

# Package: shard (via r-universe)

June 4, 2026

**Type** Package

**Title** Deterministic, Zero-Copy Parallel Execution for R

**Version** 0.1.1

**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** <https://cran.r-universe.dev>

**Date/Publication** 2026-04-05 21:50:44 UTC

**RemoteUrl** <https://github.com/cran/shard>

**RemoteRef** HEAD

**RemoteSha** 5be9b725f77d19f3070cd7dc10d2c6b3ccda8554

## Contents

[.shard_buffer . . . . .	6
[.shard_descriptor . . . . .	6
[.shard_descriptor_lazy . . . . .	7
[[.shard_descriptor . . . . .	8
[[.shard_descriptor_lazy . . . . .	8
[[<-.shard_shared_vector . . . . .	9
<-.shard_buffer . . . . .	9
<-.shard_shared_vector . . . . .	10
adapter . . . . .	11
affinity . . . . .	11
affinity_supported . . . . .	12
altrep . . . . .	12
arena . . . . .	13
arena_depth . . . . .	14
as.array.shard_buffer . . . . .	15
as.double.shard_buffer . . . . .	15
as.integer.shard_buffer . . . . .	16
as.logical.shard_buffer . . . . .	17
as.matrix.shard_buffer . . . . .	17
as.raw.shard_buffer . . . . .	18
as.vector.shard_buffer . . . . .	19
as_shared . . . . .	19
as_tibble . . . . .	20
as_tibble.shard_dataset . . . . .	21
as_tibble.shard_row_groups . . . . .	21
as_tibble.shard_table_buffer . . . . .	22
as_tibble.shard_table_handle . . . . .	22
attr<-.shard_shared_vector . . . . .	23
attributes<-.shard_shared_vector . . . . .	23
available_backings . . . . .	24
buffer . . . . .	24
buffer_advise . . . . .	25
buffer_close . . . . .	26
buffer_diagnostics . . . . .	27
buffer_info . . . . .	27
buffer_open . . . . .	28
buffer_path . . . . .	28
close.shard_shared . . . . .	29

collect . . . . .	30
collect.shard_dataset . . . . .	30
collect.shard_row_groups . . . . .	31
collect.shard_table_handle . . . . .	31
coltypes . . . . .	32
copy_report . . . . .	32
cow_report . . . . .	33
diagnostics . . . . .	34
dim.shard_buffer . . . . .	34
dim<- .shard_shared_vector . . . . .	35
dimnames<- .shard_shared_vector . . . . .	35
dispatch . . . . .	36
dispatch_chunks . . . . .	36
ergonomics . . . . .	37
factor_col . . . . .	37
fetch . . . . .	38
idx_range . . . . .	39
in_arena . . . . .	39
is_shared . . . . .	40
is_shared_vector . . . . .	40
is_view . . . . .	41
is_windows . . . . .	42
iterate_row_groups . . . . .	42
length.shard_buffer . . . . .	43
length.shard_descriptor . . . . .	43
length.shard_descriptor_lazy . . . . .	44
list_kernels . . . . .	45
materialize . . . . .	45
materialize.shard_view_block . . . . .	46
materialize.shard_view_gather . . . . .	46
mem_report . . . . .	47
names<- .shard_shared_vector . . . . .	48
pin_workers . . . . .	48
pool . . . . .	49
pool_create . . . . .	49
pool_dispatch . . . . .	50
pool_get . . . . .	51
pool_health_check . . . . .	51
pool_lapply . . . . .	52
pool_sapply . . . . .	53
pool_status . . . . .	53
pool_stop . . . . .	54
print.arena_result . . . . .	55
print.shard_apply_policy . . . . .	55
print.shard_buffer . . . . .	56
print.shard_deep_shared . . . . .	56
print.shard_descriptor . . . . .	57
print.shard_descriptor_lazy . . . . .	58

