HDF5 is for Lovers 

November 10th, 2012, PyConCa, Toronto, Ontario

Anthony Scopatz
The FLASH Center
The University of Chicago
scopatz@gmail.com

What is HDF5?

HDF5 stands for (**H**)eirarchical (**D**)ata (**F**)ormat (**5**)ive.

What is HDF5?

HDF5 stands for (**H**)eirarchical (**D**)ata (**F**)ormat (**5**)ive.

It is supported by the lovely people at HDF The HDF Group

What is HDF5?

HDF5 stands for (**H**)eirarchical (**D**)ata (**F**)ormat (**5**)ive.

It is supported by the lovely people at HDF The HDF Group

At its core HDF5 is binary file type specification.

What is HDF5?

HDF5 stands for (**H**)eirarchical (**D**)ata (**F**)ormat (**5**)ive.

It is supported by the lovely people at The HDF Group

At its core HDF5 is binary file type specification.

However, what makes HDF5 great is the numerous libraries written to interact with files of this type and their *extremely rich* feature set.

What is HDF5?

HDF5 stands for (**H**)eirarchical (**D**)ata (**F**)ormat (**5**)ive.

It is supported by the lovely people at HDF The HDF Group

At its core HDF5 is binary file type specification.

However, what makes HDF5 great is the numerous libraries written to interact with files of this type and their *extremely rich* feature set.

Which you will learn today!

A Note on the Format

Intermixed, there will be:

- Slides
- Interactive Hacking
- Exercises

A Note on the Format

Intermixed, there will be:

- Slides
- Interactive Hacking
- Exercises

Feel free to:

- Ask questions at anytime
- Explore at your own pace.

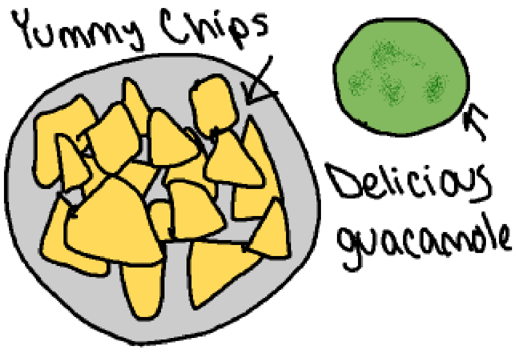
A Note on the Format

Allow me to recomend this tutorial with DWF's "Intro to Scientific Computing" tutorial. So...

A Note on the Format

Allow me to recomend this tutorial with DWF's "Intro to Scientific Computing" tutorial. So...

Get the Program Committee!



~go to that one too! (img http://mayorgia.blogspot.ca/2012/03/and-then-i-ate-worlds-best-burrito.html)

Also

Happy 10th Birthday, PyTables!

Also Also

Continuum Analytics has a promotion right now for PyConCa for 25% off of the professional version of Anaconda, their Python distribution.

Also Also

Continuum Analytics has a promotion right now for PyConCa for 25% off of the professional version of Anaconda, their Python distribution.

Go to https://store.continuum.io/cshop/anaconda and type "pycon-ca" at checkout.

Also Also

Continuum Analytics has a promotion right now for PyConCa for 25% off of the professional version of Anaconda, their Python distribution.

Go to https://store.continuum.io/cshop/anaconda and type "pycon-ca" at checkout.

It comes with HDF5 and PyTables (and h5py).

Class Makeup

By a show of hands, how many people have used:

- HDF5 before?

Class Makeup

By a show of hands, how many people have used:

- HDF5 before?
- PyTables?

Class Makeup

By a show of hands, how many people have used:

- HDF5 before?
- PyTables?
- h5py?

Class Makeup

By a show of hands, how many people have used:

- HDF5 before?
- PyTables?
- h5py?
- the HDF5 C API?

Class Makeup

By a show of hands, how many people have used:

- HDF5 before?
- PyTables?
- h5py?
- the HDF5 C API?
- SQL?

Class Makeup

By a show of hands, how many people have used:

- HDF5 before?
- PyTables?
- h5py?
- the HDF5 C API?
- SQL?
- Other binary data formats?

Setup

Please clone the repo:

```
git clone git://github.com/scopatz/hdf5-is-for-lovers.git
```

Or download a tarball from:

https://github.com/scopatz/hdf5-is-for-lovers

Warm up exercise

In IPython:

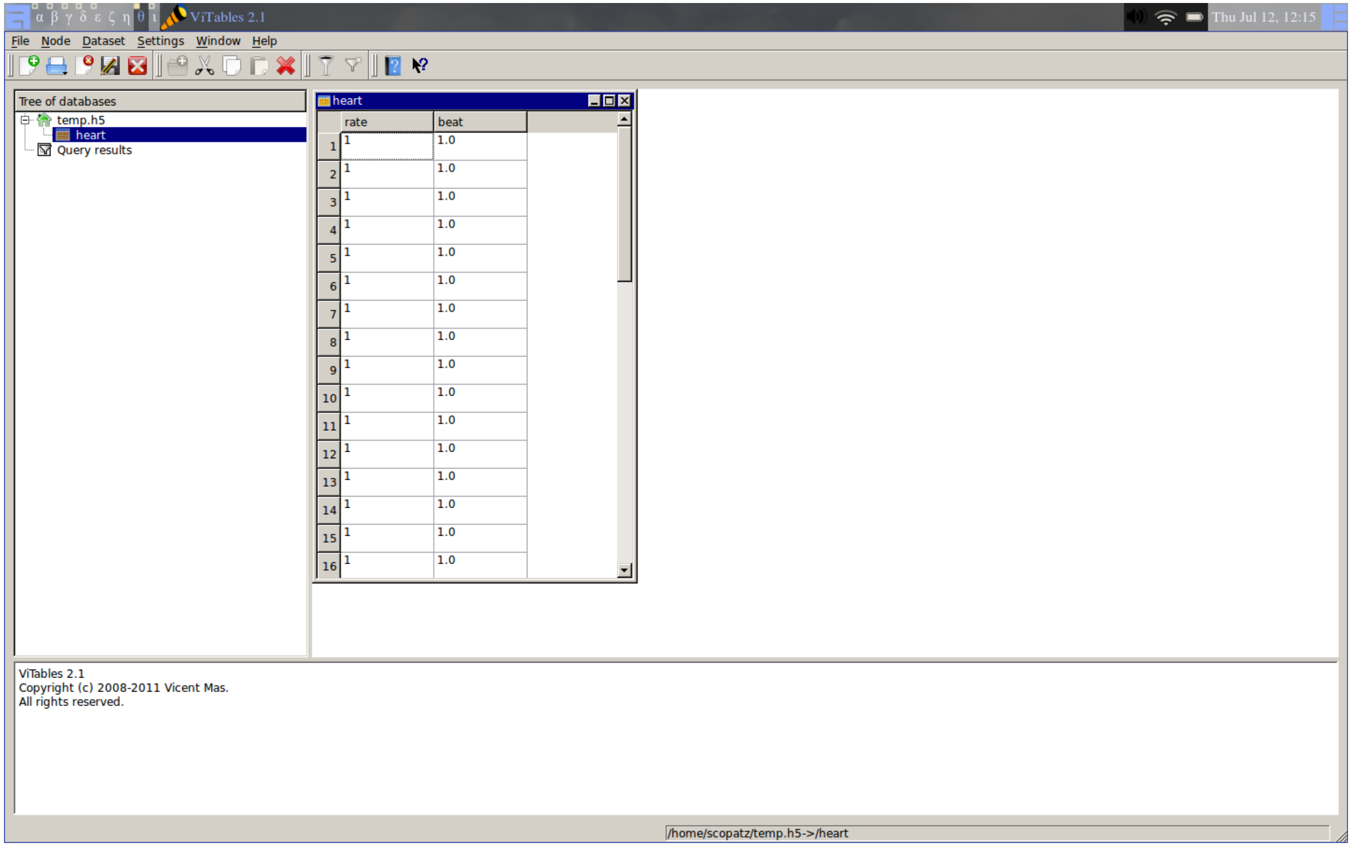
```
import numpy as np
import tables as tb

f = tb.openFile('temp.h5', 'a')
heart = np.ones(42, dtype=[('rate', int), ('beat', float)])
f.createTable('/', 'heart', heart)
f.close()
```

Or run `python exer/warmup.py`

Warm up exercise

You should see in ViTables:



A Brief Introduction

For persisting structured numerical data, binary formats are superior to plaintext.

A Brief Introduction

For persisting structured numerical data, binary formats are superior to plaintext.

For one thing, they are often smaller:

```
# small ints           # med ints
42    (4 bytes)        123456   (4 bytes)
'42'  (2 bytes)        '123456' (6 bytes)

# near-int floats      # e-notation floats
12.34   (8 bytes)      42.424242E+42   (8 bytes)
'12.34' (5 bytes)      '42.424242E+42' (13 bytes)
```

A Brief Introduction

For another, binary formats are often faster for I/O because `atoi()` and `atof()` are expensive.

A Brief Introduction

For another, binary formats are often faster for I/O because `atoi()` and `atof()` are expensive.

However, you often want some thing more than a binary chunk of data in a file.

A Brief Introduction

For another, binary formats are often faster for I/O because `atoi()` and `atof()` are expensive.

