Blosc 2 Documentation

Release 2.0.0.beta.5

Francesc Alted

Apr 22, 2020

TABLE OF CONTENTS:

Author The Blosc Development Team

Contact blosc@blosc.org

URL http://www.blosc.org

Gitter

Travis CI

Appveyor

NumFOCUS

CHAPTER

ONE

WHAT IS IT?

Blosc is a high performance compressor optimized for binary data (i.e. floating point numbers, integers and booleans). It has been designed to transmit data to the processor cache faster than the traditional, non-compressed, direct memory fetch approach via a memcpy() OS call. Blosc main goal is not just to reduce the size of large datasets on-disk or in-memory, but also to accelerate memory-bound computations.

C-Blosc2 is the new major version of C-Blosc, with full support for 64-bit containers, filter pipelining, new filters, new codecs and dictionaries for improved compression ratio. The new 64-bit data containers support both sparse (super-chunks) and sequential (frames) storage, either in-memory or on-disk. The frame is a sequential format that is very simple and meant to be used for either persistency or send to other processes or machines. Finally, the frames can be annotated with metainfo (metalayers, usermeta) that is provided by the user. More info about the improved capabilities of C-Blosc2 can be found in this talk.

C-Blosc2 tries hard to be backward compatible with both the C-Blosc1 API and in-memory format. Furthermore, if you just use the C-Blosc1 API you are guaranteed to generate compressed data containers that can be read with C-Blosc2, but getting the benefit of better performance, like for example leveraging the accelerated versions of codecs present in Intel's IPP (LZ4 is supported now and others will follow).

C-Blosc2 is currently in beta stage, so not ready to be used in production yet. Having said this, the beta stage means that the API has been declared frozen, so there is guarantee that your programs will continue to work with future versions of the library. If you want to collaborate in this development you are welcome. We need help in the different areas listed at the ROADMAP; also, be sure to read our DEVELOPING-GUIDE. Blosc is distributed using the BSD license.

CHAPTER

TWO

META-COMPRESSION AND OTHER ADVANTAGES OVER EXISTING COMPRESSORS

C-Blosc2 is not like other compressors: it should rather be called a meta-compressor. This is so because it can use different compressors and filters (programs that generally improve compression ratio). At any rate, it can also be called a compressor because it happens that it already comes with several compressor and filters, so it can actually work like so.

Currently C-Blosc2 comes with support of BloscLZ, a compressor heavily based on FastLZ, LZ4 and LZ4HC, Zstd, Lizard and Zlib, via miniz:, as well as a highly optimized (it can use SSE2, AVX2, NEON or ALTIVEC instructions, if available) shuffle and bitshuffle filters (for info on how shuffling works, see slide 17 of http://www.slideshare.net/PyData/blosc-py-data-2014).

Blosc is in charge of coordinating the different compressor and filters so that they can leverage the blocking technique as well as multi-threaded execution automatically. That makes that every codec and filter in the pipeline will run efficiently on modern CPUs, even if it was not initially designed for doing blocking or multi-threading.

Another important aspect of C-Blosc2 is that it splits large datasets in smaller containers called *chunks*, which are basically Blosc1 containers. For maximum performance, these chunks are meant to fit in the LLC (Last Level Cache) of CPUs. In practice this means that in order to leverage C-Blosc2 containers effectively, the user should ask for C-Blosc2 to uncompress the chunks, consume them before they hit main memory and then proceed with the new chunk (as in any streaming operation). We call this process *Streamed Compressed Computing* and it effectively avoids uncompressed data to travel to RAM, saving precious time in modern architectures where RAM access is very expensive compared with CPU speeds.

CHAPTER

THREE

MULTIDIMENSIONAL CONTAINERS

As said, C-Blosc2 adds a powerful mechanism for adding different metalayers on top of its containers. Caterva is a sibling library that adds such a metalayer specifying not only the dimensionality of a dataset, but also the dimensionality of the chunks inside the dataset. In addition, Caterva adds machinery for retrieving arbitrary multi-dimensional slices (aka hyper-slices) out of the multi-dimensional containers in the most efficient way. Hence, Caterva brings the convenience of multi-dimensional containers to your application very easily. For more info, check out the Caterva documentation.

CHAPTER

FOUR

COMPILING THE C-BLOSC2 LIBRARY WITH CMAKE

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

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

```
$ cd c-blosc2-sources
$ mkdir build
$ cd build
```

Now run CMake configuration and optionally specify the installation directory (e.g. '/usr' or '/usr/local'):

```
$ cmake -DCMAKE_INSTALL_PREFIX=your_install_prefix_directory ..
```

CMake allows to configure Blosc in many different ways, like prefering internal or external sources for compressors or enabling/disabling them. Please note that configuration can also be performed using UI tools provided by CMake (*ccmake* or *cmake-gui*):

```
$ ccmake ..       # run a curses-based interface
$ cmake-gui ..    # run a graphical interface
```

Build, test and install Blosc:

```
$ cmake --build .
$ ctest
$ cmake --build . --target install
```

The static and dynamic version of the Blosc library, together with header files, will be installed into the specified CMAKE_INSTALL_PREFIX.

Once you have compiled your Blosc library, you can easily link your apps with it as shown in the examples/ directory.

4.1 Handling support for codecs (LZ4, LZ4HC, Zstd, Lizard, Zlib)

C-Blosc2 comes with full sources for LZ4, LZ4HC, Zstd, Lizard and Zlib and in general, you should not worry about not having (or CMake not finding) the libraries in your system because by default the included sources will be automatically compiled and included in the C-Blosc2 library. This means that you can be confident in having a complete support for all the codecs in all the Blosc deployments (unless you are explicitly excluding support for some of them).

If you want to force Blosc to use external libraries instead of the included compression sources:

```
$ cmake -DPREFER_EXTERNAL_LZ4=ON ..
```

You can also disable support for some compression libraries:

```
$ cmake -DDEACTIVATE_SNAPPY=ON ..
```

4.2 Supported platforms

C-Blosc2 is meant to support all platforms where a C99 compliant C compiler can be found. The ones that are mostly tested are Intel (Linux, Mac OSX and Windows) and ARM (Linux), but exotic ones as IBM Blue Gene Q embedded "A2" processor are reported to work too.

For Windows, you will need at least VS2015 or higher on x86 and x64 targets (i.e. ARM is not supported on Windows).

For Mac OSX, make sure that you have installed the command line developer tools. You can always install them with:

```
$ xcode-select --install
```

4.3 Support for the LZ4 optimized version in Intel IPP

C-Blosc2 comes with support for a highly optimized version of the LZ4 codec present in Intel IPP, and actually if the cmake machinery in C-Blosc2 discovers IPP installed in your system it will use it automatically by default. Here it is a way to easily install Intel IPP in Ubuntu machines:

```
$ wget https://apt.repos.intel.com/intel-gpg-keys/GPG-PUB-KEY-INTEL-SW-PRODUCTS-2019.
↪PUB
$ apt-key add GPG-PUB-KEY-INTEL-SW-PRODUCTS-2019.PUB
$ sudo sh -c 'echo deb https://apt.repos.intel.com/ipp all main > /etc/apt/sources.
↪list.d/intel-ipp.list'
$ sudo apt-get update && sudo apt-get install intel-ipp-64bit-2019.X  # replace .X by␣
↪the latest version
```

Check Intel IPP website for instructions on how to install it for other platforms.

