Package ‘nanoarrow’

May 31, 2024

Title  Interface to the ‘nanoarrow’ ‘C’ Library
Version  0.5.0.1
Description  Provides an ‘R’ interface to the ‘nanoarrow’ ‘C’ library and the ‘Apache Arrow’ application binary interface. Functions to import and export ‘ArrowArray’, ‘ArrowSchema’, and ‘ArrowArrayStream’ ‘C’ structures to and from ‘R’ objects are provided alongside helpers to facilitate zero-copy data transfer among ‘R’ bindings to libraries implementing the ‘Arrow’ ‘C’ data interface.
License  Apache License (>= 2)
Encoding  UTF-8
RoxygenNote  7.2.3
BugReports  https://github.com/apache/arrow-nanoarrow/issues
Suggests  arrow (>= 9.0.0), bit64, blob, jsonlite, hms, rlang, testthat (>= 3.0.0), tibble, vctrs, withr
Config/testthat/edition  3
Config/build/bootstrap  TRUE
NeedsCompilation  yes
Author  Dewey Dunnington [aut, cre] (<https://orcid.org/0000-0002-9415-4582>), Apache Arrow [aut, cph], Apache Software Foundation [cph]
Maintainer  Dewey Dunnington <dewey@dunnington.ca>
Repository  CRAN
Date/Publication  2024-05-31 13:30:02 UTC

R topics documented:

  array_stream_set_finalizer ........................................ 2
  as_nanoarrow_array .............................................. 3
array_stream_set_finalizer

Register an array stream finalizer

Description

In some cases, R functions that return a nanoarrow_array_stream may require that the scope of some other object outlive that of the array stream. If there is a need for that object to be released deterministically (e.g., to close open files), you can register a function to run after the stream’s release callback is invoked from the R thread. Note that this finalizer will not be run if the stream’s release callback is invoked from a non-R thread. In this case, the finalizer and its chain of environments will be garbage-collected when nanoarrow::preserved_empty() is run.

Usage

array_stream_set_finalizer(array_stream, finalizer)

Arguments

array_stream A nanoarrow_array_stream
finalizer A function that will be called with zero arguments.

Value

A newly allocated array_stream whose release callback will call the supplied finalizer.
as_nanoarrow_array

Examples

stream <- array_stream_set_finalizer(
    basic_array_stream(list(1:5)),
    function() message("All done!"
)
stream$release()

---

as_nanoarrow_array  Convert an object to a nanoarrow array

Description

In nanoarrow an 'array' refers to the struct ArrowArray definition in the Arrow C data interface. At the R level, we attach a schema such that functionally the nanoarrow_array class can be used in a similar way as an arrow::Array. Note that in nanoarrow an arrow::RecordBatch and a non-nullable arrow::StructArray are represented identically.

Usage

as_nanoarrow_array(x, ..., schema = NULL)

Arguments

x  An object to convert to a array
...
Passed to S3 methods
schema  An optional schema used to enforce conversion to a particular type. Defaults to infer_nanoarrow_schema().

Value

An object of class 'nanoarrow_array'

Examples

(array <- as_nanoarrow_array(1:5))
as.vector(array)

(array <- as_nanoarrow_array(data.frame(x = 1:5)))
as.data.frame(array)
as_nanoarrow_array_stream

Convert an object to a nanoarrow array_stream

Description

In nanoarrow, an 'array stream' corresponds to the struct ArrowArrayStream as defined in the Arrow C Stream interface. This object is used to represent a stream of arrays with a common schema. This is similar to an arrow::RecordBatchReader except it can be used to represent a stream of any type (not just record batches). Note that a stream of record batches and a stream of non-nullable struct arrays are represented identically. Also note that array streams are mutable objects and are passed by reference and not by value.

Usage

as_nanoarrow_array_stream(x, ..., schema = NULL)

Arguments

x An object to convert to a array_stream
... Passed to S3 methods
schema An optional schema used to enforce conversion to a particular type. Defaults to infer_nanoarrow_schema().

Value

An object of class 'nanoarrow_array_stream'

Examples

(stream <- as_nanoarrow_array_stream(data.frame(x = 1:5)))
stream$get_schema()
stream$get_next()

# The last batch is returned as NULL
stream$get_next()

# Release the stream
stream$release()
as_nanoarrow_buffer  Convert an object to a nanoarrow buffer

Description
Convert an object to a nanoarrow buffer

Usage
as_nanoarrow_buffer(x, ...)

