows is written to the next group. This value must be set such that it is greater than `min_rows_per_group`. Default is 1024 * 1024.

`...`
: additional format-specific arguments. For available Parquet options, see `write_parquet()`. The available Feather options are:

  - `use_legacy_format` logical: write data formatted so that Arrow libraries versions 0.14 and lower can read it. Default is `FALSE`. You can also enable this by setting the environment variable `ARROW_PRE_0_15_IPC_FORMAT=1`.
  - `metadata_version`: A string like "V5" or the equivalent integer indicating the Arrow IPC MetadataVersion. Default (`NULL`) will use the latest version, unless the environment variable `ARROW_PRE_1_0_METADATA_VERSION=1`, in which case it will be V4.
  - `codec`: A Codec which will be used to compress body buffers of written files. Default (NULL) will not compress body buffers.
  - `null_fallback`: character to be used in place of missing values (`NA` or `NULL`) when using Hive-style partitioning. See `hive_partition()`.

Value

The input `dataset`, invisibly

Examples

```
# You can write datasets partitioned by the values in a column (here: "cyl").
# This creates a structure of the form cyl=X/part-Z.parquet.
one_level_tree <- tempfile()
write_dataset(mtcars, one_level_tree, partitioning = "cyl")
list.files(one_level_tree, recursive = TRUE)

# You can also partition by the values in multiple columns
# (here: "cyl" and "gear").
# This creates a structure of the form cyl=X/gear=Y/part-Z.parquet.
two_levels_tree <- tempfile()
write_dataset(mtcars, two_levels_tree, partitioning = c("cyl", "gear"))
list.files(two_levels_tree, recursive = TRUE)

# In the two previous examples we would have:
# X = {4,6,8}, the number of cylinders.
# Y = {3,4,5}, the number of forward gears.
# Z = {0,1,2}, the number of saved parts, starting from 0.

# You can obtain the same result as as the previous examples using arrow with
# a dplyr pipeline. This will be the same as two_levels_tree above, but the
# output directory will be different.
library(dplyr)
two_levels_tree_2 <- tempfile()
mtcars %>%
  group_by(cyl, gear) %>%
  write_dataset(two_levels_tree_2)
list.files(two_levels_tree_2, recursive = TRUE)

# And you can also turn off the Hive-style directory naming where the column
# name is included with the values by using `hive_style = FALSE`.

# Write a structure X/Y/part-Z.parquet.
two_levels_tree_no_hive <- tempfile()
mtcars %>%
  group_by(cyl, gear) %>%
  write_dataset(two_levels_tree_no_hive, hive_style = FALSE)
list.files(two_levels_tree_no_hive, recursive = TRUE)
```

`write_delim_dataset` *Write a dataset into partitioned flat files.*

Description

The `write_*_dataset()` are a family of wrappers around write_dataset to allow for easy switching between functions for writing datasets.

Usage

```
write_delim_dataset(
  dataset,
  path,
  partitioning = dplyr::group_vars(dataset),
  basename_template = "part-{i}.txt",
  hive_style = TRUE,
  existing_data_behavior = c("overwrite", "error", "delete_matching"),
  max_partitions = 1024L,
  max_open_files = 900L,
  max_rows_per_file = 0L,
  min_rows_per_group = 0L,
  max_rows_per_group = bitwShiftL(1, 20),
  col_names = TRUE,
  batch_size = 1024L,
  delim = ",",
  na = "",
  eol = "\n",
  quote = c("needed", "all", "none")
)

write_csv_dataset(
  dataset,
  path,
  partitioning = dplyr::group_vars(dataset),
  basename_template = "part-{i}.csv",
  hive_style = TRUE,
  existing_data_behavior = c("overwrite", "error", "delete_matching"),
  max_partitions = 1024L,
  max_open_files = 900L,
  max_rows_per_file = 0L,
  min_rows_per_group = 0L,
  max_rows_per_group = bitwShiftL(1, 20),
  col_names = TRUE,
  batch_size = 1024L,
  delim = ",",
  na = "",
  eol = "\n",
  quote = c("needed", "all", "none")
)

write_tsv_dataset(
  dataset,
  path,
  partitioning = dplyr::group_vars(dataset),
  basename_template = "part-{i}.tsv",
  hive_style = TRUE,
  existing_data_behavior = c("overwrite", "error", "delete_matching"),
```

```
  max_partitions = 1024L,
  max_open_files = 900L,
  max_rows_per_file = 0L,
  min_rows_per_group = 0L,
  max_rows_per_group = bitwShiftL(1, 20),
  col_names = TRUE,
  batch_size = 1024L,
  na = "",
  eol = "\n",
  quote = c("needed", "all", "none")
)
```