print.shard_dispatch_result . . . . .	58
print.shard_health_report . . . . .	59
print.shard_idx_range . . . . .	60
print.shard_pool . . . . .	60
print.shard_reduce_result . . . . .	61
print.shard_report . . . . .	62
print.shard_result . . . . .	62
print.shard_segment . . . . .	63
print.shard_shared . . . . .	64
print.shard_shared_vector . . . . .	64
print.shard_tiles . . . . .	65
print.shard_view_block . . . . .	65
print.shard_view_gather . . . . .	66
print.shard_worker . . . . .	66
queue . . . . .	67
recommendations . . . . .	67
register_kernel . . . . .	68
report . . . . .	69
results . . . . .	70
row_layout . . . . .	70
rss . . . . .	71
schema . . . . .	71
scratch_diagnostics . . . . .	72
scratch_matrix . . . . .	72
scratch_pool_config . . . . .	73
segment . . . . .	73
segment_advise . . . . .	74
segment_close . . . . .	74
segment_create . . . . .	75
segment_info . . . . .	76
segment_open . . . . .	76
segment_path . . . . .	77
segment_protect . . . . .	77
segment_read . . . . .	78
segment_report . . . . .	79
segment_size . . . . .	79
segment_write . . . . .	80
set_affinity . . . . .	81
shard_apply_matrix . . . . .	81
shard_apply_policy . . . . .	82
shard_crossprod . . . . .	83
shard_get_adapter . . . . .	84
shard_lapply_shared . . . . .	85
shard_list_adapters . . . . .	86
shard_map . . . . .	86
shard_reduce . . . . .	89
shard_register_adapter . . . . .	91
shard_share_hook . . . . .	92

shard\_unregister\_adapter . . . . . 93

shards . . . . . 94

shards\_list . . . . . 95

share . . . . . 96

share\_open . . . . . 98

shared\_advise . . . . . 98

shared\_diagnostics . . . . . 99

shared\_info . . . . . 100

shared\_reset\_diagnostics . . . . . 100

shared\_segment . . . . . 101

shared\_vector . . . . . 102

shared\_view . . . . . 103

stream\_count . . . . . 103

stream\_filter . . . . . 104

stream\_group\_count . . . . . 105

stream\_group\_sum . . . . . 105

stream\_map . . . . . 106

stream\_reduce . . . . . 107

stream\_sum . . . . . 108

stream\_top\_k . . . . . 108

succeeded . . . . . 109

table\_buffer . . . . . 110

table\_diagnostics . . . . . 110

table\_finalize . . . . . 111

table\_finalize.shard\_table\_buffer . . . . . 112

table\_finalize.shard\_table\_sink . . . . . 112

table\_sink . . . . . 113

table\_write . . . . . 114

table\_write.shard\_table\_buffer . . . . . 115

table\_write.shard\_table\_sink . . . . . 115

task\_report . . . . . 116

utils . . . . . 117

view . . . . . 117

view\_block . . . . . 118

view\_diagnostics . . . . . 118

view\_gather . . . . . 119

view\_info . . . . . 119

views . . . . . 120

worker . . . . . 120

---

[.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\_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.

