Caterva

The Blosc Developers

Apr 10, 2023

CONTENTS

1 Getting Started 3

2 Caterva Reference 7

3 Development 19

4 Release notes 23

Index 25

Caterva is an open source C library specially designed to deal with large multidimensional, chunked, compressed datasets.

Getting Started

New to *caterva*? Check out the getting started guides. They contain an introduction to *caterva'* main concepts and an installation tutorial.

To the getting started guides

API Reference

The reference guide contains a detailed description of the caterva API. The reference describes how the functions work and which parameters can be used.

To the reference guide

Development

Saw a typo in the documentation? Want to improve existing functionalities? The contributing guidelines will guide you through the process of improving caterva.

To the development guide

Release Notes

Want to see what's new in the latest release? Check out the release notes to find out!

To the release notes

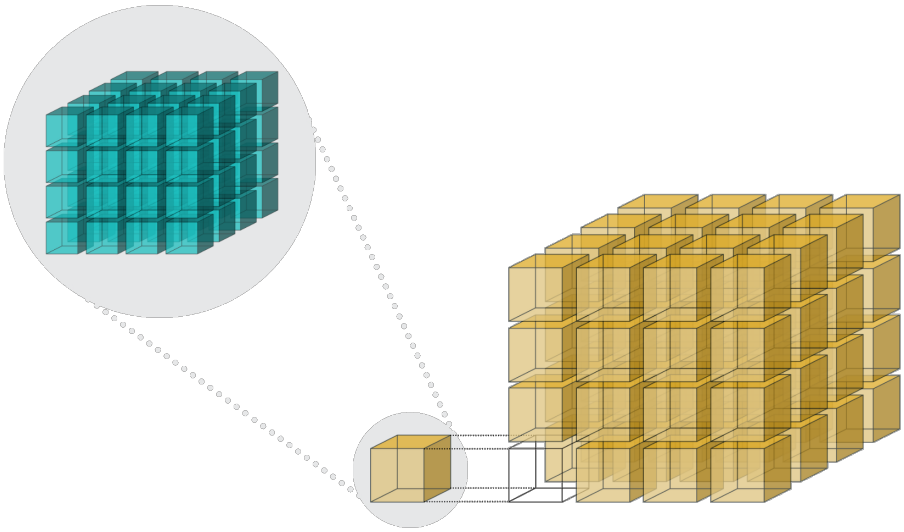
CHAPTER

ONE

GETTING STARTED

1.1 What is caterva?

Caterva is a container for multidimensional data that is specially designed to read, in an incredibly efficient way, datasets slices. To achieve this, a new chunking-based data layout has been created.



Like other libraries like Zarr, HDF5 or TileDB, Caterva stores the data into multidimensional chunks (yellow cubes). These chunks can then be read individually, improving performance when reading slices of the dataset. But also, Caterva introduces a new level of chunking. Within each chunk, the data is re-partitioned into smaller multidimensional sets called blocks (green cubes). In this way, Caterva can read blocks individually (and also in parallel) instead of chunks.

These partition levels allow to access data efficiently with a larger set of data access patterns. This is due to obtain the desired slice, instead of reading the data using the chunks, data is obtained using the blocks.

1.1.1 Blosc

In Caterva the compression is handled transparently for the user by leveraging the Blosc2 library. Blosc is an extremely fast compressor specially designed for binary data. It uses the blocking technique to reduce activity on the memory bus as much as possible. It also leverages SIMD (SSE2, AVX2 for Intel, NEON for ARM, Altivec for Power) and multi-threading capabilities present in multi-core processors so as to accelerate the compression/decompression process to a maximum.

Being able to store in an in-memory data container does not mean that data cannot be persisted. It is critical to find a way to store and retrieve data efficiently. Also, it is important to adopt open formats for reducing the maintenance burden and facilitate its adoption more quickly. Blosc2 brings such an efficient and open format for persistency.

An additional feature that introduces Blosc2 is the concept of metalayers. They are small metadata for informing about the kind of data that is stored on a Blosc2 container. They are handy for defining layers with different specs: data types, geo-spatial...

