

Package: storrtiledb (via r-universe)

October 13, 2026

Title TileDB Storage Driver for Storr Interface

Version 0.1.17

Date 2026-09-11

Maintainer Constantinos Giachalis

<26255858+cgiachalis@users.noreply.github.com>

Description Provides a TileDB driver for 'storr' interface. 'storr' is a content addressable key value store with an optional caching layer; this package contributes a new driver using TileDB Embedded storage engine as a backend. In addition, 'storrtiledb' introduces a specialized TileDBStorr class that replicates 'storr' API in order to improve speed and efficiency while enhancing 'storr' with additional features, e.g., per-key notes and Time-To-Live expiration timestamps as well as support for asynchronous and in parallel writes. Optionally, it offers alternative and faster serialisation formats through `qs2` package, suitable for saving large 'R' objects.

License MIT + file LICENSE

URL <https://github.com/cgiachalis/storrtiledb>,
<https://cgiachalis.github.io/storrtiledb/>

BugReports <https://github.com/cgiachalis/storrtiledb/issues>

Depends R (>= 4.4.0)

Imports arrow, cli, data.table, digest, mirai (>= 2.7.0), R6 (>= 2.4.0), R6.tiledb (>= 0.3.0), storrtiledb (>= 0.33.0)

Suggests litedown, qs2 (>= 0.3.1), testthat (>= 3.0.0), withr

VignetteBuilder litedown

Remotes cgiachalis/R6.tiledb

Config/roxygen2/version 8.1.0

Config/testthat/edition 3

Config/testthat/parallel true

Config/testthat/start-first StorrFragments,auto*

Encoding UTF-8
Language en-GB
Roxygen list(markdown = TRUE, old_usage = TRUE)
Config/pak/sysreqs cmake git make libuv1-dev libzstd-dev libssl-dev
Repository https://cgiachalis.r-universe.dev
Date/Publication 2026-09-27 07:33:25 UTC
RemoteUrl https://github.com/cgiachalis/storr.tiledb
RemoteRef HEAD
RemoteSha 3b8a55ad24a6072c16604154d014084843b9a610

Contents

driver_schemas	2
driver_tiledb	3
storr_consolidate	7
storr_copy	8
storr_fragments	9
storr_move	10
storr_rename	11
storr_tiledb	12
storr_timetravel	18
storr_vacuum	21
Index	23

driver_schemas	<i>TileDB Driver Schemas</i>
----------------	------------------------------

Description

Tune driver performance and storage characteristics: compression algorithms, compression levels, tile capacity, cell order, and tile order settings. This is useful for creating storage drivers for use cases that need different trade-offs (speed vs. compression, memory vs. disk).

Usage

driver_schemas(uri = NULL, ctx = NULL, none_filter = FALSE)

Arguments

- | | |
|-------------|--|
| uri | Optional URI path to TileDB driver. If not given, the default template array schemas will be used. |
| ctx | Optional tiledb_ctx object. |
| none_filter | TRUE for no filters, FALSE for default filters. Applied on default schemas and not on schemas extracted from a URI path. |

Details

Use `driver_schemas()` to:

- Create schemas with optional filters
- Dynamically apply compression to individual attributes
- Persist those customizations into the underlying TileDB arrays

Value

An object of class `TileDBDriverSchemas`.

Examples

```
ctx <- new_context()

# Create schemas without compression filters
sto_schemas <- driver_schemas(ctx = ctx, none_filter = TRUE)

# Set up a ZSTD filter with high compression
flt <- tiledb::tiledb_filter("ZSTD", ctx = ctx)
flt <- tiledb::tiledb_filter_set_option(flt, "COMPRESSION_LEVEL", 22)
fl_list <- tiledb::tiledb_filter_list(flt, ctx = ctx)

## Set filter list to 'value' attribute (CAS storage data)
sto_schemas$SchemaData$attr_value <- fl_list

# Check 'data' schema; notice 'attr_value' has a new filter list
sto_schemas$SchemaData$schema()
```

driver_tiledb

TileDB Storr Driver

Description

Storr driver using TileDB storage engine.