---

```
[<-.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.

---

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**

<code>expr</code>	An expression to evaluate within the arena scope.
<code>strict</code>	Logical. If TRUE, enables strict mode which: <ul style="list-style-type: none"> <li>• Warns if large objects (&gt; 1MB by default) escape the scope</li> <li>• Triggers garbage collection after scope exit</li> <li>• Tracks memory growth for diagnostics</li> </ul> Default is FALSE for compatibility and performance.
<code>escape_threshold</code>	Numeric. Size in bytes above which returned objects trigger a warning in strict mode. Default is 1MB (1048576 bytes). Only used when <code>strict = TRUE</code> .
<code>gc_after</code>	Logical. If TRUE, triggers garbage collection after the arena scope exits. Default is TRUE in strict mode, FALSE otherwise.
<code>diagnostics</code>	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**

<code>x</code>	A shard_buffer object.
<code>...</code>	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**

<code>x</code>	A shard_buffer object (must be 2-dimensional).
<code>...</code>	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 <a href="#">as.vector</a> .

**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**

<code>x</code>	A <code>shard_dataset</code> handle.
<code>max_bytes</code>	Accepted for API consistency.
<code>...</code>	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**

<code>x</code>	A <code>shard_row_groups</code> handle.
<code>max_bytes</code>	Accepted for API consistency; currently unused for row-groups.
<code>...</code>	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

<code>x</code>	A <code>shard_table_handle</code> or <code>shard_table_buffer</code> .
<code>max_bytes</code>	Warn if estimated payload exceeds this threshold.
<code>...</code>	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

<code>x</code>	A <code>shard_table_handle</code> .
<code>max_bytes</code>	Warn if estimated payload exceeds this threshold.
<code>...</code>	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	<i>Get available shared memory backing types</i>
--------------------	--

---

**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	<i>Shared Memory Buffers</i>
--------	------------------------------

---

**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	<i>Close a Buffer</i>
--------------	-----------------------

---

**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	<i>Buffer Diagnostics</i>
--------------------	---------------------------

---

**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	<i>Get Buffer Info</i>
-------------	------------------------

---

**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	<i>Open an Existing Buffer</i>
-------------	--------------------------------

---

**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	<i>Get Buffer Path</i>
-------------	------------------------

---

**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	<i>Collect a shard table into memory</i>
---------	--

---

**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	<i>Collect a dataset handle into memory</i>
-----------------------	---

---

**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$idix), workers = 2)
pool_stop()
cow_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$idix), 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	<i>Task Dispatch Engine</i>
----------	-----------------------------

---

**Description**

Orchestrates chunk dispatch with worker supervision and failure handling.

---

dispatch_chunks	<i>Dispatch Chunks to Worker Pool</i>
-----------------	---------------------------------------

---

**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:

	<ul style="list-style-type: none"> <li>• <code>max_huge_concurrency</code>: cap concurrent chunks with <code>footprint_class=="huge"</code>.</li> </ul>
<code>on_result</code>	Optional callback (advanced). If provided, called on the master process as <code>on_result(tag, value, worker_id)</code> for each successful chunk completion. Used by <code>shard_reduce()</code> to stream reductions.
<code>store_results</code>	Logical (advanced). If <code>FALSE</code> , successful chunk values are not retained in the returned results list (streaming use cases).
<code>retain_chunks</code>	Logical (advanced). If <code>FALSE</code> , 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	<i>Fetch Data from a Shared Object</i>
-------	--

---

### 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	<i>Contiguous index range</i>
-----------	-------------------------------

---

**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	<i>Check if Currently Inside an Arena</i>
----------	---

---

**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	<i>Check if Object is Shared</i>
-----------	----------------------------------

---

**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	<i>Check if an object is a shared vector</i>
------------------	--

---

**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	<i>Check if running on Windows</i>
------------	------------------------------------

---

**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	<i>Iterate row groups</i>
--------------------	---------------------------

---

**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	<i>List registered kernels</i>
--------------	--------------------------------

---

**Description**

List registered kernels

**Usage**

```
list_kernels()
```

**Value**

A character vector of registered kernel names.

**Examples**

```
list_kernels()
```

---

materialize	<i>Materialize Shared Object</i>
-------------	----------------------------------

---

**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	<i>Memory Usage Report</i>
------------	----------------------------

---

**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	<i>Worker Pool Management</i>
------	-------------------------------

---

**Description**

Spawn and supervise persistent R worker processes with RSS monitoring.

---

pool_create	<i>Create a Worker Pool</i>
-------------	-----------------------------

---

**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 = .default_workers(),
  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. Defaults to <code>detectCores() - 1</code> , capped at 2 during R CMD check.
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	<i>Dispatch Task to Worker</i>
---------------	--------------------------------

---

**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	<i>Get the Current Worker Pool</i>
----------	------------------------------------

---

**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	<i>Check Pool Health</i>
-------------------	--------------------------

---

**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 <code>shard_pool</code> object. If <code>NULL</code> , 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	<i>Parallel sapply with Supervision</i>
-------------	---

---

**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	<i>Get Pool Status</i>
-------------	------------------------

---

**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**

<code>x</code>	A <code>shard_buffer</code> object.
<code>...</code>	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**

<code>x</code>	A <code>shard_deep_shared</code> object.
<code>...</code>	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	<i>Register a shard kernel</i>
-----------------	--------------------------------

---

### 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	<i>Generate Shard Runtime Report</i>
--------	--------------------------------------

---

### 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: <ul style="list-style-type: none"> <li>• "summary": High-level overview (default)</li> <li>• "workers": Include per-worker details</li> <li>• "tasks": Include task execution history</li> <li>• "segments": Include shared memory segment details</li> </ul>
result	Optional. A <code>shard_result</code> object from <code>shard_map</code> 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$idix), workers = 2)
pool_stop()
report(result = res)
```

