bcolz Documentation

Release 1.1.2

Francesc Alted

February 10, 2017

Contents

Contents:

CHAPTER 1

Introduction

1.1 bcolz at glance

bcolz provides columnar, chunked data containers that can be compressed either in-memory and on-disk. Column storage allows for efficiently querying tables, as well as for cheap column addition and removal. It is based on NumPy, and uses it as the standard data container to communicate with bcolz objects, but it also comes with support for import/export facilities to/from HDF5/PyTables tables and pandas dataframes.

The building blocks of bcolz objects are the so-called `chunks` that are bits of data compressed as a whole, but that can be (partially) decompressed in order to improve the fetching of small parts of the array. This `chunked` nature of the bcolz objects, together with a buffered I/O, makes appends very cheap and fetches reasonably fast (although the modification of values can be an expensive operation).

The compression/decompression process is carried out internally by Blosc, a high-performance compressor that is optimized for binary data. The fact that Blosc splits chunks internally in so-called blocks means that only the interesting part of the chunk will decompressed (typically in L1 or L2 caches). That ensures maximum performance for I/O operation (either on-disk or in memory).

bcolz can use numexpr or dask internally (numexpr is used by default if installed, then dask and if these are not found, then the pure Python interpreter) so as to accelerate many internal vector and query operations (although it can use pure NumPy for doing so too). numexpr can optimize memory (cache) usage and uses multithreading for doing the computations, so it is blazing fast. This, in combination with carray/ctable disk-based, compressed containers, can be used for performing out-of-core computations efficiently, but most importantly *transparently*.

1.1.1 carray and ctable objects

The main data container objects in the bcolz package are:

- *carray*: container for homogeneous & heterogeneous (row-wise) data
- *ctable*: container for heterogeneous (column-wise) data

carray is very similar to a NumPy *ndarray* in that it supports the same types and basic data access interface. The main difference between them is that a *carray* can keep data compressed (both in-memory and on-disk), allowing to deal with larger datasets with the same amount of memory/disk. And another important difference is the chunked nature of the *carray* that allows data to be appended much more efficiently.

On his hand, a *ctable* is also similar to a NumPy `structured array` that shares the same properties with its *carray* brother, namely, compression and chunking. Another difference is that data is stored in a column-wise order (and not on a row-wise, like the `structured array`), allowing for very cheap column handling. This is of paramount importance when you need to add and remove columns in wide (and possibly large) in-memory and on-disk tables –doing this with regular `structured arrays` in NumPy is exceedingly slow.

Furthermore, columnar means that the tabular datasets are stored column-wise order, and this turns out to offer better opportunities to improve compression ratio. This is because data tends to expose more similarity in elements that sit in the same column rather than those in the same row, so compressors generally do a much better job when data is aligned in such column-wise order.

1.1.2 bcolz main features

bcolz objects bring several advantages over plain NumPy objects:

- Data is compressed: they take less storage space.
- Efficient shrinks and appends: you can shrink or append more data at the end of the objects very efficiently (i.e. copies of the whole array are not needed).
- Persistence comes seamlessly integrated, so you can work with on-disk arrays almost in the same way than with in-memory ones (bar some special attention to flush data being required).
- *ctable* objects have the data arranged column-wise. This allows for much better performance when working with big tables, as well as for improving the compression ratio.
- Can leverage Numexpr and Dask as virtual machines for fast operation with bcolz objects. Blosc ensures that the additional overhead of handling compressed data natively is very low.
- Advanced query capabilities. The ability of a *ctable* object to iterate over the rows whose fields fulfill some conditions (and evaluated via numexpr, dask or pure python virtual machine) allows to perform queries very efficiently.

1.1.3 bcolz limitations

bcolz does not currently come with good support in the next areas:

- Limited number of operations, at least when compared with NumPy. The supported operations are basically vectorized ones (i.e. those that are made element-by-element). But with is changing with the adoption of additional kernels like Dask (and more to come).
- Limited broadcast support. For example, NumPy lets you operate seamlessly with arrays of different shape (as long as they are compatible), but you cannot do that with bcolz. The only object that can be broadcasted currently are scalars (e.g. `bcolz.eval("x+3")`).
- Some methods (namely *carray.where()* and *carray.wheretrue()*) do not have support for multidimensional arrays.
- Multidimensional *ctable* objects are not supported. However, as the columns of these objects can be fully multidimensional, this is not regarded as an important limitation.

Installation

bcolz depends on NumPy and, optionally, Numexpr. Also, if you are going to install from sources, and a C compiler (Clang, GCC and MSVC 2008 for Python 2, and MSVC 2010 for Python 3, have been tested).

2.1 Installing from PyPI repository

Do:

```
$ easy_install -U bcolz
```

or:

```
$ pip install -U bcolz
```

2.2 Installing Windows binaries

Unofficial Windows binaries are provided by Christoph Gohlke and can be downloaded from:

http://www.lfd.uci.edu/~gohlke/pythonlibs/#bcolz

2.3 Using the Microsoft Python 2.7 Compiler

As of Sept 2014 Microsoft has made a Visual C++ compiler for Python 2.7 available for download:

http://aka.ms/vcpython27

This has been made available specifically to ease the handling of Python packages with C-extensions on Windows (installation and building wheels).

It is possible to compile bcolz with this compiler (Jan 2015), however, you may need to use the following patch:

```
diff --git i/setup.py w/setup.py
index d77d37f233..b54bfd0fa1 100644
--- i/setup.py
+++ w/setup.py
@@ -11,8 +11,8 @@ from __future__ import absolute_import
 import sys
 import os
 import glob
```

```
-from distutils.core import Extension
-from distutils.core import setup
+from setuptools import Extension
+from setuptools import setup
 import textwrap
 import re, platform
```

2.4 Installing from tarball sources

Go to the bcolz main directory and do the typical distutils dance:

```
$ python setup.py build_ext --inplace
```

In case you have Blosc installed as an external library you can link with it (disregarding the included Blosc sources) in a couple of ways:

Using an environment variable:

```
$ BLOSC_DIR=/usr/local     (or "set BLOSC_DIR=\blosc" on Win)
$ export BLOSC_DIR         (not needed on Win)
$ python setup.py build_ext --inplace
```

Using a flag:

```
$ python setup.py build_ext --inplace --blosc=/usr/local
```

It is always nice to run the tests before installing the package:

```
$ PYTHONPATH=.   (or "set PYTHONPATH=." on Windows)
$ export PYTHONPATH    (not needed on Windows)
$ python -c"import bcolz; bcolz.test()"  # add `heavy=True` if desired
```

And if everything runs fine, then install it via:

```
$ python setup.py install
```

2.5 Testing the installation

You can always test the installation from any directory with:

```
$ python -c "import bcolz; bcolz.test()"
```

CHAPTER 3

Tutorials

This section has been moved to ipython notebook tutorials.