Usage

```
driver_tiledb(uri, context = NULL, init = FALSE, ...)

driver_tiledb_create(uri, hash_algorithm = NULL, serial_format = "rds",
  compression_level = -7, driver_schemas = NULL, context = NULL)
```

Arguments

uri	The URI path of storr.
context	Optional tiledb_ctx object.
init	Should the driver be created if not exist? Default is FALSE.
...	Other arguments passed to driver's create method when init = TRUE. Valid arguments: hash_algorithm, serial_format, compression_level, keep_open and driver_schemas. If driver_schemas argument is given, the compression_level argument will be ignored.
hash_algorithm	Select a hash algorithm supported by digest : 'md5', 'sha1', 'crc32', 'sha256', 'sha512', 'xxhash32', 'xxhash64', 'murmur32', 'spookyhash', 'blake3', 'crc32c', 'xxh3_64', 'xxh3_128'. If not given, the default is 'md5'.
serial_format	Select serialization format: "rds" (default), "qs2" or "qdata". For the latter two, 'qs2' package is required.
compression_level	Set an integer value for ZSTD compression level. If NULL value is given, no compression filters will be applied to data tiles.
driver_schemas	An object of class TileDBDriverSchemas with user defined storage schemas; See driver_schemas() . If given, the compression_level argument will be ignored.

Value

- *driver_tiledb()* returns a [TileDBDriver](#) object.
- *driver_tiledb_create()* returns logical TRUE invisibly, for successful driver creation.

Details

driver_tiledb() returns the TileDB driver given a URI path. You can create a new driver using *init* argument or with *driver_tiledb_create()*.

Note that [storr_tiledb\(\)](#) abstracts the driver creation, so it's the preferred way unless you want to pass the driver onto [storr](#).

Class Methods Summary

For complete definitions, see **Methods** section in [TileDBDriver](#) and its parent class [CAS](#).

Active Fields

- hash_algorithm - Property for managing the hash algorithm (read/write)
- members_instantiated - Property indicating instantiation status (read-only)
- size - Get storr size (read-only)
- serial_format - Get serialization format (read-only)

Lifecycle

- open() - Opens the driver for read/write operations

- `create()` - Create a new driver object
- `close()` - Closes the driver
- `destroy()` - Destroy/delete the driver

Hash Management

- `get_hash()` and `mget_hash()` - Retrieve hash values for a given key and namespace
- `set_hash()` and `mset_hash()` - Set hash values with metadata like expiry date-times and notes
- `exists_hash()` - Verify the existence of specific keys, namespaces
- `del_hash()` - Remove key-namespace pairs
- `delete_namespaces()` - Clear namespaces or delete specified ones
- `list_namespaces()`, `list_keys()` - Retrieve all namespaces or keys for a given namespace

Object Management

- `get_object()` and `mget_object()` - Fetch serialized R objects using hash values
- `set_object()` and `mset_object()` - Store serialized R objects
- `exists_object()` - Verify the existence of specific objects
- `del_object()` - Remove serialized objects.
- `delete_unused_hashes()` - Remove hashes that are not in active use
- `list_hashes()` - Retrieve all hashes
- `list_unused_hashes()` - Identify unused hashes

Key-Namespace Metadata

- `get_keymeta()` - Retrieve metadata such as expiry times and notes for single key-namespace pairs
- `mget_keymeta()` - Retrieve metadata such as expiry times and notes for multiple key-namespace pairs
- `get_keymeta_unit()` and `mget_keymeta_unit()` - Fetch single metadata such as expiry times or notes
- `list_notes()` - Retrieve all notes for a given namespace

Expiration Management

- `keys_with_expiration` and `keys_without_expiration` - Retrieve the key namespace pairs with or without expiration timestamps
- `expired_keys()` and `unexpired_keys()` - Retrieve the (un)expired key namespace pairs
- `delete_expired_keys()` - Delete all expired keys or for specific namespaces
- `num_expired_keys()` and `num_unexpired_keys()` - Get the number of (un)expired keys or for specific namespaces
- `has_expired_keys()` and `has_unexpired_keys()` - Verify the existence of (un)expired keys or for specific namespaces
- `is_key_expired()` - Check if a key is expired

Notes Management

- `keys_with_notes()` - Retrieve the key namespace pairs with notes

Export Utilities

- `export_tdb()` - Export objects to another TileDB storr

Directory Methods

- `dir_tree()` - Displays directory structure
- `dump()` - Outputs a string representation of storr structure

Data Model

The underlying structure is similar to `driver_dbi()`; a content-addressed database that consists of two (array) tables.