---

results	<i>Extract Results from shard_map</i>
---------	---------------------------------------

---

**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	<i>Row layout for fixed-row table outputs</i>
------------	---

---

**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	<i>Advise OS about expected access pattern for a segment</i>
----------------	--

---

**Description**

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

**Usage**

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

**Arguments**

seg	A <code>shard_segment</code> .
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	<i>Close a shared memory segment</i>
---------------	--------------------------------------

---

**Description**

Close a shared memory segment

**Usage**

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

**Arguments**

x	A <code>shard_segment</code> 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	<i>Create a new shared memory segment</i>
----------------	---

---

**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	<i>Get segment information</i>
--------------	--------------------------------

---

**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	<i>Open an existing shared memory segment</i>
--------------	---

---

**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	<i>Get the path or name of a segment</i>
--------------	--

---

**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	<i>Make a segment read-only</i>
-----------------	---------------------------------

---

**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	<i>Read raw data from a segment</i>
--------------	-------------------------------------

---

**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	<i>Shared Memory Segment Report</i>
----------------	-------------------------------------

---

**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	<i>Get the size of a segment</i>
--------------	----------------------------------

---

**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	<i>Write data to a segment</i>
---------------	--------------------------------

---

**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	<i>Set CPU affinity for the current process</i>
--------------	---

