CATERVA
A Compressed And Multidimensional Container
For Not So Big Data

@FrancescAlted
Freelancer

Sponsored by NUMFOCUS
A Member of the Blosc Family

HDF5 European Workshop. Grenoble September 18th, 2019
About Me

• Physicist by training.

• Computer scientist by passion.

• Open Source enthusiast by philosophy.
  • PyTables (2002 - 2011)
  • Blosc (2009 - now)
The technology platform to make a difference in your relationship with large and complex data

HDF5 + a Twist

Out-of-core Expressions

Computing $0.25x^3 + 0.75x^2 - 1.5x - 2$ polynomial

Indexed Queries

Query time for complex query and 1 Grow (indexed)

Sizes for index of a 1 Grow column with different optimizations (PyTables Pro 2.1 beta2 vs PostgreSQL 8.2.6)
What is Caterva?

- It is an open source **C library and a format** that allows to store large multidimensional, chunked, compressed datasets.

- Data can be stored either **in-memory or on-disk**, but the API to handle both versions is the same.

- Compression is handled transparently for the user by adopting the **Blosc2 library**.
Why Another Data Container?

• Most of the existing data containers supporting on-the-flight compression are meant for on-disk/cloud data.

• But the memory layer can be seen as storage too, and there is a need for a container that is optimized for this.

• Caterva is designed from the ground up to use the memory layer as storage for a compressed data-container.
Accelerating I/O With Caterva

Mechanical Disk

Solid State Disk

Main Memory

Level 3 Cache

Level 2 Cache

Level 1 Cache

CPU

Other Containers

Caterva

Capacity

Speed
Why Another Format?

• 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.

• As we will see soon, Caterva brings an efficient and open format for persistency.
Caterva Brings Powerful Slicing Capabilities

• Caterva’s main feature is to be able to extract all kind of slices out of high dimensional datasets, efficiently.

• Resulting slices can be either Caterva containers or regular plain buffers (for better interaction with e.g. NumPy).
Accessing Chunked Datasets

- Those used to manipulate chunked multidimensional arrays know how critical choosing the partition size is.

You can play with a small, but representative benchmark at:

https://github.com/Blosc/cat4py/blob/master/notebooks/compare_getslice.ipynb
Performance In-Memory

Caterva is meant to read data from memory very fast!
Performance On-Disk

There is still room for optimization when reading from disk...
#include <caterva.h>

int main(){
  // Create a context
  caterva_ctx_t *ctx = caterva_new_ctx(NULL, NULL, BLOSC2_CPARAMS_DEFAULTS, BLOSC2_DPARAMS_DEFAULTS);
  ctx->cparams.typesize = sizeof(double);

  // Define the partition shape for the first array
  int8_t ndim = 3;
  int64_t pshape[] = {3, 2, 4};
  caterva_dims_t pshape = caterva_new_dims(pshape_, ndim);

  // Create the first (empty) array
  caterva_array_t *cat1 = caterva_empty_array(ctx, NULL, &pshape);

  // Define a buffer shape to fill cat1
  int64_t shape[] = {10, 10, 10};
  caterva_dims_t shape = caterva_new_dims(shape_, ndim);

  // Create a buffer to fill cat1
  size_t buf1size = 10 * 10 * 10 * sizeof(double);
  double *buf1 = (double *) malloc(buf1size * sizeof(double));

  // Fill cat1 with the above buffer
  caterva_from_buffer(cat1, &shape, buf1);

  free(buf1);
  caterva_free_array(cat1);

  return 0;
}

Example of muti-dimensional array creation
// Apply a `get_slice` to cat1 and store it into cat2
int64_t start__[] = {3, 6, 4};
caterva_dims_t start = caterva_new_dims(start__, ndim);
int64_t stop__[] = {4, 9, 8};
caterva_dims_t stop = caterva_new_dims(stop__, ndim);
int64_t pshape2__[] = {1, 2, 3};
caterva_dims_t pshape2 = caterva_new_dims(pshape2__, ndim);
caterva_array_t *cat2 = caterva_empty_array(ctx, NULL, &pshape2);

caterva_get_slice(cat2, cat1, &start, &stop);
caterva_squeeze(cat2);

// Create a buffer to store the cat2 elements
uint64_t buf2size = 1;
caterva_dims_t shape2 = caterva_get_shape(cat2);
for (int j = 0; j < shape2.ndim; ++j) {
    buf2size *= shape2.dims[j];
}
double *buf2 = (double *) malloc(buf2size * sizeof(double));

// Fill buffer with the cat2 content
caterva_to_buffer(cat2, buf2);

printf("The resulting hyperplane is:\n");
for (int64_t i = 0; i < shape2.dims[0]; ++i) {
    for (int64_t j = 0; j < shape2.dims[1]; ++j) {
        printf("%6.f", buf2[i * cat2->shape[1] + j]);
    }
    printf("\n");
}
### Brief Comparison Against Well Known Chunked Containers

<table>
<thead>
<tr>
<th>Feature</th>
<th>HDF5</th>
<th>Zarr</th>
<th>Caterva</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>One-file per container?</strong></td>
<td>Yes (&gt; 1 container)</td>
<td>No (1 file per chunk)</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Hierarchical</strong></td>
<td>Yes</td>
<td>Yes</td>
<td>No (use the filesystem)</td>
</tr>
<tr>
<td><strong>Mature</strong></td>
<td>Yes</td>
<td>Yes</td>
<td>In process</td>
</tr>
<tr>
<td><strong>In-memory version?</strong></td>
<td>Yes (sequential?)</td>
<td>Yes (sparse)</td>
<td>Yes (sequential / sparse)</td>
</tr>
</tbody>
</table>
Blosc2

• Blosc2 is the next generation of the well-known Blosc (aka Blosc1).

