Package ‘arrow’

July 25, 2020

Title Integration to 'Apache' 'Arrow'

Version 1.0.0

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, 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-tidyselect.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>

Repository CRAN

Date/Publication 2020-07-25 04:30:02 UTC

R topics documented:

`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 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(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(): 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(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 arguments:

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

`codec_is_available` *Check whether a compression codec is available*

Description

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

Usage

```
codec_is_available(type)
```

Arguments

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

Value

Logical: is `type` available?

`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

`cpu_count` *Manage the global CPU thread pool in libarrow*

Description

Manage the global CPU thread pool in libarrow

Usage

```
cpu_count()

set_cpu_count(num_threads)
```

Arguments

`num_threads` integer: New number of threads for thread pool

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()

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. `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
`list_size`	list size for `FixedSizeList` 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")
```

`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 `Dataset`s.

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

Usage

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

Arguments

`x` A string file x containing data files, or a list of `DatasetFactory` objects whose datasets should be grouped. If this argument is specified it will be used to construct a `UnionDatasetFactory` and other arguments will be ignored.

`filesystem` A FileSystem object; if omitted, the `FileSystem` will be detected from `x`

`format` A FileFormat object, or a string identifier of the format of the files in `x`. Currently supported values:

- "parquet"
- "ipc"/"arrow"/"feather", all aliases for each other; for Feather, note that only version 2 files are supported
- "csv"/"text", aliases for the same thing (because comma is the default delimiter for text files
- "tsv", equivalent to passing format = "text", delimiter = "\t"

Default is "parquet", unless a `delimiter` is also specified, in which case it is assumed to be "text".

`partitioning` One of

- A Schema, in which case the file paths relative to sources will be parsed, and path segments will be matched with the schema fields. For example, schema(year = int16(),month = int8()) would create partitions for file paths like "2019/01/file.parquet", "2019/02/file.parquet", etc.
- A character vector that defines the field names corresponding to those path segments (that is, you're providing the names that would correspond to a Schema but the types will be autodetected)
- A HivePartitioning or HivePartitioningFactory, as returned by hive_partition() which parses explicit or autodetected fields from Hive-style path segments
- NULL for no partitioning

... 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 *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

Expression *Arrow expressions*

Description

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

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

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

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

Description

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

Factory

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

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` *file selector*

Description

file selector

Factory

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

Fields

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

`FileSystem` *FileSystem classes*

Description

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

Factory

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` *class arrow::FixedWidthType*

Description

class arrow::FixedWidthType

Methods

TODO

`hive_partition` *Construct Hive partitioning*

Description

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

Usage

```
hive_partition(...)
```

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 `BufferReader` for reading from a buffer. Use these with the various table readers.

Factory

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

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

To instantiate a `MemoryMappedFile`, call `mmap_open()`.

Methods

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

install_arrow *Install or upgrade the Arrow library*

Description

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

Usage

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

Arguments

`nightly` logical: Should we install a development version of the package, or should we install from CRAN (the default).

`binary` On Linux, value to set for the environment variable `LIBARROW_BINARY`, which governs how C++ binaries are used, if at all. The default value, `TRUE`, tells the installation script to detect the Linux distribution and version and find an appropriate C++ library. `FALSE` would tell the script not to retrieve a binary and instead build Arrow C++ from source. Other valid values are strings corresponding to a Linux distribution-version, to override the value that would be detected. See `vignette("install",package = "arrow")` for further details.

`use_system` logical: Should we use `pkg-config` to look for Arrow system packages? Default is `FALSE`. If `TRUE`, source installation may be faster, but there is a risk of version mismatch.

`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 `install.packages()`)

`...` Additional arguments passed to `install.packages()`

See Also

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

install_pyarrow *Install pyarrow for use with reticulate*

Description

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

Usage

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

Arguments

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

map_batches *Apply a function to a stream of RecordBatches*

Description

As an alternative to calling collect() on a Dataset query, you can use this function to access the stream of RecordBatches in the Dataset. This lets you 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 arrow_dplyr_query object, as returned by the dplyr methods on Dataset.
FUN	A function or purrr-style lambda expression to apply to each batch
...	Additional arguments passed to FUN
.data.frame	logical: collect the resulting chunks into a single data.frame? Default TRUE

Details

This is experimental and not recommended for production use.

`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)

open_dataset *Open a multi-file dataset*

Description

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

Usage

```
open_dataset(
  sources,
  schema = NULL,
  partitioning = hive_partition(),
  unify_schemas = NULL,
  ...
)
```

Arguments

`sources` Either:

- a string path to a directory containing data files
- a list of `Dataset` objects as created by this function
- a list of `DatasetFactory` objects as created by `dataset_factory()`.

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

`partitioning` When `sources` is a file path, 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

The default is to autodetect Hive-style partitions.

`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` *Read Arrow IPC stream format*

Description

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

Usage

```
read_arrow(file, ...)

