

# Package ‘arrow’

November 20, 2021

**Title** Integration to 'Apache' 'Arrow'

**Version** 6.0.1

**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 (>= 3.3)

**License** Apache License (>= 2.0)

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

**BugReports** <https://issues.apache.org/jira/projects/ARROW/issues>

**Encoding** UTF-8

**Language** en-US

**SystemRequirements** C++11; for AWS S3 support on Linux, libcurl and openssl (optional)

**Biarch** true

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

**RoxygenNote** 7.1.2

**Config/testthat/edition** 3

**VignetteBuilder** knitr

**Suggests** DBI, dbplyr, decor, distro, dplyr, duckdb (>= 0.2.8), hms, knitr, lubridate, pkgload, reticulate, rmarkdown, stringi, stringr, testthat, tibble, withr

**Collate** 'arrowExports.R' 'enums.R' 'arrow-package.R' 'type.R' 'array-data.R' 'arrow-datum.R' 'array.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' 'deprecated.R' 'dictionary.R'

'dplyr-arrange.R' 'dplyr-collect.R' 'dplyr-count.R'  
 'dplyr-distinct.R' 'dplyr-eval.R' 'dplyr-filter.R'  
 'expression.R' 'dplyr-functions.R' 'dplyr-group-by.R'  
 'dplyr-join.R' 'dplyr-mutate.R' 'dplyr-select.R'  
 'dplyr-summarize.R' 'record-batch.R' 'table.R' 'dplyr.R'  
 'duckdb.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-tidysselect.R' 'schema.R' 'util.R'

**NeedsCompilation** yes**Author** Neal Richardson [aut, cre],

Ian Cook [aut],

Nic Crane [aut],

Jonathan Keane [aut],

Romain François [aut] (<<https://orcid.org/0000-0002-2444-4226>>),

Jeroen Ooms [aut],

Javier Luraschi [ctb],

Karl Dunkle Werner [ctb] (<<https://orcid.org/0000-0003-0523-7309>>),

Jeffrey Wong [ctb],

Apache Arrow [aut, cph]

**Maintainer** Neal Richardson <neal@ursalabs.org>**Repository** CRAN**Date/Publication** 2021-11-20 05:40:02 UTC**R topics documented:**

array . . . . .	4
ArrayData . . . . .	6
arrow_available . . . . .	6
arrow_info . . . . .	7
buffer . . . . .	8
call_function . . . . .	9
ChunkedArray . . . . .	10
Codec . . . . .	11
codec_is_available . . . . .	12
compression . . . . .	12
copy_files . . . . .	13
cpu_count . . . . .	13
create_package_with_all_dependencies . . . . .	14
CsvReadOptions . . . . .	15
CsvTableReader . . . . .	17
data-type . . . . .	17
Dataset . . . . .	20
dataset_factory . . . . .	21

DataType . . . . .	22
dictionary . . . . .	23
DictionaryType . . . . .	23
Expression . . . . .	24
FeatherReader . . . . .	24
Field . . . . .	25
FileFormat . . . . .	25
FileInfo . . . . .	27
FileSelector . . . . .	27
FileSystem . . . . .	28
FileWriteOptions . . . . .	29
FixedWidthType . . . . .	29
flight_connect . . . . .	30
flight_get . . . . .	30
flight_put . . . . .	31
FragmentScanOptions . . . . .	31
hive_partition . . . . .	32
InputStream . . . . .	33
install_arrow . . . . .	33
install_pyarrow . . . . .	35
io_thread_count . . . . .	35
list_compute_functions . . . . .	36
list_flights . . . . .	37
load_flight_server . . . . .	37
map_batches . . . . .	38
match_arrow . . . . .	38
Message . . . . .	39
MessageReader . . . . .	40
mmap_create . . . . .	40
mmap_open . . . . .	40
open_dataset . . . . .	41
OutputStream . . . . .	43
ParquetArrowReaderProperties . . . . .	44
ParquetFileReader . . . . .	44
ParquetFileWriter . . . . .	46
ParquetWriterProperties . . . . .	46
Partitioning . . . . .	47
read_arrow . . . . .	48
read_delim_arrow . . . . .	49
read_feather . . . . .	53
read_json_arrow . . . . .	54
read_message . . . . .	55
read_parquet . . . . .	55
read_schema . . . . .	56
RecordBatch . . . . .	56
RecordBatchReader . . . . .	58
RecordBatchWriter . . . . .	59
s3_bucket . . . . .	61

Scalar . . . . .	62
Scanner . . . . .	62
Schema . . . . .	63
Table . . . . .	65
to_arrow . . . . .	66
to_duckdb . . . . .	67
type . . . . .	68
unify_schemas . . . . .	69
value_counts . . . . .	70
write_arrow . . . . .	70
write_csv_arrow . . . . .	71
write_dataset . . . . .	72
write_feather . . . . .	74
write_parquet . . . . .	75
write_to_raw . . . . .	77

## Index 79

---

array	<i>Arrow Arrays</i>
-------	---------------------

---

### 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
- `$length()`: Size in the number of elements this array contains
- `$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  $O(\text{length})$

## 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\_available

*Is the C++ Arrow library available?*


---

### Description

You won't generally need to call these function, but they're made available for diagnostic purposes.