Arguments

dataset	Dataset, RecordBatch, Table, arrow_dplyr_query, or data.frame. If an arrow_dplyr_query, the query will be evaluated and the result will be written. This means that you can select(), filter(), mutate(), etc. to transform the data before it is written if you need to.
path	string path, URI, or SubTreeFileSystem referencing a directory to write to (directory will be created if it does not exist)
partitioning	Partitioning or a character vector of columns to use as partition keys (to be written as path segments). Default is to use the current group_by() columns.
basename_template	string template for the names of files to be written. Must contain "{i}", which will be replaced with an autoincremented integer to generate basenames of datafiles. For example, "part-{i}.arrow" will yield "part-0.arrow", ... If not specified, it defaults to "part-{i}.<default extension>".
hive_style	logical: write partition segments as Hive-style (key1=value1/key2=value2/file.ext) or as just bare values. Default is TRUE.
existing_data_behavior	The behavior to use when there is already data in the destination directory. Must be one of "overwrite", "error", or "delete_matching". • "overwrite" (the default) then any new files created will overwrite existing files • "error" then the operation will fail if the destination directory is not empty • "delete_matching" then the writer will delete any existing partitions if data is going to be written to those partitions and will leave alone partitions which data is not written to.
max_partitions	maximum number of partitions any batch may be written into. Default is 1024L.
max_open_files	maximum number of files that can be left opened during a write operation. If greater than 0 then this will limit the maximum number of files that can be left open. If an attempt is made to open too many files then the least recently used file will be closed. If this setting is set too low you may end up fragmenting your data into many small files. The default is 900 which also allows some # of files to be open by the scanner before hitting the default Linux limit of 1024.

max_rows_per_file
: maximum number of rows per file. If greater than 0 then this will limit how many rows are placed in any single file. Default is 0L.

min_rows_per_group
: write the row groups to the disk when this number of rows have accumulated. Default is 0L.

max_rows_per_group
: maximum rows allowed in a single group and when this number of rows is exceeded, it is split and the next set of rows is written to the next group. This value must be set such that it is greater than `min_rows_per_group`. Default is 1024 * 1024.

`col_names`
: Whether to write an initial header line with column names.

`batch_size`
: Maximum number of rows processed at a time. Default is 1024L.

`delim`
: Delimiter used to separate values. Defaults to `","` for `write_delim_dataset()` and `write_csv_dataset()`, and `"\t"` for `write_tsv_dataset()`. Cannot be changed for `write_tsv_dataset()`.

`na`
: a character vector of strings to interpret as missing values. Quotes are not allowed in this string. The default is an empty string `""`.

`eol`
: the end of line character to use for ending rows. The default is `"\n"`.

`quote`
: How to handle fields which contain characters that need to be quoted.
  - `needed` - Enclose all strings and binary values in quotes which need them, because their CSV rendering can contain quotes itself (the default)
  - `all` - Enclose all valid values in quotes. Nulls are not quoted. May cause readers to interpret all values as strings if schema is inferred.
  - `none` - Do not enclose any values in quotes. Prevents values from containing quotes ("), cell delimiters (,) or line endings (\r, \n), (following RFC4180). If values contain these characters, an error is caused when attempting to write.