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

Arguments

`file` A character file name, `raw` vector, or an Arrow input stream. If a file name, a memory-mapped Arrow InputStream will be opened and closed when finished. If an input stream is provided, it will be left open.

`...` extra parameters passed to `read_feather()`.

`as_data_frame` Should the function return a `data.frame` (default) or an Arrow Table?

Details

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

Value

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

See Also

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

read_delim_arrow *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. If a file name, a memory-mapped Arrow InputStream will be opened and closed when finished; compression will be detected from the file extension and handled automatically. If an input stream is provided, it will be left open.

`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` (default) 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 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 *Read a Feather file*

Description

Feather provides binary columnar serialization for data frames. It is designed to make reading and writing data frames efficient, and to make sharing data across data analysis languages easy. 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 InputStream will be opened and closed when finished. If an input stream is provided, it will be left open.
`col_select`	A character vector of column names to keep, as in the "select" argument to `data.table::fread()`, or a tidy selection specification of columns, as used in `dplyr::select()`.
`as_data_frame`	Should the function return a `data.frame` (default) or an Arrow Table?
`...`	additional parameters, passed to FeatherReader$create()

Value

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

See Also

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

Examples

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

read_json_arrow *Read a JSON file*

Description

Using JsonTableReader

Usage

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

Arguments

`file` A character file name, `raw` vector, or an Arrow input stream. If a file name, a memory-mapped Arrow InputStream will be opened and closed when finished; compression will be detected from the file extension and handled automatically. If an input stream is provided, it will be left open.

`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` (default) 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. If a file name, a memory-mapped Arrow InputStream will be opened and closed when finished. If an input stream is provided, it will be left open.
`col_select`	A character vector of column names to keep, as in the "select" argument to `data.table::fread()`, or a tidy selection specification of columns, as used in `dplyr::select()`.
`as_data_frame`	Should the function return a `data.frame` (default) 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` (the default).

Examples

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

`read_schema` *read a Schema from a stream*

Description

read a Schema from a stream

Usage

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

Arguments

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

Value

A Schema

`RecordBatch` *RecordBatch class*

Description

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

Usage

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

Arguments

`...` A `data.frame` or a named set of Arrays or vectors. If given a mixture of data.frames and vectors, the inputs will be autospliced together (see examples). Alternatively, you can provide a single Arrow IPC `InputStream`, `Message`, `Buffer`, or R `raw` object containing a `Buffer`.

`schema` a Schema, or `NULL` (the default) to infer the schema from the data in `...`. When providing an Arrow IPC buffer, `schema` is required.

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 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 `Field`s 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 `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

`unify_schemas` *Combine and harmonize schemas*

Description

Combine and harmonize schemas

Usage

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

Arguments

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

Value

A `Schema` with the union of fields contained in the inputs

Examples

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

## End(Not run)
```

write_arrow *Write Arrow IPC stream format*

Description

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

Usage

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

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

Arguments

`x`	`data.frame`, RecordBatch, or Table
`sink`	A string file path or OutputStream
`...`	extra parameters passed to `write_feather()`.

Details

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

Value

`x`, invisibly.

See Also

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

write_feather *Write data in the Feather format*

Description

Feather provides binary columnar serialization for data frames. It is designed to make reading and writing data frames efficient, and to make sharing data across data analysis languages easy. This function writes both the original, limited specification of the format and the version 2 specification, which is the Apache Arrow IPC file format.

Usage

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

Arguments

`x`	`data.frame`, RecordBatch, or Table
`sink`	A string file path or OutputStream
`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 `codec_is_available()`. 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 *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". Default "1.0"
`compression`	compression algorithm. Default "snappy". See details.
`compression_level`	compression level. Meaning depends on compression algorithm
`use_dictionary`	Specify if we should use dictionary encoding. Default `TRUE`
`write_statistics`	Specify if we should write statistics. Default `TRUE`
`data_page_size`	Set a target threshold for the approximate encoded size of data pages within a column chunk (in bytes). Default 1 MiB.

`properties` properties for parquet writer, derived from arguments `version`, `compression`, `compression_level`, `use_dictionary`, `write_statistics` and `data_page_size`. You should not specify any of these arguments if you also provide a `properties` argument, as they will be ignored.

`use_deprecated_int96_timestamps`
Write timestamps to INT96 Parquet format. Default `FALSE`.

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

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

`arrow_properties`
arrow specific writer properties, derived from arguments `use_deprecated_int96_timestamps`, `coerce_timestamps` and `allow_truncated_timestamps` You should not specify any of these arguments if you also provide a `properties` 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, RecordBatch, or Table
format	one of c("stream","file"), indicating the IPC format to use

Value

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

Index