### Usage

```
arrow_available()

arrow_with_dataset()

arrow_with_parquet()

arrow_with_s3()

arrow_with_json()
```

**Value**

TRUE or FALSE depending on whether the package was installed with:

- The Arrow C++ library (check with `arrow_available()`)
- Arrow Dataset support enabled (check with `arrow_with_dataset()`)
- Parquet support enabled (check with `arrow_with_parquet()`)
- JSON support enabled (check with `arrow_with_json()`)
- Amazon S3 support enabled (check with `arrow_with_s3()`)

**See Also**

If any of these are FALSE, see `vignette("install", package = "arrow")` for guidance on reinstalling the package.

**Examples**

```
arrow_available()
arrow_with_dataset()
arrow_with_parquet()
arrow_with_json()
arrow_with_s3()
```

---

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()
```

**Value**

A list including version information, boolean "capabilities", and statistics from Arrow's memory allocator, and also Arrow's run-time information.

---

buffer

*Buffer class*

---

### Description

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

### 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

### 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
```



---

call_function	<i>Call an Arrow compute function</i>
---------------	---------------------------------------

---

### 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

<code>function_name</code>	string Arrow compute function name
<code>...</code>	Function arguments, which may include <code>Array</code> , <code>ChunkedArray</code> , <code>Scalar</code> , <code>RecordBatch</code> , or <code>Table</code> .
<code>args</code>	list arguments as an alternative to specifying in <code>...</code>
<code>options</code>	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](#).

### Usage

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

### Arguments

...	Vectors to coerce
type	currently ignored

### 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
- `$as_vector()`: convert to an `R` vector
- `$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  $O(\text{length})$

**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
```

---

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	<i>Check whether a compression codec is available</i>
--------------------	---

---

### 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	<i>Compressed stream classes</i>
-------------	----------------------------------

---

### 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

---

copy_files	<i>Copy files between FileSystems</i>
------------	---------------------------------------

---

**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	<i>Manage the global CPU thread pool in libarrow</i>
-----------	--

---

**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/master/r/R/i`

- 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)
```

---

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".

`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`
- `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`
- `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.

`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.



**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.

---

 data-type

*Apache 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"))
null()
timestamp(unit = c("s", "ms", "us", "ns"), timezone = "")
decimal(precision, scale)
struct(...)
```

`list_of(type)`

`large_list_of(type)`

`fixed_size_list_of(type, list_size)`

### Arguments

<code>byte_width</code>	byte width for <code>FixedSizeBinary</code> type.
<code>unit</code>	For time/timestamp types, the time unit. <code>time32()</code> can take either "s" or "ms", while <code>time64()</code> can be "us" or "ns". <code>timestamp()</code> can take any of those four values.
<code>timezone</code>	For <code>timestamp()</code> , an optional time zone string.
<code>precision</code>	For <code>decimal()</code> , precision
<code>scale</code>	For <code>decimal()</code> , scale
<code>...</code>	For <code>struct()</code> , a named list of types to define the struct columns
<code>type</code>	For <code>list_of()</code> , a data type to make a list-of-type
<code>list_size</code>	list size for <code>FixedSizeList</code> type.

### 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)`.

### 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")
```

---

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:

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

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 be constructed using one or more [DatasetFactory](#)s. 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"),
  partitioning = NULL,
  ...
)
```

## Arguments

- |                         |  |
|-------------------------|--|
| <code>x</code>          | A string path to a directory containing data files, a vector of one or more string paths to data files, or a list of <a href="#">DatasetFactory</a> objects whose datasets should be combined. If this argument is specified it will be used to construct a <a href="#">UnionDatasetFactory</a> and other arguments will be ignored. |
| <code>filesystem</code> | A <a href="#">FileSystem</a> object; if omitted, the <a href="#">FileSystem</a> will be detected from <code>x</code>   |
| <code>format</code>     | A <a href="#">FileFormat</a> object, or a string identifier of the format of the files in <code>x</code> . 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 <ul style="list-style-type: none"> <li>• 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, <code>schema(year = int16(), month = int8())</code> would create partitions for file paths like "2019/01/file.parquet", "2019/02/file.parquet", etc.</li> <li>• 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)</li> <li>• A <code>HivePartitioning</code> or <code>HivePartitioningFactory</code>, as returned by <code>hive_partitioning()</code> which parses explicit or autodetected fields from Hive-style path segments</li> <li>• NULL for no partitioning</li> </ul>
...	Additional format-specific options, passed to <code>FileFormat\$create()</code> . 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 <code>read_csv_arrow()</code> ("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	<i>class arrow::DataType</i>
----------	------------------------------

---

### Description

class `arrow::DataType`

### Methods

TODO

---

dictionary	<i>Create a dictionary type</i>
------------	---------------------------------

---

**Description**

Create a dictionary type

**Usage**

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

**Arguments**

index_type	A DataType for the indices (default <a href="#">int32()</a> )
value_type	A DataType for the values (default <a href="#">utf8()</a> )
ordered	Is this an ordered dictionary (default FALSE)?

**Value**

A [DictionaryType](#)

**See Also**

[Other Arrow data types](#)

---

DictionaryType	<i>class DictionaryType</i>
----------------	-----------------------------

---

**Description**

class DictionaryType

**Methods**

TODO

---

Expression	<i>Arrow expressions</i>
------------	--------------------------

---

### 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.

---

FeatherReader	<i>FeatherReader class</i>
---------------	----------------------------

---

### Description

This class enables you to interact with Feather files. Create one to connect to a file or other Input-Stream, 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	<i>Field class</i>
-------	--------------------

---

### Description

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

### Usage

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

### Arguments

name	field name
type	logical type, instance of <a href="#">DataType</a>
metadata	currently ignored
nullable	TRUE if field is nullable

### Methods

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

### Examples

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

---

FileFormat	<i>Dataset file formats</i>
------------	-----------------------------

---

### 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`
- `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	<i>FileSystem entry info</i>
----------	------------------------------

---

**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	<i>file selector</i>
--------------	----------------------

---

**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")
- background\_writes: logical, whether OutputStream writes will be issued in the background, without blocking (default TRUE)

**Methods**

- `$GetFileInfo(x)`: `x` may be a [FileSelector](#) or a character vector of paths. Returns a list of [FileInfo](#)
- `$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`.

---

FileWriteOptions	<i>Format-specific write options</i>
------------------	--------------------------------------

---

**Description**

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

---

FixedWidthType	<i>class arrow::FixedWidthType</i>
----------------	------------------------------------

---

**Description**

class arrow::FixedWidthType

**Methods**

TODO

---

flight_connect	<i>Connect to a Flight server</i>
----------------	-----------------------------------

---

**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_get	<i>Get data from a Flight server</i>
------------	--------------------------------------

---

**Description**

Get data from a Flight server

**Usage**

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

**Arguments**

client	<code>pyarrow.flight.FlightClient</code> , as returned by <a href="#">flight_connect()</a>
path	string identifier under which data is stored

**Value**

A [Table](#)

---

flight_put	<i>Send data to a Flight server</i>
------------	-------------------------------------

---

**Description**

Send data to a Flight server

**Usage**

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

**Arguments**

client	pyarrow.flight.FlightClient, as returned by <a href="#">flight_connect()</a>
data	data.frame, <a href="#">RecordBatch</a> , or <a href="#">Table</a> 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.

**Value**

client, invisibly.

---

FragmentScanOptions	<i>Format-specific scan options</i>
---------------------	-------------------------------------

---

**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. `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`).

---

hive_partition	<i>Construct Hive partitioning</i>
----------------	------------------------------------

---

## 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 <a href="#">data types</a> , passed to <a href="#">schema()</a>
<code>null_fallback</code>	character to be used in place of missing values (NA or NULL) in partition columns. Default is <code>"__HIVE_DEFAULT_PARTITION__"</code> , which is what Hive uses.
<code>segment_encoding</code>	Decode partition segments after splitting paths. Default is <code>"uri"</code> (URI-decode segments). May also be <code>"none"</code> (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())
```



---

 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 BufferedReader 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 BufferedReader, 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 vignette("install", package = "arrow") 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_available()` to see if the package was configured with necessary C++ dependencies. `vignette("install", package = "arrow")` for more ways to tune installation on Linux.

---

install_pyarrow	<i>Install pyarrow for use with reticulate</i>
-----------------	--

---

**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 <code>reticulate::py_install()</code> .
nightly	logical: Should we install a development version of the package? Default is to use the official release version.
...	additional arguments passed to <code>reticulate::py_install()</code> .

---

io_thread_count	<i>Manage the global I/O thread pool in libarrow</i>
-----------------	--

---

**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
-------------	--

---

`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

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

### 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

### Examples

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

---

list_flights	<i>See available resources on a Flight server</i>
--------------	---

---

**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 <a href="#">flight_connect()</a>
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	<i>Load a Python Flight server</i>
--------------------	------------------------------------

---

**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	<i>Apply a function to a stream of RecordBatches</i>
-------------	--

---

### 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 aggregate on each chunk and pull the intermediate results into a `data.frame` for further aggregation, even if you couldn't fit the whole Dataset result in memory.

### Usage

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

### Arguments

X	A Dataset or <code>arrow_dplyr_query</code> object, as returned by the <code>dplyr</code> methods on Dataset.
FUN	A function or purrr-style lambda expression to apply to each batch
...	Additional arguments passed to FUN
.data.frame	logical: collect the resulting chunks into a single <code>data.frame</code> ? Default TRUE

### Details

This is experimental and not recommended for production use.

---

match_arrow	<i>match and %in% for Arrow objects</i>
-------------	---

---

### Description

`base::match()` is not a generic, so we can't just define Arrow methods for it. This function exposes 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` if 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

*class arrow::Message*

---

**Description**

class arrow::Message

**Methods**

TODO

---

MessageReader	<i>class arrow::MessageReader</i>
---------------	-----------------------------------

---

**Description**

class arrow::MessageReader

**Methods**

TODO

---

mmap_create	<i>Create a new read/write memory mapped file of a given size</i>
-------------	---

---

**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	<i>Open a memory mapped file</i>
-----------	----------------------------------

