Package 'shard'

April 3, 2026

Type Package

Title Deterministic, Zero-Copy Parallel Execution for R

Version 0.1.0

Description Provides a parallel execution runtime for R that emphasizes deterministic memory behavior and efficient handling of large shared inputs. 'shard' enables zero-copy parallel reads via shared/memory-mapped segments, encourages explicit output buffers to avoid large result aggregation, and supervises worker processes to mitigate memory drift via controlled recycling. Diagnostics report peak memory usage, end-of-run memory return, and hidden copy/materialization events to support reproducible performance benchmarking.

License MIT + file LICENSE

Encoding UTF-8

Language en-US

Depends R (>= 4.1.0)

Imports methods, parallel, stats, tools, utils

Suggests knitr, pkgload, rmarkdown, testthat (>= 3.0.0), ps, jsonlite, tibble, withr

VignetteBuilder knitr

RoxygenNote 7.3.3

NeedsCompilation yes

URL https://bbuchsbaum.github.io/shard/, https://github.com/bbuchsbaum/shard

BugReports https://github.com/bbuchsbaum/shard/issues

SystemRequirements POSIX shared memory (optional), memory-mapped files

Config/testthat/edition 3

Author Bradley Buchsbaum [aut, cre, cph]

Maintainer Bradley Buchsbaum <brad.buchsbaum@gmail.com>

Repository CRAN

Date/Publication 2026-04-03 08:40:02 UTC

Contents

`adapter` *Adapter Registry for Class-Specific Deep Sharing*

Description

Register custom traversal logic for specific classes during deep sharing operations. Adapters allow fine-grained control over how objects are decomposed and reconstructed.

Details

The adapter registry provides a way to customize how specific classes are handled during deep sharing. Instead of generic slot traversal for S4 objects or element-wise traversal for lists, you can provide custom functions to:

1. Extract the shareable children from an object (`children`)
2. Reconstruct the object from shared children (`replace`)

This is useful for:

- Complex S4 objects where only certain slots should be shared
- S3 objects with internal structure that differs from list structure
- Objects with accessors that should be used instead of direct slot access

See Also

`share` for the main sharing function that uses adapters.

`affinity` *CPU Affinity + mmap Advice (Advanced)*

Description

These controls are opt-in and best-effort. On unsupported platforms, they safely no-op (returning FALSE).

`affinity_supported` *Check whether CPU affinity is supported*

Description

Currently supported on Linux only.

Usage

```
affinity_supported()
```

Value

A logical scalar indicating platform support.

Examples

```
affinity_supported()
```

altrep *ALTREP Shared Vectors*

Description

ALTREP-backed zero-copy vectors for shared memory.

Details

These functions create ALTREP (Alternative Representation) vectors that are backed by shared memory segments. The key benefits are:

- **Zero-copy subsetting**: Contiguous subsets return views into the same shared memory, not copies.
- **Diagnostics**: Track when data pointers are accessed or when vectors are materialized (copied to standard R vectors).
- **Read-only protection**: Optionally prevent write access to protect shared data.

Supported types: integer, double/numeric, logical, raw.

arena *Arena Semantic Scope*

Description

Semantic scope for scratch memory that signals temporary data should not accumulate. Enables memory-conscious parallel execution.

Evaluates an expression in a semantic scope that signals scratch memory usage. This enables memory-conscious execution where temporaries are expected to be reclaimed after the scope exits.

Usage

```
arena(
  expr,
  strict = FALSE,
  escape_threshold = .arena_escape_threshold,
  gc_after = strict,
  diagnostics = FALSE
)
```

Arguments

`expr`	An expression to evaluate within the arena scope.
`strict`	Logical. If TRUE, enables strict mode which: • Warns if large objects (> 1MB by default) escape the scope • Triggers garbage collection after scope exit • Tracks memory growth for diagnostics Default is FALSE for compatibility and performance.
`escape_threshold`	Numeric. Size in bytes above which returned objects trigger a warning in strict mode. Default is 1MB (1048576 bytes). Only used when `strict = TRUE`.
`gc_after`	Logical. If TRUE, triggers garbage collection after the arena scope exits. Default is TRUE in strict mode, FALSE otherwise.
`diagnostics`	Logical. If TRUE, returns diagnostics about memory usage along with the result. Default is FALSE.

Details

The `arena()` function provides a semantic scope that signals "this code produces scratch data that should not outlive the scope." It serves two purposes:

1. **For compiled kernels**: When Rust-based kernels are available, arena() provides real scratch arenas backed by temporary shared memory segments that are automatically reclaimed.
2. **For arbitrary R code**: Triggers post-task memory checks to detect growth and potential memory leaks.

The `strict` parameter controls escape detection:

- `strict = FALSE` (default): Returns results normally, logs diagnostics about memory growth.
- `strict = TRUE`: Warns or errors if large objects escape the scope, and triggers aggressive memory reclamation.

Value

The result of evaluating `expr`. If `diagnostics = TRUE`, returns an `arena_result` object with elements `result` and `diagnostics`.

See Also

`shard_map` for parallel execution, `share` for shared memory inputs.

Examples

```
result <- arena({
  tmp <- matrix(rnorm(1e6), nrow = 1000)
  colMeans(tmp)
})
```

```
info <- arena({
  x <- rnorm(1e5)
  sum(x)
}, diagnostics = TRUE)
info$diagnostics
```

---

arena_depth *Get Current Arena Depth*

Description

Returns the nesting depth of arena scopes. Useful for debugging.

Usage

```
arena_depth()
```

Value

Integer count of nested arena scopes (0 if not in an arena).

Examples

```
arena_depth()
```

---

as.array.shard_buffer *Coerce a Shared Memory Buffer to Array*

Description

Coerce a Shared Memory Buffer to Array

Usage

```
## S3 method for class 'shard_buffer'
as.array(x, ...)
```

Arguments

x	A shard_buffer object.
...	Ignored.

Value

An array with the buffer contents and the buffer's dimensions, or a plain vector for 1-D buffers.

Examples

```
buf <- buffer("double", dim = c(2, 3, 4))
as.array(buf)
buffer_close(buf)
```

`as.double.shard_buffer` *Coerce a Shared Memory Buffer to Double*

Description

Coerce a Shared Memory Buffer to Double

Usage

```
## S3 method for class 'shard_buffer'
as.double(x, ...)
```

Arguments

`x`	A `shard_buffer` object.
`...`	Ignored.

Value

A double vector with the buffer contents.

Examples

```
buf <- buffer("double", dim = 5)
as.double(buf)
buffer_close(buf)
```

`as.integer.shard_buffer` *Coerce a Shared Memory Buffer to Integer*

Description

Coerce a Shared Memory Buffer to Integer

Usage

```
## S3 method for class 'shard_buffer'
as.integer(x, ...)
```

Arguments

x	A shard_buffer object.
...	Ignored.

Value

An integer vector with the buffer contents.

Examples

```
buf <- buffer("integer", dim = 5)
as.integer(buf)
buffer_close(buf)
```

---

as.logical.shard_buffer

Coerce a Shared Memory Buffer to Logical

Description

Coerce a Shared Memory Buffer to Logical

Usage

```
## S3 method for class 'shard_buffer'
as.logical(x, ...)
```

Arguments

x	A shard_buffer object.
...	Ignored.

Value

A logical vector with the buffer contents.

Examples

```
buf <- buffer("logical", dim = 5)
as.logical(buf)
buffer_close(buf)
```

as.matrix.shard_buffer — *Coerce a Shared Memory Buffer to Matrix*

Description

Coerce a Shared Memory Buffer to Matrix

Usage

```
## S3 method for class 'shard_buffer'
as.matrix(x, ...)
```

Arguments

x	A shard_buffer object (must be 2-dimensional).
...	Ignored.

Value

A matrix with the buffer contents and the buffer's dimensions.

Examples

```
buf <- buffer("double", dim = c(3, 4))
as.matrix(buf)
buffer_close(buf)
```

as.raw.shard_buffer — *Coerce a Shared Memory Buffer to Raw*

Description

Coerce a Shared Memory Buffer to Raw

Usage

```
## S3 method for class 'shard_buffer'
as.raw(x, ...)
```

Arguments

x	A shard_buffer object.
...	Ignored.

Value

A raw vector with the buffer contents.

Examples

```
buf <- buffer("raw", dim = 5)
as.raw(buf)
buffer_close(buf)
```

`as.vector.shard_buffer`

Coerce a Shared Memory Buffer to a Vector

Description

Coerce a Shared Memory Buffer to a Vector

Usage

```
## S3 method for class 'shard_buffer'
as.vector(x, mode = "any")
```

Arguments

`x`	A `shard_buffer` object.
`mode`	Storage mode passed to `as.vector`.

Value

A vector of the buffer's type (or coerced to `mode`).

Examples

```
buf <- buffer("double", dim = 5)
buf[1:5] <- 1:5
as.vector(buf)
buffer_close(buf)
```

as_shared *Create a shared vector from an existing R vector*

Description

Convenience function that creates a segment, writes the data, and returns an ALTREP view.

Usage

```
as_shared(x, readonly = TRUE, backing = "auto", cow = NULL)
```

Arguments

x	An atomic vector (integer, double, logical, or raw)
readonly	If TRUE, prevent write access (default: TRUE)
backing	Backing type for the segment: "auto", "mmap", or "shm"
cow	Copy-on-write policy for the resulting shared vector. One of "deny", "audit", or "allow". If NULL, defaults based on readonly.

Value

An ALTREP vector backed by shared memory

Examples

```
x <- as_shared(1:100)
is_shared_vector(x)

y <- x[1:10]
is_shared_vector(y)
```

as_tibble *Materialize a shard table handle as a data.frame/tibble*

Description

Materialize a shard table handle as a data.frame/tibble

Usage

```
as_tibble(x, max_bytes = 256 * 1024^2, ...)
```

Arguments

x	A shard table object.
max_bytes	Warn if estimated payload exceeds this threshold.
...	Reserved for future extensions.

Value

A data.frame (or tibble if the tibble package is installed).

Examples

```
s <- schema(x = float64(), y = int32())
tb <- table_buffer(s, nrow = 5L)
table_write(tb, idx_range(1, 5), data.frame(x = rnorm(5), y = 1:5))
df <- as_tibble(tb)
```

---

as_tibble.shard_dataset

Materialize a dataset handle into a data.frame/tibble

Description

Materialize a dataset handle into a data.frame/tibble

Usage

```
## S3 method for class 'shard_dataset'
as_tibble(x, max_bytes = 256 * 1024^2, ...)
```

Arguments

x	A shard_dataset handle.
max_bytes	Accepted for API consistency.
...	Reserved for future extensions.

Value

A data.frame (or tibble if the tibble package is installed).

`as_tibble.shard_row_groups`
Materialize a row-groups handle into a data.frame/tibble

Description

Materialize a row-groups handle into a data.frame/tibble

Usage

```
## S3 method for class 'shard_row_groups'
as_tibble(x, max_bytes = 256 * 1024^2, ...)
```

Arguments

`x`	A `shard_row_groups` handle.
`max_bytes`	Accepted for API consistency; currently unused for row-groups.
`...`	Reserved for future extensions.

Value

A data.frame (or tibble if the `tibble` package is installed).

`as_tibble.shard_table_buffer`
Materialize a fixed table handle or buffer

Description

Converts a `shard_table_handle` to an in-memory data.frame (or tibble if the tibble package is installed).

Usage

```
## S3 method for class 'shard_table_buffer'
as_tibble(x, max_bytes = 256 * 1024^2, ...)
```

Arguments

`x`	A `shard_table_handle` or `shard_table_buffer`.
`max_bytes`	Warn if estimated payload exceeds this threshold.
`...`	Reserved for future extensions.

Value

A data.frame (or tibble).

as_tibble.shard_table_handle

Materialize a table handle into a data.frame/tibble

Description

Materialize a table handle into a data.frame/tibble

Usage

```
## S3 method for class 'shard_table_handle'
as_tibble(x, max_bytes = 256 * 1024^2, ...)
```

Arguments

x	A shard_table_handle.
max_bytes	Warn if estimated payload exceeds this threshold.
...	Reserved for future extensions.

Value

A data.frame (or tibble if the tibble package is installed).

attr<-.shard_shared_vector

Set an Attribute on a Shared Vector

Description

Raises an error if the copy-on-write policy is "deny".

Usage

```
## S3 replacement method for class 'shard_shared_vector'
attr(x, which) <- value
```

Arguments

x	A shard_shared_vector.
which	Attribute name.
value	Attribute value.

Value

The modified object x.

attributes<-.shard_shared_vector *Set Attributes on a Shared Vector*

Description

Raises an error if the copy-on-write policy is "deny".

