PyTables

Processing And Analyzing Extremely Large Amounts Of Data In Python

Francesc Alted
falted@imk.es

Outline

- What is PyTables and why it exists?
- Interactive demonstration
- Some benchmarks
- Final remarks

Motivation

- Many scientific applications need to save and read very large amounts of data. Analysing this data effectively is a challenge.
- Computers are powerful enough to deal with very large data sets. But, the question is: can people handle very large data sets?
- Requirements:
  - Analysis is an iterative process: interactivity
  - Re-reading many times the data: efficency
  - Good framework to give the data an structure
  - Easy management
- PyTables is a Python package designed with these requirements in mind!

What PyTables offers?

- Interactivity
  - The user can take immediate action based on previous feedback
  - This greatly accelerates the process of data mining
- Efficiency
  - Improves your productivity
  - Very important when interactivity is an issue
- Hierarchical structure
  - It allows to break your data into smaller, related chunks
  - It offers you an intuitive way to categorize data
- Object-oriented interface
  - Datasets become objects that can be easily manipulated
  - In a hierarchical structure, objects facilitate data browsing

Machinery behind PyTables

PyTables relies on powerful software to achieve its goals:

- Python -- Everyone here knows that (2.2 version needed because generators are heavily used)
- HDF5 -- general purpose library and file format for storing scientific data
- numarray -- next generation of the well-known Numerical Python package
- Pyrex -- Tool to make Python extensions with a Python-like syntax

What is HDF5?

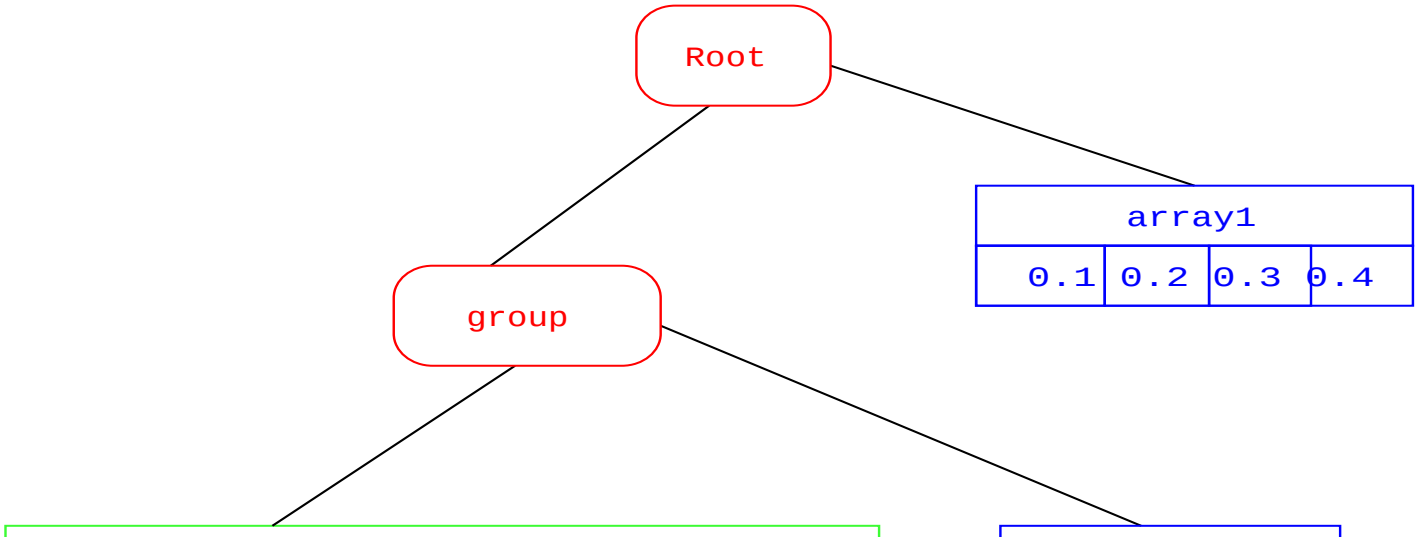
It is a general purpose library and file format for storing scientific data in a hierarchical manner. It is developed and maintained at the NCSA.