However, you often want some thing more than a binary chunk of data in a file.

> ***Note***
>
> This is the mechanism behind `numpy.save()` and `numpy.savez()`.

A Brief Introduction

Instead, you want a real *database* with the ability to store many datasets, user-defined metadata, optimized I/O, and the ability to query its contents.

A Brief Introduction

Instead, you want a real *database* with the ability to store many datasets, user-defined metadata, optimized I/O, and the ability to query its contents.

Unlike SQL, where every dataset lives in a flat namespace, HDF allows datasets to live in a nested tree structure.

A Brief Introduction

Instead, you want a real *database* with the ability to store many datasets, user-defined metadata, optimized I/O, and the ability to query its contents.

Unlike SQL, where every dataset lives in a flat namespace, HDF allows datasets to live in a nested tree structure.

In effect, HDF5 is a file system within a file.

A Brief Introduction

Instead, you want a real *database* with the ability to store many datasets, user-defined metadata, optimized I/O, and the ability to query its contents.

Unlike SQL, where every dataset lives in a flat namespace, HDF allows datasets to live in a nested tree structure.

In effect, HDF5 is a file system within a file.

(More on this later.)

A Brief Introduction

Basic dataset classes include:

- Array

A Brief Introduction

Basic dataset classes include:

- Array
- CArray (chunked array)

A Brief Introduction

Basic dataset classes include:

- Array
- CArray (chunked array)
- EArray (extendable array)

A Brief Introduction

Basic dataset classes include:

- Array
- CArray (chunked array)
- EArray (extendable array)
- VLArray (variable length array)

A Brief Introduction

Basic dataset classes include:

- Array
- CArray (chunked array)
- EArray (extendable array)
- VLArray (variable length array)
- Table (structured array w/ named fields)

A Brief Introduction

Basic dataset classes include:

- Array
- CArray (chunked array)
- EArray (extendable array)
- VLArray (variable length array)
- Table (structured array w/ named fields)

All of these must be composed of atomic types.

A Brief Introduction

There are six kinds of types supported by PyTables:

- bool: Boolean (true/false) types. 8 bits.

A Brief Introduction

There are six kinds of types supported by PyTables:

- bool: Boolean (true/false) types. 8 bits.
- int: Signed integer types. 8, 16, 32 (default) and 64 bits.

A Brief Introduction

There are six kinds of types supported by PyTables:

- bool: Boolean (true/false) types. 8 bits.
- int: Signed integer types. 8, 16, 32 (default) and 64 bits.
- uint: Unsigned integers. 8, 16, 32 (default) and 64 bits.

A Brief Introduction

There are six kinds of types supported by PyTables:

- bool: Boolean (true/false) types. 8 bits.
- int: Signed integer types. 8, 16, 32 (default) and 64 bits.
- uint: Unsigned integers. 8, 16, 32 (default) and 64 bits.
- float: Floating point types. 16, 32 and 64 (default) bits.

A Brief Introduction

There are six kinds of types supported by PyTables:

- bool: Boolean (true/false) types. 8 bits.
- int: Signed integer types. 8, 16, 32 (default) and 64 bits.
- uint: Unsigned integers. 8, 16, 32 (default) and 64 bits.
- float: Floating point types. 16, 32 and 64 (default) bits.
- complex: Complex number. 64 and 128 (default) bits.

A Brief Introduction

There are six kinds of types supported by PyTables:

- bool: Boolean (true/false) types. 8 bits.
- int: Signed integer types. 8, 16, 32 (default) and 64 bits.
- uint: Unsigned integers. 8, 16, 32 (default) and 64 bits.
- float: Floating point types. 16, 32 and 64 (default) bits.
- complex: Complex number. 64 and 128 (default) bits.
- string: Raw string types. 8-bit positive multiples.

A Brief Introduction

Other elements of the hierarchy may include:

- Groups (dirs)

A Brief Introduction

Other elements of the hierarchy may include:

- Groups (dirs)
- Links

A Brief Introduction

Other elements of the hierarchy may include:

- Groups (dirs)
- Links
- File Nodes

A Brief Introduction

Other elements of the hierarchy may include:

- Groups (dirs)
- Links
- File Nodes
- Hidden Nodes

A Brief Introduction

Other elements of the hierarchy may include:

- Groups (dirs)
- Links
- File Nodes
- Hidden Nodes

PyTables docs may be found at http://pytables.github.com/

Opening Files

```
import tables as tb
f = tb.openFile('/path/to/file', 'a')
```

Opening Files

```
import tables as tb
f = tb.openFile('/path/to/file', 'a')
```

- *'r'*: Read-only; no data can be modified.
- *'w'*: Write; a new file is created (an existing file with the same name would be deleted).
- *'a'*: Append; an existing file is opened for reading and writing, and if the file does not exist it is created.
- *'r+'*: It is similar to 'a', but the file must already exist.

Using the Hierarchy

In HDF5, all nodes stem from a root ("`/`" or `f.root`).

Using the Hierarchy

In HDF5, all nodes stem from a root (`"/"` or `f.root`).

In PyTables, you may access nodes as attributes on a Python object (`f.root.a_group.some_data`).

Using the Hierarchy

In HDF5, all nodes stem from a root (`"/"` or `f.root`).

In PyTables, you may access nodes as attributes on a Python object (`f.root.a_group.some_data`).

This is known as natural naming.

Using the Hierarchy

In HDF5, all nodes stem from a root ("`/`" or `f.root`).

In PyTables, you may access nodes as attributes on a Python object (`f.root.a_group.some_data`).

This is known as natural naming.

Creating new nodes must be done on the file handle:

```
f.createGroup('/', 'a_group', "My Group")
f.root.a_group
```

Creating Datasets

The two most common datasets are Tables & Arrays.

Creating Datasets

The two most common datasets are Tables & Arrays.

Appropriate create methods live on the file handle:

```
# integer array
f.createArray('/a_group', 'arthur_count', [1, 2, 5, 3])
```

Creating Datasets

The two most common datasets are Tables & Arrays.

Appropriate create methods live on the file handle:

```
# integer array
f.createArray('/a_group', 'arthur_count', [1, 2, 5, 3])
```

```
# tables, need descriptions
dt = np.dtype([('id', int), ('name', 'S10')])
knights = np.array([(42, 'Lancelot'), (12, 'Bedivere')], dtype=dt)
f.createTable('/', 'knights', dt)
f.root.knights.append(knights)
```

Reading Datasets

Arrays and Tables try to preserve the original flavor that they were created with.

Reading Datasets

Arrays and Tables try to preserve the original flavor that they were created with.

```
>>> print f.root.a_group.arthur_count[:]
[1, 2, 5, 3]

>>> type(f.root.a_group.arthur_count[:])
list

>>> type(f.root.a_group.arthur_count)
tables.array.Array
```

Reading Datasets

So if they come from NumPy arrays, they may be accessed in a numpy-like fashion (slicing, fancy indexing, masking).

Reading Datasets

So if they come from NumPy arrays, they may be accessed in a numpy-like fashion (slicing, fancy indexing, masking).

```
>>> f.root.knights[1]
(12, 'Bedivere')

>>> f.root.knights[:1]
array([(42, 'Lancelot')], dtype=[('id', '<i8'), ('name', 'S10')])

>>> mask = (f.root.knights.cols.id[:] < 28)
>>> f.root.knights[mask]
array([(12, 'Bedivere')], dtype=[('id', '<i8'), ('name', 'S10')])

>>> f.root.knights[([1, 0],)]
array([(12, 'Bedivere'), (42, 'Lancelot')], dtype=[('id', '<i8'), ('name', 'S10')])
```

Reading Datasets

So if they come from NumPy arrays, they may be accessed in a numpy-like fashion (slicing, fancy indexing, masking).

```
>>> f.root.knights[1]
(12, 'Bedivere')

>>> f.root.knights[:1]
array([(42, 'Lancelot')], dtype=[('id', '<i8'), ('name', 'S10')])

>>> mask = (f.root.knights.cols.id[:] < 28)
>>> f.root.knights[mask]
array([(12, 'Bedivere')], dtype=[('id', '<i8'), ('name', 'S10')])

>>> f.root.knights[([1, 0],)]
array([(12, 'Bedivere'), (42, 'Lancelot')], dtype=[('id', '<i8'), ('name', 'S10')])
```

Data accessed in this way is *memory mapped*.

Exercise

exer/peaks_of_kilimanjaro.py

Exercise

sol/peaks_of_kilimanjaro.py

Hierarchy Layout

Suppose there is a big table of like-things:

```
# people:  name,            profession,    home
people = [('Arthur',        'King',        'Camelot'),
          ('Lancelot',      'Knight',      'Lake'),
          ('Bedevere',      'Knight',      'Wales'),
          ('Witch',         'Witch',       'Village'),
          ('Guard',         'Man-at-Arms', 'Swamp Castle'),
          ('Ni',            'Knight',      'Shrubbery'),
          ('Strange Woman', 'Lady',        'Lake'),
          ...
          ]
```

Hierarchy Layout

Suppose there is a big table of like-things:

```
# people:  name,                 profession,     home
people = [('Arthur',             'King',         'Camelot'),
          ('Lancelot',           'Knight',       'Lake'),
          ('Bedevere',           'Knight',       'Wales'),
          ('Witch',              'Witch',        'Village'),
          ('Guard',              'Man-at-Arms',  'Swamp Castle'),
          ('Ni',                 'Knight',       'Shrubbery'),
          ('Strange Woman',      'Lady',         'Lake'),
          ...
          ]
```

It is tempting to throw everyone into a big `people` table.