Usage

```
## S3 replacement method for class 'shard_shared_vector'
attributes(x) <- value
```

Arguments

x	A shard_shared_vector.
value	Named list of attributes.

Value

The modified object x.

available_backings *Get available shared memory backing types*

Description

Get available shared memory backing types

Usage

```
available_backings()
```

Value

A character vector of available backing types on the current platform.

Examples

```
available_backings()
```

`buffer` *Shared Memory Buffers*

Description

Create typed writable output buffers backed by shared memory for cross-process writes during parallel execution.

Creates a typed output buffer backed by shared memory that can be written to by parallel workers using slice assignment.

Usage

```
buffer(
  type = c("double", "integer", "logical", "raw"),
  dim,
  init = NULL,
  backing = c("auto", "mmap", "shm")
)
```

Arguments

`type`	Character. Data type: "double" (default), "integer", "logical", or "raw".
`dim`	Integer vector. Dimensions of the buffer. For a vector, specify the length. For a matrix, specify `c(nrow, ncol)`. For arrays, specify all dimensions.
`init`	Initial value to fill the buffer. Default is type-appropriate zero (`0`, `0L`, `FALSE`, or `raw(0)`).
`backing`	Backing type for shared memory: "auto" (default), "mmap", or "shm".

Details

Buffers provide an explicit output mechanism for `shard_map`. Instead of returning results from workers (which requires serialization and memory copying), workers write directly to shared buffers.

Supported types:

- `"double"`: 8-byte floating point (default)
- `"integer"`: 4-byte signed integer
- `"logical"`: 4-byte logical (stored as integer)
- `"raw"`: 1-byte raw data

Buffers support slice assignment using standard R indexing: `buf[1:100] <- values`

Value

An S3 object of class "shard_buffer" that supports:

- Slice assignment: `buf[idx] <- values`
- Slice reading: `buf[idx]`
- Full extraction: `buf[]`
- Conversion to R vector: `as.vector(buf)`, `as.double(buf)`, etc.

See Also

`segment_create` for low-level segment operations, `share` for read-only shared inputs

Examples

```
out <- buffer("double", dim = 100)
out[1:10] <- rnorm(10)
result <- out[]
```

---

buffer_advise *Advise access pattern for a buffer*

Description

Advise access pattern for a buffer

Usage

```
buffer_advise(
  x,
  advice = c("normal", "sequential", "random", "willneed", "dontneed")
)
```

Arguments

x	A shard_buffer.
advice	See `segment_advise()`.

Value

A logical scalar; `TRUE` if the OS accepted the hint.

Examples

```
buf <- buffer("double", dim = 10L)
buffer_advise(buf, "sequential")
```

buffer_close *Close a Buffer*

Description

Closes the buffer and releases the underlying shared memory.

Usage

```
buffer_close(x, unlink = NULL)
```

Arguments

x	A shard_buffer object.
unlink	Whether to unlink the underlying segment.

Value

NULL, invisibly.

Examples

```
buf <- buffer("double", dim = 10)
buffer_close(buf)
```

buffer_diagnostics *Buffer Diagnostics*

Description

Returns per-process counters for shard buffer writes. shard_map uses these internally to report write volume/operations in copy_report().

Usage

```
buffer_diagnostics()
```

Value

A list with elements writes (integer count) and bytes (total bytes written) accumulated in the current process.

Examples

```
buffer_diagnostics()
```

buffer_info *Get Buffer Info*

Description

Returns information about a buffer.

Usage

```
buffer_info(x)
```

Arguments

x	A shard_buffer object.

Value

A named list with buffer properties: type, dim, n, bytes, backing, path, and readonly.

Examples

```
buf <- buffer("integer", dim = c(5, 5))
buffer_info(buf)
buffer_close(buf)
```

buffer_open *Open an Existing Buffer*

Description

Opens a shared memory buffer that was created in another process. Used by workers to attach to the parent's output buffer.

Usage

```
buffer_open(path, type, dim, backing = c("mmap", "shm"), readonly = FALSE)
```

Arguments

path	Path or shm name of the buffer's segment.
type	Character. Data type of the buffer.
dim	Integer vector. Dimensions of the buffer.
backing	Backing type: "mmap" or "shm".
readonly	Logical. Open as read-only? Default FALSE for workers.

Value

A shard_buffer object attached to the existing segment.

Examples

```
buf <- buffer("double", dim = 10)
path <- buffer_path(buf)
buf2 <- buffer_open(path, type = "double", dim = 10, backing = "mmap")
buffer_close(buf2, unlink = FALSE)
buffer_close(buf)
```

---

buffer_path *Get Buffer Path*

Description

Returns the path or name of the buffer's underlying segment. Use this to pass buffer location to workers.

Usage

```
buffer_path(x)
```

Arguments

x	A shard_buffer object.

Value

A character string with the path or name of the segment, or NULL if the segment is anonymous.

Examples

```
buf <- buffer("double", dim = 10)
buffer_path(buf)
buffer_close(buf)
```

close.shard_shared *Close a Shared Object*

Description

Releases the shared memory segment. After closing, the shared object can no longer be accessed.

Usage

```
## S3 method for class 'shard_shared'
close(con, ...)

## S3 method for class 'shard_shared_vector'
close(con, ...)

## S3 method for class 'shard_deep_shared'
close(con, ...)
```

Arguments

con	A shard_shared object.
...	Ignored.

Value

NULL (invisibly).

collect *Collect a shard table into memory*

Description

collect() is a convenience alias for as_tibble() for shard table outputs.

Usage

```
collect(x, ...)
```

Arguments

x	A shard table handle (shard_row_groups, shard_dataset, or shard_table_handle).
...	Passed to as_tibble().

Value

A data.frame (or tibble if the tibble package is installed).

Examples

```
s <- schema(x = float64(), y = int32())
tb <- table_buffer(s, nrow = 5L)
table_write(tb, idx_range(1, 5), data.frame(x = rnorm(5), y = 1:5))
handle <- table_finalize(tb)
df <- collect(handle)
```

---

collect.shard_dataset *Collect a dataset handle into memory*

Description

Collect a dataset handle into memory

Usage

```
## S3 method for class 'shard_dataset'
collect(x, ...)
```

Arguments

x	A shard_dataset handle.
...	Passed to as_tibble().

Value

A data.frame (or tibble if the tibble package is installed).

---

collect.shard_row_groups *Collect a row-groups handle into memory*

Description

Collect a row-groups handle into memory

Usage

```
## S3 method for class 'shard_row_groups'
collect(x, ...)
```

Arguments

x	A shard_row_groups handle.
...	Passed to as_tibble().

Value

A data.frame (or tibble if the `tibble` package is installed).

`collect.shard_table_handle` *Collect a table handle into memory*

Description

Collect a table handle into memory

Usage

```
## S3 method for class 'shard_table_handle'
collect(x, ...)
```

Arguments

`x`	A `shard_table_handle`.
`...`	Passed to `as_tibble()`.

Value

A data.frame (or tibble if the `tibble` package is installed).

`coltypes` *Column Types*

Description

Type constructors for schema-driven table outputs.

Usage

```
int32()

float64()

bool()

raw_col()

string_col()
```

Value

A `shard_coltype` object.

copy_report *Data Copy Report*

Description

Generates a report of data transfer and copy statistics during parallel execution.

Usage

```
copy_report(result = NULL)
```

Arguments

result	Optional. A shard_result object to extract copy stats from.

Value

An S3 object of class shard_report with type "copy" containing:

- type: "copy"
- timestamp: When the report was generated
- borrow_exports: Number of borrowed input exports
- borrow_bytes: Total bytes in borrowed inputs
- result_imports: Number of result imports
- result_bytes: Estimated bytes in results
- buffer_writes: Number of buffer write operations
- buffer_bytes: Total bytes written to buffers

Examples

```
res <- shard_map(shards(100, workers = 2), function(s) sum(s$idx), workers = 2)
pool_stop()
copy_report(res)
```

cow_report *Copy-on-Write Policy Report*

Description

Generates a report of copy-on-write behavior for borrowed inputs.

Usage

```
cow_report(result = NULL)
```

Arguments

result	Optional. A shard_result object to extract COW stats from.

Value

An S3 object of class shard_report with type "cow" containing:

- type: "cow"
- timestamp: When the report was generated
- policy: The COW policy used ("deny", "audit", "allow")
- violations: Count of COW violations detected (audit mode)
- copies_triggered: Estimated copies triggered by mutations

Examples

```
res <- shard_map(shards(100, workers = 2), function(s) sum(s$idx), workers = 2)
pool_stop()
cow_report(res)
```

diagnostics *Diagnostics API*

Description

Comprehensive diagnostics for shard parallel execution, providing insights into memory usage, worker status, task execution, and shared memory segments.

Details

The diagnostics API provides multiple views into shard's runtime behavior:

- `report()`: Primary entry point with configurable detail levels
- `mem_report()`: Memory usage across workers
- `cow_report()`: Copy-on-write policy tracking
- `copy_report()`: Data transfer statistics
- `task_report()`: Task/chunk execution statistics
- `segment_report()`: Shared memory segment information

All functions return S3 `shard_report` objects with appropriate print methods for human-readable output.

`dim.shard_buffer` *Dimensions of a Shared Memory Buffer*

Description

Dimensions of a Shared Memory Buffer

Usage

```
## S3 method for class 'shard_buffer'
dim(x)
```

Arguments

`x`	A `shard_buffer` object.

Value

An integer vector of dimensions, or `NULL` for 1-D buffers.

Examples

```
buf <- buffer("double", dim = c(4, 5))
dim(buf)
buffer_close(buf)
```

`dim<-.shard_shared_vector`
Set dim on a Shared Vector

Description

Raises an error if the copy-on-write policy is `"deny"`.

Usage

```
## S3 replacement method for class 'shard_shared_vector'
dim(x) <- value
```

Arguments

`x`	A `shard_shared_vector`.
`value`	Integer vector of dimensions.

Value

The modified object `x`.

`dimnames<-.shard_shared_vector`
Set dimnames on a Shared Vector

Description

Raises an error if the copy-on-write policy is `"deny"`.

Usage

```
## S3 replacement method for class 'shard_shared_vector'
dimnames(x) <- value
```

Arguments

`x`	A `shard_shared_vector`.
`value`	List of dimnames.

Value

The modified object `x`.

dispatch *Task Dispatch Engine*

Description

Orchestrates chunk dispatch with worker supervision and failure handling.

dispatch_chunks *Dispatch Chunks to Worker Pool*

Description

Executes a function over chunks using the worker pool with supervision. Handles worker death and recycling transparently by requeuing failed chunks.

Usage

```
dispatch_chunks(
  chunks,
  fun,
  ...,
  pool = NULL,
  health_check_interval = 10L,
  max_retries = 3L,
  timeout = 3600,
  scheduler_policy = NULL,
  on_result = NULL,
  store_results = TRUE,
  retain_chunks = TRUE
)
```

Arguments

chunks	List of chunk descriptors. Each chunk will be passed to fun.
fun	Function to execute. Receives (chunk, ...) as arguments.
...	Additional arguments passed to fun.
pool	A shard_pool object. If NULL, uses the current pool.
health_check_interval	Integer. Check pool health every N chunks (default 10).
max_retries	Integer. Maximum retries per chunk before permanent failure (default 3).
timeout	Numeric. Seconds to wait for each chunk (default 3600).
scheduler_policy	Optional list of scheduling hints (advanced). Currently:

- `max_huge_concurrency`: cap concurrent chunks with `footprint_class=="huge"`.

`on_result`	Optional callback (advanced). If provided, called on the master process as `on_result(tag, value, worker_id)` for each successful chunk completion. Used by `shard_reduce()` to stream reductions.
`store_results`	Logical (advanced). If FALSE, successful chunk values are not retained in the returned `results` list (streaming use cases).
`retain_chunks`	Logical (advanced). If FALSE, completed chunk descriptors are stored minimally (avoids retaining large shard lists in memory).

Value

A `shard_dispatch_result` object with results and diagnostics.

Examples

```
pool_create(2)
chunks <- list(list(id = 1L, x = 1), list(id = 2L, x = 2))
result <- dispatch_chunks(chunks, function(chunk) chunk$x * 2, pool = pool_get())
pool_stop()
```

---

ergonomics *Ergonomic Apply/Lapply Wrappers*

Description

Convenience wrappers that provide apply/lapply-style ergonomics while preserving shard's core contract: shared immutable inputs, supervised execution, and diagnostics.

These functions are intentionally thin wrappers around `shard_map()` and related primitives.

---

factor_col *Categorical column type*

Description

Stores factors as int32 codes plus shared levels metadata.

Usage

```
factor_col(levels)
```

Arguments

`levels`	Character vector of allowed levels.

Value

A `shard_coltype` object.