Arguments
x  An object to convert to a buffer
... Passed to S3 methods

Value
An object of class 'nanoarrow_buffer'

Examples
array <- as_nanoarrow_array(c(NA, 1:4))
array$buffers
as.raw(array$buffers[[1]])
as.raw(array$buffers[[2]])
convert_buffer(array$buffers[[1]])
convert_buffer(array$buffers[[2]])

as_nanoarrow_schema  Convert an object to a nanoarrow schema

Description
In nanoarrow a 'schema' refers to a struct ArrowSchema as defined in the Arrow C Data interface. This data structure can be used to represent an arrow::schema(), an arrow::field(), or an arrow::DataType. Note that in nanoarrow, an arrow::schema() and a non-nullable arrow::struct() are represented identically.
as_nanoarrow_vctr

Usage

as_nanoarrow_schema(x, ...)
infer_nanoarrow_schema(x, ...)
nanoarrow_schema_parse(x, recursive = FALSE)
nanoarrow_schema_modify(x, new_values, validate = TRUE)

Arguments

x An object to convert to a schema
... Passed to S3 methods
recursive Use TRUE to include a children member when parsing schemas.
new_values New schema component to assign
validate Use FALSE to skip schema validation

Value

An object of class 'nanoarrow_schema'

Examples

infer_nanoarrow_schema(integer())
infer_nanoarrow_schema(data.frame(x = integer()))

Description

This experimental vctr class allows zero or more Arrow arrays to present as an R vector without converting them. This is useful for arrays with types that do not have a non-lossy R equivalent, and helps provide an intermediary object type where the default conversion is prohibitively expensive (e.g., a nested list of data frames). These objects will not survive many vctr transformations; however, they can be sliced without copying the underlying arrays.

Usage

as_nanoarrow_vctr(x, ..., schema = NULL, subclass = character())
nanoarrow_vctr(schema = NULL, subclass = character())
**basic_array_stream**

**Arguments**

- **x**  
  An object that works with `as_nanoarrow_array_stream()`.
- **...**  
  Passed to `as_nanoarrow_array_stream()`
- **schema**  
  An optional schema
- **subclass**  
  An optional subclass of nanoarrow_vctr to prepend to the final class name.

**Details**

The nanoarrow_vctr is currently implemented similarly to `factor()`: its storage type is an `integer()` that is a sequence along the total length of the vctr and there are attributes that are required to resolve these indices to an array + offset. Sequences typically have a very compact representation in recent versions of R such that this has a cheap storage footprint even for large arrays. The attributes are currently:

- **schema**: The `nanoarrow_schema` shared by each chunk.
- **chunks**: A `list()` of `nanoarrow_array`.
- **offsets**: An `integer()` vector beginning with `0` and followed by the cumulative length of each chunk. This allows the chunk index + offset to be resolved from a logical index with \( \log(n) \) complexity.

This implementation is preliminary and may change; however, the result of `as_nanoarrow_array_stream(some_vctr[begin:end])` should remain stable.

**Value**

A vctr of class `nanoarrow_vctr`

**Examples**

```r
array <- as_nanoarrow_array(1:5)
as_nanoarrow_vctr(array)
```

---

**basic_array_stream**  
Create ArrayStreams from batches

**Description**

Create ArrayStreams from batches

**Usage**

```r
basic_array_stream(batches, schema = NULL, validate = TRUE)
```
Arguments

- **batches**: A list() of nanoarrow_array objects or objects that can be coerced via as_nanoarrow_array().
- **schema**: A nanoarrow_schema or NULL to guess based on the first schema.
- **validate**: Use FALSE to skip the validation step (i.e., if you know that the arrays are valid).

Value

An nanoarrow_array_stream

Examples

```r
(stream <- basic_array_stream(list(data.frame(a = 1, b = 2))))
as.data.frame(stream$get_next())
stream$get_next()
```

---

**convert_array**

*Convert an Array into an R vector*

Description

Converts array to the type specified by to. This is a low-level interface; most users should use as.data.frame() or as.vector() unless finer-grained control is needed over the conversion. This function is an S3 generic dispatching on to: developers may implement their own S3 methods for custom vector types.