Hierarchy Layout

However, a search over a class of people can be eliminated by splitting these tables up:

```
knight = [('Lancelot',       'Knight',       'Lake'),
          ('Bedevere',       'Knight',       'Wales'),
          ('Ni',             'Knight',       'Shrubbery'),
          ]

others = [('Arthur',         'King',         'Camelot'),
          ('Witch',          'Witch',        'Village'),
          ('Guard',          'Man-at-Arms', 'Swamp Castle'),
          ('Strange Woman', 'Lady',          'Lake'),
          ...
          ]
```

Hierarchy Layout

The profession column is now redundant:

```
knight = [('Lancelot', 'Lake'),
          ('Bedevere', 'Wales'),
          ('Ni',       'Shrubbery'),
          ]

others = [('Arthur',        'King',        'Camelot'),
          ('Witch',         'Witch',       'Village'),
          ('Guard',         'Man-at-Arms', 'Swamp Castle'),
          ('Strange Woman', 'Lady',        'Lake'),
          ...
          ]
```

Hierarchy Layout

Information can be embedded implicitly in the hierarchy as well:

```
root
  | - England
  |       | - knight
  |       | - others
  |
  | - France
  |       | - knight
  |       | - others
```

Hierarchy Layout

Why bother pivoting the data like this at all?

Hierarchy Layout

Why bother pivoting the data like this at all?

- Fewer rows to search over.

Hierarchy Layout

Why bother pivoting the data like this at all?

- Fewer rows to search over.
- Fewer rows to pull from disk.

Hierarchy Layout

Why bother pivoting the data like this at all?

- Fewer rows to search over.
- Fewer rows to pull from disk.
- Fewer columns in description.

Hierarchy Layout

Why bother pivoting the data like this at all?

- Fewer rows to search over.
- Fewer rows to pull from disk.
- Fewer columns in description.

Ultimately, it is all about *speed*, especially for big tables.

Access Time Analogy

If a processor's access of L1 cache is analogous to you finding a word on a computer screen (3 seconds), then

Access Time Analogy

If a processor's access of L1 cache is analogous to you finding a word on a computer screen (3 seconds), then

Accessing L2 cache is getting a book from a bookshelf (15 s).

Access Time Analogy

If a processor's access of L1 cache is analogous to you finding a word on a computer screen (3 seconds), then

Accessing L2 cache is getting a book from a bookshelf (15 s).

Accessing main memory is going to the break room, get a candy bar, and chatting with your co-worker (4 min).

Access Time Analogy

If a processor's access of L1 cache is analogous to you finding a word on a computer screen (3 seconds), then

Accessing L2 cache is getting a book from a bookshelf (15 s).

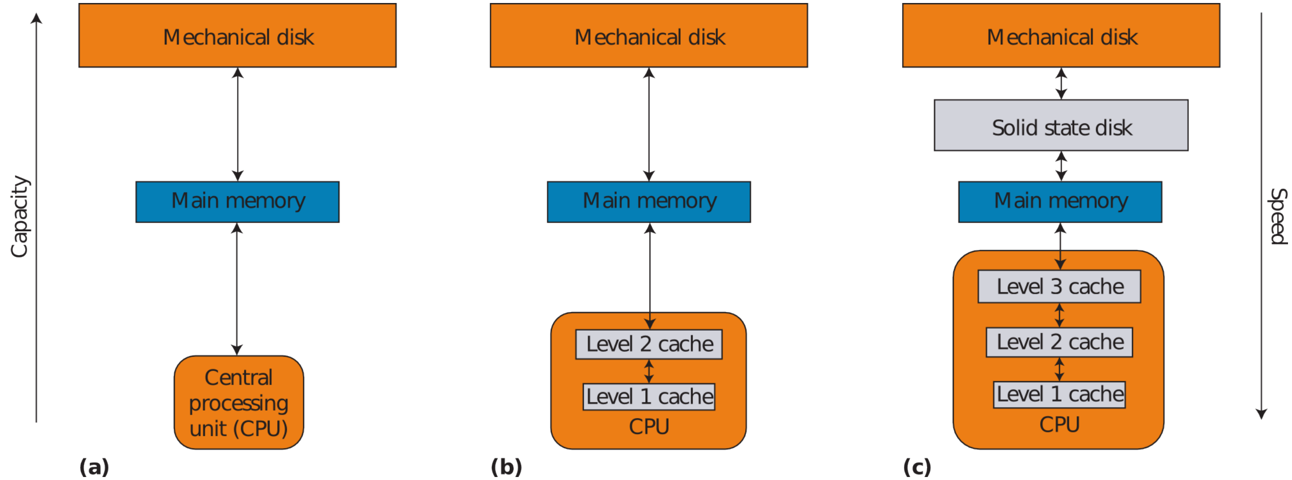
Accessing main memory is going to the break room, get a candy bar, and chatting with your co-worker (4 min).

Accessing a (mechanical) HDD is leaving your office, leaving your building, wandering the planet for a year and four months to return to your desk with the information finally made available.

Thanks K. Smith & http://duartes.org/gustavo/blog/post/what-your-computer-does-while-you-wait

Starving CPU Problem

Waiting around for access times prior to computation is known as the *Starving CPU Problem*.



Francesc Alted. 2010. Why Modern CPUs Are Starving and What Can Be Done about It. IEEE Des. Test 12, 2 (March 2010), 68-71. DOI=10.1109/MCSE.2010.51
http://dx.doi.org/10.1109/MCSE.2010.51

Tables

Tables are a high-level interface to extendable arrays of structs.

Tables

Tables are a high-level interface to extendable arrays of structs.

Sort-of.

Tables

Tables are a high-level interface to extendable arrays of structs.

Sort-of.

In fact, the struct / dtype / description concept is only a convenient way to assign meaning to bytes:

```
|  ids  |       first       |        last       |
|-------|-------------------|-------------------|
| | | | | | | | | | | | | | | | | | | | | | | | |
```

Tables

Data types may be nested (though they are stored in flattened way).

```
dt = np.dtype([('id', int),
               ('first', 'S5'),
               ('last',  'S5'),
               ('parents', [
                    ('mom_id', int),
                    ('dad_id', int),
                ]),
              ])

people = np.fromstring(np.random.bytes(dt.itemsize * 10000), dt)
f.createTable('/', 'random_peeps', people)
```