`fetch` *Fetch Data from a Shared Object*

Description

Retrieves the R object from shared memory by deserializing it. This is the primary way to access shared data in workers.

Usage

```
fetch(x, ...)

## S3 method for class 'shard_shared'
fetch(x, ...)

## S3 method for class 'shard_deep_shared'
fetch(x, ...)

## Default S3 method:
fetch(x, ...)
```

Arguments

`x`	A `shard_shared` object.
`...`	Ignored.

Details

When called in the main process, this reads from the existing segment. When called in a worker process, this opens the segment by path and deserializes the data.

The `fetch()` function is the primary way to access shared data. It can also be called as `materialize()` for compatibility.

Value

The original R object that was shared.

Examples

```
x <- 1:100
shared <- share(x)
recovered <- fetch(shared)
identical(x, recovered)
close(shared)
```

idx_range *Contiguous index range*

Description

Creates a compact, serializable range descriptor for contiguous indices. This avoids allocating an explicit index vector for large slices.

Usage

```
idx_range(start, end)
```

Arguments

start	Integer. Start index (1-based, inclusive).
end	Integer. End index (1-based, inclusive).

Value

An object of class shard_idx_range.

Examples

```
r <- idx_range(1, 100)
r
```

in_arena *Check if Currently Inside an Arena*

Description

Returns TRUE if the current execution context is within an arena() scope.

Usage

```
in_arena()
```

Value

Logical indicating whether we are in an arena scope.

Examples

```
in_arena()
arena({
  in_arena()
})
```

is_shared *Check if Object is Shared*

Description

Check if Object is Shared

Usage

```
is_shared(x)
```

Arguments

x	An object to check.

Value

A logical scalar: TRUE if x is a shared object, FALSE otherwise.

Examples

```
is_shared(1:10)

shared <- share(1:10)
is_shared(shared)
close(shared)
```

is_shared_vector *Check if an object is a shared vector*

Description

Check if an object is a shared vector

Usage

```
is_shared_vector(x)
```

Arguments

x	Any R object

Value

TRUE if x is a shard ALTREP vector, FALSE otherwise

Examples

```
seg <- segment_create(400)
segment_write(seg, 1:100, offset = 0)
x <- shared_vector(seg, "integer", length = 100)

is_shared_vector(x)
is_shared_vector(1:10)
```

`is_view` *View Predicates*

Description

View Predicates

Usage

```
is_view(x)

is_block_view(x)
```

Arguments

`x`	An object.

Value

Logical. `TRUE` if `x` is a shard view (or block view).

Examples

```
m <- share(matrix(1:20, nrow = 4))
v <- view_block(m, cols = idx_range(1, 2))
is_view(v)
is_block_view(v)
```

is_windows *Check if running on Windows*

Description

Check if running on Windows

Usage

```
is_windows()
```

Value

A logical scalar: TRUE if running on Windows, FALSE otherwise.

Examples

```
is_windows()
```

iterate_row_groups *Iterate row groups*

Description

Iterate row groups

Usage

```
iterate_row_groups(x, decode = TRUE)
```

Arguments

x	A shard_row_groups handle.
decode	Logical. If TRUE (default), native-encoded partitions are decoded to data.frames. If FALSE, native partitions are returned as their internal representation (advanced).

Value

A zero-argument iterator function that returns the next data.frame on each call, or NULL when exhausted.

Examples

```
s <- schema(x = float64())
sink <- table_sink(s, mode = "row_groups")
table_write(sink, 1L, data.frame(x = rnorm(5)))
rg <- table_finalize(sink)
it <- iterate_row_groups(rg)
chunk <- it()
```

---

length.shard_buffer *Length of a Shared Memory Buffer*

Description

Length of a Shared Memory Buffer

Usage

```
## S3 method for class 'shard_buffer'
length(x)
```

Arguments

x	A shard_buffer object.

Value

An integer scalar giving the total number of elements.

Examples

```
buf <- buffer("double", dim = 20)
length(buf)
buffer_close(buf)
```

---

length.shard_descriptor *Length of a shard_descriptor Object*

Description

Length of a shard_descriptor Object

Usage

```
## S3 method for class 'shard_descriptor'
length(x)
```

Arguments

x	A shard_descriptor object.

Value

An integer scalar giving the number of shards.

Examples

```
sh <- shards(100, block_size = 25)
length(sh)
```

---

length.shard_descriptor_lazy

Length of a shard_descriptor_lazy Object

Description

Length of a shard_descriptor_lazy Object

Usage

```
## S3 method for class 'shard_descriptor_lazy'
length(x)
```

Arguments

x	A shard_descriptor_lazy object.

Value

An integer scalar giving the number of shards.

Examples

```
sh <- shards(100, block_size = 25)
length(sh)
```

list_kernels *List registered kernels*

Description

List registered kernels

Usage

```
list_kernels()
```

Value

A character vector of registered kernel names.

Examples

```
list_kernels()
```

materialize *Materialize Shared Object*

Description

Alias for `fetch()`. Retrieves the R object from shared memory.

Usage

```
materialize(x)

## S3 method for class 'shard_shared'
materialize(x)

## Default S3 method:
materialize(x)
```

Arguments

x	A shard_shared object.

Value

The original R object.

Examples

```
shared <- share(1:100)
data <- materialize(shared)
close(shared)
```

---

materialize.shard_view_block
Materialize a block view into an R matrix

Description

Materialize a block view into an R matrix

Usage

```
## S3 method for class 'shard_view_block'
materialize(x)
```

Arguments

x	A shard_view_block object.

Value

A standard R matrix containing the selected rows and columns.

---

materialize.shard_view_gather
Materialize a gather view into an R matrix

Description

Materialize a gather view into an R matrix

Usage

```
## S3 method for class 'shard_view_gather'
materialize(x)
```

Arguments

x	A shard_view_gather object.

Value

A standard R matrix containing the gathered columns.

mem_report *Memory Usage Report*

Description

Generates a report of memory usage across all workers in the pool.

Usage

```
mem_report(pool = NULL)
```

Arguments

pool	Optional. A shard_pool object. If NULL, uses the current pool.

Value

An S3 object of class shard_report with type "memory" containing:

- type: "memory"
- timestamp: When the report was generated
- pool_active: Whether a pool exists
- n_workers: Number of workers
- rss_limit: RSS limit per worker (bytes)
- total_rss: Sum of RSS across all workers
- peak_rss: Highest RSS among workers
- mean_rss: Mean RSS across workers
- workers: Per-worker RSS details

Examples

```
p <- pool_create(2)
mem_report(p)
pool_stop(p)
```

names<-.shard_shared_vector *Set Names on a Shared Vector*

Description

Raises an error if the copy-on-write policy is "deny".

Usage

```
## S3 replacement method for class 'shard_shared_vector'
names(x) <- value
```

Arguments

x	A shard_shared_vector.
value	Character vector of names.

Value

The modified object x.

pin_workers *Pin shard workers to CPU cores*

Description

Best-effort worker pinning to improve cache locality and reduce cross-core migration. Currently supported on Linux only.

Usage

```
pin_workers(pool = NULL, strategy = c("spread", "compact"), cores = NULL)
```

Arguments

pool	Optional shard_pool. Defaults to current pool.
strategy	"spread" assigns worker i -> core i mod ncores. "compact" assigns workers to the first cores.
cores	Optional integer vector of available cores (0-based). If NULL, uses 0:(detectCores()-1).

Value

Invisibly, a logical vector per worker indicating success.

Examples

```
affinity_supported()
```

`pool` *Worker Pool Management*

Description

Spawn and supervise persistent R worker processes with RSS monitoring.

`pool_create` *Create a Worker Pool*

Description

Spawns N R worker processes that persist across multiple `shard_map()` calls. Workers are supervised and recycled when RSS drift exceeds thresholds.

Usage

```
pool_create(
  n = parallel::detectCores() - 1L,
  rss_limit = "2GB",
  rss_drift_threshold = 0.5,
  heartbeat_interval = 5,
  min_recycle_interval = 1,
  init_expr = NULL,
  packages = NULL
)
```

Arguments

`n`	Integer. Number of worker processes to spawn.
`rss_limit`	Numeric or character. Maximum RSS per worker before recycling. Can be bytes (numeric) or human-readable (e.g., "2GB"). Default is "2GB".
`rss_drift_threshold`	Numeric. Fraction of RSS increase from baseline that triggers recycling (default 0.5 = 50% growth).
`heartbeat_interval`	Numeric. Seconds between health checks (default 5).
`min_recycle_interval`	Numeric. Minimum time in seconds between recycling the same worker (default 1.0). This prevents thrashing PSOCK worker creation under extremely tight RSS limits.
`init_expr`	Expression to evaluate in each worker on startup.
`packages`	Character vector. Packages to load in workers.

Value

A shard_pool object (invisibly). The pool is also stored in the package environment for reuse.

Examples

```
p <- pool_create(2)
pool_stop(p)
```

---

pool_dispatch *Dispatch Task to Worker*

Description

Sends a task to a specific worker and waits for the result.

Usage

```
pool_dispatch(
  worker_id,
  expr,
  envir = parent.frame(),
  pool = NULL,
  timeout = 3600
)
```

Arguments

worker_id	Integer. Worker to dispatch to.
expr	Expression to evaluate.
envir	Environment containing variables needed by expr.
pool	A shard_pool object. If NULL, uses the current pool.
timeout	Numeric. Seconds to wait for result (default 3600).

Value

The result of evaluating expr in the worker.

Examples

```
p <- pool_create(2)
pool_dispatch(1, quote(1 + 1), pool = p)
pool_stop(p)
```

pool_get *Get the Current Worker Pool*

Description

Returns the active worker pool, or NULL if none exists.

Usage

```
pool_get()
```

Value

A shard_pool object or NULL.

Examples

```
p <- pool_get()
is.null(p)
```

pool_health_check *Check Pool Health*

Description

Monitors all workers, recycling those with excessive RSS drift or that have died.

Usage

```
pool_health_check(pool = NULL, busy_workers = NULL)
```

Arguments

pool	A shard_pool object. If NULL, uses the current pool.
busy_workers	Optional integer vector of worker ids that are currently running tasks (used internally by the dispatcher to avoid recycling a worker while a result is in flight).

Value

A list with health status per worker and actions taken.

Examples

```
p <- pool_create(2)
pool_health_check(p)
pool_stop(p)
```

pool_lapply *Parallel Dispatch with Async Workers*

Description

An alternative dispatch that uses parallel::parLapply-style execution but with supervision. This is a simpler interface for basic parallel apply.

Usage

```
pool_lapply(X, FUN, ..., pool = NULL, chunk_size = 1L)
```

Arguments

X	List or vector to iterate over.
FUN	Function to apply to each element.
...	Additional arguments to FUN.
pool	A shard_pool object. If NULL, uses current pool.
chunk_size	Integer. Elements per chunk (default 1).

Value

A list of results.

Examples

```
pool_create(2)
result <- pool_lapply(1:4, function(x) x^2, pool = pool_get())
pool_stop()
```

pool_sapply *Parallel sapply with Supervision*

Description

Parallel sapply with Supervision

Usage

```
pool_sapply(X, FUN, ..., simplify = TRUE, pool = NULL)
```

Arguments

X	List or vector to iterate over.
FUN	Function to apply.
...	Additional arguments to FUN.
simplify	Logical. Simplify result to vector/matrix?
pool	A shard_pool object. If NULL, uses current pool.

Value

Simplified result if possible, otherwise a list.

Examples

```
pool_create(2)
result <- pool_sapply(1:4, function(x) x^2, pool = pool_get())
pool_stop()
```

---

pool_status *Get Pool Status*

Description

Returns current status of all workers in the pool.

Usage

```
pool_status(pool = NULL)
```

Arguments

pool	A shard_pool object. If NULL, uses the current pool.

Value

A data frame with worker status information.

Examples

```
p <- pool_create(2)
pool_status(p)
pool_stop(p)
```

pool_stop *Stop the Worker Pool*

Description

Terminates all worker processes and releases resources. Waits for workers to actually terminate before returning.

Usage

```
pool_stop(pool = NULL, timeout = 5)
```

Arguments

pool	A shard_pool object. If NULL, uses the current pool.
timeout	Numeric. Seconds to wait for workers to terminate (default 5). Returns after timeout even if workers are still alive.

Value

NULL (invisibly).

Examples

```
p <- pool_create(2)
pool_stop(p)
```

print.arena_result *Print an arena_result object*

Description

Print an arena_result object

Usage

```
## S3 method for class 'arena_result'
print(x, ...)
```

Arguments

x	An arena_result object.
...	Additional arguments passed to print.

Value

Returns x invisibly.

