

# Package ‘arrow’

August 28, 2020

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

**Version** 1.0.1

**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 (>= 3.1)

**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

**LazyData** true

**SystemRequirements** C++11

**Biarch** true

**LinkingTo** Rcpp (>= 1.0.1)

**Imports** assertthat, bit64, methods, purrr, R6, Rcpp (>= 1.0.1), rlang, tidyselect, utils, vctrs

**RoxygenNote** 7.1.1

**VignetteBuilder** knitr

**Suggests** decor, dplyr, hms, knitr, lubridate, pkgload, reticulate, rmarkdown, testthat, tibble

**Collate** 'enums.R' 'arrow-package.R' 'type.R' 'array-data.R' 'array.R' 'arrowExports.R' 'buffer.R' 'chunked-array.R' 'io.R' 'compression.R' 'scalar.R' 'compute.R' 'config.R' 'csv.R' 'dataset.R' 'deprecated.R' 'dictionary.R' 'record-batch.R' 'table.R' 'expression.R' 'dplyr.R' 'feather.R' 'field.R' 'filesystem.R' 'install-arrow.R' 'ipc\_stream.R' 'json.R' 'list.R' 'memory-pool.R' 'message.R' 'parquet.R' 'python.R'

'record-batch-reader.R' 'record-batch-writer.R'  
 'reexports-bit64.R' 'reexports-tidysselect.R' 'schema.R'  
 'struct.R' 'util.R'

**NeedsCompilation** yes

**Author** Romain François [aut] (<<https://orcid.org/0000-0002-2444-4226>>),  
 Jeroen Ooms [aut],  
 Neal Richardson [aut, cre],  
 Javier Luraschi [ctb],  
 Jeffrey Wong [ctb],  
 Apache Arrow [aut, cph]

**Maintainer** Neal Richardson <[neal@ursalabs.org](mailto:neal@ursalabs.org)>

**Repository** CRAN

**Date/Publication** 2020-08-28 12:20:12 UTC

## R topics documented:

array . . . . .	3
ArrayData . . . . .	5
arrow_available . . . . .	5
buffer . . . . .	6
cast_options . . . . .	6
ChunkedArray . . . . .	7
Codec . . . . .	8
codec_is_available . . . . .	8
compression . . . . .	9
cpu_count . . . . .	9
CsvReadOptions . . . . .	10
CsvTableReader . . . . .	11
data-type . . . . .	11
Dataset . . . . .	14
dataset_factory . . . . .	15
DataType . . . . .	16
default_memory_pool . . . . .	17
dictionary . . . . .	17
DictionaryType . . . . .	18
Expression . . . . .	18
FeatherReader . . . . .	18
Field . . . . .	19
FileFormat . . . . .	20
FileInfo . . . . .	20
FileSelector . . . . .	21
FileSystem . . . . .	21
FixedWidthType . . . . .	22
hive_partition . . . . .	22
InputStream . . . . .	23
install_arrow . . . . .	24

install_pyarrow . . . . .	25
map_batches . . . . .	25
MemoryPool . . . . .	26
Message . . . . .	26
MessageReader . . . . .	26
mmap_create . . . . .	27
mmap_open . . . . .	27
open_dataset . . . . .	28
OutputStream . . . . .	29
ParquetFileReader . . . . .	30
ParquetFileWriter . . . . .	30
ParquetReaderProperties . . . . .	31
ParquetWriterProperties . . . . .	31
Partitioning . . . . .	32
read_arrow . . . . .	33
read_delim_arrow . . . . .	34
read_feather . . . . .	37
read_json_arrow . . . . .	38
read_message . . . . .	39
read_parquet . . . . .	39
read_schema . . . . .	40
RecordBatch . . . . .	40
RecordBatchReader . . . . .	42
RecordBatchWriter . . . . .	43
Scalar . . . . .	45
Scanner . . . . .	45
Schema . . . . .	46
Table . . . . .	47
type . . . . .	49
unify_schemas . . . . .	49
write_arrow . . . . .	50
write_feather . . . . .	50
write_parquet . . . . .	52
write_to_raw . . . . .	54
<b>Index</b>	<b>55</b>

---

array

*Arrow Arrays*


---

### 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
- `$ApproxEquals(other)` :
- `$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`) `i`.
- `$Filter(i, keep_na = TRUE)`: return an `Array` with values at positions where logical vector (or Arrow boolean `Array`) `i` is TRUE.
- `$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  $O(\text{length})$

---

 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 this function, but it's here in case it helps for development purposes.

