

**Hierarchical datasets in Python**

## **PyTables User Guide**

*Release 2.4.0*

**The PyTables Governance Team**

October 16, 2012



# CONTENTS

<b>I</b>	<b>The PyTables Core Library</b>	<b>7</b>
<b>1</b>	<b>Introduction</b>	<b>9</b>
1.1	Main Features . . . . .	10
1.2	The Object Tree . . . . .	11
<b>2</b>	<b>Installation</b>	<b>17</b>
2.1	Installation from source . . . . .	17
2.2	Binary installation (Windows) . . . . .	21
<b>3</b>	<b>Tutorials</b>	<b>23</b>
3.1	Getting started . . . . .	23
3.2	Browsing the <i>object tree</i> . . . . .	30
3.3	Committing data to tables and arrays . . . . .	37
3.4	Multidimensional table cells and automatic sanity checks . . . . .	40
3.5	Using links for more convenient access to nodes . . . . .	45
3.6	Exercising the Undo/Redo feature . . . . .	49
3.7	Using enumerated types . . . . .	53
3.8	Dealing with nested structures in tables . . . . .	56
3.9	Other examples in PyTables distribution . . . . .	61
<b>4</b>	<b>Library Reference</b>	<b>63</b>
4.1	tables variables and functions . . . . .	64
4.2	The File Class . . . . .	64
4.3	The Node class . . . . .	64
4.4	The Group class . . . . .	64
4.5	The Leaf class . . . . .	65
4.6	The Table class . . . . .	66
4.7	The Array class . . . . .	66
4.8	The CArray class . . . . .	67
4.9	The EArray class . . . . .	67
4.10	The VArray class . . . . .	67
4.11	The Link class . . . . .	67
4.12	The SoftLink class . . . . .	67
4.13	The ExternalLink class . . . . .	68
4.14	The UnImplemented class . . . . .	68
4.15	The Unknown class . . . . .	68
4.16	The AttributeSet class . . . . .	68
4.17	Declarative classes . . . . .	68
4.18	Helper classes . . . . .	69

4.19	The Expr class - a general-purpose expression evaluator . . . . .	70
4.20	Exceptions module . . . . .	70
<b>5</b>	<b>Optimization tips</b>	<b>71</b>
5.1	Understanding chunking . . . . .	71
5.2	Accelerating your searches . . . . .	72
5.3	Compression issues . . . . .	86
5.4	Using Psycho . . . . .	97
5.5	Getting the most from the node LRU cache . . . . .	100
5.6	Compacting your PyTables files . . . . .	104
<b>II</b>	<b>Complementary modules</b>	<b>105</b>
<b>6</b>	<b>filenode - simulating a filesystem with PyTables</b>	<b>107</b>
6.1	What is filenode? . . . . .	107
6.2	Finding a filenode node . . . . .	107
6.3	filenode - simulating files inside PyTables . . . . .	108
6.4	Complementary notes . . . . .	110
6.5	Current limitations . . . . .	111
6.6	filenode module reference . . . . .	111
<b>7</b>	<b>netcdf3 - a PyTables NetCDF3 emulation API (deprecated)</b>	<b>115</b>
7.1	What is netcdf3? . . . . .	115
7.2	Using the tables.netcdf3 package . . . . .	115
7.3	tables.netcdf3 package reference . . . . .	119
7.4	Converting between true netCDF files and tables.netcdf3 files . . . . .	121
7.5	tables.netcdf3 file structure . . . . .	122
7.6	Sharing data in tables.netcdf3 files over the Internet with OPeNDAP . . . . .	123
7.7	Differences between the Scientific.IO.NetCDF API and the tables.netcdf3 API . . . . .	123
<b>III</b>	<b>Appendixes</b>	<b>125</b>
<b>8</b>	<b>Supported data types in PyTables</b>	<b>127</b>
<b>9</b>	<b>Condition Syntax</b>	<b>129</b>
<b>10</b>	<b>PyTables parameter files</b>	<b>131</b>
10.1	Tunable parameters in parameters.py. . . . .	131
<b>11</b>	<b>Using nested record arrays (deprecated)</b>	<b>135</b>
11.1	Introduction . . . . .	135
11.2	NestedRecArray methods . . . . .	137
11.3	NestedRecord objects . . . . .	138
<b>12</b>	<b>Utilities</b>	<b>139</b>
12.1	ptdump . . . . .	139
12.2	ptrepack . . . . .	141
12.3	nctoh5 . . . . .	144
<b>13</b>	<b>PyTables File Format</b>	<b>147</b>
13.1	Mandatory attributes for a File . . . . .	147
13.2	Mandatory attributes for a Group . . . . .	147
13.3	Optional attributes for a Group . . . . .	148
13.4	Mandatory attributes, storage layout and supported data types for Leaves . . . . .	148

13.5 Optional attributes for Leaves . . . . .	152
<b>14 Bibliography</b>	<b>153</b>
<b>Index</b>	<b>155</b>



# LIST OF FIGURES

1.1	Figure 1: An HDF5 example with 2 subgroups, 2 tables and 1 array. . . . .	13
1.2	Figure 2: A PyTables object tree example. . . . .	15
3.1	Figure 1. The initial version of the data file for tutorial 1, with a view of the data objects. . . . .	29
3.2	Figure 2. The final version of the data file for tutorial 1. . . . .	41
3.3	Figure 3. General properties of the /detector/readout table. . . . .	42
3.4	Figure 4. Table hierarchy for tutorial 2. . . . .	46
5.1	Figure 1. Creation time per element for a 15 GB EArray and different chunksizes. . . . .	73
5.2	Figure 2. File sizes for a 15 GB EArray and different chunksizes. . . . .	74
5.3	Figure 3. Sequential access time per element for a 15 GB EArray and different chunksizes. . . . .	75
5.4	Figure 4. Random access time per element for a 15 GB EArray and different chunksizes. . . . .	76
5.5	Figure 5. Times for non-indexed complex queries in a small table with 10 millions of rows: the data fits in memory. . . . .	78
5.6	Figure 6. Times for non-indexed complex queries in a large table with 1 billion of rows: the data does not fit in memory. . . . .	79
5.7	Figure 7. Times for indexing an Int32 and Float64 column. . . . .	81
5.8	Figure 8. Sizes for an index of a Float64 column with 1 billion of rows. . . . .	82
5.9	Figure 9. Times for complex queries with a cold cache (mean of 5 first random queries) for different optimization levels. Benchmark made on a machine with Intel Core2 (64-bit) @ 3 GHz processor with RAID-0 disk storage. . . . .	83
5.10	Figure 10. Times for complex queries with a cold cache (mean of 5 first random queries) for different compressors. . . . .	84
5.11	Figure 11. Times for complex queries with a cold cache (mean of 5 first random queries) for different disk storage (SSD vs spinning disks). . . . .	85
5.12	Figure 12. Times for complex queries with a cold cache (mean of 5 first random queries) for unsorted and sorted tables. . . . .	87
5.13	Figure 13. Comparison between different compression libraries. . . . .	89
5.14	Figure 14. Comparison between different compression levels of Zlib. . . . .	90
5.15	Figure 15. Writing tables with several compressors. . . . .	91
5.16	Figure 16. Selecting values in tables with several compressors. The file is not in the OS cache. . . . .	92
5.17	Figure 17. Selecting values in tables with several compressors. The file is in the OS cache. . . . .	93
5.18	Figure 18. Writing in tables with different levels of compression. . . . .	94
5.19	Figure 19. Selecting values in tables with different levels of compression. The file is in the OS cache. . . . .	95
5.20	Figure 20. Comparison between different compression libraries with and without the shuffle filter. . . . .	96
5.21	Figure 21. Writing with different compression libraries with and without the shuffle filter. . . . .	97
5.22	Figure 22. Reading with different compression libraries with the shuffle filter. The file is not in OS cache. . . . .	98

5.23	Figure 23. Reading with different compression libraries with and without the shuffle filter. The file is in OS cache. . . . .	99
5.24	Figure 24. Writing tables with/without Psyc. . . . .	101
5.25	Figure 25. Reading tables with/without Psyc. . . . .	102

# LIST OF TABLES

5.1	Retrieval speed and memory consumption depending on the number of nodes in LRU cache. . . . .	103
8.1	Data types supported for array elements and tables columns in PyTables. . . . .	128

**Authors** Francesc Alted, Ivan Vilata, Scott Prater, Vicent Mas, Tom Hedley, Antonio Valentino, Jeffrey Whitaker, Anthony Scopatz, Josh Moore

**Copyright** © 2002, 2003, 2004 - Francesc Alted

© 2005, 2006, 2007 - Cárabos Coop. V.

© 2008, 2009, 2010 - Francesc Alted

© 2011-2012 - PyTables maintainers

**Date** October 16, 2012

**Version** 2.4.0

**Home Page** <http://www.pytables.org>

**Copyright Notice and Statement for PyTables User's Guide**

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- a. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- b. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- c. Neither the name of Francesc Alted nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.



## **Part I**

# **The PyTables Core Library**



# INTRODUCTION

La sabiduría no vale la pena si no es posible servirse de ella para inventar una nueva manera de preparar los garbanzos.

[Wisdom isn't worth anything if you can't use it to come up with a new way to cook garbanzos.]

—Gabriel García Márquez, A wise Catalan in “*Cien años de soledad*”

The goal of PyTables is to enable the end user to manipulate easily data *tables* and *array* objects in a hierarchical structure. The foundation of the underlying hierarchical data organization is the excellent HDF5 library (see [\[HDGF1\]](#)).

It should be noted that this package is not intended to serve as a complete wrapper for the entire HDF5 API, but only to provide a flexible, *very pythonic* tool to deal with (arbitrarily) large amounts of data (typically bigger than available memory) in tables and arrays organized in a hierarchical and persistent disk storage structure.

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* may seem to be 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 data acquisition systems, Internet services or scientific applications, for example) in an efficient manner that reduces demand on CPU time and I/O.

In order to emulate in Python records mapped to HDF5 C structs PyTables implements a special class so as to easily define all its fields and other properties. PyTables also provides a powerful interface to mine data in tables. Records in tables are also known in the HDF5 naming scheme as *compound* data types.

For example, you can define arbitrary tables in Python simply by declaring a class with named fields and type information, such as in the following example:

```
class Particle(IsDescription):
    name      = StringCol(16)      # 16-character String
    idnumber  = Int64Col()         # signed 64-bit integer
    ADCcount  = UInt16Col()        # unsigned short integer
    TDCcount  = UInt8Col()         # unsigned byte
    grid_i    = Int32Col()         # integer
    grid_j    = Int32Col()         # integer

    # A sub-structure (nested data-type)
    class Properties(IsDescription):
        pressure = Float32Col(shape=(2,3)) # 2-D float array (single-precision)
        energy   = Float64Col(shape=(2,3,4)) # 3-D float array (double-precision)
```

You then pass this class to the table constructor, fill its rows with your values, and save (arbitrarily large) collections of them to a file for persistent storage. After that, the data can be retrieved and post-processed quite easily with PyTables or even with another HDF5 application (in C, Fortran, Java or whatever language that provides a library to interface with HDF5).

Other important entities in PyTables are *array* objects, which are analogous to tables with the difference that all of their components are homogeneous. They come in different flavors, like *generic* (they provide a quick and fast way to deal with for numerical arrays), *enlargeable* (arrays can be extended along a single dimension) and *variable length* (each row in the array can have a different number of elements).

The next section describes the most interesting capabilities of PyTables.

## 1.1 Main Features

PyTables takes advantage of the object orientation and introspection capabilities offered by Python, the powerful data management features of HDF5, and NumPy's flexibility and Numexpr's high-performance manipulation of large sets of objects organized in a grid-like fashion to provide these features:

- *Support for table entities:* You can tailor your data adding or deleting records in your tables. Large numbers of rows (up to  $2^{63}$ , much more than will fit into memory) are supported as well.
- *Multidimensional and nested table cells:* You can declare a column to consist of values having any number of dimensions besides scalars, which is the only dimensionality allowed by the majority of relational databases. You can even declare columns that are made of other columns (of different types).
- *Indexing support for columns of tables:* Very useful if you have large tables and you want to quickly look up for values in columns satisfying some criteria.
- *Support for numerical arrays:* NumPy (see [\[NUMPY\]](#)), Numeric (see [\[NUMERIC\]](#)), and numarray (see [\[NUMARRAY\]](#)) arrays can be used as a useful complement of tables to store homogeneous data.
- *Enlargeable arrays:* You can add new elements to existing arrays on disk in any dimension you want (but only one). Besides, you are able to access just a slice of your datasets by using the powerful extended slicing mechanism, without need to load all your complete dataset in memory.
- *Variable length arrays:* The number of elements in these arrays can vary from row to row. This provides a lot of flexibility when dealing with complex data.
- *Supports a hierarchical data model:* Allows the user to clearly structure all data. PyTables builds up an *object tree* in memory that replicates the underlying file data structure. Access to objects in the file is achieved by walking through and manipulating this object tree. Besides, this object tree is built in a lazy way, for efficiency purposes.
- *User defined metadata:* Besides supporting system metadata (like the number of rows of a table, shape, flavor, etc.) the user may specify arbitrary metadata (as for example, room temperature, or protocol for IP traffic that was collected) that complement the meaning of actual data.
- *Ability to read/modify generic HDF5 files:* PyTables can access a wide range of objects in generic HDF5 files, like compound type datasets (that can be mapped to Table objects), homogeneous datasets (that can be mapped to Array objects) or variable length record datasets (that can be mapped to VArray objects). Besides, if a dataset is not supported, it will be mapped to a special *UnImplemented* class (see [The UnImplemented class](#)), that will let the user see that the data is there, although it will be unreachable (still, you will be able to access the attributes and some metadata in the dataset). With that, PyTables probably can access and *modify* most of the HDF5 files out there.
- *Data compression:* Supports data compression (using the *Zlib*, *LZO*, *bzip2* and *Blosc* compression libraries) out of the box. This is important when you have repetitive data patterns and don't want to spend time searching for an optimized way to store them (saving you time spent analyzing your data organization).
- *High performance I/O:* On modern systems storing large amounts of data, tables and array objects can be read and written at a speed only limited by the performance of the underlying I/O subsystem. Moreover, if your data is compressible, even that limit is surmountable!

- *Support of files bigger than 2 GB:* PyTables automatically inherits this capability from the underlying HDF5 library (assuming your platform supports the C long long integer, or, on Windows, `__int64`).
- *Architecture-independent:* PyTables has been carefully coded (as HDF5 itself) with little-endian/big-endian byte ordering issues in mind. So, you can write a file on a big-endian machine (like a Sparc or MIPS) and read it on other little-endian machine (like an Intel or Alpha) without problems. In addition, it has been tested successfully with 64 bit platforms (Intel-64, AMD-64, PowerPC-G5, MIPS, UltraSparc) using code generated with 64 bit aware compilers.

## 1.2 The Object Tree

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 HDF5 structure on disk. The HDF5 objects are read by walking through this object tree. You can get a good picture of what kind of data is kept in the object by examining the *metadata* nodes.

The different nodes in the object tree are instances of PyTables classes. There are several types of classes, but the most important ones are the Node, Group and Leaf classes. All nodes in a PyTables tree are instances of the Node class. The Group and Leaf classes are descendants of Node. Group instances (referred to as *groups* from now on) are a grouping structure containing instances of zero or more groups or leaves, together with supplementary metadata. Leaf instances (referred to as *leaves*) are containers for actual data and can not contain further groups or leaves. The Table, Array, CArray, EArray, VArray and UnImplemented classes are descendants of Leaf, and inherit all its properties.

Working with groups and leaves is similar in many ways to working with directories and files on a Unix filesystem, i.e. a node (file or directory) is always a *child* of one and only one group (directory), its *parent group* <sup>1</sup>. Inside of that group, the node is accessed by its *name*. As is the case with Unix directories and files, objects in the object tree are often referenced by giving their full (absolute) path names. In PyTables this full path can be specified either as string (such as `'/subgroup2/table3'`, using `/` as a parent/child separator) or as a complete object path written in a format known as the *natural name* schema (such as `file.root.subgroup2.table3`).

Support for *natural naming* is a key aspect of PyTables. It means that the names of instance variables of the node objects are the same as the names of its children <sup>2</sup>. This is very *Pythonic* and intuitive in many cases. Check the tutorial *Reading (and selecting) data in a table* for usage examples.

You should also be aware that not all the data present in a file is loaded into the object tree. The *metadata* (i.e. special data that describes the structure of the actual data) is loaded only when the user want to access to it (see later). Moreover, the actual data is not read until she request it (by calling a method on a particular node). Using the object tree (the metadata) you can retrieve information about the objects on disk such as table names, titles, column names, data types in columns, numbers of rows, or, in the case of arrays, their shapes, typecodes, etc. You can also search through the tree for specific kinds of data then read it and process it. In a certain sense, you can think of PyTables as a tool that applies the same introspection capabilities of Python objects to large amounts of data in persistent storage.

It is worth noting that PyTables sports a *metadata cache system* that loads nodes *lazily* (i.e. on-demand), and unloads nodes that have not been used for some time (following a *Least Recently Used* schema). It is important to stress out that the nodes enter the cache after they have been unreferenced (in the sense of Python reference counting), and that they can be revived (by referencing them again) directly from the cache without performing the de-serialization process from disk. This feature allows dealing with files with large hierarchies very quickly and with low memory consumption, while retaining all the powerful browsing capabilities of the previous implementation of the object tree. See [\[OPTIM\]](#) for more facts about the advantages introduced by this new metadata cache system.

To better understand the dynamic nature of this object tree entity, let's start with a sample PyTables script (which you can find in `examples/objecttree.py`) to create an HDF5 file:

<sup>1</sup> PyTables does not support hard links - for the moment.

<sup>2</sup> I got this simple but powerful idea from the excellent Objectify module by David Mertz (see [\[MERTZ\]](#)).

```
from tables import *

class Particle(IsDescription):
    identity = StringCol(itemsize=22, dflt=" ", pos=0) # character String
    idnumber = Int16Col(dflt=1, pos = 1) # short integer
    speed    = Float32Col(dflt=1, pos = 2) # single-precision

# Open a file in "w"rite mode
fileh = openFile("objecttree.h5", mode = "w")

# Get the HDF5 root group
root = fileh.root

# Create the groups
group1 = fileh.createGroup(root, "group1")
group2 = fileh.createGroup(root, "group2")

# Now, create an array in root group
array1 = fileh.createArray(root, "array1", ["string", "array"], "String array")

# Create 2 new tables in group1
table1 = fileh.createTable(group1, "table1", Particle)
table2 = fileh.createTable("/group2", "table2", Particle)

# Create the last table in group2
array2 = fileh.createArray("/group1", "array2", [1,2,3,4])

# Now, fill the tables
for table in (table1, table2):
    # Get the record object associated with the table:
    row = table.row

    # Fill the table with 10 records
    for i in xrange(10):
        # First, assign the values to the Particle record
        row['identity'] = 'This is particle: %2d' % (i)
        row['idnumber'] = i
        row['speed']    = i * 2.

        # This injects the Record values
        row.append()

    # Flush the table buffers
    table.flush()

# Finally, close the file (this also will flush all the remaining buffers!)
fileh.close()
```

This small program creates a simple HDF5 file called `objecttree.h5` with the structure that appears in [Figure 1](#)<sup>3</sup>. When the file is created, the metadata in the object tree is updated in memory while the actual data is saved to disk. When you close the file the object tree is no longer available. However, when you reopen this file the object tree will be reconstructed in memory from the metadata on disk (this is done in a lazy way, in order to load only the objects that are required by the user), allowing you to work with it in exactly the same way as when you originally created it.

In [Figure 2](#), you can see an example of the object tree created when the above `objecttree.h5` file is read (in fact, such an object tree is always created when reading any supported generic HDF5 file). It is worthwhile to take your time to

---

<sup>3</sup> We have used ViTables (see [\[VITABLES\]](#)) in order to create this snapshot.

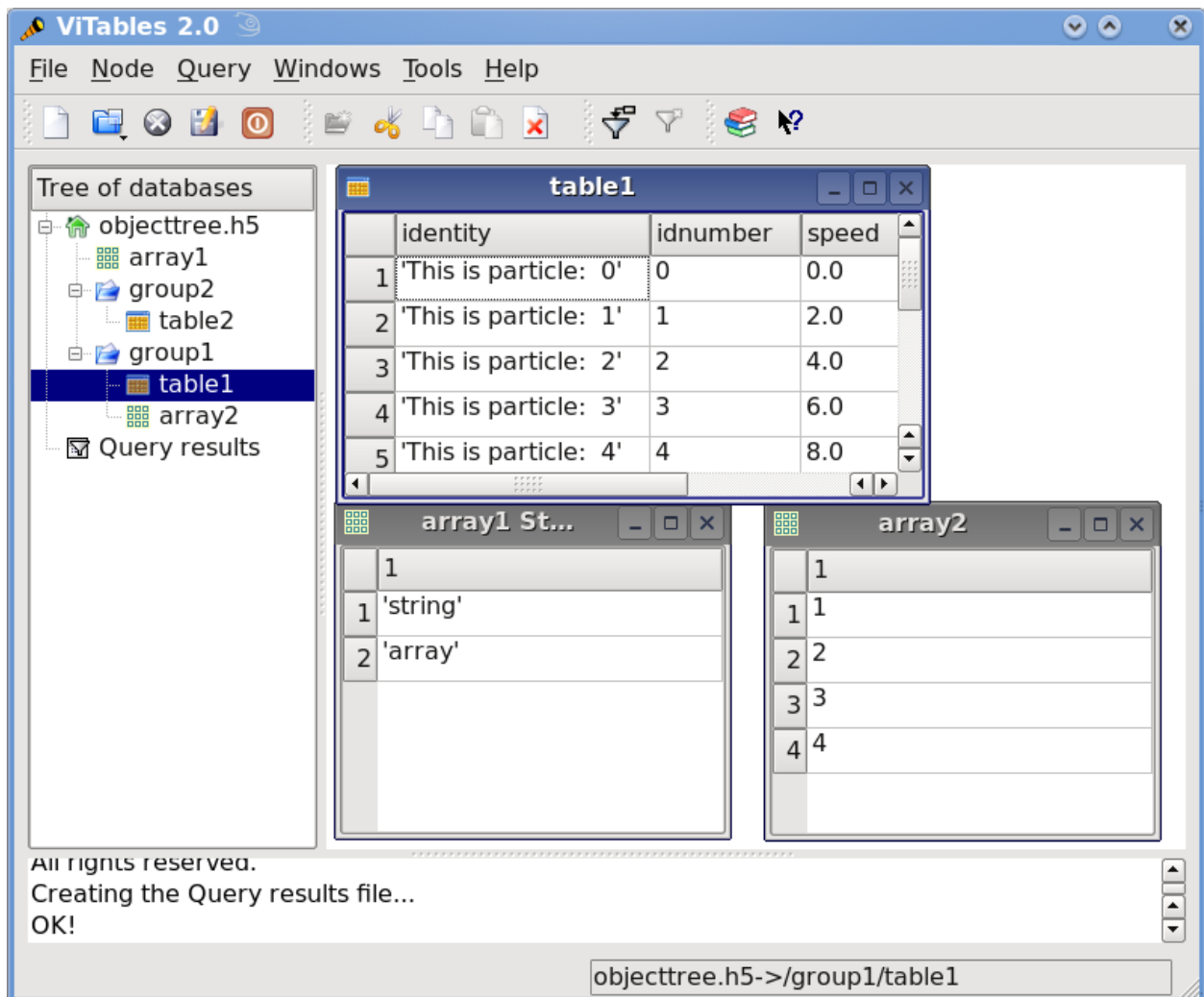


Figure 1.1: Figure 1: An HDF5 example with 2 subgroups, 2 tables and 1 array.

understand it <sup>4</sup>. It will help you understand the relationships of in-memory PyTables objects.

---

---

<sup>4</sup> Bear in mind, however, that this diagram is *not* a standard UML class diagram; it is rather meant to show the connections between the PyTables objects and some of its most important attributes and methods.

