PyTables: Processing And Analyzing Extremely Large Amounts Of Data In Python

Francesc Alted [*]
falted@openlc.org

Mercedes Fernández-Alonso [†]
fernande@exp.uji.es

2003 April, 4th

Abstract

Processing large amounts of data is a must for people working in such fields of scientific applications as Meteorology, Oceanography, Astronomy, Astrophysics, Experimental Physics or Numerical simulation to name only a few. Existing relational or object-oriented databases usually are good solutions for applications in which multiple distributed clients need to access and update a large centrally managed database (e.g., a financial trading system). However, they are not optimally designed for efficient read-only database queries to pieces, or even single attributes, of objects, a requirement for processing data in many scientific fields such as the ones mentioned above.

This paper describes PyTables [1], a Python library that addresses this need, enabling the end user to manipulate easily scientific data tables and regular homogeneous (such as Numeric [2] arrays) Python data objects in a persistent, hierarchical structure. The foundation of the underlying hierarchical data organization is the excellent HDF5 [3] C library.

1 Motivation

Many scientific applications frequently need to save and read extremely large amounts of data (frequently, this data is derived from experimental devices). Analyzing the data requires re-reading it many times in order to select the most appropriate data that reflects the scenario under study. In general, it is not necessary to modify the gathered data (except perhaps to enlarge the dataset), but simply access it multiple times from multiple points of entry.

Keeping this in mind, PyTables has been designed to save data in as little space as possible (compression is supported although it is not enabled by default), and has been optimized to read and write data as fast as possible. In addition, there is no the 2 GB limit in the library; if your filesystem can support files bigger than 2 GB, then so can PyTables. The data can be hierarchically stored and categorized. PyTables also allows to the user to append data to already created tables or arrays.

To ensure portability of PyTables files between computers with different architectures, the library has been designed to handle both big and little endian numbers: in principle, it is possible to save the data on a big-endian machine and read it (via NFS for example) on a little-endian machine.

Others Python libraries have been developed for related purposes, such as the NetCDF module of Scientific Python [4] and PyHL, a subpackage of the more generic HL-HDF5 [5] effort. These packages have some grave limitations, however. Scientific Python is based on the NetCDF [6], which doesn't support hierarchical data organization; nor is its supported data types set as rich as HDF5. PyHL is based on HDF5, but its implementation is lower-level than PyTables and less user-friendly: for example, PyHL requires the user to program a Python C-module to access the compound types. Another drawback of PyHL is that its core objects are not "appendable".

[*]Freelance Applications Programmer, 12100 Castelló, Spain
[†]Dept. Ciències Experimentals, Universitat Jaume I, 12080 Castelló, Spain

2 Implementation overview

Tables are the core data structures in PyTables and are collections of (inhomogeneous) data objects that are made persistent on disk. They are appendable insofar as you can add new objects to the collections or new attributes to existing objects. Besides, they are accessed directly for improved I/O speed.

A table is defined as a collection of records whose values are stored in fixed-length fields. All records have the same structure and all values in each field have the same data type. The terms "fixed-length" and "strict data types" seems to be quite a strange requirement for an interpreted language like Python, but they serve a useful function if the goal is to save very large quantities of data (such as is generated by many scientific applications, for example) in an efficient manner that reduces demand on CPU time and I/O.

In order to emulate records (C structs in HDF5) in Python, PyTables makes use of an special RecArray object present in the numarray library [7] with the capability to store a series of records (inhomogeneous data) in-memory. So, a RecArray buffer is setup to be used as a buffer for the I/O. This improves quite a lot the I/O and optimizes the access to data fields, as you will see later on. Moreover, PyTables also detect errors in field assignments as well as type errors.

For example, you can define arbitrary tables in Python simply by declaring a class with the name field and types information:

```
class Particle(IsDescription):
    ADCcount    = Col("Int16", 1, 0)    # signed short integer
    TDCcount    = Col("UInt8", 1, 0)    # unsigned byte
    grid_i      = Col("Int32", 1, 0)    # integer
    grid_j      = Col("Int32", 1, 0)    # integer
    idnumber    = Col("Int64", 1, 0)    #signed long long
    name        = Col("CharType", 16, "")  # 16-character String
    pressure    = Col("Float32", 2, 0)  # float  (single-precision)
    energy      = Col("Float64", 1, 0)  # double (double-precision)
```

Simply instantiate the class, fill each object instance with your values, and then save arbitrarily large collections of these objects in a file for persistent storage. This data can be retrieved and post-processed quite easily with PyTables or with another HDF5 application.