Caterva metalayer

Caterva is created by specifying a metalayer on top of a Blosc2 container for storing multidimensional information. Specifically, the Caterva metalayer is named 'caterva' and follows this format:

```
|-0-|-1-|-2-|-3-|~~~~~~~~~~~~~~~~|---|~~~~~~~~~~~~~~~~|---|~~~~~~~~~~~~~~~~|
| 9X| v | nd| 9X| shape          | 9X| chunkshape     | 9X| blockshape     |
|---|---|---|---|~~~~~~~~~~~~~~~~|---|~~~~~~~~~~~~~~~~|---|~~~~~~~~~~~~~~~~|
  ^   ^   ^   ^                    ^                    ^
  |   |   |   |                    |                    |
  |   |   |   |                    |                    +--[msgpack] fixarray with X=nd␣
↪elements
  |   |   |   |                    +--[msgpack] fixarray with X=nd elements
  |   |   |   +--[msgpack] fixarray with X=nd elements
  |   |   +--[msgpack] positive fixnum for the number of dimensions (up to 127)
  |   +--[msgpack] positive fixnum for the metalayer format version (up to 127)
  +---[msgpack] fixarray with X=5 elements
```

The *shape* section is meant to store the actual shape info:

```
|---|--8 bytes---|---|--8 bytes---|~~~~~|---|--8 bytes---|
| d3| first_dim  | d3| second_dim | ... | d3| nth_dim    |
|---|------------|---|------------|~~~~~|---|------------|
  ^               ^                     ^
  |               |                     |
  |               |                     +--[msgpack] int64
  |               +--[msgpack] int64
  +--[msgpack] int64
```

Next, the *chunkshape* section is meant to store the actual chunk shape info:

```
|---|--4 bytes---|---|--4 bytes---|~~~~~|---|--4 bytes---|
| d2| first_dim  | d2| second_dim | ... | d2| nth_dim    |
|---|------------|---|------------|~~~~~|---|------------|
  ^               ^                     ^
  |               |                     |
  |               |                     +--[msgpack] int32
```

(continues on next page)

(continued from previous page)

```
  |                +--[msgpack] int32
  +--[msgpack] int32
```

Finally, the *blockshape* section is meant to store the actual block shape info:

```
|---|--4 bytes---|---|--4 bytes---|~~~~~|---|--4 bytes---|
| d2| first_dim  | d2| second_dim | ... | d2| nth_dim    |
|---|------------|---|------------|~~~~~|---|------------|
  ^                ^                       ^
  |                |                       |
  |                |                       +--[msgpack] int32
  |                +--[msgpack] int32
  +--[msgpack] int32
```

1.2 Installation

Caterva can be built, tested and installed using CMake. The following procedure describes a typical CMake build.

1. Download the source code from Github:

   #### Unix

   ```
   git clone --recurse-submodules git@github.com:Blosc/caterva.git
   ```

   #### Windows

   ```
   git clone --recurse-submodules git@github.com:Blosc/caterva.git
   ```

2. Create the build directory inside the sources and move into it:

   #### Unix

   ```
   cd caterva
   mkdir build
   cd build
   ```

   #### Windows

   ```
   cd caterva
   mkdir build
   cd build
   ```

3. Now run CMake configuration:

Unix

```
cmake -DCMAKE_BUILD_TYPE='Debug/Release' ..
```

Windows

```
cmake ..
```

4. Build and test Caterva:

Unix

```
cmake --build .
ctest
```

Windows

```
cmake --build . --config 'Debug/Release'
ctest --build-config 'Debug/Release'
```

5. If desired, install Caterva:

Unix

```
cmake --build . --target install
```

Windows

```
cmake --build . --target install --config 'Debug/Release'
```

That's all folks!

1.3 Caterva Tutorials

Coming soon!

CHAPTER

TWO

CATERVA REFERENCE

Caterva has an internal structure managed by parameters that record the different properties of each dataset. The most fundamental parts of this structure are the Caterva context and the configuration parameters it depends on.

2.1 Context

struct **caterva_ctx_t**
Context for caterva arrays that specifies the functions used to manage memory and the compression/decompression parameters used in Blosc.