CHAPTER

FIVE

MAILING LIST

There is an official mailing list for Blosc at:

blosc@googlegroups.com http://groups.google.es/group/blosc

CHAPTER

SIX

ACKNOWLEDGMENTS

See THANKS.rst.

Enjoy data!

6.1 C-Blosc2 API

This section contains the C-Blosc2 public API and the structures needed to use it. C-Blosc2 tries to be backward compatible with both the C-Blosc1 API and format. Furthermore, if you just use the C-Blosc1 API you are guaranteed to generate compressed data containers that can be read with a Blosc1 library.

Having said that, the C-Blosc2 API gives you much more functionality, like 63-bit data containers, more filters, more support for vector instructions, support for accelerated versions of some codecs in Intel's IPP (like LZ4), the ability to work with data either in-memory or on-disk (frames) or attach metainfo to your datasets (metalayers).

6.1.1 Utility variables

This are enum values which avoid the nuisance of remembering codes and IDs.

Limits for different features

BLOSC_MIN_HEADER_LENGTH = 16
Minimum header length (Blosc1)

BLOSC_EXTENDED_HEADER_LENGTH = 32
Extended header length (Blosc2, see README_HEADER)

BLOSC_MAX_OVERHEAD = *BLOSC_EXTENDED_HEADER_LENGTH*

BLOSC_MIN_BUFFERSIZE = 128
Maximum typesize before considering source buffer as a stream of bytes Cannot be larger than 255.

BLOSC_MAX_BUFFERSIZE = (INT_MAX - *BLOSC_MAX_OVERHEAD*)
The maximum overhead during compression in bytes.

This equals to BLOSC_EXTENDED_HEADER_LENGTH now, but can be higher in future implementations- Maximum source buffer size to be compressed

BLOSC_MAX_TYPESIZE = 255

BLOSC2_MAX_FILTERS = 6
Maximum number of filters in the filter pipeline.

Codes for filters

BLOSC_NOSHUFFLE = 0
no shuffle (for compatibility with Blosc1)

BLOSC_NOFILTER = 0
no filter

BLOSC_SHUFFLE = 1
byte-wise shuffle

BLOSC_BITSHUFFLE = 2
bit-wise shuffle

BLOSC_DELTA = 3
delta filter

BLOSC_TRUNC_PREC = 4
truncate precision filter

BLOSC_LAST_FILTER = 5
sentinel

Internal flags (blosc_cbuffer_metainfo)

BLOSC_DOSHUFFLE = 0x1
byte-wise shuffle

BLOSC_MEMCPYED = 0x2
plain copy

BLOSC_DOBITSHUFFLE = 0x4
bit-wise shuffle

BLOSC_DODELTA = 0x8
delta coding

Compression dictionaries

BLOSC2_USEDICT = 0x1
use dictionaries with codec

BLOSC2_MAXDICTSIZE = 128 * 1024
maximum size for compression dicts

Compressor codecs

BLOSC_BLOSCLZ = 0

BLOSC_LZ4 = 1

BLOSC_LZ4HC = 2

BLOSC_SNAPPY = 3

BLOSC_ZLIB = 4

BLOSC_ZSTD = 5

BLOSC_LIZARD = 6

Compressor names

BLOSC_BLOSCLZ_COMPNAME "blosclz"

BLOSC_LZ4_COMPNAME "lz4"

BLOSC_LZ4HC_COMPNAME "lz4hc"

BLOSC_SNAPPY_COMPNAME "snappy"

BLOSC_ZLIB_COMPNAME "zlib"

BLOSC_ZSTD_COMPNAME "zstd"

BLOSC_LIZARD_COMPNAME "lizard"

6.1.2 Blosc1 API

This is the classic API from Blosc1 with 32-bit limited containers.

Main API

BLOSC_EXPORT void blosc_init(void)
Initialize the Blosc library environment.

You must call this previous to any other Blosc call, unless you want Blosc to be used simultaneously in a multi-threaded environment, in which case you can use the

See #blosc2_compress_ctx #blosc2_decompress_ctx pair.

BLOSC_EXPORT void blosc_destroy(void)
Destroy the Blosc library environment.

You must call this after to you are done with all the Blosc calls, unless you have not used blosc_init() before

See #blosc_init.

blosc_compress()

BLOSC_EXPORT int blosc_compress(int clevel, int doshuffle, size_t typesize, size_t nbytes,
Compress a block of data in the `src` buffer and returns the size of compressed block.

Parameters

- `clevel`: The desired compression level and must be a number between 0 (no compression) and 9 (maximum compression).
- `doshuffle`: Specifies whether the shuffle compression preconditioner should be applied or not. *BLOSC_NOFILTER* means not applying filters, *BLOSC_SHUFFLE* means applying shuffle at a byte level and *BLOSC_BITSHUFFLE* at a bit level (slower but *may* achieve better compression).
- `typesize`: Is the number of bytes for the atomic type in binary `src` buffer. This is mainly useful for the shuffle preconditioner. For implementation reasons, only a 1 < typesize < 256 will allow the shuffle filter to work. When typesize is not in this range, shuffle will be silently disabled.
- `nbytes`: The number of bytes to compress in the `src` buffer.
- `src`: The buffer containing the data to compress.
- `dest`: The buffer where the compressed data will be put, must have at least the size of `destsize`.

- `destsize`: The size of the dest buffer. Blosc guarantees that if you set `destsize` to, at least, (`nbytes` + *BLOSC_MAX_OVERHEAD*), the compression will always succeed.

Remark Compression is memory safe and guaranteed not to write `dest` more than what is specified in `destsize`. There is not a minimum for `src` buffer size `nbytes`.

Warning The `src` buffer and the `dest` buffer can not overlap.

Return The number of bytes compressed. If `src` buffer cannot be compressed into `destsize`, the return value is zero and you should discard the contents of the `dest` buffer. A negative return value means that an internal error happened. This should never happen. If you see this, please report it back together with the buffer data causing this and compression settings.

blosc_compress() honors different environment variables to control internal parameters without the need of doing that programatically. Here are the ones supported:

BLOSC_CLEVEL=(INTEGER): This will overwrite the @p clevel parameter before the compression process starts.

BLOSC_SHUFFLE=[NOSHUFFLE | SHUFFLE | BITSHUFFLE]: This will overwrite the *doshuffle* parameter before the compression process starts.

BLOSC_DELTA=(1|0): This will call *blosc_set_delta()*^ before the compression process starts.

BLOSC_TYPESIZE=(INTEGER): This will overwrite the *typesize* parameter before the compression process starts.

BLOSC_COMPRESSOR=[BLOSCLZ | LZ4 | LZ4HC | LIZARD | SNAPPY | ZLIB]: This will call *blosc_set_compressor(BLOSC_COMPRESSOR)* before the compression process starts.