Specifically, the driver storage is a TileDB Group with two member Arrays which are stored relative to Group's URI path. The next sub-sections describe the group structure and array data models.

Group

Members

1. `tbl_keys` (array): maps key-namespace pairs to hashes (and to expiry and/or notes, optional)
2. `tbl_data` (array): maps hashes to values (serialised R objects)

Metadata

- `hash_algo`: the name of hash algorithm
- `type`: group identifier, "storr"
- `serial_format`: the name of serialization format

Arrays

tbl_keys - A 2D sparse array that maps key-namespace pairs to hashes and key-metadata.

- Dimensions: namespace (*ASCII*) and key (*ASCII*)
- Attributes: hash (*ASCII*), expires_at (*DATETIME_MS*) and notes (*UTF8*)

tbl_data - A 1D sparse array that maps hashes to object values.

- Dimensions: hash (*ASCII*)
- Attributes: value (*ASCII*)

TileDB datatypes in parentheses.

See Also

`storr_tiledb()`

Examples

```
# create driver
uri <- tempfile()
dr <- driver_tiledb(uri, init = TRUE)

# members
dr$names()

# metadata
dr$get_metadata()
```

storr Consolidate	<i>Consolidate Storr Fragments</i>
-------------------	------------------------------------

Description

Perform fragment consolidation for better query performance.

Usage

```
storr Consolidate(uri, what = "all", vacuum = FALSE, async = FALSE,
  context = NULL)
```

Arguments

uri	The URI path of storr.
what	Which array should be consolidated? Defaults to "all" arrays.
vacuum	Should the old fragments (consolidated) be deleted? Default is FALSE.
async	Should it consolidate asynchronously? Default is FALSE.
context	Optional tiledb_ctx object.

Details

The function supports selective consolidation ("keys"/"data") or full ("all"). The process can be run synchronously or asynchronously and has an optional argument to delete old fragments afterwards (vacuum).

Value

When `async = FALSE`, it returns `TRUE` for successful consolidation. For `async = TRUE`, it returns a [mirai::mirai\(\)](#) object immediately; once resolved, it returns `TRUE` indicating consolidation success.

See Also[storr_vacuum\(\)](#)**Examples**

```
# URI path
uri <- tempfile()
sto <- storr_tiledb(uri, init = TRUE)

storr_consolidate(uri)
```

storr_copy*Copy Storr to another URI*

Description

Copy Storr to another URI

Usage

```
storr_copy(uri, to_uri, context = NULL)
```

Arguments

uri	The URI path of storr.
to_uri	Destination URI path to copy the storr to.
context	Optional tiledb_ctx object.

Value

The new uri path, invisibly.

See Also

Other storr-utilities: [storr_move\(\)](#), [storr_rename\(\)](#)

Examples

```
uri <- tempfile()
sto <- storr_tiledb(uri, init = TRUE)

# set key-values
sto$set("a", 1)
sto$set("b", 2)

# Copy storr to new URI
to_uri <- tempfile()
storr_copy(uri, to_uri)
```

```
sto2 <- storr_tiledb(to_uri)

sto2$list()
```

storr_fragments	<i>Storr Fragments</i>
-----------------	------------------------

Description

Functional interface that initialises a [StorrFragments](#) instance to work with TileDB Fragments.

Usage

```
storr_fragments(uri, context = NULL)
```

Arguments

uri	The URI path of storr.
context	Optional tiledb_ctx object.

Details

The class includes fragment consolidation and vacuuming methods but also provides access to [TileDBFragments](#) instances for keys and data arrays where you can further inspect and manage the fragments for the specify arrays.

Value

An object of class StorrFragments, R6.

Class Methods Summary

For complete definitions see the **Methods** section in [StorrFragments](#).