2.1.1 Configuration parameters

struct **caterva_config_t**
Configuration parameters used to create a caterva context.

Public Members

void *(***alloc**)(size_t)
The memory allocation function used internally.

void (***free**)(void*)
The memory release function used internally.

uint8_t **compcodec**
Defines the codec used in compression.

uint8_t **compmeta**
The metadata for the compressor codec.

uint8_t **complevel**
Determines the compression level used in Blosc.

int32_t **splitmode**
Whether the blocks should be split or not.

int **usedict**
: Indicates whether a dictionary is used to compress data or not.

int16_t **nthreads**
: Determines the maximum number of threads that can be used.

uint8_t **filters**[BLOSC2_MAX_FILTERS]
: Defines the filters used in compression.

uint8_t **filtersmeta**[BLOSC2_MAX_FILTERS]
: Indicates the meta filters used in Blosc.

blosc2_prefilter_fn **prefilter**
: Defines the function that is applied to the data before compressing it.

blosc2_prefilter_params ***pparams**
: Indicates the parameters of the prefilter function.

blosc2_btune ***udbtune**
: Indicates user-defined parameters for btune.

static const *caterva_config_t* **CATERVA_CONFIG_DEFAULTS**
: The default configuration parameters used in caterva.

2.1.2 Creation

int **caterva_ctx_new**(*caterva_config_t* *cfg, *caterva_ctx_t* **ctx)
: Create a context for caterva.

Parameters

- **cfg** – The configuration parameters needed for the context creation.
- **ctx** – The memory pointer where the context will be created.

Returns An error code.

2.1.3 Destruction

int **caterva_ctx_free**(*caterva_ctx_t* **ctx)
: Free a context.

Parameters **ctx** – The The context to be freed.

Returns An error code.

2.2 Array

A Caterva array is a multidimensional dataset (compressed or not) easy to handle using the Caterva functions. Furthermore, Caterva only stores item size instead of the data type and every item of a Caterva array has the same size. On the other hand, Caterva functions let users to perform different operations with these arrays like copying, slicing, setting them or converting them into buffers or files and vice versa.

struct **caterva_array_t**
A multidimensional array of data that can be compressed.

2.2.1 Parameters

General parameters

struct **caterva_params_t**
General parameters needed for the creation of a caterva array.

Public Members

uint8_t **itemsize**
The size of each item of the array.

int64_t **shape**[CATERVA_MAX_DIM]
The array shape.

int8_t **ndim**
The array dimensions.

Storage parameters

struct **caterva_storage_t**
The storage properties for an array backed by a Blosc super-chunk.

Public Members

int32_t **chunkshape**[CATERVA_MAX_DIM]
The shape of each chunk of Blosc.

int32_t **blockshape**[CATERVA_MAX_DIM]
The shape of each block of Blosc.

bool **contiguous**
Flag to indicate if the super-chunk is stored contiguously or sparsely.

char ***urlpath**
The super-chunk name.

If `urlpath` is not `NULL`, the super-chunk will be stored on disk.

caterva_metalayer_t **metalayers**[CATERVA_MAX_METALAYERS]
List with the metalayers desired.

int32_t **nmetalayers**
The number of metalayers.

struct **caterva_metalayer_t**
The metalayer data needed to store it on an array.

Public Members

char ***name**
The name of the metalayer.

uint8_t ***sdata**
The serialized data to store.

int32_t **size**
The size of the serialized data.

2.2.2 Creation

Fast constructors

int **caterva_empty**(*caterva_ctx_t* *ctx, *caterva_params_t* *params, *caterva_storage_t* *storage, *caterva_array_t* **array)
Create an empty array.

Parameters

- **ctx** – The caterva context to be used.
- **params** – The general params of the array desired.
- **storage** – The storage params of the array desired.
- **array** – The memory pointer where the array will be created.

Returns An error code.

int **caterva_zeros**(*caterva_ctx_t* *ctx, *caterva_params_t* *params, *caterva_storage_t* *storage, *caterva_array_t* **array)
Create an array, with zero being used as the default value for uninitialized portions of the array.

Parameters

- **ctx** – The caterva context to be used.
- **params** – The general params of the array.
- **storage** – The storage params of the array.
- **array** – The memory pointer where the array will be created.