---

**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()
```

---

shard_apply_matrix	<i>Apply a Function Over Matrix Columns with Shared Inputs</i>
--------------------	--

---

**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**

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

---

**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**

<code>X, Y</code>	Double matrices with the same number of rows.
<code>workers</code>	Number of worker processes.
<code>block_x, block_y</code>	Tile sizes over <code>ncol(X)</code> and <code>ncol(Y)</code> . Use "auto" (default) to autotune on the current machine.
<code>backing</code>	Backing for shared inputs and output buffer ("mmap" or "shm").
<code>materialize</code>	Whether to return the result as a standard R matrix: "never" (return buffer handle), "always", or "auto" (materialize if estimated output size is below <code>materialize_max_bytes</code> ).
<code>materialize_max_bytes</code>	Threshold for "auto" materialization.
<code>diagnostics</code>	Whether to collect <code>shard_map</code> 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 ( $p \times v$ )
- `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(e1, ...)`.

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 <code>shard_descriptor</code> from <code>shards()</code> , or an integer <code>N</code> 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 <code>kernel=</code> instead of providing <code>fun=</code> .
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 <code>buffer()</code> ). Workers write results directly to these buffers.
kernel	Optional. Name of a registered kernel (see <code>list_kernels()</code> ). If provided, <code>fun</code> must be <code>NULL</code> .
scheduler_policy	Optional list of scheduling hints (advanced). Currently: <ul style="list-style-type: none"> <li><code>max_huge_concurrency</code>: cap concurrent chunks whose kernel footprint is classified as "huge" (see <code>register_kernel()</code>).</li> </ul>
autotune	Optional. Online autotuning for scalar- <code>N</code> sharding (advanced). When <code>shards</code> is an integer <code>N</code> , <code>shard_map</code> can adjust shard block sizes over time based on observed wall time and worker RSS. Accepted values:

	<ul style="list-style-type: none"> <li>• NULL (default): enable online autotuning for <code>shard_map(N, ...)</code>, off for precomputed shard descriptors.</li> <li>• TRUE / "online": force online autotuning (only applies when shards is an integer N).</li> <li>• FALSE / "none": disable autotuning.</li> <li>• a list: <code>list(mode="online", max_rounds=..., probe_shards_per_worker=..., min_shard_time=...)</code></li> </ul>
<code>dispatch_mode</code>	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 <code>shard_map(N, ...)</code> with <code>chunk_size=1</code> and is intended for out-buffer/sink workflows (results are not gathered).
<code>dispatch_opts</code>	Optional list of dispatch-mode specific knobs (advanced). Currently: <ul style="list-style-type: none"> <li>• For <code>dispatch_mode="rpc_chunked"</code>: <ul style="list-style-type: none"> <li>– <code>auto_table</code>: logical. If TRUE, <code>shard_map</code> treats <code>data.frame/tibble</code> 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 <code>bind_rows()</code> on the master. Requires <code>out=</code> to be empty (use explicit <code>out=list(sink=table_sink(...))</code> otherwise).</li> <li>– <code>auto_table_materialize</code>: "never", "auto", or "always" (default "auto").</li> <li>– <code>auto_table_max_bytes</code>: numeric/integer. For "auto", materialize only if estimated output size <math>\leq</math> this threshold (default 256MB).</li> <li>– <code>auto_table_mode</code>: "row_groups" (default) or "partitioned".</li> <li>– <code>auto_table_path</code>: optional output directory (default <code>tempdir()</code>).</li> <li>– <code>auto_table_format</code>: "auto", "rds" (default), or "native".</li> <li>– <code>auto_table_schema</code>: optional <code>shard_schema</code> for validation/native encoding.</li> </ul> </li> <li>• For <code>dispatch_mode="shm_queue"</code>: <ul style="list-style-type: none"> <li>– <code>block_size</code>: integer. If provided, overrides the default heuristic for contiguous shard block sizing.</li> <li>– <code>queue_backing</code>: one of "mmap" or "shm" (default "mmap").</li> <li>– <code>error_log</code>: logical. If TRUE, workers write a bounded per-worker error log to disk to aid debugging failed tasks (default FALSE).</li> <li>– <code>error_log_max_lines</code>: integer. Maximum lines per worker in the error log (default 100).</li> </ul> </li> </ul>
<code>workers</code>	Integer. Number of worker processes. If NULL, uses existing pool or creates one with <code>detectCores() - 1</code> .
<code>chunk_size</code>	Integer. Shards to batch per worker dispatch (default 1). Higher values reduce RPC overhead but may hurt load balancing.
<code>profile</code>	Execution profile: "default", "memory" (aggressive recycling), or "speed" (minimal overhead). With <code>profile="speed"</code> , <code>shard_map</code> will automatically enable <code>dispatch_mode="shm_queue"</code> when possible for <code>shard_map(N, ...)</code> out-buffer workflows (scalar N, <code>chunk_size=1</code> ), unless <code>dispatch_mode</code> 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()
```

**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 <a href="#">shards()</a> , 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 <a href="#">shard_map()</a> ).
out	Named list of output buffers/sinks (same semantics as <a href="#">shard_map()</a> ).
workers	Number of worker processes.
chunk_size	Shards to batch per worker dispatch (default 1).
profile	Execution profile (same semantics as <a href="#">shard_map()</a> ).
mem_cap	Memory cap per worker (same semantics as <a href="#">shard_map()</a> ).
recycle	Worker recycling policy (same semantics as <a href="#">shard_map()</a> ).
cow	Copy-on-write policy for borrowed inputs (same semantics as <a href="#">shard_map()</a> ).
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**

<code>class</code>	A character string naming the class to register the adapter for.
<code>adapter</code>	A list containing: <ul style="list-style-type: none"> <li><b>class</b> Character string matching the <code>class</code> parameter.</li> <li><b>children</b> Function taking an object and returning a named list of child objects to traverse.</li> <li><b>replace</b> Function taking the original object and a named list of (potentially shared) children, returning a reconstructed object.</li> <li><b>path_prefix</b> Optional character string prefix for child paths in the sharing plan (default: class name).</li> </ul>

**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**

<code>x</code>	The object being traversed during deep sharing.
<code>ctx</code>	A context list containing: <ul style="list-style-type: none"> <li><b>path</b> Current node path string (e.g., "&lt;root&gt;\$data@cache")</li> <li><b>class</b> <code>class(x)</code> - the object's class vector</li> <li><b>mode</b> 'strict' or 'balanced' - sharing mode</li> <li><b>min_bytes</b> Minimum size threshold for sharing</li> <li><b>types</b> Character vector of enabled types for sharing</li> <li><b>deep</b> Logical, always TRUE when hook is called</li> </ul>

