

Data Serialization

From pickle to databases and HDF5

Francesc Alted

Freelance Developer and PyTables Creator

Advanced Scientific Programming in Python
2010 Autumn School, Trento, Italy

Outline

- 1 The Basics
 - Introduction
 - Pickling our objects
 - Relational databases
- 2 Numerical Binary Formats
 - Why we need them?
 - The NPY format
 - The HDF5 format
 - The NetCDF4 format
- 3 Adding Compression
 - Why compression?
 - Solid compression
 - Chunked compression

Outline

- 1 The Basics
 - Introduction
 - Pickling our objects
 - Relational databases
- 2 Numerical Binary Formats
 - Why we need them?
 - The NPY format
 - The HDF5 format
 - The NetCDF4 format
- 3 Adding Compression
 - Why compression?
 - Solid compression
 - Chunked compression

What “serialization” means?

“Serialization is the process of converting a data structure or object into a sequence of bits so that it can be stored in a file or memory buffer, or transmitted across a network connection link to be “resurrected” later in the same or another computer environment.”

“The basic mechanisms are to flatten object(s) into a one-dimensional stream of bits, and to turn that stream of bits back into the original object(s).”

– From <http://www.parashift.com/c++-faq-lite/serialization.html>

Serialization tools

There are literally zillions of serialization tools and formats (text, XML, or binary based), but we'll be focusing on those that are:

- Easy to use
- Space-efficient
- Fast

In particular, we are not going to discuss text-based formats (e.g. XML, CSV, JSON, YAML ...).

Serialization tools that comes with Python

Python comes with a complete toolset of modules for serialization purposes:

- pickle, and its cousin, cPickle, for quick-and-dirty serialization.
- shelve, a persistent dictionary based on DBM databases.
- A common database API for communicating with relational databases.

Serialization tools for binary data

Additionally, there are lots of third-party libraries for specialized uses. Here will center on numerical formats:

- NPY, NPZ: NumPy own's format.
- Wrappers for HDF5, a standard de-facto format and library: PyTables, h5py.
- Wrappers for NetCDF4, a widely used library based on HDF5: netcdf4-python, Scientific.IO.NetCDF.

Outline

- 1 The Basics
 - Introduction
 - Pickling our objects
 - Relational databases
- 2 Numerical Binary Formats
 - Why we need them?
 - The NPY format
 - The HDF5 format
 - The NetCDF4 format
- 3 Adding Compression
 - Why compression?
 - Solid compression
 - Chunked compression

The pickle module

Serializes an object into a stream of bytes that can be saved to a file and later restored:

Example

```
import pickle
obj = SomeObject()
f = open(filename, 'wb')
pickle.dump(obj, f)
f.close()

# ... later on
import pickle
f = open(filename, 'rb')
obj = pickle.load(f)
f.close()
```

What pickle does

- It can serialize both basic Python data structures or user-defined classes.
- Always serializes data, not code (it tries to import classes if found in the pickle).

For security reasons, programs should not unpickle data received from untrusted sources.

Its cPickle cousin

- Implemented in C (i.e. significantly faster than pickle).
- But more restrictive (does not allow subclassing of the Pickler and Unpickler objects).
- Python 3 pickle can use the C implementation transparently.

Pickling a NumPy array

```
>>> a = np.linspace(0, 100, 1e7)

>>> time pickle.dump(a, open('p1','w'))
CPU times:  user 5.89 s, sys: 0.59 s, total: 6.48 s

>>> time pickle.dump(a, open('p2','w'), pickle.HIGHEST_PROTOCOL)
CPU times:  user 0.05 s, sys: 0.12 s, total: 0.16 s

>>> time cPickle.dump(a, open('p3','w'), pickle.HIGHEST_PROTOCOL)
CPU times:  user 0.02 s, sys: 0.08 s, total: 0.11 s

>>> ls -sh p1 p2 p3
186M p1 77M p2 77M p3
```

Always try to use cPickle and HIGHEST_PROTOCOL

pickle/cPickle limitations

- You need to reload all the data in the pickle before you can use any part of it. That might be inconvenient for large datasets.
- Data can only be retrieved by other Python interpreters. You lose data portability with other languages.
- Not every object in Python can be serialized by pickle (e.g. extensions).