### Usage

```
arrow_available()
```

### Value

TRUE or FALSE depending on whether the package was installed with the Arrow C++ library. If FALSE, you'll need to install the C++ library and then reinstall the R package. See [install\\_arrow\(\)](#) for help.

### Examples

```
arrow_available()
```

---

buffer	<i>Buffer class</i>
--------	---------------------

---

**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() :
- \$ZeroPadding() :
- \$size() :
- \$capacity() :

---

cast_options	<i>Cast options</i>
--------------	---------------------

---

**Description**

Cast options

**Usage**

```
cast_options(
  safe = TRUE,
  allow_int_overflow = !safe,
  allow_time_truncate = !safe,
  allow_float_truncate = !safe
)
```

**Arguments**

safe	enforce safe conversion
allow_int_overflow	allow int conversion, !safe by default
allow_time_truncate	allow time truncate, !safe by default
allow_float_truncate	allow float truncate, !safe by default

---

 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.
- `$cast(target_type, safe = TRUE, options = cast_options(safe))`: Alter the data in the array to change its type.
- `$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](#)

---

Codec	<i>Compression Codec class</i>
-------	--------------------------------

---

### 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.

---

<code>codec_is_available</code>	<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

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



**Value**

Logical: is type available?

---

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

---

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

---

CsvReadOptions      *File reader options*

---

### Description

CsvReadOptions, CsvParseOptions, CsvConvertOptions, JsonReadOptions, and JsonParseOptions 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)

**Methods**

These classes have no implemented methods. They are containers for the options.

---

CsvTableReader	<i>Arrow CSV and JSON table reader classes</i>
----------------	--

---

**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 A character path to a local file, or an Arrow input stream
- convert\_options (CSV only), parse\_options, read\_options: see [CsvReadOptions](#)
- ... additional parameters.

**Methods**

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

---

data-type	<i>Apache Arrow data types</i>
-----------	--------------------------------

---

**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)
list_of(type)
```

```

large_list_of(type)

fixed_size_list_of(type, list_size)

struct(...)

```

### Arguments

byte_width	byte width for FixedSizeBinary type.
unit	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.
timezone	For <code>timestamp()</code> , an optional time zone string.
precision	For <code>decimal()</code> , precision
scale	For <code>decimal()</code> , scale
type	For <code>list_of()</code> , a data type to make a list-of-type
list_size	list size for FixedSizeList type.
...	For <code>struct()</code> , a named list of types to define the struct columns

### Details

A few functions have aliases:

- `utf8()` and `string()`
- `float16()` and `halffloat()`
- `float32()` and `float()`
- `bool()` and `boolean()`
- Called from `schema()` or `struct()`, `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.

### 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: A [FileSelector](#)
- 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 [DatasetFactories](#). 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 file <code>x</code> containing data files, or a list of <code>DatasetFactory</code> objects whose datasets should be grouped. If this argument is specified it will be used to construct a <code>UnionDatasetFactory</code> and other arguments will be ignored.
<code>filesystem</code>	A <a href="#">FileSystem</a> object; if omitted, the <code>FileSystem</code> 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: <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 <code>format = "text"</code>, <code>delimiter = "\t"</code></li> </ul> Default is "parquet", unless a <code>delimiter</code> is also specified, in which case it is assumed to be "text".
<code>partitioning</code>	One of

- A Schema, in which case the file paths relative to sources will be parsed, and path segments will be matched with the schema fields. For example, `schema(year = int16(), month = int8())` would create partitions for file paths like "2019/01/file.parquet", "2019/02/file.parquet", etc.
  - A character vector that defines the field names corresponding to those path segments (that is, you're providing the names that would correspond to a Schema but the types will be autodetected)
  - A `HivePartitioning` or `HivePartitioningFactory`, as returned by `hive_partition()` which parses explicit or autodetected fields from Hive-style path segments
  - NULL for no partitioning
- ... Additional format-specific options, passed to `FileFormat$create()`. For CSV options, note that you can specify them either with the Arrow C++ library naming ("delimiter", "quoting", etc.) or the readr-style naming used in `read_csv_arrow()` ("delim", "quote", etc.)

### 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



---

default_memory_pool	<i>default</i> <a href="#">arrow::MemoryPool</a>
---------------------	--

---

**Description**

default [arrow::MemoryPool](#)

**Usage**

default\_memory\_pool()

**Value**

the default [arrow::MemoryPool](#)

---

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\$compare(OP, e1, e2) takes two Expression operands, constructing an Expression which will evaluate these operands then compare them with the relation specified by OP (e.g. "==", "!=", ">", etc.) For example, to filter down to rows where the column named "alpha" is less than 5: Expression\$compare("<", Expression\$field\_ref("alpha"), Expression\$scalar(5))

Expression\$and(e1, e2), Expression\$or(e1, e2), and Expression\$not(e1) construct an Expression combining their arguments with Boolean operators.

Expression\$is\_valid(x) is essentially (an inversion of) is.na() for Expressions.

Expression\$in\_(x, set) evaluates x and returns whether or not it is a member of the set.

---

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 arguments:

- `file` an Arrow file connection object inheriting from `RandomAccessFile`.
- `mmap Logical`: whether to memory-map the file (default `TRUE`)
- ... Additional arguments, currently ignored

**Methods**

- `$Read(columns)`: Returns a `Table` of the selected columns, a vector of integer indices
- `$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)
```