**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()
```

---

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**

<code>n</code>	Integer. Total number of items to shard.
<code>block_size</code>	Block size specification. Can be: <ul style="list-style-type: none"> <li>• "auto" (default): Autotune based on worker count</li> <li>• Integer: Explicit number of items per shard</li> <li>• Character: Human-readable like "1K", "10K"</li> </ul>
<code>workers</code>	Integer. Number of workers for autotuning (default: pool size or <code>detectCores - 1</code> ).
<code>strategy</code>	Sharding strategy: "contiguous" (default) or "strided".
<code>min_shards_per_worker</code>	Integer. Minimum shards per worker for load balancing (default 4).
<code>max_shards_per_worker</code>	Integer. Maximum shards per worker to limit overhead (default 64).
<code>scratch_bytes_per_item</code>	Numeric. Expected scratch memory per item for memory budgeting.
<code>scratch_budget</code>	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)
```

---

 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 <code>serialize()</code> .
backing	Backing type: "auto" (default), "mmap", or "shm". <ul style="list-style-type: none"> <li>• "auto": Let the system choose the best option.</li> <li>• "mmap": File-backed memory mapping. Most portable.</li> <li>• "shm": POSIX shared memory or Windows named mapping.</li> </ul>
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)
```

---

share_open	<i>Open an Existing Shared Object by Path</i>
------------	---

---

### 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)
```

---

shared_advise	<i>Advise access pattern for a shared input vector/matrix</i>
---------------	---

---

### 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	<i>Get the underlying segment from a shared vector</i>
----------------	--

---

**Description**

Get the underlying segment from a shared vector

**Usage**

```
shared_segment(x)
```

**Arguments**

x                    A shard ALTREP vector

**Value**

A shared\_segment S3 object wrapping the underlying segment

**Examples**

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

---

shared_vector	<i>Create a shared vector from a segment</i>
---------------	--

---

### 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	<i>Create a view (subset) of a shared vector</i>
-------------	--

---

**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]
```

---

stream_count	<i>Stream row count</i>
--------------	-------------------------

---

**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	<i>Stream group-wise count</i>
--------------------	--------------------------------

---

**Description**

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

**Usage**

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

**Arguments**

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

**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	<i>Stream group-wise sum</i>
------------------	------------------------------

---

**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**

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

**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	<i>Stream sum of a numeric column</i>
------------	---------------------------------------

---

### 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

<code>x</code>	A <code>shard_row_groups</code> or <code>shard_dataset</code> handle.
<code>col</code>	Column name to sum.
<code>na_rm</code>	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	<i>Stream top-k rows by a numeric column</i>
--------------	--

---

### 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	<i>Allocate a fixed-row table buffer</i>
--------------	--

---

**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	<i>Table Diagnostics</i>
-------------------	--------------------------

---

**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	<i>Finalize a table buffer or sink</i>
----------------	--

---

**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**

<code>target</code>	A <code>shard_table_buffer</code> or <code>shard_table_sink</code> .
<code>materialize</code>	"never", "auto", or "always".
<code>max_bytes</code>	For "auto", materialize only if estimated bytes $\leq$ <code>max_bytes</code> .
<code>...</code>	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	<i>Create a table sink for row-group or partitioned outputs</i>
------------	---

---

**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")
```

---

<code>table_write</code>	<i>Write tabular results into a table buffer or sink</i>
--------------------------	--

---

**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**

<code>target</code>	A <code>shard_table_buffer</code> or <code>shard_table_sink</code> .
<code>rows_or_shard_id</code>	For buffers: row selector ( <code>idx_range</code> or integer vector). For sinks: shard id (integer).
<code>data</code>	A <code>data.frame</code> or named list matching the schema columns.
<code>...</code>	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	<i>Utility Functions</i>
-------	--------------------------

---

**Description**

Internal utilities for shard package.