Recommendations for using pickle

- Use it mainly for small data structures.
- If you have a lot of variables that you want to save, use a dictionary for tying them together first.
- When using the IPython shell, be sure to use the very convenient `%store` magic (it uses `pickle` under the hood).

The `shelve` module

- Provides support for persistent objects using a special “shelf” object.
- The “shelf” behaves like a disk-based dictionary (DBM-style).
- The values of the dictionary can be any object that can be pickled.

Example with `shelve`

```
>>> import shelve
>>>
>>> db = shelve.open("database", "c")
>>> db["one"] = 1
>>> db["two"] = 2
>>> db["three"] = 3
>>> db.close()
>>>
>>> db = shelve.open("database", "r")
>>> for key in db.keys():
.....:     print repr(key), repr(db[key])
.....:
'one' 1
'two' 2
'three' 3
```


Pros and cons of the `shelve` module

Pros

Easy to retrieve just a selected set of variables.
Specially handy for large pickles.

Cons

Suffers the same problems than `pickle`.

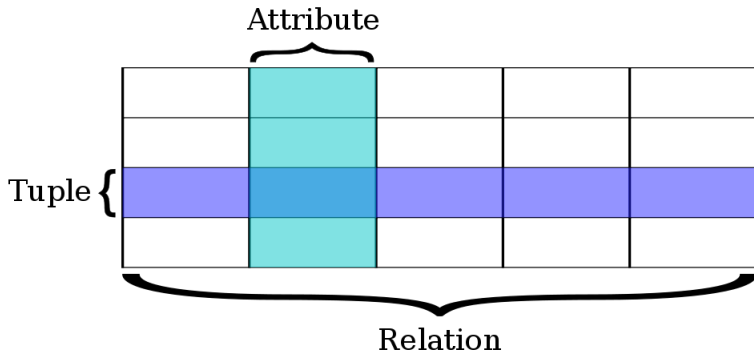
Outline

- 1 The Basics
 - Introduction
 - Pickling our objects
 - Relational databases
- 2 Numerical Binary Formats
 - Why we need them?
 - The NPY format
 - The HDF5 format
 - The NetCDF4 format
- 3 Adding Compression
 - Why compression?
 - Solid compression
 - Chunked compression

What's a relational database?

- A set of tables containing data fitted into predefined categories.
- Each table (a relation) contains one or more data categories in columns.
- Each row contains a unique instance of data for the categories defined by the columns.
- Data can be accessed in many different ways without having to reorganize the tables.

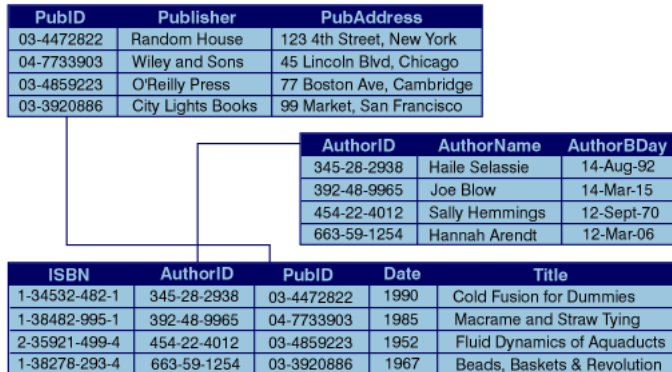
Terminology



Base and derived relations

- In a relational database, all data are stored and accessed via relations.
- Relations that store data are called "base relations", and in implementations are called "tables".
- Other relations do not store data, but are computed by applying relational operations to other relations.
- These relations are sometimes called "derived relations".
- In implementations these are called "views" or "queries".

