Data Persistence

From Pickle To Databases

Francesc Alted

PyTables Author
Barcelona Music and Audio Tecnology (BMAT)

Advanced Scientific Programming in Python
2011 Summer School, St Andrews, Scotland

Serialization vs Storage Solutions

- We follow the convention that **Serialization** is a way to make persistent data that fits in-memory.
- By **Storage Solutions** we mean ways to keep data on-disk, but without the in-memory limitation.

Sometimes the limits are fuzzy though!

Serialization vs Storage Solutions

- We follow the convention that **Serialization** is a way to make persistent data that fits in-memory.
- By **Storage Solutions** we mean ways to keep data on-disk, but without the in-memory limitation.

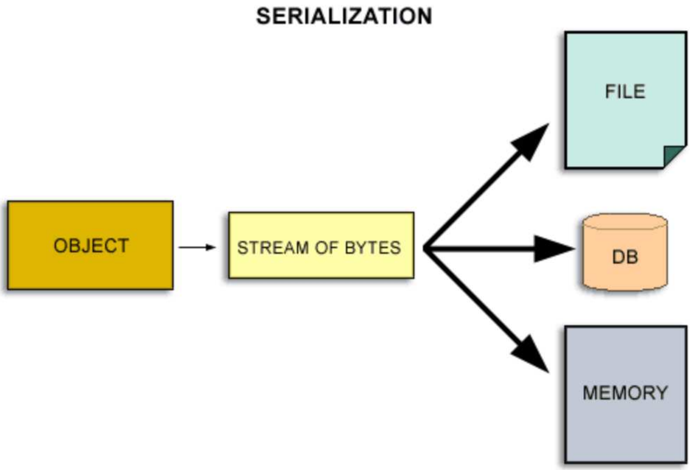
Sometimes the limits are fuzzy though!

Outline

1. Serialization tools
   - Serializing (pickling) general objects
   - The `shelve` module
   - Serializing NumPy objects

2. Storage solutions
   - Relational databases
   - Numerical binary formats: HDF5/NetCDF4
   - The PyTables database

What "Serialization" Means?



Serialization Tools

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

Outline

1. Serialization tools
   - Serializing (pickling) general objects
   - The shelve module
   - Serializing NumPy objects

2. Storage solutions
   - Relational databases
   - Numerical binary formats: HDF5/NetCDF4
   - The PyTables database

The pickle Module

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

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

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

The pickle Module

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

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

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

pickle Capabilities

- 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.

pickle Capabilities

- 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, it is a bit more restrictive (nothing grave).
- Python 3 `pickle` can use the C implementation transparently.

pickle/cPickle Limitations

- You need to reload all the data in the pickle before you can use any part of it. This is inconvenient for large datasets.
- Data can only be retrieved by other Python interpreters. You loose data portability with other languages.

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):

```
>>> A = ['hello',10,'world']
>>> %store A
>>> Exit
$ ipython
>>> print A
['hello', 10, 'world']
```

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):

```
>>> A = ['hello',10,'world']
>>> %store A
>>> Exit
$ ipython
>>> print A
['hello', 10, 'world']
```

Outline

1. Serialization tools
   - Serializing (pickling) general objects
   - The `shelve` module
   - Serializing NumPy objects

2. Storage solutions
   - Relational databases
   - Numerical binary formats: HDF5/NetCDF4
   - The PyTables database

The shelve Module

- Provides support for persitent 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()
# In another session
>>> db = shelve.open("database", "r")
>>> print db["one"]
1
>>> print db["three"]
3
```

Pros and Cons of the `shelve` Module

Pros

Easy to retrieve just a selected set of variables.

Specially useful for handling large series of pickles.

Cons

Suffers the same problems than `pickle`.

Outline

1. Serialization tools
   - Serializing (pickling) general objects
   - The shelve module
   - Serializing NumPy objects
2. Storage solutions
   - Relational databases
   - Numerical binary formats: HDF5/NetCDF4
   - The PyTables database

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`.

Pickling & Compression

```
>> time ap = cPickle.dumps(a, protocol=cPickle.HIGHEST_PROTOCOL)
CPU times:  user 0.03 s, sys:  0.07 s, total:  0.10 s
Wall time:  0.10 s

>> time apz = zlib.compress(ap)
CPU times:  user 4.68 s, sys:  0.02 s, total:  4.70 s
Wall time:  4.71 s

