

Package: altarr (via r-universe)

October 15, 2026

Title Lazy Chunked Arrays as ALTREP Vectors

Version 0.1.0.9000

Description A prototype exploring how far base R array indexing can drive planned reads of remote, chunked, encoded arrays when the array is an ALTREP vector with a 'dim' attribute and no class. Chunks come from a user-supplied fetch function, so any store (object storage, 'Zarr', 'GDAL') can sit behind it.

License MIT + file LICENSE

Encoding UTF-8

Depends R (>= 3.6.0)

Imports stats

Suggests knitr, rmarkdown

VignetteBuilder knitr

Config/roxygen2/version 8.1.0

URL <https://github.com/hypertidy/altarr>,
<https://hypertidy.github.io/altarr/>

BugReports <https://github.com/hypertidy/altarr/issues>

Roxygen list(markdown = TRUE)

Repository <https://hypertidy.r-universe.dev>

Date/Publication 2026-10-03 02:10:15 UTC

RemoteUrl <https://github.com/hypertidy/altarr>

RemoteRef HEAD

RemoteSha 8b962d5d807535929bda43f796d23f05a0b44ace

Contents

altarr	2
altarr_contract	3
altarr_example_zarr	5

altarr_extract	6
altarr_plan	7
altarr_reset	7
altarr_stats	8
altarr_zarr_v2	9
is_altarr	10
Index	11

altarr	<i>Create a lazily-read chunked array</i>
--------	---

Description

Returns an ordinary double, integer or logical vector with a dim attribute and no class. It is an ALTREP object: values are obtained chunk by chunk from fetch only when base R asks for them.

Usage

```
altarr(  
  dim,  
  chunk,  
  fetch,  
  dimnames = NULL,  
  type = c("double", "integer", "logical")  
)
```

Arguments

- dim integer dimensions.
- chunk integer chunk shape (recycled to length(dim)).
- fetch a function taking an integer matrix of 0-based chunk coordinates (one row per chunk, one column per dimension) and returning a list of vectors, one per row, in column-major order and clipped at the array edge. Chunks should be of the array's type; other numeric or logical vectors are coerced as as.double(), as.integer() or as.logical() would.
- dimnames optional dimnames.
- type the array's type: "double", "integer" or "logical".

Value

a lazy array.

Examples

```
## a 10 x 7 array in 4 x 3 chunks; each value is its own position
d <- c(10L, 7L); cs <- c(4L, 3L)
fetch <- function(chunks) {
  lapply(seq_len(nrow(chunks)), function(r) {
    start <- chunks[r, ] * cs + 1L
    end <- pmin(start + cs - 1L, d) # clipped at the edge
    i <- start[1]:end[1]; j <- start[2]:end[2]
    as.vector(outer(i, (j - 1) * d[1], "+"))
  })
}
x <- altarr(d, cs, fetch)
x[c(1, 70)]
x[cbind(10, 7)]
altarr_stats(x)[c("fetch_calls", "chunks_fetched")]
## the same chunks as an integer array (values are coerced)
xi <- altarr(d, cs, fetch, type = "integer")
typeof(xi)
sum(xi)
```

altarr_contract

The fetch contract, and what altarr does with it

Description

What a fetch function and the store behind it must guarantee, what altarr does on its side, and what base R does with a lazy array.

The fetch function

- It is called as `fetch(chunks)`, where `chunks` is an integer matrix with one row per chunk and one column per dimension, holding 0-based chunk coordinates. Rows may arrive in any order.
- It returns a list with one vector per row, in row order. Each vector holds that chunk's values in column-major (R) order, clipped at the array edge: an edge chunk is shorter, never padded.
- Each vector should be of the array's type ("double", "integer" or "logical"); other numeric or logical vectors are coerced as `as.double()`, `as.integer()` or `as.logical()` would.
- One call is one round trip. altarr asks for as many chunks per call as it can plan, so a fetch that reads its chunks concurrently (object storage, GDAL) gets the full benefit.
- It runs on R's main thread. Retries, timeouts and concurrency belong inside it, and any code it runs on other threads must not call the R API.
- An error in fetch is an R error raised from wherever the read happened: inside `[]`, `sum()`, `mean()` and so on.