I/O is buffered for optimal performance. The buffer is dynamically created and its size is computed from different parameters, such as the record size, the expected number of records in a table and the compression mode (on or off). The buffers are ordinary Python lists which grow during the process of adding records. When the buffer size limit is reached it is flushed to the disk. In the same way, when the file is closed, all the open buffers in memory are flushed.

The underlying HDF5 library writes the data in groups of chunks of a predetermined size to write to disk. It uses B-trees in memory to map data structures to disk. The more chunks that are allocated for a dataset the larger the B-tree. Large B-trees take memory and causes file storage overhead as well as more disk I/O and higher contention for the meta data cache. As a consequence, PyTables must balance between memory limits (small B-trees, larger chunks) and file storage overhead and time to access to data (big B-trees, smaller chunks). The parameters used to tune the chunk size are the record size, number of expected records and compression mode.

The tuning of chunk size and buffer size parameters affects the performance and the amount of memory consumed. The present tuning in PyTables is based on exhaustive benchmarking experiments on an Intel architecture. On other architectures, this tuning may vary (although, hopefully, only up to a small extent).

PyTables allows the user to improve the performance of his application in a variety of ways. When planning to deal with very large data sets, user should carefully read the optimization section in User's Manual to learn how to manually set PyTables parameters to boost performance.

The hierarchical model of the underlying HDF5 library allows PyTables to manage tables and arrays in a tree-like structure. In order to achieve this, an object tree entity is dynamically created imitating the file structure on disk. The user views the file objects by walking throughout this object tree. Tree metadata nodes describe what kind of data is stored in the object.

3 Usage

This section focuses on how to write and read heterogeneous data to and from PyTables files, as these are the most commonly-used features.

In order to create a table, we first define a user class that describes a compound data type:

```
# Define a user record to characterize some kind of particle
class Particle(IsDescription):
    ADCcount    = Col("Int16", 1, 0)    # signed short integer
    TDCcount    = Col("UInt8", 1, 0)    # unsigned byte
    grid_i      = Col("Int32", 1, 0)    # integer
    grid_j      = Col("Int32", 1, 0)    # integer
    idnumber    = Col("Int64", 1, 0)    #signed long long
    name        = Col("CharType", 16, "")  # 16-character String
    pressure    = Col("Float32", 2, 0)  # float  (single-precision)
    energy      = Col("Float64", 1, 0)  # double (double-precision)
```

then, we open a file for writing, and create a table for it:

```
h5file = openFile("example.h5", mode = "w", title ="Test file")
table = h5file.createTable(h5file.root, ’readout’, Particle(), "Readout ex.")
```

The third parameter is an instance of our user class that will be used to define all the characteristics of the table on disk.

Now, we can proceed to populate this table with some (synthetic) values:

```
# Get a shortcut to the Row instance in Table object
particle = table.row
# Fill the table with 10**6 particles
for i in xrange(10**6):
    # First, assign the values to the Particle record
    particle[’name’]  = ’Particle: %6d’ % (i)
    particle[’TDCcount’] = i % 256
    particle[’ADCcount’] = (i * 256) % (1 << 16)
    particle[’grid_i’] = i
    particle[’grid_j’] = 10 - i
    particle[’pressure’] = float(i*i)
    particle[’energy’] = float(particle.pressure ** 4)
    particle[’idnumber’] = i * (2 ** 34)  # This exceeds long integer range
    # Insert a new particle record
    particle.append()
# Flush the table buffers
table.flush()
```

At this point, one million records are saved to disk. Now we call the method iterrows() to access table data. It returns an iterator that cycles over all the records in the table. When called from inside a loop, the iterrows() iterator returns an object that mirrors the rows on disk, whose items and values are the same as the fields and values in the Table. This makes it very easy to process the data in table:

```
# Read actual data from table. We are interested in collecting pressure values
# on entries where TDCcount field is greater than 3 and pressure less than 50.
pressure = [ x[’pressure’] for x in table.iterrows()
                 if x[’TDCcount’] > 3 and 1 < x[’pressure’] < 50 ]
print "Field pressure elements satisfying the cuts ==>", pressure
```

This makes the utilization of comprehensive lists very user-friendly, as expressions are compact and meaningful.