Tables

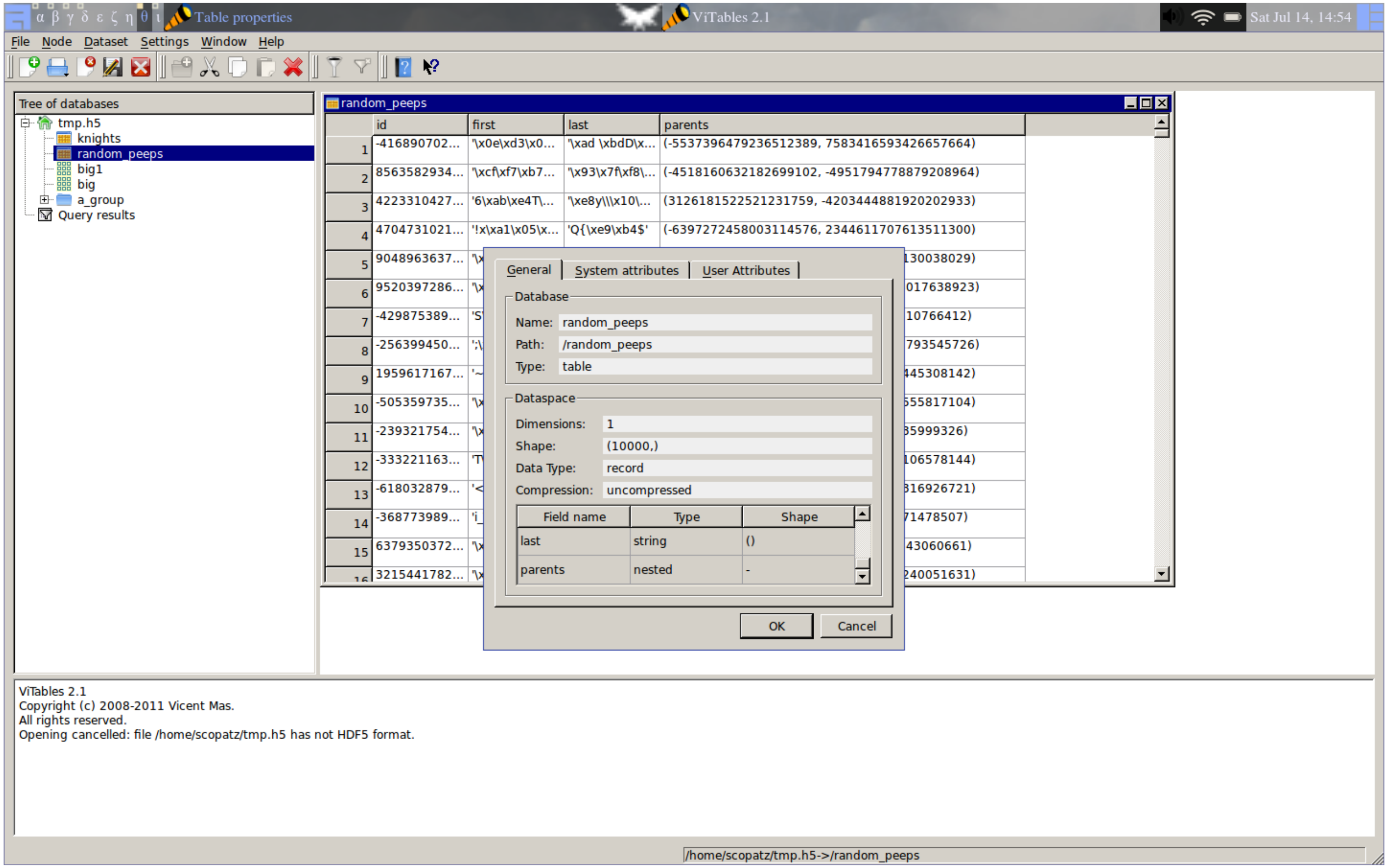


Table properties
ViTables 2.1
Sat Jul 14, 14:54
File Node Dataset Settings Window Help
Tree of databases
tmp.h5
knights
random_peeps
big1
big
a_group
Query results
random_peeps
id
first
last
parents
General
System attributes
User Attributes
Database
Name: random_peeps
Path: /random_peeps
Type: table
Dataspace
Dimensions: 1
Shape: (10000,)
Data Type: record
Compression: uncompressed
Field name
Type
Shape
last
string
()
parents
nested
-
OK
Cancel
ViTables 2.1
Copyright (c) 2008-2011 Vicent Mas.
All rights reserved.
Opening cancelled: file /home/scopatz/tmp.h5 has not HDF5 format.
/home/scopatz/tmp.h5->/random_peeps

Tables

Python already has the ability to dynamically declare the size of descriptions.

Tables

Python already has the ability to dynamically declare the size of descriptions.

This is accomplished in compiled languages through normal memory allocation and careful byte counting:

```
typedef struct mat {
  double mass;
  int atoms_per_mol;
  double comp [];
} mat;
```

Tables

```
typedef struct mat {
  double mass;
  int atoms_per_mol;
  double comp [];
} mat;

size_t mat_size = sizeof(mat) + sizeof(double)*comp_size;
hid_t desc = H5Tcreate(H5T_COMPOUND, mat_size);
hid_t comptype = H5Tarray_create2(H5T_NATIVE_DOUBLE, 1, nuc_dims);

// make the data table type
H5Tinsert(desc, "mass", HOFFSET(mat, mass), H5T_NATIVE_DOUBLE);
H5Tinsert(desc, "atoms_per_mol", HOFFSET(mat, atoms_per_mol), H5T_NATIVE_DOUBLE);
H5Tinsert(desc, "comp", HOFFSET(mat, comp), comp_type);

// make the data array for a single row, have to over-allocate
mat * mat_data  = new mat[mat_size];

// ...fill in data array...

// Write the row
H5Dwrite(data_set, desc, mem_space, data_hyperslab, H5P_DEFAULT, mat_data);
```

Exercise

exer/boatload.py

Exercise

sol/boatload.py

Chunking

Chunking is a feature with no direct analogy in NumPy.

Chunking

Chunking is a feature with no direct analogy in NumPy.

Chunking is the ability to split up a dataset into smaller blocks of equal or lesser rank.

Chunking

Chunking is a feature with no direct analogy in NumPy.

Chunking is the ability to split up a dataset into smaller blocks of equal or lesser rank.

Extra metadata pointing to the location of the chunk in the file and in dataspace must be stored.

Chunking

Chunking is a feature with no direct analogy in NumPy.

Chunking is the ability to split up a dataset into smaller blocks of equal or lesser rank.

Extra metadata pointing to the location of the chunk in the file and in dataspace must be stored.

By chunking, sparse data may be stored efficiently and datasets may extend infinitely in all dimensions.

Chunking

Chunking is a feature with no direct analogy in NumPy.

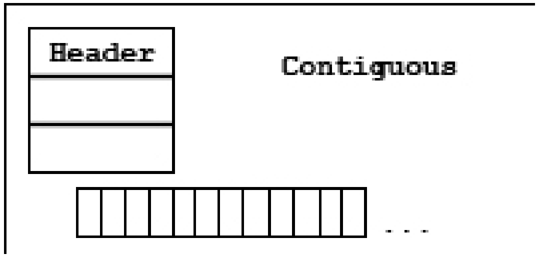
Chunking is the ability to split up a dataset into smaller blocks of equal or lesser rank.

Extra metadata pointing to the location of the chunk in the file and in dataspace must be stored.

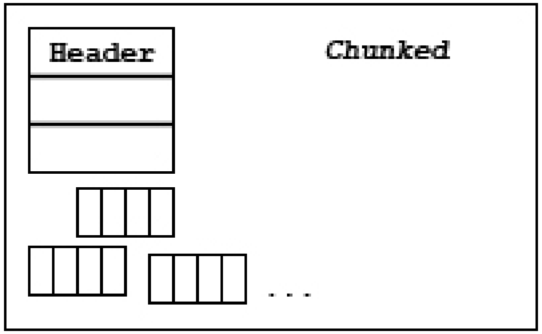
By chunking, sparse data may be stored efficiently and datasets may extend infinitely in all dimensions.

Note: Currently, PyTables only allows one extendable dim.

Chunking



Contiguous Dataset



Chunked Dataset

Chunking

All I/O happens by chunk. This is important for:

- edge chunks may extend beyond the dataset

Chunking

All I/O happens by chunk. This is important for:

- edge chunks may extend beyond the dataset
- default fill values are set in unallocated space

Chunking

All I/O happens by chunk. This is important for:

- edge chunks may extend beyond the dataset
- default fill values are set in unallocated space
- reading and writing in parallel

Chunking

All I/O happens by chunk. This is important for:

- edge chunks may extend beyond the dataset
- default fill values are set in unallocated space
- reading and writing in parallel
- small chunks are good for accessing some of data

Chunking

All I/O happens by chunk. This is important for:

- edge chunks may extend beyond the dataset
- default fill values are set in unallocated space
- reading and writing in parallel
- small chunks are good for accessing some of data
- large chunks are good for accessing lots of data

Chunking

Any chunked dataset allows you to set the chunksize.

```
f.createTable('/', 'omnomnom', data, chunkshape=(42,42))
```

Chunking

Any chunked dataset allows you to set the chunksize.

```
f.createTable('/', 'omnomnom', data, chunkshape=(42,42))
```

For example, a 4x4 chunked array could have a 3x3 chunksize.

Chunking

Any chunked dataset allows you to set the chunksize.

```
f.createTable('/', 'omnomnom', data, chunkshape=(42,42))
```

For example, a 4x4 chunked array could have a 3x3 chunksize.

However, it could not have a 12x12 chunksize, since the ranks must be less than or equal to that of the array.

Chunking

Any chunked dataset allows you to set the chunksize.

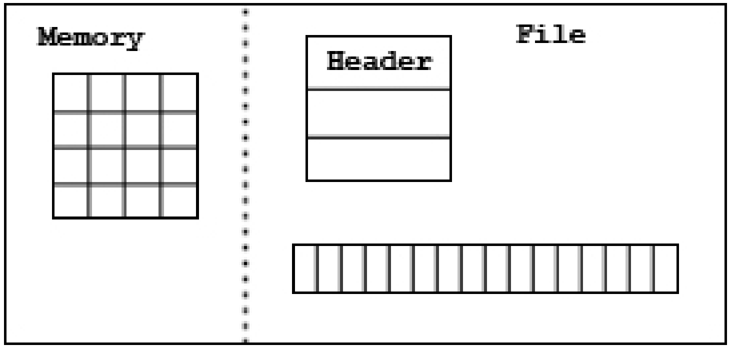
```
f.createTable('/', 'omnomnom', data, chunkshape=(42,42))
```

For example, a 4x4 chunked array could have a 3x3 chunksize.

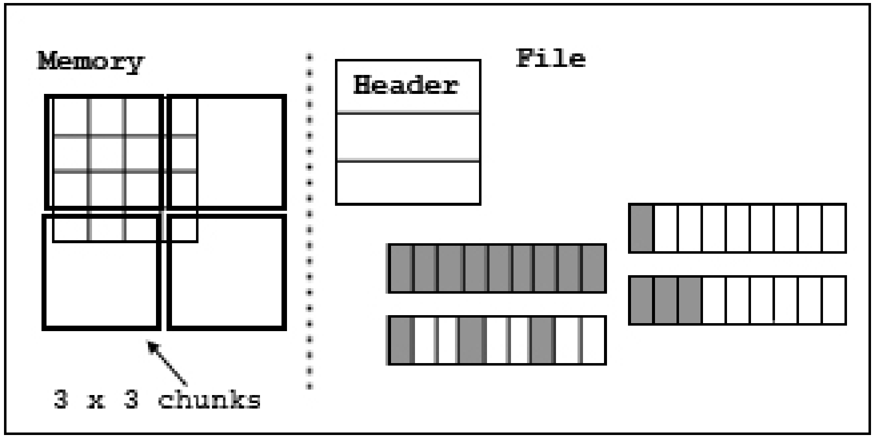
However, it could not have a 12x12 chunksize, since the ranks must be less than or equal to that of the array.