Examples

```
info <- arena({ sum(1:10) }, diagnostics = TRUE)
print(info)
```

---

print.shard_apply_policy *Print a shard_apply_policy Object*

Description

Print a shard_apply_policy Object

Usage

```
## S3 method for class 'shard_apply_policy'
print(x, ...)
```

Arguments

x	A shard_apply_policy object.
...	Ignored.

Value

The input x, invisibly.

---

print.shard_buffer *Print a Shared Memory Buffer*

Description

Print a Shared Memory Buffer

Usage

```
## S3 method for class 'shard_buffer'
print(x, ...)
```

Arguments

x	A shard_buffer object.
...	Ignored.

Value

The input x, invisibly.

Examples

```
buf <- buffer("double", dim = 10)
print(buf)
buffer_close(buf)
```

---

print.shard_deep_shared

Print a Deep-Shared Object

Description

Print a Deep-Shared Object

Usage

```
## S3 method for class 'shard_deep_shared'
print(x, ...)
```

Arguments

x	A shard_deep_shared object.
...	Ignored.

Value

The input x, invisibly.

Examples

```
lst <- list(a = 1:10, b = 11:20)
shared <- share(lst, deep = TRUE, min_bytes = 1)
print(shared)
close(shared)
```

`print.shard_descriptor` *Print a shard_descriptor Object*

Description

Print a shard_descriptor Object

Usage

```
## S3 method for class 'shard_descriptor'
print(x, ...)
```

Arguments

`x`	A `shard_descriptor` object.
`...`	Further arguments (ignored).

Value

The input `x`, invisibly.

Examples

```
sh <- shards(100, block_size = 25)
print(sh)
```

`print.shard_descriptor_lazy` *Print a shard_descriptor_lazy Object*

Description

Print a shard_descriptor_lazy Object

Usage

```
## S3 method for class 'shard_descriptor_lazy'
print(x, ...)
```

Arguments

`x`	A `shard_descriptor_lazy` object.
`...`	Further arguments (ignored).

Value

The input x, invisibly.

Examples

```
sh <- shards(100, block_size = 25)
print(sh)
```

`print.shard_dispatch_result` *Print a shard_dispatch_result Object*

Description

Print a shard_dispatch_result Object

Usage

```
## S3 method for class 'shard_dispatch_result'
print(x, ...)
```

Arguments

x	A `shard_dispatch_result` object.
...	Further arguments (ignored).

Value

The input x, invisibly.

Examples

```
pool_create(2)
chunks <- list(list(id = 1L, x = 1), list(id = 2L, x = 2))
result <- dispatch_chunks(chunks, function(chunk) chunk$x, pool = pool_get())
print(result)
pool_stop()
```

print.shard_health_report *Print a shard_health_report Object*

Description

Print a shard_health_report Object

Usage

```
## S3 method for class 'shard_health_report'
print(x, ...)
```

Arguments

x	A shard_health_report object.
...	Further arguments (ignored).

Value

The input x, invisibly.

Examples

```
p <- pool_create(2)
r <- pool_health_check(p)
print(r)
pool_stop(p)
```

print.shard_idx_range *Print a shard_idx_range object*

Description

Print a shard_idx_range object

Usage

```
## S3 method for class 'shard_idx_range'
print(x, ...)
```

Arguments

x	A shard_idx_range object.
...	Additional arguments (ignored).

Value

Returns x invisibly.

Examples

```
r <- idx_range(1, 10)
print(r)
```

`print.shard_pool` *Print a shard_pool Object*

Description

Print a shard_pool Object

Usage

```
## S3 method for class 'shard_pool'
print(x, ...)
```

Arguments

x	A shard_pool object.
...	Further arguments (ignored).

Value

The input x, invisibly.

Examples

```
p <- pool_create(2)
print(p)
pool_stop(p)
```

print.shard_reduce_result *Print a shard_reduce_result Object*

Description

Print a shard_reduce_result Object

Usage

```
## S3 method for class 'shard_reduce_result'
print(x, ...)
```

Arguments

x	A shard_reduce_result object.
...	Further arguments (ignored).

Value

The input x, invisibly.

Examples

```
res <- shard_reduce(4L, map = function(s) sum(s$idx),
  combine = `+`, init = 0, workers = 2)
pool_stop()
print(res)
```

print.shard_report *Print a shard_report Object*

Description

Print a shard_report Object

Usage

```
## S3 method for class 'shard_report'
print(x, ...)
```

Arguments

x	A shard_report object.
...	Ignored.

Value

The input x, invisibly.

Examples

```
res <- shard_map(shards(100, workers = 2), function(s) sum(s$idx), workers = 2)
pool_stop()
rpt <- report(result = res)
print(rpt)
```

---

print.shard_result *Print a shard_result Object*

Description

Print a shard_result Object

Usage

```
## S3 method for class 'shard_result'
print(x, ...)
```

Arguments

x	A shard_result object.
...	Further arguments (ignored).

Value

The input x, invisibly.

Examples

```
result <- shard_map(4L, function(shard) shard$idx, workers = 2)
pool_stop()
print(result)
```

print.shard_segment *Print a Shared Memory Segment*

Description

Print a Shared Memory Segment

Usage

```
## S3 method for class 'shard_segment'
print(x, ...)
```

Arguments

x	A shard_segment object.
...	Ignored.

Value

The input x, invisibly.

Examples

```
seg <- segment_create(1024)
print(seg)
segment_close(seg)
```

print.shard_shared *Print a Shared Object*

Description

Print a Shared Object

Usage

```
## S3 method for class 'shard_shared'
print(x, ...)
```

Arguments

x	A shard_shared object.
...	Ignored.

Value

The input x, invisibly.

Examples

```
shared <- share(1:10)
print(shared)
close(shared)
```

---

print.shard_shared_vector *Print a Shared Vector*

Description

Print method for shard_shared_vector objects. Drops the wrapper class and delegates to the underlying R print method.

Usage

```
## S3 method for class 'shard_shared_vector'
print(x, ...)
```

Arguments

x	A shard_shared_vector.
...	Additional arguments passed to print.

Value

The input x, invisibly.

---

print.shard_tiles *Print a shard_tiles object*

Description

Print a shard_tiles object

Usage

```
## S3 method for class 'shard_tiles'
print(x, ...)
```

Arguments

x	A shard_tiles object.
...	Additional arguments (ignored).

Value

Returns x invisibly.

---

print.shard_view_block

Print a shard_view_block object

Description

Print a shard_view_block object

Usage

```
## S3 method for class 'shard_view_block'
print(x, ...)
```

Arguments

x	A shard_view_block object.
...	Additional arguments (ignored).

Value

Returns x invisibly.

---

print.shard_view_gather

Print a shard_view_gather object

Description

Print a shard_view_gather object

Usage

```
## S3 method for class 'shard_view_gather'
print(x, ...)
```

Arguments

x	A shard_view_gather object.
...	Additional arguments (ignored).

Value

Returns x invisibly.

---

print.shard_worker *Print a shard_worker Object*

Description

Print a shard_worker Object

Usage

```
## S3 method for class 'shard_worker'
print(x, ...)
```

Arguments

x	A shard_worker object.
...	Further arguments (ignored).

Value

The input x, invisibly.

Examples

```
p <- pool_create(1)
print(p$workers[[1]])
pool_stop(p)
```

---

queue *Chunk Queue Management*

Description

Queue management for dispatching chunks to workers with requeue support.

recommendations *Performance Recommendations*

Description

Uses run telemetry (copy/materialization stats, packing volume, buffer/table writes, scratch pool stats) to produce actionable recommendations.

Usage

```
recommendations(result)
```

Arguments

result	A shard_result from shard_map().

Value

A character vector of recommendations (possibly empty).

Examples

```
res <- shard_map(shards(100, workers = 2), function(s) sum(s$idx), workers = 2)
pool_stop()
recommendations(res)
```

register_kernel *Register a shard kernel*

Description

Registers a named kernel implementation that can be selected by shard_map(..., kernel = "name").

Usage

```
register_kernel(
  name,
  impl,
  signature = NULL,
  footprint = NULL,
  supports_views = TRUE,
  description = NULL
)
```

Arguments

`name`	Kernel name (string).
`impl`	Function implementing the kernel. It must accept the shard descriptor as its first argument.
`signature`	Optional short signature string for documentation.
`footprint`	Optional footprint hint. Either a constant (bytes) or a function `(shard, ...) -> list(class='tiny'|'m`
`supports_views`	Logical. Whether the kernel is intended to operate on shard views without slice materialization.
`description`	Optional human-readable description.

Details

A "kernel" is just a function that shard_map can call for each shard. The registry lets shard_map attach additional metadata (footprint hints, supports_views) for scheduling/autotuning.

Value

Invisibly, the registered kernel metadata.

Examples

```
list_kernels()
```

`report` *Generate Shard Runtime Report*

Description

Primary entry point for shard diagnostics. Generates a comprehensive report of the current runtime state including pool status, memory usage, and execution statistics.

Usage

```
report(level = c("summary", "workers", "tasks", "segments"), result = NULL)
```

Arguments

`level`	Character. Detail level for the report: • `"summary"`: High-level overview (default) • `"workers"`: Include per-worker details • `"tasks"`: Include task execution history • `"segments"`: Include shared memory segment details
`result`	Optional. A `shard_result` object from `shard_map` to include execution diagnostics from.

Value

An S3 object of class `shard_report` containing:

- `level`: The requested detail level
- `timestamp`: When the report was generated
- `pool`: Pool status information (if pool exists)
- `memory`: Memory usage summary
- `workers`: Per-worker details (if level includes workers)
- `tasks`: Task execution details (if level includes tasks)
- `segments`: Segment details (if level includes segments)
- `result_diagnostics`: Diagnostics from shard_result (if provided)

Examples

```
res <- shard_map(shards(100, workers = 2), function(s) sum(s$idx), workers = 2)
pool_stop()
report(result = res)
```

---

results *Extract Results from shard_map*

Description

Extract Results from shard_map

Usage

```
results(x, flatten = TRUE)
```

Arguments

`x`	A shard_result object.
`flatten`	Logical. Flatten nested results?

Value

List or vector of results.

Examples

```
result <- shard_map(4L, function(shard) shard$idx[[1L]], workers = 2)
pool_stop()
results(result)
```

row_layout *Row layout for fixed-row table outputs*

Description

Computes disjoint row ranges for each shard via prefix-sum, enabling lock-free writes where each shard writes to a unique region.

Usage

```
row_layout(shards, rows_per_shard)
```

Arguments

shards	A shard_descriptor.
rows_per_shard	Either a scalar integer or a function(shard)->integer.

Value

A named list mapping shard id (character) to an idx_range(start, end).

Examples

```
sh <- shards(100, block_size = 25)
layout <- row_layout(sh, rows_per_shard = 25L)
```

rss *RSS Monitoring Utilities*

Description

Cross-platform utilities for monitoring process memory usage.

schema *Define a table schema*

Description

A schema is a named set of columns with explicit types. It is used to allocate table buffers and validate writes.

Usage

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

Arguments

`...`	Named columns with type specs (e.g., `int32()`, `float64()`).

Value

A `shard_schema` object.

Examples

```
s <- schema(x = float64(), y = int32(), label = string_col())
s
```

scratch_diagnostics *Scratch pool diagnostics*

Description

Scratch pool diagnostics

Usage

```
scratch_diagnostics()
```

Value

A list with counters and current pool bytes.

Examples

```
scratch_diagnostics()
```

scratch_matrix *Get a scratch matrix*

Description

Allocates (or reuses) a double matrix in the worker scratch pool.

Usage

```
scratch_matrix(nrow, ncol, key = NULL)
```

Arguments

nrow, ncol	Dimensions.
key	Optional key to control reuse. Defaults to a shape-derived key.

Value

A double matrix of dimensions nrow by ncol.

Examples

```
m <- scratch_matrix(10, 5)
dim(m)
```

scratch_pool_config *Configure scratch pool limits*

Description

Configure scratch pool limits

Usage

```
scratch_pool_config(max_bytes = Inf)
```

Arguments

max_bytes	Maximum scratch pool bytes allowed in a worker. If exceeded, the worker is flagged for recycle at the next safe point.

Value

NULL, invisibly.

Examples

```
cfg <- scratch_pool_config(max_bytes = 100 * 1024^2)
```

segment *Shared Memory Segment*

Description

Low-level shared memory segment operations for cross-process data sharing. These functions provide the foundation for the higher-level `share()` and `buffer()` APIs.

Details

Segments can be backed by:

- `"shm"`: POSIX shared memory (Linux/macOS) or named file mapping (Windows). Faster but may have size limitations.
- `"mmap"`: File-backed memory mapping. Works on all platforms and supports larger sizes.
- `"auto"`: Let the system choose the best option.

All segments are created with secure permissions (0600 on Unix) and are automatically cleaned up when the R object is garbage collected.

segment_advise *Advise OS about expected access pattern for a segment*

Description

This calls `madvise()` on the segment mapping when available.