**Arguments**

<code>name</code>	field name
<code>type</code>	logical type, instance of <a href="#">DataType</a>
<code>metadata</code>	currently ignored

**Methods**

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

**Examples**

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

---

 FileFormat

*Dataset file formats*


---

### Description

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

### Factory

FileFormat\$create() takes the following arguments:

- format: A string identifier of the file format. Currently supported values:
  - "parquet"
  - "ipc"/"arrow"/"feather", all aliases for each other; for Feather, note that only version 2 files are supported
  - "csv"/"text", aliases for the same thing (because comma is the default delimiter for text files)
  - "tsv", equivalent to passing format = "text", delimiter = "\t"
- ...: Additional format-specific options
  - ‘format = "parquet"’:
    - 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.
    - dict\_columns: Names of columns which should be read as dictionaries.

format = "text": see [CsvReadOptions](#). 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.)

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

---

 FileInfo

*FileSystem entry info*


---

### Description

FileSystem entry info

### Methods

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

**Active bindings**

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

---

FileSelector	<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	<i>FileSystem classes</i>
------------	---------------------------

---

**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**

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

- no argument is needed for instantiating a LocalFileSystem
- base\_path and base\_fs for instantiating a SubTreeFileSystem

**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.

---

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

---

**Description**

class arrow::FixedWidthType

**Methods**

TODO

---

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(...)`

**Arguments**

... named list of [data types](#), passed to [schema\(\)](#)

**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	<i>Install or upgrade the Arrow library</i>
---------------	---

---

### 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),
  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 <code>vignette("install", package = "arrow")</code> 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.
minimal	logical: If building from source, should we build without optional dependencies (compression libraries, for example)? Default is FALSE.
repos	character vector of base URLs of the repositories to install from (passed to <code>install.packages()</code> )
...	Additional arguments passed to <code>install.packages()</code>

### 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> .

---

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 dplyr 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.

---

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

---

**Description**

class arrow::MemoryPool

**Methods**

TODO

---

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

---

**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	<i>Open a multi-file dataset</i>
--------------	----------------------------------