---

**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)



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(),
  unify_schemas = NULL,
  format = c("parquet", "arrow", "ipc", "feather", "csv", "tsv", "text"),
  ...
)
```

**Arguments**

- |              |   |
|--------------|---|
| sources      | <p>One of:</p> <ul style="list-style-type: none"> <li>• a string path or URI to a directory containing data files</li> <li>• a string path or URI to a single file</li> <li>• a character vector of paths or URIs to individual data files</li> <li>• a list of Dataset objects as created by this function</li> <li>• a list of DatasetFactory objects as created by <code>dataset_factory()</code>.</li> </ul> <p>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.</p>   |
| schema       | <p><a href="#">Schema</a> for the Dataset. If NULL (the default), the schema will be inferred from the data sources.</p>  |
| partitioning | <p>When sources is a directory path/URI, one of:</p> <ul style="list-style-type: none"> <li>• 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, <code>schema(year = int16(), month = int8())</code> would create partitions for file paths like "2019/01/file.parquet", "2019/02/file.parquet", etc.</li> <li>• 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)</li> <li>• a HivePartitioning or HivePartitioningFactory, as returned by <code>hive_partitioning()</code> which parses explicit or autodetected fields from Hive-style path segments</li> <li>• NULL for no partitioning</li> </ul> |

	The default is to autodetect Hive-style partitions. When sources is not a directory path/URI, partitioning is ignored.
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 <a href="#">FileFormat</a> 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: <ul style="list-style-type: none"> <li>• "parquet"</li> <li>• "ipc"/"arrow"/"feather", all aliases for each other; for Feather, note that only version 2 files are supported</li> <li>• "csv"/"text", aliases for the same thing (because comma is the default delimiter for text files)</li> <li>• "tsv", equivalent to passing format = "text", delimiter = "\t"</li> </ul> Default is "parquet", unless a delimiter is also specified, in which case it is assumed to be "text".
...	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.

### Value

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

### See Also

`vignette("dataset", package = "arrow")`

### Examples

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

data <- dplyr::group_by(mtcars, cyl)
write_dataset(data, tf)

# 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(data, 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()))

```

---

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`)
- ... 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
- `close` Close the writer. Note: does not close the sink. [arrow::io::OutputStream](#) has its own `close()` method.

---

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\_arrow

*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_arrow(file, ...)
```

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

## Arguments

<code>file</code>	A character file name or URI, raw vector, an Arrow input stream, or a <code>FileSystem</code> with path ( <code>SubTreeFileSystem</code> ). If a file name or URI, an Arrow <a href="#">InputStream</a> will be opened and closed when finished. If an input stream is provided, it will be left open.
<code>...</code>	extra parameters passed to <code>read_feather()</code> .
<code>as_data_frame</code>	Should the function return a <code>data.frame</code> (default) or an Arrow <a href="#">Table</a> ?

## Details

`read_arrow()`, a wrapper around `read_ipc_stream()` and `read_feather()`, is deprecated. You should explicitly choose the function that will read the desired IPC format (stream or file) since a file or `InputStream` may contain either.

## Value

A `data.frame` if `as_data_frame` is `TRUE` (the default), or an Arrow [Table](#) otherwise

## See Also

[read\\_feather\(\)](#) for writing IPC files. [RecordBatchReader](#) for a lower-level interface.



---

read_delim_arrow	<i>Read a CSV or other delimited file with Arrow</i>
------------------	--

---

### Description

These functions uses the Arrow C++ CSV reader to read into a `data.frame`. 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  
)
```

```
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_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, raw vector, an Arrow input stream, or a <code>FileSystem</code> with path ( <code>SubTreeFileSystem</code> ). If a file name, a memory-mapped Arrow <a href="#">InputStream</a> 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.
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 <code>TRUE</code> , the value <code>""""</code> represents a single quote, <code>\</code> .
escape_backslash	Does the file use backslashes to escape special characters? This is more general than <code>escape_double</code> as backslashes can be used to escape the delimiter character, the quote character, or to add special characters like <code>\n</code> .
schema	<a href="#">Schema</a> that describes the table. If provided, it will be used to satisfy both <code>col_names</code> and <code>col_types</code> .
col_names	If <code>TRUE</code> , the first row of the input will be used as the column names and will not be included in the data frame. If <code>FALSE</code> , column names will be generated by Arrow, starting with <code>"f0"</code> , <code>"f1"</code> , ..., <code>"fN"</code> . Alternatively, you can specify a character vector of column names.

col_types	A compact string representation of the column types, 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 <code>data.table::fread()</code> , or a <a href="#">tidy selection specification</a> of columns, as used in <code>dplyr::select()</code> .
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, <code>strings_can_be_null</code> .)
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 <a href="#">file reader options</a> . If given, this overrides any parsing options provided in other arguments (e.g. <code>delim</code> , <code>quote</code> , etc.).
convert_options	see <a href="#">file reader options</a>
read_options	see <a href="#">file reader options</a>
as_data_frame	Should the function return a <code>data.frame</code> (default) or an Arrow <a href="#">Table</a> ?
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: <ul style="list-style-type: none"> <li>• NULL: the default, which uses the ISO-8601 parser</li> <li>• a character vector of <a href="#">strptime</a> parse strings</li> <li>• a list of <a href="#">TimestampParser</a> objects</li> </ul>

## Details

`read_csv_arrow()` and `read_tsv_arrow()` are wrappers around `read_delim_arrow()` that specify a delimiter.

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 `data.frame`, 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()
- "t": time32()
- "\_": 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"))
```

---

read_feather	<i>Read a Feather file</i>
--------------	----------------------------

---

## 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. This function reads both the original, limited specification of the format and the version 2 specification, which is the Apache Arrow IPC file format.

