Package 'arrow'

October 7, 2019

Title Integration to 'Apache' 'Arrow'

Version 0.15.0

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

Depends R (>= 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

RoxygenNote 6.1.1

VignetteBuilder knitr

Suggests covr, fs, hms, knitr, lubridate, rmarkdown, testthat, tibble, vctrs

Collate 'enums.R' 'arrow-package.R' 'type.R' 'array-data.R' 'array.R' 'arrowExports.R' 'buffer.R' 'chunked-array.R' 'io.R' 'compression.R' 'compute.R' 'csv.R' 'dictionary.R' 'feather.R' 'field.R' 'filesystem.R' 'install-arrow.R' 'json.R' 'list.R' 'memory-pool.R' 'message.R' 'parquet.R' 'read-record-batch.R' 'read-table.R' 'record-batch-reader.R' 'record-batch-writer.R' 'record-batch.R' 'reexports-bit64.R' 'reexports-tidyselect.R' 'schema.R' 'struct.R' 'table.R' 'util.R' 'write-arrow.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>

Repository CRAN

Date/Publication 2019-10-07 19:00:02 UTC

R topics documented:

`array` *Array class*

Description

Array base type. Immutable data array with some logical type and some length.

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.

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
- `$Slice(offset,length = NULL)` : Construct a zero-copy slice of the array with the indicated offset and length. If length is `NULL`, the slice goes until the end of the array.
- `$RangeEquals(other,start_idx,end_idx,other_start_idx)` :
- `$View(type)`: Construct a zero-copy view of this array with the given type.
- `$Validate()` : Perform any validation checks to determine obvious inconsistencies within the array's internal data. This can be an expensive check, potentially `O(length)`

`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 *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

- `$is_mutable()` :
- `$ZeroPadding()` :
- `$size()` :
- `$capacity()`:

`cast_options` *Cast options*

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

- `$length()`
- `$chunk(i)`
- `$as_vector()`
- `$Slice(offset,length = NULL)`
- `$cast(target_type,safe = TRUE,options = cast_options(safe))`
- `$null_count()`
- `$chunks()`
- `$num_chunks()`
- `$type()`
- `$View(type)`: Construct a zero-copy view of this chunked array with the given type.
- `$Validate()` : Perform any validation checks to determine obvious inconsistencies within the array's internal data. This can be an expensive check, potentially `O(length)`

See Also

Array

`Codec` *Compression Codec class*

Description

Codecs allow you to create compressed input and output streams.

Factory

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

- `type`: string name of the compression method. See CompressionType for a list of possible values. `type` may be upper- or lower-cased. Support for compression methods depends on build-time flags for the C++ library. Most builds support at least "gzip" and "snappy".
- `compression_level`: compression level, the default value (`NA`) uses the default compression level for the selected compression `type`.

compression *Compressed stream classes*

Description

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

Factory

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

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

Methods

Methods are inherited from InputStream and OutputStream, respectively

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 *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 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 *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()

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)

struct(...)
```

Arguments

`unit`	For time/timestamp types, the time unit. `time32()` can take either "s" or "ms", while `time64()` can be "us" or "ns". `timestamp()` can take any of those four values.
`timezone`	For `timestamp()`, an optional time zone string.
`precision`	For `decimal()`, precision
`scale`	For `decimal()`, scale
`type`	For `list_of()`, a data type to make a list-of-type
`...`	For `struct()`, 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")
```

`DataType` *class arrow::DataType*

Description

class arrow::DataType

Methods

TODO

`default_memory_pool` *default arrow::MemoryPool*

Description

default arrow::MemoryPool

Usage

```
default_memory_pool()
```

Value

the default arrow::MemoryPool

dictionary *Create a dictionary type*

Description

Create a dictionary type

Usage

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

Arguments

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

Value

A DictionaryType

See Also

Other Arrow data types

DictionaryType *class DictionaryType*

Description

class DictionaryType

Methods

TODO

FeatherTableReader *FeatherTableReader class*

Description

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

Factory

The `FeatherTableReader$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`).
- `mmap` Logical: whether to memory-map the file (default `TRUE`)
- `...` Additional arguments, currently ignored

Methods

- `$GetDescription()`
- `$HasDescription()`
- `$version()`
- `$num_rows()`
- `$num_columns()`
- `$GetColumnName()`
- `$GetColumn()`
- `$Read(columns)`

FeatherTableWriter *FeatherTableWriter class*

Description

This class enables you to write Feather files. See its usage in `write_feather()`.