---

### 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,
  ...
)
```

### Arguments

sources	<p>Either:</p> <ul style="list-style-type: none"> <li>• a string path to a directory containing 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>
schema	<a href="#">Schema</a> for the dataset. If NULL (the default), the schema will be inferred from the data sources.
partitioning	<p>When sources is a file path, 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_partition()</code> which parses explicit or autodetected fields from Hive-style path segments</li> <li>• NULL for no partitioning</li> </ul> <p>The default is to autodetect Hive-style partitions.</p>
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

file path (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).

... additional arguments passed to `dataset_factory()` when sources is a file path, 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")`

---

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.

---

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 [ParquetReaderProperties](#)
- mmap Logical: whether to memory-map the file (default TRUE)
- ... Additional arguments, currently ignored

### Methods

- `$ReadTable(col_select)`: get an `arrow::Table` from the file, possibly with columns filtered by a character vector of column names or a `tidyselect` specification.
- `$GetSchema()`: get the `arrow::Schema` of the data in the file

### 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$ReadTable(starts_with("c"))
  tab$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` or a string which is interpreted as a file path
- properties An instance of [ParquetWriterProperties](#)
- arrow\_properties An instance of `ParquetArrowWriterProperties`

---

ParquetReaderProperties

*ParquetReaderProperties class*

---

**Description**

This class holds settings to control how a Parquet file is read by [ParquetFileReader](#).

**Factory**

The `ParquetReaderProperties#create()` factory method instantiates the object and takes 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)`

---

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](#)

---

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.



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.

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	<i>Read Arrow IPC stream format</i>
------------	-------------------------------------

---

## 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

file	A character file name, raw vector, or an Arrow input stream. If a file name, a memory-mapped Arrow <a href="#">InputStream</a> will be opened and closed when finished. If an input stream is provided, it will be left open.
...	extra parameters passed to <code>read_feather()</code> .
as_data_frame	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.

---

<code>read_delim_arrow</code>	<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,
  col_names = TRUE,
  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
)
```

```
read_csv_arrow(
  file,
  quote = "\"",
  escape_double = TRUE,
  escape_backslash = FALSE,
  col_names = TRUE,
  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
  )

read_tsv_arrow(
  file,
  quote = "\"",
  escape_double = TRUE,
  escape_backslash = FALSE,
  col_names = TRUE,
  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
)

```

## Arguments

file	A character file name, raw vector, or an Arrow input stream. 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 TRUE, the value """" represents a single quote, \".
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> .
col_names	If TRUE, the first row of the input will be used as the column names and will not be included in the data frame. If FALSE, column names will be generated by Arrow, starting with "f0", "f1", ..., "fN". Alternatively, you can specify a character vector of column names.
col_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, strings_can_be_null.)
skip_empty_rows	Should blank rows be ignored altogether? If TRUE, blank rows will not be represented at all. If FALSE, they will be filled with missings.
skip	Number of lines to skip before reading data.
parse_options	see <a href="#">file reader options</a> . If given, this overrides any parsing options provided in other arguments (e.g. delim, quote, 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 data.frame (default) or an Arrow <a href="#">Table</a> ?

## 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.

## 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, raw vector, or an Arrow input stream. If a file name, a memory-mapped 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="#">FeatherReader\$create()</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	<i>Read a JSON file</i>
-----------------	-------------------------

---

## Description

Using [JsonTableReader](#)

## Usage

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

## Arguments

file	A character file name, raw vector, or an Arrow input stream. 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> ?
...	Additional options, passed to <code>json_table_reader()</code>

## Value

A `data.frame`, or an `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 InputStream
--------	----------------

---

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 = ParquetReaderProperties$create(),
  ...
)
```

**Arguments**

file	A character file name, raw vector, or an Arrow input stream. If a file name, a memory-mapped 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="#">ParquetReaderProperties</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)`)
- `$GetColumnByName(name)`: Extract an `Array` by string name
- `$RemoveColumn(i)`: Drops a column from the batch by integer position
- `$select(spec)`: Return a new record batch with a selection of columns. This supports the usual `character`, `numeric`, and `logical` selection methods as well as "tidy select" expressions.
- `$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`.
- `$serialize()`: Returns a raw vector suitable for interprocess communication
- `$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.
- `$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 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()

```

---

 Scalar

*Arrow scalars*


---

### Description

A Scalar holds a single value of an Arrow type.