BLOSC_NTHREADS=(INTEGER): This will call *blosc_set_nthreads(BLOSC_NTHREADS)* before the compression process starts.

BLOSC_BLOCKSIZE=(INTEGER): This will call *blosc_set_blocksize(BLOSC_BLOCKSIZE)* before the compression process starts. *NOTE:* The blocksize is a critical parameter with important restrictions in the allowed values, so use this with care.

BLOSC_NOLOCK=(ANY VALUE): This will call *blosc2_compress_ctx()* under the hood, with the *compressor*, *blocksize* and *numinternalthreads* parameters set to the same as the last calls to *blosc_set_compressor*, *blosc_set_blocksize* and *blosc_set_nthreads*. *BLOSC_CLEVEL*, *BLOSC_SHUFFLE*, *BLOSC_DELTA* and *BLOSC_TYPESIZE* environment vars will also be honored.

blosc_decompress()

BLOSC_EXPORT int blosc_decompress(const void * src, void * dest, size_t destsize)

Decompress a block of compressed data in `src`, put the result in `dest` and returns the size of the decompressed block.

Parameters

- `src`: The buffer to be decompressed.
- `dest`: The buffer where the decompressed data will be put.
- `destsize`: The size of the `dest` buffer.

Warning The `src` buffer and the `dest` buffer can not overlap.

Remark Decompression is memory safe and guaranteed not to write the `dest` buffer more than what is specified in `destsize`.

Remark In case you want to keep under control the number of bytes read from source, you can call #blosc_cbuffer_sizes first to check whether the `nbytes` (i.e. the number of bytes to be read from `src` buffer by this function) in the compressed buffer is ok with you.

Return The number of bytes decompressed. If an error occurs, e.g. the compressed data is corrupted or the output buffer is not large enough, then 0 (zero) or a negative value will be returned instead.

blosc_decompress honors different environment variables to control internal parameters without the need of doing that programatically. Here are the ones supported:

BLOSC_NTHREADS=(INTEGER): This will call *blosc_set_nthreads(BLOSC_NTHREADS)* before the proper decompression process starts.

BLOSC_NOLOCK=(ANY VALUE): This will call *blosc2_decompress_ctx* under the hood, with the *numinternalthreads* parameter set to the same value as the last call to *blosc_set_nthreads*.

`BLOSC_EXPORT int blosc_getitem(const void * src, int start, int nitems, void * dest)`
Get `nitems` (of `typesize` size) in `src` buffer starting in `start`.

The items are returned in `dest` buffer, which has to have enough space for storing all items.

Parameters

- `src`: The compressed buffer from data will be decompressed.
- `start`: The position of the first item (of `typesize` size) from where data will be retrieved.
- `nitems`: The number of items (of `typesize` size) that will be retrieved.
- `dest`: The buffer where the decompressed data retrieved will be put.

Return The number of bytes copied to `dest` or a negative value if some error happens.

`BLOSC_EXPORT int blosc_get_nthreads(void)`
Returns the current number of threads that are used for compression/decompression.

`BLOSC_EXPORT int blosc_set_nthreads(int nthreads)`
Initialize a pool of threads for compression/decompression.

If `nthreads` is 1, then the serial version is chosen and a possible previous existing pool is ended. If this is not called, `nthreads` is set to 1 internally.

Parameters

- `nthreads`: The number of threads to use.

Return The previous number of threads.

`BLOSC_EXPORT const char* blosc_get_compressor(void)`
Get the current compressor that is used for compression.

Return The string identifying the compressor being used.

`BLOSC_EXPORT int blosc_set_compressor(const char * compname)`
Select the compressor to be used.

The supported ones are “blosclz”, “lz4”, “lz4hc”, “snappy”, “zlib” and “ztsd”. If this function is not called, then “blosclz” will be used.

Parameters

- `compname`: The name identifier of the compressor to be set.

Return The code for the compressor (>=0). In case the compressor is not recognized, or there is not support for it in this build, it returns a -1.

BLOSC_EXPORT void blosc_set_delta(int dodelta)

Select the delta coding filter to be used.

This call should always succeed.

Parameters

- `dodelta`: A value >0 will activate the delta filter. If 0, it will be de-activated

BLOSC_EXPORT int blosc_free_resources(void)

Free possible memory temporaries and thread resources.

Use this when you are not going to use Blosc for a long while.

Return A 0 if succeeds, in case of problems releasing the resources, it returns a negative number.

Compressed buffer information

BLOSC_EXPORT void blosc_cbuffer_sizes(const void * cbuffer, size_t * nbytes, size_t * cbyte

Get information about a compressed buffer, namely the number of uncompressed bytes (`nbytes`) and compressed (`cbytes`).

It also returns the `blocksize` (which is used internally for doing the compression by blocks).

You only need to pass the first BLOSC_EXTENDED_HEADER_LENGTH bytes of a compressed buffer for this call to work.

Parameters

- `cbuffer`: The buffer of compressed data.
- `nbytes`: The pointer where the number of uncompressed bytes will be put.
- `cbytes`: The pointer where the number of compressed bytes will be put.
- `blocksize`: The pointer where the block size will be put.

This function should always succeed.

BLOSC_EXPORT void blosc_cbuffer_metainfo(const void * cbuffer, size_t * typesize, int * fla

Get information about a compressed buffer, namely the type size (`typesize`), as well as some internal `flags`.

You can use the `BLOSC_DOSHUFFLE`, `BLOSC_DOBITSHUFFLE`, `BLOSC_DODELTA` and `BLOSC_MEMCPYED` symbols for extracting the interesting bits (e.g. `flags & BLOSC_DOSHUFFLE` says whether the buffer is byte-shuffled or not).

Parameters

- `cbuffer`: The buffer of compressed data.
- `typesize`: The pointer where the type size will be put.
- `flags`: The pointer of the integer where the additional info is encoded. The `flags` is a set of bits, where the currently used ones are:
  - bit 0: whether the shuffle filter has been applied or not
  - bit 1: whether the internal buffer is a pure memcpy or not
  - bit 2: whether the bitshuffle filter has been applied or not
  - bit 3: whether the delta coding filter has been applied or not

This function should always succeed.

BLOSC_EXPORT void blosc_cbuffer_versions(const void * cbuffer, int * version, int * versio

Get information about a compressed buffer, namely the internal Blosc format version (`version`) and the format for the internal Lempel-Ziv compressor used (`versionlz`).

This function should always succeed.

Parameters

- `cbuffer`: The buffer of compressed data.
- `version`: The pointer where the Blosc format version will be put.
- `versionlz`: The pointer where the Lempel-Ziv version will be put.

BLOSC_EXPORT const char* blosc_cbuffer_complib(const void * cbuffer)

Get the compressor library/format used in a compressed buffer.

This function should always succeed.

Parameters

- `cbuffer`: The buffer of compressed data.

Return The string identifying the compressor library/format used.

Utility functions

BLOSC_EXPORT int blosc_compcode_to_compname(int compcode, const char ** compname)

Get the compressor name associated with the compressor code.