Factory

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

- `stream` An `OutputStream`

Methods

- `$GetDescription()`
- `$HasDescription()`
- `$version()`
- `$num_rows()`
- `$num_columns()`
- `$GetColumnName()`
- `$GetColumn()`
- `$Read(columns)`

`Field` *Field class*

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

`name`	field name
`type`	logical type, instance of DataType
`metadata`	currently ignored

Methods

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

Examples

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

FileStats *FileSystem entry stats*

Description

FileSystem entry stats

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.

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

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

- `$GetTargetStats(x)`: `x` may be a Selector or a character vector of paths. Returns a list of FileStats
- `$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 *class arrow::FixedWidthType*

Description

class arrow::FixedWidthType

Methods

TODO

InputStream *InputStream classes*

Description

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

Factory

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

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

To instantiate a MemoryMappedFile, call mmap_open().

Methods

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

---

install_arrow *Help installing the Arrow C++ library*

Description

Binary package installations should come with a working Arrow C++ library, but when installing from source, you'll need to obtain the C++ library first. This function offers guidance on how to get the C++ library depending on your operating system and package version.

Usage

```
install_arrow()
```

Examples

```
install_arrow()
```

MemoryPool *class arrow::MemoryPool*

Description

class arrow::MemoryPool

Methods

TODO

Message *class arrow::Message*

Description

class arrow::Message

Methods

TODO

MessageReader *class arrow::MessageReader*

Description

class arrow::MessageReader

Methods

TODO

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

Description

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

Usage

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

Arguments

path	file path
size	size in bytes

Value

a arrow::io::MemoryMappedFile

mmap_open *Open a memory mapped file*

Description

Open a memory mapped file

Usage

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

Arguments

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

OutputStream *OutputStream classes*

Description

FileOutputStream is for writing to a file; BufferOutputStream and FixedSizeBufferWriter write to buffers; MockOutputStream just reports back how many bytes it received, for testing purposes. 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.
- x For FixedSizeBufferWriter, a Buffer or an object that can be made into a buffer via buffer().

MockOutputStream$create() does not take any arguments.

Methods

- $tell(): return the position in the stream
- $close(): close the stream
- $write(x): send x to the stream
- $capacity(): for BufferOutputStream
- $getvalue(): 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()
tab <- pq$ReadTable(starts_with("c"))
tab$schema
```

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)`

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

Description

These functions uses the Arrow C++ CSV reader to read into a `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
`delim`	Single character used to separate fields within a record.
`quote`	Single character used to quote strings.
`escape_double`	Does the file escape quotes by doubling them? i.e. If this option is `TRUE`, the value `""""` represents a single quote, `\"`.
`escape_backslash`	Does the file use backslashes to escape special characters? This is more general than `escape_double` as backslashes can be used to escape the delimiter character, the quote character, or to add special characters like `\n`.

col_names	If TRUE, the first row of the input will be used as the column names and will not be included in the data frame. If FALSE, column names will be generated by Arrow, starting with "f0", "f1", ..., "fN". Alternatively, you can specify a character vector of column names.
col_select	A character vector of column names to keep, as in the "select" argument to data.table::fread(), or a tidy selection specification of columns, as used in dplyr::select().
na	A character vector of strings to interpret as missing values.
quoted_na	Should missing values inside quotes be treated as missing values (the default) or strings. (Note that this is different from the the Arrow C++ default for the corresponding convert option, strings_can_be_null.)
skip_empty_rows	Should blank rows be ignored altogether? If TRUE, blank rows will not be represented at all. If FALSE, they will be filled with missings.
skip	Number of lines to skip before reading data.
parse_options	see file reader options. If given, this overrides any parsing options provided in other arguments (e.g. delim, quote, etc.).
convert_options	see file reader options
read_options	see file reader options
as_data_frame	Should the function return a data.frame or an arrow::Table?

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 an Table if as_data_frame = FALSE.

Examples

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

read_feather *Read a Feather file*

Description

Read a Feather file

Usage

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

Arguments

file	A character file path, a raw vector, or InputStream, passed to FeatherTableReader$create().
col_select	A character vector of column names to keep, as in the "select" argument to data.table::fread(), or a tidy selection specification of columns, as used in dplyr::select().
as_data_frame	Should the function return a data.frame or an arrow::Table?
...	additional parameters

Value

A data.frame if as_data_frame is TRUE (the default), or an arrow::Table otherwise

Examples