• New features:
  • Enlargeable 64-bit containers: in-memory or on-disk
  • New compression codecs
  • New filters
  • Metalayers
  • User metadata
Decompression Speed

- Blosc (cname='zstd', clevel=5, shuffle=2): 61.1X
- Blosc (cname='zstd', clevel=1, shuffle=2): 50.2X
- Blosc (cname='zlib', clevel=5, shuffle=2): 37.6X
- Blosc (cname='lz4hc', clevel=5, shuffle=2): 41.8X
- Blosc (cname='lz4hc', clevel=1, shuffle=2): 35.0X
- Blosc (cname='lz4', clevel=9, shuffle=2): 32.0X
- Blosc (cname='lz4', clevel=5, shuffle=2): 31.0X
- Blosc (cname='lz4', clevel=1, shuffle=2): 29.1X
- Blosc (cname='lz4', clevel=9, shuffle=2): 30.6X
- Blosc (cname='lz4', clevel=5, shuffle=0): 30.7X
- Blosc (cname='lz4', clevel=1, shuffle=0): 28.3X
- Blosc (cname='snappy', clevel=9, shuffle=2): 47.4X
- Blosc (cname='zstd', clevel=5, shuffle=0): 42.2X
- Blosc (cname='zstd', clevel=1, shuffle=0): 38.7X
- Blosc (cname='zlib', clevel=5, shuffle=0): 28.3X
- Blosc (cname='lz4hc', clevel=5, shuffle=0): 20.5X
- Blosc (cname='lz4hc', clevel=9, shuffle=0): 18.9X
- Blosc (cname='lz4', clevel=5, shuffle=0): 18.1X
- Blosc (cname='lz4', clevel=9, shuffle=0): 13.4X
- Blosc (cname='lz4', clevel=0, shuffle=0): 48.7X
- LZMA (format=1, check=1, preset=1, filters=None): 48.7X
- BZ2 (level=1): 70.9X
- Zlib (level=1): 28.0X
- None: 1.0X

Containers in Blosc2

Super-chunk

- Codec
- Filter pipeline
- MetaLayers
- Pointers
- UserMeta

- Chunk 0
- Chunk 1
- Chunk 2

Frame

- Codec
- Filter pipeline
- MetaLayers
- Chunk 0
- Chunk 1
- Chunk 2
- Chunk Index
- UserMeta
- Fingerprint

- Header
- Trailer

- Sparse
- In-memory

- Sequential
- In-memory / On-disk
MetaLayers in Blosc2

- Metalayers 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: multi-dimensions, data types, geo-spatial...
Multiple layers to target different data aspects
Caterva MetaLayer

Caterva specifies a metalayer on top of a Blosc2 container for storing multidimensional information:

```c
typedef struct {
    int8_t ndim;
    //!< The number of dimensions
    uint64_t dims[CATERVA_MAXDIM];
    //!< The size of each dimension
    int32_t pdims[CATERVA_MAXDIM];
    //!< The size of each partition dimension
} caterva_dims_t;
```

This metalayer can be modified so that the shapes can be updated (e.g. an array can **grow or shrink**).
Why Caterva is Type Agnostic?

- There are too many data type systems floating around.
- Multi-dimensionality is orthogonal to the data type.
- This is why we decided not to make the type part of Caterva.
- The interested parties can always define a metalayer for endowing the desired type system to the data.

Example: add a metalayer for specifying the data type
https://github.com/Blosc/cat4py/blob/master/notebooks/array-metalayer.ipynb
Frame Format and MetaLayers Specs

- The format for a Blosc2 frame is completely specified at:

- The format for a Caterva metalayer:
  - https://github.com/Blosc/Caterva/blob/master/README_CATERVA_METALAYER.rst

Everything specified in the msgpack format.
Where Caterva Can Help?

• Whenever there is a need to deal with multidimensional datasets as fast as possible.

• Provide a backend for other packages (bcolz? zarr?).

  • Caterva is written in portable C99, so no limitations to be wrapped from other languages than e.g. Python.

• Allow to create different metalayers that adapt to user’s needs.
Where You Can Help?

- Blosc2, Caterva and cat4py (Caterva’s Python wrapper), are all open source, so you can always contribute with ideas and code.

- If you like the concepts behind the Blosc project as a whole, and you don’t have time to contribute with code, please donate to:
Overview

• Caterva is a **C library and a format** for handling multidimensional data **on top of Blosc2 containers**.

• The main goal is to efficiently **leverage fast storage** like memory, persistent memory (Intel Optane) or SSDs.

• You can use **metalayers for adapting** Caterva containers to your own needs.

https://github.com/Blosc/caterva
https://github.com/Blosc/c-blosc2
Acknowledgements

• First and foremost to Aleix Alcacer who contributed most of the code behind Caterva.

• Christian Steiner, for suggestions and improvements on Blosc2 / Caterva projects.

• Pepe Aracil, for his proposal for using msgpack for serializing Blosc2 containers.

• Last but not least, NumFOCUS for providing funding for developing Blosc2 and Caterva.
Thank You!

Questions?