Value

The input `dataset`, invisibly.

See Also

`write_dataset()`

`write_feather` *Write a Feather file (an Arrow IPC file)*

Description

Feather provides binary columnar serialization for data frames. It is designed to make reading and writing data frames efficient, and to make sharing data across data analysis languages easy. `write_feather()` can write both the Feather Version 1 (V1), a legacy version available starting in 2016, and the Version 2 (V2), which is the Apache Arrow IPC file format. The default version is V2. V1 files are distinct from Arrow IPC files and lack many features, such as the ability to store all Arrow data tyeps, and compression support. `write_ipc_file()` can only write V2 files.

Usage

```
write_feather(
  x,
  sink,
  version = 2,
  chunk_size = 65536L,
  compression = c("default", "lz4", "lz4_frame", "uncompressed", "zstd"),
  compression_level = NULL
)

write_ipc_file(
  x,
  sink,
  chunk_size = 65536L,
  compression = c("default", "lz4", "lz4_frame", "uncompressed", "zstd"),
  compression_level = NULL
)
```

Arguments

x	data.frame, RecordBatch, or Table
sink	A string file path, connection, URI, or OutputStream, or path in a file system (SubTreeFileSystem)
version	integer Feather file version, Version 1 or Version 2. Version 2 is the default.
chunk_size	For V2 files, the number of rows that each chunk of data should have in the file. Use a smaller chunk_size when you need faster random row access. Default is 64K. This option is not supported for V1.
compression	Name of compression codec to use, if any. Default is "lz4" if LZ4 is available in your build of the Arrow C++ library, otherwise "uncompressed". "zstd" is the other available codec and generally has better compression ratios in exchange for slower read and write performance. "lz4" is shorthand for the "lz4_frame" codec. See codec_is_available() for details. TRUE and FALSE can also be used in place of "default" and "uncompressed". This option is not supported for V1.
compression_level	If compression is "zstd", you may specify an integer compression level. If omitted, the compression codec's default compression level is used.

Value

The input x, invisibly. Note that if sink is an OutputStream, the stream will be left open.

See Also

RecordBatchWriter for lower-level access to writing Arrow IPC data.

Schema for information about schemas and metadata handling.

Examples

```
# We recommend the ".arrow" extension for Arrow IPC files (Feather V2).
tf1 <- tempfile(fileext = ".feather")
tf2 <- tempfile(fileext = ".arrow")
tf3 <- tempfile(fileext = ".arrow")
on.exit({
  unlink(tf1)
  unlink(tf2)
  unlink(tf3)
})
write_feather(mtcars, tf1, version = 1)
write_feather(mtcars, tf2)
write_ipc_file(mtcars, tf3)
```

---

write_ipc_stream *Write Arrow IPC stream format*

Description

Apache Arrow defines two formats for serializing data for interprocess communication (IPC): a "stream" format and a "file" format, known as Feather. `write_ipc_stream()` and `write_feather()` write those formats, respectively.

Usage

```
write_ipc_stream(x, sink, ...)
```

Arguments

`x`	`data.frame`, RecordBatch, or Table
`sink`	A string file path, connection, URI, or OutputStream, or path in a file system (`SubTreeFileSystem`)
`...`	extra parameters passed to `write_feather()`.

Value

`x`, invisibly.

See Also

`write_feather()` for writing IPC files. `write_to_raw()` to serialize data to a buffer. RecordBatchWriter for a lower-level interface.

Examples

```
tf <- tempfile()
on.exit(unlink(tf))
write_ipc_stream(mtcars, tf)
```

`write_parquet` *Write Parquet file to disk*

Description

Parquet is a columnar storage file format. This function enables you to write Parquet files from R.