Usage

```
segment_advise(
  seg,
  advice = c("normal", "sequential", "random", "willneed", "dontneed")
)
```

Arguments

`seg`	A shard_segment.
`advice`	One of "normal", "sequential", "random", "willneed", "dontneed".

Value

A logical scalar; `TRUE` if the OS accepted the hint.

Examples

```
seg <- segment_create(1024)
segment_advise(seg, "sequential")
```

segment_close *Close a shared memory segment*

Description

Close a shared memory segment

Usage

```
segment_close(x, unlink = NULL)
```

Arguments

x	A shard_segment object
unlink	Whether to unlink the underlying file/shm (default: FALSE for opened segments, TRUE for owned segments)

Value

NULL, invisibly.

Examples

```
seg <- segment_create(1024)
segment_close(seg)
```

segment_create *Create a new shared memory segment*

Description

Create a new shared memory segment

Usage

```
segment_create(
  size,
  backing = c("auto", "mmap", "shm"),
  path = NULL,
  readonly = FALSE
)
```

Arguments

size	Size of the segment in bytes
backing	Backing type: "auto", "mmap", or "shm"
path	Optional file path for mmap backing (NULL for temp file)
readonly	Create as read-only (after initial write)

Value

A shard_segment object backed by shared memory.

Examples

```
seg <- segment_create(1024 * 1024)
segment_info(seg)
segment_close(seg)
```

---

segment_info *Get segment information*

Description

Get segment information

Usage

```
segment_info(x)
```

Arguments

x	A shard_segment object

Value

A named list with segment metadata including size, backing, path, readonly, and owns.

Examples

```
seg <- segment_create(1024)
segment_info(seg)
segment_close(seg)
```

segment_open *Open an existing shared memory segment*

Description

Open an existing shared memory segment

Usage

```
segment_open(path, backing = c("mmap", "shm"), readonly = TRUE)
```

Arguments

path	Path or shm name of the segment
backing	Backing type: "mmap" or "shm"
readonly	Open as read-only

Value

A shard_segment object attached to the existing segment.

Examples

```
seg <- segment_create(1024, backing = "mmap")
path <- segment_path(seg)
seg2 <- segment_open(path, backing = "mmap", readonly = TRUE)
segment_close(seg2, unlink = FALSE)
segment_close(seg)
```

segment_path *Get the path or name of a segment*

Description

Get the path or name of a segment

Usage

```
segment_path(x)
```

Arguments

x	A shard_segment object

Value

The path string, or NULL for anonymous segments.

Examples

```
seg <- segment_create(1024, backing = "mmap")
segment_path(seg)
segment_close(seg)
```

---

segment_protect *Make a segment read-only*

Description

Make a segment read-only

Usage

```
segment_protect(x)
```

Arguments

x	A shard_segment object

Value

The shard_segment object, invisibly.

Examples

```
seg <- segment_create(1024)
segment_protect(seg)
segment_close(seg)
```

segment_read *Read raw data from a segment*

Description

Read raw data from a segment

Usage

```
segment_read(x, offset = 0, size = NULL)
```

Arguments

x	A shard_segment object
offset	Byte offset to start reading (0-based)
size	Number of bytes to read

Value

A raw vector containing the bytes read from the segment.

Examples

```
seg <- segment_create(1024)
segment_write(seg, as.integer(1:4), offset = 0)
segment_read(seg, offset = 0, size = 16)
segment_close(seg)
```

segment_report *Shared Memory Segment Report*

Description

Generates a report of active shared memory segments in the current session.

Usage

```
segment_report()
```

Details

This function reports on segments that are currently accessible. Note that segments are automatically cleaned up when their R objects are garbage collected, so this only shows segments with live references.

Value

An S3 object of class `shard_report` with type `"segment"` containing:

- `type`: "segment"
- `timestamp`: When the report was generated
- `n_segments`: Number of tracked segments
- `total_bytes`: Total bytes across all segments
- `segments`: List of segment details

Examples

```
segment_report()
```

`segment_size` *Get the size of a segment*

Description

Get the size of a segment

Usage

```
segment_size(x)
```

Arguments

x	A shard_segment object

Value

Size in bytes as a numeric scalar.

Examples

```
seg <- segment_create(1024)
segment_size(seg)
segment_close(seg)
```

segment_write *Write data to a segment*

Description

Write data to a segment

Usage

```
segment_write(x, data, offset = 0)
```

Arguments

x	A shard_segment object
data	Data to write (raw, numeric, integer, or logical vector)
offset	Byte offset to start writing (0-based)

Value

Number of bytes written, invisibly.

Examples

```
seg <- segment_create(1024)
segment_write(seg, as.integer(1:10), offset = 0)
segment_close(seg)
```

set_affinity *Set CPU affinity for the current process*

Description

Intended to be called inside a worker process (e.g., via `clusterCall()`). On unsupported platforms, returns FALSE.

Usage

```
set_affinity(cores)
```

Arguments

cores	Integer vector of 0-based CPU core ids.

Value

A logical scalar; `TRUE` on success, `FALSE` if not supported.

Examples

```
affinity_supported()
```

`shards` *Shard Descriptor Creation*

Description

Create shard descriptors for parallel execution with autotuning.

Produces shard descriptors (index ranges) for use with `shard_map()`. Supports autotuning based on worker count and memory constraints.

Usage

```
shards(
  n,
  block_size = "auto",
  workers = NULL,
  strategy = c("contiguous", "strided"),
  min_shards_per_worker = 4L,
  max_shards_per_worker = 64L,
  scratch_bytes_per_item = 0,
  scratch_budget = 0
)
```

Arguments

`n`	Integer. Total number of items to shard.
`block_size`	Block size specification. Can be: • `"auto"` (default): Autotune based on worker count • Integer: Explicit number of items per shard • Character: Human-readable like `"1K"`, `"10K"`
`workers`	Integer. Number of workers for autotuning (default: pool size or detectCores - 1).
`strategy`	Sharding strategy: `"contiguous"` (default) or `"strided"`.
`min_shards_per_worker`	Integer. Minimum shards per worker for load balancing (default 4).
`max_shards_per_worker`	Integer. Maximum shards per worker to limit overhead (default 64).
`scratch_bytes_per_item`	Numeric. Expected scratch memory per item for memory budgeting.
`scratch_budget`	Character or numeric. Total scratch memory budget (e.g., "1GB").

Value

A shard_descriptor object containing:

- n: Total items
- block_size: Computed block size
- strategy: Strategy used
- shards: List of shard descriptors with id, start, end, idx fields

Examples

```
blocks <- shards(1e6, workers = 8)
length(blocks$shards)

blocks <- shards(1000, block_size = 100)

blocks$shards[[1]]$idx
```

---

shards_list *Create Shards from an Explicit Index List*

Description

Constructs a shard_descriptor from a user-supplied list of index vectors. This is useful for non-contiguous workloads like searchlights/feature sets where each shard operates on an arbitrary subset.

Usage

```
shards_list(idxs)
```

Arguments

idxs	List of integer vectors (1-based indices). Each element becomes one shard with fields id, idx, and len.

Value

A shard_descriptor list describing the chunk layout.

Examples

```
sh <- shards_list(list(1:10, 11:20, 21:30))
length(sh)
```

shard_apply_matrix *Apply a Function Over Matrix Columns with Shared Inputs*

Description

A convenience wrapper for the common "per-column apply" pattern. The matrix is shared once and each worker receives a zero-copy column view when possible.

Usage

```
shard_apply_matrix(
  X,
  MARGIN = 2,
  FUN,
  VARS = NULL,
  workers = NULL,
  ...,
  policy = shard_apply_policy()
)
```

Arguments

X	A numeric/integer/logical matrix (or a shared matrix created by share()).
MARGIN	Must be 2 (columns).
FUN	Function of the form function(v, ...) returning a scalar atomic.
VARS	Optional named list of extra variables. Large atomic VARS are auto-shared based on policy$auto_share_min_bytes.
workers	Number of workers (passed to shard_map()).
...	Additional arguments forwarded to FUN.
policy	A shard_apply_policy() object.

Details

Current limitation: MARGIN must be 2 (columns). Row-wise apply would require strided/gather slicing and is intentionally explicit in shard via views/kernels.

Value

An atomic vector of length ncol(X) with the results.

Examples

```
X <- matrix(rnorm(400), 20, 20)
shard_apply_matrix(X, MARGIN = 2, FUN = mean)
pool_stop()
```

shard_apply_policy *Apply Wrapper Policy*

Description

Centralizes safe defaults and guardrails for apply/lapply convenience wrappers.

Usage

```
shard_apply_policy(
  auto_share_min_bytes = "1MB",
  max_gather_bytes = "256MB",
  cow = c("deny", "audit", "allow"),
  profile = c("default", "memory", "speed"),
  block_size = "auto",
  backing = c("auto", "mmap", "shm")
)
```

Arguments

auto_share_min_bytes	Minimum object size for auto-sharing (default "1MB").
max_gather_bytes	Maximum estimated gathered result bytes before refusing to run (default "256MB").
cow	Copy-on-write policy for borrowed inputs. One of "deny", "audit", or "allow". Default "deny".
profile	Execution profile passed through to shard_map(). One of "default", "memory", or "speed". Default "default".
block_size	Shard block size for apply-style workloads. Default "auto".
backing	Backing type used when auto-sharing ("auto", "mmap", "shm").

Value

An object of class shard_apply_policy.

Examples

```
cfg <- shard_apply_policy()
cfg
```

shard_crossprod *Parallel crossprod() using shard views + output buffers*

Description

Computes `crossprod(X, Y)` (i.e. `t(X) %*% Y`) using:

- shared/mmap-backed inputs (one copy),
- block views (no slice materialization),
- BLAS-3 dgemm in each tile,
- an explicit shared output buffer (no gather/bind spikes).

Usage

```
shard_crossprod(
  X,
  Y,
  workers = NULL,
  block_x = "auto",
  block_y = "auto",
  backing = c("mmap", "shm"),
  materialize = c("auto", "never", "always"),
  materialize_max_bytes = 512 * 1024^2,
  diagnostics = TRUE
)
```

Arguments

`X, Y`	Double matrices with the same number of rows.
`workers`	Number of worker processes.
`block_x, block_y`	Tile sizes over `ncol(X)` and `ncol(Y)`. Use `"auto"` (default) to autotune on the current machine.
`backing`	Backing for shared inputs and output buffer (`"mmap"` or `"shm"`).
`materialize`	Whether to return the result as a standard R matrix: `"never"` (return buffer handle), `"always"`, or `"auto"` (materialize if estimated output size is below `materialize_max_bytes`).
`materialize_max_bytes`	Threshold for `"auto"` materialization.
`diagnostics`	Whether to collect shard_map diagnostics.

Details

This is intended as an ergonomic entry point for the "wow" path: users shouldn't have to manually call `share()`, `view_block()`, `buffer()`, `tiles2d()`, and `shard_map()` for common patterns.

Value

A list with:

- `buffer`: shard_buffer for the result (p x v)
- `value`: materialized matrix if requested, otherwise NULL
- `run`: the underlying shard_result from shard_map
- `tile`: chosen tile sizes

Examples

```
X <- matrix(rnorm(2000), 100, 20)
Y <- matrix(rnorm(2000), 100, 20)
res <- shard_crossprod(X, Y, block_x = 50, block_y = 10, workers = 2)
pool_stop()
res$value
```

---

shard_get_adapter *Get Adapter for an Object*

Description

Retrieves the registered adapter for an object's class. Checks all classes in the object's class hierarchy, returning the first matching adapter.

Usage

```
shard_get_adapter(x)
```

Arguments

`x`	An R object.

Value

The adapter list if one is registered for any of the object's classes, or NULL if no adapter is registered.

Examples

```
shard_get_adapter(1:10)
```

shard_lapply_shared *Apply a Function Over a List with Optional Auto-Sharing*

Description

A convenience wrapper for list workloads that need supervision and shared inputs. Large atomic list elements are auto-shared based on policy.

Usage

```
shard_lapply_shared(
  x,
  FUN,
  VARS = NULL,
  workers = NULL,
  ...,
  policy = shard_apply_policy()
)
```

Arguments

x	A list.
FUN	Function of the form function(el, ...).
VARS	Optional named list of extra variables (auto-shared when large).
workers	Number of workers (passed to shard_map()).
...	Additional arguments forwarded to FUN.
policy	A shard_apply_policy() object.

Details

This wrapper enforces guardrails to avoid accidental huge gathers: it estimates the total gathered result size from a probe call and refuses to run if it exceeds policy$max_gather_bytes.

Value

A list of results, one per element of x.