Example of relational database



Queries with SQL language

Simple query involving one single table (relation):

```
SELECT AuthorName FROM AUTHORS WHERE AuthorBDay > 1970
```

Complex query involving multiple relations:

```
SELECT AuthorName FROM AUTHORS a, BOOKS b, PUBLISHERS p
  WHERE AuthorBDay > 1970
        AND a.AuthorID = b.AuthorID
        AND b.PubID = p.PubID
        AND p.Publisher = "Random House"
  GROUP BY AuthorBDay
```

Beware: complex queries can consume a lot of resources!

Relational database API specification

- The Python community has developed a standard API for accessing relational databases in a uniform way (PEP 249).
- Specific database modules (e.g. MySQL, Oracle, Postgres ...) follow this specification, but may add more features.
- Python comes with SQLite, a relational database accessible via the `sqlite3` module.

ORM (Object Relational Mapping)

- The relational database API in Python is powerful, but pretty rough to use and *not object-oriented*.
- Many projects have appeared to add an object-oriented layer on top of this API:
 - SQLAlchemy
 - Django's native ORM
 - Storm
 - Elixir
 - SQLAlchemy (the one that started it all)
 - ... probably a lot more ...

Creating a database with an ORM (Storm)

```
class Kind:
    __storm_table__ = 'kinds'
    id = Int(primary=True)
    name = Unicode()

class Thing:
    __storm_table__ = 'things'
    id = Int(primary=True)
    name = Unicode()
    description = Unicode()
    kind_id = Int()
    kind = Reference(kind_id, Kind.id)

db = create_database('sqlite:'); store = Store(db)
kind = Kind(name='Flowers'); store.add(kind)
thing = Thing(name='Red Rose'); thing.kind = kind;
store.add(thing)
store.commit()
```

Querying with an ORM (Storm)

```
>>> result = store.find((Kind, Thing),  
...   Thing.kind_id == Kind.id,  
...   Thing.name.like(u"% Rose %"))  
  
>>> [(kind.name, thing.name) for kind, thing in result]  
[(u'Flowers', u'Red Rose'), (u'Jars', u'Rose Vase')]
```

RDBMs highlights

They offer ACID (atomicity, consistency, isolation, durability) properties, that can be translated into:

- Referential integrity.
- Transaction support.
- Data consistency.

+ Indexing capabilities (accelerate queries in large tables).

But this comes with a price...

RDBMs drawbacks

- Insertions are SLOOOOW.
- Not very space-efficient.
- Not well adapted to handle large numerical datasets (no direct interface with NumPy).
- You need a knowledgeable RDBM administrator to squeeze all the performance out of them.

Outline

- 1 The Basics
 - Introduction
 - Pickling our objects
 - Relational databases
- 2 Numerical Binary Formats
 - Why we need them?
 - The NPY format
 - The HDF5 format
 - The NetCDF4 format
- 3 Adding Compression
 - Why compression?
 - Solid compression
 - Chunked compression

What's a numerical binary format?

- It is a format specialized in saving and retrieving large amounts of numerical data.
- Usually come with libraries that can understand that format.
- They range from the very simple (NPY) to rather complex and powerful (HDF5).
- There are a really huge number of numerical formats depending on the needs. Will center just on a few.

Why we need a binary format?

- They are closer to memory representation.
- Their representation is space-efficient (1 byte in-memory \approx 1 bytes on disk).
- They are CPU-friendly (in general you do not have to convert from one representation to another).

NumPy: the real cornerstone of numerical interfaces

- NumPy is the standard de-facto for dealing with numerical data in-memory.
- Hence, most of the interfaces to numerical formats in the Python world use NumPy to interact with the database.
- In some cases the integration is so tight that it could be difficult to say if you are working with NumPy or the interface.

Outline

- 1 The Basics
 - Introduction
 - Pickling our objects
 - Relational databases
- 2 Numerical Binary Formats
 - Why we need them?
 - The NPY format
 - The HDF5 format
 - The NetCDF4 format
- 3 Adding Compression
 - Why compression?
 - Solid compression
 - Chunked compression

The NPY format

- Created back in 2007 for overcoming limitations of pickle for NumPy arrays as well as `numpy.tofile()` / `numpy.fromfile()` functions (see “A Simple File Format for NumPy Arrays” NEP).
- It is a binary format, so it is space-efficient.
- It comes integrated with NumPy.