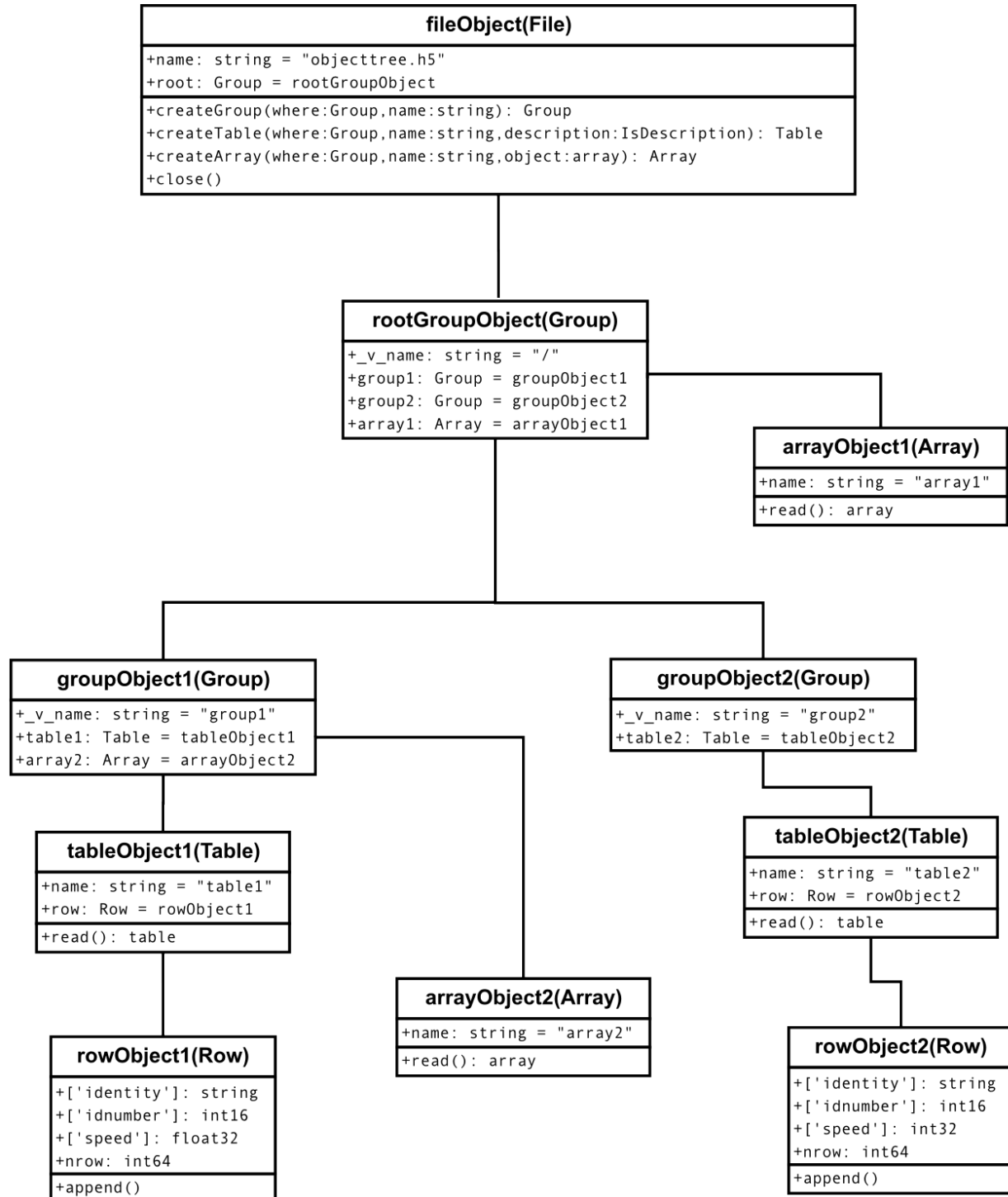


Figure 1.2: Figure 2: A PyTables object tree example.



# INSTALLATION

Make things as simple as possible, but not any simpler.

—Albert Einstein

The Python Distutils are used to build and install PyTables, so it is fairly simple to get the application up and running. If you want to install the package from sources you can go on reading to the next section.

However, if you are running Windows and want to install precompiled binaries, you can jump straight to [Binary installation \(Windows\)](#). In addition, binary packages are available for many different Linux distributions, MacOSX and other Unices. Just check the package repository for your preferred operating system.

## 2.1 Installation from source

These instructions are for both Unix/MacOS X and Windows systems. If you are using Windows, it is assumed that you have a recent version of MS Visual C++ compiler installed. A GCC compiler is assumed for Unix, but other compilers should work as well.

Extensions in PyTables have been developed in Cython (see [\[CYTHON\]](#)) and the C language. You can rebuild everything from scratch if you have Cython installed, but this is not necessary, as the Cython compiled source is included in the source distribution.

To compile PyTables you will need a recent version of Python, the HDF5 (C flavor) library from <http://www.hdfgroup.org>, and the NumPy (see [\[NUMPY\]](#)) and Numexpr (see [\[NUMEXPR\]](#)) packages. Although you won't need numarray (see [\[NUMARRAY\]](#)) or Numeric (see [\[NUMERIC\]](#)) in order to compile PyTables, they are supported; you only need a reasonably recent version of them ( $\geq 1.5.2$  for numarray and  $\geq 24.2$  for Numeric) if you plan on using them in your applications. If you already have numarray and/or Numeric installed, the test driver module will detect them and will run the tests for numarray and/or Numeric automatically.

**Warning:** The use of numarray and Numeric in PyTables is now deprecated. Support for these packages will be dropped in future versions.

### 2.1.1 Prerequisites

First, make sure that you have

- Python  $\geq 2.4$  (Python 3.x is not supported currently),
- HDF5  $\geq 1.8.4$ ,
- NumPy  $\geq 1.4.1$ ,

- [Numexpr](#)  $\geq$  1.4.1 and
- [Cython](#)  $\geq$  0.13

installed (for testing purposes, we are using [HDF5](#) 1.8.9, [NumPy](#) 1.6.1 and [Numexpr](#) 1.4.2 currently). If you don't, fetch and install them before proceeding.

---

**Note:** Users of Python 2.4.x also need to install [ctypes](#)

---

**Note:** Currently PyTables does not use [setuptools](#) so do not expect that the setup.py script automatically install all packages PyTables depends on.

---

Compile and install these packages (but see [Windows prerequisites](#) for instructions on how to install precompiled binaries if you are not willing to compile the prerequisites on Windows systems).

For compression (and possibly improved performance), you will need to install the Zlib (see [\[ZLIB\]](#)), which is also required by HDF5 as well. You may also optionally install the excellent LZO compression library (see [\[LZO\]](#) and [Compression issues](#)). The high-performance bzip2 compression library can also be used with PyTables (see [\[BZIP2\]](#)). The Blosc (see [\[BLOSC\]](#)) compression library is embedded in PyTables, so you don't need to install it separately.

### Unix

setup.py will detect HDF5, LZO, or bzip2 libraries and include files under `/usr` or `/usr/local`; this will cover most manual installations as well as installations from packages. If setup.py can not find `libhdf5`, `libhdf5` (or `liblzo`, or `libbz2` that you may wish to use) or if you have several versions of a library installed and want to use a particular one, then you can set the path to the resource in the environment, by setting the values of the `HDF5_DIR`, `LZO_DIR`, or `BZIP2_DIR` environment variables to the path to the particular resource. You may also specify the locations of the resource root directories on the setup.py command line. For example:

```
--hdf5=/stuff/hdf5-1.8.9
--lzo=/stuff/lzo-2.02
--bzip2=/stuff/bzip2-1.0.5
```

If your HDF5 library was built as a shared library not in the runtime load path, then you can specify the additional linker flags needed to find the shared library on the command line as well. For example:

```
--lflags="-Xlinker -rpath -Xlinker /stuff/hdf5-1.8.9/lib"
```

You may also want to try setting the `LD_LIBRARY_PATH` environment variable to point to the directory where the shared libraries can be found. Check your compiler and linker documentation as well as the Python Distutils documentation for the correct syntax or environment variable names. It is also possible to link with specific libraries by setting the `LIBS` environment variable:

```
LIBS="hdf5-1.8.9 ns1"
```

Finally, you can give additional flags to your compiler by passing them to the `--cflags` flag:

```
--cflags="-w -O3 -msse2"
```

In the above case, a gcc compiler is used and you instructed it to suppress all the warnings and set the level 3 of optimization. Finally, if you are running Linux in 32-bit mode, and you know that your CPU has support for SSE2 vector instructions, you may want to pass the `-msse2` flag that will accelerate Blosc operation.

### Windows

You can get ready-to-use Windows binaries and other development files for most of the following libraries from the GnuWin32 project (see [\[GNUWIN32\]](#)). In case you cannot find the LZO binaries in the GnuWin32 repository, you can find them at <http://sourceforge.net/projects/pytables/files/lzo-win>. Once you have installed the prerequisites, setup.py needs to know where the necessary library *stub* (.lib) and *header* (.h) files are installed. You can set the path to the include and dll directories for the HDF5 (mandatory) and LZO or BZIP2 (optional) libraries in the environment, by setting the values of the HDF5\_DIR, LZO\_DIR, or BZIP2\_DIR environment variables to the path to the particular resource. For example:

```
set HDF5_DIR=c:\\stuff\\hdf5-1.8.5-32bit-VS2008-IVF101\\release
set LZO_DIR=c:\\Program Files (x86)\\GnuWin32
set BZIP2_DIR=c:\\Program Files (x86)\\GnuWin32
```

You may also specify the locations of the resource root directories on the setup.py command line. For example:

```
--hdf5=c:\\stuff\\hdf5-1.8.5-32bit-VS2008-IVF101\\release
--lzo=c:\\Program Files (x86)\\GnuWin32
--bzip2=c:\\Program Files (x86)\\GnuWin32
```

### Development version (Unix)

Installation of the development version is very similar to installation from a source package (described above). There are two main differences:

1. sources have to be downloaded from the [PyTables source repository](#) hosted on [GitHub](#). Git (see [\[GIT\]](#)) is used as VCS. The following command create a local copy of latest development version sources:
- ```
$ git clone https://github.com/PyTables/PyTables.git
```
2. sources in the git repository do not include pre-built documentation and pre-generated C code of Cython extension modules. To be able to generate them, both Cython (see [\[CYTHON\]](#)) and sphinx >= 1.0.7 (see [\[SPHINX\]](#)) are mandatory prerequisites.

## 2.1.2 PyTables package installation

Once you have installed the HDF5 library and the NumPy and Numexpr packages, you can proceed with the PyTables package itself.

1. Run this command from the main PyTables distribution directory, including any extra command line arguments as discussed above:

```
$ python setup.py build_ext --inplace
```

2. To run the test suite, execute any of these commands.

**Unix** In the sh shell and its variants:

```
$ env PYTHONPATH=. python tables/tests/test_all.py
```

or, if you prefer:

```
$ env PYTHONPATH=. python -c "import tables; tables.test()"
```

---

**Note:** the syntax used above overrides original contents of the PYTHONPATH environment variable. If this is not the desired behaviour and the user just wants to add some path before existing ones, then the safest syntax to use is the following:

```
$ env PYTHONPATH=./${PYTHONPATH:+:$PYTHONPATH} python tables/tests/test_all.py
```

Please refer to your **sh** documentation for details.

---

## Windows

Open the command prompt (cmd.exe or command.com) and type:

```
> set PYTHONPATH=.;%PYTHONPATH%
> python tables\\tests\\test_all.py
```

or:

```
> set PYTHONPATH=.;%PYTHONPATH%
> python -c "import tables; tables.test()"
```

Both commands do the same thing, but the latter still works on an already installed PyTables (so, there is no need to set the `PYTHONPATH` variable for this case). However, before installation, the former is recommended because it is more flexible, as you can see below. If you would like to see verbose output from the tests simply add the `-v` flag and/or the word `verbose` to the first of the command lines above. You can also run only the tests in a particular test module. For example, to execute just the `test_types` test suite, you only have to specify it:

```
# change to backslashes for win
$ python tables/tests/test_types.py -v
```

You have other options to pass to the `test_all.py` driver:

```
# change to backslashes for win
$ python tables/tests/test_all.py --heavy
```

The command above runs every test in the test unit. Beware, it can take a lot of time, CPU and memory resources to complete:

```
# change to backslashes for win
$ python tables/tests/test_all.py --print-versions
```

The command above shows the versions for all the packages that PyTables relies on. Please be sure to include this when reporting bugs:

```
# only under Linux 2.6.x
$ python tables/tests/test_all.py --show-memory
```

The command above prints out the evolution of the memory consumption after each test module completion. It's useful for locating memory leaks in PyTables (or packages behind it). Only valid for Linux 2.6.x kernels. And last, but not least, in case a test fails, please run the failing test module again and enable the verbose output:

```
$ python tables/tests/test_<module>.py -v verbose
```

and, very important, obtain your PyTables version information by using the `--print-versions` flag (see above) and send back both outputs to developers so that we may continue improving PyTables. If you run into problems because Python can not load the HDF5 library or other shared libraries.

## Unix

Try setting the `LD_LIBRARY_PATH` or equivalent environment variable to point to the directory where the missing libraries can be found.

## Windows

Put the DLL libraries (hdf5dll.dll and, optionally, lzo1.dll and bzip2.dll) in a directory listed in your PATH environment variable. The setup.py installation program will print out a warning to that effect if the libraries can not be found.

3. To install the entire PyTables Python package, change back to the root distribution directory and run the following command (make sure you have sufficient permissions to write to the directories where the PyTables files will be installed):

```
$ python setup.py install
```

Of course, you will need super-user privileges if you want to install PyTables on a system-protected area. You can select, though, a different place to install the package using the `--prefix` flag:

```
$ python setup.py install --prefix="/home/myuser/mystuff"
```

Have in mind, however, that if you use the `--prefix` flag to install in a non-standard place, you should properly setup your PYTHONPATH environment variable, so that the Python interpreter would be able to find your new PyTables installation. You have more installation options available in the Distutils package. Issue a:

```
$ python setup.py install --help
```

for more information on that subject.

That's it! Now you can skip to the next chapter to learn how to use PyTables.

## 2.2 Binary installation (Windows)

This section is intended for installing precompiled binaries on Windows platforms. You may also find it useful for instructions on how to install *binary prerequisites* even if you want to compile PyTables itself on Windows.

**Warning:** Since PyTables 2.2b3, Windows binaries are distributed with SSE2 instructions enabled. If your processor does not have support for SSE2, then you will not be able to use these binaries.

### 2.2.1 Windows prerequisites

First, make sure that you have Python 2.4, NumPy 1.4.1 and Numexpr 1.4.1 or higher installed (PyTables binaries have been built using NumPy 1.5 and Numexpr 1.4.1). The binaries already include DLLs for HDF5 (1.8.4, 1.8.9), zlib1 (1.2.3), szlib (2.0, uncompression support only) and bzip2 (1.0.5) for Windows (2.8.0). The LZO DLL can't be included because of license issues (but read below for directives to install it if you want so).

To enable compression with the optional LZO library (see the [Compression issues](#) for hints about how it may be used to improve performance), fetch and install the LZO from <http://sourceforge.net/projects/pytables/files/lzo-win> (choose v1.x for Windows 32-bit and v2.x for Windows 64-bit). Normally, you will only need to fetch that package and copy the included lzo1.dll/lzo2.dll file in a directory in the PATH environment variable (for example C:\WINDOWS\SYSTEM) or python\_installation\_path\Lib\site-packages\tables (the last directory may not exist yet, so if you want to install the DLL there, you should do so *after* installing the PyTables package), so that it can be found by the PyTables extensions.

Please note that PyTables has internal machinery for dealing with uninstalled optional compression libraries, so, you don't need to install the LZO dynamic library if you don't want to.

### 2.2.2 PyTables package installation

Download the tables-<version>.win32-py<version>.exe file and execute it.

You can (and *you should*) test your installation by running the next commands:

```
>>> import tables
>>> tables.test()
```

on your favorite python shell. If all the tests pass (possibly with a few warnings, related to the potential unavailability of LZO lib) you already have a working, well-tested copy of PyTables installed! If any test fails, please copy the output of the error messages as well as the output of:

```
>>> tables.print_versions()
```

and mail them to the developers so that the problem can be fixed in future releases.

You can proceed now to the next chapter to see how to use PyTables.

# TUTORIALS

Seràs la clau que obre tots els panys, seràs la llum, la llum il.limitada, seràs confi on l'aurora comença,  
seràs forment, escala il.luminada!

—Lyrics: Vicent Andrés i Estellés. Music: Ovidi Montllor, Toti Soler, M'aclame a tu

This chapter consists of a series of simple yet comprehensive tutorials that will enable you to understand PyTables' main features. If you would like more information about some particular instance variable, global function, or method, look at the doc strings or go to the library reference in [Library Reference](#). If you are reading this in PDF or HTML formats, follow the corresponding hyperlink near each newly introduced entity.

Please note that throughout this document the terms *column* and *field* will be used interchangeably, as will the terms *row* and *record*.

## 3.1 Getting started

In this section, we will see how to define our own records in Python and save collections of them (i.e. a *table*) into a file. Then we will select some of the data in the table using Python cuts and create NumPy arrays to store this selection as separate objects in a tree.

In *examples/tutorial1-1.py* you will find the working version of all the code in this section. Nonetheless, this tutorial series has been written to allow you reproduce it in a Python interactive console. I encourage you to do parallel testing and inspect the created objects (variables, docs, children objects, etc.) during the course of the tutorial!

### 3.1.1 Importing tables objects

Before starting you need to import the public objects in the tables package. You normally do that by executing:

```
>>> import tables
```

This is the recommended way to import tables if you don't want to pollute your namespace. However, PyTables has a contained set of first-level primitives, so you may consider using the alternative:

```
>>> from tables import *
```

If you are going to work with NumPy arrays (and normally, you will) you will also need to import functions from the numpy package. So most PyTables programs begin with:

```
>>> import tables      # but in this tutorial we use "from tables import *"
>>> import numpy
```

### 3.1.2 Declaring a Column Descriptor

Now, imagine that we have a particle detector and we want to create a table object in order to save data retrieved from it. You need first to define the table, the number of columns it has, what kind of object is contained in each column, and so on.

Our particle detector has a TDC (Time to Digital Converter) counter with a dynamic range of 8 bits and an ADC (Analogical to Digital Converter) with a range of 16 bits. For these values, we will define 2 fields in our record object called TDCcount and ADCcount. We also want to save the grid position in which the particle has been detected, so we will add two new fields called grid\_i and grid\_j. Our instrumentation also can obtain the pressure and energy of the particle. The resolution of the pressure-gauge allows us to use a single-precision float to store pressure readings, while the energy value will need a double-precision float. Finally, to track the particle we want to assign it a name to identify the kind of the particle it is and a unique numeric identifier. So we will add two more fields: name will be a string of up to 16 characters, and idnumber will be an integer of 64 bits (to allow us to store records for extremely large numbers of particles).

Having determined our columns and their types, we can now declare a new Particle class that will contain all this information:

```
>>> from tables import *
>>> class Particle(IsDescription):
...     name      = StringCol(16)      # 16-character String
...     idnumber  = Int64Col()         # Signed 64-bit integer
...     ADCcount  = UInt16Col()        # Unsigned short integer
...     TDCcount  = UInt8Col()         # unsigned byte
...     grid_i    = Int32Col()         # 32-bit integer
...     grid_j    = Int32Col()         # 32-bit integer
...     pressure  = Float32Col()       # float (single-precision)
...     energy    = Float64Col()       # double (double-precision)
>>>
```

This definition class is self-explanatory. Basically, you declare a class variable for each field you need. As its value you assign an instance of the appropriate Col subclass, according to the kind of column defined (the data type, the length, the shape, etc). See the [The Col class and its descendants](#) for a complete description of these subclasses. See also [Supported data types in PyTables](#) for a list of data types supported by the Col constructor.

From now on, we can use Particle instances as a descriptor for our detector data table. We will see later on how to pass this object to construct the table. But first, we must create a file where all the actual data pushed into our table will be saved.

### 3.1.3 Creating a PyTables file from scratch

Use the top-level `openFile()` function to create a PyTables file:

```
>>> h5file = openFile("tutorial1.h5", mode = "w", title = "Test file")
```

`openFile()` is one of the objects imported by the `'from tables import *'` statement. Here, we are saying that we want to create a new file in the current working directory called “tutorial1.h5” in “w”rite mode and with an descriptive title string (“Test file”). This function attempts to open the file, and if successful, returns the File (see [The File Class](#)) object instance `h5file`. The root of the object tree is specified in the instance’s `root` attribute.

### 3.1.4 Creating a new group

Now, to better organize our data, we will create a group called *detector* that branches from the root node. We will save our particle data table in this group:

```
>>> group = h5file.createGroup("/", 'detector', 'Detector information')
```

Here, we have taken the File instance `h5file` and invoked its `File.createGroup()` method to create a new group called *detector* branching from “/” (another way to refer to the `h5file.root` object we mentioned above). This will create a new Group (see *The Group class*) object instance that will be assigned to the variable `group`.

### 3.1.5 Creating a new table

Let’s now create a Table (see *The Table class*) object as a branch off the newly-created group. We do that by calling the `File.createTable()` method of the `h5file` object:

```
>>> table = h5file.createTable(group, 'readout', Particle, "Readout example")
```

We create the Table instance under `group`. We assign this table the node name “*readout*”. The `Particle` class declared before is the *description* parameter (to define the columns of the table) and finally we set “*Readout example*” as the Table title. With all this information, a new Table instance is created and assigned to the variable `table`.

If you are curious about how the object tree looks right now, simply print the File instance variable `h5file`, and examine the output:

```
>>> print h5file
tutorial1.h5 (File) 'Test file'
Last modif.: 'Wed Mar  7 11:06:12 2007'
Object Tree:
/ (RootGroup) 'Test file'
/detector (Group) 'Detector information'
/detector/readout (Table(0,)) 'Readout example'
```

As you can see, a dump of the object tree is displayed. It’s easy to see the Group and Table objects we have just created. If you want more information, just type the variable containing the File instance:

```
>>> h5file
File(filename='tutorial1.h5', title='Test file', mode='w', rootUEP='/', filters=Filters(complevel=0,
/ (RootGroup) 'Test file'
/detector (Group) 'Detector information'
/detector/readout (Table(0,)) 'Readout example'
description := {
  "ADCcount": UInt16Col(shape=(), dflt=0, pos=0),
  "TDCcount": UInt8Col(shape=(), dflt=0, pos=1),
  "energy": Float64Col(shape=(), dflt=0.0, pos=2),
  "grid_i": Int32Col(shape=(), dflt=0, pos=3),
  "grid_j": Int32Col(shape=(), dflt=0, pos=4),
  "idnumber": Int64Col(shape=(), dflt=0, pos=5),
  "name": StringCol(itemsize=16, shape=(), dflt='', pos=6),
  "pressure": Float32Col(shape=(), dflt=0.0, pos=7)}
byteorder := 'little'
chunkshape := (87,)
```

More detailed information is displayed about each object in the tree. Note how `Particle`, our table descriptor class, is printed as part of the *readout* table description information. In general, you can obtain much more information about the objects and their children by just printing them. That introspection capability is very useful, and I recommend that you use it extensively.

The time has come to fill this table with some values. First we will get a pointer to the Row (see *The Row class*) instance of this table instance:

```
>>> particle = table.row
```

The row attribute of table points to the Row instance that will be used to write data rows into the table. We write data simply by assigning the Row instance the values for each row as if it were a dictionary (although it is actually an *extension class*), using the column names as keys.

Below is an example of how to write rows:

```
>>> for i in xrange(10):
...     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)
...     # Insert a new particle record
...     particle.append()
>>>
```

This code should be easy to understand. The lines inside the loop just assign values to the different columns in the Row instance particle (see [The Row class](#)). A call to its append() method writes this information to the table I/O buffer.

After we have processed all our data, we should flush the table's I/O buffer if we want to write all this data to disk. We achieve that by calling the table.flush() method:

```
>>> table.flush()
```

Remember, flushing a table is a *very important* step as it will not only help to maintain the integrity of your file, but also will free valuable memory resources (i.e. internal buffers) that your program may need for other things.

### 3.1.6 Reading (and selecting) data in a table

Ok. We have our data on disk, and now we need to access it and select from specific columns the values we are interested in. See the example below:

```
>>> table = h5file.root.detector.readout
>>> pressure = [x['pressure'] for x in table.iterrows() if x['TDCcount'] > 3 and 20 <= x['pressure']]
>>> pressure
[25.0, 36.0, 49.0]
```

The first line creates a “shortcut” to the *readout* table deeper on the object tree. As you can see, we use the *natural naming* schema to access it. We also could have used the h5file.getNode() method, as we will do later on.

You will recognize the last two lines as a Python list comprehension. It loops over the rows in *table* as they are provided by the Table.iterrows() iterator. The iterator returns values until all the data in table is exhausted. These rows are filtered using the expression:

```
x['TDCcount'] > 3 and 20 <= x['pressure'] < 50
```

So, we are selecting the values of the pressure column from filtered records to create the final list and assign it to pressure variable.