---

 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
- `filter`: A Expression to filter the scanned rows by, or `TRUE` (default) to keep all rows.
- `use_threads`: logical: should scanning use multithreading? Default `TRUE`
- `...`: Additional arguments, currently ignored

## Methods

ScannerBuilder has the following methods:

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

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

---

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 of [data types](#)

## 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.

**Examples**

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

---

Table

*Table class*


---

**Description**

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

**Factory**

The `Table$create()` function takes the following arguments:

- `...` arrays, chunked arrays, or R vectors, with names; alternatively, an unnamed series of [record batches](#) may also be provided, which will be stacked as rows in the table.
- `schema` a [Schema](#), or `NULL` (the default) to infer the schema from the data in `...`

**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)`)
- `$getColumnByName(name)`: Extract a `ChunkedArray` by string name
- `$field(i)`: Extract a `Field` from the table schema by integer position
- `$select(spec)`: Return a new table with a selection of columns. This supports the usual character, numeric, and logical selection methods as well as "tidy select" expressions.
- `$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`.
- `$serialize(output_stream, ...)`: Write the table to the given [OutputStream](#)
- `$cast(target_schema, safe = TRUE, options = cast_options(safe))`: Alter the schema of the record batch.

There are also some active bindings:

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

## Examples

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



---

type	<i>infer the arrow Array type from an R vector</i>
------	--

---

**Description**

infer the arrow Array type from an R vector

**Usage**

```
type(x)
```

**Arguments**

x                    an R vector

**Value**

an arrow logical type

---

unify_schemas	<i>Combine and harmonize schemas</i>
---------------	--------------------------------------

---

**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

**Examples**

```
## Not run:  
a <- schema(b = double(), c = bool())  
z <- schema(b = double(), k = utf8())  
unify_schemas(a, z)  
  
## End(Not run)
```

---

write_arrow	<i>Write Arrow IPC stream format</i>
-------------	--------------------------------------

---

### 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, <a href="#">RecordBatch</a> , or <a href="#">Table</a>
sink	A string file path or <a href="#">OutputStream</a>
...	extra parameters passed to <code>write_feather()</code> .

### 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.

---

write_feather	<i>Write data in the Feather format</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 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 or <a href="#">OutputStream</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 <a href="#">codec_is_available()</a> . 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.

## 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 = NULL,
  compression_level = NULL,
  use_dictionary = NULL,
  write_statistics = NULL,
  data_page_size = NULL,
  properties = ParquetWriterProperties$create(x, version = version, compression =
    compression, compression_level = compression_level, use_dictionary = use_dictionary,
    write_statistics = write_statistics, data_page_size = data_page_size),
  use_deprecated_int96_timestamps = FALSE,
  coerce_timestamps = NULL,
  allow_truncated_timestamps = FALSE,