```
try({
  tf <- tempfile()
  on.exit(unlink(tf))
  write_feather(iris, tf)
  df <- read_feather(tf)
  dim(df)
  # Can select columns
  df <- read_feather(tf, col_select = starts_with("Sepal"))
})
```

read_json_arrow *Read a JSON file*

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
col_select	A character vector of column names to keep, as in the "select" argument to data.table::fread(), or a tidy selection specification of columns, as used in dplyr::select().
as_data_frame	Should the function return a data.frame or an arrow::Table?
...	Additional options, passed to json_table_reader()

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 *Read a Message from a stream*

Description

Read a Message from a stream

Usage

```
read_message(stream)
```

Arguments

stream	an InputStream

`read_parquet` *Read a Parquet file*

Description

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

Usage

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

Arguments

file	A character file name, raw vector, or an Arrow input stream
col_select	A character vector of column names to keep, as in the "select" argument to `data.table::fread()`, or a tidy selection specification of columns, as used in `dplyr::select()`.
as_data_frame	Should the function return a `data.frame` or an arrow::Table?
props	ParquetReaderProperties
...	Additional arguments passed to `ParquetFileReader$create()`

Value

A arrow::Table, or a `data.frame` if `as_data_frame` is `TRUE`.

Examples

```
df <- read_parquet(system.file("v0.7.1.parquet", package="arrow"))
head(df)
```

read_record_batch *read arrow::RecordBatch as encapsulated IPC message, given a known arrow::Schema*

Description

read arrow::RecordBatch as encapsulated IPC message, given a known arrow::Schema

Usage

```
read_record_batch(obj, schema)
```

Arguments

obj	a arrow::Message, a arrow::io::InputStream, a Buffer, or a raw vector
schema	a arrow::Schema

Value

a arrow::RecordBatch

read_schema *read a Schema from a stream*

Description

read a Schema from a stream

Usage

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

Arguments

stream	a stream
...	currently ignored

read_table *Read an arrow::Table from a stream*

Description

Read an arrow::Table from a stream

Usage

```
read_table(stream)

read_arrow(stream)
```

Arguments

stream	stream. • a arrow::RecordBatchFileReader: read an arrow::Table from all the record batches in the reader • a arrow::RecordBatchStreamReader: read an arrow::Table from the remaining record batches in the reader • a string file path: interpret the file as an arrow binary file format, and uses a arrow::RecordBatchFileReader to process it. • a raw vector: read using a arrow::RecordBatchStreamReader

Details

The methods using arrow::RecordBatchFileReader and arrow::RecordBatchStreamReader offer the most flexibility. The other methods are for convenience.

Value

- `read_table` returns an arrow::Table
- `read_arrow` returns a `data.frame`

RecordBatch *RecordBatch class*

Description

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

Usage

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

Arguments

...	A data.frame or a named set of Arrays or vectors. If given a mixture of data.frames and vectors, the inputs will be autospliced together (see examples).
schema	a Schema, or NULL (the default) to infer the schema from the data in ...

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".

A caveat about the [method for row operations: only "slicing" is currently supported. That is, you can select a continuous range of rows from the table, but you can't filter with a logical vector or take an arbitrary selection of rows by integer indices.

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.
- $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
- $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

`RecordBatchFileReader` and `RecordBatchStreamReader` are interfaces for generating record batches from different input sources.

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

- `$read_next_batch()`: Returns a `RecordBatch`
- `$schema()`: Returns a Schema
- `$batches()`: Returns a list of `RecordBatches`
- `$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.

RecordBatchWriter *RecordBatchWriter classes*

Description

`RecordBatchFileWriter` and `RecordBatchStreamWriter` are interfaces for writing record batches to either the binary file or streaming format.

Usage