- Can store two primary objects: datasets and groups
  - Dataset: multidimensional array of data elements
  - Group: Structure for organinzing objects in the HDF5 file
- Very flexible and well tested in scientific environments
- Officially supported API's: C, Fortran and Java
- Being already used in: Meteorology, Oceanography, Astronomy, Astrophysics, Numerical simulation and many others

PyTables highlights

- General Python library to deal with scientific data
- Support of Numerical Python and numarray objects
- Appendable tables
- Can read generic HDF5 files
- Data compression support (for tables)
- Support of files bigger than 2 GB (unlimited data size in practice)
- Architecture-independent (is aware of big/low endian issues)

A first example



table

idnumber	identity	speed
0	Particle id: 0	0
1	Particle id: 1	2
2	Particle id: 2	4
3	Particle id: 3	6
4	Particle id: 4	8
5	Particle id: 5	10
6	Particle id: 6	12
7	Particle id: 7	14
8	Particle id: 8	16
9	Particle id: 9	18

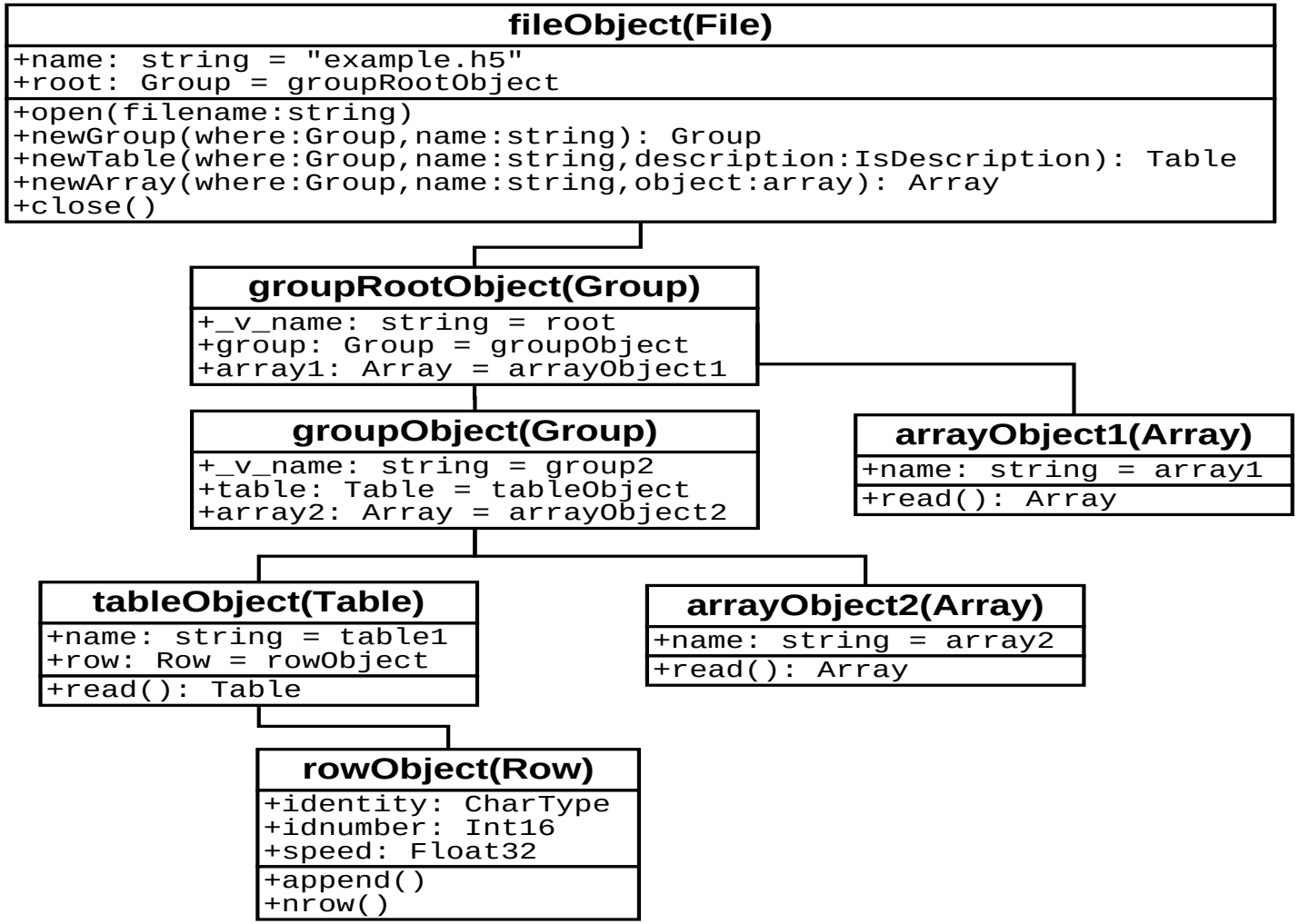
The PyTables code

```
from tables import *

class Particle(IsDescription):
    identity = Col("CharType", 16, " ", pos = 0)  # character String
    speed = Col("Float32", 1, pos = 2)  # single-precision
    idnumber = Col("Int16", 1, pos = 1)  # short integer

fileh = openFile("example.h5", mode = "w")
array = fileh.createArray(fileh.root, "array1", [.1,.2,.3,.4], "Float array")
group = fileh.createGroup(fileh.root, "group")
table = fileh.createTable(group, "table", Particle, "Table with 3 fields")
array = fileh.createArray(group, "array2", [1,2,3,4], "Int array")
row = table.row
for i in xrange(10):
    row['identity']  = 'Particle id: %3d' % (i)
    row['idnumber'] = i
    row['speed']  = i * 2.
    row.append()

fileh.close()
```