Sources must be pinned

R assumes a vector's values never change. altarr re-fetches chunks after they are evicted from the cache, and a saved recipe re-reads its source when loaded later. So the source behind fetch must be immutable: a snapshot (an Icechunk snapshot id, a versioned object), a fixed set of chunk references (kerchunk, blocklist), or content-addressed chunks. A mutable store can silently return different values for the same element within one session.

Saving arrays (recipes)

saveRDS() writes the recipe (dim, chunk shape, type and the fetch function), never the values. The fetch function travels with its environment, which brings two traps:

- A fetch defined at top level refers to global variables, and the global environment is saved as a reference, not its contents. The recipe loads, but reading fails in a new session with "object not found".
- A fetch built inside a function carries that function's environment. Arguments are promises until used, so an unforced argument still points back at the caller's variable.

Build fetch functions in a small factory that force()s what it needs: a path or URL and a few parameters, not data. See the example.

What base R does with a lazy array

Measured on R 4.3 with materialization refused, so every attempt shows:

Planned (one fetch call per batch of chunks): `x[i]` (including logical and negative subscripts), `x[cbind(i, j, k)]`, `altarr_extract()`, `sum()`, `min()`, `max()`, `mean()`, `prod()`, `anyNA()` (which stops at the first NA), `which()` on logical arrays, and `str()`.

Element by element (correct, lazy, one chunk per round trip): `x[i, j, k]` (until R offers an array-subset hook), `x[[i]]`, `head()` on an array, and `is.na()`, so also the `is.na()` pass in `mean(na.rm = TRUE)`, `summary()` and `quantile()`.

Materializes the whole array: printing, arithmetic and comparison (`x + 1`, `x > 0`), maths functions, `range()`, `c()`, `aperm()`, `apply()`, `colSums()` and relatives, and assignment (`x[1] <- v`). Refused above `getOption("altarr.max_materialize")` values.

Reads nothing: `dim()`, `length()`, `dimnames<-`, `identical()`, `saveRDS()`, copying (`y <- x`), and `as.vector()`, which stays lazy.

Options

`altarr.cache_bytes` chunk cache budget per array, in bytes (default 256 MiB). Least recently used chunks are evicted first; a copy shares its original's cache.

`altarr.batch_chunks` chunks per fetch call for whole-array passes: reductions, scans and materialization (default 64).

`altarr.max_materialize` the most values a whole-array operation may materialize (default 1e6). It decides only whether materializing is allowed; the operation triggers it, not the size.

Known limits

- Lazy reads, not lazy computation: arithmetic materializes.
- `x[i, j, k]` reads element by element until R's `ArraySubset` offers its subscripts to `ALTREP`; `altarr_extract()` shows what that would give.
- Dimension names must be character, as for any R array.

Examples

```
## a fetch factory: the recipe carries a path and parameters, not data
make_fetch <- function(path, dim, chunk) {
  force(path); force(dim); force(chunk)
  function(chunks) {
    lapply(seq_len(nrow(chunks)), function(r) {
      start <- chunks[r, ] * chunk + 1L
      n <- pmin(chunk, dim - start + 1L) # clipped at the edge
      rep(nchar(path), prod(n))         # stand-in for reading 'path'
    })
  }
}
d <- c(40L, 30L); cs <- c(16L, 16L)
x <- altarr(d, cs, make_fetch("s3://bucket/array.zarr", d, cs))
x[cbind(c(1, 40), c(1, 30))]
f <- tempfile(fileext = ".rds")
saveRDS(x, f)
file.size(f) # a few hundred bytes
```

altarr_example_zarr	<i>Write a small example Zarr v2 store</i>
---------------------	--

Description

Writes a zlib-compressed, C-order Zarr v2 array of doubles into `dir` and returns the path, ready for `altarr_zarr_v2()`. The store is made on demand rather than shipped, because Zarr v2 metadata files (`.zarray`) are hidden files, which R packages should not carry.