Parameters

- `compcode`: The code identifying the compressor
- `compname`: The pointer to a string where the compressor name will be put.

Return The compressor code. If the compressor code is not recognized, or there is not support for it in this build, -1 is returned.

BLOSC_EXPORT int blosc_compname_to_compcode(const char * compname)

Get the compressor code associated with the compressor name.

Parameters

- `compname`: The string containing the compressor name.

Return The compressor code. If the compressor name is not recognized, or there is not support for it in this build, -1 is returned instead.

BLOSC_EXPORT const char* blosc_list_compressors(void)

Get a list of compressors supported in the current build.

This function does not leak, so you should not free() the returned list.

Return The comma separated string with the list of compressor names supported.

This function should always succeed.

BLOSC_EXPORT const char* blosc_get_version_string(void)

Get the version of Blosc in string format.

Return The string with the current Blosc version. Useful for dynamic libraries.

BLOSC_EXPORT int blosc_get_complib_info(const char * compname, char ** complib, char ** ver

Get info from compression libraries included in the current build.

Parameters

- `compname`: The compressor name that you want info from.
- `complib`: The pointer to a string where the compression library name, if available, will be put.
- `version`: The pointer to a string where the compression library version, if available, will be put.

Warning You are in charge of the `complib` and `version` strings, you should free() them so as to avoid leaks.

Return The code for the compression library (>=0). If it is not supported, this function returns -1.

6.1.3 Context API

In Blosc 2 there is a special *blosc2_context* struct that is created from compression and decompression parameters. This allows the compression and decompression to happen in multithreaded scenarios, without the need for using the global lock.

struct blosc2_cparams

The parameters for creating a context for compression purposes.

In parenthesis it is shown the default value used internally when a 0 (zero) in the fields of the struct is passed to a function.

Public Members

uint8_t **compcode**
The compressor codec.

uint8_t **clevel**
The compression level (5).

int **use_dict**
Use dicts or not when compressing (only for ZSTD).

int32_t **typesize**
The type size (8).

int16_t **nthreads**
The number of threads to use internally (1).

int32_t **blocksize**
The requested size of the compressed blocks (0; meaning automatic).

void ***schunk**
The associated schunk, if any (NULL).

uint8_t **filters**[**BLOSC2_MAX_FILTERS**]
The (sequence of) filters.

uint8_t **filters_meta**[**BLOSC2_MAX_FILTERS**]
The metadata for filters.

blosc2_prefilter_fn **prefilter**
The prefilter function.

blosc2_prefilter_params ***pparams**
The prefilter parameters.

Warning: doxygenvariable: Cannot find variable "BLOSC_CPARAMS_DEFAULTS" in doxygen xml output for project "blosc2" from directory: ./doxygen/xml

struct blosc2_dparams

The parameters for creating a context for decompression purposes.

In parenthesis it is shown the default value used internally when a 0 (zero) in the fields of the struct is passed to a function.

Public Members

int **nthreads**

The number of threads to use internally (1).

void ***schunk**

The associated schunk, if any (NULL).

Warning: doxygenvariable: Cannot find variable "BLOSC_DPARAMS_DEFAULTS" in doxygen xml output for project "blosc2" from directory: ./doxygen/xml

BLOSC_EXPORT blosc2_context* blosc2_create_cctx(blosc2_cparams cparams)

Create a context for *_*ctx*() compression functions.

Parameters

- `cparams`: The *blosc2_cparams* struct with the compression parameters.

Return A pointer to the new context. NULL is returned if this fails.

BLOSC_EXPORT blosc2_context* blosc2_create_dctx(blosc2_dparams dparams)

Create a context for *_ctx() decompression functions.

Parameters

- `dparams`: The *blosc2_dparams* struct with the decompression parameters.

Return A pointer to the new context. NULL is returned if this fails.

BLOSC_EXPORT void blosc2_free_ctx(blosc2_context * context)

Free the resources associated with a context.

This function should always succeed and is valid for contexts meant for both compression and decompression.

Parameters

- `context`: The context to free.

**BLOSC_EXPORT int blosc2_compress_ctx(blosc2_context * context, size_t nbytes, const void *

Context interface to Blosc compression.

This does not require a call to #blosc_init and can be called from multithreaded applications without the global lock being used, so allowing Blosc be executed simultaneously in those scenarios.

Parameters

- `context`: A blosc2_context struct with the different compression params.
- `nbytes`: The number of bytes to be compressed from the `src` buffer.

- `src`: The buffer containing the data to be compressed.
- `dest`: The buffer where the compressed data will be put.
- `destsize`: The size in bytes of the `dest` buffer.

Return The number of bytes compressed. If `src` buffer cannot be compressed into `destsize`, the return value is zero and you should discard the contents of the `dest` buffer. A negative return value means that an internal error happened. It could happen that context is not meant for compression (which is stated in stderr). Otherwise, please report it back together with the buffer data causing this and compression settings.

BLOSC_EXPORT int blosc2_decompress_ctx(blosc2_context * context, const void * src, void * d

Context interface to Blosc decompression.

This does not require a call to #blosc_init and can be called from multithreaded applications without the global lock being used, so allowing Blosc be executed simultaneously in those scenarios.

Parameters

- `context`: The blosc2_context struct with the different compression params.
- `src`: The buffer of compressed data.
- `dest`: The buffer where the decompressed data will be put.
- `destsize`: The size in bytes of the `dest` buffer.

Warning The `src` buffer and the `dest` buffer can not overlap.

Remark Decompression is memory safe and guaranteed not to write the `dest` buffer more than what is specified in `destsize`.

Remark In case you want to keep under control the number of bytes read from source, you can call #blosc_cbuffer_sizes first to check the `nbytes` (i.e. the number of bytes to be read from `src` buffer by this function) in the compressed buffer.

Remark If #blosc2_set_maskout is called prior to this function, its `block_maskout` parameter will be honored for just *one single* shot; i.e. the maskout in context will be automatically reset to NULL, so mask won't be used next time (unless #blosc2_set_maskout is called again).

Return The number of bytes decompressed (i.e. the maskout blocks are not counted). If an error occurs, e.g. the compressed data is corrupted, `destsize` is not large enough or context is not meant for decompression, then 0 (zero) or a negative value will be returned instead.

BLOSC_EXPORT int blosc2_set_maskout(blosc2_context * ctx, bool * maskout, int nblocks)

Set a maskout so as to avoid decompressing specified blocks.

Parameters

- `ctx`: The decompression context to update.
- `maskout`: The boolean mask for the blocks where decompression is to be avoided.

Parameters

- `nblocks`: The number of blocks in maskout above.

Remark The maskout is valid for contexts *only* meant for decompressing a chunk via #blosc2_decompress_ctx. Once a call to #blosc2_decompress_ctx is done, this mask is reset so that next call to #blosc2_decompress_ctx will decompress the whole chunk.

Return If success, a 0 values is returned. An error is signaled with a negative int.

BLOSC_EXPORT int blosc2_getitem_ctx(blosc2_context * context, const void * src, int start,
Context interface counterpart for #blosc_getitem.