Returns An error code.

int **caterva_full**(*caterva_ctx_t* *ctx, *caterva_params_t* *params, *caterva_storage_t* *storage, void *fill_value, *caterva_array_t* **array)

Create an array, with `fill_value` being used as the default value for uninitialized portions of the array.

Parameters

- **ctx** – The caterva context to be used.
- **params** – The general params of the array.
- **storage** – The storage params of the array.
- **fill_value** – Default value for uninitialized portions of the array.
- **array** – The memory pointer where the array will be created.

Returns An error code.

From/To buffer

int **caterva_from_buffer**(*caterva_ctx_t* *ctx, void *buffer, int64_t buffersize, *caterva_params_t* *params, *caterva_storage_t* *storage, *caterva_array_t* **array)

Create a caterva array from the data stored in a buffer.

Parameters

- **ctx** – The caterva context to be used.
- **buffer** – The buffer where source data is stored.
- **buffersize** – The size (in bytes) of the buffer.
- **params** – The general params of the array desired.
- **storage** – The storage params of the array desired.
- **array** – The memory pointer where the array will be created.

Returns An error code.

int **caterva_to_buffer**(*caterva_ctx_t* *ctx, *caterva_array_t* *array, void *buffer, int64_t buffersize)

Extract the data into a C buffer from a caterva array.

Parameters

- **ctx** – The caterva context to be used.
- **array** – The caterva array.
- **buffer** – The buffer where the data will be stored.
- **buffersize** – Size (in bytes) of the buffer.

Returns An error code.

From/To file

int **caterva_open**(*caterva_ctx_t* *ctx, const char *urlpath, *caterva_array_t* **array)
Read a caterva array from disk.

Parameters

- **ctx** – The caterva context to be used.
- **urlpath** – The urlpath of the caterva array on disk.
- **array** – The memory pointer where the array will be created.

Returns An error code.

int **caterva_save**(*caterva_ctx_t* *ctx, *caterva_array_t* *array, char *urlpath)
Save caterva array into a specific urlpath.

Parameters

- **ctx** – The context to be used.
- **array** – The array to be saved.
- **urlpath** – The urlpath where the array will be stored.

Returns An error code.

From Blosc object

int **caterva_from_schunk**(*caterva_ctx_t* *ctx, blosc2_schunk *schunk, *caterva_array_t* **array)
Create a caterva array from a super-chunk.

It can only be used if the array is backed by a blosc super-chunk.

Parameters

- **ctx** – The caterva context to be used.
- **schunk** – The blosc super-chunk where the caterva array is stored.
- **array** – The memory pointer where the array will be created.

Returns An error code.

From/To cframe buffer

int **caterva_from_cframe**(*caterva_ctx_t* *ctx, uint8_t *cframe, int64_t cframe_len, bool copy, *caterva_array_t* **array)
Create a caterva array from a serialized super-chunk.

Parameters

- **ctx** – The caterva context to be used.
- **cframe** – The buffer of the in-memory array.
- **cframe_len** – The size (in bytes) of the in-memory array.
- **copy** – Whether caterva should make a copy of the cframe data or not. The copy will be made to an internal sparse frame.
- **array** – The memory pointer where the array will be created.

Returns An error code.

int **caterva_to_cframe**(*caterva_ctx_t* *ctx, *caterva_array_t* *array, uint8_t **cframe, int64_t *cframe_len, bool *needs_free)

Create a serialized super-chunk from a caterva array.

Parameters

- **ctx** – The caterva context to be used.
- **array** – The caterva array to be serialized.
- **cframe** – The pointer of the buffer where the in-memory array will be copied.
- **cframe_len** – The length of the in-memory array buffer.
- **needs_free** – Whether the buffer should be freed or not.

Returns An error code

Copying

int **caterva_copy**(*caterva_ctx_t* *ctx, *caterva_array_t* *src, *caterva_storage_t* *storage, *caterva_array_t* **array)

Make a copy of the array data.

The copy is done into a new caterva array.