Usage

```r
convert_array(array, to = NULL, ...)
```

Arguments

- **array**: A nanoarrow_array.
- **to**: A target prototype object describing the type to which array should be converted, or NULL to use the default conversion as returned by infer_nanoarrow_ptype(). Alternatively, a function can be passed to perform an alternative calculation of the default ptype as a function of array and the default inference of the prototype.
- **...**: Passed to S3 methods
### Details

Conversions are implemented for the following R vector types:

- **logical()**: Any numeric type can be converted to `logical()` in addition to the `bool` type. For numeric types, any non-zero value is considered `TRUE`.

- **integer()**: Any numeric type can be converted to `integer()`, however, a warning will be signaled if the any value is outside the range of the 32-bit integer.

- **double()**: Any numeric type can be converted to `double()`. This conversion currently does not warn for values that may not roundtrip through a floating-point double (e.g., very large uint64 and int64 values).

- **character()**: String and large string types can be converted to `character()`. The conversion does not check for valid UTF-8: if you need finer-grained control over encodings, use `to = blob::blob()`.

- **factor()**: Dictionary-encoded arrays of strings can be converted to `factor()`. However, this must be specified explicitly (i.e., `convert_array(array, factor())`) because arrays arriving in chunks can have dictionaries that contain different levels. Use `convert_array(array, factor(levels = c(...)))` to materialize an array into a vector with known levels.

- **Date**: Only the date32 type can be converted to an R Date vector.

- **hms::hms()**: Time32 and time64 types can be converted to `hms::hms()`.

- **difftime()**: Time32, time64, and duration types can be converted to R `difftime()` vectors. The value is converted to match the `units()` attribute of to.

- **blob::blob()**: String, large string, binary, and large binary types can be converted to `blob::blob()`.

- **vctrs::list_of()**: List, large list, and fixed-size list types can be converted to `vctrs::list_of()`.

- **data.frame()**: Struct types can be converted to `data.frame()`.

- **vctrs::unspecified()**: Any type can be converted to `vctrs::unspecified()`; however, a warning will be raised if any non-null values are encountered.

In addition to the above conversions, a null array may be converted to any target prototype except `data.frame()`. Extension arrays are currently converted as their storage type.

### Value

An R vector of type to.

### Examples

```r
array <- as_nanoarrow_array(data.frame(x = 1:5))
str(convert_array(array))
str(convert_array(array, to = data.frame(x = double())))
```
**convert_array_stream**  
*Convert an Array Stream into an R vector*

**Description**
Converts `array_stream` to the type specified by `to`. This is a low-level interface; most users should use `as.data.frame()` or `as.vector()` unless finer-grained control is needed over the conversion. See `convert_array()` for details of the conversion process; see `infer_nanoarrow_ptype()` for default inferences of `to`.

**Usage**

```r
convert_array_stream(array_stream, to = NULL, size = NULL, n = Inf)
```

```r
collect_array_stream(array_stream, n = Inf, schema = NULL, validate = TRUE)
```

**Arguments**
- `array_stream` A `nanoarrow_array_stream`.
- `to` A target prototype object describing the type to which `array` should be converted, or `NULL` to use the default conversion as returned by `infer_nanoarrow_ptype()`. Alternatively, a function can be passed to perform an alternative calculation of the default ptype as a function of `array` and the default inference of the prototype.
- `size` The exact size of the output, if known. If specified, slightly more efficient implementation may be used to collect the output.
- `n` The maximum number of batches to pull from the array stream.
- `schema` A `nanoarrow_schema` or `NULL` to guess based on the first schema.
- `validate` Use `FALSE` to skip the validation step (i.e., if you know that the arrays are valid).

**Value**
- `convert_array_stream()`: An R vector of type `to`.
- `collect_array_stream()`: A list() of `nanoarrow_array`

**Examples**
```r
stream <- as_nanoarrow_array_stream(data.frame(x = 1:5))
str(convert_array_stream(stream))
str(convert_array_stream(stream, to = data.frame(x = double())))

stream <- as_nanoarrow_array_stream(data.frame(x = 1:5))
collect_array_stream(stream)
```
infer_nanoarrow_ptype

Infer an R vector prototype

Description

Resolves the default to value to use in convert_array() and convert_array_stream(). The default conversions are:

Usage

infer_nanoarrow_ptype(x)

Arguments

x A nanoarrow_schema, nanoarrow_array, or nanoarrow_array_stream.