NPY exposes the simplest API for NumPy

Available via save/load NumPy functions:

```
>>> data = numpy.arange(1e7)
>>> numpy.save('test.npy', data)
>>> data2 = numpy.load('test.npy')
>>> numpy.alltrue(data == data2)
True
```

Simple to use!

Memory-mapping and NPY

You can open a NPY file in memmap-mode for accessing data directly from disk:

```
>>> mmapdata = numpy.load('test.npy', mmap_mode='r+')
>>> mmapdata
memmap([ 0.00000000e+00, 1.00000000e+00, 2.00000000e+00, ...,
         9.99999700e+06, 9.99999800e+06, 9.99999900e+06])
>>> mmapdata[-5:] + data[:5]
memmap([ 9999995., 9999997., 9999999., 10000001., 10000003.])
>>> del mmapdata          # close access to 'test.npy'
```

Saving several arrays with NPZ

The NPY format has a special mode that can save several arrays in one single zip file (but no compression is used at all!):

```
>>> a = np.linspace(0, 100, 1e7)
>>> sina = np.sin(a)
>>> np.savez("test.npz", a=a, sina=sina)
>>> !file test.npz
test.npz: Zip archive data, at least v2.0 to extract
>>> arrs = np.load("test.npz")
>>> arrs <numpy.lib.npyio.NpzFile object at 0x1622090>
>>> arrs.items()
[('a', array([ 0.000000e+00, 1.000010e-05, 2.000020e-05, ...,
              9.999800e+01, 9.999900e+01, 1.000000e+02])),
 ('sina', array([ 0.000000e+00, 1.000010e-05, 2.000020e-05, ...,
                 -5.063828e-01, -5.063742e-01, -5.063656e-01]))]
```

Pros and cons of NPY

Pros:

- Binary format, so space-efficient.
- Avoids duplication of data in memory during saving/loading operations.
- Array data accessible through memory-mapping.

Cons:

- The memory mapping feature only allows to deal with files that do not exceed the available virtual memory.
- Non-standard format outside the NumPy community.
- No other features than basic input/output (e.g. no metadata allowed).

Outline

- 1 The Basics
 - Introduction
 - Pickling our objects
 - Relational databases
- 2 Numerical Binary Formats
 - Why we need them?
 - The NPY format
 - **The HDF5 format**
 - The NetCDF4 format
- 3 Adding Compression
 - Why compression?
 - Solid compression
 - Chunked compression

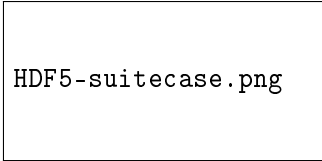
The HDF5 format

- HDF5 (Hierarchical Data Format v5) is a library and file format for storing and managing da any kind of data:
<http://www.hdfgroup.org/HDF5/doc/H5.format.html>
- It supports an unlimited variety of datatypes, and is designed for flexible and efficient I/O and for high volume and complex data.
- Originally developed at the NCSA, and currently maintained by The THG Group, a non for-profit organization.
- HDF5 has been around for over twenty years, and has become a standard de-facto format supported by many applications (MatLab, IDL, R, Mathematica ...).

Outstanding features of HDF5

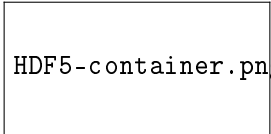
- Can store all kinds of data in a variety of ways.
- Runs on most systems.
- Lots of tools to access data.
- Long term format support (HDF-EOS, CGNS).
- Library and format emphasis on I/O efficiency and different kinds of storage.

An HDF5 “file” is a container



HDF5-suitecase.png

Structures to organize objects



HDF5-container.png

Python interfaces

h5py is an attempt to map the HDF5 feature set to NumPy as closely as possible:

- It also provides access to nearly all of the HDF5 C API (the so-called low-level API).
- Not designed to go beyond HDF5/NumPy capabilities.