  arrow_properties = ParquetArrowWriterProperties$create(use_deprecated_int96_timestamps
    = use_deprecated_int96_timestamps, coerce_timestamps = coerce_timestamps,
    allow_truncated_timestamps = allow_truncated_timestamps)
)
```

## Arguments

x	An <code>arrow::Table</code> , or an object convertible to it.
sink	an <code>arrow::io::OutputStream</code> or a string which is interpreted as a file path
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

<code>data_page_size</code>	Set a target threshold for the approximate encoded size of data pages within a column chunk (in bytes). Default 1 MiB.
<code>properties</code>	properties for parquet writer, derived from arguments <code>version</code> , <code>compression</code> , <code>compression_level</code> , <code>use_dictionary</code> , <code>write_statistics</code> and <code>data_page_size</code> . You should not specify any of these arguments if you also provide a <code>properties</code> argument, as they will be ignored.
<code>use_deprecated_int96_timestamps</code>	Write timestamps to INT96 Parquet format. Default FALSE.
<code>coerce_timestamps</code>	Cast timestamps a particular resolution. Can be NULL, "ms" or "us". Default NULL (no casting)
<code>allow_truncated_timestamps</code>	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
<code>arrow_properties</code>	arrow specific writer properties, derived from arguments <code>use_deprecated_int96_timestamps</code> , <code>coerce_timestamps</code> and <code>allow_truncated_timestamps</code> You should not specify any of these arguments if you also provide a <code>properties</code> argument, as they will be ignored.

## 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

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, <a href="#">RecordBatch</a> , or <a href="#">Table</a>
format	one of c("stream", "file"), indicating the IPC format to use

### Value

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

# Index

- [\\$NewScan\(\)](#), 29
- [Array](#), 8, 40
- [Array \(array\)](#), 3
- [array](#), 3
- [ArrayData](#), 4, 5
- [Arrays](#), 7
- [arrow::io::MemoryMappedFile](#), 27
- [arrow::io::OutputStream](#), 31, 52
- [arrow::MemoryPool](#), 17
- [arrow::Table](#), 40, 52
- [arrow\\_available](#), 5
- [arrow\\_available\(\)](#), 24
  
- [binary \(data-type\)](#), 11
- [bool \(data-type\)](#), 11
- [boolean \(data-type\)](#), 11
- [Buffer](#), 23, 42
- [Buffer \(buffer\)](#), 6
- [buffer](#), 6
- [BufferOutputStream \(OutputStream\)](#), 29
- [BufferedReader \(InputStream\)](#), 23
  
- [cast\\_options](#), 6
- [chunked arrays](#), 47
- [chunked\\_array \(ChunkedArray\)](#), 7
- [ChunkedArray](#), 7
- [Codec](#), 8, 9
- [codec\\_is\\_available](#), 8
- [codec\\_is\\_available\(\)](#), 8, 51, 53
- [compressed input and output streams](#), 8
- [CompressedInputStream \(compression\)](#), 9
- [CompressedOutputStream \(compression\)](#), 9
- [compression](#), 9
- [cpu\\_count](#), 9
- [CsvConvertOptions \(CsvReadOptions\)](#), 10
- [CsvFileFormat \(FileFormat\)](#), 20
- [CsvParseOptions \(CsvReadOptions\)](#), 10
- [CsvReadOptions](#), 10, 11, 20
- [CsvTableReader](#), 11, 36
  
- [data type](#), 4
- [data types](#), 23, 46
- [data-type](#), 11
- [Dataset](#), 14, 15, 18, 29, 45, 46
- [dataset\\_factory](#), 15
- [dataset\\_factory\(\)](#), 14, 28
- [DatasetFactory](#), 15
- [DatasetFactory \(Dataset\)](#), 14
- [DataType](#), 16, 19
- [date32 \(data-type\)](#), 11
- [date64 \(data-type\)](#), 11
- [decimal \(data-type\)](#), 11
- [default\\_memory\\_pool](#), 17
- [dictionary](#), 17
- [dictionary\(\)](#), 13
- [DictionaryArray \(array\)](#), 3
- [DictionaryType](#), 17, 18
- [DirectoryPartitioning \(Partitioning\)](#), 32
- [DirectoryPartitioningFactory \(Partitioning\)](#), 32
  
- [Expression](#), 18, 46
  
- [FeatherReader](#), 18, 37
- [FeatherReader\\$create\(\)](#), 37
- [Field](#), 19, 46
- [field \(Field\)](#), 19
- [fields](#), 40
- [file reader options](#), 36
- [FileFormat](#), 14, 15, 20
- [FileInfo](#), 20, 22
- [FileOutputStream \(OutputStream\)](#), 29