We could have used a normal for loop to accomplish the same purpose, but I find comprehension syntax to be more compact and elegant.

PyTables do offer other, more powerful ways of performing selections which may be more suitable if you have very large tables or if you need very high query speeds. They are called *in-kernel* and *indexed* queries, and you can use them through Table.where() and other related methods.

Let's use an in-kernel selection to query the name column for the same set of cuts:

```
>>> names = [ x['name'] for x in table.where(""(TDCcount > 3) & (20 <= pressure) & (pressure < 50)"))
>>> names
['Particle:      5', 'Particle:      6', 'Particle:      7']
```

In-kernel and indexed queries are not only much faster, but as you can see, they also look more compact, and are among the greatest features for PyTables, so be sure that you use them a lot. See [Condition Syntax](#) and [Accelerating your searches](#) for more information on in-kernel and indexed selections.

That's enough about selections for now. The next section will show you how to save these selected results to a file.

### 3.1.7 Creating new array objects

In order to separate the selected data from the mass of detector data, we will create a new group columns branching off the root group. Afterwards, under this group, we will create two arrays that will contain the selected data. First, we create the group:

```
>>> gcolumns = h5file.createGroup(h5file.root, "columns", "Pressure and Name")
```

Note that this time we have specified the first parameter using *natural naming* (`h5file.root`) instead of with an absolute path string ("`/`").

Now, create the first of the two Array objects we've just mentioned:

```
>>> h5file.createArray(gcolumns, 'pressure', array(pressure),
"Pressure column selection")
/cOLUMNS/pressure (Array(3,)) 'Pressure column selection'
  atom := Float64Atom(shape=(), dflt=0.0)
  maindim := 0
  flavor := 'numpy'
  byteorder := 'little'
  chunkshape := None
```

We already know the first two parameters of the `File.createArray()` methods (these are the same as the first two in `createTable`): they are the parent group *where* Array will be created and the Array instance *name*. The third parameter is the *object* we want to save to disk. In this case, it is a NumPy array that is built from the selection list we created before. The fourth parameter is the *title*.

Now, we will save the second array. It contains the list of strings we selected before: we save this object as-is, with no further conversion:

```
>>> h5file.createArray(gcolumns, 'name', names, "Name column selection")
/cOLUMNS/name (Array(3,)) 'Name column selection'
  atom := StringAtom(itemsize=16, shape=(), dflt='')
  maindim := 0
  flavor := 'python'
  byteorder := 'irrelevant'
  chunkshape := None
```

As you can see, `File.createArray()` accepts *names* (which is a regular Python list) as an *object* parameter. Actually, it accepts a variety of different regular objects (see `createArray()`) as parameters. The *flavor* attribute (see the output above) saves the original kind of object that was saved. Based on this *flavor*, PyTables will be able to retrieve exactly the same object from disk later on.

Note that in these examples, the `createArray` method returns an Array instance that is not assigned to any variable. Don't worry, this is intentional to show the kind of object we have created by displaying its representation. The Array objects have been attached to the object tree and saved to disk, as you can see if you print the complete object tree:

```
>>> print h5file
tutorial1.h5 (File) 'Test file'
Last modif.: 'Wed Mar  7 19:40:44 2007'
Object Tree:
/ (RootGroup) 'Test file'
/columns (Group) 'Pressure and Name'
/columns/name (Array(3,)) 'Name column selection'
/columns/pressure (Array(3,)) 'Pressure column selection'
/detector (Group) 'Detector information'
/detector/readout (Table(10,)) 'Readout example'
```

### 3.1.8 Closing the file and looking at its content

To finish this first tutorial, we use the close method of the h5file File object to close the file before exiting Python:

```
>>> h5file.close()
>>> ^D
$
```

You have now created your first PyTables file with a table and two arrays. You can examine it with any generic HDF5 tool, such as h5dump or h5ls. Here is what the tutorial1.h5 looks like when read with the h5ls program.

```
$ h5ls -rd tutorial1.h5
/columns                               Group
/columns/name                         Dataset {3}
  Data:
    (0) "Particle:      5", "Particle:      6", "Particle:      7"
/columns/pressure                     Dataset {3}
  Data:
    (0) 25, 36, 49
/detector                             Group
/detector/readout                     Dataset {10/Inf}
  Data:
    (0) {0, 0, 0, 0, 10, 0, "Particle:      0", 0},
    (1) {256, 1, 1, 1, 9, 17179869184, "Particle:      1", 1},
    (2) {512, 2, 256, 2, 8, 34359738368, "Particle:      2", 4},
    (3) {768, 3, 6561, 3, 7, 51539607552, "Particle:      3", 9},
    (4) {1024, 4, 65536, 4, 6, 68719476736, "Particle:      4", 16},
    (5) {1280, 5, 390625, 5, 5, 85899345920, "Particle:      5", 25},
    (6) {1536, 6, 1679616, 6, 4, 103079215104, "Particle:      6", 36},
    (7) {1792, 7, 5764801, 7, 3, 120259084288, "Particle:      7", 49},
    (8) {2048, 8, 16777216, 8, 2, 137438953472, "Particle:      8", 64},
    (9) {2304, 9, 43046721, 9, 1, 154618822656, "Particle:      9", 81}
```

Here's the output as displayed by the “ptdump” PyTables utility (located in utils/ directory).

```
$ ptdump tutorial1.h5
/ (RootGroup) 'Test file'
/columns (Group) 'Pressure and Name'
/columns/name (Array(3,)) 'Name column selection'
/columns/pressure (Array(3,)) 'Pressure column selection'
/detector (Group) 'Detector information'
/detector/readout (Table(10,)) 'Readout example'
```

You can pass the `-v` or `-d` options to ptdump if you want more verbosity. Try them out!

Also, in [Figure 1](#), you can admire how the tutorial1.h5 looks like using the ViTables graphical interface.

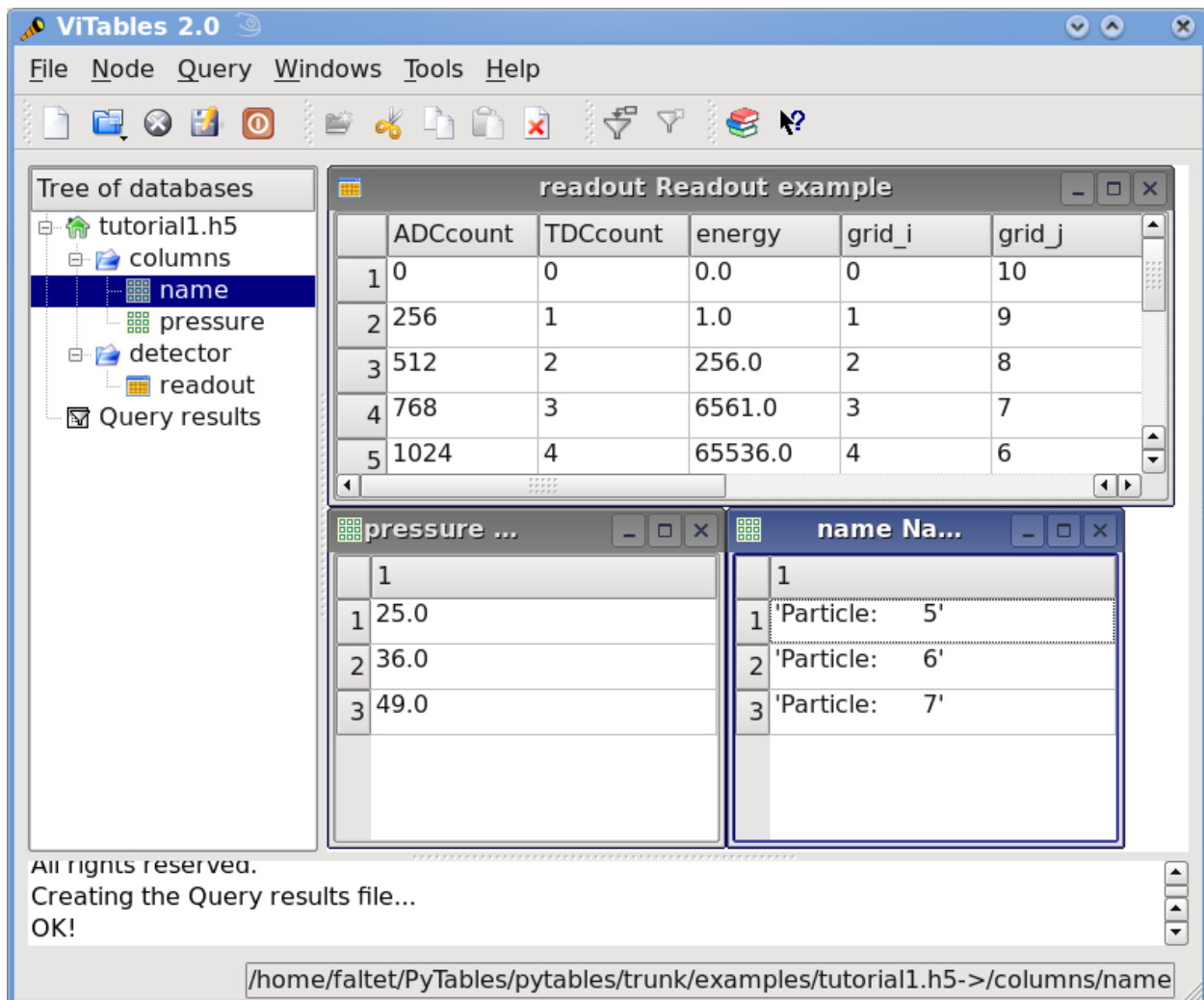


Figure 3.1: Figure 1. The initial version of the data file for tutorial 1, with a view of the data objects.

## 3.2 Browsing the *object tree*

In this section, we will learn how to browse the tree and retrieve data and also meta-information about the actual data.

In *examples/tutorial1-2.py* you will find the working version of all the code in this section. As before, you are encouraged to use a python shell and inspect the object tree during the course of the tutorial.

### 3.2.1 Traversing the object tree

Let's start by opening the file we created in last tutorial section:

```
>>> h5file = openFile("tutorial1.h5", "a")
```

This time, we have opened the file in “a”ppend mode. We use this mode to add more information to the file.

PyTables, following the Python tradition, offers powerful introspection capabilities, i.e. you can easily ask information about any component of the object tree as well as search the tree.

To start with, you can get a preliminary overview of the object tree by simply printing the existing File instance:

```
>>> print h5file
tutorial1.h5 (File) 'Test file'
Last modif.: 'Wed Mar  7 19:50:57 2007'
Object Tree:
/ (RootGroup) 'Test file'
/columns (Group) 'Pressure and Name'
/columns/name (Array(3,)) 'Name column selection'
/columns/pressure (Array(3,)) 'Pressure column selection'
/detector (Group) 'Detector information'
/detector/readout (Table(10,)) 'Readout example'
```

It looks like all of our objects are there. Now let's make use of the File iterator to see how to list all the nodes in the object tree:

```
>>> for node in h5file:
...     print node
/ (RootGroup) 'Test file'
/columns (Group) 'Pressure and Name'
/detector (Group) 'Detector information'
/columns/name (Array(3,)) 'Name column selection'
/columns/pressure (Array(3,)) 'Pressure column selection'
/detector/readout (Table(10,)) 'Readout example'
```

We can use the `File.walkGroups()` method of the File class to list only the *groups* on tree:

```
>>> for group in h5file.walkGroups():
...     print group
/ (RootGroup) 'Test file'
/columns (Group) 'Pressure and Name'
/detector (Group) 'Detector information'
```

Note that `File.walkGroups()` actually returns an *iterator*, not a list of objects. Using this iterator with the `listNodes()` method is a powerful combination. Let's see an example listing of all the arrays in the tree:

```
>>> for group in h5file.walkGroups("/"):
...     for array in h5file.listNodes(group, classname='Array'):
...         print array
/columns/name (Array(3,)) 'Name column selection'
/columns/pressure (Array(3,)) 'Pressure column selection'
```

`File.listNodes()` returns a list containing all the nodes hanging off a specific Group. If the *classname* keyword is specified, the method will filter out all instances which are not descendants of the class. We have asked for only Array instances. There exist also an iterator counterpart called `File.iterNodes()` that might be handy in some situations, like for example when dealing with groups with a large number of nodes behind it.

We can combine both calls by using the `File.walkNodes()` special method of the File object. For example:

```
>>> for array in h5file.walkNodes("/", "Array"):
...     print array
/columns/name (Array(3,)) 'Name column selection'
/columns/pressure (Array(3,)) 'Pressure column selection'
```

This is a nice shortcut when working interactively.

Finally, we will list all the Leaf, i.e. Table and Array instances (see *The Leaf class* for detailed information on Leaf class), in the /detector group. Note that only one instance of the Table class (i.e. readout) will be selected in this group (as should be the case):

```
>>> for leaf in h5file.root.detector._f_walkNodes('Leaf'):
...     print leaf
/detector/readout (Table(10,)) 'Readout example'
```

We have used a call to the `Group._f_walkNodes()` method, using the *natural naming* path specification.

Of course you can do more sophisticated node selections using these powerful methods. But first, let's take a look at some important PyTables object instance variables.

### 3.2.2 Setting and getting user attributes

PyTables provides an easy and concise way to complement the meaning of your node objects on the tree by using the `AttributeSet` class (see *The AttributeSet class*). You can access this object through the standard attribute `attrs` in Leaf nodes and `_v_attrs` in Group nodes.

For example, let's imagine that we want to save the date indicating when the data in /detector/readout table has been acquired, as well as the temperature during the gathering process:

```
>>> table = h5file.root.detector.readout
>>> table.attrs.gath_date = "Wed, 06/12/2003 18:33"
>>> table.attrs.temperature = 18.4
>>> table.attrs.temp_scale = "Celsius"
```

Now, let's set a somewhat more complex attribute in the /detector group:

```
>>> detector = h5file.root.detector
>>> detector._v_attrs.stuff = [5, (2.3, 4.5), "Integer and tuple"]
```

Note how the `AttributeSet` instance is accessed with the `_v_attrs` attribute because `detector` is a Group node. In general, you can save any standard Python data structure as an attribute node. See *The AttributeSet class* for a more detailed explanation of how they are serialized for export to disk.

Retrieving the attributes is equally simple:

```
>>> table.attrs.gath_date
'Wed, 06/12/2003 18:33'
>>> table.attrs.temperature
18.399999999999999
>>> table.attrs.temp_scale
'Celsius'
>>> detector._v_attrs.stuff
[5, (2.2999999999999998, 4.5), 'Integer and tuple']
```

You can probably guess how to delete attributes:

```
>>> del table.attrs.gath_date
```

If you want to examine the current user attribute set of /detector/table, you can print its representation (try hitting the TAB key twice if you are on a Unix Python console with the rlcompleter module active):

```
>>> table.attrs
/detector/readout._v_attrs (AttributeSet), 23 attributes:
[CLASS := 'TABLE',
 FIELD_0_FILL := 0,
 FIELD_0_NAME := 'ADCcount',
 FIELD_1_FILL := 0,
 FIELD_1_NAME := 'TDCcount',
 FIELD_2_FILL := 0.0,
 FIELD_2_NAME := 'energy',
 FIELD_3_FILL := 0,
 FIELD_3_NAME := 'grid_i',
 FIELD_4_FILL := 0,
 FIELD_4_NAME := 'grid_j',
 FIELD_5_FILL := 0,
 FIELD_5_NAME := 'idnumber',
 FIELD_6_FILL := '',
 FIELD_6_NAME := 'name',
 FIELD_7_FILL := 0.0,
 FIELD_7_NAME := 'pressure',
 FLAVOR := 'numpy',
 NROWS := 10,
 TITLE := 'Readout example',
 VERSION := '2.6',
 temp_scale := 'Celsius',
 temperature := 18.399999999999999]
```

We've got all the attributes (including the *system* attributes). You can get a list of *all* attributes or only the *user* or *system* attributes with the `_f_list()` method:

```
>>> print table.attrs._f_list("all")
['CLASS', 'FIELD_0_FILL', 'FIELD_0_NAME', 'FIELD_1_FILL', 'FIELD_1_NAME',
 'FIELD_2_FILL', 'FIELD_2_NAME', 'FIELD_3_FILL', 'FIELD_3_NAME', 'FIELD_4_FILL',
 'FIELD_4_NAME', 'FIELD_5_FILL', 'FIELD_5_NAME', 'FIELD_6_FILL', 'FIELD_6_NAME',
 'FIELD_7_FILL', 'FIELD_7_NAME', 'FLAVOR', 'NROWS', 'TITLE', 'VERSION',
 'temp_scale', 'temperature']
>>> print table.attrs._f_list("user")
['temp_scale', 'temperature']
>>> print table.attrs._f_list("sys")
['CLASS', 'FIELD_0_FILL', 'FIELD_0_NAME', 'FIELD_1_FILL', 'FIELD_1_NAME',
 'FIELD_2_FILL', 'FIELD_2_NAME', 'FIELD_3_FILL', 'FIELD_3_NAME', 'FIELD_4_FILL',
 'FIELD_4_NAME', 'FIELD_5_FILL', 'FIELD_5_NAME', 'FIELD_6_FILL', 'FIELD_6_NAME',
 'FIELD_7_FILL', 'FIELD_7_NAME', 'FLAVOR', 'NROWS', 'TITLE', 'VERSION']
```

You can also rename attributes:

```
>>> table.attrs._f_rename("temp_scale", "tempScale")
>>> print table.attrs._f_list()
['tempScale', 'temperature']
```

And, from PyTables 2.0 on, you are allowed also to set, delete or rename system attributes:

```
>>> table.attrs._f_rename("VERSION", "version")
>>> table.attrs.VERSION
```

**Traceback (most recent call last):**

```

File "<stdin>", line 1, in <module>
File "tables/attributeset.py", line 222, in __getattr__
(name, self._v__nodePath)
AttributeError: Attribute 'VERSION' does not exist in node: '/detector/readout'
>>> table.attrs.version
'2.6'

```

*Caveat emptor:* you must be careful when modifying system attributes because you may end fooling PyTables and ultimately getting unwanted behaviour. Use this only if you know what are you doing.

So, given the caveat above, we will proceed to restore the original name of VERSION attribute:

```

>>> table.attrs._f_rename("version", "VERSION")
>>> table.attrs.VERSION
'2.6'

```

Ok. that's better. If you would terminate your session now, you would be able to use the h5ls command to read the /detector/readout attributes from the file written to disk.

```

$ h5ls -vr tutorial1.h5/detector/readout
Opened "tutorial1.h5" with sec2 driver.
/detector/readout      Dataset {10/Inf}
  Attribute: CLASS      scalar
    Type:      6-byte null-terminated ASCII string
    Data:      "TABLE"
  Attribute: VERSION    scalar
    Type:      4-byte null-terminated ASCII string
    Data:      "2.6"
  Attribute: TITLE      scalar
    Type:      16-byte null-terminated ASCII string
    Data:      "Readout example"
  Attribute: NROWS      scalar
    Type:      native long long
    Data:      10
  Attribute: FIELD_0_NAME scalar
    Type:      9-byte null-terminated ASCII string
    Data:      "ADCcount"
  Attribute: FIELD_1_NAME scalar
    Type:      9-byte null-terminated ASCII string
    Data:      "TDCcount"
  Attribute: FIELD_2_NAME scalar
    Type:      7-byte null-terminated ASCII string
    Data:      "energy"
  Attribute: FIELD_3_NAME scalar
    Type:      7-byte null-terminated ASCII string
    Data:      "grid_i"
  Attribute: FIELD_4_NAME scalar
    Type:      7-byte null-terminated ASCII string
    Data:      "grid_j"
  Attribute: FIELD_5_NAME scalar
    Type:      9-byte null-terminated ASCII string
    Data:      "idnumber"
  Attribute: FIELD_6_NAME scalar
    Type:      5-byte null-terminated ASCII string
    Data:      "name"
  Attribute: FIELD_7_NAME scalar
    Type:      9-byte null-terminated ASCII string
    Data:      "pressure"

```

```
Attribute: FLAVOR      scalar
  Type:      5-byte null-terminated ASCII string
  Data:      "numpy"
Attribute: tempScale scalar
  Type:      7-byte null-terminated ASCII string
  Data:      "Celsius"
Attribute: temperature scalar
  Type:      native double
  Data:      18.4
Location: 0:1:0:1952
Links: 1
Modified: 2006-12-11 10:35:13 CET
Chunks: {85} 3995 bytes
Storage: 470 logical bytes, 3995 allocated bytes, 11.76% utilization
Type:      struct {
                "ADCcount"          +0      native unsigned short
                "TDCcount"          +2      native unsigned char
                "energy"             +3      native double
                "grid_i"             +11     native int
                "grid_j"             +15     native int
                "idnumber"           +19     native long long
                "name"               +27     16-byte null-terminated ASCII string
                "pressure"           +43     native float
            } 47 bytes
```

Attributes are a useful mechanism to add persistent (meta) information to your data.

### 3.2.3 Getting object metadata

Each object in PyTables has *metadata* information about the data in the file. Normally this *meta-information* is accessible through the node instance variables. Let's take a look at some examples:

```
>>> print "Object:", table
Object: /detector/readout (Table(10,)) 'Readout example'
>>> print "Table name:", table.name
Table name: readout
>>> print "Table title:", table.title
Table title: Readout example
>>> print "Number of rows in table:", table.nrows
Number of rows in table: 10
>>> print "Table variable names with their type and shape:"
Table variable names with their type and shape:
>>> for name in table.colnames:
...     print name, ' := %s, %s' % (table.coldtypes[name], table.coldtypes[name].shape)
ADCcount := uint16, ()
TDCcount := uint8, ()
energy := float64, ()
grid_i := int32, ()
grid_j := int32, ()
idnumber := int64, ()
name := |S16, ()
pressure := float32, ()
```

Here, the name, title, nrows, colnames and coldtypes attributes (see `Table` for a complete attribute list) of the `Table` object gives us quite a bit of information about the table data.

You can interactively retrieve general information about the public objects in PyTables by asking for help:

```

>>> help(table)
Help on Table in module tables.table:
class Table(tableExtension.Table, tables.leaf.Leaf)
| This class represents heterogeneous datasets in an HDF5 file.
|
| Tables are leaves (see the 'Leaf' class) whose data consists of a
| unidimensional sequence of *rows*, where each row contains one or
| more *fields*. Fields have an associated unique *name* and
| *position*, with the first field having position 0. All rows have
| the same fields, which are arranged in *columns*.
[snip]
|
| Instance variables
| -----
|
| The following instance variables are provided in addition to those
| in 'Leaf'. Please note that there are several 'col' dictionaries
| to ease retrieving information about a column directly by its path
| name, avoiding the need to walk through 'Table.description' or
| 'Table.cols'.
|
| autoIndex
|     Automatically keep column indexes up to date?
|
|     Setting this value states whether existing indexes should be
|     automatically updated after an append operation or recomputed
|     after an index-invalidating operation (i.e. removal and
|     modification of rows). The default is true.
[snip]
| rowsize
|     The size in bytes of each row in the table.
|
| Public methods -- reading
| -----
|
| * col(name)
| * iterrows([start][, stop][, step])
| * itersequence(sequence)
| * itersorted(sortby[, checkCSI][, start][, stop][, step])
| * read([start][, stop][, step][, field][, coords])
| * readCoordinates(coords[, field])
| * readSorted(sortby[, checkCSI][, field][, start][, stop][, step])
| * __getitem__(key)
| * __iter__()
|
| Public methods -- writing
| -----
|
| * append(rows)
| * modifyColumn([start][, stop][, step][, column][, colname])
[snip]

```

Try getting help with other object docs by yourself:

```

>>> help(h5file)
>>> help(table.removeRows)

```

To examine metadata in the `/columns/pressure` Array object:

```
>>> pressureObject = h5file.getNode("/columns", "pressure")
>>> print "Info on the object:", repr(pressureObject)
Info on the object: /columns/pressure (Array(3,)) 'Pressure column selection'
    atom := Float64Atom(shape=(), dflt=0.0)
    maindim := 0
    flavor := 'numpy'
    byteorder := 'little'
    chunkshape := None
>>> print "  shape: ==>", pressureObject.shape
    shape: ==> (3,)
>>> print "  title: ==>", pressureObject.title
    title: ==> Pressure column selection
>>> print "  atom: ==>", pressureObject.atom
    atom: ==> Float64Atom(shape=(), dflt=0.0)
```

Observe that we have used the `File.getNode()` method of the `File` class to access a node in the tree, instead of the natural naming method. Both are useful, and depending on the context you will prefer one or the other. `File.getNode()` has the advantage that it can get a node from the pathname string (as in this example) and can also act as a filter to show only nodes in a particular location that are instances of class *classname*. In general, however, I consider natural naming to be more elegant and easier to use, especially if you are using the name completion capability present in interactive console. Try this powerful combination of natural naming and completion capabilities present in most Python consoles, and see how pleasant it is to browse the object tree (well, as pleasant as such an activity can be).