---

view	<i>Create a view over a shared matrix</i>
------	---

---

**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 <code>idx_range()</code> .
cols	Column selector. NULL (all cols) or <code>idx_range()</code> .
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))
```

---

view_block	<i>Create a contiguous block view</i>
------------	---------------------------------------

---

**Description**

Create a contiguous block view

**Usage**

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

**Arguments**

x	A shared (share()) 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	<i>View diagnostics</i>
------------------	-------------------------

---

**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	<i>Create a gather (indexed) view over a shared matrix</i>
-------------	--

---

**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	<i>Introspection for a view</i>
-----------	---------------------------------

---

**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)
```

---

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.

---

worker

*Individual Worker Control*

---

**Description**

Spawn, monitor, and control individual R worker processes.

# Index

[.shard\_buffer, 6  
[.shard\_descriptor, 6  
[.shard\_descriptor\_lazy, 7  
[<-.shard\_buffer, 9  
[<-.shard\_shared\_vector, 10  
[[.shard\_descriptor, 8  
[[.shard\_descriptor\_lazy, 8  
[[<-.shard\_shared\_vector, 9

adapter, 11  
affinity, 11  
affinity\_supported, 12  
altrep, 12  
arena, 13  
arena\_depth, 14  
as.array.shard\_buffer, 15  
as.double.shard\_buffer, 15  
as.integer.shard\_buffer, 16  
as.logical.shard\_buffer, 17  
as.matrix.shard\_buffer, 17  
as.raw.shard\_buffer, 18  
as.vector, 19  
as.vector.shard\_buffer, 19  
as\_shared, 19  
as\_tibble, 20  
as\_tibble.shard\_dataset, 21  
as\_tibble.shard\_row\_groups, 21  
as\_tibble.shard\_table\_buffer, 22  
as\_tibble.shard\_table\_handle, 22  
attr<-.shard\_shared\_vector, 23  
attributes<-.shard\_shared\_vector, 23  
available\_backings, 24

bool (coltypes), 32  
buffer, 24  
buffer\_advise, 25  
buffer\_close, 26  
buffer\_diagnostics, 27  
buffer\_info, 27  
buffer\_open, 28

buffer\_path, 28

close.shard\_deep\_shared  
    (close.shard\_shared), 29  
close.shard\_shared, 29  
close.shard\_shared\_vector  
    (close.shard\_shared), 29  
collect, 30  
collect.shard\_dataset, 30  
collect.shard\_row\_groups, 31  
collect.shard\_table\_handle, 31  
coltypes, 32  
copy\_report, 32  
cow\_report, 33

diagnostics, 34  
dim.shard\_buffer, 34  
dim<-.shard\_shared\_vector, 35  
dimnames<-.shard\_shared\_vector, 35  
dispatch, 36  
dispatch\_chunks, 36

ergonomics, 37

factor\_col, 37  
fetch, 38  
float64 (coltypes), 32

idx\_range, 39  
in\_arena, 39  
int32 (coltypes), 32  
is\_block\_view (is\_view), 41  
is\_shared, 40  
is\_shared\_vector, 40  
is\_view, 41  
is\_windows, 42  
iterate\_row\_groups, 42

length.shard\_buffer, 43  
length.shard\_descriptor, 43  
length.shard\_descriptor\_lazy, 44

[list\\_kernels](#), 45  
[list\\_kernels\(\)](#), 87  
  
[materialize](#), 45  
[materialize.shard\\_view\\_block](#), 46  
[materialize.shard\\_view\\_gather](#), 46  
[mem\\_report](#), 47  
  