PyTables builds up an additional abstraction layer on top of HDF5 and NumPy where it implements things like:

- An enhanced type system (enumerated, time, variable length types and default values supported).
- An engine for enabling complex queries and out-of-core computations (using Numexpr behind the scenes).
- Advanced indexing capabilities (Pro version).

Creating an HDF5 file

```
>>> import tables
>>> f = tables.openFile("example.h5", "w")
>>> group = f.createGroup("/", "reduced_data")
>>> ds = f.createArray(group, "array", np.array([1,2,3,4]))
>>> ds
/reduced_data/array (Array(4,)) ''
  atom := Int64Atom(shape=(), dflt=0)
  maindim := 0
  flavor := 'numpy'
  byteorder := 'little'
  chunkshape := None
>>> f.close()
```

Creating a table

```
>>> gen = ((i, i*2, i**3) for i in xrange(1000000))
>>> sa = numpy.fromiter(gen, dtype="i4,i8,f8")
>>> tab = f.createTable(f.root, 'table', sa)
>>> tab
/table (Table(1000000,)) ''
  description := {
    "f0":  Int32Col(shape=(), dflt=0, pos=0),
    "f1":  Int64Col(shape=(), dflt=0, pos=1),
    "f2":  Float64Col(shape=(), dflt=0.0, pos=2)}
  byteorder := 'little'
  chunkshape := (8192,)
```

Querying a table

```
>>> tab[3]
(3, 6, 27.0)
>>> tab[3:2000]
array([(3, 6, 27.0), (4, 8, 64.0), (5, 10, 125.0), ...,
      (1997, 3994, 7964053973.0), (1998, 3996, 7976023992.0),
      (1999, 3998, 7988005999.0)],
      dtype=[('f0', '<i4'), ('f1', '<i4'), ('f2', '<f8')])
>>> tab[[3,100]]
array([(3, 6, 27.0), (100, 200, 1000000.0)],
      dtype=[('f0', '<i4'), ('f1', '<i4'), ('f2', '<f8')])
>>> [v[:] for v in tab.where("(f0 > 1) & (f2 < 100)")]
[(2, 4, 8.0), (3, 6, 27.0), (4, 8, 64.0)]
```


Modifying table data

```
>>> tab[0] = (3, 3, 3.0)
>>> tab[:4]
array([(3, 3, 3.0), (1, 2, 1.0), (2, 4, 8.0), (3, 6, 27.0)],
      dtype=[('f0', '<i4'), ('f1', '<i8'), ('f2', '<f8')])
>>> tab[[1, 3]] = [(4, 4, 4.0)]*2
>>> tab[:4]
array([(3, 3, 3.0), (4, 4, 4.0), (2, 4, 8.0), (4, 4, 4.0)],
      dtype=[('f0', '<i4'), ('f1', '<i8'), ('f2', '<f8')])
>>> for row in tab.where("(f0 < 4) & (f2 <= 8.)"):
...     row['f1'] = 0
...     row.update()
...
>>> tab[:4]
array([(3, 0, 3.0), (4, 4, 4.0), (2, 0, 8.0), (4, 4, 4.0)],
      dtype=[('f0', '<i4'), ('f1', '<i8'), ('f2', '<f8')])
```

Annotating your datasets

```
>>> print tab /table (Table(1000000,)) "  
>>> tab.attrs.TITLE = "sample data"  
>>> print tab  
/table (Table(1000000,)) 'sample data'  
>>> tab.attrs.CLASS  
'TABLE'  
>>> tab.attrs.mycomment = "Enjoy data!"  
>>> tab.attrs.complementary_data = np.array([3,2,3])  
>>> tab.attrs.complementary_data  
array([3, 2, 3])
```

Outline

- 1 The Basics
 - Introduction
 - Pickling our objects
 - Relational databases
- 2 Numerical Binary Formats
 - Why we need them?
 - The NPY format
 - The HDF5 format
 - The NetCDF4 format
- 3 Adding Compression
 - Why compression?
 - Solid compression
 - Chunked compression

The NetCDF4 format

- NetCDF (network Common Data Form) is a set of libraries data formats that support array-oriented scientific data.
- NetCDF4 uses HDF5 as the underlying storage layer.
- Creating a netCDF4 file with the netCDF4 library results in an HDF5 file.
- Very spread in Oceanography, Meteorology and similar disciplines.