Manipulating the chunksize is a great way to fine-tune an application.

Chunking



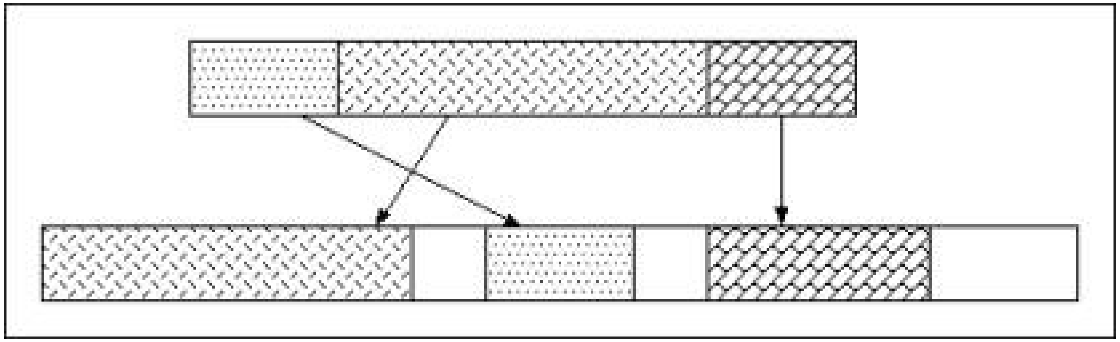
Contiguous 4x4 Dataset



Chunked 4x4 Dataset

Chunking

Note that the addresses of chunks in dataspace (memory) has no bearing on their arrangement in the actual file.



Dataspace (top) vs File (bottom) Chunk Locations

In-Core vs Out-of-Core

Calculations depend on the current memory layout.

In-Core vs Out-of-Core

Calculations depend on the current memory layout.

Recall access time analogy (wander Earth for 16 months).

In-Core vs Out-of-Core

Calculations depend on the current memory layout.

Recall access time analogy (wander Earth for 16 months).

Definitions:

In-Core vs Out-of-Core

Calculations depend on the current memory layout.

Recall access time analogy (wander Earth for 16 months).

Definitions:

- Operations which require all data to be in memory are *in-core* and may be memory bound (NumPy).

In-Core vs Out-of-Core

Calculations depend on the current memory layout.

Recall access time analogy (wander Earth for 16 months).

Definitions:

- Operations which require all data to be in memory are *in-core* and may be memory bound (NumPy).
- Operations where the dataset is external to memory are *out-of-core* (or *in-kernel*) and may be CPU bound.

In-Core Operations

Say, `a` and `b` are arrays sitting in memory:

```
a = np.array(...)
b = np.array(...)
c = 42 * a + 28 * b + 6
```

In-Core Operations

Say, `a` and `b` are arrays sitting in memory:

```
a = np.array(...)
b = np.array(...)
c = 42 * a + 28 * b + 6
```

The expression for `c` creates three temporary arrays!

In-Core Operations

Say, `a` and `b` are arrays sitting in memory:

```
a = np.array(...)
b = np.array(...)
c = 42 * a + 28 * b + 6
```

The expression for `c` creates three temporary arrays!

For `N` operations, `N-1` temporaries are made.

In-Core Operations

Say, `a` and `b` are arrays sitting in memory:

```
a = np.array(...)
b = np.array(...)
c = 42 * a + 28 * b + 6
```

The expression for `c` creates three temporary arrays!

For `N` operations, `N-1` temporaries are made.

Wastes memory and is slow. Pulling from disk is slower.

In-Core Operations

A less memory intensive implementation would be an element-wise evaluation:

```
c = np.empty(...)
for i in range(len(c)):
    c[i] = 42 * a[i] + 28 * b[i] + 6
```

In-Core Operations

A less memory intensive implementation would be an element-wise evaluation:

```
c = np.empty(...)
for i in range(len(c)):
    c[i] = 42 * a[i] + 28 * b[i] + 6
```

But if `a` and `b` were HDF5 arrays on disk, individual element access time would kill you.

In-Core Operations

A less memory intensive implementation would be an element-wise evaluation:

```
c = np.empty(...)
for i in range(len(c)):
    c[i] = 42 * a[i] + 28 * b[i] + 6
```

But if `a` and `b` were HDF5 arrays on disk, individual element access time would kill you.

Even with in memory NumPy arrays, there are problems with gratuitous Python type checking.

Out-of-Core Operations

Say there was a virtual machine (or kernel) which could be fed arrays and perform specified operations.

Out-of-Core Operations

Say there was a virtual machine (or kernel) which could be fed arrays and perform specified operations.

Giving this machine only chunks of data at a time, it could function on infinite-length data using only finite memory.

Out-of-Core Operations

Say there was a virtual machine (or kernel) which could be fed arrays and perform specified operations.

Giving this machine only chunks of data at a time, it could function on infinite-length data using only finite memory.

```
for i in range(0, len(a), 256):
    r0, r1 = a[i:i+256], b[i:i+256]
    multiply(r0, 42, r2)
    multiply(r1, 28, r3)
    add(r2, r3, r2); add(r2,  6, r2)
    c[i:i+256] = r2
```

Out-of-Core Operations

This is the basic idea behind numexpr, which provides a general virtual machine for NumPy arrays.

Out-of-Core Operations

This is the basic idea behind numexpr, which provides a general virtual machine for NumPy arrays.

This problem lends itself nicely to parallelism.

Out-of-Core Operations

This is the basic idea behind numexpr, which provides a general virtual machine for NumPy arrays.

This problem lends itself nicely to parallelism.

Numexpr has low-level multithreading, avoiding the GIL.

Out-of-Core Operations

This is the basic idea behind numexpr, which provides a general virtual machine for NumPy arrays.

This problem lends itself nicely to parallelism.

Numexpr has low-level multithreading, avoiding the GIL.

PyTables implements a `tb.Expr` class which backends to the numexpr VM but has additional optimizations for disk reading and writing.

Out-of-Core Operations

This is the basic idea behind numexpr, which provides a general virtual machine for NumPy arrays.

This problem lends itself nicely to parallelism.

Numexpr has low-level multithreading, avoiding the GIL.

PyTables implements a `tb.Expr` class which backends to the numexpr VM but has additional optimizations for disk reading and writing.

The full array need never be in memory.

Out-of-Core Operations

Fully out-of-core expression example:

```
shape = (10, 10000)
f = tb.openFile("/tmp/expression.h5", "w")

a = f.createCArray(f.root, 'a', tb.Float32Atom(dflt=1.), shape)
b = f.createCArray(f.root, 'b', tb.Float32Atom(dflt=2.), shape)
c = f.createCArray(f.root, 'c', tb.Float32Atom(dflt=3.), shape)
out = f.createCArray(f.root, 'out', tb.Float32Atom(dflt=3.), shape)

expr = tb.Expr("a*b+c")
expr.setOutput(out)
d = expr.eval()

print "returned-->", repr(d)
f.close()
```

Querying

The most common operation is asking an existing dataset whether its elements satisfy some criteria. This is known as *querying*.

Querying

The most common operation is asking an existing dataset whether its elements satisfy some criteria. This is known as *querying*.

Because querying is so common PyTables defines special methods on Tables.

Querying

The most common operation is asking an existing dataset whether its elements satisfy some criteria. This is known as *querying*.

Because querying is so common PyTables defines special methods on Tables.

```
tb.Table.where(cond)
tb.Table.getWhereList(cond)
tb.Table.readWhere(cond)
tb.Table.whereAppend(dest, cond)
```

Querying

The conditions used in `where()` calls are strings which are evaluated by numexpr. These expressions must return boolean values.

Querying

The conditions used in `where()` calls are strings which are evaluated by numexpr. These expressions must return boolean values.

They are executed in the context of table itself combined with `locals()` and `globals()`.

Querying

The conditions used in `where()` calls are strings which are evaluated by numexpr. These expressions must return boolean values.

They are executed in the context of table itself combined with `locals()` and `globals()`.