3.1 Tutorial on carray objects

This section has been moved to ipython notebook tutorial_carray.

3.2 Tutorial on ctable objects

This section has been moved to ipython notebook tutorial_ctable.

3.3 Writing bcolz extensions

Did you like bcolz but you couldn't find exactly the functionality you were looking for? You can write an extension and implement complex operations on top of bcolz containers.

Before you start writing your own extension, let's see some examples of real projects made on top of bcolz:

- ***Bquery*: a query and aggregation framework, among other things it** provides group-by functionality for bcolz containers. See https://github.com/visualfabriq/bquery
- ***Bdot*: provides big dot products (by making your RAM bigger on the** inside). Supports `matrix . vector` and `matrix . matrix` for most common numpy numeric data types. See https://github.com/tailwind/bdot

Though not a extension itself, it is worth mentioning *Dask*. Dask plays nicely with bcolz and provides multi-core execution on larger-than-memory datasets using blocked algorithms and task scheduling. See https://github.com/dask/dask.

In addition, bcolz also interacts well with *itertools*, *Pytoolz* or *Cytoolz* too and they might offer you already the amount of performance and functionality you are after.

In the next section we will go through all the steps needed to write your own extension on top of bcolz.

3.3.1 How to use bcolz as part of the infrastructure

Go to the root directory of bcolz, inside `docs/my_package/` you will find a small extension example.

Before you can run this example you will need to install the following packages. Run `pip install cython`, `pip install numpy` and `pip install bcolz` to install these packages. In case you prefer Conda package management system execute `conda install cython numpy bcolz` and you should be ready to go. See `requirements.txt`:

```
cython>=0.20
numpy>=1.7.0
bcolz>=0.8.0
```

Once you have those packages installed, change your working directory to `docs/my_package/`, please see pkg. example and run `python setup.py build_ext --inplace` from the terminal, if everything ran smoothly you should be able to see a binary file `my_extension/example_ext.so` next to the `.pyx` file.

If you have any problems compiling these extensions, please make sure you have a recent version of bcolz as old versions (pre 0.8) don't contain the necessary `.pxd` file which provides a Cython interface to the carray Cython module.

The `setup.py` file is where you will need to tell the compiler, the name of you package, the location of external libraries (in case you want to use them), compiler directives and so on. See bcolz setup.py as a possible reference for a more complete example. Along your project grows in complexity you might be interested in including other options to your *Extension* object, e.g. *include_dirs* to include a list of directories to search for C/C++ header files your code might be dependent on.

See `my_package/setup.py`:

```
from setuptools import setup, Extension
from Cython.Distutils import build_ext
from numpy.distutils.misc_util import get_numpy_include_dirs


# Sources
sources = ["my_extension/example_ext.pyx"]

setup(
    name="my_package",
    description='My description',
    license='MY_LICENSE',
    ext_modules=[
        Extension(
            "my_extension.example_ext",
            sources=sources,
        ),
    ],
    cmdclass={"build_ext": build_ext},
    packages=['my_extension'],
)
```

The `.pyx` files is going to be the place where Cython code implementing the extension will be, in the example below the function will return a sum of all integers inside the carray.

See `my_package/my_extension/example_ext.pyx`

Keep in mind that carrays are great for sequential access, but random access will highly likely trigger decompression of a different chunk for each randomly accessed value.

For more information about Cython visit http://docs.cython.org/index.html

```
import cython
import bcolz as bz
from bcolz.carray_ext cimport carray
from numpy cimport ndarray, npy_int64
```

```
@cython.overflowcheck(True)
@cython.boundscheck(False)
@cython.wraparound(False)
cpdef my_function(carray ca):
    """
        Function for example purposes

        >>> import bcolz as bz
        >>> import my_extension.example_ext as my_mod
        >>> c = bz.carray([i for i in range(1000)], dtype='i8')
        >>> my_mod.my_function(c)
        499500

    """

    cdef:
        ndarray ca_segment
        Py_ssize_t len_ca_segment
        npy_int64 sum=0

    for ca_segment in bz.iterblocks(ca):
        len_ca_segment = len(ca_segment)
        for i in range(len_ca_segment):
            sum = sum + ca_segment[i]

    return sum
```

Let's test our extension:

```
>>> import bcolz
>>> import my_extension.example_ext as my_mod
>>> c = bcolz.carray([i for i in range(1000)], dtype='i8')
>>> my_mod.my_function(c)
499500
```

Library Reference

4.1 First level variables

`bcolz.__version__`
The version of the bcolz package.

`bcolz.dask_here`
Whether the minimum version of dask has been detected.

`bcolz.min_dask_version`
The minimum version of dask needed (dask is optional).

`bcolz.min_numexpr_version`
The minimum version of numexpr needed (numexpr is optional).

`bcolz.ncores`
The number of cores detected.

`bcolz.numexpr_here`
Whether the minimum version of numexpr has been detected.

4.2 Top level classes

class `bcolz.cparams`(*clevel=None*, *shuffle=None*, *cname=None*, *quantize=None*)
Class to host parameters for compression and other filters.

Parameters **clevel** : int (0 <= clevel < 10)

The compression level.

shuffle : int

The shuffle filter to be activated. Allowed values are bcolz.NOSHUFFLE (0), bcolz.SHUFFLE (1) and bcolz.BITSHUFFLE (2). The default is bcolz.SHUFFLE.

cname : string ('blosclz', 'lz4', 'lz4hc', 'snappy', 'zlib')

Select the compressor to use inside Blosc.

quantize : int (number of significant digits)

Quantize data to improve (lossy) compression. Data is quantized using np.around(scale*data)/scale, where scale is 2**bits, and bits is determined from the quantize value. For example, if quantize=1, bits will be 4. 0 means that the quantization is disabled.

