Package ‘arrow’

February 14, 2020

Title Integration to ‘Apache’ ‘Arrow’

Version 0.16.0.2

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


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 7.0.2

VignetteBuilder knitr

Suggests dplyr, hms, knitr, lubridate,pkgload, rmarkdown, testthat, tibble

'reexports-bit64.R' 'reexports-tidyselect.R' 'schema.R'
'struct.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**  2020-02-14 12:20:05 UTC

**R topics documented:**

array .................................................. 3
ArrayData ........................................... 5
arrow_available .................................... 5
buffer ................................................ 6
cast_options ....................................... 6
ChunkedArray ....................................... 7
Codec ................................................ 8
codec_is_available ................................. 8
compression ....................................... 9
CsvReadOptions ..................................... 9
CsvTableReader .................................... 10
data-type ........................................... 11
Dataset ............................................. 13
DataType .......................................... 14
default_memory_pool ............................... 14
dictionary ......................................... 14
DictionaryType ..................................... 15
Expression ......................................... 15
FeatherTableReader ................................. 16
FeatherTableWriter ................................. 16
Field .............................................. 17
FileFormat ......................................... 18
FileSelector ....................................... 18
FileStats .......................................... 19
FileSystem ......................................... 19
FixedWidthType ................................... 20
hive_partition .................................... 20
InputStream ........................................ 21
install_arrow ....................................... 22
MemoryPool ......................................... 23
Message ........................................... 23
Array

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

```r
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 Arrow Array) `i`.
- `$filter(i)`: return an `Array` with values at positions where logical vector (or Arrow boolean Array) `i` is TRUE.
- `$rangeEquals(other, start_id, end_id, other_start_id)`:  
- `$cast(target_type, safe = TRUE, options = cast_options(safe))`: Alter the data in the array to change its type.
ArrayData

• $View(type)$: 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**

```r
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**

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

**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 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**

```r
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)`: 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
 Codec allows 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
 Codec allows you to create compressed input and output streams.

**Usage**

codec_is_available(type)

**Arguments**

<table>
<thead>
<tr>
<th>type</th>
<th>A string, one of &quot;uncompressed&quot;, &quot;snappy&quot;, &quot;gzip&quot;, &quot;brotli&quot;, &quot;zstd&quot;, &quot;lz4&quot;, &quot;lzo&quot;, or &quot;bz2&quot;, case insensitive.</th>
</tr>
</thead>
</table>
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

---

**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 strings 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) char-
    acters? (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 fol-
loving 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

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

### 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")

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Multi-file datasets</th>
</tr>
</thead>
</table>

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).

DatasetFactory is used to help in the creation of Datasets.

Start a new scan of the data

Return the Dataset’s Schema

Value

A ScannerBuilder

Factory

The Dataset$create() method instantiates a Dataset and takes the following arguments:

- sources: a list of Source objects
- schema: a Schema

The DatasetFactory$create() takes the following arguments:

- sources: a list of SourceFactory objects

Methods

A Dataset has the following methods:

- $NewScan(): Returns a ScannerBuilder for building a query
- $schema: Active binding, returns the Schema of the Dataset

A DatasetFactory has:

- $Inspect(): Returns a common Schema for the Sources in the factory.
- $Finish(schema): Returns a Dataset
See Also

`open_dataset()` for a simple interface to creating 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`)?
**DictionaryType**

**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. FieldExpressions refer to columns in the Dataset and are compared to ScalarExpressions using ComparisonExpressions. ComparisonExpressions may be combined with AndExpression or OrExpression and negated with NotExpression. IsValidExpression is essentially `is.na()` for Expressions.

**Factory**

FieldExpression$create(name) takes a string name as input. This string should refer to a column in a Dataset at the time it is evaluated, but you can construct a FieldExpression independently of any Dataset.

ScalarExpression$create(x) takes a scalar (length-1) R value as input.

ComparisonExpression$create(OP,e1,e2) takes a string operator name (e.g. "==", "+", ">", etc.) and two Expression objects.

AndExpression$create(e1,e2) and OrExpression$create(e1,e2) take two Expression objects, while NotExpression$create(e1) and IsValidExpression$create(e1) take a single Expression.
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
Field

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

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

Examples

field("x", int32())
### Description

A `FileFormat` holds information about how to read and parse the files included in a `Source`. 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 format of the files in `path`. Currently supported options are "parquet", "arrow", and "ipc" (an alias for the Arrow file format)
- `...`: Additional format-specific options

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

---

### 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_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.
**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**

- `$GetTargetStats(x)`: `x` may be a FileSelector or a character vector of paths. Returns a list of FileStats
- `$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

**Description**

class arrow::FixedWidthType

**Methods**

Todo

---

### hive_partition

**Description**

Construct Hive partitioning

**Usage**

hive_partition(...)

**Arguments**

... named list of data types, passed to schema()

**Details**

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

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

Examples

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

---

**InputStream**

**InputStream classes**

---

Description

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

Factory

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

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

To instantiate a MemoryMappedFile, call mmap_open().

Methods

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

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

```r
install_arrow(
  nightly = FALSE,
  binary = Sys.getenv("LIBARROW_BINARY", TRUE),
  use_system = Sys.getenv("ARROW_USE_PKG_CONFIG", 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.
- **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",package = "arrow") for more ways to tune installation on Linux.
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**
```r
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**
```r
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(), ...)