The `where()` method itself returns an iterator over all matched (hit) rows:

```
for row in table.where('(col1 < 42) & (col2 == col3)'):
    # do something with row
```

Querying

For a speed comparison, here is a complex query using regular Python:

```
result = [row['col2'] for row in table if (
          ((row['col4'] >= lim1 and row['col4'] < lim2) or
          ((row['col2'] > lim3 and row['col2'] < lim4])) and
          ((row['col1']+3.1*row['col2']+row['col3']*row['col4']) > lim5)
          )]
```

Querying

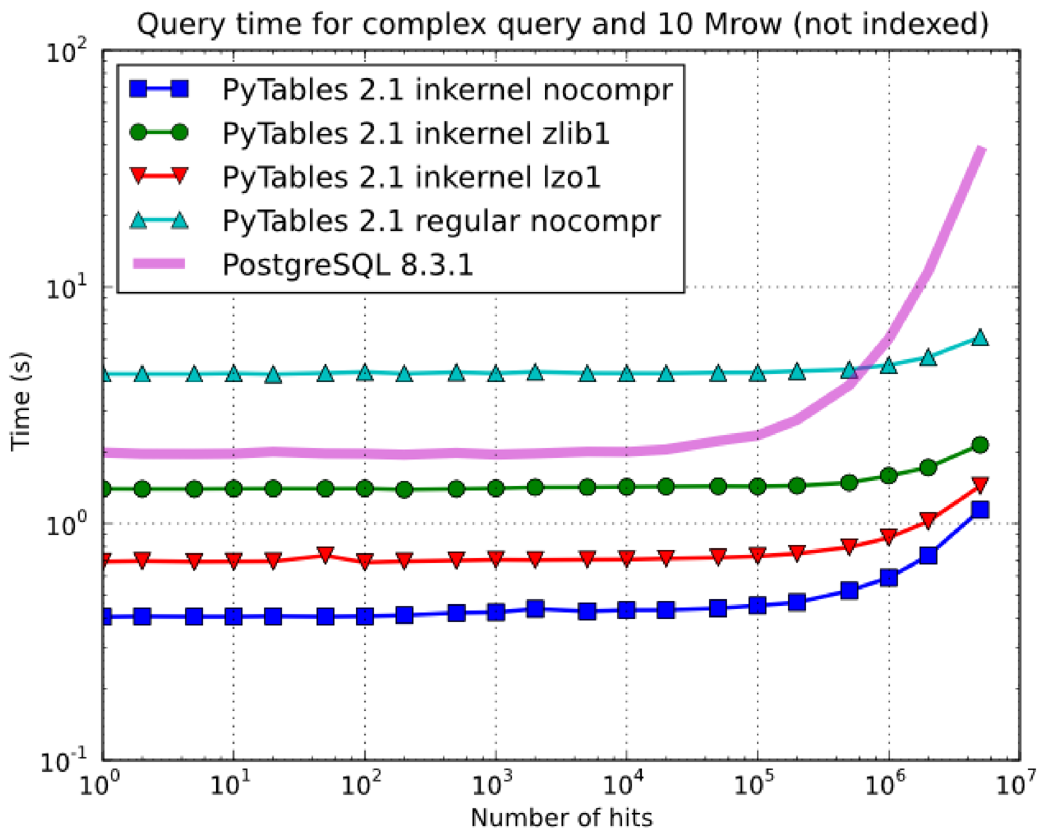
For a speed comparison, here is a complex query using regular Python:

```
result = [row['col2'] for row in table if (
          ((row['col4'] >= lim1 and row['col4'] < lim2) or
          ((row['col2'] > lim3 and row['col2'] < lim4])) and
          ((row['col1']+3.1*row['col2']+row['col3']*row['col4']) > lim5)
          )]
```

And this is the equivalent out-of-core search:

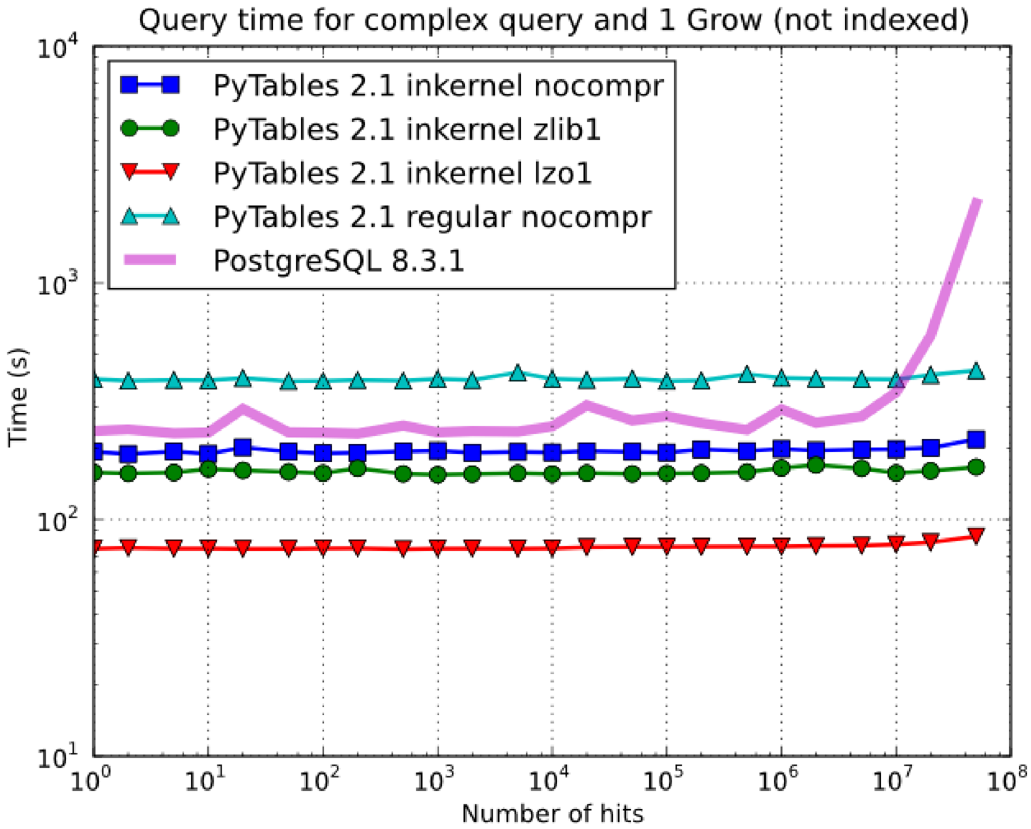
```
result = [row['col2'] for row in table.where(
            '(((col4 >= lim1) & (col4 < lim2)) | '
            '((col2 > lim3) & (col2 < lim4)) &   '
            '((col1+3.1*col2+col3*col4) > lim5)) ')]
```

Querying



Complex query with 10 million rows. Data fits in memory.

Querying



Complex query with 1 billion rows. Too big for memory.

Exercise

exer/crono.py

Exercise

sol/crono.py

Compression

A more general way to solve the starving CPU problem is through *compression*.

Compression

A more general way to solve the starving CPU problem is through *compression*.

Compression is when the dataset is piped through a zipping algorithm on write and the inverse unzipping algorithm on read.

Compression

A more general way to solve the starving CPU problem is through *compression*.

Compression is when the dataset is piped through a zipping algorithm on write and the inverse unzipping algorithm on read.

Each chunk is compressed independently, so chunks end up with a varying number bytes.

Compression

A more general way to solve the starving CPU problem is through *compression*.

Compression is when the dataset is piped through a zipping algorithm on write and the inverse unzipping algorithm on read.

Each chunk is compressed independently, so chunks end up with a varying number bytes.

Has some storage overhead, but may drastically reduce file sizes for very regular data.

Compression

At first glance this is counter-intuitive. (*Why?*)

Compression

At first glance this is counter-intuitive. (*Why?*)

Compression/Decompression is clearly more CPU intensive than simply blitting an array into memory.

Compression

At first glance this is counter-intuitive. (*Why?*)

Compression/Decompression is clearly more CPU intensive than simply blitting an array into memory.

However, because there is *less total information* to transfer, the time spent unpacking the array can be far less than moving the array around wholesale.

Compression

At first glance this is counter-intuitive. (*Why?*)

Compression/Decompression is clearly more CPU intensive than simply blitting an array into memory.

However, because there is *less total information* to transfer, the time spent unpacking the array can be far less than moving the array around wholesale.

This is kind of like power steering, you can either tell wheels how to turn manually or you can tell the car how you want the wheels turned.

Compression

Compression is a guaranteed feature of HDF5 itself.

Compression

Compression is a guaranteed feature of HDF5 itself.

At minimum, HDF5 requires zlib.

Compression

Compression is a guaranteed feature of HDF5 itself.

At minimum, HDF5 requires zlib.

The compression capabilities feature a plugin architecture which allow for a variety of different algorithms, including user defined ones!

Compression

Compression is a guaranteed feature of HDF5 itself.

At minimum, HDF5 requires zlib.

The compression capabilities feature a plugin architecture which allow for a variety of different algorithms, including user defined ones!

PyTables supports:

• zlib (default), • lzo, • bzip2, and • blosc.

Compression

Compression is enabled in PyTables through *filters*.

Compression

Compression is enabled in PyTables through *filters*.

```
# complevel goes from [0,9]
filters = tb.Filters(complevel=5, complib='blosc', ...)
```

Compression

Compression is enabled in PyTables through *filters*.

```
# complevel goes from [0,9]
filters = tb.Filters(complevel=5, complib='blosc', ...)

# filters may be set on the whole file,
f = tb.openFile('/path/to/file', 'a', filters=filters)
f.filters = filters
```

Compression

Compression is enabled in PyTables through *filters*.

```
# complevel goes from [0,9]
filters = tb.Filters(complevel=5, complib='blosc', ...)

# filters may be set on the whole file,
f = tb.openFile('/path/to/file', 'a', filters=filters)
f.filters = filters

# filters may also be set on most other nodes
f.createTable('/', 'table', desc, filters=filters)
f.root.group._v_filters = filters
```

Compression

Compression is enabled in PyTables through *filters*.

```
# complevel goes from [0,9]
filters = tb.Filters(complevel=5, complib='blosc', ...)

# filters may be set on the whole file,
f = tb.openFile('/path/to/file', 'a', filters=filters)
f.filters = filters

# filters may also be set on most other nodes
f.createTable('/', 'table', desc, filters=filters)
f.root.group._v_filters = filters
```

Filters only act on chunked datasets.