In case some of the parameters are not passed, they will be

set to a default (see 'setdefaults()' method).

See also:

`cparams.setdefaults`

Attributes

`clevel`	The compression level.
`cname`	The compressor name.
`quantize`	Quantize filter.
`shuffle`	Shuffle filter.

Methods

`setdefaults`([clevel, shuffle, cname, quantize])	Change the defaults for compression params.

static **`setdefaults`**(*clevel=None*, *shuffle=None*, *cname=None*, *quantize=None*)

Change the defaults for compression params.

Parameters **clevel** : int (0 <= clevel < 10)

The compression level.

shuffle : int

The shuffle filter to be activated. Allowed values are bcolz.NOSHUFFLE (0), bcolz.SHUFFLE (1) and bcolz.BITSHUFFLE (2). The default is bcolz.SHUFFLE.

cname : string ('blosclz', 'lz4', 'lz4hc', 'snappy', 'zlib')

Select the compressor to use inside Blosc.

quantize : int (number of significant digits)

Quantize data to improve (lossy) compression. Data is quantized using np.around(scale*data)/scale, where scale is 2**bits, and bits is determined from the quantize value. For example, if quantize=1, bits will be 4. 0 means that the quantization is disabled.

If this method is not called, the defaults will be set as in

defaults.py:

(``{clevel=5, shuffle=bcolz.SHUFFLE, cname='lz4', quantize=None}``).

class `bcolz.attrs.`**`attrs`**(*rootdir*, *mode*, *_new=False*)

Accessor for attributes in carray/ctable objects.

This class behaves very similarly to a dictionary, and attributes can be appended in the typical way:

```
attrs['myattr'] = value
```

And can be retrieved similarly:

```
value = attrs['myattr']
```

Attributes can be removed with:

```
del attrs['myattr']
```

This class also honors the *__iter__* and *__len__* special functions. Moreover, a *getall()* method returns all the attributes as a dictionary.

CAVEAT: The values should be able to be serialized with JSON for persistence.

Methods

getall

Also, see the `carray` and `ctable` classes below.

4.3 Top level functions

`bcolz.arange`([*start*], *stop*[, *step*], *dtype=None*, ***kwargs*)

Return evenly spaced values within a given interval.

Values are generated within the half-open interval `[start, stop)` (in other words, the interval including *start* but excluding *stop*). For integer arguments the function is equivalent to the Python built-in range function, but returns a carray rather than a list.

Parameters **start** : number, optional

Start of interval. The interval includes this value. The default start value is 0.

stop : number

End of interval. The interval does not include this value.

step : number, optional

Spacing between values. For any output *out*, this is the distance between two adjacent values, `out[i+1] - out[i]`. The default step size is 1. If *step* is specified, *start* must also be given.

dtype : dtype

The type of the output array. If *dtype* is not given, infer the data type from the other input arguments.

kwargs : list of parameters or dictionary

Any parameter supported by the carray constructor.

Returns **out** : carray

Bcolz object made of evenly spaced values.

For floating point arguments, the length of the result is `ceil((stop - start)/step)`. Because of floating point overflow, this rule may result in the last element of *out* being greater than *stop*.

`bcolz.eval`(*expression*, *vm=None*, *out_flavor=None*, *user_dict=None*, *blen=None*, ***kwargs*)

Evaluate an *expression* and return the result.

Parameters **expression** : string

A string forming an expression, like '2*a+3*b'. The values for 'a' and 'b' are variable names to be taken from the calling function's frame. These variables may be scalars, carrays or NumPy arrays.

vm : string

The virtual machine to be used in computations. It can be 'numexpr', 'python' or 'dask'. The default is to use 'numexpr' if it is installed.

out_flavor : string

The flavor for the *out* object. It can be 'bcolz' or 'numpy'. If None, the value is get from *bcolz.defaults.out_flavor*.

user_dict : dict

An user-provided dictionary where the variables in expression can be found by name.

blen : int

The length of the block to be evaluated in one go internally. The default is a value that has been tested experimentally and that offers a good enough peformance / memory usage balance.

kwargs : list of parameters or dictionary

Any parameter supported by the carray constructor.

Returns **out** : bcolz or numpy object

The outcome of the expression. In case out_flavor='bcolz', you can adjust the properties of this object by passing any additional arguments supported by the carray constructor in *kwargs*.

`bcolz.`**`fill`**(*shape*, *dtype=float*, *dflt=None*, ***kwargs*)

Return a new carray or ctable object of given shape and type, filled with *dflt*.

Parameters **shape** : int

Shape of the new array, e.g., `(2, 3)`.

dflt : Python or NumPy scalar

The value to be used during the filling process. If None, values are filled with zeros. Also, the resulting carray will have this value as its *dflt* value.

dtype : data-type, optional

The desired data-type for the array, e.g., *numpy.int8*. Default is *numpy.float64*.

kwargs : list of parameters or dictionary

Any parameter supported by the carray constructor.

Returns **out** : carray or ctable

Bcolz object filled with *dflt* values with the given shape and dtype.

See also:

ones, *zeros*

`bcolz.`**`fromiter`**(*iterable*, *dtype*, *count*, ***kwargs*)

Create a carray/ctable from an *iterable* object.

Parameters **iterable** : iterable object

An iterable object providing data for the carray.

dtype : numpy.dtype instance

Specifies the type of the outcome object.

count : int

The number of items to read from iterable. If set to -1, means that the iterable will be used until exhaustion (not recommended, see note below).

kwargs : list of parameters or dictionary

Any parameter supported by the carray/ctable constructors.

Returns **out** : a carray/ctable object

Notes

Please specify *count* to both improve performance and to save memory. It allows *fromiter* to avoid looping the iterable twice (which is slooow). It avoids memory leaks to happen too (which can be important for large iterables).

`bcolz.iterblocks`(*cobj*, *blen=None*, *start=0*, *stop=None*)

Iterate over a *cobj* (carray/ctable) in blocks of size *blen*.

Parameters **cobj** : carray/ctable object

The bcolz object to be iterated over.

blen : int

The length of the block that is returned. The default is the chunklen, or for a ctable, the minimum of the different column chunklens.

start : int

Where the iterator starts. The default is to start at the beginning.

stop : int

Where the iterator stops. The default is to stop at the end.

Returns **out** : iterable

This iterable returns data blocks as NumPy arrays of homogeneous or structured types, depending on whether *cobj* is a carray or a ctable object.