Active Fields

- **fob_keys** - [TileDBFragments](#) instance for the keys array.
- **fob_data** - [TileDBFragments](#) instance for the data array.
- **size** - VFS directory size for the TileDB store.

Fragment Management

- **consolidate()** - Consolidate fragments (options: all, keys, data); supports vacuum and sync/async modes.
- **vacuum()** - Remove old consolidated fragments (options: all, keys, data); supports sync/async.
- **reload_finno()** - Refresh fragment-info caches for both arrays.

Inspection

- **frag_num()** - Total number of fragments (keys + data).
- **to_vacuum_num()** - Number of fragments pending vacuum.

See Also

[storr Consolidate\(\)](#) and [storr Vacuum\(\)](#)

Examples

```
# URI path
uri <- tempfile()
sto <- storr_tiledb(uri, init = TRUE)

fosto <- storr_fragments(uri)

fosto$frag_num()
```

storr_move

Move Storr to another URI

Description

Move Storr to another URI

Usage

```
storr_move(uri, newuri, context = NULL)
```

Arguments

uri	The URI path of storr.
newuri	Destination URI path to move the storr to.
context	Optional tiledb_ctx object.

Value

The new uri path, invisibly.

See Also

Other storr-utilities: [storr_copy\(\)](#), [storr_rename\(\)](#)

Examples

```

uri <- tempfile()
sto <- storr_tiledb(uri, init = TRUE)

# set key-values
sto$set("a", 1)
sto$set("b", 2)

# Move storr to new URI
to_uri <- tempfile()
storr_move(uri, newuri = to_uri)

sto2 <- storr_tiledb(to_uri)

sto2$list()

```

storr_rename

Rename Storr URI

Description

It renames the driver's basename, i.e., 'path/oldname' to 'path/newname'.

Usage

```
storr_rename(uri, newname, context = NULL)
```

Arguments

uri	The URI path of storr.
newname	Suffix to rename storr URI path.
context	Optional tiledb_ctx object.

Value

The new uri path, invisibly.

See Also

Other storr-utilities: [storr_copy\(\)](#), [storr_move\(\)](#)

Examples

```
uri <- tempfile()
sto <- storr_tiledb(uri, init = TRUE)

# set key-values
sto$set("a", 1)
sto$set("b", 2)

# Rename storr
newuri <- storr_rename(uri, newname = "new-storr")

sto2 <- storr_tiledb(newuri)

sto2$list()
```

storr_tiledb

A Storr using TileDB Engine

Description

Create a 'storr' using TileDB driver for storage.

Usage

```
storr_tiledb(uri, default_namespace = "objects", context = NULL,
  init = FALSE, serial_format = "rds", hash_algorithm = NULL,
  async = FALSE, ...)
```

Arguments

uri	The URI path of storr.
default_namespace	The default namespace: "objects".
context	Optional tiledb_ctx object.
init	Should the driver be created if not exist? Default is FALSE.
serial_format	Select serialization format: "rds" (default), "qs2" or "qdata". For the latter two, 'qs2' package is required.
hash_algorithm	Select a hash algorithm supported by digest : 'md5', 'sha1', 'crc32', 'sha256', 'sha512', 'xxhash32', 'xxhash64', 'murmur32', 'spookyhash', 'blake3', 'crc32c', 'xxh3_64', 'xxh3_128'. If not given, the default is 'md5'.
async	Should the mirai daemons be enabled for async functions? Default is FALSE. Each storr instance has its own independent set of daemons. See Details.
...	Other arguments passed to driver when init = TRUE. Valid arguments: <code>compression_level</code> and <code>driver_schemas</code> . If <code>driver_schemas</code> argument is given, the <code>compression_level</code> argument will be ignored. For more details, see driver_tiledb_create() .

Value

An object of class [TileDBStorr](#), R6.

Details

'[storr](#)' is a content addressed key-value store with an optional caching layer.

The `storr_tiledb` generates a [TileDBStorr](#) object with identical interface as `storr` that additionally supports metadata next to key-values (notes and expiration timestamps) as well as asynchronous writes using the [mirai](#) framework.