>> time apb = blosc.compress(ap, a.dtype.itemsize)
CPU times:  user 0.26 s, sys:  0.00 s, total:  0.26 s
Wall time:  0.03 s

>> len(ap)/1024., len(apz)/1024., len(apb)/1024.
(78125.1318359375, 51752.8623046875, 7455.8310546875)
```

Compresion can be a huge advantage, most specially with Blosc.

Outline

What Is a Relational Database?

- A relational database matches data by using common characteristics found within the data set.
- The resulting groups of data are organized and are much easier for many people to understand.

Example of the Relational Model

PubID	Publisher	PubAddress
03-4472822	Random House	123 4th Street, New York
04-7733903	Wiley and Sons	45 Lincoln Blvd, Chicago
03-4859223	O'Reilly Press	77 Boston Ave, Cambridge
03-3920886	City Lights Books	99 Market, San Francisco

AuthorID	AuthorName	AuthorBDay
345-28-2938	Haile Selassie	14-Aug-92
392-48-9965	Joe Blow	14-Mar-15
454-22-4012	Sally Hemmings	12-Sept-70
663-59-1254	Hannah Arendt	12-Mar-06

ISBN	AuthorID	PubID	Date	Title
1-34532-482-1	345-28-2938	03-4472822	1990	Cold Fusion for Dummies
1-38482-995-1	392-48-9965	04-7733903	1985	Macrame and Straw Tying
2-35921-499-4	454-22-4012	03-4859223	1952	Fluid Dynamics of Aquaducts
1-38278-293-4	663-59-1254	03-3920886	1967	Beads, Baskets & Revolution

Queries with the 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.

Code Example

```
mycursor.execute(
    "SELECT match_id from match_cleanmatch “
    "where cleanmatch_id = %s "
    " AND customer_id = %s",
    (cleanmatch_id, customer_id))
rows = self.cursor.fetchall()
mycursor.execute(
    "DELETE FROM cleanmatch_ where id = %s",
    (cleanmatch_id, ))
self.db.commit()
```

RDBMs Highlights

- 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 and updates are SLOOOW.
- Not very disk space efficient.
- Not well adapted to handle large numerical datasets (no direct interface with NumPy).

Outline

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.
- There are a really huge number of numerical formats depending on the needs.

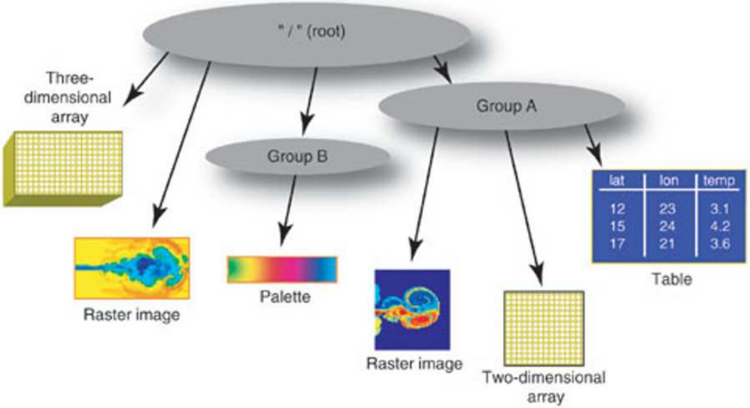
Why We Need a Binary Format?

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

Drawbacks of Binary Formats

- Lack of standarization (way too many formats out there). But some (HDF5, NetCDF4) are spreading a lot.
- Lack of security features (e.g. no ACID support). Performance is way more important.
- Easy to corrupt files under some conditions (e.g. power outage). Next version of HDF5 (1.10) will implement journaling so as to fix this.

HDF5: Hierachical Data Structures



NetCDF4

network Common Data Form v4

- NetCDF is a set of libraries and 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

Interfaces to binary formats (HDF5, NetCDF4):

- Interfaces to HDF5:
  - PyTables
  - h5py
- Interfaces to NetCDF4:
  - netcdf4-python
  - Scientific.IO.NetCDF

All these use NumPy as the default memory container for I/O.

Python Interfaces

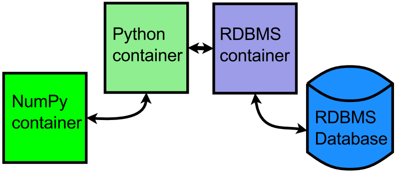
Interfaces to binary formats (HDF5, NetCDF4):

- Interfaces to HDF5:
  - PyTables
  - h5py
- Interfaces to NetCDF4:
  - netcdf4-python
  - Scientific.IO.NetCDF

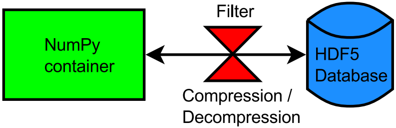
All these use NumPy as the default memory container for I/O.

Advantages of Using NumPy As Memory Container

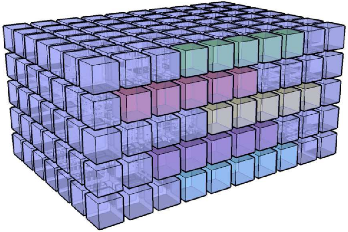
Interfaces for RDBMS in Python lacks support for direct NumPy containers (very inefficient!).



All of the Python interfaces mentioned before are using NumPy as default container.



Easing Disk Access Via NumPy Paradigm



- `array[1]`
- `array[3:1000, ..., :10]`
- `(array1**3 / array2) - sin(array3)` (PyTables)

There is a lot of value in adopting this paradigm: you don't need to learn another one!

Outline

Easy To Use

Natural naming