Usage

```
write_parquet(
  x,
  sink,
  chunk_size = NULL,
  version = "2.4",
  compression = default_parquet_compression(),
  compression_level = NULL,
  use_dictionary = NULL,
  write_statistics = NULL,
  data_page_size = NULL,
  use_deprecated_int96_timestamps = FALSE,
  coerce_timestamps = NULL,
  allow_truncated_timestamps = FALSE
)
```

Arguments

`x`	`data.frame`, RecordBatch, or Table
`sink`	A string file path, connection, URI, or OutputStream, or path in a file system (`SubTreeFileSystem`)
`chunk_size`	how many rows of data to write to disk at once. This directly corresponds to how many rows will be in each row group in parquet. If `NULL`, a best guess will be made for optimal size (based on the number of columns and number of rows), though if the data has fewer than 250 million cells (rows x cols), then the total number of rows is used.
`version`	parquet version: "1.0", "2.0" (deprecated), "2.4" (default), "2.6", or "latest" (currently equivalent to 2.6). Numeric values are coerced to character.
`compression`	compression algorithm. Default "snappy". See details.
`compression_level`	compression level. Meaning depends on compression algorithm
`use_dictionary`	logical: use dictionary encoding? Default `TRUE`
`write_statistics`	logical: include statistics? Default `TRUE`
`data_page_size`	Set a target threshold for the approximate encoded size of data pages within a column chunk (in bytes). Default 1 MiB.

use_deprecated_int96_timestamps
: logical: write timestamps to INT96 Parquet format, which has been deprecated? Default FALSE.

coerce_timestamps
: Cast timestamps a particular resolution. Can be NULL, "ms" or "us". Default NULL (no casting)

allow_truncated_timestamps
: logical: Allow loss of data when coercing timestamps to a particular resolution. E.g. if microsecond or nanosecond data is lost when coercing to "ms", do not raise an exception. Default FALSE.

Details

Due to features of the format, Parquet files cannot be appended to. If you want to use the Parquet format but also want the ability to extend your dataset, you can write to additional Parquet files and then treat the whole directory of files as a Dataset you can query. See the dataset article for examples of this.

The parameters compression, compression_level, use_dictionary and write_statistics support various patterns:

- The default NULL leaves the parameter unspecified, and the C++ library uses an appropriate default for each column (defaults listed above)
- A single, unnamed, value (e.g. a single string for compression) applies to all columns
- An unnamed vector, of the same size as the number of columns, to specify a value for each column, in positional order
- A named vector, to specify the value for the named columns, the default value for the setting is used when not supplied

The compression argument can be any of the following (case-insensitive): "uncompressed", "snappy", "gzip", "brotli", "zstd", "lz4", "lzo" or "bz2". Only "uncompressed" is guaranteed to be available, but "snappy" and "gzip" are almost always included. See codec_is_available(). The default "snappy" is used if available, otherwise "uncompressed". To disable compression, set compression = "uncompressed". Note that "uncompressed" columns may still have dictionary encoding.

Value

the input x invisibly.

See Also

ParquetFileWriter for a lower-level interface to Parquet writing.

Examples

```
tf1 <- tempfile(fileext = ".parquet")
write_parquet(data.frame(x = 1:5), tf1)

# using compression
if (codec_is_available("gzip")) {
```

```
  tf2 <- tempfile(fileext = ".gz.parquet")
  write_parquet(data.frame(x = 1:5), tf2, compression = "gzip", compression_level = 5)
}
```

`write_to_raw` *Write Arrow data to a raw vector*

Description

`write_ipc_stream()` and `write_feather()` write data to a sink and return the data (`data.frame`, `RecordBatch`, or `Table`) they were given. This function wraps those so that you can serialize data to a buffer and access that buffer as a `raw` vector in R.

Usage

```
write_to_raw(x, format = c("stream", "file"))
```

Arguments

`x`	`data.frame`, RecordBatch, or Table
`format`	one of `c("stream", "file")`, indicating the IPC format to use

Value

A `raw` vector containing the bytes of the IPC serialized data.

Examples

```
# The default format is "stream"
mtcars_raw <- write_to_raw(mtcars)
```

Index