If you look at the type attribute of the `pressureObject` object, you can verify that it is a “float64” array. By looking at its shape attribute, you can deduce that the array on disk is unidimensional and has 3 elements. See `Array` or the internal doc strings for the complete `Array` attribute list.

### 3.2.4 Reading data from Array objects

Once you have found the desired `Array`, use the `read()` method of the `Array` object to retrieve its data:

```
>>> pressureArray = pressureObject.read()
>>> pressureArray
array([ 25.,  36.,  49.])
>>> print "pressureArray is an object of type:", type(pressureArray)
pressureArray is an object of type: <type 'numpy.ndarray'>
>>> nameArray = h5file.root.columns.name.read()
>>> print "nameArray is an object of type:", type(nameArray)
nameArray is an object of type: <type 'list'>
>>>
>>> print "Data on arrays nameArray and pressureArray:"
Data on arrays nameArray and pressureArray:
>>> for i in range(pressureObject.shape[0]):
...     print nameArray[i], "-->", pressureArray[i]
Particle:      5 --> 25.0
Particle:      6 --> 36.0
Particle:      7 --> 49.0
```

You can see that the `Array.read()` method returns an authentic NumPy object for the `pressureObject` instance by looking at the output of the `type()` call. A `read()` of the `nameArray` object instance returns a native Python list (of strings). The type of the object saved is stored as an HDF5 attribute (named `FLAVOR`) for objects on disk. This attribute is then read as `Array` meta-information (accessible through in the `Array.attrs.FLAVOR` variable), enabling the read array to be converted into the original object. This provides a means to save a large variety of objects as arrays with the guarantee that you will be able to later recover them in their original form. See `File.createArray()` for a complete list of supported objects for the `Array` object class.

## 3.3 Committing data to tables and arrays

We have seen how to create tables and arrays and how to browse both data and metadata in the object tree. Let's examine more closely now one of the most powerful capabilities of PyTables, namely, how to modify already created tables and arrays <sup>1</sup>

### 3.3.1 Appending data to an existing table

Now, let's have a look at how we can add records to an existing table on disk. Let's use our well-known *readout* Table object and append some new values to it:

```
>>> table = h5file.root.detector.readout
>>> particle = table.row
>>> for i in xrange(10, 15):
...     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)
...     particle.append()
>>> table.flush()
```

It's the same method we used to fill a new table. PyTables knows that this table is on disk, and when you add new records, they are appended to the end of the table <sup>2</sup>.

If you look carefully at the code you will see that we have used the `table.row` attribute to create a table row and fill it with the new values. Each time that its `append()` method is called, the actual row is committed to the output buffer and the row pointer is incremented to point to the next table record. When the buffer is full, the data is saved on disk, and the buffer is reused again for the next cycle.

*Caveat emptor:* Do not forget to always call the `flush()` method after a write operation, or else your tables will not be updated!

Let's have a look at some rows in the modified table and verify that our new data has been appended:

```
>>> for r in table.iterrows():
...     print "%-16s | %11.1f | %11.4g | %6d | %6d | %8d \\\n" % \
...           (r['name'], r['pressure'], r['energy'], r['grid_i'], r['grid_j'],
...            r['TDCcount'])
Particle:      0 |      0.0 |      0 |      0 |      0 |      0 |
Particle:      1 |      1.0 |      1 |      1 |      9 |      1 |
Particle:      2 |      4.0 |    256 |      2 |      8 |      2 |
Particle:      3 |      9.0 |   6561 |      3 |      7 |      3 |
Particle:      4 |     16.0 |  6.554e+04 |      4 |      6 |      4 |
Particle:      5 |     25.0 |  3.906e+05 |      5 |      5 |      5 |
Particle:      6 |     36.0 |  1.68e+06 |      6 |      4 |      6 |
Particle:      7 |     49.0 |  5.765e+06 |      7 |      3 |      7 |
Particle:      8 |     64.0 |  1.678e+07 |      8 |      2 |      8 |
Particle:      9 |     81.0 |  4.305e+07 |      9 |      1 |      9 |
Particle:     10 |    100.0 |    1e+08 |     10 |      0 |     10 |
Particle:     11 |    121.0 |  2.144e+08 |     11 |     -1 |     11 |
Particle:     12 |    144.0 |  4.3e+08 |     12 |     -2 |     12 |
```

<sup>1</sup> Appending data to arrays is also supported, but you need to create special objects called *EArray* (see [The EArray class](#) for more info).

<sup>2</sup> Note that you can append not only scalar values to tables, but also fully multidimensional array objects.

```
Particle:      13 |      169.0 |  8.157e+08 |      13 |      -3 |      13 |
Particle:      14 |      196.0 |  1.476e+09 |      14 |      -4 |      14 |
```

### 3.3.2 Modifying data in tables

Ok, until now, we've been only reading and writing (appending) values to our tables. But there are times that you need to modify your data once you have saved it on disk (this is specially true when you need to modify the real world data to adapt your goals ;). Let's see how we can modify the values that were saved in our existing tables. We will start modifying single cells in the first row of the Particle table:

```
>>> print "Before modif-->", table[0]
Before modif--> (0, 0, 0.0, 0, 10, 0L, 'Particle:      0', 0.0)
>>> table.cols.TDCcount[0] = 1
>>> print "After modifying first row of ADCcount-->", table[0]
After modifying first row of ADCcount--> (0, 1, 0.0, 0, 10, 0L, 'Particle:      0', 0.0)
>>> table.cols.energy[0] = 2
>>> print "After modifying first row of energy-->", table[0]
After modifying first row of energy--> (0, 1, 2.0, 0, 10, 0L, 'Particle:      0', 0.0)
```

We can modify complete ranges of columns as well:

```
>>> table.cols.TDCcount[2:5] = [2,3,4]
>>> print "After modifying slice [2:5] of TDCcount-->", table[0:5]
After modifying slice [2:5] of TDCcount-->
[(0, 1, 2.0, 0, 10, 0L, 'Particle:      0', 0.0)
 (256, 1, 1.0, 1, 9, 17179869184L, 'Particle:      1', 1.0)
 (512, 2, 256.0, 2, 8, 34359738368L, 'Particle:      2', 4.0)
 (768, 3, 6561.0, 3, 7, 51539607552L, 'Particle:      3', 9.0)
 (1024, 4, 65536.0, 4, 6, 68719476736L, 'Particle:      4', 16.0)]
>>> table.cols.energy[1:9:3] = [2,3,4]
>>> print "After modifying slice [1:9:3] of energy-->", table[0:9]
After modifying slice [1:9:3] of energy-->
[(0, 1, 2.0, 0, 10, 0L, 'Particle:      0', 0.0)
 (256, 1, 2.0, 1, 9, 17179869184L, 'Particle:      1', 1.0)
 (512, 2, 256.0, 2, 8, 34359738368L, 'Particle:      2', 4.0)
 (768, 3, 6561.0, 3, 7, 51539607552L, 'Particle:      3', 9.0)
 (1024, 4, 3.0, 4, 6, 68719476736L, 'Particle:      4', 16.0)
 (1280, 5, 390625.0, 5, 5, 85899345920L, 'Particle:      5', 25.0)
 (1536, 6, 1679616.0, 6, 4, 103079215104L, 'Particle:      6', 36.0)
 (1792, 7, 4.0, 7, 3, 120259084288L, 'Particle:      7', 49.0)
 (2048, 8, 16777216.0, 8, 2, 137438953472L, 'Particle:      8', 64.0)]
```

Check that the values have been correctly modified! *Hint:* remember that column TDCcount is the second one, and that energy is the third. Look for more info on modifying columns in `Column.__setitem__()`.

PyTables also lets you modify complete sets of rows at the same time. As a demonstration of these capability, see the next example:

```
>>> table.modifyRows(start=1, step=3,
...                   rows=[(1, 2, 3.0, 4, 5, 6L, 'Particle:  None', 8.0),
...                          (2, 4, 6.0, 8, 10, 12L, 'Particle: None*2', 16.0)])
2
>>> print "After modifying the complete third row-->", table[0:5]
After modifying the complete third row-->
[(0, 1, 2.0, 0, 10, 0L, 'Particle:      0', 0.0)
 (1, 2, 3.0, 4, 5, 6L, 'Particle:  None', 8.0)
 (512, 2, 256.0, 2, 8, 34359738368L, 'Particle:      2', 4.0)]
```

```
(768, 3, 6561.0, 3, 7, 51539607552L, 'Particle:      3', 9.0)
(2, 4, 6.0, 8, 10, 12L, 'Particle: None*2', 16.0)]
```

As you can see, the `modifyRows()` call has modified the rows second and fifth, and it returned the number of modified rows.

Apart of `Table.modifyRows()`, there exists another method, called `Table.modifyColumn()` to modify specific columns as well.

Finally, it exists another way of modifying tables that is generally more handy than the described above. This new way uses the method `Row.update()` of the `Row` instance that is attached to every table, so it is meant to be used in table iterators. Look at the next example:

```
>>> for row in table.where('TDCcount <= 2'):
...     row['energy'] = row['TDCcount']*2
...     row.update()
>>> print "After modifying energy column (where TDCcount <=2)-->", table[0:4]
After modifying energy column (where TDCcount <=2)-->
[(0, 1, 2.0, 0, 10, 0L, 'Particle:      0', 0.0)
 (1, 2, 4.0, 4, 5, 6L, 'Particle:   None', 8.0)
 (512, 2, 4.0, 2, 8, 34359738368L, 'Particle:      2', 4.0)
 (768, 3, 6561.0, 3, 7, 51539607552L, 'Particle:      3', 9.0)]
```

**Note:** The authors find this way of updating tables (i.e. using `Row.update()`) to be both convenient and efficient. Please make sure to use it extensively.

### 3.3.3 Modifying data in arrays

We are going now to see how to modify data in array objects. The basic way to do this is through the use of `Array.__setitem__()` special method. Let's see at how modify data on the `pressureObject` array:

```
>>> pressureObject = h5file.root.columns.pressure
>>> print "Before modif-->", pressureObject[:]
Before modif--> [ 25.  36.  49.]
>>> pressureObject[0] = 2
>>> print "First modif-->", pressureObject[:]
First modif--> [  2.  36.  49.]
>>> pressureObject[1:3] = [2.1, 3.5]
>>> print "Second modif-->", pressureObject[:]
Second modif--> [  2.   2.1  3.5]
>>> pressureObject[:,2] = [1,2]
>>> print "Third modif-->", pressureObject[:]
Third modif--> [ 1.   2.1  2. ]
```

So, in general, you can use any combination of (multidimensional) extended slicing

With the sole exception that you cannot use negative values for step. to refer to indexes that you want to modify. See `Array.__getitem__()` for more examples on how to use extended slicing in PyTables objects.

Similarly, with an array of strings:

```
>>> nameObject = h5file.root.columns.name
>>> print "Before modif-->", nameObject[:]
Before modif--> ['Particle:      5', 'Particle:      6', 'Particle:      7']
>>> nameObject[0] = 'Particle:   None'
>>> print "First modif-->", nameObject[:]
First modif--> ['Particle:   None', 'Particle:      6', 'Particle:      7']
```

```
>>> nameObject[1:3] = ['Particle:      0', 'Particle:      1']
>>> print "Second modif-->", nameObject[:]
Second modif--> ['Particle:   None', 'Particle:      0', 'Particle:      1']
>>> nameObject[:,2] = ['Particle:     -3', 'Particle:     -5']
>>> print "Third modif-->", nameObject[:,2]
Third modif--> ['Particle:     -3', 'Particle:      0', 'Particle:     -5']
```

### 3.3.4 And finally... how to delete rows from a table

We'll finish this tutorial by deleting some rows from the table we have. Suppose that we want to delete the 5th to 9th rows (inclusive):

```
>>> table.removeRows(5,10)
5
```

`Table.removeRows()` deletes the rows in the range (start, stop). It returns the number of rows effectively removed.

We have reached the end of this first tutorial. Don't forget to close the file when you finish:

```
>>> h5file.close()
>>> ^D
$
```

In [Figure 2](#) you can see a graphical view of the PyTables file with the datasets we have just created. In [Figure 3](#), *General properties of the /detector/readout table*, are displayed the general properties of the table /detector/readout.

## 3.4 Multidimensional table cells and automatic sanity checks

Now it's time for a more real-life example (i.e. with errors in the code). We will create two groups that branch directly from the root node, Particles and Events. Then, we will put three tables in each group. In Particles we will put tables based on the Particle descriptor and in Events, the tables based the Event descriptor.

Afterwards, we will provision the tables with a number of records. Finally, we will read the newly-created table /Events/TEvent3 and select some values from it, using a comprehension list.

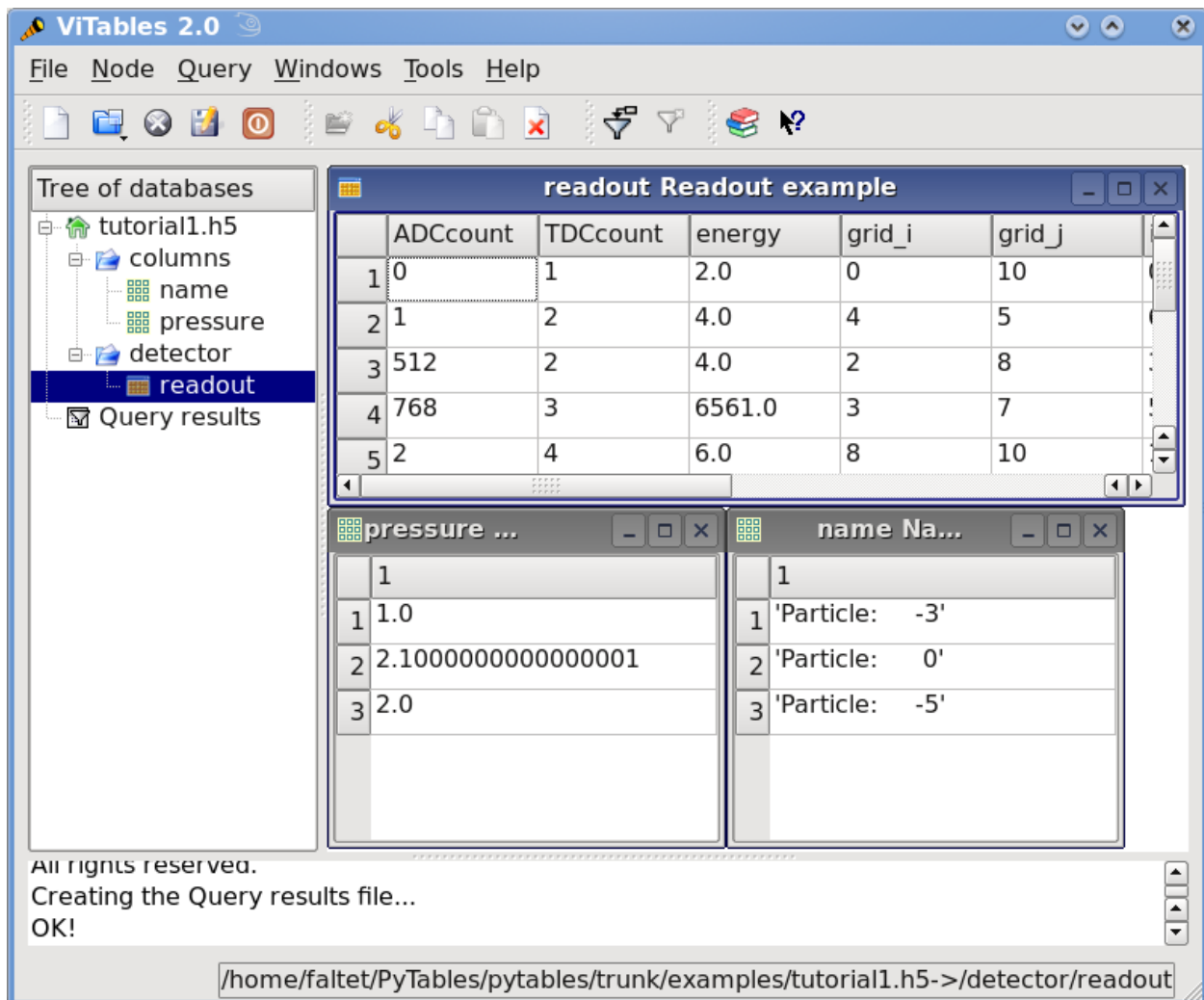
Look at the next script (you can find it in `examples/tutorial2.py`). It appears to do all of the above, but it contains some small bugs. Note that this Particle class is not directly related to the one defined in last tutorial; this class is simpler (note, however, the *multidimensional* columns called pressure and temperature).

We also introduce a new manner to describe a Table as a structured NumPy dtype (or even as a dictionary), as you can see in the Event description. See `File.createTable()` about the different kinds of descriptor objects that can be passed to this method:

```
from tables import *
from numpy import *

# Describe a particle record
class Particle(IsDescription):
    name          = StringCol(itemsize=16)    # 16-character string
    lati          = Int32Col()                 # integer
    longi         = Int32Col()                 # integer
    pressure      = Float32Col(shape=(2,3))   # array of floats (single-precision)
    temperature   = Float64Col(shape=(2,3))   # array of doubles (double-precision)

# Native NumPy dtype instances are also accepted
```

Figure 3.2: **Figure 2. The final version of the data file for tutorial 1.**

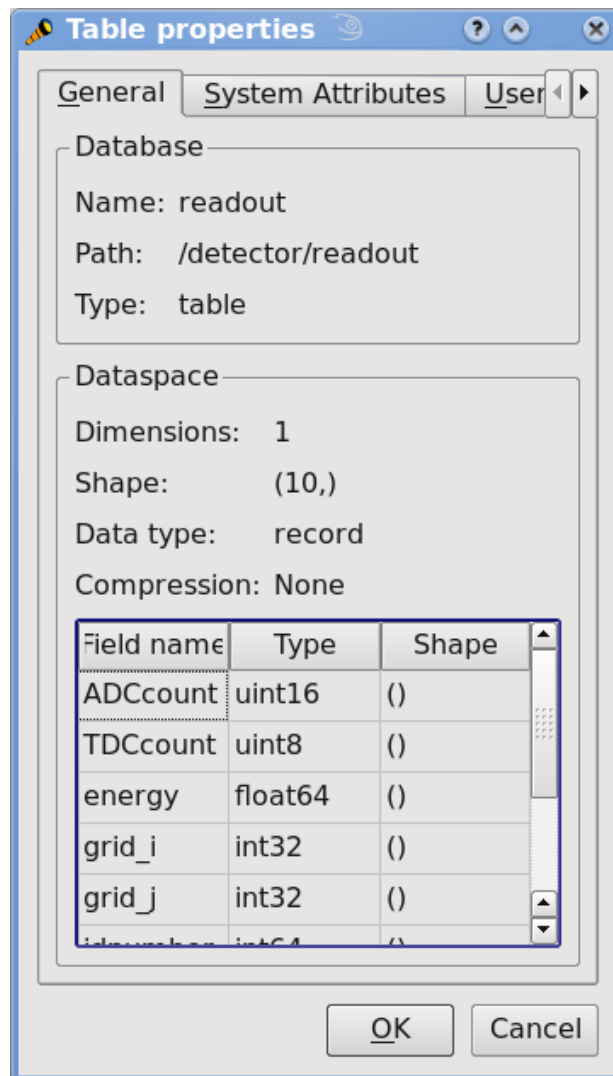


Figure 3.3: Figure 3. General properties of the /detector/readout table.

```

Event = dtype([
    ("name"      , "S16"),
    ("TDCcount"  , uint8),
    ("ADCcount"  , uint16),
    ("xcoord"    , float32),
    ("ycoord"    , float32)
])

# And dictionaries too (this defines the same structure as above)
# Event = {
#     "name"      : StringCol(itemsize=16),
#     "TDCcount"  : UInt8Col(),
#     "ADCcount"  : UInt16Col(),
#     "xcoord"    : Float32Col(),
#     "ycoord"    : Float32Col(),
# }

# Open a file in "w"rite mode
fileh = openFile("tutorial2.h5", mode = "w")

# Get the HDF5 root group
root = fileh.root

# Create the groups:
for groupname in ("Particles", "Events"):
    group = fileh.createGroup(root, groupname)

# Now, create and fill the tables in Particles group
gparticles = root.Particles

# Create 3 new tables
for tablename in ("TParticle1", "TParticle2", "TParticle3"):
    # Create a table
    table = fileh.createTable("/Particles", tablename, Particle, "Particles: "+tablename)

    # Get the record object associated with the table:
    particle = table.row

    # Fill the table with 257 particles
    for i in xrange(257):
        # First, assign the values to the Particle record
        particle['name'] = 'Particle: %6d' % (i)
        particle['lati'] = i
        particle['longi'] = 10 - i

        ##### Detectable errors start here. Play with them!
        particle['pressure'] = array(i*arange(2*3)).reshape((2,4)) # Incorrect
        #particle['pressure'] = array(i*arange(2*3)).reshape((2,3)) # Correct
        ##### End of errors

        particle['temperature'] = (i**2) # Broadcasting

        # This injects the Record values
        particle.append()

    # Flush the table buffers
    table.flush()

```

```
# Now, go for Events:
for tablename in ("TEvent1", "TEvent2", "TEvent3"):
    # Create a table in Events group
    table = fileh.createTable(root.Events, tablename, Event, "Events: "+tablename)

    # Get the record object associated with the table:
    event = table.row

    # Fill the table with 257 events
    for i in xrange(257):
        # First, assign the values to the Event record
        event['name'] = 'Event: %6d' % (i)
        event['TDCcount'] = i % (1<<8)    # Correct range

        ##### Detectable errors start here. Play with them!
        event['xcoor'] = float(i**2)      # Wrong spelling
        #event['xcoord'] = float(i**2)    # Correct spelling
        event['ADCcount'] = "sss"         # Wrong type
        #event['ADCcount'] = i * 2        # Correct type
        ##### End of errors

        event['ycoord'] = float(i)**4

        # This injects the Record values
        event.append()

    # Flush the buffers
    table.flush()

# Read the records from table "/Events/TEvent3" and select some
table = root.Events.TEvent3
e = [ p['TDCcount'] for p in table if p['ADCcount'] < 20 and 4 <= p['TDCcount'] < 15 ]
print "Last record ==>", p
print "Selected values ==>", e
print "Total selected records ==> ", len(e)

# Finally, close the file (this also will flush all the remaining buffers!)
fileh.close()
```

### 3.4.1 Shape checking

If you look at the code carefully, you'll see that it won't work. You will get the following error.

```
$ python tutorial2.py
Traceback (most recent call last):
  File "tutorial2.py", line 51, in ?
    particle['pressure'] = array(i*arange(2*3), shape=(2,4)) # Incorrect
  File ".../numarray/numarraycore.py", line 400, in array
    a.setshape(shape)
  File ".../numarray/generic.py", line 702, in setshape
    raise ValueError("New shape is not consistent with the old shape")
ValueError: New shape is not consistent with the old shape
```

This error indicates that you are trying to assign an array with an incompatible shape to a table cell. Looking at the source, we see that we were trying to assign an array of shape (2,4) to a pressure element, which was defined with the shape (2,3).

In general, these kinds of operations are forbidden, with one valid exception: when you assign a *scalar* value to a multidimensional column cell, all the cell elements are populated with the value of the scalar. For example:

```
particle['temperature'] = (i**2)      # Broadcasting
```

The value `i**2` is assigned to all the elements of the temperature table cell. This capability is provided by the NumPy package and is known as *broadcasting*.

### 3.4.2 Field name checking

After fixing the previous error and rerunning the program, we encounter another error.

```
$ python tutorial2.py
Traceback (most recent call last):
  File "tutorial2.py", line 73, in ?
    event['xcoor'] = float(i**2)      # Wrong spelling
  File "tableExtension.pyx", line 1094, in tableExtension.Row.__setitem__
  File "tableExtension.pyx", line 127, in tableExtension.getNestedFieldCache
  File "utilsExtension.pyx", line 331, in utilsExtension.getNestedField
KeyError: 'no such column: xcoor'
```

This error indicates that we are attempting to assign a value to a non-existent field in the *event* table object. By looking carefully at the Event class attributes, we see that we misspelled the *xcoord* field (we wrote *xcoor* instead). This is unusual behavior for Python, as normally when you assign a value to a non-existent instance variable, Python creates a new variable with that name. Such a feature can be dangerous when dealing with an object that contains a fixed list of field names. PyTables checks that the field exists and raises a `KeyError` if the check fails.