It uses similar parameters than the blosc_getitem() function plus a `context` parameter.

Return The number of bytes copied to `dest` or a negative value if some error happens.

6.1.4 Super-chunk API

This API describes the new Blosc 2 container, the super-chunk (or *schunk* for short), that is typically stored sparsely in-memory (see the *frames* section below for other storage methods, including on-disk ones).

typedef blosc2_schunk

struct blosc2_schunk
This struct is the standard container for Blosc 2 compressed data.

This is essentially a container for Blosc 1 chunks of compressed data, and it allows to overcome the 32-bit limitation in Blosc 1. Optionally, a *blosc2_frame* can be attached so as to store the compressed chunks contiguously.

Public Members

uint8_t **compcode**
The default compressor. Each chunk can override this.

uint8_t **clevel**
The compression level and other compress params.

int32_t **typesize**
The type size.

int32_t **blocksize**
The requested size of the compressed blocks (0; meaning automatic).

int32_t **chunksize**
Size of each chunk. 0 if not a fixed chunksize.

uint8_t **filters**[**BLOSC2_MAX_FILTERS**]
The (sequence of) filters. 8-bit per filter.

uint8_t **filters_meta**[**BLOSC2_MAX_FILTERS**]
Metadata for filters. 8-bit per meta-slot.

int32_t **nchunks**
Number of chunks in super-chunk.

int64_t **nbytes**
The data size + metadata size + header size (uncompressed).

int64_t **cbytes**
The data size + metadata size + header size (compressed).

uint8_t ****data**
Pointer to chunk data pointers buffer.

size_t **data_len**
Length of the chunk data pointers buffer.

blosc2_frame ***frame**
Pointer to frame used as store for chunks.

uint8_t* ctx; Context for the thread holder. NULL if not acquired.

blosc2_context ***cctx**
Context for compression.

blosc2_context ***dctx**
Context for decompression.

struct *blosc2_metalayer* ***metalayers**[16]
The array of metalayers.

int16_t **nmetalayers**
The number of metalayers in the frame.

BLOSC_EXPORT blosc2_schunk* blosc2_new_schunk(blosc2_cparams cparams, blosc2_dparams dparam
Create a new super-chunk.

Parameters

- `cparams`: The compression parameters.
- `dparams`: The decompression parameters.
- `frame`: The frame to be used. NULL if not needed.

Return The new super-chunk

BLOSC_EXPORT int blosc2_free_schunk(blosc2_schunk * schunk)
Release resources from a super-chunk.

Parameters

- `schunk`: The super-chunk to be freed.

Return 0 if succeeds.

BLOSC_EXPORT int blosc2_schunk_append_buffer(blosc2_schunk * schunk, void * src, size_t nb
Append a `src` data buffer to a super-chunk.

Parameters

- `schunk`: The super-chunk where data will be appended.
- `src`: The buffer of data to compress.
- `nbytes`: The size of the `src` buffer.

Return The number of chunks in super-chunk. If some problem is detected, this number will be negative.

BLOSC_EXPORT int blosc2_schunk_decompress_chunk(blosc2_schunk * schunk, int nchunk, void *
Decompress and return the `nchunk` chunk of a super-chunk.

If the chunk is uncompressed successfully, it is put in the `*dest` pointer.

Parameters

- `schunk`: The super-chunk from where the chunk will be decompressed.
- `nchunk`: The chunk to be decompressed (0 indexed).
- `dest`: The buffer where the decompressed data will be put.
- `nbytes`: The size of the area pointed by `*dest`.

Warning You must make sure that you have space enough to store the uncompressed data.

Return The size of the decompressed chunk. If some problem is detected, a negative code is returned instead.

BLOSC_EXPORT int blosc2_schunk_get_chunk(blosc2_schunk * schunk, int nchunk, uint8_t ** ch

Return a compressed chunk that is part of a super-chunk in the `chunk` parameter.

Parameters

- `schunk`: The super-chunk from where to extract a chunk.
- `nchunk`: The chunk to be extracted (0 indexed).
- `chunk`: The pointer to the chunk of compressed data.
- `needs_free`: The pointer to a boolean indicating if it is the user's responsibility to free the chunk returned or not.

Warning If the super-chunk is backed by a frame that is disk-based, a buffer is allocated for the (compressed) chunk, and hence a free is needed. You can check if the chunk requires a free with the `needs_free` parameter. If the chunk does not need a free, it means that a pointer to the location in the super-chunk (or the backing in-memory frame) is returned in the `chunk` parameter.

Return The size of the (compressed) chunk. If some problem is detected, a negative code is returned instead.

BLOSC_EXPORT int blosc2_schunk_get_cparams(blosc2_schunk * schunk, blosc2_cparams ** cpar

Return the `cparams` associated to a super-chunk.

Parameters

- `schunk`: The super-chunk from where to extract the compression parameters.
- `cparams`: The pointer where the compression params will be returned.

Warning A new struct is allocated, and the user should free it after use.

Return 0 if succeeds. Else a negative code is returned.

BLOSC_EXPORT int blosc2_schunk_get_dparams(blosc2_schunk * schunk, blosc2_dparams ** dpar

Return the `dparams` struct associated to a super-chunk.

Parameters

- `schunk`: The super-chunk from where to extract the decompression parameters.
- `dparams`: The pointer where the decompression params will be returned.

Warning A new struct is allocated, and the user should free it after use.

Return 0 if succeeds. Else a negative code is returned.

6.1.5 Frame API

The Blosc 2 Frame struct is essentially a store for a super-chunk that can be contiguous in memory, or serialized to disk.

struct blosc2_frame

Public Members

char ***fname**
The name of the file; if NULL, this is in-memory.

uint8_t ***sdata**
The in-memory serialized data.

uint8_t ***coffsets**
Pointers to the (compressed, on-disk) chunk offsets.

int64_t **len**
The current length of the frame in (compressed) bytes.

int64_t **maxlen**
The maximum length of the frame; if 0, there is no maximum.

uint32_t **trailer_len**
The current length of the trailer in (compressed) bytes.

BLOSC_EXPORT int64_t blosc2_schunk_to_frame(blosc2_schunk * schunk, blosc2_frame * frame)
Create a frame from a super-chunk.

If frame->fname is NULL, a frame is created in memory; else it is created on disk.

Parameters

- `schunk`: The super-chunk from where the frame will be created.
- `frame`: The pointer for the frame that will be populated.

Return The size in bytes of the frame. If an error occurs it returns a negative value.

BLOSC_EXPORT blosc2_schunk* blosc2_schunk_from_frame(blosc2_frame * frame, bool copy)
Create a super-chunk from a frame.

Parameters

- `frame`: The frame from which the super-chunk will be created.
- `copy`: If true, a new, sparse in-memory super-chunk is created. Else, a frame-backed one is created (i.e. no copies are made)

Return The super-chunk corresponding to the frame.

BLOSC_EXPORT int blosc2_free_frame(blosc2_frame * frame)
Free all memory from a frame.

Parameters

- `frame`: The frame to be freed.

Return 0 if succeeds.