```
# access to file:/group1/table
table = file.root.group1.table
```

Support for generalized and fancy indexing

```
array[idx, start:stop, :, start:stop:step]    # hyperslicing
array[1, [1,5,10], ..., -1]      # sparse reads (since 2.2)
```

Support for efficient queries

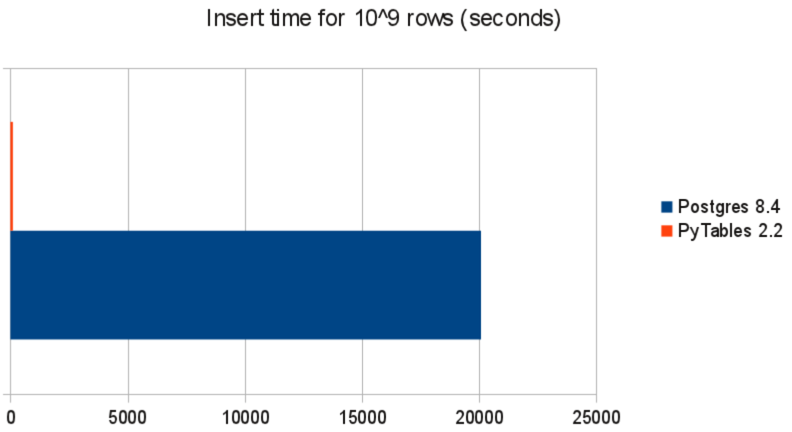
```
# get the values in col1 that satisfy a certain condition
[r['col1'] for r in table.where((1.3 < col3) & (col2 <= 2.))]
```

How PyTables Fights CPU Starvation?

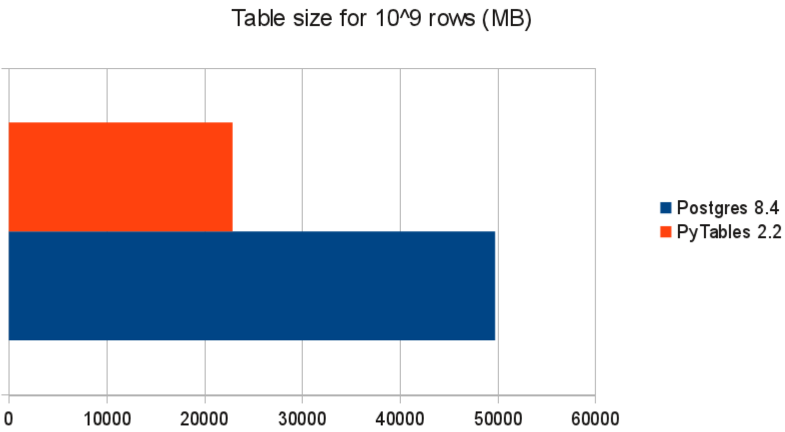
Basically, by applying blocking techniques and by leveraging high performance packages like:

- **HDF5** A library & format thought out for managing very large datasets in an efficient way.
- **NumPy** A Python package for handling large homogeneous and heterogeneous datasets.
- **Numexpr** Increase the performance of NumPy in complex operations by applying blocking.
- **Blosc** A high-performance compressor meant for binary data (available in the short future).

Advantages of Using HDF5 As Disk Container (I)

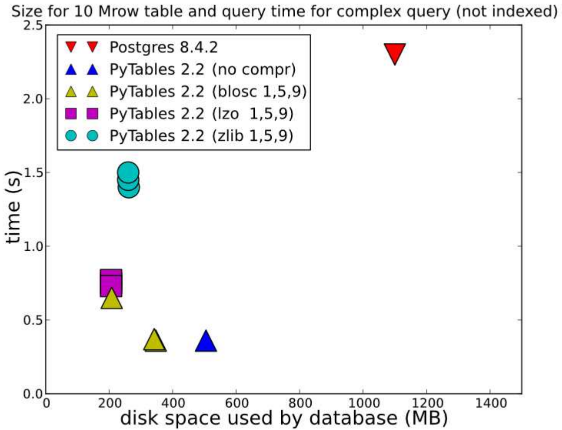


Advantages of Using HDF5 As Disk Container (II)



HDF5 + Numexpr + Blosc