### 3.4.3 Data type checking

Finally, the last issue which we will find here is a `TypeError` exception.

```
$ python tutorial2.py
Traceback (most recent call last):
  File "tutorial2.py", line 75, in ?
    event['ADCCount'] = "sss"        # Wrong type
  File "tableExtension.pyx", line 1111, in tableExtension.Row.__setitem__
TypeError: invalid type (<type 'str'>) for column 'ADCCount'
```

And, if we change the affected line to read:

```
event.ADCCount = i * 2              # Correct type
```

we will see that the script ends well.

You can see the structure created with this (corrected) script in [Figure 4](#). In particular, note the multidimensional column cells in table /Particles/TParticle2.

## 3.5 Using links for more convenient access to nodes

Links are special nodes that can be used to create additional paths to your existing nodes. PyTables supports three kinds of links: hard links, soft links (aka symbolic links) and external links.

Hard links let the user create additional paths to access another node in the same file, and once created, they are indistinguishable from the referred node object, except that they have different paths in the object tree. For example, if the referred node is, say, a Table object, then the new hard link will become a Table object itself. From this point on,

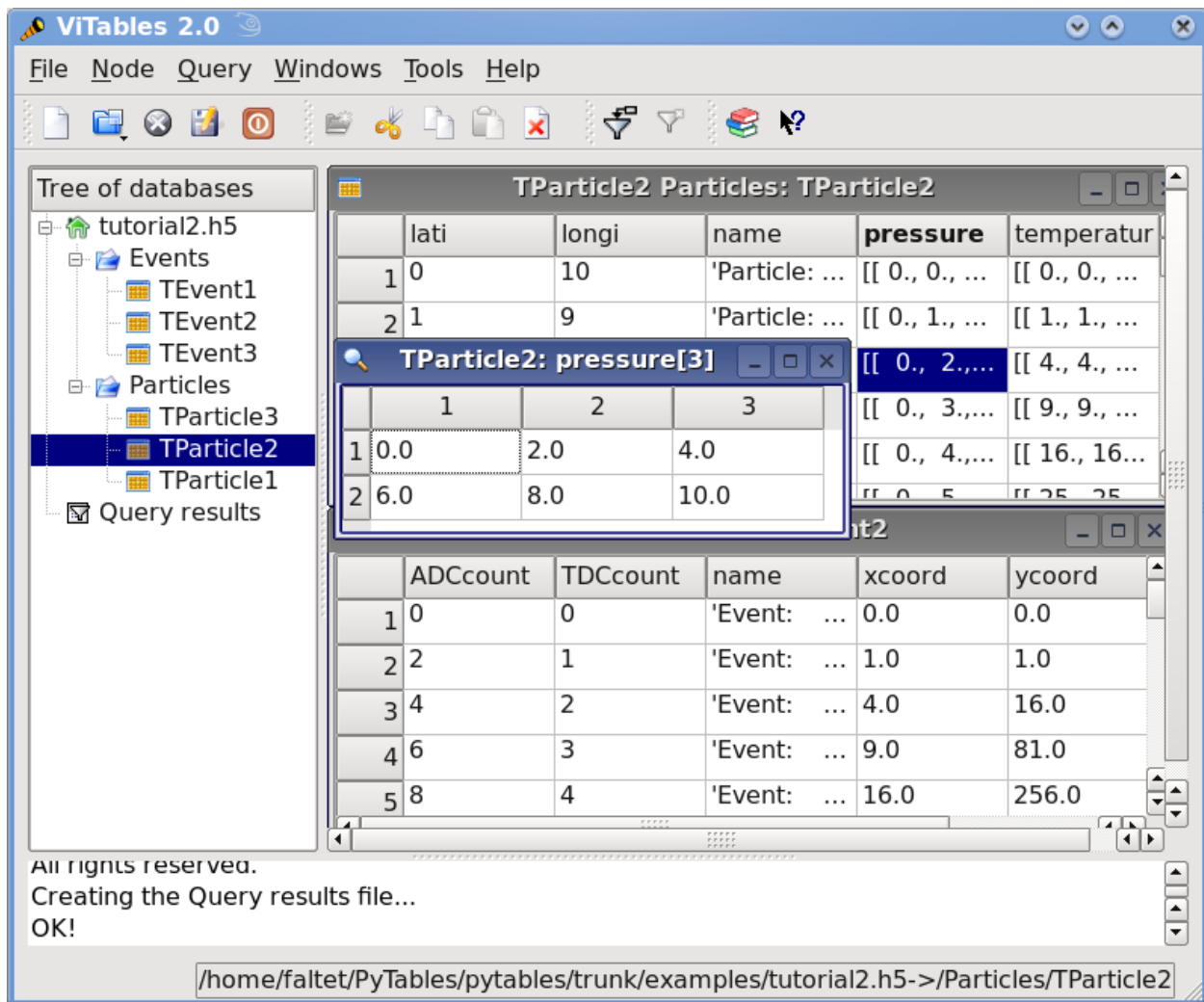


Figure 3.4: Figure 4. Table hierarchy for tutorial 2.

you will be able to access the same Table object from two different paths: the original one and the new hard link path. If you delete one path to the table, you will be able to reach it via the other path.

Soft links are similar to hard links, but they keep their own personality. When you create a soft link to another node, you will get a new SoftLink object that *refers* to that node. However, in order to access the referred node, you need to *dereference* it.

Finally, external links are like soft links, with the difference that these are meant to point to nodes in *external* files instead of nodes in the same file. They are represented by the ExternalLink class and, like soft links, you need to dereference them in order to get access to the pointed node.

### 3.5.1 Interactive example

Now we are going to learn how to deal with links. You can find the code used in this section in `examples/links.py`.

First, let's create a file with some group structure:

```
>>> import tables as tb
>>> f1 = tb.openFile('links1.h5', 'w')
>>> g1 = f1.createGroup('/', 'g1')
>>> g2 = f1.createGroup(g1, 'g2')
```

Now, we will put some datasets on the /g1 and /g1/g2 groups:

```
>>> a1 = f1.createCArray(g1, 'a1', tb.Int64Atom(), shape=(10000,))
>>> t1 = f1.createTable(g2, 't1', {'f1': tb.IntCol(), 'f2': tb.FloatCol()})
```

We can start the party now. We are going to create a new group, say /g1, where we will put our links and will start creating one hard link too:

```
>>> g1 = f1.createGroup('/', 'g1')
>>> ht = f1.createHardLink(g1, 'ht', '/g1/g2/t1') # ht points to t1
>>> print "%s is a hard link to: %s" % (ht, t1)
'/g1/ht (Table(0,)) ' is a hard link to: '/g1/g2/t1 (Table(0,)) '
```

You can see how we've created a hard link in /g1/ht which is pointing to the existing table in /g1/g2/t1. Have look at how the hard link is represented; it looks like a table, and actually, it is an *real* table. We have two different paths to access that table, the original /g1/g2/t1 and the new one /g1/ht. If we remove the original path we still can reach the table by using the new path:

```
>>> t1.remove()
>>> print "table continues to be accessible in: %s" % f1.getNode('/g1/ht')
table continues to be accessible in: '/g1/ht (Table(0,)) '
```

So far so good. Now, let's create a couple of soft links:

```
>>> la1 = f1.createSoftLink(g1, 'la1', '/g1/a1') # la1 points to a1
>>> print "%s is a soft link to: %s" % (la1, la1.target)
'/g1/la1 (SoftLink) -> /g1/a1' is a soft link to: '/g1/a1'
>>> lt = f1.createSoftLink(g1, 'lt', '/g1/g2/t1') # lt points to t1
>>> print "%s is a soft link to: %s" % (lt, lt.target)
'/g1/lt (SoftLink) -> /g1/g2/t1 (dangling)' is a soft link to: '/g1/g2/t1'
```

Okay, we see how the first link /g1/la1 points to the array /g1/a1. Notice how the link prints as a SoftLink, and how the referred node is stored in the target instance attribute. The second link (/g1/lt) pointing to /g1/g2/t1 also has been created successfully, but by better inspecting the string representation of it, we see that is labeled as '(dangling)'. Why is this? Well, you should remember that we recently removed the /g1/g2/t1 path to access table t1. When printing it,

the object knows that it points to *nowhere* and reports this. This is a nice way to quickly know whether a soft link points to an existing node or not.

So, let's re-create the removed path to t1 table:

```
>>> t1 = f1.createHardLink('/g1/g2', 't1', '/g1/ht')
>>> print "'%s' is not dangling anymore" % (lt,)
'/g1/lt (SoftLink) -> /g1/g2/t1' is not dangling anymore
```

and the soft link is pointing to an existing node now.

Of course, for soft links to serve any actual purpose we need a way to get the pointed node. It happens that soft links are callable, and that's the way to get the referred nodes back:

```
>>> plt = lt()
>>> print "dereferenced lt node: '%s'" % plt
dereferenced lt node: '/g1/g2/t1 (Table(0,))'
>>> pla1 = lal()
>>> print "dereferenced lal node: '%s'" % pla1
dereferenced lal node: '/g1/a1 (CArray(10000,))'
```

Now, plt is a Python reference to the t1 table while pla1 refers to the a1 array. Easy, uh?

Let's suppose now that a1 is an array whose access speed is critical for our application. One possible solution is to move the entire file into a faster disk, say, a solid state disk so that access latencies can be reduced quite a lot. However, it happens that our file is too big to fit into our shiny new (although small in capacity) SSD disk. A solution is to copy just the a1 array into a separate file that would fit into our SSD disk. However, our application would be able to handle two files instead of only one, adding significantly more complexity, which is not a good thing.

External links to the rescue! As we've already said, external links are like soft links, but they are designed to link objects in external files. Back to our problem, let's copy the a1 array into a different file:

```
>>> f2 = tb.openFile('links2.h5', 'w')
>>> new_a1 = a1.copy(f2.root, 'a1')
>>> f2.close() # close the other file
```

And now, we can remove the existing soft link and create the external link in its place:

```
>>> lal.remove()
>>> lal = f1.createExternalLink(g1, 'lal', 'links2.h5:/a1')
>>> print "'%s' is an external link to: '%s'" % (lal, lal.target)
'/g1/lal (ExternalLink) -> links2.h5:/a1' is an external link to: 'links2.h5:/a1'
```

Let's try dereferencing it:

```
>>> new_a1 = lal() # dereferencing lal returns a1 in links2.h5
>>> print "dereferenced lal node: '%s'" % new_a1
dereferenced lal node: '/a1 (CArray(10000,))'
```

Well, it seems like we can access the external node. But just to make sure that the node is in the other file:

```
>>> print "new_a1 file:", new_a1._v_file.filename
new_a1 file: links2.h5
```

Okay, the node is definitely in the external file. So, you won't have to worry about your application: it will work exactly the same no matter the link is internal (soft) or external.

Finally, here it is a dump of the objects in the final file, just to get a better idea of what we ended with:

```
>>> f1.close()
>>> exit()
$ ptdump links1.h5
```

```

/ (RootGroup) ''
/g1 (Group) ''
/g1/a1 (CArray(10000,)) ''
/g1 (Group) ''
/g1/ht (Table(0,)) ''
/g1/la1 (ExternalLink) -> links2.h5:/a1
/g1/lt (SoftLink) -> /g1/g2/t1
/g1/g2 (Group) ''
/g1/g2/t1 (Table(0,)) ''

```

This ends this tutorial. I hope it helped you to appreciate how useful links can be. I'm sure you will find other ways in which you can use links that better fit your own needs.

## 3.6 Exercising the Undo/Redo feature

PyTables has integrated support for undoing and/or redoing actions. This functionality lets you put marks in specific places of your hierarchy manipulation operations, so that you can make your HDF5 file pop back (*undo*) to a specific mark (for example for inspecting how your hierarchy looked at that point). You can also go forward to a more recent marker (*redo*). You can even do jumps to the marker you want using just one instruction as we will see shortly.

You can undo/redo all the operations that are related to object tree management, like creating, deleting, moving or renaming nodes (or complete sub-hierarchies) inside a given object tree. You can also undo/redo operations (i.e. creation, deletion or modification) of persistent node attributes. However, when actions include *internal* modifications of datasets (that includes `Table.append`, `Table.modifyRows` or `Table.removeRows` among others), they cannot be undone/redone currently.

This capability can be useful in many situations, like for example when doing simulations with multiple branches. When you have to choose a path to follow in such a situation, you can put a mark there and, if the simulation is not going well, you can go back to that mark and start another path. Other possible application is defining coarse-grained operations which operate in a transactional-like way, i.e. which return the database to its previous state if the operation finds some kind of problem while running. You can probably devise many other scenarios where the Undo/Redo feature can be useful to you<sup>3</sup>.

### 3.6.1 A basic example

In this section, we are going to show the basic behavior of the Undo/Redo feature. You can find the code used in this example in `examples/tutorial3-1.py`. A somewhat more complex example will be explained in the next section.

First, let's create a file:

```

>>> import tables
>>> fileh = tables.openFile("tutorial3-1.h5", "w", title="Undo/Redo demo 1")

```

And now, activate the Undo/Redo feature with the method `File.enableUndo()` of `File`:

```

>>> fileh.enableUndo()

```

From now on, all our actions will be logged internally by PyTables. Now, we are going to create a node (in this case an Array object):

```

>>> one = fileh.createArray('/', 'anarray', [3,4], "An array")

```

Now, mark this point:

---

<sup>3</sup> You can even *hide* nodes temporarily. Will you be able to find out how?

```
>>> fileh.mark()
1
```

We have marked the current point in the sequence of actions. In addition, the `mark()` method has returned the identifier assigned to this new mark, that is 1 (mark #0 is reserved for the implicit mark at the beginning of the action log). In the next section we will see that you can also assign a *name* to a mark (see `File.mark()` for more info on `mark()`). Now, we are going to create another array:

```
>>> another = fileh.createArray('/', 'anotherarray', [4,5], "Another array")
```

Right. Now, we can start doing funny things. Let's say that we want to pop back to the previous mark (that whose value was 1, do you remember?). Let's introduce the `undo()` method (see `File.undo()`):

```
>>> fileh.undo()
```

Fine, what do you think it happened? Well, let's have a look at the object tree:

```
>>> print fileh
tutorial3-1.h5 (File) 'Undo/Redo demo 1'
Last modif.: 'Tue Mar 13 11:43:55 2007'
Object Tree:
/ (RootGroup) 'Undo/Redo demo 1'
/anarray (Array(2,)) 'An array'
```

What happened with the `/anotherarray` node we've just created? You guess it, it has disappeared because it was created *after* the mark 1. If you are curious enough you may well ask where it has gone. Well, it has not been deleted completely; it has been just moved into a special, hidden, group of PyTables that renders it invisible and waiting for a chance to be reborn.

Now, unwind once more, and look at the object tree:

```
>>> fileh.undo()
>>> print fileh
tutorial3-1.h5 (File) 'Undo/Redo demo 1'
Last modif.: 'Tue Mar 13 11:43:55 2007'
Object Tree:
/ (RootGroup) 'Undo/Redo demo 1'
```

Oops, `/anarray` has disappeared as well!. Don't worry, it will revisit us very shortly. So, you might be somewhat lost right now; in which mark are we?. Let's ask the `File.getCurrentMark()` method in the file handler:

```
>>> print fileh.getCurrentMark()
0
```

So we are at mark #0, remember? Mark #0 is an implicit mark that is created when you start the log of actions when calling `File.enableUndo()`. Fine, but you are missing your too-young-to-die arrays. What can we do about that? `File.redo()` to the rescue:

```
>>> fileh.redo()
>>> print fileh
tutorial3-1.h5 (File) 'Undo/Redo demo 1'
Last modif.: 'Tue Mar 13 11:43:55 2007'
Object Tree:
/ (RootGroup) 'Undo/Redo demo 1'
/anarray (Array(2,)) 'An array'
```

Great! The `/anarray` array has come into life again. Just check that it is alive and well:

```
>>> fileh.root.anarray.read()
[3, 4]
```

```
>>> fileh.root.anarray.title
'An array'
```

Well, it looks pretty similar than in its previous life; what's more, it is exactly the same object!:

```
>>> fileh.root.anarray is one
True
```

It just was moved to the the hidden group and back again, but that's all! That's kind of fun, so we are going to do the same with /anotherarray:

```
>>> fileh.redo()
>>> print fileh
tutorial3-1.h5 (File) 'Undo/Redo demo 1'
Last modif.: 'Tue Mar 13 11:43:55 2007'
Object Tree:
/ (RootGroup) 'Undo/Redo demo 1'
/anarray (Array(2,)) 'An array'
/anotherarray (Array(2,)) 'Another array'
```

Welcome back, /anotherarray! Just a couple of sanity checks:

```
>>> assert fileh.root.anotherarray.read() == [4,5]
>>> assert fileh.root.anotherarray.title == "Another array"
>>> fileh.root.anotherarray is another
True
```

Nice, you managed to turn your data back into life. Congratulations! But wait, do not forget to close your action log when you don't need this feature anymore:

```
>>> fileh.disableUndo()
```

That will allow you to continue working with your data without actually requiring PyTables to keep track of all your actions, and more importantly, allowing your objects to die completely if they have to, not requiring to keep them anywhere, and hence saving process time and space in your database file.

### 3.6.2 A more complete example

Now, time for a somewhat more sophisticated demonstration of the Undo/Redo feature. In it, several marks will be set in different parts of the code flow and we will see how to jump between these marks with just one method call. You can find the code used in this example in `examples/tutorial3-2.py`

Let's introduce the first part of the code:

```
import tables

# Create an HDF5 file
fileh = tables.openFile('tutorial3-2.h5', 'w', title='Undo/Redo demo 2')

#'-***** enable undo/redo log  -*****-'
fileh.enableUndo()

# Start undoable operations
fileh.createArray('/', 'otherarray1', [3,4], 'Another array 1')
fileh.createGroup('/', 'agroup', 'Group 1')

# Create a 'first' mark
fileh.mark('first')
```

```
fileh.createArray('/agroup', 'otherarray2', [4,5], 'Another array 2')
fileh.createGroup('/agroup', 'agroup2', 'Group 2')

# Create a 'second' mark
fileh.mark('second')
fileh.createArray('/agroup/agroup2', 'otherarray3', [5,6], 'Another array 3')

# Create a 'third' mark
fileh.mark('third')
fileh.createArray('/', 'otherarray4', [6,7], 'Another array 4')
fileh.createArray('/agroup', 'otherarray5', [7,8], 'Another array 5')
```

You can see how we have set several marks interspersed in the code flow, representing different states of the database. Also, note that we have assigned *names* to these marks, namely 'first', 'second' and 'third'.

Now, start doing some jumps back and forth in the states of the database:

```
# Now go to mark 'first'
fileh.goto('first')
assert '/otherarray1' in fileh
assert '/agroup' in fileh
assert '/agroup/agroup2' not in fileh
assert '/agroup/otherarray2' not in fileh
assert '/agroup/agroup2/otherarray3' not in fileh
assert '/otherarray4' not in fileh
assert '/agroup/otherarray5' not in fileh

# Go to mark 'third'
fileh.goto('third')
assert '/otherarray1' in fileh
assert '/agroup' in fileh
assert '/agroup/agroup2' in fileh
assert '/agroup/otherarray2' in fileh
assert '/agroup/agroup2/otherarray3' in fileh
assert '/otherarray4' not in fileh
assert '/agroup/otherarray5' not in fileh

# Now go to mark 'second'
fileh.goto('second')
assert '/otherarray1' in fileh
assert '/agroup' in fileh
assert '/agroup/agroup2' in fileh
assert '/agroup/otherarray2' in fileh
assert '/agroup/agroup2/otherarray3' not in fileh
assert '/otherarray4' not in fileh
assert '/agroup/otherarray5' not in fileh
```

Well, the code above shows how easy is to jump to a certain mark in the database by using the `File.goto()` method.

There are also a couple of implicit marks for going to the beginning or the end of the saved states: 0 and -1. Going to mark #0 means go to the beginning of the saved actions, that is, when method `fileh.enableUndo()` was called. Going to mark #-1 means go to the last recorded action, that is the last action in the code flow.

Let's see what happens when going to the end of the action log:

```
# Go to the end
fileh.goto(-1)
assert '/otherarray1' in fileh
assert '/agroup' in fileh
assert '/agroup/agroup2' in fileh
```

```

assert '/agroup/otherarray2' in fileh
assert '/agroup/agroup2/otherarray3' in fileh
assert '/otherarray4' in fileh
assert '/agroup/otherarray5' in fileh

# Check that objects have come back to life in a sane state
assert fileh.root.otherarray1.read() == [3,4]
assert fileh.root.agroup.otherarray2.read() == [4,5]
assert fileh.root.agroup.agroup2.otherarray3.read() == [5,6]
assert fileh.root.otherarray4.read() == [6,7]
assert fileh.root.agroup.otherarray5.read() == [7,8]

```

Try yourself going to the beginning of the action log (remember, the mark #0) and check the contents of the object tree.

We have nearly finished this demonstration. As always, do not forget to close the action log as well as the database:

```

#'-*-*-*-*-*-*-*-*- disable undo/redo log -*-*-*-*-*-*-*-*-
fileh.disableUndo()
# Close the file
fileh.close()

```

You might want to check other examples on Undo/Redo feature that appear in `examples/undo-redo.py`.

## 3.7 Using enumerated types

PyTables includes support for handling enumerated types. Those types are defined by providing an exhaustive *set* or *list* of possible, named values for a variable of that type. Enumerated variables of the same type are usually compared between them for equality and sometimes for order, but are not usually operated upon.

Enumerated values have an associated *name* and *concrete value*. Every name is unique and so are concrete values. An enumerated variable always takes the concrete value, not its name. Usually, the concrete value is not used directly, and frequently it is entirely irrelevant. For the same reason, an enumerated variable is not usually compared with concrete values out of its enumerated type. For that kind of use, standard variables and constants are more adequate.

PyTables provides the Enum (see *The Enum class*) class to provide support for enumerated types. Each instance of Enum is an enumerated type (or *enumeration*). For example, let us create an enumeration of colors

All these examples can be found in `examples/enum.py`:

```

>>> import tables
>>> colorList = ['red', 'green', 'blue', 'white', 'black']
>>> colors = tables.Enum(colorList)

```

Here we used a simple list giving the names of enumerated values, but we left the choice of concrete values up to the Enum class. Let us see the enumerated pairs to check those values:

```

>>> print "Colors:", [v for v in colors]
Colors: [('blue', 2), ('black', 4), ('white', 3), ('green', 1), ('red', 0)]

```

Names have been given automatic integer concrete values. We can iterate over the values in an enumeration, but we will usually be more interested in accessing single values. We can get the concrete value associated with a name by accessing it as an attribute or as an item (the later can be useful for names not resembling Python identifiers):

```

>>> print "Value of 'red' and 'white':", (colors.red, colors.white)
Value of 'red' and 'white': (0, 3)
>>> print "Value of 'yellow':", colors.yellow
Value of 'yellow':

```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File ".../tables/misc/enum.py", line 230, in __getattr__
    raise AttributeError(*ke.args)
AttributeError: no enumerated value with that name: 'yellow'
>>>
>>> print "Value of 'red' and 'white':", (colors['red'], colors['white'])
Value of 'red' and 'white': (0, 3)
>>> print "Value of 'yellow':", colors['yellow']
Value of 'yellow':
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File ".../tables/misc/enum.py", line 189, in __getitem__
    raise KeyError("no enumerated value with that name: %r" % (name,))
KeyError: "no enumerated value with that name: 'yellow'"
```

See how accessing a value that is not in the enumeration raises the appropriate exception. We can also do the opposite action and get the name that matches a concrete value by using the `__call__()` method of Enum:

```
>>> print "Name of value %s:" % colors.red, colors(colors.red)
Name of value 0: red
>>> print "Name of value 1234:", colors(1234)
Name of value 1234:
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File ".../tables/misc/enum.py", line 320, in __call__
    raise ValueError(
ValueError: no enumerated value with that concrete value: 1234
```

You can see what we made as using the enumerated type to *convert* a concrete value into a name in the enumeration. Of course, values out of the enumeration can not be converted.

### 3.7.1 Enumerated columns

Columns of an enumerated type can be declared by using the `EnumCol` (see *The Col class and its descendants*) class. To see how this works, let us open a new PyTables file and create a table to collect the simulated results of a probabilistic experiment. In it, we have a bag full of colored balls; we take a ball out and annotate the time of extraction and the color of the ball:

```
>>> h5f = tables.openFile('enum.h5', 'w')
>>> class BallExt(tables.IsDescription):
...     ballTime = tables.Time32Col()
...     ballColor = tables.EnumCol(colors, 'black', base='uint8')
>>> tbl = h5f.createTable('/', 'extractions', BallExt, title="Random ball extractions")
>>>
```

We declared the `ballColor` column to be of the enumerated type `colors`, with a default value of `black`. We also stated that we are going to store concrete values as unsigned 8-bit integer values<sup>4</sup>.

Let us use some random values to fill the table:

```
>>> import time
>>> import random
>>> now = time.time()
>>> row = tbl.row
>>> for i in range(10):
```

---

<sup>4</sup> In fact, only integer values are supported right now, but this may change in the future.