Python interfaces for NetCDF4

Scientific.IO.NetCDF: <http://dirac.cnrs-orleans.fr/ScientificPython>

netcdf4-python: <http://code.google.com/p/netcdf4-python>

Creating a NetCDF4 file

```
>>> from netCDF4 import Dataset
>>> rootgrp = Dataset('test.nc', 'w', format='NETCDF4')
>>> fcstgrp = rootgrp.createGroup('forecasts')
>>> analgrp = rootgrp.createGroup('analyses')
>>> print rootgrp.groups
{'analyses': <netCDF4._Group object at 0x24a54c30>,
 'forecasts': <netCDF4._Group object at 0x24a54bd0>}
>>> rootgrp.close()
```

Outline

- 1 The Basics
 - Introduction
 - Pickling our objects
 - Relational databases
- 2 Numerical Binary Formats
 - Why we need them?
 - The NPY format
 - The HDF5 format
 - The NetCDF4 format
- 3 **Adding Compression**
 - **Why compression?**
 - Solid compression
 - Chunked compression

Why compression?

- Files takes less space (the obvious reason).
- I/O speed can benefit a lot.
- If compression speed is good enough, it is a nice way to shelve arrays in-memory.

Two compression paradigms

- Solid:** Data is compressed and decompressed as a whole. A compressed buffer must be decompressed completely before usage. The typical case is compressing a pickle.
- Chunked:** Data is stored compressed in chunks and a chunk is decompressed only when it is needed. Typical case is HDF5 / NetCDF4 files (or compressed filesystems).

Outline

- 1 The Basics
 - Introduction
 - Pickling our objects
 - Relational databases
- 2 Numerical Binary Formats
 - Why we need them?
 - The NPY format
 - The HDF5 format
 - The NetCDF4 format
- 3 **Adding Compression**
 - Why compression?
 - **Solid compression**
 - Chunked compression

Creating a compressed pickle

Python comes with a series of compressors that are ready to use.
The usual method is to compress a pickle:

```
>>> a = np.linspace(0, 100, 1e7)
>>> pa = cPickle.dumps(a, cPickle.HIGHEST_PROTOCOL)
>>> a.size*a.itemsize, len(pa)
(80000000, 80000135)
>>> zpa = zlib.compress(pa, 9)
>>> len(pa), len(zpa), len(pa) / len(zpa)
(80000135, 52946378, 1.5109652071006632)
>>> bloscpa = blosc.compress(pa, a.itemsize, 9)
>>> len(pa), len(bloscpa), len(pa) / len(bloscpa)
(80000135, 8028398, 9.9646448768484071)
```

I/O with a compressed pickle

Compressed pickles can be saved easily. Simply treat them as binary streams:

```
>>> f = open("my_cpickle.bin", "wb")
>>> f.write(zpa)
>>> f.close()
>>> f = open("my_cpickle.bin", "rb")
>>> zpa = f.read()
>>> f.close()
```

Unpickling a compressed pickle

Just decompress it first:

```
>>> pa = zlib.decompress(zpa)
>>> a2 = cPickle.loads(pa)
>>> np.alltrue(a == a2)
True
```

Very easy!

Be sure to use compression if you are short of disk.

Resource consumption for solid compression

Memory: You need to book some spare memory to keep the compressed pickle.

CPU: Compressors consume quite a lot of it, but you may always find a compressor that fits your needs.

For example, for a pickle of `np.linspace(0, 100, 1e7)`:

(all compressors with level 9)	memcpy	blosc	zlib	bzip2
final size (MB)	76	7.7	50	55
compress throughput (MB/s)	3500	3600	4.8	4.5
decompress throughput (MB/s)	3500	3500	120	9.9

Hardware: 2 x Intel E5520 @ 2.27GHz, 8 MB third level cache.