For a full detailed explanation on PyTables operation, see the User’s Manual [8].

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.

4 Some benchmarks

In order to see if PyTables is competitve or not with other data persistent management systems, some benchmarks has been done. The benchmarks were conducted on a laptop with a 2 GHz Intel Mobile chip, an IDE disk at 4200 rpm, and 256 MB of RAM. The operating system was a Debian 3.0 with a Linux kernel version 2.4.20 and gcc 2.95.3. Python 2.2.2, numarray 0.4 and hdf5 1.4.5 were also used.

There were used 2 different record sizes, 16 bytes and 56 bytes, using the next column definitions for the Tables:

```
class Small(IsDescription):
    var1 = Col("CharType", 16)
    var2 = Col("Int32", 1)
    var3 = Col("Float64", 1)

class Medium(IsDescription):
    name        = Col('CharType', 16)
    float1      = Col("Float64", 1)
    float2      = Col("Float64", 1)
    ADCcount    = Col("Int32", 1)
    grid_i      = Col("Int32", 1)
    grid_j      = Col("Int32", 1)
    pressure    = Col("Float32",1)
    energy      = Col("Float64", 1)
```

As a reference, the same tests have been repeated using the cPickle and struct modules that comes with Python. For struct and cPickle a Berkeley DB 4.1 recno (record based) database has been used. The results can be seen in table 1.

These figures compares quite well with the serializer modules that comes with Python: PyTables can write records between 20 and 30 times faster than cPickle and between 3 and 10 times faster than struct; and retrieves information around 100 times faster than cPickle and between 8 and 10 times faster than struct. Note that PyTables uses a bit more memory (a factor 2 more or less) than the other modules. This is not grave, however.

One can see that compression doesn't hurt a lot of performance (less in reading than writing) in PyTables, and, if the data is compressible, it can be used for reducing the file size in many situations.

It also worth noting that the resulting PyTables file sizes are even smaller that the struct files when compression is off and from 5 to 10 times smaller if using compression (keep in mind that the sample data used in the tests is synthetic). The difference with cPickle is still bigger (always favorable to PyTables). This is because PyTables saves data in binary format, while cPickle serialize the data using a format that consumes fairly more space.

Bear in mind that this elementary benchmark has been done for *in-core* objects (i.e. with file sizes that fit easily in the memory of the operating system). This is because, cPickle and struct performance is bounded by the CPU, and you can expect their performance to not significantly change during out-of-core tests. See later for a more detailed discussion for *out-of-core* working sets during the comparison against a relational database.

Finally, let's end with a comparison with one relational database, SQLite [10]. A series of benchmarks has been conducted with the same small and medium sizes that were used to obtain the tables before. See at the results in table 2. Note that these results has been obtained using PyTables together with Psyco (see 11), an extension module to speed up the execution of any Python code.

Also, you can examine a much detailed series of plots (see figures 1, 2, 3 and 4) comparing specifically write/read speed (in Krow/s) vs number of rows in tables. Please, note that the horizontal axis (the number of rows in tables) has a logarithmic scale in all the cases.

From these figures there can be extracted some interesting conclusions:

- The read/write speed, both in PyTables and SQLite, is high when the test runs in-core, and it is limited to the I/O disk speed (10 MB/s for writing and 15 MB/s for reading) for out-of-core tests.
- For small row sizes (*in-core* case), more than 800 Krow/s can achieved in PyTables. This is surprising for an interpreted language like Python.
- SQLite consumes far less memory than PyTables. However, the memory consumption never grows too high (17 MB for tables of 500 MB of size) in PyTables.
- For the same record definitions, the SQLite file takes more space on disk than PyTables. This is a direct consequence of how the data is saved (in binary format for PyTables, a string representation for SQLite).
- When the file size fits in cache memory (i.e. the data is *in-core*), the SQLite selects over the tables are between 1.3 and 2.5 times faster than PyTables.
- For very small table sizes, SQLite is much faster that PyTables. However, for such small row number ranges (tables < 10000 rows), there is no too much point in that one selection takes 7 ms (SQLite) or 25 ms (PyTables). When doing that interactively the user perception is the same: the response is instantaneous.
- When the file doesn't fit in the cache (>200 MB), SQLite performs at maximum I/O disk speed, but as we seen before, SQLite needs more space for data representation, so PyTables performs better (between a factor 1.3 and 2.5, depending on the record size and if Psyco is used or not).
- Psyco helps PyTables to improve its performance, even in the case that the file doesn't fit in the filesystem cache.
- Finally, note that SQLite performs much worse than PyTables when writing records. Chances are that I have not chosen the best algorithm in the SQLite benchmark code.

As SQLite is one of the fastest (free) relational databases on the market, one can conclude that PyTables has no competence (at least among those that I'm aware of!) when it comes to deal (both writing and reading) with very large amounts of (scientific) data in Python.

Of course, it should be stressed that PyTables achieves great performace yet allowing the user access to their data in a very Pythonic manner, with no need to construct complicated SQL queries to select the data he is interested in: it's all done in pure Python!. That fact indeed smooths the learning curve and improves productivity.

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.

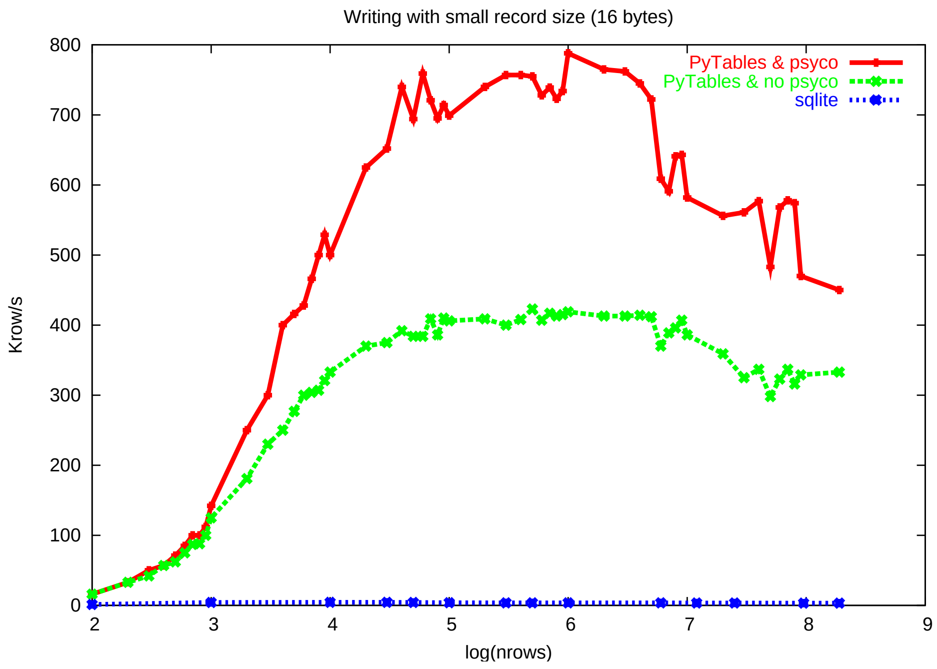

Figure 1: Comparison for writing rows between PyTables (with and without Psyco), and SQLite for small record sizes.