See also:

`whereblocks`

`bcolz.ones`(*shape*, *dtype=float*, ***kwargs*)

Return a new carray object of given shape and type, filled with ones.

Parameters **shape** : int

Shape of the new array, e.g., `(2,3)`.

dtype : data-type, optional

The desired data-type for the array, e.g., *numpy.int8*. Default is *numpy.float64*.

kwargs : list of parameters or dictionary

Any parameter supported by the carray constructor.

Returns **out** : carray or ctable

Bcolz object of ones with the given shape and dtype.

See also:

fill, *zeros*

`bcolz.zeros(shape, dtype=float, **kwargs)`

Return a new carray object of given shape and type, filled with zeros.

Parameters **shape** : int

Shape of the new array, e.g., `(2, 3)`.

dtype : data-type, optional

The desired data-type for the array, e.g., *numpy.int8*. Default is *numpy.float64*.

kwargs : list of parameters or dictionary

Any parameter supported by the carray constructor.

Returns **out** : carray or ctable

Bcolz object of zeros with the given shape and dtype.

See also:

fill, *ones*

`bcolz.open(rootdir, mode='a')`

Open a disk-based carray/ctable.

Parameters **rootdir** : pathname (string)

The directory hosting the carray/ctable object.

mode : the open mode (string)

Specifies the mode in which the object is opened. The supported values are:

- 'r' for read-only
- 'w' for emptying the previous underlying data
- 'a' for allowing read/write on top of existing data

Returns **out** : a carray/ctable object or IOError (if not objects are found)

`bcolz.walk(dir, classname=None, mode='a')`

Recursively iterate over carray/ctable objects hanging from *dir*.

Parameters **dir** : string

The directory from which the listing starts.

classname : string

If specified, only object of this class are returned. The values supported are 'carray' and 'ctable'.

mode : string

The mode in which the object should be opened.

Returns **out** : iterator

Iterator over the objects found.

4.4 Top level printing functions

bcolz.**array2string**(*a*, *max_line_width=None*, *precision=None*, *suppress_small=None*, *separator=' '*, *prefix=""*, *style=repr*, *formatter=None*)

Return a string representation of a carray/ctable object.

This is the same function than in NumPy. Please refer to NumPy documentation for more info.

See Also: `set_printoptions()`, `get_printoptions()`

bcolz.**get_printoptions**()

Return the current print options.

This is the same function than in NumPy. For more info, please refer to the NumPy documentation.

See Also: `array2string()`, `set_printoptions()`

bcolz.**set_printoptions**(*precision=None*, *threshold=None*, *edgeitems=None*, *linewidth=None*, *suppress=None*, *nanstr=None*, *infstr=None*, *formatter=None*)

Set printing options.

These options determine the way floating point numbers in carray objects are displayed. This is the same function than in NumPy. For more info, please refer to the NumPy documentation.

See Also: `array2string()`, `get_printoptions()`

4.5 Utility functions

bcolz.**set_nthreads**(*nthreads*)

Sets the number of threads to be used during bcolz operation.

This affects to both Blosc and Numexpr (if available). If you want to change this number only for Blosc, use *blosc_set_nthreads* instead.

Parameters **nthreads** : int

The number of threads to be used during bcolz operation.

Returns **out** : int

The previous setting for the number of threads.

See also:

`blosc_set_nthreads`

bcolz.**blosc_set_nthreads**(*nthreads*)

Sets the number of threads that Blosc can use.

Parameters **nthreads** : int

The desired number of threads to use.

Returns **out** : int

The previous setting for the number of threads.

bcolz.**detect_number_of_cores**()

Return the number of cores in this system.

bcolz.**blosc_version**()

Return the version of the Blosc library.

`bcolz.print_versions()`

Print all the versions of packages that bcolz relies on.

`bcolz.test(verbose=False, heavy=False)`

Run all the tests in the test suite.

If *verbose* is set, the test suite will emit messages with full verbosity (not recommended unless you are looking into a certain problem).

If *heavy* is set, the test suite will be run in *heavy* mode (you should be careful with this because it can take a lot of time and resources from your computer).

4.6 The carray class

class `bcolz.carray`

A compressed and enlargeable data container either in-memory or on-disk.

carray exposes a series of methods for dealing with the compressed container in a NumPy-like way.

Parameters **array** : a NumPy-like object

This is taken as the input to create the carray. It can be any Python object that can be converted into a NumPy object. The data type of the resulting carray will be the same as this NumPy object.

cparams : instance of the *cparams* class, optional

Parameters to the internal Blosc compressor.

dtype : NumPy dtype

Force this *dtype* for the carray (rather than the *array* one).

dflt : Python or NumPy scalar

The value to be used when enlarging the carray. If None, the default is filling with zeros.

expectedlen : int, optional

A guess on the expected length of this object. This will serve to decide the best *chunklen* used for compression and memory I/O purposes.

chunklen : int, optional

The number of items that fits into a chunk. By specifying it you can explicitely set the chunk size used for compression and memory I/O. Only use it if you know what are you doing.

rootdir : str, optional

The directory where all the data and metadata will be stored. If specified, then the carray object will be disk-based (i.e. all chunks will live on-disk, not in memory) and persistent (i.e. it can be restored in other session, e.g. via the *open()* top-level function).

safe : bool (defaults to True)

Coerces inputs to array types. Set to false if you always give correctly typed, strided, and shaped arrays and if you never use Object dtype.

mode : str, optional

The mode that a *persistent* carray should be created/opened. The values can be:

- 'r' for read-only

- ‘w’ for read/write. During carray creation, the *rootdir* will be removed if it exists. During carray opening, the carray will be resized to 0.
- ‘a’ for append (possible data inside *rootdir* will not be removed).

Attributes

atomsize	atomsize: ‘int’
attrs	The attribute accessor.
cbytes	The compressed size of this object (in bytes).
chunklen	The chunklen of this object (in rows).
chunks	chunks: object
cparams	The compression parameters for this object.
dflt	The default value of this object.
dtype	The dtype of this object.
itemsize	itemsize: ‘int’
leftover_array	Array containing the leftovers chunk (uncompressed chunk)
leftover_bytes	Number of bytes in the leftover_array
leftover_elements	Number of elements in the leftover_array
leftover_ptr	Pointer referring to the leftover_array
len	The length (leading dimension) of this object.
mode	The mode used to create/open the *mode*.
nbytes	The original (uncompressed) size of this object (in bytes).
nchunks	Number of chunks in the carray
ndim	The number of dimensions of this object.
nleftover	The number of leftover elements.
partitions	List of tuples indicating the bounds for each chunk
rootdir	The on-disk directory used for persistency.
safe	Whether or not to perform type/shape checks on every operation.
shape	The shape of this object.
size	The size of this object.