First example output

```
$ h5ls -rd example.h5
/array1                  Dataset {4}
    Data:
        (0) 0.1, 0.2, 0.3, 0.4
/group                   Group
/group/array2            Dataset {4}
    Data:
        (0) 1, 2, 3, 4
/group/table             Dataset {10/Inf}
    Data:
        (0) {0, "Particle id:   0", 0}, {1, "Particle id:   1", 2},
        (2) {2, "Particle id:   2", 4}, {3, "Particle id:   3", 6},
        (4) {4, "Particle id:   4", 8}, {5, "Particle id:   5", 10},
        (6) {6, "Particle id:   6", 12}, {7, "Particle id:   7", 14},
        (8) {8, "Particle id:   8", 16}, {9, "Particle id:   9", 18}
```

The object tree



How fast is fast?

- Several benchmarks have been conducted in order to know if PyTables is competitive with existing tools to save data persistently.
- Comparisons has been made with cPickle, struct, shelve and SQLite (a relational database).
- The benchmarks tested writing and selecting table data that fulfill a series of conditions.
- Two basic parameters where changed in each test to comparatively measure I/O performance:
  - The row size
  - The number of rows in the table

The record descriptions

The record sizes used are of two different lengths:

- 16 Bytes

```
class Small(IsDescription):
    var1 = Col("CharType", 4, "")
    var2 = Col("Int32", 1, 0)
    var3 = Col("Float64", 1, 0)
```

- 56 bytes

```
class Medium(IsDescription):
    name        = Col("CharType", 16, "")
    float1      = Col("Float64", 2, NA.arange(2))
    ADCcount    = Col("Int32", 1, 0)
    grid_i      = Col("Int32", 1, 0)
    grid_j      = Col("Int32", 1, 0)
    pressure    = Col("Float32", 1, 0)
    energy      = Col("Float64", 1, 0)
```

The selection mechanism

- PyTables:
  - e = [p['var1'] for p in table.iterrows()
    if p['var2'] < 20]
- cPickle:
  - while rec:
    record = cPickle.loads(rec[1])
    if record['var2'] < 20:
    e.append(record['var1'])
- struct:
  - while rec:
    record = struct.unpack(isrec._v_fmt, rec[1])
    if record[1] < 20:
    e.append(record[0])
- SQLite:
  - cursor.execute("select var1 from table where var2 < 20")

Note: cPickle and struct tests use a RECNO Berkeley DB (4.1.3 version) in order to emulate records efficently.