Compression

Tips for choosing compression parameters:

Compression

Tips for choosing compression parameters:

- A mid-level (5) compression is sufficient. No need to go all the way up (9).

Compression

Tips for choosing compression parameters:

- A mid-level (5) compression is sufficient. No need to go all the way up (9).
- Use zlib if you must guarantee complete portability.

Compression

Tips for choosing compression parameters:

- A mid-level (5) compression is sufficient. No need to go all the way up (9).
- Use zlib if you must guarantee complete portability.
- Use blosc all other times. It is optimized for HDF5.

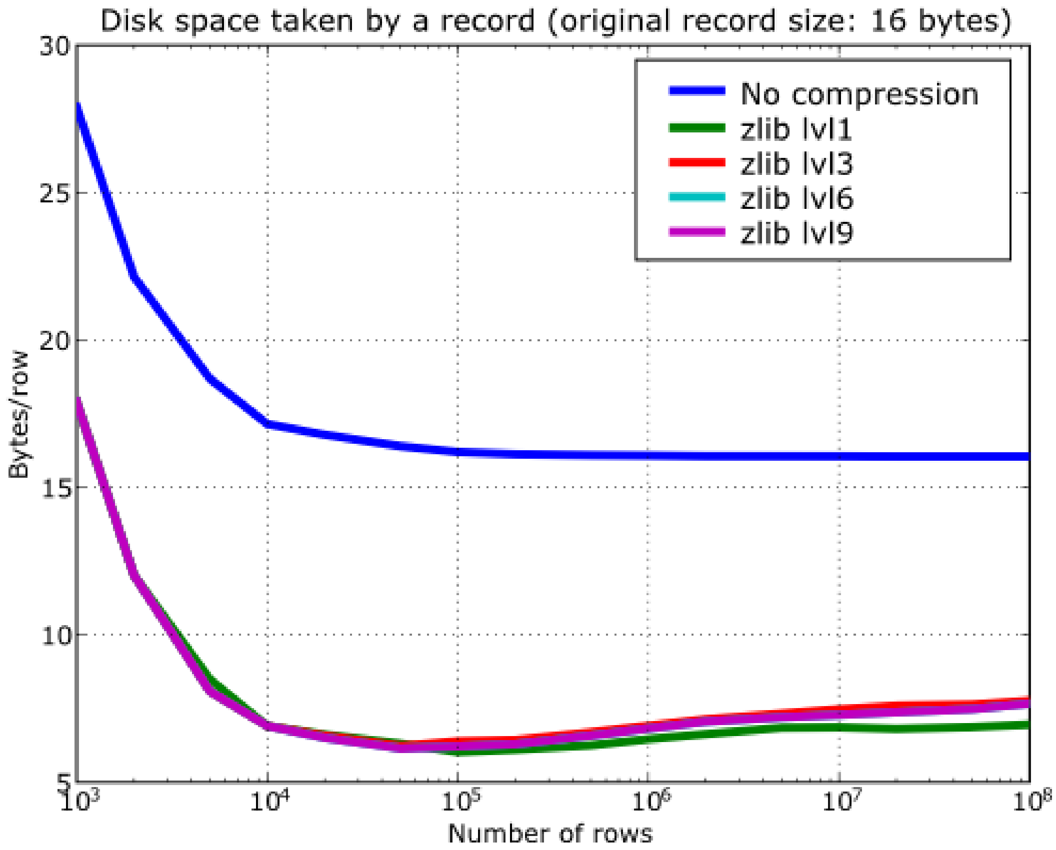
Compression

Tips for choosing compression parameters:

- A mid-level (5) compression is sufficient. No need to go all the way up (9).
- Use zlib if you must guarantee complete portability.
- Use blosc all other times. It is optimized for HDF5.

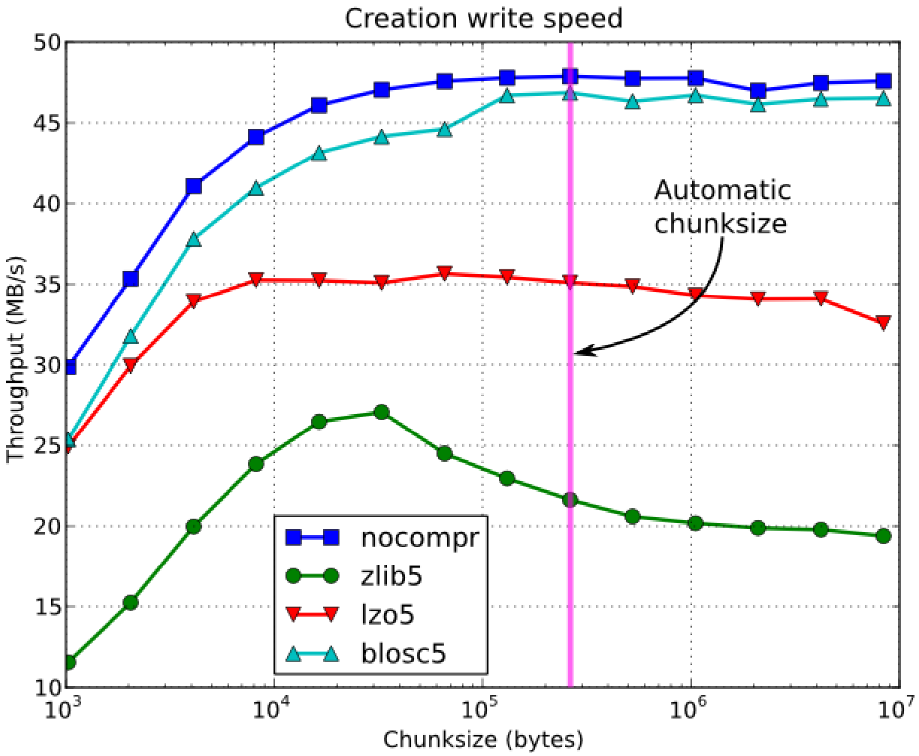
But why? (I don't have time to go into the details of blosc. However here are some justifications...)

Compression



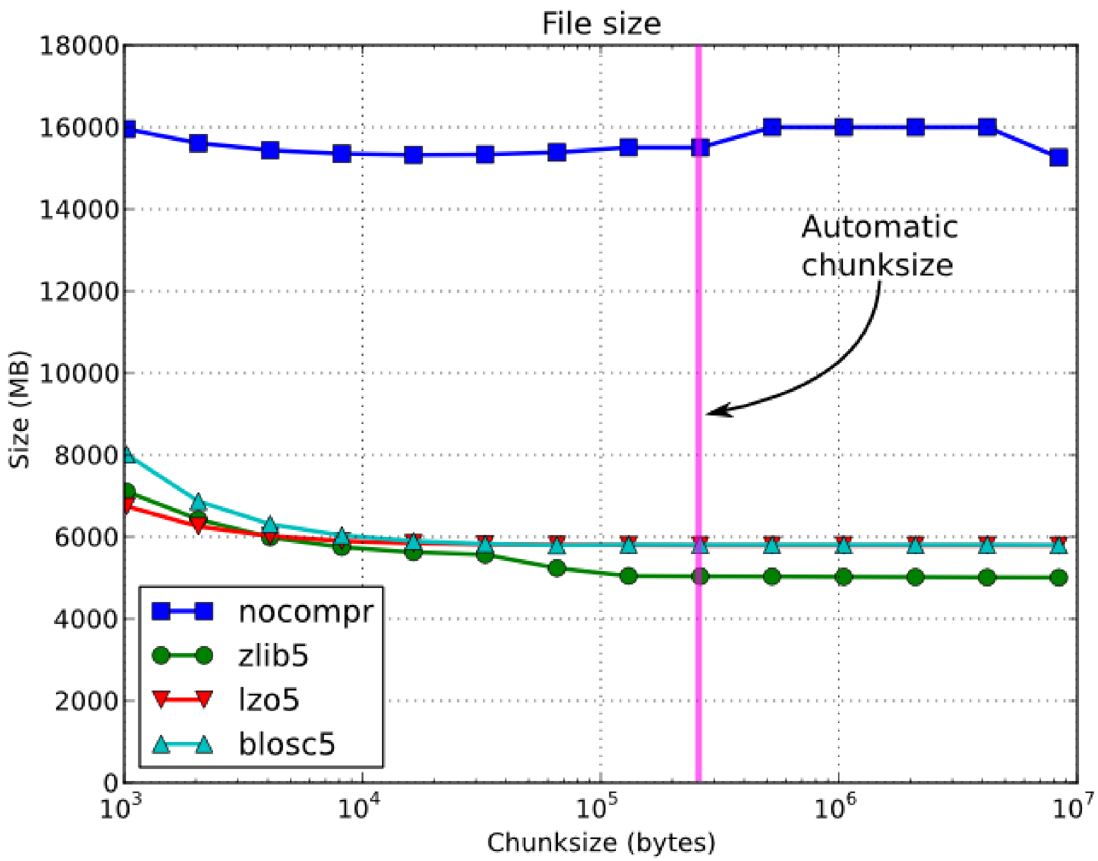
Comparison of different compression levels of zlib.