`storr_tiledb()` and `storr(driver_tiledb())` can not be used interchangeably if you use the extra features (i.e., expiration timestamps). The latter is the standard `storr` interface and the former produces a stand-alone R6 class that replicates the `storr` interface with additional features.

Another difference, but not visible to the user, is that the `storr_tiledb`'s cache layer uses hash tables via [hashtab\(\)](#) instead of environments.

Serialization:

R objects are saved to storage engine as string representation of the raw vector. The default serialization format ("rds") uses the [serialize\(\)](#) function. Optionally, to store large objects efficiently, the package supports "qs2" and "qdata" formats powered by '[qs2](#)' package (must be installed) using for string encoding/decoding the base91 format.

Cache option:

The in-memory caching layer is enabled by default and is controlled via the global option `storr.tiledb.cache`:

```
# Disable cache
options(storr.tiledb.cache = FALSE)
```

Buffer size:

The buffer allocation size is set to 3 MB per column when fetching data. Use [set_allocation_size_preference\(\)](#) to set a different limit.

Compression:

The storage compression filter is "ZSTD" and is applied to dimensions, attributes, coords and offsets. The compression level is configurable through `compression_level` argument with default level -7 that balances compression ratio and speed.

To create a driver without compression filters, set `compression_level = NULL`.

Note that a "RLE" filter is specifically used for validity bitmaps and is not configurable. For more flexibility, see next section [Schemas Configuration](#).

Schemas Configuration:

To support different use cases with respect to speed and compression, a user can create a custom driver schemas using the entry point functional wrapper [driver_schemas\(\)](#); this gives access to TileDB array schemas in the content-addressable storage (CAS) system that can be modified in order to tune TileDB's engine performance and storage characteristics: compression algorithms, compression levels, tile and cell order. For details, see [Examples](#) section.

Async Evaluation:

storr_tiledb uses [mirai](#) package to set keys asynchronously. Async is enabled either at initialisation or automatically when using one of the async methods.

Each 'storr' instantiation comes with its own independent set of daemons using a unique compute profile (namespace). To access the specific compute profile, i.e., to launch more daemons via `mirai::launch_local()` use the active field `$async_info` :

```
# Retrieve mirai's compute profile
sto$async_info["profile"]
```

By default, async process launches two daemons. This is configurable with `storr.tiledb.mirai.daemons` option:

```
# set mirai daemons
options(storr.tiledb.mirai.daemons = 1L)
```

In addition, to set a memory budget (in MB) for queued task payloads at the dispatcher, use `storr.tiledb.mirai.memory` option:

```
# set mirai memory budget
options(storr.tiledb.mirai.memory = 100) # 100MB
```

The daemons associated with the specific compute profile are reset to zero when the 'storr' object is deleted/garbage collected.

Key Expiration:

Keys with expiration time-stamps are not automatically cleared; they can be fetched post expiration datetime unless they are removed by using one of `$clear_expired_keys()` or `gc(clear_expired = TRUE)` method.

Alternatively, use `$is_expired_key()` before a getter method for more refined control.

Class Methods Summary

For complete definitions, see **Methods** section in [TileDBStorr](#).

Active Fields

- `async_info` - Get mirai daemon information (read-only)
- `size` - Get storr size (read-only)

Initialisation & Lifecycle

- `new()` - Initialise a `TileDBStorr` object with a `TileDB` driver, default namespace, and optional async support
- `destroy()` - Destroy/delete the storr and clean up the driver

Cache Management

- `flush_cache()` - Remove all items from both object and metadata hash tables

Single Key-Value Operations

- `set()` - Set a key-value pair with optional metadata (`expires_at`, `notes`)
- `get()` - Retrieve an object by key-namespace pair
- `update()` - Update a key-value pair and retain key-metadata
- `set_by_value()` - Set a key-value pair using the object's hash as the key
- `get_value()` - Retrieve an object given its hash
- `get_all()` - Retrieve an object and its metadata by key-namespace pair

Multiple Key-Value Operations

- `mset()` - Set multiple key-value pairs in batch
- `mget()` - Get multiple objects by key-namespace pairs
- `mupdate()` - Update multiple objects by key-namespace pairs and retain key-metadata
- `mset_by_value()` - Set multiple key-value pairs using their hashes as keys
- `mget_value()` - Get multiple objects by their hashes
- `mget_all()` - Retrieve multiple objects and its metadata by key-namespace pairs