Examples

```
res <- shard_lapply_shared(as.list(1:4), function(x) x^2)
pool_stop()
res
```

shard_list_adapters *List Registered Adapters*

Description

Returns a character vector of all classes with registered adapters.

Usage

```
shard_list_adapters()
```

Value

Character vector of class names with registered adapters.

Examples

```
shard_list_adapters()
```

shard_map *Parallel Execution with shard_map*

Description

Core parallel execution engine with supervision, shared inputs, and output buffers.

Executes a function over shards in parallel with worker supervision, shared inputs, and explicit output buffers. This is the primary entry point for shard's parallel execution model.

Usage

```
shard_map(
  shards,
  fun = NULL,
  borrow = list(),
  out = list(),
  kernel = NULL,
  scheduler_policy = NULL,
  autotune = NULL,
  dispatch_mode = c("rpc_chunked", "shm_queue"),
  dispatch_opts = NULL,
  workers = NULL,
  chunk_size = 1L,
  profile = c("default", "memory", "speed"),
  mem_cap = "2GB",
  recycle = TRUE,
```

```
  cow = c("deny", "audit", "allow"),
  seed = NULL,
  diagnostics = TRUE,
  packages = NULL,
  init_expr = NULL,
  timeout = 3600,
  max_retries = 3L,
  health_check_interval = 10L
)
```

Arguments

shards	A shard_descriptor from shards(), or an integer N to auto-generate shards.
fun	Function to execute per shard. Receives the shard descriptor as first argument, followed by borrowed inputs and outputs. You can also select a registered kernel via kernel= instead of providing fun=.
borrow	Named list of shared inputs. These are exported to workers once and reused across shards. Treated as read-only by default.
out	Named list of output buffers (from buffer()). Workers write results directly to these buffers.
kernel	Optional. Name of a registered kernel (see list_kernels()). If provided, fun must be NULL.
scheduler_policy	Optional list of scheduling hints (advanced). Currently: • max_huge_concurrency: cap concurrent chunks whose kernel footprint is classified as "huge" (see register_kernel()).
autotune	Optional. Online autotuning for scalar-N sharding (advanced). When shards is an integer N, shard_map can adjust shard block sizes over time based on observed wall time and worker RSS. Accepted values: • NULL (default): enable online autotuning for shard_map(N, ...), off for precomputed shard descriptors. • TRUE / "online": force online autotuning (only applies when shards is an integer N). • FALSE / "none": disable autotuning. • a list: list(mode="online", max_rounds=..., probe_shards_per_worker=..., min_shard_time=...)
dispatch_mode	Dispatch mode (advanced). "rpc_chunked" is the default supervised socket-based dispatcher. "shm_queue" is an opt-in fast mode that uses a shared-memory task queue to reduce per-task overhead for tiny tasks. In v1, "shm_queue" is only supported for shard_map(N, ...) with chunk_size=1 and is intended for out-buffer/sink workflows (results are not gathered).
dispatch_opts	Optional list of dispatch-mode specific knobs (advanced). Currently: • For dispatch_mode="rpc_chunked":

  - auto_table: logical. If TRUE, shard_map treats data.frame/tibble return values as row-group outputs and writes them to a table sink automatically (one partition per shard id). This avoids building a large list of tibbles and calling bind_rows() on the master. Requires out= to be empty (use explicit out=list(sink=table_sink(...)) otherwise).
  - auto_table_materialize: "never", "auto", or "always" (default "auto").
  - auto_table_max_bytes: numeric/integer. For "auto", materialize only if estimated output size <= this threshold (default 256MB).
  - auto_table_mode: "row_groups" (default) or "partitioned".
  - auto_table_path: optional output directory (default tempdir()).
  - auto_table_format: "auto", "rds" (default), or "native".
  - auto_table_schema: optional shard_schema for validation/native encoding.
- For dispatch_mode="shm_queue":
  - block_size: integer. If provided, overrides the default heuristic for contiguous shard block sizing.
  - queue_backing: one of "mmap" or "shm" (default "mmap").
  - error_log: logical. If TRUE, workers write a bounded per-worker error log to disk to aid debugging failed tasks (default FALSE).
  - error_log_max_lines: integer. Maximum lines per worker in the error log (default 100).

workers
: Integer. Number of worker processes. If NULL, uses existing pool or creates one with detectCores() - 1.

chunk_size
: Integer. Shards to batch per worker dispatch (default 1). Higher values reduce RPC overhead but may hurt load balancing.

profile
: Execution profile: "default", "memory" (aggressive recycling), or "speed" (minimal overhead). With profile="speed", shard_map will automatically enable dispatch_mode="shm_queue" when possible for shard_map(N, ...) out-buffer workflows (scalar N, chunk_size=1), unless dispatch_mode is explicitly specified.

mem_cap
: Memory cap per worker (e.g., "2GB"). Workers exceeding this are recycled.

recycle
: Logical or numeric. If TRUE, recycle workers on RSS drift. If numeric, specifies drift threshold (default 0.5 = 50% growth).

cow
: Copy-on-write policy for borrowed inputs: "deny" (error on mutation), "audit" (detect and flag), or "allow" (permit with tracking).

seed
: Integer. RNG seed for reproducibility. If NULL, no seed is set.

diagnostics
: Logical. Collect detailed diagnostics (default TRUE).

packages
: Character vector. Additional packages to load in workers.

init_expr
: Expression to evaluate in each worker on startup.

timeout
: Numeric. Seconds to wait for each shard (default 3600).

max_retries
: Integer. Maximum retries per shard on failure (default 3).

health_check_interval
: Integer. Check worker health every N shards (default 10).

Value

A `shard_result` object containing:

- `results`: List of results from each shard (if fun returns values)
- `failures`: Any permanently failed shards
- `diagnostics`: Timing, memory, and worker statistics
- `pool_stats`: Pool-level statistics

Examples

```
blocks <- shards(1000, workers = 2)
result <- shard_map(blocks, function(shard) {
  sum(shard$idx^2)
}, workers = 2)
pool_stop()
```

---

shard_reduce *Streaming Reductions over Shards*

Description

Reduce shard results without gathering all per-shard returns on the master.

`shard_reduce()` executes `map()` over shards in parallel and combines results using an associative `combine()` function. Unlike `shard_map()`, it does not accumulate all per-shard results on the master; it streams partials as chunks complete.

Usage

```
shard_reduce(
  shards,
  map,
  combine,
  init,
  borrow = list(),
  out = list(),
  workers = NULL,
  chunk_size = 1L,
  profile = c("default", "memory", "speed"),
  mem_cap = "2GB",
  recycle = TRUE,
  cow = c("deny", "audit", "allow"),
  seed = NULL,
  diagnostics = TRUE,
  packages = NULL,
  init_expr = NULL,
```

```
  timeout = 3600,
  max_retries = 3L,
  health_check_interval = 10L
)
```

Arguments

shards	A shard_descriptor from shards(), or an integer N.
map	Function executed per shard. Receives shard descriptor as first argument, followed by borrowed inputs and outputs.
combine	Function (acc, value) -> acc used to combine results. Should be associative for deterministic behavior under chunking.
init	Initial accumulator value.
borrow	Named list of shared inputs (same semantics as shard_map()).
out	Named list of output buffers/sinks (same semantics as shard_map()).
workers	Number of worker processes.
chunk_size	Shards to batch per worker dispatch (default 1).
profile	Execution profile (same semantics as shard_map()).
mem_cap	Memory cap per worker (same semantics as shard_map()).
recycle	Worker recycling policy (same semantics as shard_map()).
cow	Copy-on-write policy for borrowed inputs (same semantics as shard_map()).
seed	RNG seed for reproducibility.
diagnostics	Logical; collect diagnostics (default TRUE).
packages	Additional packages to load in workers.
init_expr	Expression to evaluate in each worker on startup.
timeout	Seconds to wait for each chunk.
max_retries	Maximum retries per chunk.
health_check_interval	Check worker health every N completions.

Details

For performance and memory efficiency, reduction is performed in two stages:

1. per-chunk partial reduction inside each worker, and
2. streaming combine of partials on the master.

Value

A shard_reduce_result with fields:

- value: final accumulator
- failures: any permanently failed chunks
- diagnostics: run telemetry including reduction stats
- queue_status, pool_stats

Examples

```
res <- shard_reduce(
  100L,
  map = function(s) sum(s$idx),
  combine = function(acc, x) acc + x,
  init = 0,
  workers = 2
)
pool_stop()
res$value
```

`shard_register_adapter` *Register an Adapter for Class-Specific Traversal*

Description

Registers a custom adapter for a specific class. When deep sharing encounters an object of this class, it will use the adapter's `children()` function to extract shareable components instead of generic traversal.

Usage

```
shard_register_adapter(class, adapter)
```

Arguments

`class`	A character string naming the class to register the adapter for.
`adapter`	A list containing:
	class Character string matching the `class` parameter.
	children Function taking an object and returning a named list of child objects to traverse.
	replace Function taking the original object and a named list of (potentially shared) children, returning a reconstructed object.
	path_prefix Optional character string prefix for child paths in the sharing plan (default: class name).

Value

Invisibly returns the previous adapter for this class (if any), or NULL if no adapter was registered.

Examples

```
shard_list_adapters()
```

shard_share_hook *Deep Sharing Hook for Custom Classes*

Description

S3 generic that allows classes to customize deep sharing behavior. Override this for your class to control which slots/elements are traversed, force sharing of small objects, or transform objects before traversal.

Usage

```
shard_share_hook(x, ctx)

## Default S3 method:
shard_share_hook(x, ctx)
```

Arguments

x	The object being traversed during deep sharing.
ctx	A context list containing:
	path Current node path string (e.g., "<root>$data@cache")
	class class(x) - the object's class vector
	mode 'strict' or 'balanced' - sharing mode
	min_bytes Minimum size threshold for sharing
	types Character vector of enabled types for sharing
	deep Logical, always TRUE when hook is called

Value

A list with optional fields:

skip_slots Character vector of S4 slot names to not traverse

skip_paths Character vector of paths to not traverse

force_share_paths Character vector of paths to force share (ignore min_bytes)

rewrite Function(x) -> x to transform object before traversal

Return an empty list for default behavior (no customization).

Examples

```
shard_share_hook.MyModelClass <- function(x, ctx) {
    list(
        skip_slots = "cache",
        force_share_paths = paste0(ctx$path, "@coefficients")
    )
}
```

```
shard_share_hook.LazyData <- function(x, ctx) {
    list(
        rewrite = function(obj) {
            obj$data <- as.matrix(obj$data)
            obj
        }
    )
}
```

---

shard_unregister_adapter *Unregister an Adapter*

Description

Removes a previously registered adapter for a class. After unregistration, objects of this class will use default traversal behavior during deep sharing.

Usage

```
shard_unregister_adapter(class)
```

Arguments

class	A character string naming the class to unregister.

Value

Invisibly returns the removed adapter, or NULL if no adapter was registered for this class.

Examples

```
shard_list_adapters()
```

---

share *Zero-Copy Shared Objects*

Description

Create shared memory representations of R objects for efficient parallel access without duplication.

Creates a shared memory representation of an R object. The object is serialized once and can be accessed by multiple worker processes without copying.

Usage

```
share(
  x,
  backing = c("auto", "mmap", "shm"),
  readonly = TRUE,
  name = NULL,
  deep = FALSE,
  min_bytes = 64 * 1024 * 1024,
  cycle = c("error", "skip"),
  mode = c("balanced", "strict")
)
```

Arguments

x	An R object to share. Supports vectors, matrices, arrays, lists, data frames, and any object that can be serialized with `serialize()`.
backing	Backing type: "auto" (default), "mmap", or "shm". • `"auto"`: Let the system choose the best option. • `"mmap"`: File-backed memory mapping. Most portable. • `"shm"`: POSIX shared memory or Windows named mapping.
readonly	Logical. If TRUE (default), the segment is protected after writing, making it read-only. Set to FALSE only if you need to modify the shared data (advanced use case).
name	Optional name for the shared object. If NULL (default), a unique name is generated. Named shares can be opened by name in other processes.
deep	Logical. If TRUE, recursively traverse lists and data.frames, sharing individual components that meet the size threshold. When FALSE (default), the entire object is serialized as one unit.
min_bytes	Minimum size in bytes for an object to be shared when deep=TRUE. Objects smaller than this threshold are kept in-place. Default is 64MB (64 * 1024 * 1024).
cycle	How to handle cyclic references when deep=TRUE. Either "error" (default) to stop with an error, or "skip" to skip cyclic references.
mode	Sharing mode when deep=TRUE. Either "balanced" (default) to continue on hook errors and non-shareable types, or "strict" to error.

Details

The `share()` function is the primary high-level API for creating zero-copy shared inputs. When you share an object:

1. The object is serialized into a shared memory segment
2. The segment is marked read-only (protected)
3. A lightweight handle is returned that can be passed to workers
4. Workers attach to the segment and deserialize on demand

This approach eliminates per-worker duplication of large inputs. The data exists once in shared memory, and all workers read from the same location.