BLOSC_EXPORT int64_t blosc2_frame_to_file(blosc2_frame * frame, const char * fname)
Write an in-memory frame out to a file.

The frame used must be an in-memory frame, i.e. frame->fname == NULL.

Parameters

- `frame`: The frame to be written into a file.
- `fname`: The name of the file.

Return The size of the frame. If negative, an error happened (including that the original frame is not in-memory).

BLOSC_EXPORT blosc2_frame* blosc2_frame_from_file(const char * fname)
Initialize a frame out of a file.

Parameters

- `fname`: The file name.

Return The frame created from the file.

6.1.6 Metalayer functions

Metalayers are meta-information that can be attached to super-chunks. They can also be serialized to disk.

typedef blosc2_metalayer

struct blosc2_metalayer
This struct is meant to store metadata information inside a *blosc2_schunk*, allowing to specify, for example, how to interpret the contents included in the schunk.

Public Members

char ***name**
The metalayer identifier for Blosc client (e.g. Caterva).

uint8_t ***content**
The serialized (msgpack preferably) content of the metalayer.

int32_t **content_len**
The length in bytes of the content.

BLOSC_EXPORT int blosc2_has_metalayer(blosc2_schunk * schunk, const char * name)
Find whether the schunk has a metalayer or not.

Parameters

- `schunk`: The super-chunk from which the metalayer will be checked.
- `name`: The name of the metalayer to be checked.

Return If successful, return the index of the metalayer. Else, return a negative value.

BLOSC_EXPORT int blosc2_add_metalayer(blosc2_schunk * schunk, const char * name, uint8_t
Add content into a new metalayer.

Parameters

- `schunk`: The super-chunk to which the metalayer should be added.
- `name`: The name of the metalayer.
- `content`: The content of the metalayer.
- `content_len`: The length of the content.

Return If successful, the index of the new metalayer. Else, return a negative value.

BLOSC_EXPORT int blosc2_update_metalayer(blosc2_schunk * schunk, const char * name, uint8_
Update the content of an existing metalayer.

Parameters

- `schunk`: The frame containing the metalayer.
- `name`: The name of the metalayer to be updated.
- `content`: The new content of the metalayer.
- `content_len`: The length of the content.

Note Contrarily to #blosc2_add_metalayer the updates to metalayers are automatically serialized into a possible attached frame.

Return If successful, the index of the metalayer. Else, return a negative value.

**BLOSC_EXPORT int blosc2_get_metalayer(blosc2_schunk * schunk, const char * name, uint8_t *

Get the content out of a metalayer.

Parameters

- `schunk`: The frame containing the metalayer.
- `name`: The name of the metalayer.
- `content`: The pointer where the content will be put.
- `content_len`: The length of the content.

Warning The `**content` receives a malloc'ed copy of the content. The user is responsible of freeing it.

Return If successful, the index of the new metalayer. Else, return a negative value.

6.1.7 Usermeta functions

Usermeta is a variable-sized chunk that is attached to a super-chunk and can also be serialized to disk.

BLOSC_EXPORT int blosc2_update_usermeta(blosc2_schunk * schunk, uint8_t * content, int32_t

Update content into a usermeta chunk.

If the `schunk` has an attached frame, the later will be updated accordingly too.

Parameters

- `schunk`: The super-chunk to which one should add the usermeta chunk.
- `content`: The content of the usermeta chunk.
- `content_len`: The length of the content.
- `cparams`: The parameters for compressing the usermeta chunk.

Note The previous content, if any, will be overwritten by the new content. The user is responsible to keep the new content in sync with any previous content.

Return If successful, return the number of compressed bytes that takes the content. Else, a negative value.

BLOSC_EXPORT int blosc2_get_usermeta(blosc2_schunk * schunk, uint8_t ** content)

6.1.8 Timing functions

Time measurement utilities.

BLOSC_EXPORT void blosc_set_timestamp(struct timespec * timestamp)

BLOSC_EXPORT double blosc_elapsed_nsecs(struct timespec start_time, struct timespec end_ti

BLOSC_EXPORT double blosc_elapsed_secs(struct timespec start_time, struct timespec end_time

6.1.9 Low level functions

Use them only if you are an expert!

BLOSC_EXPORT int blosc_get_blocksize(void)
Get the internal blocksize to be used during compression.

0 means that an automatic blocksize is computed internally.

Return The size in bytes of the internal block size.

BLOSC_EXPORT void blosc_set_blocksize(size_t blocksize)
Force the use of a specific blocksize.

If 0, an automatic blocksize will be used (the default).

Warning The blocksize is a critical parameter with important restrictions in the allowed values, so use this with care.

BLOSC_EXPORT void blosc_set_schunk(blosc2_schunk * schunk)
Set pointer to super-chunk.

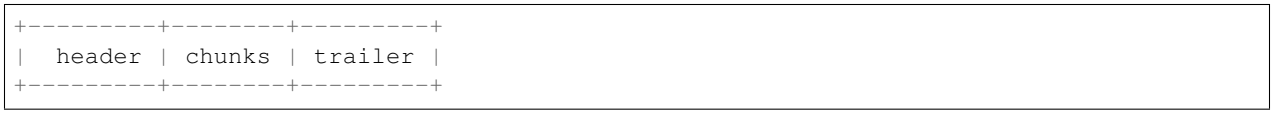
If NULL, no super-chunk will be available (the default).

6.2 Blosc Examples

6.3 Blosc2 Frame Format

Blosc (as of version 2.0.0) has a frame format that allows to store different data chunks sequentially, either in-memory or on-disk.

The frame is composed by a header, a chunks section and a trailer, which are variable-length and stored sequentially:



These are described below.

6.3.1 The header section

The header of a frame is encoded via msgpack and it follows the next format:

```
|-0-|-1-|-2-|-3-|-4-|-5-|-6-|-7-|-8-|-9-|-A-|-B-|-C-|-D-|-E-|-F-|-10|-11|-12|-13|-14|-
↪15|-16|-17|
| 9X| aX| "b2frame\0"                   | d2| header_size   | cf| frame_size
↪          |
|---|---|-------------------------------|---|---------------|---|----------------------
↪----------|
  ^   ^       ^                           ^                   ^
  |   |       |                           |                   |
  |   |       |                           |                   +--[msgpack] uint64
  |   |       |                           |
  |   |       |                           +--[msgpack] int32
  |   |       +---magic number, currently "b2frame"
  |   +------[msgpack] str with 8 elements
  +---[msgpack] fixarray with X=0xD (13) elements

|-18|-19|-1A|-1B|-1C|-1D|-1E|-1F|-20|-21|-22|-23|-24|-25|-26|-27|-28|-29|-2A|-2B|-2C|-
↪2D|-2E|
| a4|_f0|_f1|_f2|_f3| d3| uncompressed_size             | d3| compressed_size
↪       |
|---|---|---|---|---|---|-------------------------------|---|-------------------------
↪------|
  ^   ^   ^   ^   ^   ^                                   ^
  |   |   |   |   |   |                                   +--[msgpack] int64
  |   |   |   |   |   +--[msgpack] int64
  |   |   |   |   +-- reserved flags
  |   |   |   +--codec_flags (see below)
  |   |   +---reserved flags
  |   +------general_flags (see below)
  +---[msgpack] str with 4 elements (flags)

|-2F|-30|-31|-32|-33|-34|-35|-36|-37|-38|-39|-3A|-3B|-3C|-3D|-3E|-3F|
| d2| type_size     | d2| chunk_size    | d1| tcomp | d1|tdecomp| cX|
|---|---------------|---|---------------|---|-------|---|-------|---|
  ^                   ^                   ^     ^     ^     ^     ^
  |                   |                   |     |     |     |     +--[msgpack] bool
↪for has_usermeta
  |                   |                   |     |     |     +--number of threads for
↪decompression
  |                   |                   |     |     +-- [msgpack] int16
  |                   |                   |     +--number of threads for compression
  |                   |                   +---[msgpack] int16
  |                   +------[msgpack] int32
  +---[msgpack] int32
```