Methods

append(self, array)	Append a numpy *array* to this instance.
copy(self, **kwargs)	Return a copy of this object.
flush(self)	Flush data in internal buffers to disk.
free_cachemem(self)	Release in-memory cached chunk
iter(self[, start, stop, step, limit, skip, ...])	Iterator with *start*, *stop* and *step* bounds.
next	
purge(self)	Remove the underlying data for on-disk arrays.
reshape(self, newshape)	Returns a new carray containing the same data with a new shape.
resize(self, nitems)	Resize the instance to have *nitems*.
sum(self[, dtype])	Return the sum of the array elements.
trim(self, nitems)	Remove the trailing *nitems* from this instance.
view(self)	Create a light weight view of the data in the original carray.
where(self, boolarr[, limit, skip])	Iterator that returns values of this object where *boolarr* is true.
wheretrue(self[, limit, skip])	Iterator that returns indices where this object is true.

`__getitem__`
x.__getitem__(key) <==> x[key]

Returns values based on *key*. All the functionality of `ndarray.__getitem__()` is supported (including fancy indexing), plus a special support for expressions:

Parameters **key** : string

It will be interpret as a boolean expression (computed via *eval*) and the elements where these values are true will be returned as a NumPy array.

See also:

`eval`

`__setitem__`
x.__setitem__(key, value) <==> x[key] = value

Sets values based on *key*. All the functionality of `ndarray.__setitem__()` is supported (including fancy indexing), plus a special support for expressions:

Parameters **key** : string

It will be interpret as a boolean expression (computed via *eval*) and the elements where these values are true will be set to *value*.

See also:

`eval`

`append`(*self*, *array*)
Append a numpy *array* to this instance.

Parameters **array** : NumPy-like object

The array to be appended. Must be compatible with shape and type of the carray.

`atomsize`
atomsize: 'int'

`attrs`
The attribute accessor.

See also:

`attrs.attrs`

`cbytes`
The compressed size of this object (in bytes).

`chunklen`
The chunklen of this object (in rows).

`chunks`
chunks: object

`copy`(*self*, ***kwargs*)
Return a copy of this object.

Parameters **kwargs** : list of parameters or dictionary

Any parameter supported by the carray constructor.

Returns **out** : carray object

The copy of this object.

cparams
The compression parameters for this object.

dflt
The default value of this object.

dtype
The dtype of this object.

flush (*self*)
Flush data in internal buffers to disk.

This call should typically be done after performing modifications (__settitem__(), append()) in persistence mode. If you don't do this, you risk losing part of your modifications.

free_cachemem (*self*)
Release in-memory cached chunk

itemsize
itemsize: 'int'

iter (*self*, *start=0*, *stop=None*, *step=1*, *limit=None*, *skip=0*, *_next=False*)
Iterator with *start*, *stop* and *step* bounds.

Parameters **start** : int

The starting item.

stop : int

The item after which the iterator stops.

step : int

The number of items incremented during each iteration. Cannot be negative.

limit : int

A maximum number of elements to return. The default is return everything.

skip : int

An initial number of elements to skip. The default is 0.

Returns **out** : iterator

See also:

where, *wheretrue*

leftover_array
Array containing the leftovers chunk (uncompressed chunk)

leftover_bytes
Number of bytes in the leftover_array

leftover_elements
Number of elements in the leftover_array

leftover_ptr
Pointer referring to the leftover_array

len
The length (leading dimension) of this object.

mode
The mode used to create/open the *mode*.

nbytes
The original (uncompressed) size of this object (in bytes).

nchunks
Number of chunks in the carray

ndim
The number of dimensions of this object.

next

nleftover
The number of leftover elements.

partitions
List of tuples indicating the bounds for each chunk

purge (*self*)
Remove the underlying data for on-disk arrays.

reshape (*self*, *newshape*)
Returns a new carray containing the same data with a new shape.

Parameters newshape : int or tuple of ints

The new shape should be compatible with the original shape. If an integer, then the result will be a 1-D array of that length. One shape dimension can be -1. In this case, the value is inferred from the length of the array and remaining dimensions.

Returns reshaped_array : carray

A copy of the original carray.

resize (*self*, *nitems*)
Resize the instance to have *nitems*.

Parameters nitems : int

The final length of the object. If *nitems* is larger than the actual length, new items will appended using *self.dflt* as filling values.

rootdir
The on-disk directory used for persistency.

safe
Whether or not to perform type/shape checks on every operation.

shape
The shape of this object.

size
The size of this object.

sum (*self*, *dtype=None*)
Return the sum of the array elements.

Parameters dtype : NumPy dtype

The desired type of the output. If `None`, the dtype of *self* is used. An exception is when *self* has an integer type with less precision than the default platform integer. In that case, the default platform integer is used instead (NumPy convention).

Returns out : NumPy scalar with *dtype*

trim (*self*, *nitems*)
Remove the trailing *nitems* from this instance.

Parameters **nitems** : int

The number of trailing items to be trimmed. If negative, the object is enlarged instead.

view (*self*)
Create a light weight view of the data in the original carray.

Returns **out** : carray object

The view of this object.

See also:

copy

where (*self*, *boolarr*, *limit=None*, *skip=0*)
Iterator that returns values of this object where *boolarr* is true.

This is currently only useful for boolean carrays that are unidimensional.

Parameters **boolarr** : a carray or NumPy array of boolean type

The boolean values.

limit : int

A maximum number of elements to return. The default is return everything.

skip : int

An initial number of elements to skip. The default is 0.

Returns **out** : iterator

See also:

iter, *wheretrue*

wheretrue (*self*, *limit=None*, *skip=0*)
Iterator that returns indices where this object is true.

This is currently only useful for boolean carrays that are unidimensional.

Parameters **limit** : int

A maximum number of elements to return. The default is return everything.

skip : int

An initial number of elements to skip. The default is 0.