Parameters

- **ctx** – The caterva context to be used.
- **src** – The array from which data is copied.
- **storage** – The storage params of the array desired.
- **array** – The memory pointer where the array will be created.

Returns An error code

2.2.3 Slicing

int **caterva_get_slice_buffer**(*caterva_ctx_t* *ctx, *caterva_array_t* *array, int64_t *start, int64_t *stop, void *buffer, int64_t *buffershape, int64_t buffersize)

Get a slice from an array and store it into a C buffer.

Parameters

- **ctx** – The caterva context to be used.
- **array** – The array from which the slice will be extracted.
- **start** – The coordinates where the slice will begin.
- **stop** – The coordinates where the slice will end.
- **buffershape** – The shape of the buffer.
- **buffer** – The buffer where the data will be stored.
- **buffersize** – The size (in bytes) of the buffer.

Returns An error code.

int **caterva_set_slice_buffer**(*caterva_ctx_t* *ctx, void *buffer, int64_t *buffershape, int64_t buffersize, int64_t *start, int64_t *stop, *caterva_array_t* *array)

Set a slice into a caterva array from a C buffer.

Parameters

- **ctx** – The caterva context to be used.
- **buffer** – The buffer where the slice data is.
- **buffersize** – The size (in bytes) of the buffer.
- **start** – The coordinates where the slice will begin.
- **stop** – The coordinates where the slice will end.
- **buffershape** – The shape of the buffer.
- **array** – The caterva array where the slice will be set

Returns An error code.

int **caterva_get_slice**(*caterva_ctx_t* *ctx, *caterva_array_t* *src, const int64_t *start, const int64_t *stop, *caterva_storage_t* *storage, *caterva_array_t* **array)

Get a slice from an array and store it into a new array.

Parameters

- **ctx** – The caterva context to be used.
- **src** – The array from which the slice will be extracted
- **start** – The coordinates where the slice will begin.
- **stop** – The coordinates where the slice will end.
- **storage** – The storage params of the array desired.
- **array** – The memory pointer where the array will be created.

Returns An error code.

int **caterva_squeeze**(*caterva_ctx_t* *ctx, *caterva_array_t* *array)

Squeeze a caterva array.

This function remove single-dimensional entries from the shape of a caterva array.

Parameters

- **ctx** – The caterva context to be used.
- **array** – The caterva array.

Returns An error code

int **caterva_squeeze_index**(*caterva_ctx_t* *ctx, *caterva_array_t* *array, const bool *index)

Squeeze a caterva array.

This function remove selected single-dimensional entries from the shape of a caterva array.

Parameters

- **ctx** – The caterva context to be used.
- **array** – The caterva array.
- **index** – Indexes of the single-dimensional entries to remove.

Returns An error code

2.2.4 Destruction

int **caterva_free**(*caterva_ctx_t* *ctx, *caterva_array_t* **array)
Free an array.

Parameters

- **ctx** – The caterva context to be used.
- **array** – The memory pointer where the array is placed.

Returns An error code.

int **caterva_remove**(*caterva_ctx_t* *ctx, char *urlpath)
Remove a Caterva file from the file system.

Both backends are supported.

Parameters

- **ctx** – The caterva context to be used.
- **urlpath** – The urlpath of the array to be removed.

Returns An error code

2.3 Metalayers

Metalayers are metadata for informing about the properties of data that is stored on a container. Caterva implements its own metalayer on top of C-Blosc2 for storing multidimensional information. In general, you can use metalayers for adapting Blosc2 containers (and in particular, Caterva arrays) to your own needs.

2.3.1 Fixed-length metalayers

These small metalayers are stored in the header of the frame. This allows adding chunks to the frame without the need to rewrite the whole meta information and data coming after it. However, fixed-length metalayers cannot be resized and once the first chunk of data is added to the super-chunk, no more fixed-length metalayers can be added either.

int **caterva_meta_exists**(*caterva_ctx_t* *ctx, *caterva_array_t* *array, const char *name, bool *exists)
Check if a metalayer exists or not.

Parameters

- **ctx** – The context to be used.
- **array** – The array where the check will be done.
- **name** – The name of the metalayer to check.
- **exists** – Pointer where the result will be stored.