- [FileSelector](#), 14, 21, 22
- [FileSystem](#), 14, 15, 21
- [FileSystemDataset \(Dataset\)](#), 14
- [FileSystemDatasetFactory](#), 32
- [FileSystemDatasetFactory \(Dataset\)](#), 14
- [fixed\\_size\\_binary \(data-type\)](#), 11
- [fixed\\_size\\_list\\_of \(data-type\)](#), 11
- [FixedSizeListArray \(array\)](#), 3

- FixedSizeListType (data-type), 11
- FixedWidthType, 22
- float (data-type), 11
- float16 (data-type), 11
- float32 (data-type), 11
- float64 (data-type), 11
  
- halffloat (data-type), 11
- hive\_partition, 22
- hive\_partition(), 16, 28
- hive\_partition(...), 33
- HivePartitioning, 23
- HivePartitioning (Partitioning), 32
- HivePartitioningFactory (Partitioning), 32
  
- input file, 22
- input stream, 22
- InputStream, 9, 23, 33, 35, 37–39, 42
- install\_arrow, 24
- install\_arrow(), 5
- install\_pyarrow, 25
- int16 (data-type), 11
- int32 (data-type), 11
- int32(), 17
- int64 (data-type), 11
- int8 (data-type), 11
- IpcFileFormat (FileFormat), 20
  
- JsonParseOptions (CsvReadOptions), 10
- JsonReadOptions (CsvReadOptions), 10
- JsonTableReader, 38
- JsonTableReader (CsvTableReader), 11
  
- large\_binary (data-type), 11
- large\_list\_of (data-type), 11
- large\_utf8 (data-type), 11
- LargeListArray (array), 3
- list\_of (data-type), 11
- ListArray (array), 3
- LocalFileSystem (FileSystem), 21
  
- map\_batches, 25
- MemoryMappedFile (InputStream), 23
- MemoryPool, 26
- Message, 26
- MessageReader, 26
- mmap\_create, 27
- mmap\_open, 27
  
- mmap\_open(), 23
  
- null (data-type), 11
  
- open\_dataset, 28
- open\_dataset(), 14–16
- Other Arrow data types, 17
- output stream, 22
- OutputStream, 9, 29, 48, 50, 51
  
- ParquetFileFormat (FileFormat), 20
- ParquetFileReader, 30, 31
- ParquetFileWriter, 30, 31
- ParquetReaderProperties, 30, 31, 39
- ParquetWriterProperties, 31, 31
- Partitioning, 32
  
- RandomAccessFile, 42
- RandomAccessFile (InputStream), 23
- read.csv(), 10
- read\_arrow, 33
- read\_csv\_arrow (read\_delim\_arrow), 34
- read\_csv\_arrow(), 10, 11, 16, 20
- read\_delim\_arrow, 34
- read\_feather, 37
- read\_feather(), 18, 33, 34, 42
- read\_ipc\_stream (read\_arrow), 33
- read\_ipc\_stream(), 42
- read\_json\_arrow, 38
- read\_json\_arrow(), 10, 11
- read\_message, 39
- read\_parquet, 39
- read\_schema, 40
- read\_tsv\_arrow (read\_delim\_arrow), 34
- ReadableFile (InputStream), 23
- record batches, 47
- record\_batch (RecordBatch), 40
- RecordBatch, 40, 44, 50, 51, 54
- RecordBatchFileReader  
(RecordBatchReader), 42
- RecordBatchFileWriter  
(RecordBatchWriter), 43
- RecordBatchReader, 34, 37, 42
- RecordBatchStreamReader  
(RecordBatchReader), 42
- RecordBatchStreamWriter  
(RecordBatchWriter), 43
- RecordBatchWriter, 43, 50, 51
  
- S3FileSystem (FileSystem), 21



Scalar, [45](#)  
Scanner, [18](#), [45](#)  
ScannerBuilder, [14](#)  
ScannerBuilder (Scanner), [45](#)  
Schema, [14](#), [28](#), [31](#), [33](#), [40–42](#), [44](#), [46](#), [46](#), [47](#),  
[49](#)  
schema (Schema), [46](#)  
schema(), [11](#), [23](#)  
Schemas, [19](#)  
set\_cpu\_count (cpu\_count), [9](#)  
string (data-type), [11](#)  
struct (data-type), [11](#)  
StructArray (array), [3](#)  
StructScalar (array), [3](#)  
SubTreeFileSystem (FileSystem), [21](#)  
  
Table, [7](#), [33](#), [34](#), [36–39](#), [42](#), [44](#), [46](#), [47](#), [50](#), [51](#),  
[54](#)  
tidy selection specification, [35](#), [37–39](#)  
time32 (data-type), [11](#)  
time64 (data-type), [11](#)  
timestamp (data-type), [11](#)  
type, [49](#)  
  
uint16 (data-type), [11](#)  
uint32 (data-type), [11](#)  
uint64 (data-type), [11](#)  
uint8 (data-type), [11](#)  
unify\_schemas, [49](#)  
UnionDataset (Dataset), [14](#)  
utf8 (data-type), [11](#)  
utf8(), [17](#)  
  
write\_arrow, [50](#)  
write\_feather, [50](#)  
write\_feather(), [44](#), [50](#), [54](#)  
write\_ipc\_stream (write\_arrow), [50](#)  
write\_ipc\_stream(), [44](#), [54](#)  
write\_parquet, [32](#), [52](#)  
write\_to\_raw, [54](#)  
write\_to\_raw(), [44](#), [50](#)