Details

- null to vctrs::unspecified()
- boolean to logical()
- int8, uint8, int16, uint16, and int13 to integer()
- uint32, int64, uint64, float, and double to double()
- string and large string to character()
- struct to data.frame()
- binary and large binary to blob::blob()
- list, large_list, and fixed_size_list to vctrs::list_of()
- time32 and time64 to hms::hms()
- duration to difftime()
- date32 to as.Date()
- timestamp to as.POSIXct()

Additional conversions are possible by specifying an explicit value for to. For details of each conversion, see convert_array().

Value

An R vector of zero size describing the target into which the array should be materialized.

Examples

infer_nanoarrow_ptype(as_nanoarrow_array(1:10))
**inference_nanoarrow_ptype_extension**

*Implement Arrow extension types*

---

**Description**

Implement Arrow extension types

**Usage**

```r
infer_nanoarrow_ptype_extension(
    extension_spec,
    x,
    ..., 
    warn_unregistered = TRUE
)
```

```r
convert_array_extension(
    extension_spec,
    array,
    to,
    ..., 
    warn_unregistered = TRUE
)
```

```r
as_nanoarrow_array_extension(extension_spec, x, ..., schema = NULL)
```

**Arguments**

extension_spec  An extension specification inheriting from ‘nanoarrow_extension_spec’.

x, array, to, schema, ...

Passed from `infer_nanoarrow_ptype()`, `convert_array()`, `as_nanoarrow_array()`, and/or `as_nanoarrow_array_stream()`.

warn_unregistered

Use FALSE to infer/convert based on the storage type without a warning.

**Value**

- `infer_nanoarrow_ptype_extension()`: The R vector prototype to be used as the default conversion target.
- `convert_array_extension()`: An R vector of type to.
- `as_nanoarrow_array_extension()`: A `nanoarrow_array` of type schema.
Description

Create a new array or from an existing array, modify one or more parameters. When importing an array from elsewhere, `nanoarrow_array_set_schema()` is useful to attach the data type information to the array (without this information there is little that nanoarrow can do with the array since its content cannot be otherwise interpreted). `nanoarrow_array_modify()` can create a shallow copy and modify various parameters to create a new array, including setting children and buffers recursively. These functions power the `$<-` operator, which can modify one parameter at a time.

Usage

```r
nanoarrow_array_init(schema)

nanoarrow_array_set_schema(array, schema, validate = TRUE)

nanoarrow_array_modify(array, new_values, validate = TRUE)
```

Arguments

- **schema**
  - A `nanoarrow_schema` to attach to this array.
- **array**
  - A `nanoarrow_array`.
- **validate**
  - Use `FALSE` to skip validation. Skipping validation may result in creating an array that will crash R.
- **new_values**
  - A named list() of values to replace.

Value

- `nanoarrow_array_init()` returns a possibly invalid but initialized array with a given schema.
- `nanoarrow_array_set_schema()` returns `array`, invisibly. Note that `array` is modified in place by reference.
- `nanoarrow_array_modify()` returns a shallow copy of `array` with the modified parameters such that the original array remains valid.

Examples

```r
nanoarrow_array_init(na_string())

# Modify an array using $ and <-
array <- as_nanoarrow_array(1:5)
array$length <- 4
as.vector(array)

# Modify potentially more than one component at a time
```
array <- as_nanoarrow_array(1:5)
as.vector(nanoarrow_array_modify(array, list(length = 4)))

# Attach a schema to an array
array <- as_nanoarrow_array(-1L)
nanoarrow_array_set_schema(array, na_uint32())
as.vector(array)

---

nanoarrow_buffer_init Create and modify nanoarrow buffers

Description

Create and modify nanoarrow buffers

Usage

nanoarrow_buffer_init()
nanoarrow_buffer_append(buffer, new_buffer)
convert_buffer(buffer, to = NULL)

Arguments

buffer, new_buffer
  nanoarrow_buffers.
to
  A target prototype object describing the type to which array should be converted, or NULL to use the default conversion as returned by infer_nanoarrow_ptype(). Alternatively, a function can be passed to perform an alternative calculation of the default ptype as a function of array and the default inference of the prototype.

Value

- nanoarrow_buffer_init(): An object of class 'nanoarrow_buffer'
- nanoarrow_buffer_append(): Returns buffer, invisibly. Note that buffer is modified in place by reference.