Returns **out** : iterator

See also:

iter, *where*

4.7 The ctable class

class bcolz.ctable.**ctable** (*columns=None*, *names=None*, ***kwargs*)
This class represents a compressed, column-wise table.

Create a new ctable from *cols* with optional *names*.

Parameters **columns** : tuple or list of column objects

The list of column data to build the ctable object. These are typically carrays, but can also be a list of NumPy arrays or a pure NumPy structured array. A list of lists or tuples is valid too, as long as they can be converted into carray objects.

names : list of strings or string

The list of names for the columns. The names in this list must be valid Python identifiers, must not start with an underscore, and has to be specified in the same order as the *cols*. If not passed, the names will be chosen as 'f0' for the first column, 'f1' for the second and so on so forth (NumPy convention).

kwargs : list of parameters or dictionary

Allows to pass additional arguments supported by carray constructors in case new carrays need to be built.

Notes

Columns passed as carrays are not be copied, so their settings will stay the same, even if you pass additional arguments (cparams, chunklen...).

Attributes

`cbytes`	The compressed size of this object (in bytes).
`cparams`	The compression parameters for this object.
`dtype`	The data type of this object (numpy dtype).
`names`	The column names of the object (list).
`nbytes`	The original (uncompressed) size of this object (in bytes).
`ndim`	The number of dimensions of this object.
`shape`	The shape of this object.
`size`	The size of this object.

Methods

`addcol`(newcol[, name, pos, move])	Add a new *newcol* object as column.
`append`(cols)	Append *cols* to this ctable.
`copy`(**kwargs)	Return a copy of this ctable.
`delcol`([name, pos, keep])	Remove the column named *name* or in position *pos*.
`eval`(expression, **kwargs)	Evaluate the *expression* on columns and return the result.
`fetchwhere`(expression[, outcols, limit, ...])	Fetch the rows fulfilling the *expression* condition.
`flush`()	Flush data in internal buffers to disk.
`free_cachemem`()	Get rid of internal caches to free memory.
`fromdataframe`(df, **kwargs)	Return a ctable object out of a pandas dataframe.
`fromhdf5`(filepath[, nodepath])	Return a ctable object out of a compound HDF5 dataset (PyTables Table).
`iter`([start, stop, step, outcols, limit, ...])	Iterator with *start*, *stop* and *step* bounds.
`resize`(nitems)	Resize the instance to have *nitems*.
`todataframe`([columns, orient])	Return a pandas dataframe out of this object.
`tohdf5`(filepath[, nodepath, mode, cparams, ...])	Write this object into an HDF5 file.
`trim`(nitems)	Remove the trailing *nitems* from this instance.

Continued or

Table 4.7 – continued from previous page

where(expression[, outcols, limit, skip, ...])	Iterate over rows where *expression* is true.
whereblocks(expression[, blen, outcols, ...])	Iterate over the rows that fullfill the *expression* condition on this ctable, in blocks

addcol(*newcol*, *name=None*, *pos=None*, *move=False*, ***kwargs*)

Add a new *newcol* object as column.

Parameters newcol : carray, ndarray, list or tuple

If a carray is passed, no conversion will be carried out. If conversion to a carray has to be done, *kwargs* will apply.

name : string, optional

The name for the new column. If not passed, it will receive an automatic name.

pos : int, optional

The column position. If not passed, it will be appended at the end.

move: boolean, optional

If the new column is an existing, disk-based carray should it a) copy the data directory (False) or b) move the data directory (True)

kwargs : list of parameters or dictionary

Any parameter supported by the carray constructor.

See also:

delcol

Notes

You should not specificy both *name* and *pos* arguments, unless they are compatible.

append(*cols*)

Append *cols* to this ctable.

Parameters cols : list/tuple of scalar values, NumPy arrays or carrays

It also can be a NumPy record, a NumPy recarray, or another ctable.

cbytes

The compressed size of this object (in bytes).

cols = None

The ctable columns accessor.

copy(***kwargs*)

Return a copy of this ctable.

Parameters kwargs : list of parameters or dictionary

Any parameter supported by the carray/ctable constructor.

Returns out : ctable object

The copy of this ctable.

cparams

The compression parameters for this object.

delcol(*name=None*, *pos=None*, *keep=False*)

Remove the column named *name* or in position *pos*.

Parameters **name: string, optional**

The name of the column to remove.

pos: int, optional

The position of the column to remove.

keep: boolean

For disk-backed columns: keep the data on disk?

See also:

addcol

Notes

You must specify at least a *name* or a *pos*. You should not specify both *name* and *pos* arguments, unless they are compatible.

dtype

The data type of this object (numpy dtype).

eval(*expression*, ***kwargs*)

Evaluate the *expression* on columns and return the result.

Parameters **expression** : string

A string forming an expression, like '2*a+3*b'. The values for 'a' and 'b' are variable names to be taken from the calling function's frame. These variables may be column names in this table, scalars, carrays or NumPy arrays.

kwargs : list of parameters or dictionary

Any parameter supported by the *eval()* top level function.

Returns **out** : bcolz object

The outcome of the expression. You can tailor the properties of this object by passing additional arguments supported by the carray constructor in *kwargs*.

See also:

eval

fetchwhere(*expression*, *outcols=None*, *limit=None*, *skip=0*, *out_flavor=None*, *user_dict={}*, *vm=None*, ***kwargs*)

Fetch the rows fulfilling the *expression* condition.

Parameters **expression** : string or carray

A boolean Numexpr expression or a boolean carray.

outcols : list of strings or string

The list of column names that you want to get back in results. Alternatively, it can be specified as a string such as 'f0 f1' or 'f0, f1'. If None, all the columns are returned. If the special name '**nrow__**' is present, the number of row will be included in output.

limit : int

A maximum number of elements to return. The default is return everything.

skip : int

An initial number of elements to skip. The default is 0.

out_flavor : string

The flavor for the *out* object. It can be 'bcolz' or 'numpy'. If None, the value is get from *bcolz.defaults.out_flavor*.

user_dict : dict

An user-provided dictionary where the variables in expression can be found by name.

vm : string

The virtual machine to be used in computations. It can be 'numexpr', 'python' or 'dask'. The default is to use 'numexpr' if it is installed.

kwargs : list of parameters or dictionary

Any parameter supported by the carray constructor.