Arguments

sources  Either a string path to a directory containing data files, or a list of SourceFactory objects as created by open_source().
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

...  additional arguments passed to open_source() when sources is a file path, otherwise ignored.

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")
Create a Source for a Dataset

Description

A Dataset can have one or more Sources. A Source contains one or more Fragments, such as files, of a common storage location, format, and partitioning. This function helps you construct a Source that you can pass to `open_dataset()`.

Usage

```r
open_source(
  path,
  filesystem = c("auto", "local"),
  format = c("parquet", "arrow", "ipc"),
  partitioning = NULL,
  allow_non_existent = FALSE,
  recursive = TRUE,
  ...
)
```

Arguments

- **path** A string file path containing data files
- **filesystem** A string identifier for the filesystem corresponding to path. Currently only "local" is supported.
- **format** A string identifier of the format of the files in path. Currently supported options are "parquet", "arrow", and "ipc" (an alias for the Arrow file format)
- **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 autodected)
  - A HivePartitioning or HivePartitioningFactory, as returned by `hive_partition()` which parses explicit or autodected fields from Hive-style path segments
  - `NULL` for no partitioning
- **allow_non_existent** logical: is path allowed to not exist? Default FALSE. See `FileSelector`.
- **recursive** logical: should files be discovered in subdirectories of path? Default TRUE.
- **...** Additional arguments passed to the FileSystem $create() method
Details

If you only have a single Source, such as a directory containing Parquet files, you can call open_dataset() directly. Use open_source() when you want to combine different directories, file systems, or file formats.

Value

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

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**

```r
f <- system.file("v0.7.1.parquet", package="arrow")
pq <- ParquetFileReader$create(f)
pq$GetSchema()
tab <- pq$ReadTable(starts_with("c"))
tab$schema
```

---

**ParquetFileWriter**  
*ParquetFileWriter class*

**Description**

This class enables you to interact with Parquet files.
ParquetReaderProperties

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 Source**

Description

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

---

**read_delim_arrow**

*Read a CSV or other delimited file with Arrow*

**Description**

These functions uses the Arrow C++ CSV reader to read into a 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**

```r
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
)
```

```r
read_csv_arrow(
```

```r
```
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 a Table if `as_data_frame = FALSE`.

### Examples

```r
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

Description

Read a Feather file

Usage

```r
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

```r
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

Description

Using `JsonTableReader`

Usage

```r
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

```r
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**

```r
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 (the default).

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 Message, InputStream, or Buffer
- **...**
  - currently ignored

**Value**

- A Schema
RecordBatch

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(\ldots, schema = \texttt{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.
• $Take(i): return an RecordBatch with rows at positions given by integers (R vector or Array Array) i.
• $Filter(i): 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
• $columns: Returns a list of Arrays
Examples

```r
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

```r
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

---

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. Use a `ScannerBuilder`, from a `Dataset`'s `$NewScan()` method, to construct one.

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`.
- `$UseThreads(threads)`: Logical: should the scan use multithreading? The method’s default input is `TRUE`, but you must call the method to enable multithreading because the scanner default is `FALSE`.
- `$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`. 
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

Active bindings

- `$names`: returns the field names (called in `names(Schema)`)
- `$num_fields`: returns the number of fields (called in `length(Schema)`)
- `$fields`: returns the list of Fields in the Schema, suitable for iterating over
- `$hasMetadata`: logical: does this Schema have extra metadata?
- `$metadata`: returns the extra metadata, if present, else NULL

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```
Sources for a Dataset

Description

A Dataset can have one or more Sources. A Source contains one or more Fragments, such as files, of a common type and partitioning. SourceFactory is used to create a Source, inspect the Schema of the fragments contained in it, and declare a partitioning. FileSystemSourceFactory is a subclass of SourceFactory for discovering files in the local file system, the only currently supported file system.

In general, you'll deal with SourceFactory rather than Source itself.

Return the Source's Schema

Factory

For the SourceFactory$create() factory method, see open_source(), an alias for it.