Outline

- 1 The Basics
 - Introduction
 - Pickling our objects
 - Relational databases
- 2 Numerical Binary Formats
 - Why we need them?
 - The NPY format
 - The HDF5 format
 - The NetCDF4 format
- 3 **Adding Compression**
 - Why compression?
 - Solid compression
 - **Chunked compression**

Chunked compression

- Data is stored compressed in chunks (on-disk or in-memory) and chunks are decompressed when needed only.
- HDF5 / NetCDF4 support this paradigm.
- Saves disk and memory resources and may, in some situations, even accelerate the I/O speed.

Examples with PyTables/HDF5

- PyTables includes support for a fair number of compressors: Zlib, Bzip2, LZO and Blosc.
- It also supports “shuffle”, an interesting filter designed to improved compression ratios.
- You can choose whatever combination that proves to be more convenient for your needs.

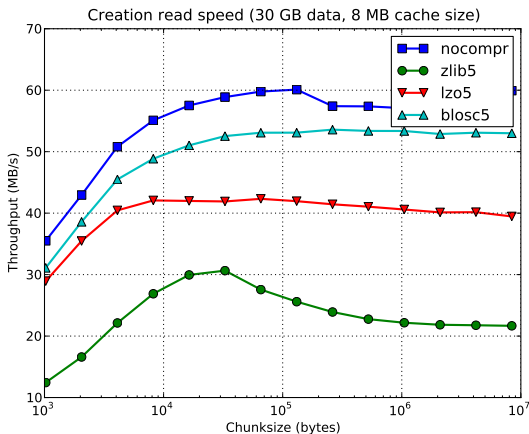
Querying compressed data

The dataset is a table with real data used in astronomy:

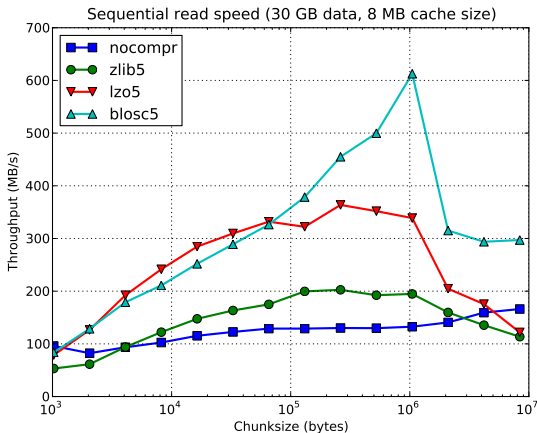
- 30 columns, most of them floating points and some ints
- Around 77000 entries
- Query: all entries where “ra” > 19 (3% selectivity)

(all compressors with level 5)	no compr	blosc	zlib	bzip2
table size (MB)	10	5.3	4.7	4.6
creation throughput (MB/s)	330	250	22	7.7
query throughput (MB/s)	170	140	38	10

Effect of chunked compression when writing large datasets



Effect of chunked compression when reading large datasets



When should you use compression?

- Your data has to be compressible (sparse matrices, time series, data with low entropy, ...).
- Whether your disk space is tight or your datasets are large.
- You want to optimize I/O speed.
- It's fun!

Summary

- Pickle is the most basic, but still powerful, way to serialize Python data. But it is mainly meant for small datasets and it is not portable.
- Relational databases are portable, mature and solid as a rock. However, they do not interact well with NumPy and write performance is pretty lame.
- HDF5 / NetCDF4 formats show best performance, Python APIs interacts well with NumPy and are extremely portable. They lack safety features.
- Using compression allows you to deal with more data using the same resources. In general, they can save I/O time to disk.

More Info



David Beazley

Python – Essential Reference

Addison-Wesley, 2009



Robert Kern

NPY: A Simple File Format for NumPy Arrays

NumPy Enhancement Proposal, December 2007

► The HDF Group

What is HDF5?

<http://www.hdfgroup.org/HDF5/whatishdf5.html>

Thank You!

Contact:

faltet@pytables.org