Returns out : bcolz or numpy object

The outcome of the expression. In case out_flavor='bcolz', you can adjust the properties of this object by passing any additional arguments supported by the carray constructor in *kwargs*.

See also:

`whereblocks`

flush()

Flush data in internal buffers to disk.

This call should typically be done after performing modifications (__settitem__(), append()) in persistence mode. If you don't do this, you risk losing part of your modifications.

free_cachemem()

Get rid of internal caches to free memory.

This call can typically be made after reading from a carray/ctable so as to free the memory used internally to cache data blocks/chunks.

static **fromdataframe**(*df*, ***kwargs*)

Return a ctable object out of a pandas dataframe.

Parameters df : DataFrame

A pandas dataframe.

kwargs : list of parameters or dictionary

Any parameter supported by the ctable constructor.

Returns out : ctable object

A ctable filled with values from *df*.

See also:

`ctable.todataframe`

Notes

The 'object' dtype will be converted into a 'S'tring type, if possible. This allows for much better storage savings in bcolz.

static fromhdf5 (*filepath*, *nodepath='/ctable'*, ***kwargs*)
Return a ctable object out of a compound HDF5 dataset (PyTables Table).

Parameters **filepath** : string

The path of the HDF5 file.

nodepath : string

The path of the node inside the HDF5 file.

kwargs : list of parameters or dictionary

Any parameter supported by the ctable constructor.

Returns **out** : ctable object

A ctable filled with values from the HDF5 node.

See also:

`ctable.tohdf5`

iter (*start=0*, *stop=None*, *step=1*, *outcols=None*, *limit=None*, *skip=0*, *out_flavor=<function namedtuple>*)
Iterator with *start*, *stop* and *step* bounds.

Parameters **start** : int

The starting item.

stop : int

The item after which the iterator stops.

step : int

The number of items incremented during each iteration. Cannot be negative.

outcols : list of strings or string

The list of column names that you want to get back in results. Alternatively, it can be specified as a string such as 'f0 f1' or 'f0, f1'. If None, all the columns are returned. If the special name '**nrow__**' is present, the number of row will be included in output.

limit : int

A maximum number of elements to return. The default is return everything.

skip : int

An initial number of elements to skip. The default is 0.

out_flavor : namedtuple, tuple or ndarray

Whether the returned rows are namedtuples or tuples. Default are named tuples.

Returns **out** : iterable

See also:

`where`

names
The column names of the object (list).

nbytes
The original (uncompressed) size of this object (in bytes).

ndim
The number of dimensions of this object.

resize (*nitems*)
Resize the instance to have *nitems*.

Parameters **nitems** : int

The final length of the instance. If *nitems* is larger than the actual length, new items will appended using *self.dflt* as filling values.

shape
The shape of this object.

size
The size of this object.

todataframe (*columns=None*, *orient='columns'*)
Return a pandas dataframe out of this object.

Parameters **columns** : sequence of column labels, optional

Must be passed if orient='index'.

orient : {'columns', 'index'}, default 'columns'

The "orientation" of the data. If the keys of the input correspond to column labels, pass 'columns' (default). Otherwise if the keys correspond to the index, pass 'index'.

Returns **out** : DataFrame

A pandas DataFrame filled with values from this object.

See also:

`ctable.fromdataframe`

tohdf5 (*filepath*, *nodepath='/ctable'*, *mode='w'*, *cparams=None*, *cname=None*)
Write this object into an HDF5 file.

Parameters **filepath** : string

The path of the HDF5 file.

nodepath : string

The path of the node inside the HDF5 file.

mode : string

The mode to open the PyTables file. Default is 'w'rite mode.

cparams : cparams object

The compression parameters. The defaults are the same than for the current bcolz environment.

cname : string

Any of the compressors supported by PyTables (e.g. 'zlib'). The default is to use 'blosc' as meta-compressor in combination with one of its compressors (see *cparams* parameter above).

See also:

`ctable.fromhdf5`

trim(*nitems*)
Remove the trailing *nitems* from this instance.

Parameters nitems : int

The number of trailing items to be trimmed.

where(*expression*, *outcols=None*, *limit=None*, *skip=0*, *out_flavor=<function namedtuple>*, *user_dict={}*, *vm=None*)
Iterate over rows where *expression* is true.

Parameters expression : string or carray

A boolean Numexpr expression or a boolean carray.

outcols : list of strings or string

The list of column names that you want to get back in results. Alternatively, it can be specified as a string such as 'f0 f1' or 'f0, f1'. If None, all the columns are returned. If the special name '**nrow__**' is present, the number of row will be included in output.

limit : int

A maximum number of elements to return. The default is return everything.

skip : int

An initial number of elements to skip. The default is 0.

out_flavor : namedtuple, tuple or ndarray

Whether the returned rows are namedtuples or tuples. Default are named tuples.

user_dict : dict

An user-provided dictionary where the variables in expression can be found by name.

vm : string

The virtual machine to be used in computations. It can be 'numexpr', 'python' or 'dask'. The default is to use 'numexpr' if it is installed.

Returns out : iterable

See also:

`iter`

whereblocks(*expression*, *blen=None*, *outcols=None*, *limit=None*, *skip=0*, *user_dict={}*, *vm=None*)
Iterate over the rows that fullfill the *expression* condition on this ctable, in blocks of size *blen*.

Parameters expression : string or carray

A boolean Numexpr expression or a boolean carray.

blen : int

The length of the block that is returned. The default is the chunklen, or for a ctable, the minimum of the different column chunklens.

outcols : list of strings or string

The list of column names that you want to get back in results. Alternatively, it can be specified as a string such as 'f0 f1' or 'f0, f1'. If None, all the columns are returned. If the special name '**nrow__**' is present, the number of row will be included in output.

limit : int

A maximum number of elements to return. The default is return everything.

skip : int

An initial number of elements to skip. The default is 0.

user_dict : dict

An user-provided dictionary where the variables in expression can be found by name.

vm : string

The virtual machine to be used in computations. It can be 'numexpr', 'python' or 'dask'. The default is to use 'numexpr' if it is installed.

Returns **out** : iterable

The iterable returns numpy objects of blen length.

See also:

`See` py:func:<*bcolz.toplevel.iterblocks*> in toplevel functions.