Immutability Contract: Shared objects are immutable by design. Any attempt to modify shared data in a worker will fail. This guarantees deterministic behavior and prevents accidental copy-on-write.

Value

A `shard_shared` object (when deep=FALSE) or `shard_deep_shared` object (when deep=TRUE) containing:

- `path`: The path or name of the shared segment
- `backing`: The backing type used
- `size`: Total size in bytes
- `readonly`: Whether the segment is protected
- `class_info`: Original class information

See Also

`segment_create` for low-level segment operations, `pool_create` for worker pool management.

Examples

```
mat <- matrix(rnorm(1e4), nrow = 100)
shared_mat <- share(mat)
recovered <- fetch(shared_mat)
identical(mat, recovered)
close(shared_mat)
```

---

shared_advise *Advise access pattern for a shared input vector/matrix*

Description

Advise access pattern for a shared input vector/matrix

Usage

```
shared_advise(
  x,
  advice = c("normal", "sequential", "random", "willneed", "dontneed")
)
```

Arguments

`x`	A shard shared vector (from `share()`).
`advice`	See `segment_advise()`.

Value

A logical scalar; TRUE if the OS accepted the hint.

Examples

```
x <- as_shared(1:100)
shared_advise(x, "sequential")
```

---

shared_diagnostics *Get diagnostics for a shared vector*

Description

Get diagnostics for a shared vector

Usage

```
shared_diagnostics(x)
```

Arguments

x	A shard ALTREP vector

Value

A list with diagnostic information:

dataptr_calls Number of times DATAPTR was accessed

materialize_calls Number of times vector was copied to standard R vector

length Number of elements

offset Byte offset into underlying segment

readonly Whether write access is prevented

type R type of the vector

Examples

```
seg <- segment_create(400)
segment_write(seg, 1:100, offset = 0)
x <- shared_vector(seg, "integer", length = 100)

sum(x)

shared_diagnostics(x)
```

shared_info *Get Information About a Shared Object*

Description

Get Information About a Shared Object

Usage

```
shared_info(x)
```

Arguments

x	A shard_shared object.

Value

A named list with fields path, backing, size, readonly, class_info, and segment_info.

Examples

```
shared <- share(1:100)
shared_info(shared)
close(shared)
```

shared_reset_diagnostics *Reset diagnostic counters for a shared vector*

Description

Reset diagnostic counters for a shared vector

Usage

```
shared_reset_diagnostics(x)
```

Arguments

x	A shard ALTREP vector

Value

x (invisibly)

Examples

```
seg <- segment_create(400)
segment_write(seg, 1:100, offset = 0)
x <- shared_vector(seg, "integer", length = 100)

sum(x)
shared_diagnostics(x)$dataptr_calls

shared_reset_diagnostics(x)
shared_diagnostics(x)$dataptr_calls
```

---

shared_segment *Get the underlying segment from a shared vector*

Description

Get the underlying segment from a shared vector

Usage

```
shared_segment(x)
```

Arguments

x	A shard ALTREP vector

Value

A `shard_segment` S3 object wrapping the underlying segment

Examples

```
x <- as_shared(1:100)
shared_segment(x)
```

shared_vector *Create a shared vector from a segment*

Description

Create a shared vector from a segment

Usage

```
shared_vector(
  segment,
  type = c("double", "integer", "logical", "raw"),
  offset = 0,
  length = NULL,
  readonly = TRUE,
  cow = NULL
)
```

Arguments

segment	A shard_segment object
type	Vector type: "integer", "double"/"numeric", "logical", or "raw"
offset	Byte offset into segment (default: 0)
length	Number of elements. If NULL, calculated from segment size.
readonly	If TRUE, prevent write access via DATAPTR (default: TRUE)
cow	Copy-on-write policy for mutation attempts. One of "deny", "audit", or "allow". If NULL, defaults to "deny" when readonly=TRUE and "allow" otherwise.

Value

An ALTREP vector backed by shared memory

Examples

```
seg <- segment_create(400)
segment_write(seg, 1:100, offset = 0)

x <- shared_vector(seg, "integer", length = 100)
x[1:10]

shared_diagnostics(x)
```

shared_view *Create a view (subset) of a shared vector*

Description

Create a view (subset) of a shared vector

Usage

```
shared_view(x, start, length)
```

Arguments

x	A shard ALTREP vector
start	Start index (1-based, like R)
length	Number of elements

Value

An ALTREP view into the same shared memory

Examples

```
seg <- segment_create(800)
segment_write(seg, 1:100, offset = 0)
x <- shared_vector(seg, "integer", length = 100)

y <- shared_view(x, start = 10, length = 11)
y[1]
```

share_open *Open an Existing Shared Object by Path*

Description

Opens a shared object that was created by another process. This is useful for workers that need to attach to shared data without having the original shard_shared object.

Usage

```
share_open(path, backing = c("mmap", "shm"), size = NULL)
```

Arguments

path	Path to the shared segment.
backing	Backing type: "mmap" or "shm".
size	Size of the segment in bytes. If NULL, attempts to detect.

Value

A shard_shared object attached to the existing segment.

Examples

```
shared <- share(1:50)
info <- shared_info(shared)
reopened <- share_open(info$path, backing = "mmap")
close(reopened)
close(shared)
```

---

stream_count *Stream row count*

Description

Stream row count

Usage

```
stream_count(x)
```

Arguments

x	A shard_row_groups or shard_dataset handle.

Value

A single integer giving the total number of rows across all partitions.

Examples

```
s <- schema(x = float64())
sink <- table_sink(s, mode = "row_groups")
table_write(sink, 1L, data.frame(x = rnorm(5)))
rg <- table_finalize(sink)
stream_count(rg)
```

stream_filter *Stream-filter a dataset/row-groups into a new partitioned dataset*

Description

Reads each partition, filters rows, and writes a new partitioned dataset. Output is written as one partition per input partition (empty partitions are allowed). This avoids materializing all results.

Usage

```
stream_filter(x, predicate, path = NULL, ...)
```

Arguments

x	A shard_row_groups or shard_dataset handle.
predicate	Function (chunk, ...) -> logical row mask (length == nrow(chunk)).
path	Output directory. If NULL, a temp dir is created.
...	Passed to predicate().

Value

A shard_dataset handle pointing to the filtered partitions.

Examples

```
s <- schema(x = float64())
sink <- table_sink(s, mode = "row_groups")
table_write(sink, 1L, data.frame(x = c(1.0, 2.0, 3.0)))
rg <- table_finalize(sink)
filtered <- stream_filter(rg, predicate = function(chunk) chunk$x > 1.5)
```

stream_group_count *Stream group-wise count*

Description

Counts rows per group across partitions without collecting. Optimized for factor groups (factor_col()).

Usage

```
stream_group_count(x, group)
```

Arguments

x	A shard_row_groups or shard_dataset handle.
group	Group column name (recommended: factor_col()).

Value

A data.frame with columns group (factor) and n (integer).

Examples

```
s <- schema(g = factor_col(c("a", "b")), x = float64())
sink <- table_sink(s, mode = "row_groups")
table_write(sink, 1L,
  data.frame(g = factor(c("a", "b", "a"), levels = c("a", "b")), x = c(1, 2, 3)))
rg <- table_finalize(sink)
stream_group_count(rg, "g")
```

---

stream_group_sum *Stream group-wise sum*

Description

Computes sum(value) by group across partitions without collecting. This is optimized for factor groups (factor_col()).

Usage

```
stream_group_sum(x, group, value, na_rm = TRUE)
```

Arguments

x	A shard_row_groups or shard_dataset handle.
group	Group column name (recommended: factor_col()).
value	Numeric column name to sum.
na_rm	Logical; drop rows where value is NA (default TRUE).

Value

A data.frame with columns group (factor) and sum (numeric).

Examples

```
s <- schema(g = factor_col(c("a", "b")), x = float64())
sink <- table_sink(s, mode = "row_groups")
table_write(sink, 1L,
  data.frame(g = factor(c("a", "b", "a"), levels = c("a", "b")), x = c(1, 2, 3)))
rg <- table_finalize(sink)
stream_group_sum(rg, "g", "x")
```

---

stream_map *Stream over row-groups/datasets and map*

Description

Applies `f()` to each partition and returns the list of per-partition results. This is still much cheaper than collecting the full dataset when `f()` returns a small summary per partition.

Usage

```
stream_map(x, f, ...)

## S3 method for class 'shard_row_groups'
stream_map(x, f, ...)

## S3 method for class 'shard_dataset'
stream_map(x, f, ...)
```

Arguments

`x`	A `shard_row_groups` or `shard_dataset` handle.
`f`	Function `(chunk, ...) -> value`.
`...`	Passed to `f()`.

Value

A list of per-partition values, one element per row-group file.

Examples

```
s <- schema(x = float64())
sink <- table_sink(s, mode = "row_groups")
table_write(sink, 1L, data.frame(x = rnorm(5)))
rg <- table_finalize(sink)
nrows <- stream_map(rg, nrow)
```

`stream_reduce` *Stream over row-groups/datasets and reduce*

Description

Applies `f()` to each partition (row-group) and combines results with `combine()` into a single accumulator. This keeps peak memory bounded by the largest single partition (plus your accumulator).

Usage

```
stream_reduce(x, f, init, combine, ...)

## S3 method for class 'shard_row_groups'
stream_reduce(x, f, init, combine, ...)

## S3 method for class 'shard_dataset'
stream_reduce(x, f, init, combine, ...)
```

Arguments

`x`	A `shard_row_groups` or `shard_dataset` handle.
`f`	Function `(chunk, ...) -> value` producing a per-partition value.
`init`	Initial accumulator value.
`combine`	Function `(acc, value) -> acc` to update the accumulator.
`...`	Passed to `f()`.

Value

The final accumulator value after processing all partitions.

Examples

```
s <- schema(x = float64())
sink <- table_sink(s, mode = "row_groups")
table_write(sink, 1L, data.frame(x = rnorm(5)))
rg <- table_finalize(sink)
total <- stream_reduce(rg, f = nrow, init = 0L, combine = `+`)
```

stream_sum *Stream sum of a numeric column*

Description

Computes the sum of `col` across all partitions without collecting the full dataset. When partitions are native-encoded, this avoids decoding string columns entirely.

Usage

```
stream_sum(x, col, na_rm = TRUE)
```

Arguments

`x`	A `shard_row_groups` or `shard_dataset` handle.
`col`	Column name to sum.
`na_rm`	Logical; drop NAs (default TRUE).

Value

A single numeric value giving the sum of the column across all partitions.

Examples

```
s <- schema(x = float64())
sink <- table_sink(s, mode = "row_groups")
table_write(sink, 1L, data.frame(x = c(1.0, 2.0, 3.0)))
rg <- table_finalize(sink)
stream_sum(rg, "x")
```

stream_top_k *Stream top-k rows by a numeric column*

Description

Finds the top `k` rows by `col` without collecting the full dataset.

Usage

```
stream_top_k(x, col, k = 10L, decreasing = TRUE, na_drop = TRUE)
```

Arguments

x	A shard_row_groups or shard_dataset handle.
col	Column name to rank by.
k	Number of rows to keep.
decreasing	Logical; TRUE for largest values (default TRUE).
na_drop	Logical; drop rows where col is NA (default TRUE).

Details

For native-encoded partitions, this selects candidate rows using the numeric column without decoding strings, then decodes only the chosen rows for the returned result.

Value

A data.frame (or tibble if the tibble package is installed) with at most k rows ordered by col.

Examples

```
s <- schema(x = float64())
sink <- table_sink(s, mode = "row_groups")
table_write(sink, 1L, data.frame(x = c(3.0, 1.0, 2.0)))
rg <- table_finalize(sink)
stream_top_k(rg, "x", k = 2L)
```

---

succeeded *Check if shard_map Succeeded*

Description

Check if shard_map Succeeded

Usage

```
succeeded(x)
```

Arguments

x	A shard_result object.

Value

Logical. TRUE if no failures.

Examples

```
result <- shard_map(4L, function(shard) shard$idx[[1L]], workers = 2)
pool_stop()
succeeded(result)
```

`table_buffer` *Allocate a fixed-row table buffer*

Description

Allocates a columnar table output: one typed buffer per column, each of length `nrow`. Intended for lock-free disjoint row-range writes in shard_map.

Usage

```
table_buffer(schema, nrow, backing = c("auto", "mmap", "shm"))
```

Arguments

`schema`	A `shard_schema`.
`nrow`	Total number of rows in the final table.
`backing`	Backing type for buffers (`"auto"`, `"mmap"`, `"shm"`).