Examples

buffer <- nanoarrow_buffer_init()
nanoarrow_buffer_append(buffer, 1:5)
array <- nanoarrow_array_modify(
  nanoarrow_array_init(na_int32()),
  list(length = 5, buffers = list(NULL, buffer)))
nanoarrow_extension_array

Create Arrow extension arrays

Description

Create Arrow extension arrays

Usage

nanoarrow_extension_array(
  storage_array,
  extension_name,
  extension_metadata = NULL
)

Arguments

storage_array A nanoarrow_array.
extension_name For na_extension(), the extension name. This is typically namespaced separated by dots (e.g., arrow.r.vctrs).
extension_metadata A string or raw vector defining extension metadata. Most Arrow extension types define extension metadata as a JSON object.

Value

A nanoarrow_array with attached extension schema.

Examples

nanoarrow_extension_array(1:10, "some_ext", '{"key": "value"}')
nanoarrow_extension_spec

Register Arrow extension types

Description

Register Arrow extension types

Usage

nanoarrow_extension_spec(data = list(), subclass = character())

register_nanoarrow_extension(extension_name, extension_spec)

unregister_nanoarrow_extension(extension_name)

resolve_nanoarrow_extension(extension_name)

Arguments

data               Optional data to include in the extension type specification
subclass           A subclass for the extension type specification. Extension methods will dispatch on this object.
extension_name     An Arrow extension type name (e.g., arrow.r.vctrs)
extension_spec     An extension specification inheriting from 'nanoarrow_extension_spec'.

Value

• nanoarrow_extension_spec() returns an object of class 'nanoarrow_extension_spec'.
• register_nanoarrow_extension() returns extension_spec, invisibly.
• unregister_nanoarrow_extension() returns extension_name, invisibly.
• resolve_nanoarrow_extension() returns an object of class 'nanoarrow_extension_spec' or NULL if the extension type was not registered.

Examples

nanoarrow_extension_spec("mynamespace.mytype", subclass = "mypackage.mytype_spec")
nanoarrow_pointer_is_valid

Description

The nanoarrow_schema, nanoarrow_array, and nanoarrow_array_stream classes are represented in R as external pointers (EXTPTRSXP). When these objects go out of scope (i.e., when they are garbage collected or shortly thereafter), the underlying object’s release() callback is called if the underlying pointer is non-null and if the release() callback is non-null.

Usage

nanoarrow_pointer_is_valid(ptr)
nanoarrow_pointer_addr_dbl(ptr)
nanoarrow_pointer_addr_chr(ptr)
nanoarrow_pointer_addr_pretty(ptr)
nanoarrow_pointer_release(ptr)
nanoarrow_pointer_move(ptr_src, ptr_dst)
nanoarrow_pointer_export(ptr_src, ptr_dst)
nanoarrow_allocate_schema()
nanoarrow_allocate_array()
nanoarrow_allocate_array_stream()
nanoarrow_pointer_set_protected(ptr_src, protected)

Arguments

ptr, ptr_src, ptr_dst
An external pointer to a struct ArrowSchema, struct ArrowArray, or struct ArrowArrayStream.

protected
An object whose scope must outlive that of ptr. This is useful for array streams since at least two specifications involving the array stream specify that the stream is only valid for the lifecycle of another object (e.g., an AdbcStatement or OGR-Dataset).
When interacting with other C Data Interface implementations, it is important to keep in mind that the R object wrapping these pointers is always passed by reference (because it is an external pointer) and may be referred to by another R object (e.g., an element in a list() or as a variable assigned in a user’s environment). When importing a schema, array, or array stream into nanoarrow this is not a problem: the R object takes ownership of the lifecycle and memory is released when the R object is garbage collected. In this case, one can use `nanoarrow_pointer_move()` where `ptr_dst` was created using `nanoarrow_allocate_*()`.

The case of exporting is more complicated and as such has a dedicated function, `nanoarrow_pointer_export()`, that implements different logic schemas, arrays, and array streams:

- Schema objects are (deep) copied such that a fresh copy of the schema is exported and made the responsibility of some other C data interface implementation.
- Array objects are exported as a shell around the original array that preserves a reference to the R object. This ensures that the buffers and children pointed to by the array are not copied and that any references to the original array are not invalidated.
- Array stream objects are moved: the responsibility for the object is transferred to the other C data interface implementation and any references to the original R object are invalidated. Because these objects are mutable, this is typically what you want (i.e., you should not be pulling arrays from a stream accidentally from two places).

If you know the lifecycle of your object (i.e., you created the R object yourself and never passed references to it elsewhere), you can slightly more efficiently call `nanoarrow_pointer_move()` for all three pointer types.