Delivering extreme performance (while keeping disk requeriments low)



Advanced Capabilities in Forthcoming PyTables 2.3

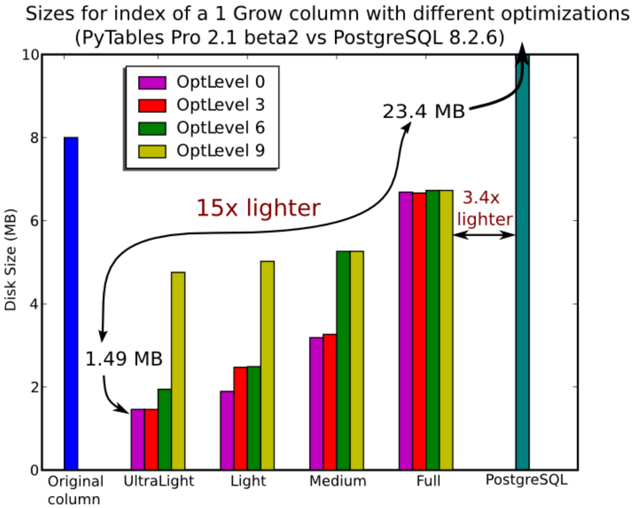
All the features in extinct PyTables Pro have been implemented in the next open source PyTables version:

Column indexing
: Queries in tables having up to 1 billion rows can be typically done in less than 1 second.

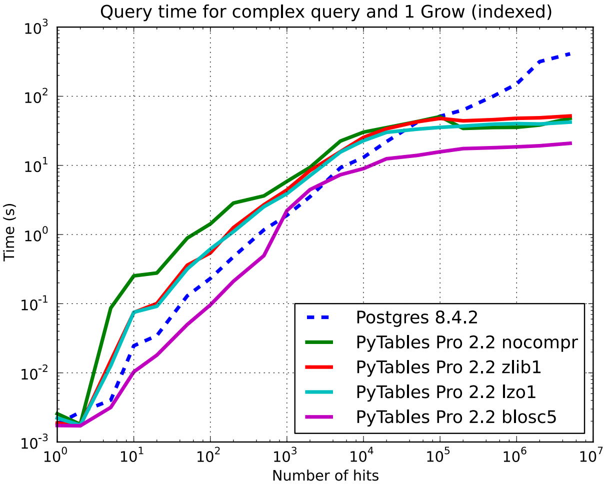
Customizable index quality
: The indexes can be created with an optimization level (specified as a number ranging from 0 to 9).

Improved cache system
: for both metadata and regular data. Allows for maximum speed during intensive node browsing and data queries.

Customizable Indexes



Indexed Query Performance



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.
- PyTables adds additional bells and whistles beyond HDF5 and NumPy capabilites: efficient queries, indexing and on-disk operations.

More Info

- David Beazley
  Python – Essential Reference
  4th edition
  Addisson-Wesley, 2009
- Alan Beaulieu
  Learning SQL
  2nd edition
  O'Reilly Media, 2009
- PyTables Governance Team
  PyTables: hierarchical datasets
  https://github.com/PyTables

Questions?

Contact:

faltet@pytables.org