Returns An error code

int **caterva_meta_get**(*caterva_ctx_t* *ctx, *caterva_array_t* *array, const char *name, *caterva_metalayer_t* *meta)
Get a metalayer from a Caterva array.

> **Warning:** The contents of `meta` are allocated inside the function. Therefore, they must be released with a `free`.

Parameters

- **ctx** – The context to be used.
- **array** – The array where the metalayer will be added.
- **name** – The vl-metalayer name.
- **meta** – Pointer to the metalayer where the data will be stored.

Returns An error code

int **caterva_meta_update**(*caterva_ctx_t* *ctx, *caterva_array_t* *array, *caterva_metalayer_t* *meta)
Update a metalayer content in a Caterva array.

Parameters

- **ctx** – The context to be used.
- **array** – The array where the metalayer will be updated.
- **meta** – The metalayer to update.

Returns An error code

2.3.2 Variable-length metalayers

Unlike fixed-length metalayers, these can be resized and are stored in the trailer section of the frame. This provides users a lot of flexibility to define their own metadata.

int **caterva_vlmeta_add**(*caterva_ctx_t* *ctx, *caterva_array_t* *array, *caterva_metalayer_t* *vlmeta)
Add a vl-metalayer to the Caterva array.

Parameters

- **ctx** – The context to be used.
- **array** – The array where the metalayer will be added.
- **name** – The vl-metalayer to add.

Returns An error code

int **caterva_vlmeta_exists**(*caterva_ctx_t* *ctx, *caterva_array_t* *array, const char *name, bool *exists)
Check if a vl-metalayer exists or not.

Parameters

- **ctx** – The context to be used.
- **array** – The array where the check will be done.
- **name** – The name of the vl-metalayer to check.
- **exists** – Pointer where the result will be stored.

Returns An error code

int **caterva_vlmeta_get**(*caterva_ctx_t* *ctx, *caterva_array_t* *array, const char *name, *caterva_metalayer_t* *vlmeta)
Get a vl-metalayer from a Caterva array.

> **Warning:** The contents of `vlmeta` are allocated inside the function. Therefore, they must be released with a `free`.

Parameters

- **ctx** – The context to be used.
- **array** – The array where the vl-metalayer will be added.
- **name** – The vl-metalayer name.
- **vlmeta** – Pointer to the metalayer where the data will be stored.

Returns An error code

int **caterva_vlmeta_update**(*caterva_ctx_t* *ctx, *caterva_array_t* *array, *caterva_metalayer_t* *vlmeta)
Update a vl-metalayer content in a Caterva array.

Parameters

- **ctx** – The context to be used.
- **array** – The array where the vl-metalayer will be updated.
- **vlmeta** – The vl-metalayer to update.

Returns An error code

2.4 High-level APIs

Here you can find a list with the wrappers available for Caterva:

- Python Caterva: a Pythonic wrapper of Caterva.

CHAPTER

THREE

DEVELOPMENT

3.1 Contributing to Caterva

Caterva is a community maintained project. We want to make contributing to this project as easy and transparent as possible.

3.1.1 Asking for help

If you have a question about how to use Caterva, please post your question on StackOverflow using the "caterva" tag.

3.1.2 Bug reports

We use GitHub issues to track public bugs. Please ensure your description is clear and has sufficient instructions to be able to reproduce the issue. The ideal report should contain the following:

1. Summarize the problem: Include details about your goal, describe expected and actual results and include any error messages.

2. Describe what you've tried: Show what you've tried, tell us what you found and why it didn't meet your needs.

3. Minimum reproducible example: Share the minimum amount of code needed to reproduce your issue. You can format the code nicely using markdown:

```
```C
#import <caterva.h>

int main() {
    ...
}
```
```

4. Determine the environment: Indicates the Caterva version and the operating system the code is running on.

3.1.3 Contributing to code

We actively welcome your code contributions. By contributing to Caterva, you agree that your contributions will be licensed under the <LICENSE> file of the project.

Fork the repo

Make a fork of the Caterva repository and clone it:

```
git clone https://github.com/<your-github-username>/caterva
```

Create your branch

Before you do any new work or submit a pull request, please open an issue on GitHub to report the bug or propose the feature you'd like to add.