**Value**

- `nanoarrow_pointer_is_valid()` returns TRUE if the pointer is non-null and has a non-null release callback.
- `nanoarrow_pointer_addr_dbl()` and `nanoarrow_pointer_addr_chr()` return pointer representations that may be helpful to facilitate moving or exporting nanoarrow objects to other libraries.
- `nanoarrow_pointer_addr_pretty()` gives a pointer representation suitable for printing or error messages.
- `nanoarrow_pointer_release()` returns `ptr`, invisibly.
- `nanoarrow_pointer_move()` and `nanoarrow_pointer_export()` return `ptr_dst`, invisibly.
- `nanoarrow_allocate_array()`, `nanoarrow_allocate_schema()`, and `nanoarrow_allocate_array_stream()` return an array, a schema, and an array stream, respectively.
nanoarrow_version

 nanoarrow_version

  Underlying 'nanoarrow' C library build

Description

Underlying 'nanoarrow' C library build

Usage

nanoarrow_version(runtime = TRUE)

Arguments

runtime

  Compare TRUE and FALSE values to detect a possible ABI mismatch.

Value

A string identifying the version of nanoarrow this package was compiled against.

Examples

nanoarrow_version()

na_type

 na_type

  Create type objects

Description

In nanoarrow, types, fields, and schemas are all represented by a 

  nanoarrow_schema. These functions are convenience constructors to create 

  these objects in a readable way. Use na_type() to construct types 

  based on the constructor name, which is also the name that prints/is returned by 

  nanoarrow_schema_parse().

Usage

na_type(
  type_name,
  byte_width = NULL,
  unit = NULL,
  timezone = NULL,
  column_types = NULL,
  item_type = NULL,
  key_type = NULL,
  value_type = NULL,
  index_type = NULL,
ordered = NULL,
list_size = NULL,
keys_sorted = NULL,
storage_type = NULL,
extension_name = NULL,
extension_metadata = NULL,
nullable = NULL
)

na_na(nullable = TRUE)

na_bool(nullable = TRUE)

na_int8(nullable = TRUE)

na_uint8(nullable = TRUE)

na_int16(nullable = TRUE)

na_uint16(nullable = TRUE)

na_int32(nullable = TRUE)

na_uint32(nullable = TRUE)

na_int64(nullable = TRUE)

na_uint64(nullable = TRUE)

na_half_float(nullable = TRUE)

na_float(nullable = TRUE)

na_double(nullable = TRUE)

na_string(nullable = TRUE)

na_large_string(nullable = TRUE)

na_binary(nullable = TRUE)

na_large_binary(nullable = TRUE)

na_fixed_size_binary(byte_width, nullable = TRUE)

na_date32(nullable = TRUE)

na_date64(nullable = TRUE)
na_time32(unit = c("ms", "s"), nullable = TRUE)
na_time64(unit = c("us", "ns"), nullable = TRUE)
na_duration(unit = c("ms", "s", "us", "ns"), nullable = TRUE)
na_interval_months(nullable = TRUE)
na_interval_day_time(nullable = TRUE)
na_interval_month_day_nano(nullable = TRUE)
na_timestamp(unit = c("us", "ns", "s", "ms"), timezone = "", nullable = TRUE)
na_decimal128(precision, scale, nullable = TRUE)
na_decimal256(precision, scale, nullable = TRUE)
na_struct(column_types = list(), nullable = FALSE)
na_sparse_union(column_types = list())
na_dense_union(column_types = list())
na_list(item_type, nullable = TRUE)
na_large_list(item_type, nullable = TRUE)
na_fixed_size_list(item_type, list_size, nullable = TRUE)
na_map(key_type, item_type, keys_sorted = FALSE, nullable = TRUE)
na_dictionary(value_type, index_type = na_int32(), ordered = FALSE)
na_extension(storage_type, extension_name, extension_metadata = "")

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>type_name</td>
<td>The name of the type (e.g., &quot;int32&quot;). This form of the constructor is useful for writing tests that loop over many types.</td>
</tr>
<tr>
<td>byte_width</td>
<td>For <code>na_fixed_size_binary()</code>, the number of bytes occupied by each item.</td>
</tr>
<tr>
<td>unit</td>
<td>One of 's' (seconds), 'ms' (milliseconds), 'us' (microseconds), or 'ns' (nanoseconds).</td>
</tr>
<tr>
<td>timezone</td>
<td>A string representing a timezone name. The empty string &quot;&quot; represents a naive point in time (i.e., one that has no associated timezone).</td>
</tr>
<tr>
<td>column_types</td>
<td>A list() of <code>nanarrow_schemas</code>.</td>
</tr>
</tbody>
</table>
item_type For \texttt{na\_list()}, \texttt{na\_large\_list()}, \texttt{na\_fixed\_size\_list()}, and \texttt{na\_map()}, the \texttt{nanoarrow\_schema} representing the item type.