Metadata Operations

- `update_keymeta()` - Update metadata (`expires_at`, `notes`) for a key
- `mupdate_keymeta()` - Update metadata for multiple keys
- `get_keymeta()` - Retrieve metadata for a key
- `mget_keymeta()` - Retrieve metadata for multiple keys
- `get_keymeta_expires_at()` - Retrieve expiration metadata for a key
- `get_keymeta_notes()` - Retrieve notes metadata for a key
- `mget_keymeta_expires_at()` - Retrieve expiration metadata for multiple keys
- `mget_keymeta_notes()` - Retrieve notes metadata for multiple keys
- `clear_keymeta()` - Clear metadata (set to NA) for key(s)

Asynchronous Operations

- `set_async()` - Set a key-value pair, asynchronously
- `mset_async()` - Set multiple key-value pairs, asynchronously
- `set_by_value_async()` - Set a key-value pair using hash, asynchronously
- `mset_by_value_async()` - Set multiple key-value pairs using hashes, asynchronously
- `update_async()` - Update a key-value pair and retain key-metadata, asynchronously
- `mupdate_async()` - Update multiple key-value pairs and retain key-metadata, asynchronously
- `update_keymeta_async()` - Update metadata, asynchronously
- `mupdate_keymeta_async()` - Update multiple metadata, asynchronously
- `clear_keymeta_async()` - Clear metadata, asynchronously

Object Hash Management

- `set_value()` - Add an R object and return its hash (internal use)

- `mset_value()` - Add multiple R objects and return their hashes (internal use)
- `get_hash()` - Get hash value for a key-namespace pair
- `mget_hash()` - Get hash values for multiple keys
- `hash_object()` - Create a hash digest for an R object

Key Management

- `exists()` - Check if key-namespace pair(s) exist
- `exists_object()` - Check if object(s) with given hash(es) exist
- `del()` - Delete key-namespace pair(s)
- `fill()` - Set one or more keys to the same value
- `duplicate()` - Duplicate/copy keys from source to destination

Expiration Management

- `keys_with_expiration()` - List keys that have expiration timestamps
- `expired_keys()` - Get keys that have already expired
- `has_expired_keys()` - Check if any keys are expired
- `is_key_expired()` - Check if a key is expired
- `clear_expired_keys()` - Remove expired key-namespace pairs

Notes Management

- `keys_with_notes()` - List keys that have notes

Listing

- `list()` - List all keys in a namespace
- `list_notes()` - List all notes in a namespace
- `list_hashes()` - List all stored object hashes
- `list_unused_hashes()` - List all stored object unused hashes
- `list_namespaces()` - List all namespaces

Storage Management

- `clear()` - Clear a namespace or all namespaces
- `gc()` - Garbage collect unused hashes
- `index_export()` - Export object index as data.table
- `index_import()` - Import objects from index
- `import()` - Import objects from another storr/list/environment
- `export()` - Export objects to another storr/list/environment
- `export_tdb()` - Export objects to another TileDB storr

See Also

[driver_tiledb\(\)](#) and [storr_tdb0\(\)](#) for standard interface.

Examples

```

# URI path
uri <- tempfile()
sto <- storr_tiledb(uri, init = TRUE)

# set key-values
sto$set("a", 1)
sto$set("b", 1, namespace = "ns1", notes = "note1")

# listing methods
sto$list("ns1") # b
sto$list_namespaces() # "ns1"      "objects"
sto$list_hashes() # "632336c518ae1c89ecf26ae5fbec5860"

# get methods
sto$get("a") # 1
sto$get("b", "ns1") # 1
sto$get_keymeta("b", "ns1") # list(expires_at = NA, notes = "note1")

#-----
#   Storr with encryption
#-----

# Requires a TileDB Context with encryption configuration parameters
key <- "0123456789abcdeF0123456789abcdeF"
config <- tiledb::tiledb_config()
config["sm.encryption_type"] <- "AES_256_GCM";
config["sm.encryption_key"] <- key
ctx <- new_context(config)

# Create a storr with context that encapsulates encryption configuration
uri_enc <- tempfile()
stoe <- storr_tiledb(uri_enc, init = TRUE, context = ctx)

stoe$set("a", 1)
stoe$get("a") # 1

# No access without the key
try(storr_tiledb(uri_enc)) # This fails

# Pass the context with encryption parameters
stoe_new <- storr_tiledb(uri_enc, context = ctx)
stoe_new$get("a") # 1

#-----
#   Storr without compression
#-----

uri_nocomp <- tempfile()

sto_nocomp <- storr_tiledb(uri_nocomp, init = TRUE, compression_level = NULL)