Benchmark platform description

- Laptop with Pentium IV @ 2 GHz and 256 MB RAM
- Disk IDE @ 4200 RPM
- PyTables 0.4
- HDF5 1.4.5
- numarray 0.4
- Linux Debian 3.0
- GCC 2.95 compiler

Comparing cPikle and struct with PyTables

Package	Record length	Krows/s		MB/s		total Krows	file size	memory used	%CPU	
		write	read	write	read		(MB)	(MB)	write	read
cPickle	small	23.0	4.3	0.65	0.12	30	2.3	6.0	100	100
cPickle	small	22.0	4.3	0.60	0.12	300	24	7.0	100	100
cPickle	medium	12.3	2.0	0.68	0.11	30	5.8	6.2	100	100
cPickle	medium	8.8	2.0	0.44	0.11	300	61	6.2	100	100
struct	small	61	71	1.6	1.9	30	1.0	5.0	100	100
struct	small	56	65	1.5	1.8	300	10	5.8	100	100
struct	medium	51	52	2.7	2.8	30	1.8	5.8	100	100
struct	medium	18	50	1.0	2.7	300	18	6.2	100	100
PyTables	small	434	469	6.8	7.3	30	0.49	6.5	100	100
PyTables	small (c)	326	435	5.1	6.8	30	0.12	6.5	100	100
PyTables	small	663	728	10.4	11.4	300	4.7	7.0	99	100
PyTables	medium	194	340	10.6	18.6	30	1.7	7.2	100	100
PyTables	medium (c)	142	306	7.8	16.6	30	0.3	7.2	100	100
PyTables	medium	274	589	14.8	32.2	300	16.0	9.0	100	100

Table 1: Comparing PyTables performance with cPickle and struct serializer modules in Standard Library. (c) means that a compression is used.

Conclusions from first benchmark (cPickle & struct)

- Writing
  - Between 20 and 30 times faster than cPickle
  - Between 3 and 10 time faster than struct
- Reading
  - Around 100 times faster than cPickle
  - Around 10 times faster than struct

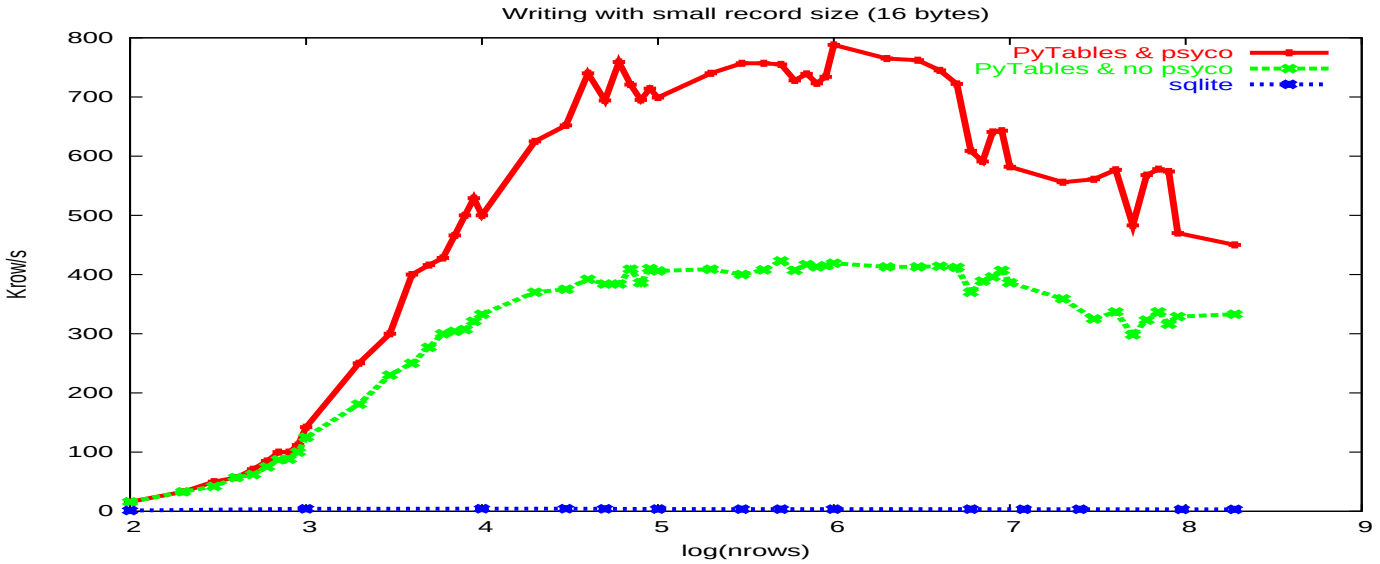
PyTables is far superior to cPickle and struct for any amount of data