CHAPTER 5

Optimization tips

5.1 Changing explicitly the length of chunks

You may want to use explicitly the *chunklen* parameter to fine-tune your compression levels:

```
>>> a = np.arange(1e7)
>>> bcolz.carray(a)
carray((10000000,), float64)  nbytes: 76.29 MB; cbytes: 2.57 MB; ratio: 29.72
  cparams := cparams(clevel=5, shuffle=1)
[0.0, 1.0, 2.0, ..., 9999997.0, 9999998.0, 9999999.0]
>>> bcolz.carray(a).chunklen
16384   # 128 KB = 16384 * 8 is the default chunk size for this carray
>>> bcolz.carray(a, chunklen=512)
carray((10000000,), float64)  nbytes: 76.29 MB; cbytes: 10.20 MB; ratio: 7.48
  cparams := cparams(clevel=5, shuffle=1)
[0.0, 1.0, 2.0, ..., 9999997.0, 9999998.0, 9999999.0]
>>> bcolz.carray(a, chunklen=8*1024)
carray((10000000,), float64)  nbytes: 76.29 MB; cbytes: 1.50 MB; ratio: 50.88
  cparams := cparams(clevel=5, shuffle=1)
[0.0, 1.0, 2.0, ..., 9999997.0, 9999998.0, 9999999.0]
```

You see, the length of the chunk affects very much compression levels and the performance of I/O to carrays too.

In general, however, it is safer (and quicker!) to use the *expectedlen* parameter (see next section).

5.2 Informing about the length of your carrays

If you are going to add a lot of rows to your carrays, be sure to use the *expectedlen* parameter in creating time to inform the constructor about the expected length of your final carray; this allows bcolz to fine-tune the length of its chunks more easily. For example:

```
>>> a = np.arange(1e7)
>>> bcolz.carray(a, expectedlen=10).chunklen
512
>>> bcolz.carray(a, expectedlen=10*1000).chunklen
4096
>>> bcolz.carray(a, expectedlen=10*1000*1000).chunklen
16384
>>> bcolz.carray(a, expectedlen=10*1000*1000*1000).chunklen
131072
```

5.3 Lossy compression via the quantize filter

Using the *quantize* filter for allowing lossy compression on floating point data. Data is quantized using `np.around(scale*data)/scale`, where scale is 2**bits, and bits is determined from the quantize value. For example, if quantize=1, bits will be 4. 0 means that the quantization is disabled.

Here it is an example of what you can get from the quantize filter:

```
In [9]: a = np.cumsum(np.random.random_sample(1000*1000)-0.5)

In [10]: bcolz.carray(a, cparams=bcolz.cparams(quantize=0))  # no quantize
Out[10]:
carray((1000000,), float64)
  nbytes: 7.63 MB; cbytes: 6.05 MB; ratio: 1.26
  cparams := cparams(clevel=5, shuffle=1, cname='blosclz', quantize=0)
[ -2.80946077e-01  -7.63925274e-01  -5.65575047e-01 ...,   3.59036158e+02
   3.58546624e+02   3.58258860e+02]

In [11]: bcolz.carray(a, cparams=bcolz.cparams(quantize=1))
Out[11]:
carray((1000000,), float64)
  nbytes: 7.63 MB; cbytes: 1.41 MB; ratio: 5.40
  cparams := cparams(clevel=5, shuffle=1, cname='blosclz', quantize=1)
[ -2.50000000e-01  -7.50000000e-01  -5.62500000e-01 ...,   3.59036158e+02
   3.58546624e+02   3.58258860e+02]

In [12]: bcolz.carray(a, cparams=bcolz.cparams(quantize=2))
Out[12]:
carray((1000000,), float64)
  nbytes: 7.63 MB; cbytes: 2.20 MB; ratio: 3.47
  cparams := cparams(clevel=5, shuffle=1, cname='blosclz', quantize=2)
[ -2.81250000e-01  -7.65625000e-01  -5.62500000e-01 ...,   3.59036158e+02
   3.58546624e+02   3.58258860e+02]

In [13]: bcolz.carray(a, cparams=bcolz.cparams(quantize=3))
Out[13]:
carray((1000000,), float64)
  nbytes: 7.63 MB; cbytes: 2.30 MB; ratio: 3.31
  cparams := cparams(clevel=5, shuffle=1, cname='blosclz', quantize=3)
[ -2.81250000e-01  -7.63671875e-01  -5.65429688e-01 ...,   3.59036158e+02
   3.58546624e+02   3.58258860e+02]
```

As you can see, the compression ratio can improve pretty significantly when using the quantize filter. It is important to note that by using quantize you are loosing precision on your floating point data.

Also note how the first elements in the quantized arrays have less significant digits, but not the last ones. This is a side effect due to how bcolz stores the trainling data that do not fit in a whole chunk. But in general you should expect a loss in precision.

Defaults for bcolz operation

You can tailor the behaviour of bcolz by changing the values of certain some special top level variables whose defaults are listed here. You can change these values in two ways:

- In your program: the changes will be temporary. For example:

```
bcolz.defaults.out_flavor = "numpy"
```

- Manually modify the `defaults.py` module of the bcolz package: the changes will be persistent. For example, replace:

```
defaults.out_flavor = "bcolz"
```

  by:

```
defaults.out_flavor = "numpy"
```

Generally, only the former is needed.

6.1 Defaults in contexts

bcolz allows to set short-lived defaults in contexts. For example:

```
with bcolz.defaults_ctx(vm="python", cparams=bcolz.cparams(clevel=0)):
    cout = bcolz.eval("(x + 1) < 0")
```

means that the *bcolz.eval* operation will be made using a "python" virtual machine and no compression for the *cout* output.

6.2 List of default values

out_flavor
The flavor for the output object in `eval()` and others that call this indirectly. It can be 'bcolz' or 'numpy'. Default is 'bcolz'.

vm
The virtual machine to be used in computations (via `eval()`). It can be 'python', 'numexpr' or 'dask'. Default is 'numexpr', if installed. If not, 'dask' is used, if installed. And if neither of these are installed, then the 'python' interpreter is used (via numpy).

cparams

The defaults for parameters used in compression (dict). The default is {'clevel': 5, 'shuffle': True, 'cname': 'lz4', quantize: 0}.

See Also: `cparams.setdefaults()`

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

Index

Symbols

A

B

C

D

E

F

G

I

L

M

N

O

P

R

S

T

V

W

Z