```

```

#-----
#   Storr with custom driver schemas
#-----

## Step 1: Modify schemas ---

ctx <- new_context()

# 'TileDBDriverSchemas' instance to modify 'SchemaKeys' and 'SchemaData'
cds <- driver_schemas(ctx = ctx)

# Set "NONE" filters to 'keys' and 'namespace' dimensions in 'keys' schema
cds$SchemaKeys$dim_key <- NA
cds$SchemaKeys$dim_namespace <- NA

# Set ZSTD filter with high compression to 'value' attribute in 'data' schema
flt <- tiledb::tiledb_filter("ZSTD", ctx = ctx)
flt <- tiledb::tiledb_filter_set_option(flt, "COMPRESSION_LEVEL", 22)
fl_list <- tiledb::tiledb_filter_list(flt, ctx = ctx)

# Apply filter to 'value' attribute
cds$SchemaData$attr_value <- fl_list

## Step 2: Pass modified schemas to storr ---

# Create a 'storr' with custom schemas using 'driver_schemas' argument
uric <- tempfile()

stoc <- storr_tiledb(uric, init = TRUE, driver_schemas = cds)

```

storr_timetravel

A Storr with Time-Travel

Description

Generate an instance of [StorrTimeTravel](#), a variant of [TileDBStorr](#) designed to query key-value data at specific points in time and in read-only mode with no write capabilities.

Usage

```
storr_timetravel(uri, default_namespace = "objects", context = NULL,
  timestamp = NULL)
```

Arguments

`uri` The URI path of storr.

default_namespace	The default namespace: "objects".
context	Optional tiledb_ctx object.
timestamp	Set a TileDB timestamp range that the resource will be opened at. Effective only for "READ" mode. Valid options: • A NULL value (default) • An R object coercible to POSIXct with length 1 which used for end timestamp, or length 2 with start, end timestamps • An object of class tiledb_timestamp. See set_tiledb_timestamp() Set a new timestamp with active field \$timestamp, see examples.

Value

An object of class [StorrTimeTravel](#), R6.

Class Methods Summary

For complete definitions, see **Methods** section in [StorrTimeTravel](#).

Active Fields

- timestamp - Get or set a TileDB timestamp range that the 'storr' will be opened at.

Single Key-Value Operations

- get() - Retrieve an object by key-namespace pair
- get_value() - Retrieve an object given its hash
- get_all() - Retrieve an object and its metadata by key-namespace pair

Multiple Key-Value Operations

- mget() - Get multiple objects by key-namespace pairs
- mget_value() - Get multiple objects by their hashes
- mget_all() - Retrieve multiple objects and its metadata by key-namespace pairs

Metadata Operations

- get_keymeta() - Retrieve metadata for a key
- mget_keymeta() - Retrieve metadata for multiple keys
- get_keymeta_expires_at() - Retrieve expiration metadata for a key
- get_keymeta_notes() - Retrieve notes metadata for a key
- mget_keymeta_expires_at() - Retrieve expiration metadata for multiple keys
- mget_keymeta_notes() - Retrieve notes metadata for multiple keys

Object Hash Management

- get_hash() - Get hash value for a key-namespace pair
- mget_hash() - Get hash values for multiple keys

Key Management

- `exists()` - Check if key-namespace pair(s) exist
- `exists_object()` - Check if object(s) with given hash(es) exist

Expiration Management

- `keys_with_expiration()` - List keys that have expiration timestamps
- `expired_keys()` - Get keys that have already expired
- `has_expired_keys()` - Check if any keys are expired
- `is_key_expired()` - Check if a key is expired