Compression



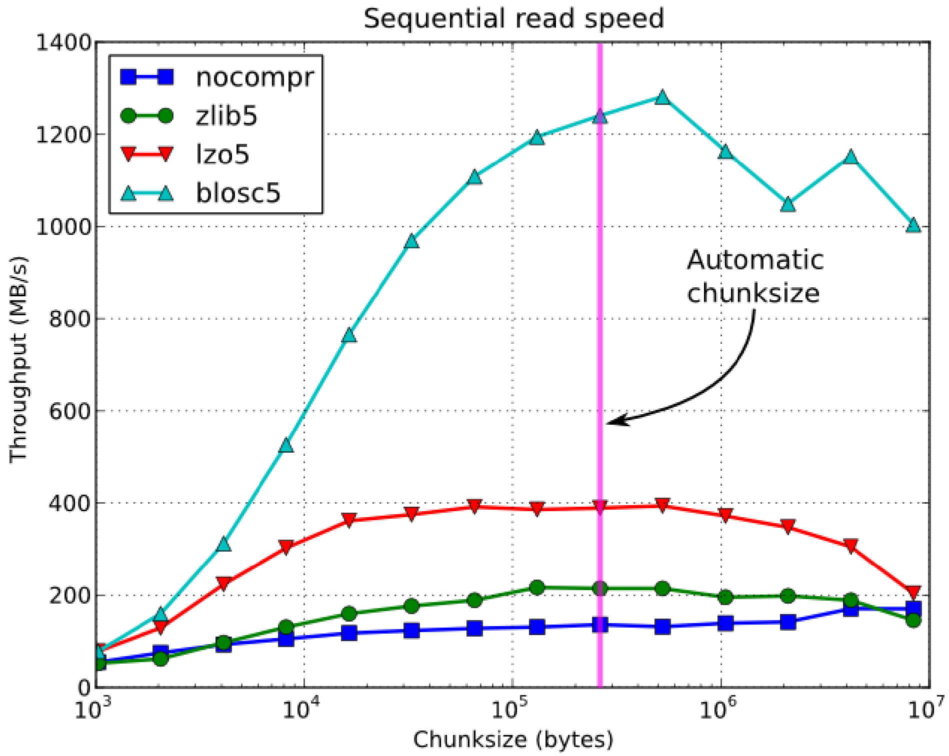
Creation time per element for a 15 GB EArray and different chunksizes.

Compression



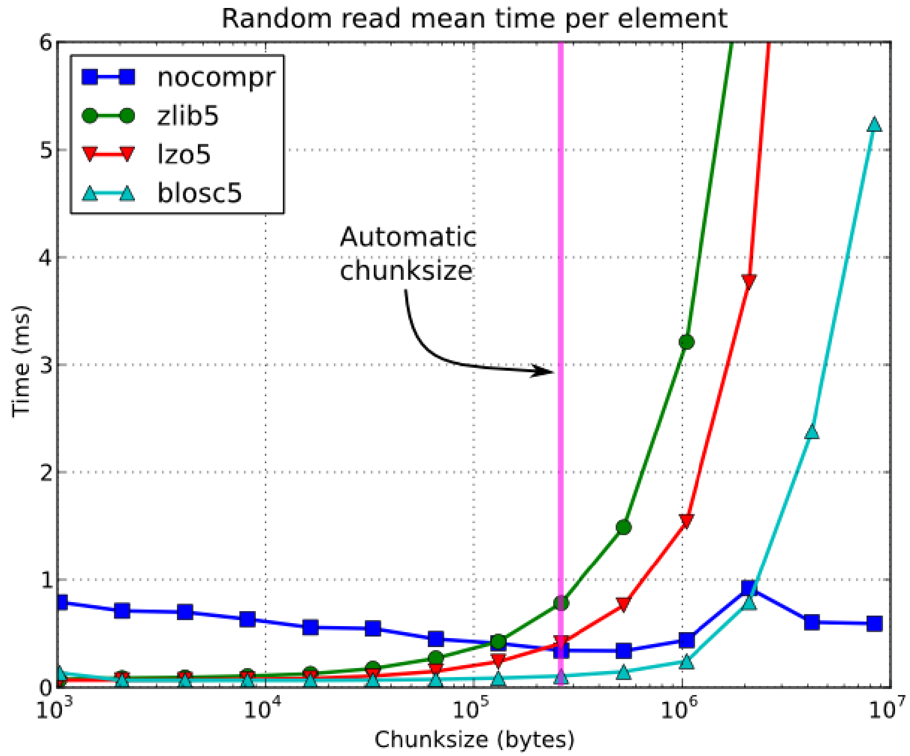
File sizes for a 15 GB EArray and different chunksizes.

Compression



Sequential access time per element for a 15 GB EArray and different chunksizes.

Compression



Random access time per element for a 15 GB EArray and different chunksizes.

Exercise

exer/spam_filter.py

Exercise

sol/spam_filter.py

Other Python Data Structures

Overwhelmingly, numpy arrays have been the in-memory data structure of choice.

Other Python Data Structures

Overwhelmingly, numpy arrays have been the in-memory data structure of choice.

Using lists or tuples instead of arrays follows analogously.

Other Python Data Structures

Overwhelmingly, numpy arrays have been the in-memory data structure of choice.

Using lists or tuples instead of arrays follows analogously.

It is data structures like sets and dictionaries which do not quite map.

Other Python Data Structures

Overwhelmingly, numpy arrays have been the in-memory data structure of choice.

Using lists or tuples instead of arrays follows analogously.

It is data structures like sets and dictionaries which do not quite map.

However, as long as all elements may be cast into the same atomic type, these structures can be stored in HDF5 with relative ease.

Sets

Example of serializing and deserializing sets:

```
>>> s = {1.0, 42, 77.7, 6E+01, True}

>>> f.createArray('/', 's', [float(x) for x in s])
/s (Array(4,)) ''
  atom := Float64Atom(shape=(), dflt=0.0)
  maindim := 0
  flavor := 'python'
  byteorder := 'little'
  chunkshape := None

>>> set(f.root.s)
set([1.0, 42.0, 77.7, 60.0])
```

Exercise

exer/dict_table.py

Exercise

sol/dict_table.py

What Was Missed

- Walking Nodes
- File Nodes
- Indexing
- Migrating to / from SQL
- HDF5 in other database formats
- Other Databases in HDF5
- HDF5 as a File System

Acknowledgements

Many thanks to everyone who made this possible!

Acknowledgements

Many thanks to everyone who made this possible!

- The HDF Group

Acknowledgements

Many thanks to everyone who made this possible!

- The HDF Group
- The PyTables Governance Team:
  - Josh Moore, • Antonio Valentino, • Josh Ayers

Acknowledgements

(Cont.)

- The NumPy Developers

Acknowledgements

(Cont.)

- The NumPy Developers
- h5py, the symbiotic project

Acknowledgements

(Cont.)

- The NumPy Developers
- h5py, the symbiotic project
- Francesc Alted 

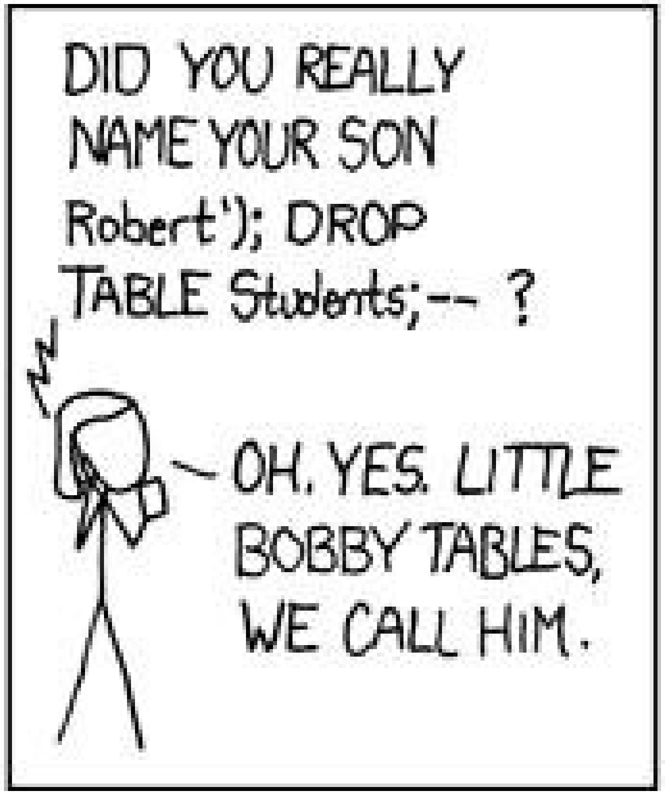
Acknowledgements

(Cont.)

- The NumPy Developers
- h5py, the symbiotic project
- Francesc Alted 

Shameless Plug: *We are always looking for more hands. Join Now!*

Questions


DID YOU REALLY NAME YOUR SON Robert'); DROP TABLE Students;-- ?
OH. YES. LITTLE BOBBY TABLES, WE CALL HIM.