## Usage

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

## Arguments

file	A character file name or URI, raw vector, an Arrow input stream, or a <code>FileSystem</code> with path ( <code>SubTreeFileSystem</code> ). If a file name or URI, an Arrow <a href="#">InputStream</a> 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 <code>data.table::fread()</code> , or a <a href="#">tidy selection specification</a> of columns, as used in <code>dplyr::select()</code> .
as_data_frame	Should the function return a <code>data.frame</code> (default) or an Arrow <a href="#">Table</a> ?
...	additional parameters, passed to <a href="#">make_readable_file()</a> .

## Value

A `data.frame` 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

```
tf <- tempfile()
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\_json\_arrow      *Read a JSON file*

---

## Description

Using [JsonTableReader](#)

## Usage

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

## Arguments

file	A character file name or URI, raw vector, an Arrow input stream, or a <code>FileSystem</code> with path ( <code>SubTreeFileSystem</code> ). If a file name, a memory-mapped Arrow <a href="#">InputStream</a> 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.
col_select	A character vector of column names to keep, as in the "select" argument to <code>data.table::fread()</code> , or a <a href="#">tidy selection specification</a> of columns, as used in <code>dplyr::select()</code> .
as_data_frame	Should the function return a <code>data.frame</code> (default) or an Arrow <a href="#">Table</a> ?
schema	<a href="#">Schema</a> that describes the table.
...	Additional options passed to <code>JsonTableReader\$create()</code>

## Value

A `data.frame`, 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)
df <- read_json_arrow(tf)
```

---

read_message	<i>Read a Message from a stream</i>
--------------	-------------------------------------

---

**Description**

Read a Message from a stream

**Usage**

```
read_message(stream)
```

**Arguments**

stream	an <code>InputStream</code>
--------	-----------------------------

---

read_parquet	<i>Read a Parquet file</i>
--------------	----------------------------

---

**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(),
  ...
)
```

**Arguments**

file	A character file name or URI, raw vector, an Arrow input stream, or a <code>FileSystem</code> with path ( <code>SubTreeFileSystem</code> ). If a file name or URI, an Arrow <a href="#">InputStream</a> 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 <code>data.table::fread()</code> , or a <a href="#">tidy selection specification</a> of columns, as used in <code>dplyr::select()</code> .
as_data_frame	Should the function return a <code>data.frame</code> (default) or an Arrow <a href="#">Table</a> ?
props	<a href="#">ParquetArrowReaderProperties</a>
...	Additional arguments passed to <code>ParquetFileReader\$create()</code>

**Value**

A [arrow::Table](#), or a `data.frame` if `as_data_frame` is `TRUE` (the default).

**Examples**

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

---

<code>read_schema</code>	<i>read a Schema from a stream</i>
--------------------------	------------------------------------

---

**Description**

read a Schema from a stream

**Usage**

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

**Arguments**

<code>stream</code>	a Message, InputStream, or Buffer
<code>...</code>	currently ignored

**Value**

A [Schema](#)

---

<code>RecordBatch</code>	<i>RecordBatch class</i>
--------------------------	--------------------------

---

**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](#).

**Usage**

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



**Arguments**

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

**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)`)
- `$RenameColumns(value)`: Set all column names (called by `names(batch) <-value`)
- `$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`) `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 Arrays

## 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")])
```

---

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)
- `$batches()`: Returns a list of `RecordBatches`
- `$read_table()`: Collects the reader's `RecordBatches` 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()
```

---

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 <code>S3FileSystem\$create()</code>

## 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("ursa-labs-taxi-data")
```

---

 Scalar

*Arrow scalars*


---

### Description

A Scalar holds a single value of an Arrow type.

### 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

### 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()
```

---

 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
- `use_async`: logical: should the async scanner (performs better on high-latency/highly parallel filesystems like S3) be used? Default FALSE
- `...`: Additional arguments, currently ignored

## Methods

ScannerBuilder has the following methods:

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

---

Schema

*Schema class*

---

## Description

A Schema is a list of [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.

## Usage

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

## Arguments

... named list containing [data types](#) or a list of [fields](#) containing the fields for the 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.

## 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.

## Examples

```
df <- data.frame(col1 = 2:4, col2 = c(0.1, 0.3, 0.5))
tab1 <- arrow_table(df)
tab1$metadata
tab2 <- arrow_table(df, schema = schema(col1 = int8(), col2 = float32()))
tab2$metadata
```



---

Table	<i>Table class</i>
-------	--------------------

---

### 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.

### Usage

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

### Arguments

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

### 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)`)
- `$RenameColumns(value)`: Set all column names (called by `names(tab) <-value`)
- `$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 (`tbl$metadata <- new_metadata`). All list elements are coerced to string. See `schema()` for more information.
- `$columns`: Returns a list of `ChunkedArrays`

## 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")])
```

---

to\_arrow

*Create an Arrow object from others*

---

## Description

This can be used in pipelines that pass data back and forth between Arrow and other processes (like DuckDB).

## Usage

```
to_arrow(.data)
```

## Arguments

`.data` the object to be converted

**Value**

an arrow\_dplyr\_query object, to be used in dplyr pipelines.