Then it follows the info about the filter pipeline. There is place for a pipeline that is 8 slots deep, and there is a reserved byte per every filter code and another byte for a possible associated meta-info:

```
|-40|-41|-42|-43|-44|-45|-46|-47|-48|-49|-4A|-4B|-4C|-4D|-4E|-4F|-50|-51|
| d2| X | filter_codes                  | filter_meta                   |
|---|---|-------------------------------|-------------------------------|
  ^   ^
  |   |
  |   +--number of filters
  +--[msgpack] fixext 16
```

In addition, a frame can be completed with meta-information about the stored data; these data blocks are called metalayers and it is up to the user to store whatever data they want there, with the only (strong) suggestion that they have to be in the msgpack format. Here it is the format for the case that there exist some metalayers:

```
|-52|-53|-54|-55|-56|-----------------------
| 93| cd| idx   | de| map_of_metalayers
|---|---------------|-----------------------
  ^   ^     ^       ^
  |   |     |       |
  |   |     |       +--[msgpack] map 16 with N keys
  |   |     +--size of the map (index) of offsets
  |   +--[msgpack] uint16
  +-- [msgpack] fixarray with 3 elements
```

map of metalayers This is a *msgpack-formattted* map for the different metalayers. The keys will be a string (0xa0 + namelen) for the names of the metalayers, followed by an int32 (0xd2) for the *offset* of the value of this metalayer. The actual value will be encoded as a bin32 (0xc6) value later in frame.

Description for different fields in header

header_size (`int32`) Size of the header of the frame (including metalayers).

frame_size (`uint64`) Size of the whole frame (including compressed data).

general_flags (`uint8`) General flags.

0 to 3 Format version

4 and 5 Enumerated for chunk offsets. `:0`:

16-bit

1 32-bit

2 64-bit

3 Reserved

6 Chunks of fixed length (0) or variable length (1)

7 Reserved

filter_flags (`uint8`) Filter flags that are the defaults for all the chunks in storage.

bit 0 If set, blocks are *not* split in sub-blocks.

bit 1 Filter pipeline is described in bits 3 to 6; else in *_filter_pipeline* system metalayer.

bit 2 Reserved

bit 3 Whether the shuffle filter has been applied or not.

bit 4 Whether the internal buffer is a pure memcpy or not.

bit 5 Whether the bitshuffle filter has been applied or not.

bit 6 Whether the delta codec has been applied or not.

bit 7 Reserved

codec_flags (`uint8`) Compressor enumeration (defaults for all the chunks in storage).

0 to 3 Enumerated for codecs (up to 16) `:0`:

`blosclz`

1 `lz4` or `lz4hc`

2 `snappy`

3 `zlib`

4 `zstd`

5 `lizard`

4 to 7 Compression level (up to 16)

reserved_flags (`uint8`) Space reserved.

uncompressed_size (`int64`) Size of uncompressed data in frame (excluding metadata).

compressed_size (`int64`) Size of compressed data in frame (excluding metadata).

type_size (`int32`) Size of each item.

chunk_size (`int32`) Size of each data chunk. 0 if not a fixed chunksize.

tcomp (`int16`) Number of threads for compression. If 0, same than *cctx*.

tdecomp (`int16`) Number of threads for decompression. If 0, same than *dctx*.

6.3.2 The chunks section

Here there is the actual data chunks stored sequentially:

```
+========+========+========+===========+
| chunk0 | chunk1 |   ...  | chunk idx |
+========+========+========+===========+
```

The different chunks are described in the chunk format document. The *chunk idx* is an index for the different chunks in this section. It is made by the 64-bit offsets to the different chunks and compressed into a new chunk, following the regular Blosc chunk format.

6.3.3 The trailer section

Here it is data that can change in size, mainly the *usermeta* chunk:

```
|-0-|-1-|-2-|-3-|-4-|-5-|-6-|====================|---|---------------|---|---
↪|=================|
| 9X| aX| c6| usermeta_len  |   usermeta_chunk    | ce| trailer_len   | d8|fpt|␣
↪fingerprint     |
|---|---|---|---------------|====================|---|---------------|---|---
↪|=================|
  ^   ^   ^       ^                                ^       ^           ^   ^
  |   |   |       |                                |       |           |   +--␣
↪fingerprint type
  |   |   |       |                                |       |           +--[msgpack]␣
↪fixext 16
  |   |   |       |                                |       +-- trailer length␣
↪(network endian)
  |   |   |       |                                +--[msgpack] uint32 for trailer␣
↪length
  |   |   |       +--[msgpack] usermeta length (network endian)
```

(continues on next page)

(continued from previous page)

```
|   |   +---[msgpack] bin32 for usermeta
|   +------[msgpack] int8 for trailer version
+---[msgpack] fixarray with X=4 elements
```

Description for different fields in trailer

usermeta_len (`int32`) The length of the usermeta chunk.

usermeta_chunk (`varlen`) The usermeta chunk (a Blosc chunk).

trailer_len (`uint32`) Size of the trailer of the frame (including usermeta chunk).

fpt (`int8`) Fingerprint type: 0 -> no fp; 1 -> 32-bit; 2 -> 64-bit; 3 -> 128-bit

fingerprint (`uint128`) Fix storage space for the fingerprint, padded to the left.

CHAPTER

SEVEN

BLOSC2 FRAME FORMAT

Blosc (as of version 2.0.0) has a frame format that allows to store different data chunks sequentially, either in-memory or on-disk.

The frame is composed by a header, a chunks section and a trailer, which are variable-length and stored sequentially:

```
+---------+--------+---------+
|  header | chunks | trailer |
+---------+--------+---------+
```

These are described below.