Comparing SQLite with PyTables (tabular values)

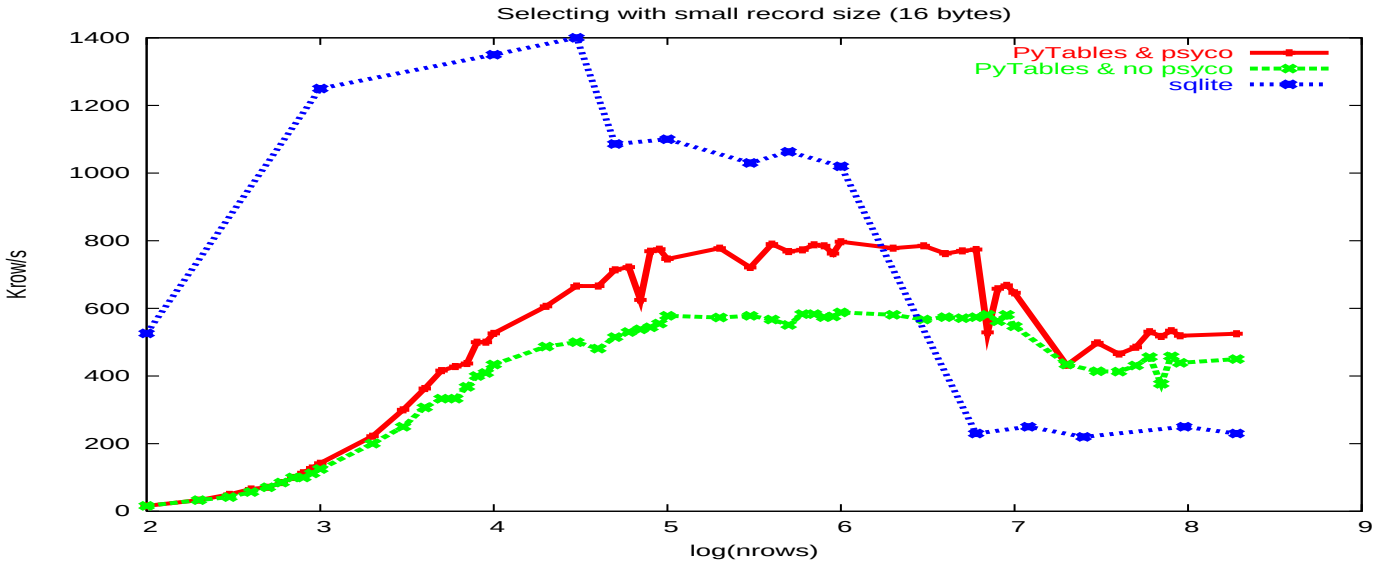
Package	Record length	Krows/s		MB/s		total Krows	file size	memory used	%CPU	
		write	read	write	read		(MB)	(MB)	write	read
SQLite (ic)	small	3.66	1030	0.18	51.0	300	15.0	5.0	100	100
SQLite (oc)	small	3.58	230	0.19	12.4	6000	322.0	5.0	100	25
SQLite (ic)	medium	2.90	857	0.27	80.0	300	28.0	5.0	100	100
SQLite (oc)	medium	2.90	120	0.30	13.1	3000	314.0	5.0	100	11
PyTables (ic)	small	679	797	10.6	12.5	3000	46.0	10.0	98	98
PyTables (oc)	small	576	590	9.0	9.2	30000	458.0	11.0	78	76
PyTables (ic)	medium	273	589	14.8	32.2	300	16.0	9.0	100	100
PyTables (oc)	medium	184	229	10.0	12.5	10000	534.0	17.0	56	40