Notes Management

- `keys_with_notes()` - List keys that have notes

Listing

- `list()` - List all keys in a namespace
- `list_notes()` - List all notes in a namespace
- `list_hashes()` - List all stored object hashes
- `list_unused_hashes()` - List all stored object unused hashes
- `list_namespaces()` - List all namespaces

Storage Management

- `index_export()` - Export object index as data.table
- `export()` - Export objects to another storr/list/environment
- `export_tdb()` - Export objects to another TileDB storr

Examples

```
## Not run:
# URI path
uri <- tempfile()
sto <- storr_tiledb(uri, init = TRUE)

# set key-values
t0 <- Sys.time()
sto$set("a", 1)

t1 <- Sys.time()
sto$set("b", 2)

t2 <- Sys.time()

# Open storr with time-travel support at t0
stor <- storr_timetravel(uri, timestamp = t0)

# Read at t0
```

```

stor$list_hashes() # character(0), no hashes at t0

stor$get("a") # key 'a' ('objects') not found

# Read at t1
stor$timestamp <- t1
stor$get("a") # 1
stor$get("b") # key 'b' ('objects') not found

# Read at t2
stor$timestamp <- t2
stor$get("b") # 2

## End(Not run)

```

storr_vacuum	<i>Vacuum Storr Fragments</i>
--------------	-------------------------------

Description

Remove old consolidated fragments to free space.

Usage

```
storr_vacuum(uri, what = "all", async = FALSE, context = NULL)
```

Arguments

uri	The URI path of storr.
what	Which array to vacuum? Defaults to "all" arrays.
async	Should it vacuum asynchronously? Default is FALSE.
context	Optional tiledb_ctx object.

Details

The function supports selective vacuuming ("keys"/"data") or full ("all") and it can be run synchronously or asynchronously.

Value

When `async = FALSE`, it returns `TRUE` for successful vacuuming. For `async = TRUE`, it returns a `mirai::mirai()` object immediately; once resolved, it returns `TRUE` indicating vacuum success.

See Also

[storr Consolidate\(\)](#)

Examples

```
# URI path
uri <- tempfile()
sto <- storr_tiledb(uri, init = TRUE)

storr_vacuum(uri)
```

Index

- * **storr-utilities**
 - storr_copy, [8](#)
 - storr_move, [10](#)
 - storr_rename, [11](#)
- CAS, [4](#)
- digest, [4](#), [12](#)
- driver_dbi(), [6](#)
- driver_schemas, [2](#)
- driver_schemas(), [4](#), [13](#)
- driver_tiledb, [3](#)
- driver_tiledb(), [16](#)
- driver_tiledb_create(driver_tiledb), [3](#)
- driver_tiledb_create(), [12](#)
- hashtab(), [13](#)
- mirai, [12](#), [14](#)
- mirai::mirai(), [7](#), [21](#)
- serialize(), [13](#)
- set_allocation_size_preference(), [13](#)
- set_tiledb_timestamp(), [19](#)
- storr, [4](#)
- storr_consolidate, [7](#)
- storr_consolidate(), [10](#), [21](#)
- storr_copy, [8](#)
- storr_copy(), [10](#), [11](#)
- storr_fragments, [9](#)
- storr_move, [10](#)
- storr_move(), [8](#), [11](#)
- storr_rename, [11](#)
- storr_rename(), [8](#), [10](#)
- storr_tdb0(), [16](#)
- storr_tiledb, [12](#)
- storr_tiledb(), [4](#), [6](#)
- storr_timetravel, [18](#)
- storr_vacuum, [21](#)
- storr_vacuum(), [8](#), [10](#)
- StorrFragments, [9](#)
- StorrTimeTravel, [18](#), [19](#)
- tiledb_ctx, [2](#), [4](#), [7–12](#), [19](#), [21](#)
- TileDBDriver, [4](#)
- TileDBDriverSchemas, [3](#), [4](#)
- TileDBFragments, [9](#)
- TileDBStorr, [13](#), [14](#), [18](#)
- ‘storr’, [13](#)