**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 = FALSE
)
```

**Arguments**

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

**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. Currently, this cleanup can, however, sometimes lead to hangs if tables are created and deleted in quick succession, hence the default value of `FALSE`

**Value**

A `tbl` of the new table in DuckDB

**Examples**

```
library(dplyr)

ds <- InMemoryDataset$create(mtcars)

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

---

type

*infer the arrow Array type from an R vector*

---

**Description**

infer the arrow Array type from an R vector

**Usage**

```
type(x)
```

**Arguments**

x                    an R vector

**Value**

an arrow logical type

## Examples

```
type(1:10)
type(1L:10L)
type(c(1, 1.5, 2))
type(c("A", "B", "C"))
type(mtcars)
type(Sys.Date())
```

---

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)
```

---

write_arrow	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_arrow(x, sink, ...)
```

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

**Arguments**

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

**Details**

`write_arrow()`, a wrapper around `write_ipc_stream()` and `write_feather()` with some non-standard behavior, is deprecated. You should explicitly choose the function that will write the desired IPC format (stream or file) since either can be written to a file or `OutputStream`.

**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_csv_arrow	<i>Write CSV file to disk</i>
-----------------	-------------------------------

---

**Description**

Write CSV file to disk

**Usage**

```
write_csv_arrow(x, sink, include_header = TRUE, batch_size = 1024L)
```

**Arguments**

<code>x</code>	<code>data.frame</code> , <a href="#">RecordBatch</a> , or <a href="#">Table</a>
<code>sink</code>	A string file path, URI, or <a href="#">OutputStream</a> , or path in a file system ( <code>SubTreeFileSystem</code> )
<code>include_header</code>	Whether to write an initial header line with column names
<code>batch_size</code>	Maximum number of rows processed at a time. Default is 1024.

**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"),
  partitioning = dplyr::group_vars(dataset),
  basename_template = paste0("part-{}.", as.character(format)),
  hive_style = TRUE,
  existing_data_behavior = c("overwrite", "error", "delete_matching"),
  ...
)
```

## Arguments

dataset	<a href="#">Dataset</a> , <a href="#">RecordBatch</a> , <a href="#">Table</a> , <a href="#">arrow_dplyr_query</a> , or <a href="#">data.frame</a> . If an <a href="#">arrow_dplyr_query</a> , the query will be evaluated and the result will be written. This means that you can <a href="#">select()</a> , <a href="#">filter()</a> , <a href="#">mutate()</a> , etc. to transform the data before it is written if you need to.
path	string path, URI, or <a href="#">SubTreeFileSystem</a> 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 <a href="#">FileFormat</a> )
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 <a href="#">group_by()</a> 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-{}." will yield "part-0.feather", ....
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.

...

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\_feather

*Write data in the Feather format*


---

## 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. This function writes both the original, limited specification of the format and the version 2 specification, which is the Apache Arrow IPC file format.

## Usage

```

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

```

## Arguments

x	data.frame, <a href="#">RecordBatch</a> , or <a href="#">Table</a>
sink	A string file path, URI, or <a href="#">OutputStream</a> , or path in a file system ( <a href="#">SubTreeFileSystem</a> )
version	integer Feather file version. Version 2 is the current. Version 1 is the more limited legacy format.

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 See <code>codec_is_available()</code> . 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

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

---

write_parquet	<i>Write Parquet file to disk</i>
---------------	-----------------------------------

---

### 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 = NULL,
  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,
    properties = NULL,
    arrow_properties = NULL
  )

```

## Arguments

x	data.frame, <a href="#">RecordBatch</a> , or <a href="#">Table</a>
sink	A string file path, URI, or <a href="#">OutputStream</a> , or path in a file system (SubTreeFileSystem)
chunk_size	chunk size in number of rows. If NULL, the total number of rows is used.
version	parquet version, "1.0" or "2.0". Default "1.0". Numeric values are coerced to character.
compression	compression algorithm. Default "snappy". See details.
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.
use_deprecated_int96_timestamps	Write timestamps to INT96 Parquet format. Default FALSE.
coerce_timestamps	Cast timestamps a particular resolution. Can be NULL, "ms" or "us". Default NULL (no casting)
allow_truncated_timestamps	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
properties	A <a href="#">ParquetWriterProperties</a> object, used instead of the options enumerated in this function's signature. Providing <code>properties</code> as an argument is deprecated; if you need to assemble <a href="#">ParquetWriterProperties</a> outside of <code>write_parquet()</code> , use <a href="#">ParquetFileWriter</a> instead.
arrow_properties	A <a href="#">ParquetArrowWriterProperties</a> object. Like <code>properties</code> , this argument is deprecated.

## 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 `vignette("dataset", package = "arrow")` 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.