Usage

```
altarr_example_zarr(
  dir = tempfile("altarr-example-", fileext = ".zarr"),
  dim = c(70L, 50L, 30L),
  chunk = c(20L, 16L, 7L)
)
```

Arguments

dir	directory to write the store into (created if needed).
dim	R dimensions of the array (the Zarr shape is the reverse).
chunk	R chunk shape (the Zarr chunk shape is the reverse). Edge chunks are padded in the store, as Zarr requires.

Details

Every value is its own 0-based position in R's column-major order, so a result can be checked by eye: in the array returned by `altarr_zarr_v2()`, element $x[i, j, k]$ is $(i - 1) + \text{dim}[1] * (j - 1) + \text{dim}[1] * \text{dim}[2] * (k - 1)$, and $x[n]$ is $n - 1$.

Value

the store path, invisibly.

Examples

```
path <- altarr_example_zarr()
x <- altarr_zarr_v2(path)
dim(x)
x[c(1, 2, 71)] # 0, 1, 70
```

<code>altarr_extract</code>	<i>Rectangular (hyperslab) extraction with one planned fetch</i>
-----------------------------	--

Description

Same semantics as `x[i, j, k, drop = drop]`, but the per-dimension subscripts are handed to the ALTREP object whole: the touched chunks are planned as a cartesian product and fetched in one call. This is what base R's `ArraySubset` could do if it offered its normalized subscripts to an ALTREP method.

Usage

```
altarr_extract(x, ..., drop = TRUE)
```

Arguments

<code>x</code>	an altarr array
<code>...</code>	one subscript per dimension (missing means all)
<code>drop</code>	as for <code>[]</code>

Value

an ordinary (not lazy) array or vector.

Examples

```
x <- altarr_zarr_v2(altarr_example_zarr())
r <- altarr_extract(x, 1:40, 1:20, 1:3)
dim(r)
identical(r, x[1:40, 1:20, 1:3])
altarr_extract(x, 1, 1:3, 1) # dropped to a vector, as `[]` would
```

altarr_plan

*The chunk plan for a subscript, as a data frame***Description**

One row per chunk that `x[i, j, k]` would touch: 0-based chunk coordinates, plus each chunk's 1-based start and its extent along every dimension. The plan is a table; the array is what assembling it gives.

Usage

```
altarr_plan(x, ...)
```

Arguments

`x` an altarr array
`...` one subscript per dimension (missing means all)

Value

a data frame, one row per chunk.

Examples

```
x <- altarr_zarr_v2(altarr_example_zarr())
altarr_plan(x, 1:40, 1:20, 1:3)
altarr_plan(x, 70, , 1) # a missing subscript means the whole dimension
```

altarr_reset

*Reset counters, and optionally drop the chunk cache***Description**

Reset counters, and optionally drop the chunk cache

Usage

```
altarr_reset(x, cache = TRUE)
```

Arguments

`x` an altarr array
`cache` drop cached chunks too?

Value

`x`, invisibly.