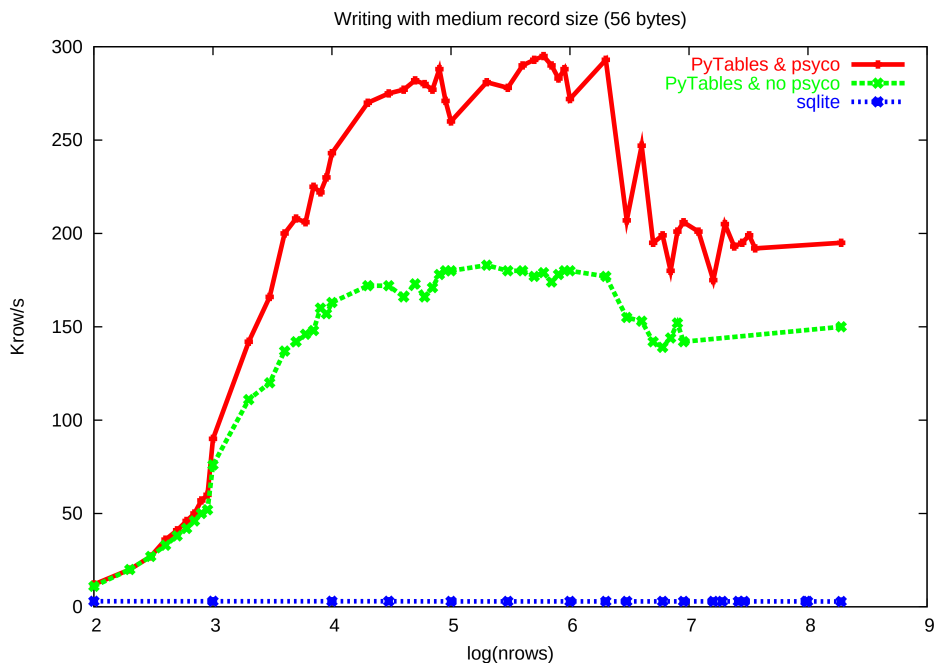

Figure 2: Comparison for writing rows between PyTables (with and without Psyco), and SQLite for medium record sizes.

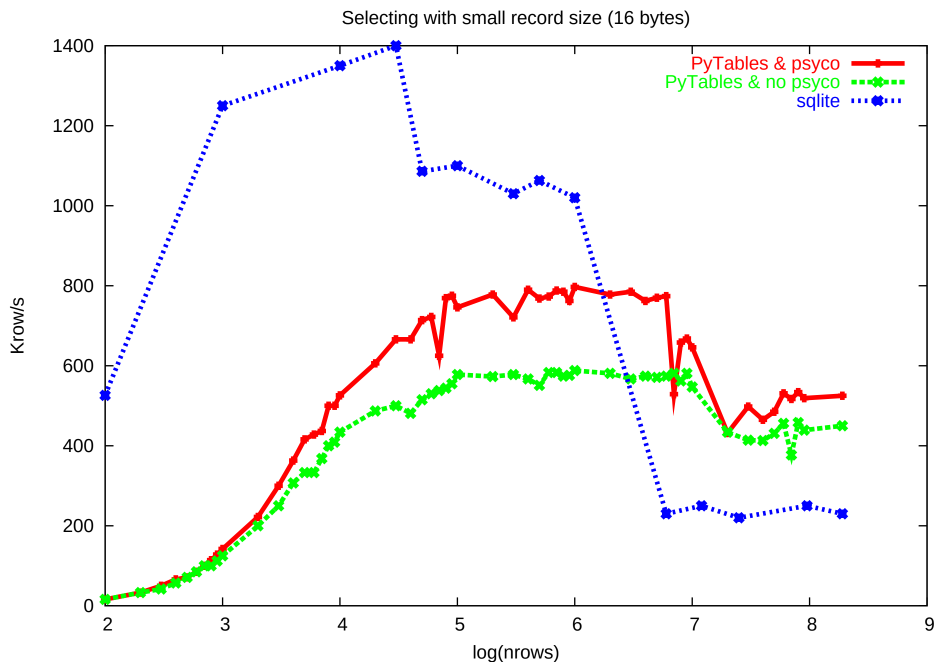

Figure 3: Comparison for writing rows between PyTables (with and without Psyco), and SQLite for small record sizes.