### 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

- `$NewScan()`, [42](#)
- all Arrow functions, [9](#)
- Array, [11](#), [56](#)
- Array (array), [4](#)
- array, [4](#)
- ArrayData, [5](#), [6](#)
- Arrays, [10](#)
- arrow::io::MemoryMappedFile, [40](#)
- arrow::io::OutputStream, [46](#)
- arrow::Table, [56](#)
- arrow\_available, [6](#)
- arrow\_available(), [34](#)
- arrow\_info, [7](#)
- arrow\_info(), [15](#)
- arrow\_table (Table), [65](#)
- arrow\_with\_dataset (arrow\_available), [6](#)
- arrow\_with\_json (arrow\_available), [6](#)
- arrow\_with\_parquet (arrow\_available), [6](#)
- arrow\_with\_s3 (arrow\_available), [6](#)
  
- binary (data-type), [17](#)
- bool (data-type), [17](#)
- boolean (data-type), [17](#)
- Buffer, [33](#), [58](#)
- Buffer (buffer), [8](#)
- buffer, [8](#)
- BufferOutputStream (OutputStream), [43](#)
- BufferedReader (InputStream), [33](#)
  
- call\_function, [9](#)
- call\_function(), [36](#)
- chunked arrays, [65](#)
- chunked\_array (ChunkedArray), [10](#)
- ChunkedArray, [10](#), [45](#)
- Codec, [11](#), [12](#), [73](#)
- codec\_is\_available, [12](#)
- codec\_is\_available(), [11](#), [75](#), [77](#)
- compressed input and output streams, [11](#)
- CompressedInputStream (compression), [12](#)
- CompressedOutputStream (compression), [12](#)
- compression, [12](#)
- copy\_files, [13](#)
- cpu\_count, [13](#)
- create\_package\_with\_all\_dependencies, [14](#)
- CsvConvertOptions, [26](#), [32](#)
- CsvConvertOptions (CsvReadOptions), [15](#)
- CsvFileFormat (FileFormat), [25](#)
- CsvFragmentScanOptions, [26](#)
- CsvFragmentScanOptions (FragmentScanOptions), [31](#)
- CsvParseOptions, [26](#)
- CsvParseOptions (CsvReadOptions), [15](#)
- CsvReadOptions, [15](#), [17](#), [26](#), [32](#)
- CsvTableReader, [17](#), [51](#)
- CsvWriteOptions (CsvReadOptions), [15](#)
  
- data type, [4](#)
- data types, [32](#), [63](#), [64](#)
- data-type, [17](#)
- Dataset, [20](#), [21](#), [24](#), [42](#), [62](#), [63](#), [72](#), [76](#)
- dataset\_factory, [21](#)
- dataset\_factory(), [20](#), [41](#)
- DatasetFactory, [21](#)
- DatasetFactory (Dataset), [20](#)
- DataType, [22](#), [25](#)
- date32 (data-type), [17](#)
- date64 (data-type), [17](#)
- decimal (data-type), [17](#)
- dictionary, [23](#)
- dictionary(), [19](#)
- DictionaryArray (array), [4](#)
- DictionaryType, [23](#), [23](#)
- DirectoryPartitioning (Partitioning), [47](#)
- DirectoryPartitioningFactory (Partitioning), [47](#)
  
- Expression, [24](#), [63](#)

- FeatherReader, [24](#), [53](#)
- Field, [25](#), [63](#)
- field (Field), [25](#)
- fields, [56](#), [64](#)
- file reader options, [51](#)
- FileFormat, [20](#), [21](#), [25](#), [42](#), [72](#)
- FileInfo, [27](#), [29](#)
- FileOutputStream (OutputStream), [43](#)
- FileSelector, [20](#), [27](#), [29](#)
- FileSystem, [20](#), [21](#), [28](#)
- FileSystemDataset (Dataset), [20](#)
- FileSystemDatasetFactory, [47](#)
- FileSystemDatasetFactory (Dataset), [20](#)
- FileWriteOptions, [29](#)
- fixed\_size\_binary (data-type), [17](#)
- fixed\_size\_list\_of (data-type), [17](#)
- FixedSizeListArray (array), [4](#)
- FixedSizeListType (data-type), [17](#)
- FixedWidthType, [29](#)
- flight\_connect, [30](#)
- flight\_connect(), [30](#), [31](#), [37](#)
- flight\_get, [30](#)
- flight\_path\_exists (list\_flights), [37](#)
- flight\_put, [31](#)
- float (data-type), [17](#)
- float16 (data-type), [17](#)
- float32 (data-type), [17](#)
- float64 (data-type), [17](#)
- FragmentScanOptions, [26](#), [31](#)
  
- halffloat (data-type), [17](#)
- hive\_partition, [32](#)
- hive\_partition(), [22](#), [41](#), [73](#)
- hive\_partition(...), [48](#)
- HivePartitioning, [32](#)
- HivePartitioning (Partitioning), [47](#)
- HivePartitioningFactory (Partitioning), [47](#)
  
- InMemoryDataset (Dataset), [20](#)
- input file, [29](#)
- input stream, [29](#)
- InputStream, [12](#), [17](#), [33](#), [48](#), [50](#), [53–55](#), [58](#)
- install\_arrow, [33](#)
- install\_pyarrow, [35](#)
- int16 (data-type), [17](#)
- int32 (data-type), [17](#)
- int32(), [23](#)
- int64 (data-type), [17](#)
  
- int8 (data-type), [17](#)
- io\_thread\_count, [35](#)
- IpcFileFormat (FileFormat), [25](#)
- is\_in (match\_arrow), [38](#)
  
- JsonParseOptions (CsvReadOptions), [15](#)
- JsonReadOptions (CsvReadOptions), [15](#)
- JsonTableReader, [54](#)
- JsonTableReader (CsvTableReader), [17](#)
  