Value

A `shard_table_buffer` object with one shared buffer per schema column.

Examples

```
s <- schema(x = float64(), y = int32())
tb <- table_buffer(s, nrow = 100L)
```

`table_diagnostics` *Table Diagnostics*

Description

Per-process counters for table writes (number of table_write calls, rows, and bytes written). shard_map uses deltas of these counters to produce run-level diagnostics in copy_report().

Usage

```
table_diagnostics()
```

Value

A list with `writes`, `rows`, and `bytes`.

`table_finalize` *Finalize a table buffer or sink*

Description

For a `shard_table_buffer`, this returns a lightweight in-memory handle (or a materialized data.frame/tibble, depending on `materialize`).

Usage

```
table_finalize(
  target,
  materialize = c("never", "auto", "always"),
  max_bytes = 256 * 1024^2,
  ...
)
```

Arguments

`target`	A `shard_table_buffer` or `shard_table_sink`.
`materialize`	`"never"`, `"auto"`, or `"always"`.
`max_bytes`	For `"auto"`, materialize only if estimated bytes <= max_bytes.
`...`	Reserved for future extensions.

Details

For a `shard_table_sink`, this returns a row-group handle referencing the written partitions (or materializes them if requested).

Value

A `shard_table_handle`, `shard_row_groups`, or materialized data.frame/tibble depending on `target` type and `materialize`.

Examples

```
s <- schema(x = float64(), y = int32())
tb <- table_buffer(s, nrow = 5L)
table_write(tb, idx_range(1, 5), data.frame(x = rnorm(5), y = 1:5))
handle <- table_finalize(tb)
```

table_finalize.shard_table_buffer *Finalize a table buffer*

Description

Finalize a table buffer

Usage

```
## S3 method for class 'shard_table_buffer'
table_finalize(
  target,
  materialize = c("never", "auto", "always"),
  max_bytes = 256 * 1024^2,
  ...
)
```

Arguments

target	A shard_table_buffer.
materialize	"never", "auto", or "always".
max_bytes	For "auto", materialize only if estimated bytes <= max_bytes.
...	Reserved for future extensions.

Value

A shard_table_handle or a materialized data.frame/tibble.

table_finalize.shard_table_sink *Finalize a sink*

Description

Finalize a sink

Usage

```
## S3 method for class 'shard_table_sink'
table_finalize(
  target,
  materialize = c("never", "auto", "always"),
  max_bytes = 256 * 1024^2,
  ...
)
```

Arguments

target	A shard_table_sink.
materialize	"never", "auto", or "always".
max_bytes	For "auto", materialize only if estimated bytes <= max_bytes.
...	Reserved for future extensions.

Value

A shard_row_groups handle (or a materialized data.frame/tibble).

---

table_sink *Create a table sink for row-group or partitioned outputs*

Description

A table sink supports variable-sized outputs without returning large data.frames to the master. Each shard writes a separate row-group file.

Usage

```
table_sink(
  schema,
  mode = c("row_groups", "partitioned"),
  path = NULL,
  format = c("auto", "rds", "native")
)
```

Arguments

schema	A shard_schema. If NULL, a schema-less sink is created (RDS format only). This is primarily intended for doShard/foreach compatibility where output schemas may not be known in advance.
mode	"row_groups" (temp, managed) or "partitioned" (persistent path).
path	Directory to write row-group files. If NULL, a temp dir is created.
format	Storage format for partitions: "rds" (data.frame RDS), "native" (columnar encoding with string offsets+bytes), or "auto" (selects "native" if the schema contains string_col(); otherwise "rds").

Details

v1.1 implementation notes:

- Storage format is per-shard RDS (portable, CRAN-friendly).
- This guarantees bounded master memory during execution; final collection may still be large if you materialize.

Value

A shard_table_sink object.

Examples

```
s <- schema(x = float64(), label = string_col())
sink <- table_sink(s, mode = "row_groups")
```

---

table_write *Write tabular results into a table buffer or sink*

Description

table_write() is the common write path for shard table outputs:

- For fixed-size outputs, write into a shard_table_buffer using a row selector.
- For variable-size outputs, write into a shard_table_sink using a shard id.

Usage

```
table_write(target, rows_or_shard_id, data, ...)
```

Arguments

target	A shard_table_buffer or shard_table_sink.
rows_or_shard_id	For buffers: row selector (idx_range or integer vector). For sinks: shard id (integer).
data	A data.frame or named list matching the schema columns.
...	Reserved for future extensions.

Value

NULL (invisibly).

Examples

```
s <- schema(x = float64(), y = int32())
tb <- table_buffer(s, nrow = 10L)
table_write(tb, idx_range(1, 5), data.frame(x = rnorm(5), y = 1:5))
```

table_write.shard_table_buffer

Write into a table buffer

Description

Write into a table buffer

Usage

```
## S3 method for class 'shard_table_buffer'
table_write(target, rows_or_shard_id, data, ...)
```

Arguments

target	A shard_table_buffer.
rows_or_shard_id	Row selector (idx_range or integer vector).
data	A data.frame or named list matching the schema columns.
...	Reserved for future extensions.

Value

NULL (invisibly).

table_write.shard_table_sink

Write a shard's row-group output

Description

Write a shard's row-group output

Usage

```
## S3 method for class 'shard_table_sink'
table_write(target, rows_or_shard_id, data, ...)
```

Arguments

target	A shard_table_sink.
rows_or_shard_id	Integer shard id used to name the row-group file.
data	A data.frame matching the sink schema.
...	Reserved for future extensions.

Value

NULL (invisibly).

`task_report` *Task Execution Report*

Description

Generates a report of task/chunk execution statistics from a shard_map result.

Usage

```
task_report(result = NULL)
```

Arguments

`result`	A `shard_result` object from `shard_map`.

Value

An S3 object of class `shard_report` with type `"task"` containing:

- `type`: "task"
- `timestamp`: When the report was generated
- `shards_total`: Total number of shards
- `shards_processed`: Number of shards successfully processed
- `shards_failed`: Number of permanently failed shards
- `chunks_dispatched`: Number of chunk batches dispatched
- `total_retries`: Total number of retry attempts
- `duration`: Total execution duration (seconds)
- `throughput`: Shards processed per second
- `queue_status`: Final queue status

Examples

```
res <- shard_map(shards(100, workers = 2), function(s) sum(s$idx), workers = 2)
pool_stop()
task_report(res)
```

utils *Utility Functions*

Description

Internal utilities for shard package.

view *Create a view over a shared matrix*

Description

Create a view over a shared matrix

Usage

```
view(x, rows = NULL, cols = NULL, type = c("auto", "block", "gather"))
```

Arguments

x	A shared (share()d) atomic matrix (double/integer/logical/raw).
rows	Row selector. NULL (all rows) or idx_range().
cols	Column selector. NULL (all cols) or idx_range().
type	View type. "block" or "gather" (or "auto").

Value

A shard_view_block or shard_view_gather object depending on the selectors provided.

Examples

```
m <- share(matrix(1:20, nrow = 4))
v <- view(m, cols = idx_range(1, 2))
```

views *Zero-copy Views*

Description

Views are explicit slice descriptors over shared arrays/matrices. They avoid creating slice-sized allocations (e.g. Y[, a:b]) by carrying only metadata plus a reference to the shared backing.

Details

This is a low-level optimization handle: arbitrary base R operations may materialize a view; use materialize() explicitly when you want a standard matrix/array.

view_block *Create a contiguous block view*

Description

Create a contiguous block view

Usage

```
view_block(x, rows = NULL, cols = NULL)
```

Arguments

x	A shared (share()d) atomic matrix.
rows	NULL or idx_range().
cols	NULL or idx_range().

Value

A shard_view_block object representing the contiguous block slice.

Examples

```
m <- share(matrix(1:20, nrow = 4))
v <- view_block(m, cols = idx_range(1, 2))
```

view_diagnostics *View diagnostics*

Description

Returns global counters for view creation/materialization. This is a simple first step; in future this should be integrated into shard_map run-level diagnostics.

Usage

```
view_diagnostics()
```

Value

A list with counters.

view_gather *Create a gather (indexed) view over a shared matrix*

Description

Gather views describe non-contiguous column (or row) subsets without allocating a slice-sized matrix. shard-aware kernels can then choose to pack the requested indices into scratch explicitly (bounded and reportable) or run gather-aware compute paths.

Usage

```
view_gather(x, rows = NULL, cols)
```

Arguments

x	A shared (share()d) atomic matrix (double/integer/logical/raw).
rows	Row selector. NULL (all rows) or idx_range().
cols	Integer vector of column indices (1-based).

Details

v1 note: only column-gather views are implemented (rows may be NULL or idx_range()).

Value

A shard_view_gather object describing the indexed column view.

Examples

```
m <- share(matrix(1:20, nrow = 4))
v <- view_gather(m, cols = c(1L, 3L))
```

view_info *Introspection for a view*

Description

Returns metadata about a view without forcing materialization.

Usage

```
view_info(v)
```

Arguments

v	A shard view.

Value

A named list with fields: dtype, dim, slice_dim, rows, cols, layout, fast_path, nbytes_est, and base_is_shared.

Examples

```
m <- share(matrix(1:20, nrow = 4))
v <- view_block(m, cols = idx_range(1, 2))
view_info(v)
```

---

worker *Individual Worker Control*

Description

Spawn, monitor, and control individual R worker processes.

[.shard_buffer *Extract Buffer Elements*

Description

Extract Buffer Elements

Usage

```
## S3 method for class 'shard_buffer'
x[i, j, ..., drop = TRUE]
```

Arguments

x	A shard_buffer object.
i	Index or indices.
j	Optional second index (for matrices).
...	Additional indices (for arrays).
drop	Whether to drop dimensions.

Value

A vector or array of values read from the buffer.

Examples

```
buf <- buffer("double", dim = 10)
buf[1:5] <- 1:5
buf[1:3]
buffer_close(buf)
```

`[.shard_descriptor` *Subset Shard Descriptor*

Description

Subset Shard Descriptor

Usage

```
## S3 method for class 'shard_descriptor'
x[i]
```

Arguments

`x`	A shard_descriptor object.
`i`	Index or indices.

Value

A subset of the object.

Examples

```
sh <- shards(100, block_size = 25)
sh[1:2]
```

`[.shard_descriptor_lazy` *Subset a shard_descriptor_lazy Object*

Description

Subset a shard_descriptor_lazy Object

Usage

```
## S3 method for class 'shard_descriptor_lazy'
x[i]
```

Arguments

x	A shard_descriptor_lazy object.
i	Index or indices.

Value

A subset of the object.

Examples

```
sh <- shards(100, block_size = 25)
sh[1:2]
```

[<-.shard_buffer *Assign to Buffer Elements*

Description

Assign to Buffer Elements

Usage

```
## S3 replacement method for class 'shard_buffer'
x[i, j, ...] <- value
```

Arguments

x	A shard_buffer object.
i	Index or indices.
j	Optional second index (for matrices).
...	Additional indices (for arrays).
value	Values to assign.

Value

The modified shard_buffer object, invisibly.

Examples

```
buf <- buffer("double", dim = 10)
buf[1:5] <- rnorm(5)
buffer_close(buf)
```

[<-.shard_shared_vector *Subset-assign a Shared Vector*

Description

Replacement method for `shard_shared_vector`. Raises an error if the copy-on-write policy is `"deny"`.

Usage

```
## S3 replacement method for class 'shard_shared_vector'
x[...] <- value
```

Arguments

`x`	A `shard_shared_vector`.
`...`	Indices.
`value`	Replacement value.

Value

The modified object `x`.

[[.shard_descriptor *Get Single Shard*

Description

Get Single Shard

Usage

```
## S3 method for class 'shard_descriptor'
x[[i]]
```

Arguments

`x`	A shard_descriptor object.
`i`	Index.

Value

A subset of the object.

Examples

```
sh <- shards(100, block_size = 25)
sh[[1]]
```

`[[.shard_descriptor_lazy` *Extract a Single Shard from a shard_descriptor_lazy Object*

Description

Extract a Single Shard from a shard_descriptor_lazy Object

Usage

```
## S3 method for class 'shard_descriptor_lazy'
x[[i]]
```

Arguments

`x`	A `shard_descriptor_lazy` object.
`i`	Index.

Value

A subset of the object.

Examples

```
sh <- shards(100, block_size = 25)
sh[[1]]
```

`[[<-.shard_shared_vector` *Double-bracket Subset-assign a Shared Vector*

Description

Replacement method for `shard_shared_vector`. Raises an error if the copy-on-write policy is `"deny"`.

Usage

```
## S3 replacement method for class 'shard_shared_vector'
x[[...]] <- value
```

Arguments

x	A shard_shared_vector.
...	Indices.
value	Replacement value.

Value

The modified object x.

Index