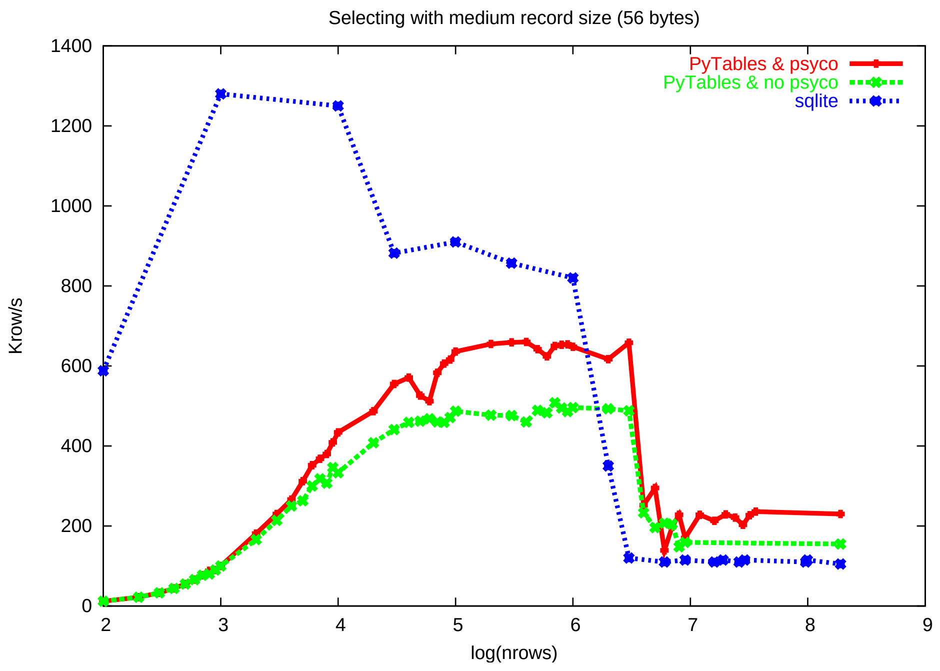

Figure 4: Comparison for writing rows between PyTables (with and without Psyco), and SQLite for medium record sizes.

5 Unit tests

Unittest [12] is a superb utility to test libraries in a series of the most common (but sometimes also unusual) user cases, that comes with the Python Standard Library.

PyTables comes with more than 200 orthogonal tests, which carefully checks almost any functionality present in the package. Besides, that ensures a high quality and stability over the different releases, as more tests will be added.

6 Conclusions

We have shown that even though Python is an interpreted (and therefore, slow) language, with the adoption of a buffered I/O, a good selection of underlying C libraries and with the help of some extension types (Python objects written in C for better performance), it can be feasible to use it to process and analyze extremely large amounts of data. This enables the user to mine interactively his data and get an immediate feel for its characteristics. The very Pythonic feel of PyTables also smooths the learning curve and improves productivity.

By combining PyTables with complete Python scientific libraries such as SciPy [9], it becomes possible to easily develop simple-yet-powerful applications to analyze all sorts of scientific data, using only open source tools.

7 Availability

The library is freely available under the terms of BSD license in the PyTables home page [1].

8 Thanks

We are specially grateful to Scott Prater who has revised this document and made it more readable.

9 References

1. *PyTables Home Page.* `http://pytables.sourceforge.net`
2. *Numerical Python.* Adds a fast, compact, multidimensional array language facility to Python. `http://www.pfdubois.com/numpy/`
3. *What is HDF5?.* Concise description about HDF5 capabilities and its differences from earlier versions (HDF4). `http://hdf.ncsa.uiuc.edu/whatishdf5.html`
4. *Scientific Python library.* `http://starship.python.net/~hinsen/ScientificPython/`
5. *A High Level Interface to the HDF5 File Format.* `ftp://ftp.ncsa.uiuc.edu/HDF/HDF5/contrib/hl-hdf5/README.html`
6. *NetCDF FAQ: What is NetCDF?* `http://www.unidata.ucar.edu/packages/netcdf/faq.html#whatisit`
7. *Numarray Python.* The next generation of Numerical Python. `http://www.pfdubois.com/numpy/`
8. *PyTables User's Manual.* `http://pytables.sourceforge.net/html-doc/usersguide-html.html`
9. *SciPy Home Page.* `http://www.scipy.org`
10. SQLite. *An Embeddable SQL Database Engine.* `http://www.sqlite.org/`
11. Psyco. *A Python extension module which can massively speed up the execution of any Python code.* `http://psyco.sourceforge.net/`
12. *Unittest Python module documentation.* `http://www.python.org/doc/current/lib/module-unittest.html`