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

- filesystem: A FileSystem
- selector: A FileSelector
- format: A string identifier of the format of the files in path. Currently supported options are "parquet", "arrow", and "ipc" (an alias for the Arrow file format)

Methods

Source has one defined method:

- $schema: Active binding, returns the Schema of the Source

SourceFactory and its subclasses have the following methods:

- $inspect(): Walks the files in the directory and returns a common Schema
- $finish(schema): Returns a Source

See Also

Dataset for what to do with a Source
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:

- \ldots\) 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 \ldots\)

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.
- $Take(i): return a 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): return a 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
• $\text{cast}(\text{target\_schema}, \text{safe} = \text{TRUE}, \text{options} = \text{cast\_options}(\text{safe}))$: Alter the schema of the record batch.

There are also some active bindings

• $\text{num\_columns}$

• $\text{num\_rows}$

• $\text{schema}$

• $\text{columns}$: Returns a list of ChunkedArrays

Examples

```r
tab <- Table$create(name = rownames(mtcars), mtcars)
dim(tab)
dim(head(tab))
names(tab)
tab$\text{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.

Value
the input x invisibly.

write_feather

Write data in the Feather format

Description
Write data in the Feather format

Usage
write_feather(x, sink)
write_parquet argument

**Arguments**

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

**Value**

the input `x` invisibly.

**Examples**

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

**Description**

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

**Usage**

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

$NewScan(), 25

AndExpression (Expression), 15
Array, 8, 38
Array (array), 3
array, 3
ArrayData, 4, 5
Arrays, 7
arrow :: io :: InputStream, 37
arrow :: io :: MemoryMappedFile, 24
arrow :: io :: OutputStream, 29, 48
arrow :: Message, 37
arrow :: RecordBatch, 37, 46
arrow :: RecordBatchFileReader, 38
arrow :: RecordBatchFileWriter, 46
arrow :: RecordBatchStreamReader, 38
arrow :: RecordBatchStreamWriter, 46
arrow :: RecordBatchWriter, 46
arrow :: Schema, 37
arrow :: Table, 33–36, 38, 46, 48
arrow_available, 5
arrow_available(), 22

bool (data-type), 11
boolean (data-type), 11
Buffer, 21, 27, 37, 40
Buffer (buffer), 6
buffer, 6
BufferOutputStream (OutputStream), 27
BufferReader (InputStream), 21

cast_options, 6
chunked arrays, 44
chunked_array (ChunkedArray), 7
ChunkedArray, 7
Codec, 8, 9
codec_is_available, 8
codec_is_available(), 8, 49
ComparisonExpression (Expression), 15

compressed input and output streams, 8
CompressedInputStream (compression), 9
CompressedOutputStream (compression), 9
compression, 9
CsvConvertOptions (CsvReadOptions), 9
CsvParseOptions (CsvReadOptions), 9
CsvReadOptions, 9, 10
CsvTableReader, 10, 33
data type, 4
data types, 20, 42
data-type, 11
Dataset, 13, 15, 25, 26, 41–43
DatasetFactory (Dataset), 13
DataType, 14, 17
date32 (data-type), 11
date64 (data-type), 11
decimal (data-type), 11
default_memory_pool, 14
dictionary, 14
dictionary(), 13
DictionaryArray (array), 3
DictionaryType, 15, 15
DirectoryPartitioning (Partitioning), 30
DirectoryPartitioningFactory (Partitioning), 30

Expression, 15, 41

FeatherTableReader, 16
FeatherTableWriter, 16
Field, 17, 42
field (Field), 17
FieldExpression (Expression), 15
fields, 38
file reader options, 33
FormatException, 18
FileOutputStream (OutputStream), 27
FileSelector, 18, 19, 26, 43
FileStats, 19, 19
RecordBatchWriter, 40

ScalarExpression (Expression), 15
Scanner, 15, 41
ScannerBuilder, 13
ScannerBuilder (Scanner), 41
Schema, 13, 25, 29, 31, 37–41, 42, 43, 44
schema (Schema), 42
schema(), 11, 20
Schemas, 17
Source, 13, 26, 43
SourceFactory, 13
SourceFactory (Source), 43
string (data-type), 11
struct (data-type), 11
StructArray (array), 3
SubTreeFileSystem (FileSystem), 19

Table, 7, 41, 42, 44
tidy selection specification, 33–36
time32 (data-type), 11
time64 (data-type), 11
timestamp (data-type), 11
type, 45

uint16 (data-type), 11
uint32 (data-type), 11
uint64 (data-type), 11
uint8 (data-type), 11
utf8 (data-type), 11
utf8(), 14

write_arrow, 46
write_feather, 46
write_Feather(), 16
write_parquet, 30, 47