The header of a frame is encoded via msgpack and it follows the next format:

```
|-0-|-1-|-2-|-3-|-4-|-5-|-6-|-7-|-8-|-9-|-A-|-B-|-C-|-D-|-E-|-F-|-10|-11|-12|-13|-14|-
↪15|-16|-17|
| 9X| aX| "b2frame\0"                   | d2| header_size   | cf| frame_size          ⌴
↪          |
|---|---|-------------------------------|---|---------------|---|---------------------
↪----------|
  ^   ^       ^                           ^                   ^
  |   |       |                           |                   |
  |   |       |                           |                   +--[msgpack] uint64
  |   |       |                           |
  |   |       |                           +--[msgpack] int32
  |   |       +---magic number, currently "b2frame"
  |   +------[msgpack] str with 8 elements
  +---[msgpack] fixarray with X=0xD (13) elements

|-18|-19|-1A|-1B|-1C|-1D|-1E|-1F|-20|-21|-22|-23|-24|-25|-26|-27|-28|-29|-2A|-2B|-2C|-
↪2D|-2E|
| a4|_f0|_f1|_f2|_f3| d3| uncompressed_size             | d3| compressed_size        ⌴
↪      |
|---|---|---|---|---|---|-------------------------------|---|-------------------------
↪------|
  ^   ^   ^   ^   ^   ^                                   ^
  |   |   |   |   |   |                                   +--[msgpack] int64
  |   |   |   |   |   +--[msgpack] int64
  |   |   |   |   +-- reserved flags
  |   |   |   +--codec_flags (see below)
  |   |   +---reserved flags
  |   +------general_flags (see below)
  +---[msgpack] str with 4 elements (flags)

|-2F|-30|-31|-32|-33|-34|-35|-36|-37|-38|-39|-3A|-3B|-3C|-3D|-3E|-3F|
```

(continues on next page)

(continued from previous page)

```
| d2| type_size     | d2| chunk_size    | d1| tcomp | d1|tdecomp| cX|
|---|---------------|---|---------------|---|-------|---|-------|---|
  ^                   ^                   ^     ^     ^     ^     ^
  |                   |                   |     |     |     |     +--[msgpack] bool␣
↪for has_usermeta
  |                   |                   |     |     |     +--number of threads for␣
↪decompression
  |                   |                   |     |     +-- [msgpack] int16
  |                   |                   |     +--number of threads for compression
  |                   |                   +---[msgpack] int16
  |                   +------[msgpack] int32
  +---[msgpack] int32
```

Then it follows the info about the filter pipeline. There is place for a pipeline that is 8 slots deep, and there is a reserved byte per every filter code and another byte for a possible associated meta-info:

```
|-40|-41|-42|-43|-44|-45|-46|-47|-48|-49|-4A|-4B|-4C|-4D|-4E|-4F|-50|-51|
| d2| X | filter_codes                   | filter_meta                     |
|---|---|--------------------------------|---------------------------------|
  ^   ^
  |   |
  |   +--number of filters
  +--[msgpack] fixext 16
```

In addition, a frame can be completed with meta-information about the stored data; these data blocks are called metalayers and it is up to the user to store whatever data they want there, with the only (strong) suggestion that they have to be in the msgpack format. Here it is the format for the case that there exist some metalayers:

```
|-52|-53|-54|-55|-56|-----------------------
| 93| cd| idx   | de| map_of_metalayers
|---|---------------|-----------------------
  ^   ^     ^       ^
  |   |     |       |
  |   |     |       +--[msgpack] map 16 with N keys
  |   |     +--size of the map (index) of offsets
  |   +--[msgpack] uint16
  +-- [msgpack] fixarray with 3 elements
```

map of metalayers This is a *msgpack-formattted* map for the different metalayers. The keys will be a string (0xa0 + namelen) for the names of the metalayers, followed by an int32 (0xd2) for the *offset* of the value of this metalayer. The actual value will be encoded as a bin32 (0xc6) value later in frame.

header_size (`int32`) Size of the header of the frame (including metalayers).

frame_size (`uint64`) Size of the whole frame (including compressed data).

general_flags (`uint8`) General flags.

0 to 3 Format version

4 and 5 Enumerated for chunk offsets. :0:

16-bit

1 32-bit

2 64-bit

3 Reserved

6 Chunks of fixed length (0) or variable length (1)

7 Reserved

filter_flags (`uint8`) Filter flags that are the defaults for all the chunks in storage.

bit 0 If set, blocks are *not* split in sub-blocks.

bit 1 Filter pipeline is described in bits 3 to 6; else in *_filter_pipeline* system metalayer.

bit 2 Reserved

bit 3 Whether the shuffle filter has been applied or not.

bit 4 Whether the internal buffer is a pure memcpy or not.

bit 5 Whether the bitshuffle filter has been applied or not.

bit 6 Whether the delta codec has been applied or not.

bit 7 Reserved

codec_flags (`uint8`) Compressor enumeration (defaults for all the chunks in storage).

0 to 3 Enumerated for codecs (up to 16) :`0`:

`blosclz`

1 `lz4` or `lz4hc`

2 `snappy`

3 `zlib`

4 `zstd`

5 `lizard`

4 to 7 Compression level (up to 16)

reserved_flags (`uint8`) Space reserved.

uncompressed_size (`int64`) Size of uncompressed data in frame (excluding metadata).

compressed_size (`int64`) Size of compressed data in frame (excluding metadata).

type_size (`int32`) Size of each item.

chunk_size (`int32`) Size of each data chunk. 0 if not a fixed chunksize.

tcomp (`int16`) Number of threads for compression. If 0, same than *cctx*.

tdecomp (`int16`) Number of threads for decompression. If 0, same than *dctx*.

Here there is the actual data chunks stored sequentially:

```
+========+========+========+===========+
| chunk0 | chunk1 |   ...  | chunk idx |
+========+========+========+===========+
```

The different chunks are described in the chunk format document. The *chunk idx* is an index for the different chunks in this section. It is made by the 64-bit offsets to the different chunks and compressed into a new chunk, following the regular Blosc chunk format.

Here it is data that can change in size, mainly the *usermeta* chunk:

```
|-0-|-1-|-2-|-3-|-4-|-5-|-6-|====================|---|---------------|---|---
↪|==================|
| 9X| aX| c6| usermeta_len  |   usermeta_chunk    | ce| trailer_len   | d8|fpt|␣
↪fingerprint     |
|---|---|---|---------------|====================|---|---------------|---|---
↪|==================|
  ^   ^   ^       ^                                  ^       ^           ^   ^
  |   |   |       |                                  |       |           |   +--␣
↪fingerprint type
  |   |   |       |                                  |       |           +--[msgpack]␣
↪fixext 16
  |   |   |       |                                  |       +-- trailer length␣
↪(network endian)
  |   |   |       |                                  +--[msgpack] uint32 for trailer␣
↪length
  |   |   |       +--[msgpack] usermeta length (network endian)
  |   |   +---[msgpack] bin32 for usermeta
  |   +------[msgpack] int8 for trailer version
  +---[msgpack] fixarray with X=4 elements
```

usermeta_len (`int32`) The length of the usermeta chunk.

usermeta_chunk (`varlen`) The usermeta chunk (a Blosc chunk).

trailer_len (`uint32`) Size of the trailer of the frame (including usermeta chunk).

fpt (`int8`) Fingerprint type: 0 -> no fp; 1 -> 32-bit; 2 -> 64-bit; 3 -> 128-bit

fingerprint (`uint128`) Fix storage space for the fingerprint, padded to the left.

INDEX

B