```
...     row['ballTime'] = now + i
...     row['ballColor'] = colors[random.choice(colorList)] # notice this
...     row.append()
>>>
```

Notice how we used the `__getitem__()` call of `colors` to get the concrete value to store in `ballColor`. You should know that this way of appending values to a table does automatically check for the validity on enumerated values. For instance:

```
>>> row['ballTime'] = now + 42
>>> row['ballColor'] = 1234
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "tableExtension.pyx", line 1086, in tableExtension.Row.__setitem__
  File ".../tables/misc/enum.py", line 320, in __call__
    "no enumerated value with that concrete value: %r" % (value,)
ValueError: no enumerated value with that concrete value: 1234
```

But take care that this check is *only* performed here and not in other methods such as `tbl.append()` or `tbl.modifyRows()`. Now, after flushing the table we can see the results of the insertions:

```
>>> tbl.flush()
>>> for r in tbl:
...     ballTime = r['ballTime']
...     ballColor = colors(r['ballColor']) # notice this
...     print "Ball extracted on %d is of color %s." % (ballTime, ballColor)
Ball extracted on 1173785568 is of color green.
Ball extracted on 1173785569 is of color black.
Ball extracted on 1173785570 is of color white.
Ball extracted on 1173785571 is of color black.
Ball extracted on 1173785572 is of color black.
Ball extracted on 1173785573 is of color red.
Ball extracted on 1173785574 is of color green.
Ball extracted on 1173785575 is of color red.
Ball extracted on 1173785576 is of color white.
Ball extracted on 1173785577 is of color white.
```

As a last note, you may be wondering how to have access to the enumeration associated with `ballColor` once the file is closed and reopened. You can call `tbl.getEnum('ballColor')` (see `Table.getEnum()`) to get the enumeration back.

### 3.7.2 Enumerated arrays

`EArray` and `VArray` leaves can also be declared to store enumerated values by means of the `EnumAtom` (see *The Atom class and its descendants*) class, which works very much like `EnumCol` for tables. Also, `Array` leaves can be used to open native HDF enumerated arrays.

Let us create a sample `EArray` containing ranges of working days as bidimensional values:

```
>>> workingDays = {'Mon': 1, 'Tue': 2, 'Wed': 3, 'Thu': 4, 'Fri': 5}
>>> dayRange = tables.EnumAtom(workingDays, 'Mon', base='uint16')
>>> earr = h5f.createEArray('/', 'days', dayRange, (0, 2), title="Working day ranges")
>>> earr.flavor = 'python'
```

Nothing surprising, except for a pair of details. In the first place, we use a *dictionary* instead of a list to explicitly set concrete values in the enumeration. In the second place, there is no explicit `Enum` instance created! Instead, the dictionary is passed as the first argument to the constructor of `EnumAtom`. If the constructor gets a list or a dictionary instead of an enumeration, it automatically builds the enumeration from it.

Now let us feed some data to the array:

```
>>> wdays = earr.getEnum()
>>> earr.append([(wdays.Mon, wdays.Fri), (wdays.Wed, wdays.Fri)])
>>> earr.append([(wdays.Mon, 1234)])
```

Please note that, since we had no explicit Enum instance, we were forced to use `getEnum()` (see [EArray methods](#)) to get it from the array (we could also have used `dayRange.enum`). Also note that we were able to append an invalid value (1234). Array methods do not check the validity of enumerated values.

Finally, we will print the contents of the array:

```
>>> for (d1, d2) in earr:
...     print "From %s to %s (%d days)." % (wdays(d1), wdays(d2), d2-d1+1)
From Mon to Fri (5 days).
From Wed to Fri (3 days).
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
  File "../tables/misc/enum.py", line 320, in __call__
    "no enumerated value with that concrete value: %r" % (value,)
ValueError: no enumerated value with that concrete value: 1234
```

That was an example of operating on concrete values. It also showed how the value-to-name conversion failed because of the value not belonging to the enumeration.

Now we will close the file, and this little tutorial on enumerated types is done:

```
>>> h5f.close()
```

## 3.8 Dealing with nested structures in tables

PyTables supports the handling of nested structures (or nested datatypes, as you prefer) in table objects, allowing you to define arbitrarily nested columns.

An example will clarify what this means. Let's suppose that you want to group your data in pieces of information that are more related than others pieces in your table. So you may want to tie them up together in order to have your table better structured but also be able to retrieve and deal with these groups more easily.

You can create such a nested substructures by just nesting subclasses of `IsDescription`. Let's see one example (okay, it's a bit silly, but will serve for demonstration purposes):

```
from tables import *

class Info(IsDescription):
    """A sub-structure of Test"""
    _v_pos = 2 # The position in the whole structure
    name = StringCol(10)
    value = Float64Col(pos=0)

colors = Enum(['red', 'green', 'blue'])

class NestedDescr(IsDescription):
    """A description that has several nested columns"""
    color = EnumCol(colors, 'red', base='uint32')
    info1 = Info()

    class info2(IsDescription):
        _v_pos = 1
```

```

name = StringCol(10)
value = Float64Col(pos=0)

class info3(IsDescription):
    x = Float64Col(dflt=1)
    y = UInt8Col(dflt=1)

```

The root class is `NestedDescr` and both `info1` and `info2` are *substructures* of it. Note how `info1` is actually an instance of the class `Info` that was defined prior to `NestedDescr`. Also, there is a third substructure, namely `info3` that hangs from the substructure `info2`. You can also define positions of substructures in the containing object by declaring the special class attribute `_v_pos`.

### 3.8.1 Nested table creation

Now that we have defined our nested structure, let's create a *nested* table, that is a table with columns that contain other subcolumns:

```

>>> fileh = openFile("nested-tut.h5", "w")
>>> table = fileh.createTable(fileh.root, 'table', NestedDescr)

```

Done! Now, we have to feed the table with some values. The problem is how we are going to reference to the nested fields. That's easy, just use a `'/'` character to separate names in different nested levels. Look at this:

```

>>> row = table.row
>>> for i in range(10):
...     row['color'] = colors[['red', 'green', 'blue'][i%3]]
...     row['info1/name'] = "name1-%s" % i
...     row['info2/name'] = "name2-%s" % i
...     row['info2/info3/y'] = i
...     # All the rest will be filled with defaults
...     row.append()
>>> table.flush()
>>> table.nrows
10

```

You see? In order to fill the fields located in the substructures, we just need to specify its full path in the table hierarchy.

### 3.8.2 Reading nested tables

Now, what happens if we want to read the table? What kind of data container will we get? Well, it's worth trying it:

```

>>> nra = table[:4]
>>> nra
array([((1.0, 0), 'name2-0', 0.0), ('name1-0', 0.0), 0L),
      ((1.0, 4), 'name2-4', 0.0), ('name1-4', 0.0), 1L),
      ((1.0, 8), 'name2-8', 0.0), ('name1-8', 0.0), 2L)],
      dtype=[('info2', [(('info3', [(('x', '>f8'), ('y', '\u1')]),
      ('name', '\S10'), ('value', '>f8'))]),
      ('info1', [(('name', '\S10'), ('value', '>f8'))]),
      ('color', '>u4')])

```

What we got is a NumPy array with a *compound, nested datatype* (its dtype is a list of name-datatype tuples). We read one row for each four in the table, giving a result of three rows.

**Note:** When using the numarray flavor (deprecated), you will get an instance of the NestedRecArray class that lives in the tables.nra package. NestedRecArray is actually a subclass of the RecArray object of the numarray.records module. You can get more info about NestedRecArray object in *Using nested record arrays (deprecated)*.

---

You can make use of the above object in many different ways. For example, you can use it to append new data to the existing table object:

```
>>> table.append(nra)
>>> table.nrows
13
```

Or, to create new tables:

```
>>> table2 = fileh.createTable(fileh.root, 'table2', nra)
>>> table2[:]
array([((1.0, 0), 'name2-0', 0.0), ('name1-0', 0.0), 0L),
      ((1.0, 4), 'name2-4', 0.0), ('name1-4', 0.0), 1L),
      ((1.0, 8), 'name2-8', 0.0), ('name1-8', 0.0), 2L)],
      dtype=[('info2', [('info3', [('x', '<f8'), ('y', '\\|u1')]),
                          ('name', '\\|S10'), ('value', '<f8')]),
              ('info1', [('name', '\\|S10'), ('value', '<f8')]),
              ('color', '<u4')])
```

Finally, we can select nested values that fulfill some condition:

```
>>> names = [ x['info2/name'] for x in table if x['color'] == colors.red ]
>>> names
['name2-0', 'name2-3', 'name2-6', 'name2-9', 'name2-0']
```

Note that the row accessor does not provide the natural naming feature, so you have to completely specify the path of your desired columns in order to reach them.

### 3.8.3 Using Cols accessor

We can use the cols attribute object (see *The Cols class*) of the table so as to quickly access the info located in the interesting substructures:

```
>>> table.cols.info2[1:5]
array([((1.0, 1), 'name2-1', 0.0), ((1.0, 2), 'name2-2', 0.0),
      ((1.0, 3), 'name2-3', 0.0), ((1.0, 4), 'name2-4', 0.0)],
      dtype=[('info3', [('x', '<f8'), ('y', '\\|u1')]), ('name', '\\|S10'),
              ('value', '<f8')])
```

Here, we have made use of the cols accessor to access to the *info2* substructure and an slice operation to get access to the subset of data we were interested in; you probably have recognized the natural naming approach here. We can continue and ask for data in *info3* substructure:

```
>>> table.cols.info2.info3[1:5]
array([(1.0, 1), (1.0, 2), (1.0, 3), (1.0, 4)],
      dtype=[('x', '<f8'), ('y', '\\|u1')])
```

You can also use the `_f_col` method to get a handler for a column:

```
>>> table.cols._f_col('info2')
/table.cols.info2 (Cols), 3 columns
  info3 (Cols(), Description)
    name (Column(), \\|S10)
    value (Column(), float64)
```

Here, you've got another Cols object handler because *info2* was a nested column. If you select a non-nested column, you will get a regular Column instance:

```
>>> table.cols._f_col('info2/info3/y')
/table.cols.info2.info3.y (Column(), uint8, idx=None)
```

To sum up, the cols accessor is a very handy and powerful way to access data in your nested tables. Don't be afraid of using it, specially when doing interactive work.

### 3.8.4 Accessing meta-information of nested tables

Tables have an attribute called *description* which points to an instance of the Description class (see *The Description class*) and is useful to discover different meta-information about table data.

Let's see how it looks like:

```
>>> table.description
{
  "info2": {
    "info3": {
      "x": Float64Col(shape=(), dflt=1.0, pos=0),
      "y": UInt8Col(shape=(), dflt=1, pos=1)},
    "name": StringCol(itemsizes=10, shape=(), dflt='', pos=1),
    "value": Float64Col(shape=(), dflt=0.0, pos=2)},
  "info1": {
    "name": StringCol(itemsizes=10, shape=(), dflt='', pos=0),
    "value": Float64Col(shape=(), dflt=0.0, pos=1)},
  "color": EnumCol(enum=Enum({'blue': 2, 'green': 1, 'red': 0}), dflt='red',
                    base=UInt32Atom(shape=(), dflt=0), shape=(), pos=2)}
```

As you can see, it provides very useful information on both the formats and the structure of the columns in your table.

This object also provides a natural naming approach to access to subcolumns metadata:

```
>>> table.description.info1
{"name": StringCol(itemsizes=10, shape=(), dflt='', pos=0),
 "value": Float64Col(shape=(), dflt=0.0, pos=1)}
>>> table.description.info2.info3
{"x": Float64Col(shape=(), dflt=1.0, pos=0),
 "y": UInt8Col(shape=(), dflt=1, pos=1)}
```

There are other variables that can be interesting for you:

```
>>> table.description._v_nestedNames
[('info2', [(('info3', [(('x', 'y'), ('name', 'value'))],
  ('info1', [(('name', 'value'), ('color'))])])])])
>>> table.description.info1._v_nestedNames
[('name', 'value')]
```

*\_v\_nestedNames* provides the names of the columns as well as its structure. You can see that there are the same attributes for the different levels of the Description object, because the levels are *also* Description objects themselves.

There is a special attribute, called *\_v\_nestedDescr*, that can be useful to create nested structured arrays that imitate the structure of the table (or a subtable thereof):

```
>>> import numpy
>>> table.description._v_nestedDescr
[('info2', [(('info3', [(('x', '()f8'), ('y', '()u1'))], ('name', '()S10'),
  ('value', '()f8'))], ('info1', [(('name', '()S10'), ('value', '()f8'))])])])]
```

```
('color', '()u4')]  
>>> numpy.rec.array(None, shape=0,  
                    dtype=table.description._v_nestedDescr)  
recarray([],  
         dtype=[('info2', [(('info3', [(('x', '>f8'), ('y', '|u1'))],  
    ('name', '|S10'), ('value', '>f8'))],  
    ('info1', [(('name', '|S10'), ('value', '>f8'))],  
    ('color', '>u4'))])  
>>> numpy.rec.array(None, shape=0,  
                    dtype=table.description.info2._v_nestedDescr)  
recarray([],  
         dtype=[('info3', [(('x', '>f8'), ('y', '|u1'))], ('name', '|S10'),  
    ('value', '>f8'))])  
>>> from tables import nra  
>>> nra.array(None, descr=table.description._v_nestedDescr)  
array(  
    [],  
    descr=[('info2', [(('info3', [(('x', '()f8'), ('y', '()u1'))],  
    ('name', '()S10'), ('value', '()f8'))], ('info1', [(('name', '()S10'),  
    ('value', '()f8'))], ('color', '()u4'))],  
    shape=0)
```

You can see we have created two equivalent arrays: one with NumPy (the first) and one with the nra package (the last). The later implements nested record arrays for numarray (see `NestedRecArray`).

Finally, there is a special iterator of the `Description` class, called `_f_walk` that is able to return you the different columns of the table:

```
>>> for coldescr in table.description._f_walk():  
...     print "column-->", coldescr  
column--> Description([(('info3', [(('x', '()f8'), ('y', '()u1'))],  
    ('name', '()S10'), ('value', '()f8'))],  
    ('info1', [(('name', '()S10'), ('value', '()f8'))],  
    ('color', '()u4'))])  
column--> EnumCol(enum=Enum({'blue': 2, 'green': 1, 'red': 0}), dflt='red',  
    base=UInt32Atom(shape=(), dflt=0), shape=(), pos=2)  
column--> Description([(('info3', [(('x', '()f8'), ('y', '()u1'))], ('name', '()S10'),  
    ('value', '()f8'))])  
column--> StringCol(itemsz=10, shape=(), dflt='', pos=1)  
column--> Float64Col(shape=(), dflt=0.0, pos=2)  
column--> Description([(('name', '()S10'), ('value', '()f8'))])  
column--> StringCol(itemsz=10, shape=(), dflt='', pos=0)  
column--> Float64Col(shape=(), dflt=0.0, pos=1)  
column--> Description([(('x', '()f8'), ('y', '()u1'))])  
column--> Float64Col(shape=(), dflt=1.0, pos=0)  
column--> UInt8Col(shape=(), dflt=1, pos=1)
```

See the [The Description class](#) for the complete listing of attributes and methods of `Description`.

Well, this is the end of this tutorial. As always, do not forget to close your files:

```
>>> fileh.close()
```

Finally, you may want to have a look at your resulting data file.

```
$ ptdump -d nested-tut.h5  
/ (RootGroup) ''  
/table (Table(13,)) ''  
  Data dump:  
[0] (((1.0, 0), 'name2-0', 0.0), ('name1-0', 0.0), 0L)
```

```

[1] (((1.0, 1), 'name2-1', 0.0), ('name1-1', 0.0), 1L)
[2] (((1.0, 2), 'name2-2', 0.0), ('name1-2', 0.0), 2L)
[3] (((1.0, 3), 'name2-3', 0.0), ('name1-3', 0.0), 0L)
[4] (((1.0, 4), 'name2-4', 0.0), ('name1-4', 0.0), 1L)
[5] (((1.0, 5), 'name2-5', 0.0), ('name1-5', 0.0), 2L)
[6] (((1.0, 6), 'name2-6', 0.0), ('name1-6', 0.0), 0L)
[7] (((1.0, 7), 'name2-7', 0.0), ('name1-7', 0.0), 1L)
[8] (((1.0, 8), 'name2-8', 0.0), ('name1-8', 0.0), 2L)
[9] (((1.0, 9), 'name2-9', 0.0), ('name1-9', 0.0), 0L)
[10] (((1.0, 0), 'name2-0', 0.0), ('name1-0', 0.0), 0L)
[11] (((1.0, 4), 'name2-4', 0.0), ('name1-4', 0.0), 1L)
[12] (((1.0, 8), 'name2-8', 0.0), ('name1-8', 0.0), 2L)
/table2 (Table(3,)) ''
Data dump:
[0] (((1.0, 0), 'name2-0', 0.0), ('name1-0', 0.0), 0L)
[1] (((1.0, 4), 'name2-4', 0.0), ('name1-4', 0.0), 1L)
[2] (((1.0, 8), 'name2-8', 0.0), ('name1-8', 0.0), 2L)

```

Most of the code in this section is also available in `examples/nested-tut.py`.

All in all, PyTables provides a quite comprehensive toolset to cope with nested structures and address your classification needs. However, caveat emptor, be sure to not nest your data too deeply or you will get inevitably messed interpreting too intertwined lists, tuples and description objects.

## 3.9 Other examples in PyTables distribution

Feel free to examine the rest of examples in directory `examples/`, and try to understand them. We have written several practical sample scripts to give you an idea of the PyTables capabilities, its way of dealing with HDF5 objects, and how it can be used in the real world.



# LIBRARY REFERENCE

PyTables implements several classes to represent the different nodes in the object tree. They are named File, Group, Leaf, Table, Array, CArray, EArray, VArray and UnImplemented. Another one allows the user to complement the information on these different objects; its name is AttributeSet. Finally, another important class called IsDescription allows to build a Table record description by declaring a subclass of it. Many other classes are defined in PyTables, but they can be regarded as helpers whose goal is mainly to declare the *data type properties* of the different first class objects and will be described at the end of this chapter as well.

An important function, called `openFile` is responsible to create, open or append to files. In addition, a few utility functions are defined to guess if the user supplied file is a *PyTables* or *HDF5* file. These are called `isPyTablesFile()` and `isHDF5File()`, respectively. There exists also a function called `whichLibVersion()` that informs about the versions of the underlying C libraries (for example, *HDF5* or *Zlib*) and another called `print_versions()` that prints all the versions of the software that PyTables relies on. Finally, `test()` lets you run the complete test suite from a Python console interactively.

Let's start discussing the first-level variables and functions available to the user, then the different classes defined in PyTables.

## 4.1 tables variables and functions

### 4.1.1 Global variables

### 4.1.2 Global functions

## 4.2 The File Class

### 4.2.1 File properties

### 4.2.2 File methods - file handling

### 4.2.3 File methods - hierarchy manipulation

### 4.2.4 File methods - tree traversal

### 4.2.5 File methods - Undo/Redo support

### 4.2.6 File methods - attribute handling

## 4.3 The Node class

### 4.3.1 Node instance variables - location dependent

### 4.3.2 Node instance variables - location independent

### 4.3.3 Node instance variables - attribute shorthands

### 4.3.4 Node methods - hierarchy manipulation

### 4.3.5 Node methods - attribute handling

## 4.4 The Group class

### 4.4.1 Group properties

### 4.4.2 Group methods

---

**Important:** *Caveat:* The following methods are documented for completeness, and they can be used without any problem. However, you should use the high-level counterpart methods in the File class (see *The File Class*, because they are most used in documentation and examples, and are a bit more powerful than those exposed here.

---

The following methods are provided in addition to those in Node (see *The Node class*):

### 4.4.3 Group special methods

Following are described the methods that automatically trigger actions when a Group instance is accessed in a special way.

This class defines the `__setattr__()`, `__getattr__()` and `__delattr__()` methods, and they set, get and delete *ordinary Python attributes* as normally intended. In addition to that, `__getattr__()` allows getting *child nodes* by their name for the sake of easy interaction on the command line, as long as there is no Python attribute with the same name. Groups also allow the interactive completion (when using readline) of the names of child nodes. For instance:

```
# get a Python attribute
nchild = group._v_nchildren

# Add a Table child called 'table' under 'group'.
h5file.createTable(group, 'table', myDescription)
table = group.table           # get the table child instance
group.table = 'foo'          # set a Python attribute