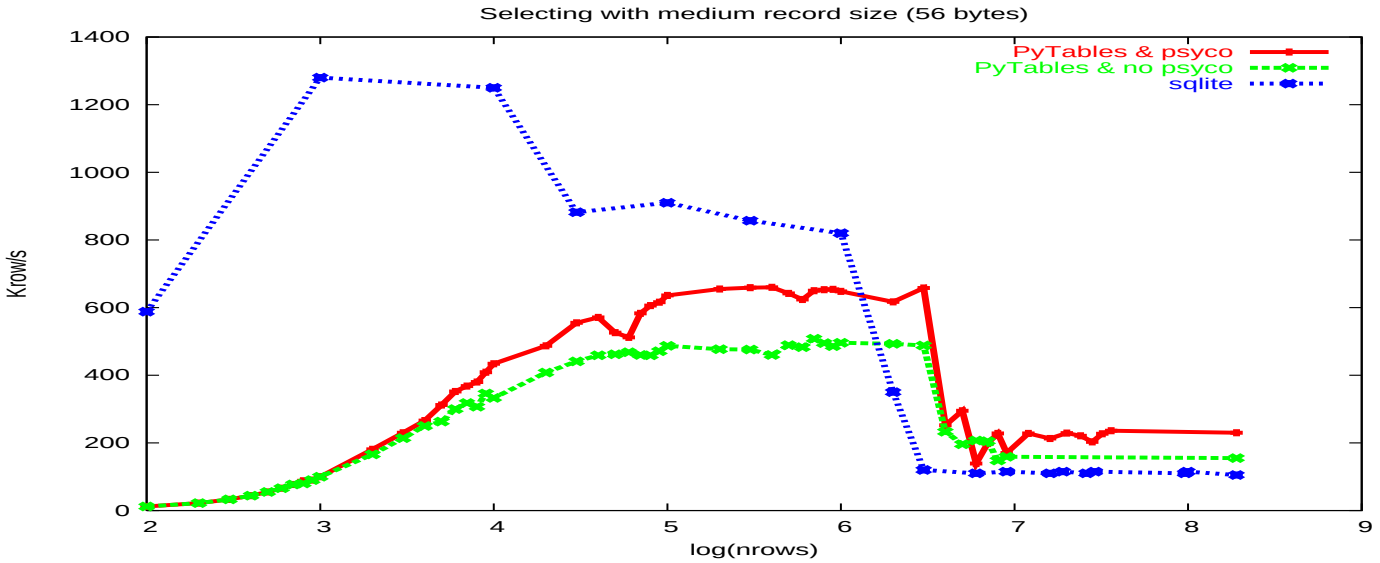
Table 2: Effect of different record sizes and table lengths in SQLite performance. (ic) means that the test ran in-core while (oc) means out-of-core.

Comparing SQLite with PyTables (writing)



Comparing SQLite with PyTables (selecting)





Conclusions from second benchmark (SQLite)

Writing

- PyTables is around 100 times faster than SQLite
- Caveat: I did not attempt to optimize SQLite for inserts

Reading

- In-core selects (i.e. file size fits in cache memory)
  - PyTables achieves between 60% and 80% of SQLite speed
- Out-of-core selects (i.e. file size do not fit in cache memory)
  - PyTables outperforms SQLite by a factor of two (that depends on the kind of record)

PyTables beats SQLite when processing very large amounts of data!

(while being close of it for smaller sizes)

Current PyTables limitations and plans for future

- Elements in columns can not have more than one dimension
- Attributes in nodes only support string values (but cPikle is there!)
- Unlimited arrays are not supported (perhaps in the next release)
- Compression for arrays not available (will be available when unlimited arrays are implemented)
- Object elements can not be related to other elements

Final remarks

- PyTables allows you to process your data interactively and quickly.
- If you have large amounts of data, an interpreted language like Python is enough in order to get maximum performance:
  PyTables (+ Psyco) is only limited by disk I/O speed.
- PyTables has been designed to excel in retrieving and selecting data very fast, but is also very fast when writing (I didn’t expect this result - a welcome surprise).

PyTables is for real work!

- More than 200 tests units are now incorporated. More will be added and quality will only improves as PyTables evolves.
- PyTables is already in beta and its API is stable.
- It comes with complete documentation both in doc strings format as well as a high quality 40 pages user's manual in PDF and HTML formats.
- Download the last version (0.4, released on March, 18th) and use it for free from:

http://pytables.sourceforge.net