key_type The \texttt{nanoarrow\_schema} representing the \texttt{na\_map()} key type.

value_type The \texttt{nanoarrow\_schema} representing the \texttt{na\_dictionary()} or \texttt{na\_map()} value type.

index_type The \texttt{nanoarrow\_schema} representing the \texttt{na\_dictionary()} index type.

ordered Use \texttt{TRUE} to assert that the order of values in the dictionary are meaningful.

list_size The number of elements in each item in a \texttt{na\_fixed\_size\_list()}.

keys_sorted Use \texttt{TRUE} to assert that keys are sorted.

storage_type For \texttt{na\_extension()}, the underlying value type.

extension_name For \texttt{na\_extension()}, the extension name. This is typically namespaced separated by dots (e.g., arrow.r.vctrs).

extension_metadata A string or raw vector defining extension metadata. Most Arrow extension types define extension metadata as a JSON object.

nullable Use \texttt{FALSE} to assert that this field cannot contain null values.

precision The total number of digits representable by the decimal type

scale The number of digits after the decimal point in a decimal type

Value A \texttt{nanoarrow\_schema}

Examples
\begin{verbatim}
na\_int32()
na\_struct(list(col1 = na\_int32()))
\end{verbatim}

\underline{na\_vctrs} \hspace{1cm} \textit{Vctrs extension type}

Description

The Arrow format provides a rich type system that can handle most R vector types; however, many R vector types do not roundtrip perfectly through Arrow memory. The vctrs extension type uses \texttt{vctrs::vec\_data()}, \texttt{vctrs::vec\_restore()}, and \texttt{vctrs::vec\_ptype()} in calls to \texttt{as\_nanoarrow\_array()} and \texttt{convert\_array()} to ensure roundtrip fidelity.

Usage
\begin{verbatim}
na\_vctrs(ptype, storage\_type = \texttt{NULL})
\end{verbatim}
**Arguments**

ptype

A `vctrs` prototype as returned by `vctrs::vec_ptype()`. The prototype can be of arbitrary size, but a zero-size vector is sufficient here.

storage_type

For `na_extension()`, the underlying value type.

**Value**

A `nanoarrow_schema`.

**Examples**

```r
cmp <- as.POSIXlt("2000-01-02 03:45", tz = "UTC")
array <- as_nanoarrow_array(vctr, schema = na_vctrs(vctr))
infer_nanoarrow_ptype(array)
convert_array(array)
```

**Description**

Reads connections, file paths, URLs, or raw vectors of serialized Arrow data. Arrow documentation typically refers to this format as "Arrow IPC", since its origin was as a means to transmit tables between processes (e.g., multiple R sessions). This format can also be written to and read from files or URLs and is essentially a high performance equivalent of a CSV file that does a better job maintaining types.

**Usage**

```r
read_nanoarrow(x, ..., lazy = FALSE)
```

**Arguments**

x

A `raw()` vector, connection, or file path from which to read binary data. Common extensions indicating compression (.gz, .bz2, .zip) are automatically uncompressed.

...  

Currently unused.

lazy

By default, `read_nanoarrow()` will read and discard a copy of the reader's schema to ensure that invalid streams are discovered as soon as possible. Use `lazy = TRUE` to defer this check until the reader is actually consumed.
Details

The nanoarrow package does not currently have the ability to write serialized IPC data: use `arrow::write_ipc_stream()` to write data from R, or use the equivalent writer from another Arrow implementation in Python, C++, Rust, JavaScript, Julia, C#, and beyond.

The media type of an Arrow stream is `application/vnd.apache.arrow.stream` and the recommended file extension is `.arrows`.