[names<- .shard\\_shared\\_vector](#), 48  
  
[pin\\_workers](#), 48  
[pool](#), 49  
[pool\\_create](#), 49, 97  
[pool\\_dispatch](#), 50  
[pool\\_get](#), 51  
[pool\\_health\\_check](#), 51  
[pool\\_lapply](#), 52  
[pool\\_sapply](#), 53  
[pool\\_status](#), 53  
[pool\\_stop](#), 54  
[print.arena\\_result](#), 55  
[print.shard\\_apply\\_policy](#), 55  
[print.shard\\_buffer](#), 56  
[print.shard\\_deep\\_shared](#), 56  
[print.shard\\_descriptor](#), 57  
[print.shard\\_descriptor\\_lazy](#), 58  
[print.shard\\_dispatch\\_result](#), 58  
[print.shard\\_health\\_report](#), 59  
[print.shard\\_idx\\_range](#), 60  
[print.shard\\_pool](#), 60  
[print.shard\\_reduce\\_result](#), 61  
[print.shard\\_report](#), 62  
[print.shard\\_result](#), 62  
[print.shard\\_segment](#), 63  
[print.shard\\_shared](#), 64  
[print.shard\\_shared\\_vector](#), 64  
[print.shard\\_tiles](#), 65  
[print.shard\\_view\\_block](#), 65  
[print.shard\\_view\\_gather](#), 66  
[print.shard\\_worker](#), 66  
  
[queue](#), 67  
  
[raw\\_col \(coltypes\)](#), 32  
[recommendations](#), 67  
[register\\_kernel](#), 68  
[register\\_kernel\(\)](#), 87  
[report](#), 69  
[results](#), 70  
  
[row\\_layout](#), 70  
[rss](#), 71  
  
[schema](#), 71  
[scratch\\_diagnostics](#), 72  
[scratch\\_matrix](#), 72  
[scratch\\_pool\\_config](#), 73  
[segment](#), 73  
[segment\\_advise](#), 74  
[segment\\_advise\(\)](#), 26, 99  
[segment\\_close](#), 74  
[segment\\_create](#), 25, 75, 97  
[segment\\_info](#), 76  
[segment\\_open](#), 76  
[segment\\_path](#), 77  
[segment\\_protect](#), 77  
[segment\\_read](#), 78  
[segment\\_report](#), 79  
[segment\\_size](#), 79  
[segment\\_write](#), 80  
[set\\_affinity](#), 81  
[shard\\_apply\\_matrix](#), 81  
[shard\\_apply\\_policy](#), 82  
[shard\\_apply\\_policy\(\)](#), 82, 85  
[shard\\_crossprod](#), 83  
[shard\\_get\\_adapter](#), 84  
[shard\\_lapply\\_shared](#), 85  
[shard\\_list\\_adapters](#), 86  
[shard\\_map](#), 14, 25, 69, 86, 116  
[shard\\_map\(\)](#), 37, 67, 82, 83, 85, 90  
[shard\\_reduce](#), 89  
[shard\\_reduce\(\)](#), 37  
[shard\\_register\\_adapter](#), 91  
[shard\\_share\\_hook](#), 92  
[shard\\_unregister\\_adapter](#), 93  
[shards](#), 94  
[shards\(\)](#), 87, 90  
[shards\\_list](#), 95  
[share](#), 11, 14, 25, 96  
[share\(\)](#), 82, 99  
[share\\_open](#), 98  
[shared\\_advise](#), 98  
[shared\\_diagnostics](#), 99  
[shared\\_info](#), 100  
[shared\\_reset\\_diagnostics](#), 100  
[shared\\_segment](#), 101  
[shared\\_vector](#), 102  
[shared\\_view](#), 103  
[stream\\_count](#), 103

- stream\_filter, [104](#)
- stream\_group\_count, [105](#)
- stream\_group\_sum, [105](#)
- stream\_map, [106](#)
- stream\_reduce, [107](#)
- stream\_sum, [108](#)
- stream\_top\_k, [108](#)
- string\_col (coltypes), [32](#)
- succeeded, [109](#)
  
- table\_buffer, [110](#)
- table\_diagnostics, [110](#)
- table\_finalize, [111](#)
- table\_finalize.shard\_table\_buffer, [112](#)
- table\_finalize.shard\_table\_sink, [112](#)
- table\_sink, [113](#)
- table\_write, [114](#)
- table\_write.shard\_table\_buffer, [115](#)
- table\_write.shard\_table\_sink, [115](#)
- task\_report, [116](#)
  
- utils, [117](#)
  
- view, [117](#)
- view\_block, [118](#)
- view\_diagnostics, [118](#)
- view\_gather, [119](#)
- view\_info, [119](#)
- views, [120](#)
  
- worker, [120](#)