- large\_binary (data-type), [17](#)
- large\_list\_of (data-type), [17](#)
- large\_utf8 (data-type), [17](#)
- LargeListArray (array), [4](#)
- list\_compute\_functions, [36](#)
- list\_flights, [37](#)
- list\_of (data-type), [17](#)
- ListArray (array), [4](#)
- load\_flight\_server, [37](#)
- LocalFileSystem (FileSystem), [28](#)
  
- make\_readable\_file(), [53](#)
- map\_batches, [38](#)
- match\_arrow, [38](#)
- MemoryMappedFile (InputStream), [33](#)
- Message, [39](#)
- MessageReader, [40](#)
- mmap\_create, [40](#)
- mmap\_open, [40](#)
- mmap\_open(), [33](#)
  
- null (data-type), [17](#)
  
- open\_dataset, [41](#)
- open\_dataset(), [20–22](#)
- Other Arrow data types, [23](#)
- output stream, [29](#)
- OutputStream, [12](#), [43](#), [66](#), [70](#), [71](#), [74–76](#)
  
- ParquetArrowReaderProperties, [44](#), [45](#), [55](#)
- ParquetFileFormat (FileFormat), [25](#)
- ParquetFileReader, [44](#), [44](#)
- ParquetFileWriter, [46](#), [46](#)
- ParquetFragmentScanOptions (FragmentScanOptions), [31](#)
- ParquetWriterProperties, [46](#), [46](#)
- Partitioning, [47](#)
  
- RandomAccessFile, [58](#)
- RandomAccessFile (InputStream), [33](#)



- read.csv(), [16](#)
- read\_arrow, [48](#)
- read\_csv\_arrow (read\_delim\_arrow), [49](#)
- read\_csv\_arrow(), [15](#), [17](#), [22](#), [26](#)
- read\_delim\_arrow, [49](#)
- read\_feather, [53](#)
- read\_feather(), [24](#), [48](#), [59](#)
- read\_ipc\_stream (read\_arrow), [48](#)
- read\_ipc\_stream(), [59](#)
- read\_json\_arrow, [54](#)
- read\_json\_arrow(), [15](#), [17](#)
- read\_message, [55](#)
- read\_parquet, [55](#)
- read\_schema, [56](#)
- read\_tsv\_arrow (read\_delim\_arrow), [49](#)
- ReadableFile (InputStream), [33](#)
- record batches, [65](#)
- record\_batch (RecordBatch), [56](#)
- RecordBatch, [31](#), [56](#), [60](#), [70–72](#), [74](#), [76](#), [78](#)
- RecordBatchFileReader
  - (RecordBatchReader), [58](#)
- RecordBatchFileWriter
  - (RecordBatchWriter), [59](#)
- RecordBatchReader, [48](#), [53](#), [58](#)
- RecordBatchStreamReader
  - (RecordBatchReader), [58](#)
- RecordBatchStreamWriter
  - (RecordBatchWriter), [59](#)
- RecordBatchWriter, [59](#), [71](#), [75](#)
  
- s3\_bucket, [61](#)
- S3FileSystem (FileSystem), [28](#)
- Scalar, [62](#)
- Scanner, [24](#), [62](#)
- ScannerBuilder, [21](#)
- ScannerBuilder (Scanner), [62](#)
- Schema, [20](#), [21](#), [41](#), [46–48](#), [50](#), [52](#), [54](#), [56–58](#), [60](#), [63](#), [63](#), [65](#), [69](#), [75](#)
- schema (Schema), [63](#)
- schema(), [17](#), [32](#)
- Schemas, [25](#)
- set\_cpu\_count (cpu\_count), [13](#)
- set\_io\_thread\_count (io\_thread\_count), [35](#)
  
- string (data-type), [17](#)
- strptime, [16](#), [51](#)
- strptime(), [16](#)
- struct (data-type), [17](#)
- StructArray (array), [4](#)
- StructScalar (array), [4](#)
- SubTreeFileSystem (FileSystem), [28](#)
  
- Table, [10](#), [30](#), [31](#), [46](#), [48](#), [51](#), [53–55](#), [58](#), [60](#), [63](#), [65](#), [70–72](#), [74](#), [76](#), [78](#)
- tidy selection specification, [51](#), [53–55](#)
- time32 (data-type), [17](#)
- time64 (data-type), [17](#)
- timestamp (data-type), [17](#)
- TimestampParser, [16](#), [51](#)
- TimestampParser (CsvReadOptions), [15](#)
- to\_arrow, [66](#)
- to\_duckdb, [67](#)
- type, [68](#)
  
- uint16 (data-type), [17](#)
- uint32 (data-type), [17](#)
- uint64 (data-type), [17](#)
- uint8 (data-type), [17](#)
- unify\_schemas, [69](#)
- UnionDataset (Dataset), [20](#)
- utf8 (data-type), [17](#)
- utf8(), [23](#)
  
- value\_counts, [70](#)
  
- write\_arrow, [70](#)
- write\_csv\_arrow, [71](#)
- write\_dataset, [72](#)
- write\_feather, [74](#)
- write\_feather(), [60](#), [70](#), [71](#), [77](#)
- write\_ipc\_stream (write\_arrow), [70](#)
- write\_ipc\_stream(), [60](#), [77](#)
- write\_parquet, [46](#), [47](#), [75](#)
- write\_parquet(), [73](#)
- write\_to\_raw, [77](#)
- write\_to\_raw(), [60](#), [71](#)