```
writer <- RecordBatchStreamWriter$create(sink, schema)

writer$write_batch(batch)
writer$write_table(table)
writer$close()
```

Factory

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

- `sink` A character file name or an `OutputStream`.
- `schema` A Schema for the data to be written.

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

Schema *Schema class*

Description

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.

Usage

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

Arguments

...	named list of data types

Usage

```
s <- schema(...)

s$ToString()
s$num_fields()
s$field(i)
```

Methods

- `$ToString()`: convert to a string
- `$num_fields()`: returns the number of fields
- `$field(i)`: returns the field at index `i` (0-based)

`Selector` *file selector*

Description

file selector

Usage

```
Selector
```

Factory

The `$create()` factory method instantiates a `Selector` 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_non_existent`: 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.

`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".

A caveat about the `[` method for row operations: only "slicing" is currently supported. That is, you can select a continuous range of rows from the table, but you can't filter with a `logical` vector or take an arbitrary selection of rows by integer indices.

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.
- `$serialize(output_stream,...)`: Write the table to the given OutputStream
- `$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`
- `$schema`
- `$columns`: Returns a list of `ChunkedArray`s

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` *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

write_arrow *Write Arrow formatted data*

Description

Write Arrow formatted data

Usage

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

Arguments

x	an arrow::Table, an arrow::RecordBatch or a data.frame
sink	where to serialize to • A arrow::RecordBatchWriter: the `$write()` of `x` is used. The stream is left open. This uses the streaming format or the binary file format depending on the type of the writer. • A string file path: `x` is serialized with a arrow::RecordBatchFileWriter, i.e. using the binary file format. • A raw vector: typically of length zero (its data is ignored, and only used for dispatch). `x` is serialized using the streaming format, i.e. using the arrow::RecordBatchStreamWriter
...	extra parameters, currently ignored `write_arrow` is a convenience function, the classes arrow::RecordBatchFileWriter and arrow::RecordBatchStreamWriter can be used for more flexibility.

write_feather *Write data in the Feather format*

Description

Write data in the Feather format

Usage

```
write_feather(x, sink)
```

Arguments

x	`data.frame` or RecordBatch
sink	A file path or an OutputStream

Examples

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

`write_parquet` *Write Parquet file to disk*

Description

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

Usage

```
write_parquet(x, sink, chunk_size = NULL, version = 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 arrow::Table, or an object convertible to it.
`sink`	an arrow::io::OutputStream 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".
`compression`	compression algorithm. No compression by default.
`compression_level`	compression level.
`use_dictionary`	Specify if we should use dictionary encoding.
`write_statistics`	Specify if we should write statistics
`data_page_size`	Set a target threshhold for the approximate encoded size of data pages within a column chunk. If omitted, the default data page size (1Mb) is used.

properties
: properties for parquet writer, derived from arguments `version`, `compression`, `compression_level`, `use_dictionary`, `write_statistics` and `data_page_size`

use_deprecated_int96_timestamps
: Write timestamps to INT96 Parquet format

coerce_timestamps
: Cast timestamps a particular resolution. can be NULL, "ms" or "us"

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

arrow_properties
: arrow specific writer properties, derived from arguments `use_deprecated_int96_timestamps`, `coerce_timestamps` and `allow_truncated_timestamps`

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 - 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.

Value

NULL, invisibly

Examples

```
tf1 <- tempfile(fileext = ".parquet")
write_parquet(data.frame(x = 1:5), tf2)

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

Index