Examples

```
x <- altarr_zarr_v2(altarr_example_zarr())
x[1:5]
x[1:5] # the second read comes from the cache
altarr_stats(x)[c("extract_subset", "fetch_calls", "chunks_cached")]
altarr_reset(x)
altarr_stats(x)[c("fetch_calls", "chunks_cached")]
```

altarr_stats

Counters and cache state for an altarr array

Description

`elt` counts per-element reads (the `x[i, j, k]` path), `extract_subset` counts planned subset calls (`x[i]`, `x[cbind(...)]`), `hyperslab` counts `altarr_extract()` calls, `fetch_calls` counts calls to the fetch function (round trips) and `chunks_fetched` the chunks they returned. `reduce` counts whole-array `sum()`, `min()` and `max()` calls, which stream the chunk grid in batches of `getOption("altarr.batch_chunks", 64)` chunks (one fetch call per batch) without caching what they read. `region` counts contiguous-run reads from functions R iterates by region (`mean()`, `prod()`, `anyNA()`, and reductions on R's wrapper class). `evictions` counts chunks dropped from the cache to keep it within its budget. Also reported: `cache_bytes` (bytes currently cached) and `cache_budget` (`getOption("altarr.cache_bytes")`, default 256 MiB). The cache is least recently used first, and shared by an array and its copies.

Usage

```
altarr_stats(x)
```

Arguments

`x` an altarr array

Value

a named numeric vector of counters and cache state.

Examples

```
x <- altarr_zarr_v2(altarr_example_zarr())
x[1:5]
x[1:5] # the second read comes from the cache
altarr_stats(x)[c("extract_subset", "fetch_calls", "chunks_cached")]
altarr_reset(x)
altarr_stats(x)[c("fetch_calls", "chunks_cached")]
```

altarr_zarr_v2

A lazy array over a local Zarr v2 store

Description

A deliberately small reader, to show that a real chunked, compressed store fits behind the fetch contract in a few lines. Handles Zarr v2 arrays with dtype <f8, <f4, <i4, <i2, |i1, <u2, |u1 or |b1, compressor null or zlib, order C or F, and missing chunk files (read as fill_value). Zarr's padded edge chunks are clipped.

This is a demonstration, not a Zarr implementation: no Zarr v3, no remote stores, no other codecs. For real stores, use a full Zarr reader and hand its chunk reads to [altarr\(\)](#) as the fetch function.

Usage

```
altarr_zarr_v2(path, unpack = TRUE)
```

Arguments

path	directory of the Zarr v2 array (containing .zarray)
unpack	apply CF scale_factor, add_offset, _FillValue and missing_value from .zattrs?

Details

The array's type follows the dtype: floats give a double array, integers an integer array, |b1 a logical array. With unpack = TRUE, CF packing attributes found in .zattrs are applied inside the fetch: values equal to _FillValue or missing_value become NA, and if scale_factor or add_offset is present the array is double, holding the unpacked values. The Zarr fill_value itself is not treated as missing.

A C-order Zarr array of shape (s1, ..., sn) is returned with R dim c(sn, ..., s1): C-order bytes are already column-major for the reversed shape, so no transposition is ever done.

Value

a lazy array, see [altarr\(\)](#)

Examples

```
path <- altarr_example_zarr()
x <- altarr_zarr_v2(path)
dim(x) # the Zarr shape, reversed
x[cbind(70, 50, 30)]
str(x)
```

is_altarr	<i>Is this a (still lazy-capable) altarr array?</i>
-----------	---

Description

Is this a (still lazy-capable) altarr array?

Usage

```
is_altarr(x)
```

Arguments

x	object
---	--------

Value

TRUE if x is an altarr array, or R's wrapper around one.

Examples

```
x <- altarr_zarr_v2(altarr_example_zarr())
is_altarr(x)
is_altarr(x[1:3]) # a subset is an ordinary vector
is_altarr(array(1:4, c(2, 2)))
```

Index

altarr, [2](#)
altarr(), [9](#)
altarr_contract (altarr_contract), [3](#)
altarr_contract, [3](#)
altarr_example_zarr, [5](#)
altarr_extract, [6](#)
altarr_extract(), [4](#), [5](#), [8](#)
altarr_plan, [7](#)
altarr_reset, [7](#)
altarr_stats, [8](#)
altarr_zarr_v2, [9](#)
altarr_zarr_v2(), [5](#)

is_altarr, [10](#)