Then create a new, separate branch for each piece of work you want to do.

Update docstrings

If you've changed APIs, update the involved docstrings using the doxygen format.

Run the test suite

If you have added code that needs to be tested, add the necessary tests and verify that all tests pass successfully.

3.2 Roadmap

Caterva is a multidimensional container for binary data. It is a pure C library, allowing for better interoperatibility between different languages (although Python stands out high in the list).

This document lists the goals for a production release of Caterva.

3.2.1 Existing features

- **Built on top of Blosc2:** besides transparent compression, this allows to store large compressed datasets either in-memory or on-disk. In addition, Caterva inherits all the improvements that are being introduced in Blosc2 (see https://github.com/Blosc/c-blosc2/blob/main/ROADMAP.rst).
- **Two-level multidimensional chunking:** like other libraries, Caterva stores the data into multidimensional chunks for efficient slicing. But in addition, Caterva introduces a new level of chunking. Within each chunk, the data is re-chunked into smaller multidimensional sets called blocks, leading to more fine-grained, and hence, to even more efficient slicing capabilities.
- **Update values:** it allows to populate an array in any order.

3.2.2 Actions to be done

- **Resize array dimensions:** this will allow to increase or decrease in size any dimension of the arrays.
- **Improve slicing capabilities:** currently Caterva only supports basic slicing based on *start:stop* ranges; we would like to extend this to *start:stop:step* as well as selections based on an array of booleans (similar to NumPy).
- **Add support for DLPack:** support for DLPack would be nice for being able to share data between different frameworks and devices. This should complement (or even replace in the long term) the existing plainbuffer support. See this dicussion for more insight on what advantages could the support for DLPack bring for Caterva.
- **Support for multidimensional filters:** this will improve the in-memory spatial locally for data that is **n-dim closer** in the array; by n-dim closer we mean that the multidimensional norm (in an Euclidean space) between two different positions of elements is shorter. This may led to better compression opportunities when spatial locality (Euclidean space) is high.
- **Support for multidimensional codecs:** this is the equivalent for multidim filters, but for codecs. Multidim codecs can leverage n-dim spatial locality in order to compress better/faster. Such codecs could be used in combination with others, uni-dim codecs (e.g. LZ4), so as to get better ratios.
- **Provide wheels:** this will make the installation much easier for the user.

3.2.3 Outreaching

- **Improve the main Caterva README:** this should allow a better overview at first glance of all the features that Caterva offers right now.
- **Attend to meetings and conferences:** it is very important to plan going to conferences for advertising Caterva and meeting people in-person. We need to decide which meetings to attend. As there are not that much conferences about C libraries, it is important to leverage the python-caterva wrapper so as to try to promote Caterva on Python conferences too.
- Other outreaching activities would be to produce videos of the kind 'Caterva in 10 minutes', or producing compelling tutorials (preferably based on Jupyter notebook, and using services like binder that allows a low entry level for quick trials).

3.3 Code of Conduct

The Blosc community has adopted a Code of Conduct that we expect project participants to adhere to. Please read the full text so that you can understand what actions will and will not be tolerated.

CHAPTER

FOUR

RELEASE NOTES

4.1 Changes from 0.5.0 to 0.5.1

XXX version-specific blurb XXX

4.2 Changes from 0.4.0 to 0.5.0

- Redesign the caterva guts and perform a code refactorization in order to simplify the code. This includes API renaming. A performance improvement is obtained as can be seen in https://github.com/Blosc/caterva/pull/69#issuecomment-843835622
- Implement a set_slice for arrays backed by Blosc. This allows users to update the values in the array whenever and wherever they want.
- Implement constructors (empty, zeros, full) using the special-values features introduced in Blosc.
- Use the *pydata_sphinx_theme* in the documentation.

4.3 Changes from 0.3.3 to 0.4.0

- API renaming. The function names simulate a *namespace* and some variable names have been changed to those used by the general community (*part* -> *chunk*).
- Add a new level of multi-dimensionality. As a result, unlike other libraries, Caterva supports two levels of multi-dimensional chunking (chunks and blocks).
- Improve library compilation to allow users to avoid building tests and examples.
- Simplify the test's suite for a proper integration in Windows.
- Update documentation to improve the library description and to add the *Release notes* and the *Roadmap* in a new section.