Value

A `nanoarrow_array_stream`

Examples

```r
as.data.frame(read_nanoarrow(example_ipc_stream()))
```
Index

array, 18
array stream, 18
array_stream_set_finalizer, 2
arrays, 4
arrow::field(), 5
arrow::RecordBatchReader, 4
arrow::schema(), 5
arrow::struct(), 5
arrow::write_ipc_stream(), 24
diffftime(), 9, 11
double(), 9, 11
example_ipc_stream (read_nanoarrow), 23
factor(), 9
hms::hms(), 9, 11
infer_nanoarrow_ptype, 11
infer_nanoarrow_ptype(), 8, 10, 12, 14
infer_nanoarrow_ptype_extension, 12
infer_nanoarrow_schema
   (as_nanoarrow_schema), 5
infer_nanoarrow_schema(), 3, 4
integer(), 9, 11
list(), 8
logical(), 9, 11
na_binary (na_type), 19
na_bool (na_type), 19
na_date32 (na_type), 19
na_date64 (na_type), 19
na_decimal128 (na_type), 19
na_decimal256 (na_type), 19
na_dense_union (na_type), 19
na_dictionary (na_type), 19
na_dictionary(), 22
na_double (na_type), 19
na_duration (na_type), 19
na_extension (na_type), 19
na_extension(), 15, 22, 23
na_fixed_size_binary (na_type), 19
na_fixed_size_binary(), 21
na_fixed_size_list (na_type), 19
na_fixed_size_list(), 22
na_float (na_type), 19
na_half_float (na_type), 19
na_int16 (na_type), 19
na_int32 (na_type), 19
na_int64 (na_type), 19
na_int8 (na_type), 19
na_interval_day_time (na_type), 19
na_interval_month_day_nano (na_type), 19
na_interval_months (na_type), 19
na_large_binary (na_type), 19
na_large_list (na_type), 19
na_large_list(), 22
na_large_string (na_type), 19
na_list (na_type), 19
na_list(), 22
na_map (na_type), 19
na_map(), 22
na_na (na_type), 19
na_sparse_union (na_type), 19
na_string (na_type), 19
na_struct (na_type), 19
na_time32 (na_type), 19
na_time64 (na_type), 19
na_timestamp (na_type), 19
na_type, 19
na_type(), 19
na_uint16 (na_type), 19
na_uint32 (na_type), 19
na_uint64 (na_type), 19
na_uint8 (na_type), 19
na_vctrs(), 22
nanoarrow_allocate_array (nanoarrow_pointer_is_valid), 17
nanoarrow_allocate_array_stream (nanoarrow_pointer_is_valid), 17
nanoarrow_allocate_schema (nanoarrow_pointer_is_valid), 17
nanoarrow_array, 8, 10–13, 15, 17
nanoarrow_array_init, 13
nanoarrow_array_modify (nanoarrow_array_init), 13
nanoarrow_array_set_schema (nanoarrow_array_init), 13
nanoarrow_array_stream, 2, 8, 10, 11, 17, 24
nanoarrow_buffer, 14
nanoarrow_buffer_append (nanoarrow_buffer_init), 14
nanoarrow_buffer_init, 14
nanoarrow_extension_array, 15
nanoarrow_extension_spec, 16
nanoarrow_pointer_addr_chr (nanoarrow_pointer_is_valid), 17
nanoarrow_pointer_addr_db1 (nanoarrow_pointer_is_valid), 17
nanoarrow_pointer_addr_pretty (nanoarrow_pointer_is_valid), 17
nanoarrow_pointer_export (nanoarrow_pointer_is_valid), 17
nanoarrow_pointer_export(), 18
nanoarrow_pointer_is_valid, 17
nanoarrow_pointer_move (nanoarrow_pointer_is_valid), 17
nanoarrow_pointer_move(), 18
nanoarrow_pointer_release (nanoarrow_pointer_is_valid), 17
nanoarrow_pointer_set_protected (nanoarrow_pointer_is_valid), 17
nanoarrow_schema, 7, 8, 10, 11, 13, 17, 19, 21–23
nanoarrow_schema_modify (as_nanoarrow_schema), 5
nanoarrow_schema_parse (as_nanoarrow_schema), 5
nanoarrow_schema_parse(), 19
nanoarrow_vctr (as_nanoarrow_vctr), 6
nanoarrow_version, 19
read_nanoarrow, 23
register_nanoarrow_extension (nanoarrow_extension_spec), 16
resolve_nanoarrow_extension (nanoarrow_extension_spec), 16
schema, 3, 4, 18
units(), 9
unregister_nanoarrow_extension (nanoarrow_extension_spec), 16
vctrs::list_of(), 9, 11
vctrs::unspecified(), 9, 11
INDEX

vctrs::vec_data(). 22
vctrs::vec_type(). 22, 23
vctrs::vec_restore(). 22