4.4 Changes from 0.3.0 to 0.3.3

- Fixing that 0.3.1 and 0.3.2 tags were not made in master :-/

4.5 Changes from 0.2.2 to 0.3.0

- Big code and API refactorization. As result, the API is more consistent and hopefully more intuitive to use. For more info on the new API, see https://caterva.readthedocs.io.

4.6 Changes from 0.2.1 to 0.2.2

- Add a *caterva_from_sframe()* function.

4.7 Changes from 0.2.0 to 0.2.1

- Both static and dynamic libraries are created by default now. If you want to disable the creation of one of them, just set the cmake options *SHARED_LIB=OFF* or *STATIC_LIB=OFF*.
- Add a *copy* parameter to *caterva_from_file()*.

INDEX

C

caterva_array_t (*C++ struct*), 9
CATERVA_CONFIG_DEFAULTS (*C++ member*), 8
caterva_config_t (*C++ struct*), 7
caterva_config_t::alloc (*C++ member*), 7
caterva_config_t::compcodec (*C++ member*), 7
caterva_config_t::complevel (*C++ member*), 7
caterva_config_t::compmeta (*C++ member*), 7
caterva_config_t::filters (*C++ member*), 8
caterva_config_t::filtersmeta (*C++ member*), 8
caterva_config_t::free (*C++ member*), 7
caterva_config_t::nthreads (*C++ member*), 8
caterva_config_t::pparams (*C++ member*), 8
caterva_config_t::prefilter (*C++ member*), 8
caterva_config_t::splitmode (*C++ member*), 7
caterva_config_t::udbtune (*C++ member*), 8
caterva_config_t::usedict (*C++ member*), 7
caterva_copy (*C++ function*), 13
caterva_ctx_free (*C++ function*), 8
caterva_ctx_new (*C++ function*), 8
caterva_ctx_t (*C++ struct*), 7
caterva_empty (*C++ function*), 10
caterva_free (*C++ function*), 15
caterva_from_buffer (*C++ function*), 11
caterva_from_cframe (*C++ function*), 12
caterva_from_schunk (*C++ function*), 12
caterva_full (*C++ function*), 10
caterva_get_slice (*C++ function*), 14
caterva_get_slice_buffer (*C++ function*), 13
caterva_meta_exists (*C++ function*), 15
caterva_meta_get (*C++ function*), 15
caterva_meta_update (*C++ function*), 16
caterva_metalayer_t (*C++ struct*), 10
caterva_metalayer_t::name (*C++ member*), 10
caterva_metalayer_t::sdata (*C++ member*), 10
caterva_metalayer_t::size (*C++ member*), 10
caterva_open (*C++ function*), 12
caterva_params_t (*C++ struct*), 9
caterva_params_t::itemsize (*C++ member*), 9
caterva_params_t::ndim (*C++ member*), 9
caterva_params_t::shape (*C++ member*), 9
caterva_remove (*C++ function*), 15
caterva_save (*C++ function*), 12
caterva_set_slice_buffer (*C++ function*), 13
caterva_squeeze (*C++ function*), 14
caterva_squeeze_index (*C++ function*), 14
caterva_storage_t (*C++ struct*), 9
caterva_storage_t::blockshape (*C++ member*), 9
caterva_storage_t::chunkshape (*C++ member*), 9
caterva_storage_t::contiguous (*C++ member*), 9
caterva_storage_t::metalayers (*C++ member*), 10
caterva_storage_t::nmetalayers (*C++ member*), 10
caterva_storage_t::urlpath (*C++ member*), 9
caterva_to_buffer (*C++ function*), 11
caterva_to_cframe (*C++ function*), 13
caterva_vlmeta_add (*C++ function*), 16
caterva_vlmeta_exists (*C++ function*), 16
caterva_vlmeta_get (*C++ function*), 16
caterva_vlmeta_update (*C++ function*), 17
caterva_zeros (*C++ function*), 10