# (PyTables warns you here about using the name of a child node.)
foo = group.table             # get a Python attribute
del group.table               # delete a Python attribute
table = group.table           # get the table child instance again
```

## 4.5 The Leaf class

### 4.5.1 Leaf properties

`Leaf.size_in_memory`

The size of this leaf's data in bytes when it is fully loaded into memory.

### 4.5.2 Leaf instance variables - aliases

The following are just easier-to-write aliases to their Node (see *The Node class*) counterparts (indicated between parentheses):

### 4.5.3 Leaf methods

## 4.6 The Table class

### 4.6.1 Table properties

### 4.6.2 Table methods - reading

### 4.6.3 Table methods - writing

### 4.6.4 Table methods - querying

### 4.6.5 Table methods - other

### 4.6.6 The Description class

#### Description methods

### 4.6.7 The Row class

#### Row methods

#### Row special methods

### 4.6.8 The Cols class

#### Cols properties

#### Cols methods

### 4.6.9 The Column class

#### Column instance variables

#### Column methods

#### Column special methods

## 4.7 The Array class

### 4.7.1 Array instance variables

#### `Array.atom`

An Atom (see *The Atom class and its descendants*) instance representing the *type* and *shape* of the atomic objects to be saved.

#### `Array.nrow`

On iterators, this is the index of the current row.

### 4.7.2 Array methods

### 4.7.3 Array special methods

The following methods automatically trigger actions when an `Array` instance is accessed in a special way (e.g. `array[2:3,...,::2]` will be equivalent to a call to `array.__getitem__((slice(2, 3, None), Ellipsis, slice(None, None, 2)))`).

## 4.8 The CArray class

## 4.9 The EArray class

### 4.9.1 EArray methods

## 4.10 The VArray class

### 4.10.1 VArray properties

### 4.10.2 VArray methods

### 4.10.3 VArray special methods

The following methods automatically trigger actions when a `VArray` instance is accessed in a special way (e.g., `varray[2:5]` will be equivalent to a call to `varray.__getitem__(slice(2, 5, None))`).

## 4.11 The Link class

### 4.11.1 Link instance variables

### 4.11.2 Link methods

The following methods are useful for copying, moving, renaming and removing links.

## 4.12 The SoftLink class

### 4.12.1 SoftLink special methods

The following methods are specific for dereferencing and representing soft links.

## 4.13 The ExternalLink class

### 4.13.1 ExternalLink methods

### 4.13.2 ExternalLink special methods

The following methods are specific for dereferencing and representing external links.

## 4.14 The Unimplemented class

## 4.15 The Unknown class

## 4.16 The AttributeSet class

### 4.16.1 AttributeSet properties

### 4.16.2 AttributeSet methods

## 4.17 Declarative classes

In this section a series of classes that are meant to *declare* datatypes that are required for creating primary PyTables datasets are described.

### 4.17.1 The Atom class and its descendants

#### Atom properties

#### Atom methods

#### Atom factory methods

#### Atom Sub-classes

#### Pseudo atoms

Now, there come three special classes, ObjectAtom, VLStringAtom and VLUnicodeAtom, that actually do not descend from Atom, but which goal is so similar that they should be described here. Pseudo-atoms can only be used with VLArray datasets (see [The VLArray class](#)), and they do not support multidimensional values, nor multiple values per row.

They can be recognised because they also have kind, type and shape attributes, but no size, itemsize or dtype ones. Instead, they have a base atom which defines the elements used for storage.

See `examples/vlarray1.py` and `examples/vlarray2.py` for further examples on VLArray datasets, including object serialization and string management.

**ObjectAtom**

**VLStringAtom**

**VLUnicodeAtom**

## 4.17.2 The Col class and its descendants

### Col instance variables

In addition to the variables that they inherit from the Atom class, Col instances have the following attributes.

`Col._v_pos`

The *relative* position of this column with regard to its column siblings.

### Col factory methods

### Col sub-classes

## 4.17.3 The IsDescription class

## 4.17.4 Description helper functions

# 4.18 Helper classes

This section describes some classes that do not fit in any other section and that mainly serve for ancillary purposes.

## 4.18.1 The Filters class

### Filters methods

## 4.18.2 The Index class

### Index instance variables

`tables.index.Index.nelements`

The number of currently indexed rows for this column.

Index methods

Index special methods

### 4.18.3 The IndexArray class

### 4.18.4 The Enum class

Enum special methods

## 4.19 The Expr class - a general-purpose expression evaluator

### 4.19.1 Expr methods

### 4.19.2 Expr special methods

## 4.20 Exceptions module

In the `exceptions` module exceptions and warnings that are specific to PyTables are declared.

---

# OPTIMIZATION TIPS

... durch planmässiges Tattonieren.

[... through systematic, palpable experimentation.]

—Johann Karl Friedrich Gauss [asked how he came upon his theorems]

On this chapter, you will get deeper knowledge of PyTables internals. PyTables has many tunable features so that you can improve the performance of your application. If you are planning to deal with really large data, you should read carefully this section in order to learn how to get an important efficiency boost for your code. But if your datasets are small (say, up to 10 MB) or your number of nodes is contained (up to 1000), you should not worry about that as the default parameters in PyTables are already tuned for those sizes (although you may want to adjust them further anyway). At any rate, reading this chapter will help you in your life with PyTables.

## 5.1 Understanding chunking

The underlying HDF5 library that is used by PyTables allows for certain datasets (the so-called *chunked* datasets) to take the data in bunches of a certain length, named *chunks*, and write them on disk as a whole, i.e. the HDF5 library treats chunks as atomic objects and disk I/O is always made in terms of complete chunks. This allows data filters to be defined by the application to perform tasks such as compression, encryption, check-summing, etc. on entire chunks.

HDF5 keeps a B-tree in memory that is used to map chunk structures on disk. The more chunks that are allocated for a dataset the larger the B-tree. Large B-trees take memory and cause file storage overhead as well as more disk I/O and higher contention for the metadata cache. Consequently, it's important to balance between memory and I/O overhead (small B-trees) and time to access data (big B-trees).

In the next couple of sections, you will discover how to inform PyTables about the expected size of your datasets for allowing a sensible computation of the chunk sizes. Also, you will be presented some experiments so that you can get a feeling on the consequences of manually specifying the chunk size. Although doing this latter is only reserved to experienced people, these benchmarks may allow you to understand more deeply the chunk size implications and let you quickly start with the fine-tuning of this important parameter.

### 5.1.1 Informing PyTables about expected number of rows in tables or arrays

PyTables can determine a sensible chunk size to your dataset size if you helps it by providing an estimation of the final number of rows for an extensible leaf<sup>1</sup>. You should provide this information at leaf creation time by passing this value to the `expectedrows` argument of the `File.createTable()` method or `File.createEArray()` method (see [The EArray class](#)). For `VArray` leaves, you must pass the expected size in MBytes by using the argument `expectedsizeinMB` of `File.createVArray()` (see [The VArray class](#)) instead.

<sup>1</sup> CArray nodes, though not extensible, are chunked and have their optimum chunk size automatically computed at creation time, since their final shape is known.

When your leaf size is bigger than 10 MB (take this figure only as a reference, not strictly), by providing this guess you will be optimizing the access to your data. When the table or array size is larger than, say 100MB, you are *strongly* suggested to provide such a guess; failing to do that may cause your application to do very slow I/O operations and to demand *huge* amounts of memory. You have been warned!

### 5.1.2 Fine-tuning the chunksize

**Warning:** This section is mostly meant for experts. If you are a beginner, you must know that setting manually the chunksize is a potentially dangerous action.

Most of the time, informing PyTables about the extent of your dataset is enough. However, for more sophisticated applications, when one has special requirements for doing the I/O or when dealing with really large datasets, you should really understand the implications of the chunk size in order to be able to find the best value for your own application.

You can specify the chunksize for every chunked dataset in PyTables by passing the chunkshape argument to the corresponding constructors. It is important to point out that chunkshape is not exactly the same thing than a chunksize; in fact, the chunksize of a dataset can be computed multiplying all the dimensions of the chunkshape among them and multiplying the outcome by the size of the atom.

We are going to describe a series of experiments where an EArray of 15 GB is written with different chunksizes, and then it is accessed in both sequential (i.e. first element 0, then element 1 and so on and so forth until the data is exhausted) and random mode (i.e. single elements are read randomly all through the dataset). These benchmarks have been carried out with PyTables 2.1 on a machine with an Intel Core2 processor @ 3 GHz and a RAID-0 made of two SATA disks spinning at 7200 RPM, and using GNU/Linux with an XFS filesystem. The script used for the benchmarks is available in `bench/optimal-chunksize.py`.

In figures [Figure 1](#), [Figure 2](#), [Figure 3](#) and [Figure 4](#), you can see how the chunksize affects different aspects, like creation time, file sizes, sequential read time and random read time. So, if you properly inform PyTables about the extent of your datasets, you will get an automatic chunksize value (256 KB in this case) that is pretty optimal for most of uses. However, if what you want is, for example, optimize the creation time when using the Zlib compressor, you may want to reduce the chunksize to 32 KB (see [Figure 1](#)). Or, if your goal is to optimize the sequential access time for an dataset compressed with Blosc, you may want to increase the chunksize to 512 KB (see [Figure 3](#)).

You will notice that, by manually specifying the chunksize of a leave you will not normally get a drastic increase in performance, but at least, you have the opportunity to fine-tune such an important parameter for improve performance.

Finally, it is worth noting that adjusting the chunksize can be specially important if you want to access your dataset by blocks of certain dimensions. In this case, it is normally a good idea to set your chunkshape to be the same than these dimensions; you only have to be careful to not end with a too small or too large chunksize. As always, experimenting prior to pass your application into production is your best ally.

## 5.2 Accelerating your searches

---

**Note:** Many of the explanations and plots in this section and the forthcoming ones still need to be updated to include Blosc (see [\[BLOSC\]](#)), the new and powerful compressor added in PyTables 2.2 series. You should expect it to be the fastest compressor among all the described here, and its use is strongly recommended whenever you need extreme speed and not a very high compression ratio.

---

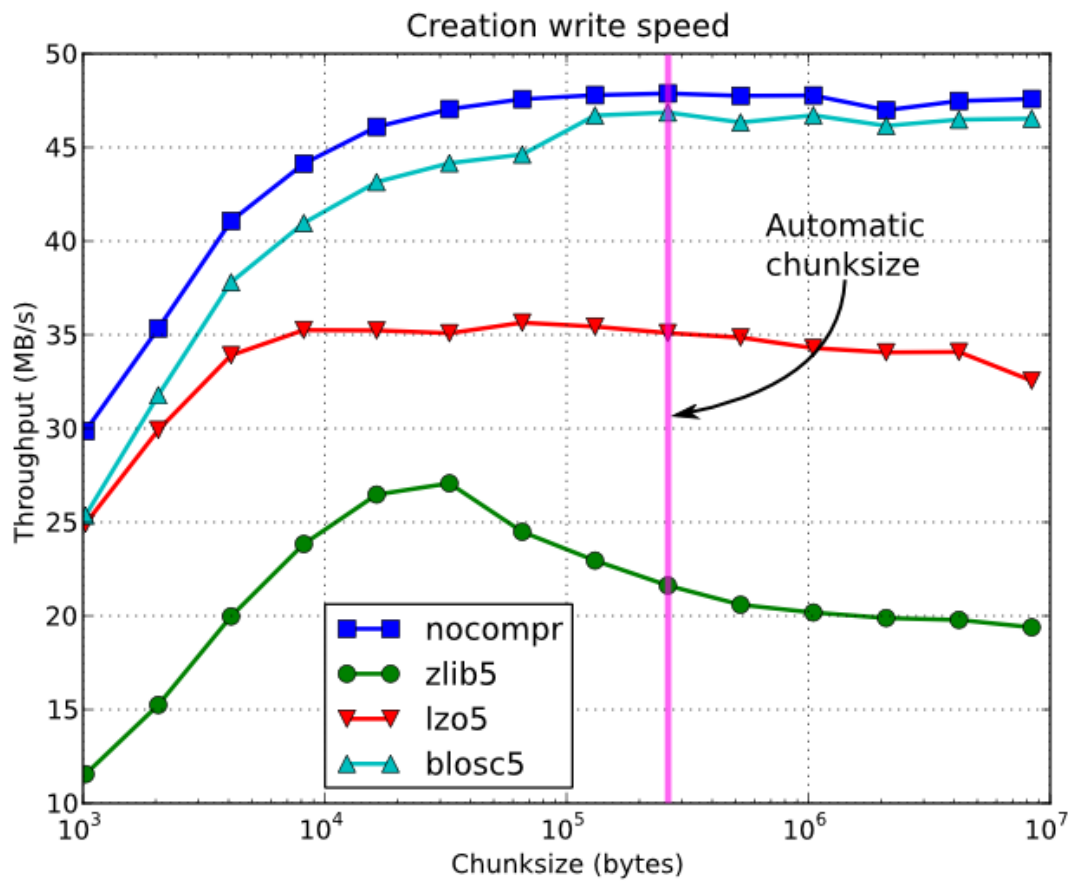


Figure 5.1: Figure 1. Creation time per element for a 15 GB EArray and different chunksizes.

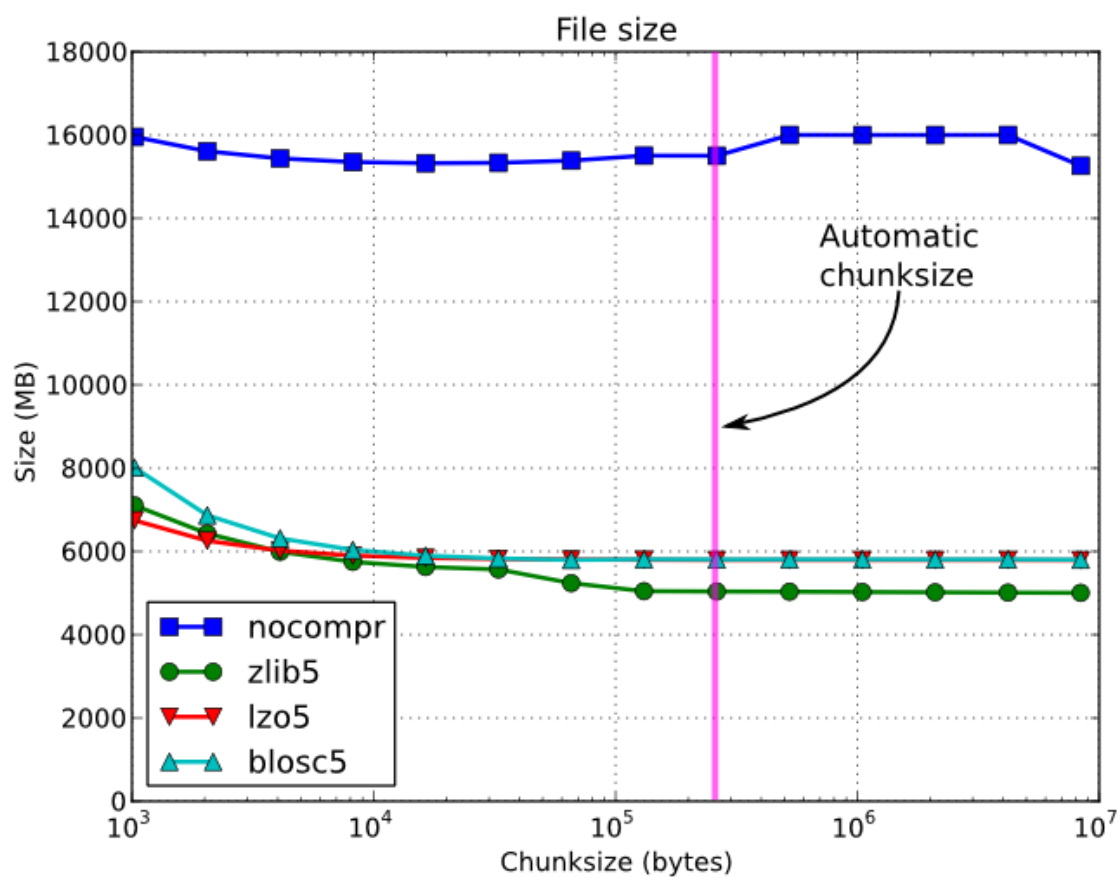


Figure 5.2: Figure 2. File sizes for a 15 GB EArray and different chunksizes.

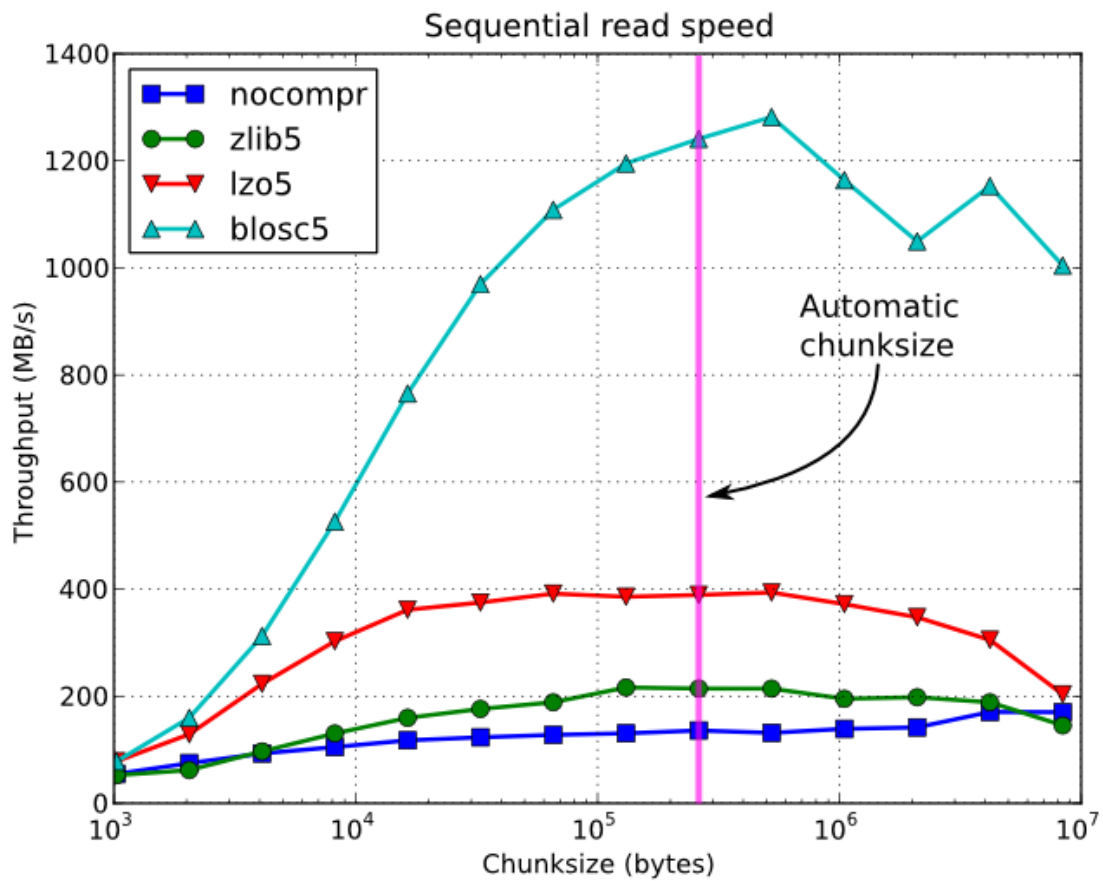


Figure 5.3: Figure 3. Sequential access time per element for a 15 GB EArray and different chunksizes.

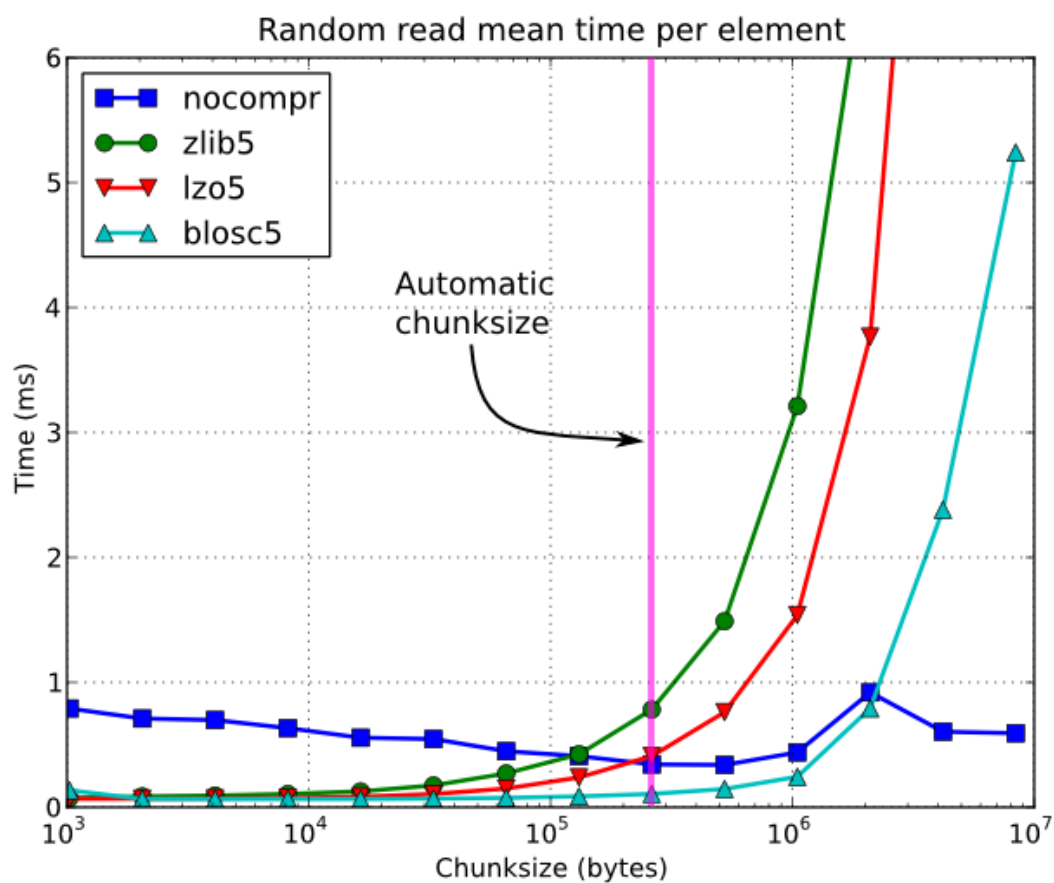


Figure 5.4: **Figure 4.** Random access time per element for a 15 GB EArray and different chunksizes.

Searching in tables is one of the most common and time consuming operations that a typical user faces in the process of mining through his data. Being able to perform queries as fast as possible will allow more opportunities for finding the desired information quicker and also allows to deal with larger datasets.

PyTables offers many sort of techniques so as to speed-up the search process as much as possible and, in order to give you hints to use them based, a series of benchmarks have been designed and carried out. All the results presented in this section have been obtained with synthetic, random data and using PyTables 2.1. Also, the tests have been conducted on a machine with an Intel Core2 (64-bit) @ 3 GHz processor with RAID-0 disk storage (made of four spinning disks @ 7200 RPM), using GNU/Linux with an XFS filesystem. The script used for the benchmarks is available in `bench/indexed_search.py`. As your data, queries and platform may be totally different for your case, take this just as a guide because your mileage may vary (and will vary).

In order to be able to play with tables with a number of rows as large as possible, the record size has been chosen to be rather small (24 bytes). Here it is its definition:

```
class Record(tables.IsDescription):
    col1 = tables.Int32Col()
    col2 = tables.Int32Col()
    col3 = tables.Float64Col()
    col4 = tables.Float64Col()
```

In the next sections, we will be optimizing the times for a relatively complex query like this:

```
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)
)]
```

(for future reference, we will call this sort of queries *regular* queries). So, if you want to see how to greatly improve the time taken to run queries like this, keep reading.

## 5.2.1 In-kernel searches

PyTables provides a way to accelerate data selections inside of a single table, through the use of the *Table methods - querying* iterator and related query methods. This mode of selecting data is called *in-kernel*. Let's see an example of an *in-kernel* query based on the *regular* one mentioned above:

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

This simple change of mode selection can improve search times quite a lot and actually make PyTables very competitive when compared against typical relational databases as you can see in [Figure 5](#) and [Figure 6](#).

By looking at [Figure 5](#) you can see how in the case that table data fits easily in memory, in-kernel searches on uncompressed tables are generally much faster (10x) than standard queries as well as PostgreSQL (5x). Regarding compression, we can see how Zlib compressor actually slows down the performance of in-kernel queries by a factor 3.5x; however, it remains faster than PostgreSQL (40%). On his hand, LZO compressor only decreases the performance by a 75% with respect to uncompressed in-kernel queries and is still a lot faster than PostgreSQL (3x). Finally, one can observe that, for low selectivity queries (large number of hits), PostgreSQL performance degrades quite steadily, while in PyTables this slow down rate is significantly smaller. The reason of this behaviour is not entirely clear to the authors, but the fact is clearly reproducible in our benchmarks.

But, why in-kernel queries are so fast when compared with regular ones?. The answer is that in regular selection mode the data for all the rows in table has to be brought into Python space so as to evaluate the condition and decide if the corresponding field should be added to the result list. On the contrary, in the in-kernel mode, the condition is

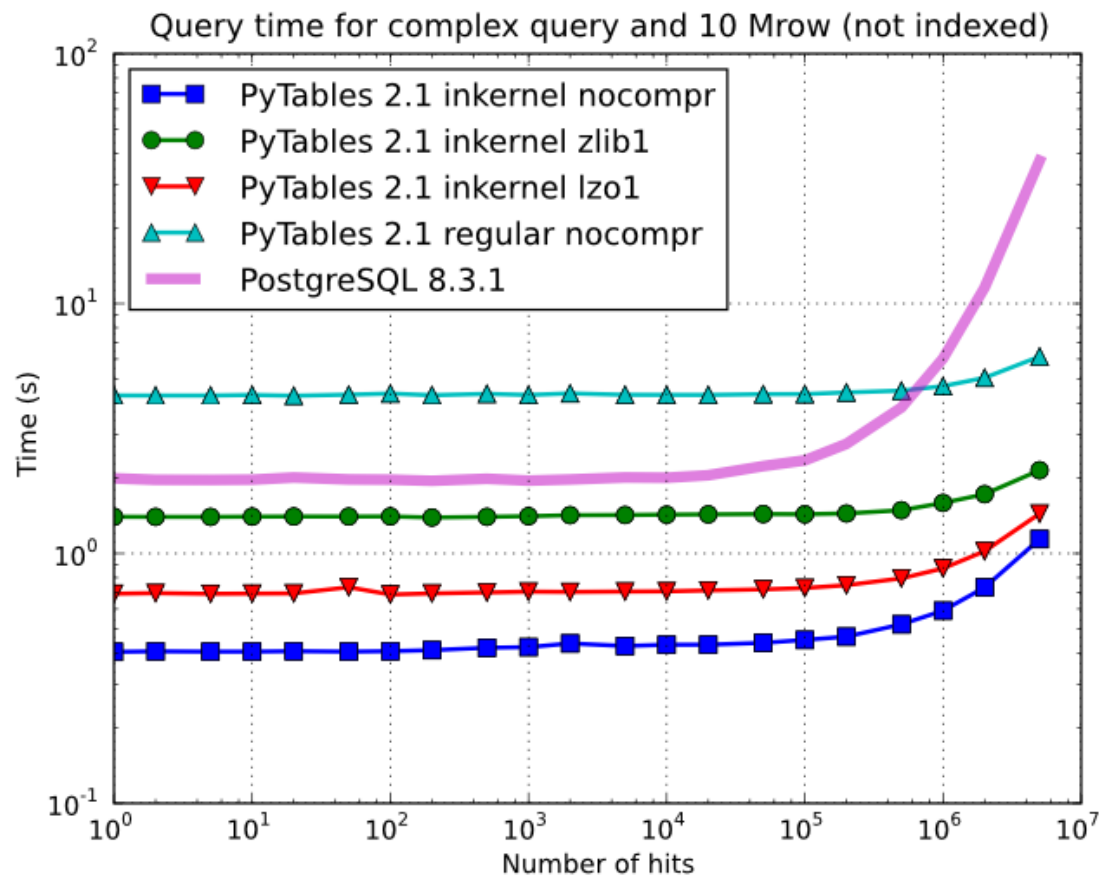


Figure 5.5: Figure 5. Times for non-indexed complex queries in a small table with 10 millions of rows: the data fits in memory.

passed to the PyTables kernel (hence the name), written in C, and evaluated there at full C speed (with the help of the integrated Numexpr package, see ), so that the only values that are brought to Python space are the rows that fulfilled the condition. Hence, for selections that only have a relatively small number of hits (compared with the total amount of rows), the savings are very large. It is also interesting to note the fact that, although for queries with a large number of hits the speed-up is not as high, it is still very important.

On the other hand, when the table is too large to fit in memory (see [Figure 6](#)), the difference in speed between regular and in-kernel is not so important, but still significant (2x). Also, and curiously enough, large tables compressed with Zlib offers slightly better performance (around 20%) than uncompressed ones; this is because the additional CPU spent by the uncompressor is compensated by the savings in terms of net I/O (one has to read less actual data from disk). However, when using the extremely fast LZO compressor, it gives a clear advantage over Zlib, and is up to 2.5x faster than not using compression at all. The reason is that LZO decompression speed is much faster than Zlib, and that allows PyTables to read the data at full disk speed (i.e. the bottleneck is in the I/O subsystem, not in the CPU). In this case the compression rate is around 2.5x, and this is why the data can be read 2.5x faster. So, in general, using the LZO compressor is the best way to ensure best reading/querying performance for out-of-core datasets (more about how compression affects performance in [Compression issues](#)).

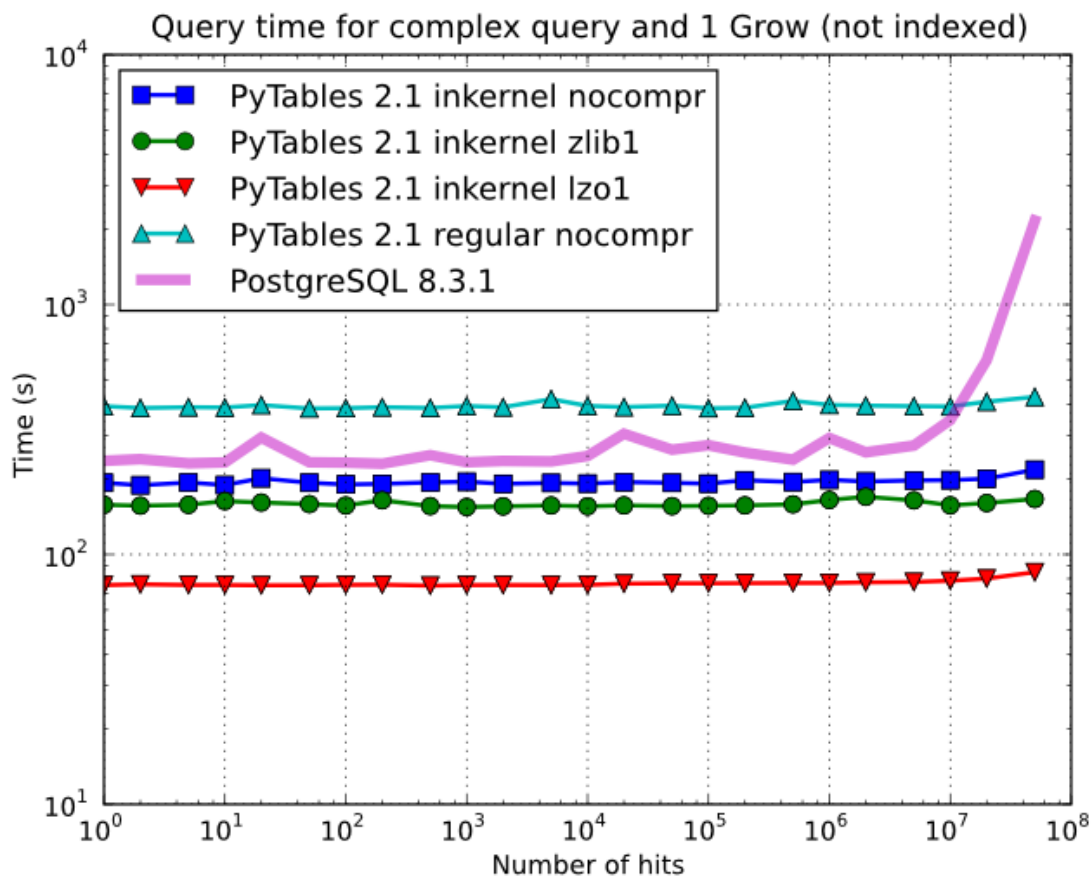


Figure 5.6: **Figure 6. Times for non-indexed complex queries in a large table with 1 billion of rows: the data does not fit in memory.**

Furthermore, you can mix the *in-kernel* and *regular* selection modes for evaluating arbitrarily complex conditions making use of external functions. Look at this example:

```
result = [ row['var2']  
           for row in table.where('(var3 == "foo") & (var1 <= 20)')  
           if your_function(row['var2']) ]
```

Here, we use an *in-kernel* selection to choose rows according to the values of the `var3` and `var1` fields. Then, we apply a *regular* selection to complete the query. Of course, when you mix the *in-kernel* and *regular* selection modes you should pass the most restrictive condition to the *in-kernel* part, i.e. to the `where()` iterator. In situations where it is not clear which is the most restrictive condition, you might want to experiment a bit in order to find the best combination.

However, since in-kernel condition strings allow rich expressions allowing the coexistence of multiple columns, variables, arithmetic operations and many typical functions, it is unlikely that you will be forced to use external regular selections in conditions of small to medium complexity. See [Condition Syntax](#) for more information on in-kernel condition syntax.

## 5.2.2 Indexed searches

When you need more speed than *in-kernel* selections can offer you, PyTables offers a third selection method, the so-called *indexed* mode (based on the highly efficient OPSI indexing engine). In this mode, you have to decide which column(s) you are going to apply your selections over, and index them. Indexing is just a kind of sorting operation over a column, so that searches along such a column (or columns) will look at this sorted information by using a *binary search* which is much faster than the *sequential search* described in the previous section.

You can index the columns you want by calling the `Column.createIndex()` method on an already created table. For example:

```
indexrows = table.cols.var1.createIndex()  
indexrows = table.cols.var2.createIndex()  
indexrows = table.cols.var3.createIndex()
```

will create indexes for all `var1`, `var2` and `var3` columns.

After you have indexed a series of columns, the PyTables query optimizer will try hard to discover the usable indexes in a potentially complex expression. However, there are still places where it cannot determine that an index can be used. See below for examples where the optimizer can safely determine if an index, or series of indexes, can be used or not.

Example conditions where an index can be used:

- `var1 >= "foo"` (`var1` is used)
- `var1 >= mystr` (`var1` is used)
- `(var1 >= "foo") & (var4 > 0.0)` (`var1` is used)
- `("bar" <= var1) & (var1 < "foo")` (`var1` is used)
- `((("bar" <= var1) & (var1 < "foo"))) & (var4 > 0.0)` (`var1` is used)
- `(var1 >= "foo") & (var3 > 10)` (`var1` and `var3` are used)
- `(var1 >= "foo") | (var3 > 10)` (`var1` and `var3` are used)
- `~(var1 >= "foo") | ~(var3 > 10)` (`var1` and `var3` are used)

Example conditions where an index can *not* be used:

- `var4 > 0.0` (`var4` is not indexed)
- `var1 != 0.0` (range has two pieces)
- `~(("bar" <= var1) & (var1 < "foo")) & (var4 > 0.0)` (negation of a complex boolean expression)

**Note:** From PyTables 2.3 on, several indexes can be used in a single query.

**Note:** If you want to know for sure whether a particular query will use indexing or not (without actually running it), you are advised to use the `Table.willQueryUseIndexing()` method.

One important aspect of the new indexing in PyTables ( $\geq 2.3$ ) is that it has been designed from the ground up with the goal of being capable to effectively manage very large tables. To this goal, it sports a wide spectrum of different quality levels (also called optimization levels) for its indexes so that the user can choose the best one that suits her needs (more or less size, more or less performance).

In [Figure 7](#), you can see that the times to index columns in tables can be really short. In particular, the time to index a column with 1 billion rows (1 Gigarow) with the lowest optimization level is less than 4 minutes while indexing the same column with full optimization (so as to get a completely sorted index or CSI) requires around 1 hour. These are rather competitive figures compared with a relational database (in this case, PostgreSQL 8.3.1, which takes around 1.5 hours for getting the index done). This is because PyTables is geared towards read-only or append-only tables and takes advantage of this fact to optimize the indexes properly. On the contrary, most relational databases have to deliver decent performance in other scenarios as well (specially updates and deletions), and this fact leads not only to slower index creation times, but also to indexes taking much more space on disk, as you can see in [Figure 8](#).

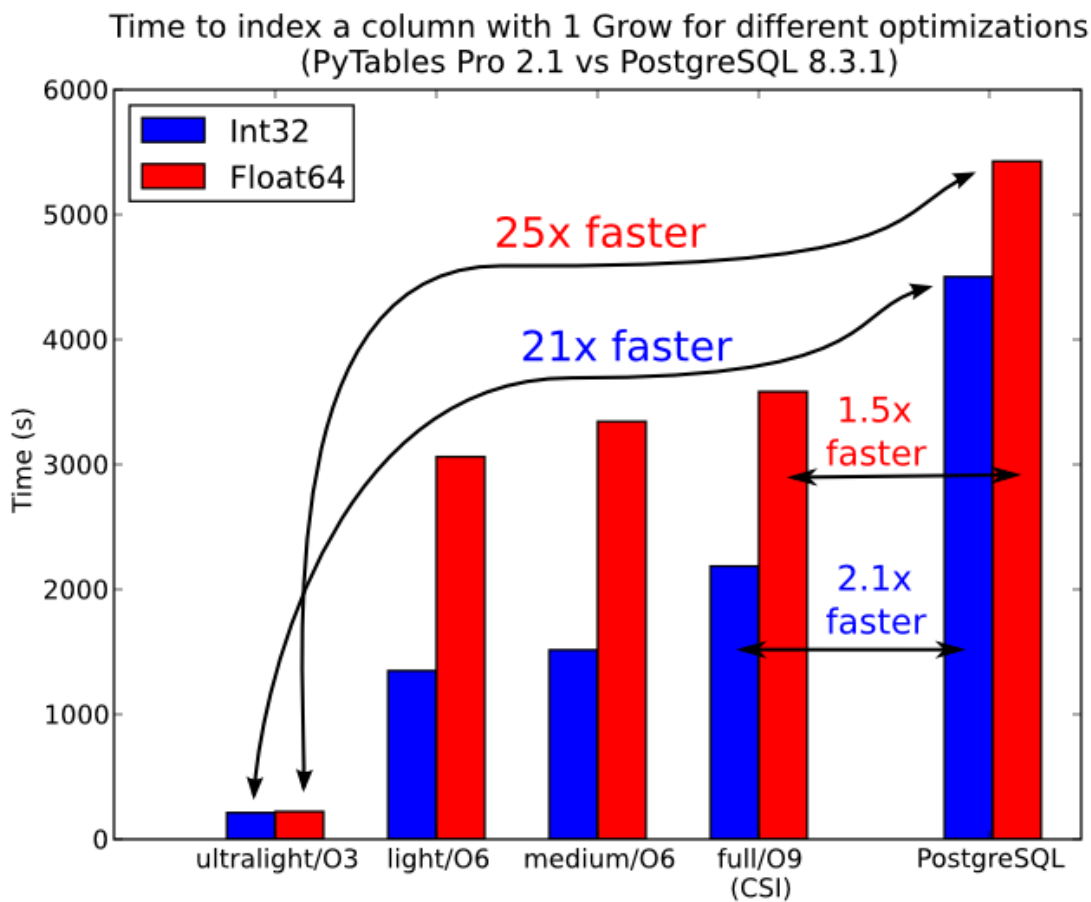


Figure 5.7: Figure 7. Times for indexing an Int32 and Float64 column.

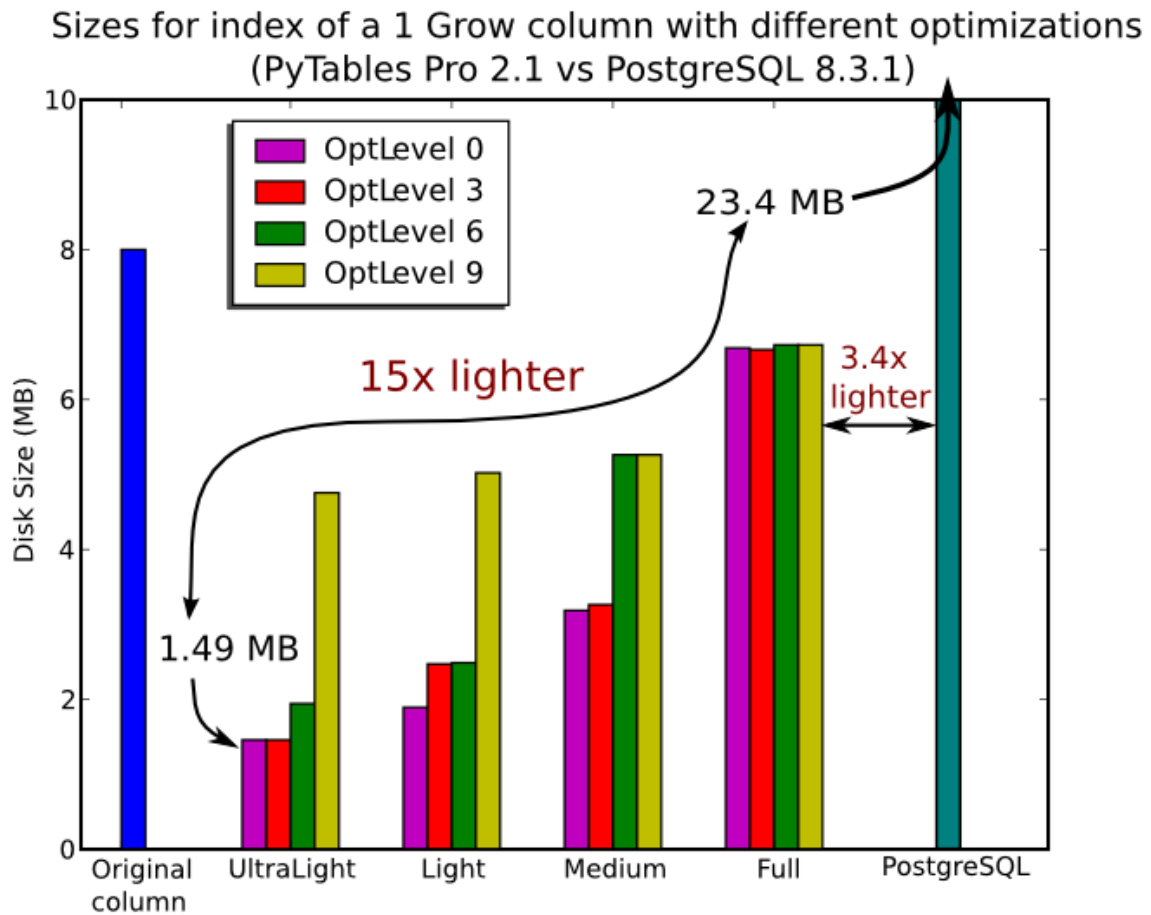


Figure 5.8: Figure 8. Sizes for an index of a Float64 column with 1 billion of rows.

The user can select the index quality by passing the desired `optlevel` and `kind` arguments to the `Column.createIndex()` method. We can see in figures [Figure 7](#) and [Figure 8](#) how the different optimization levels affects index time creation and index sizes.

So, which is the effect of the different optimization levels in terms of query times? You can see that in [Figure 9](#).

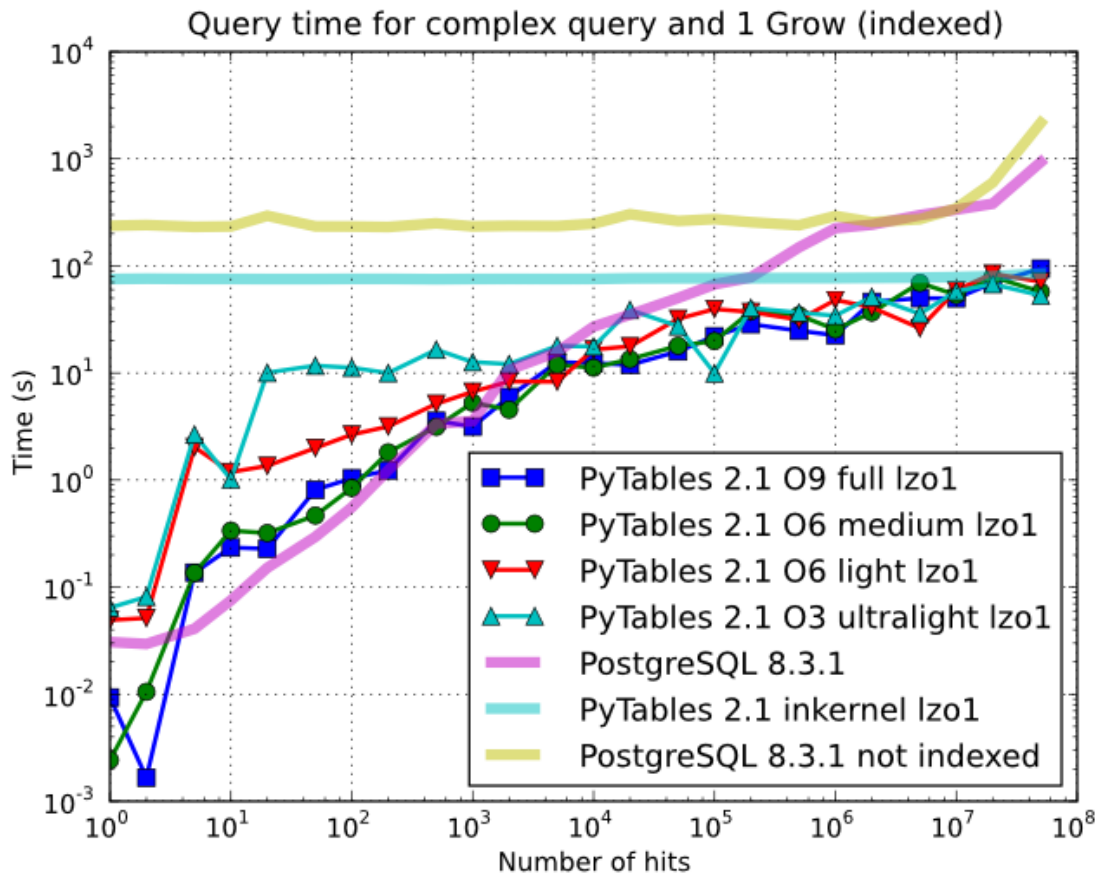


Figure 5.9: **Figure 9. Times for complex queries with a cold cache (mean of 5 first random queries) for different optimization levels. Benchmark made on a machine with Intel Core2 (64-bit) @ 3 GHz processor with RAID-0 disk storage.**

Of course, compression also has an effect when doing indexed queries, although not very noticeable, as can be seen in [Figure 10](#). As you can see, the difference between using no compression and using Zlib or LZO is very little, although LZO achieves relatively better performance generally speaking.

You can find a more complete description and benchmarks about OPSI, the indexing system of PyTables ( $\geq 2.3$ ) in [\[OPSI\]](#).

### 5.2.3 Indexing and Solid State Disks (SSD)

Lately, the long promised Solid State Disks (SSD for brevity) with decent capacities and affordable prices have finally hit the market and will probably stay in coexistence with the traditional spinning disks for the foreseeable future (separately or forming *hybrid* systems). SSD have many advantages over spinning disks, like much less power consumption and better throughput. But of paramount importance, specially in the context of accelerating indexed queries, is its

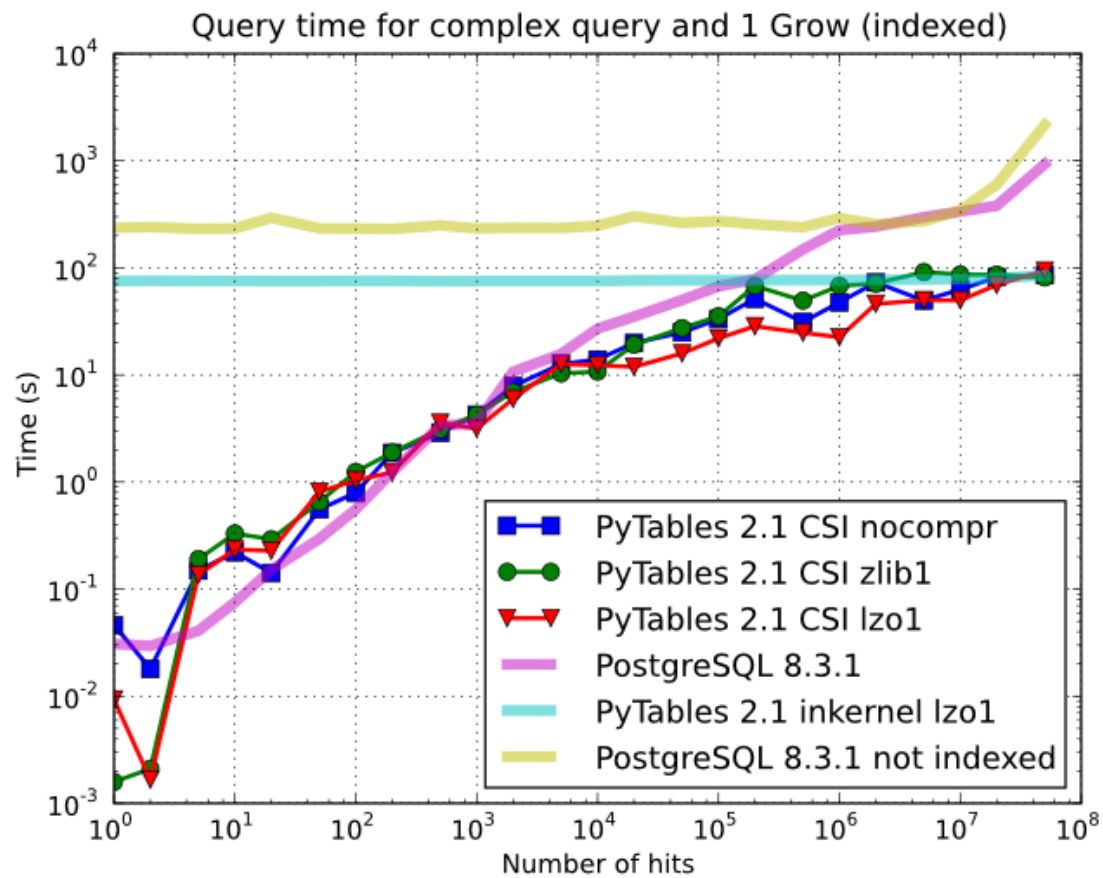


Figure 5.10: **Figure 10.** Times for complex queries with a cold cache (mean of 5 first random queries) for different compressors.

very reduced latency during disk seeks, which is typically 100x better than traditional disks. Such a huge improvement has to have a clear impact in reducing the query times, specially when the selectivity is high (i.e. the number of hits is small).

In order to offer an estimate on the performance improvement we can expect when using a low-latency SSD instead of traditional spinning disks, the benchmark in the previous section has been repeated, but this time using a single SSD disk instead of the four spinning disks in RAID-0. The result can be seen in [Figure 11](#). There one can see how a query in a table of 1 billion of rows with 100 hits took just 1 tenth of second when using a SSD, instead of 1 second that needed the RAID made of spinning disks. This factor of 10x of speed-up for high-selectivity queries is nothing to sneeze at, and should be kept in mind when really high performance in queries is needed. It is also interesting that using compression with LZO does have a clear advantage over when no compression is done.

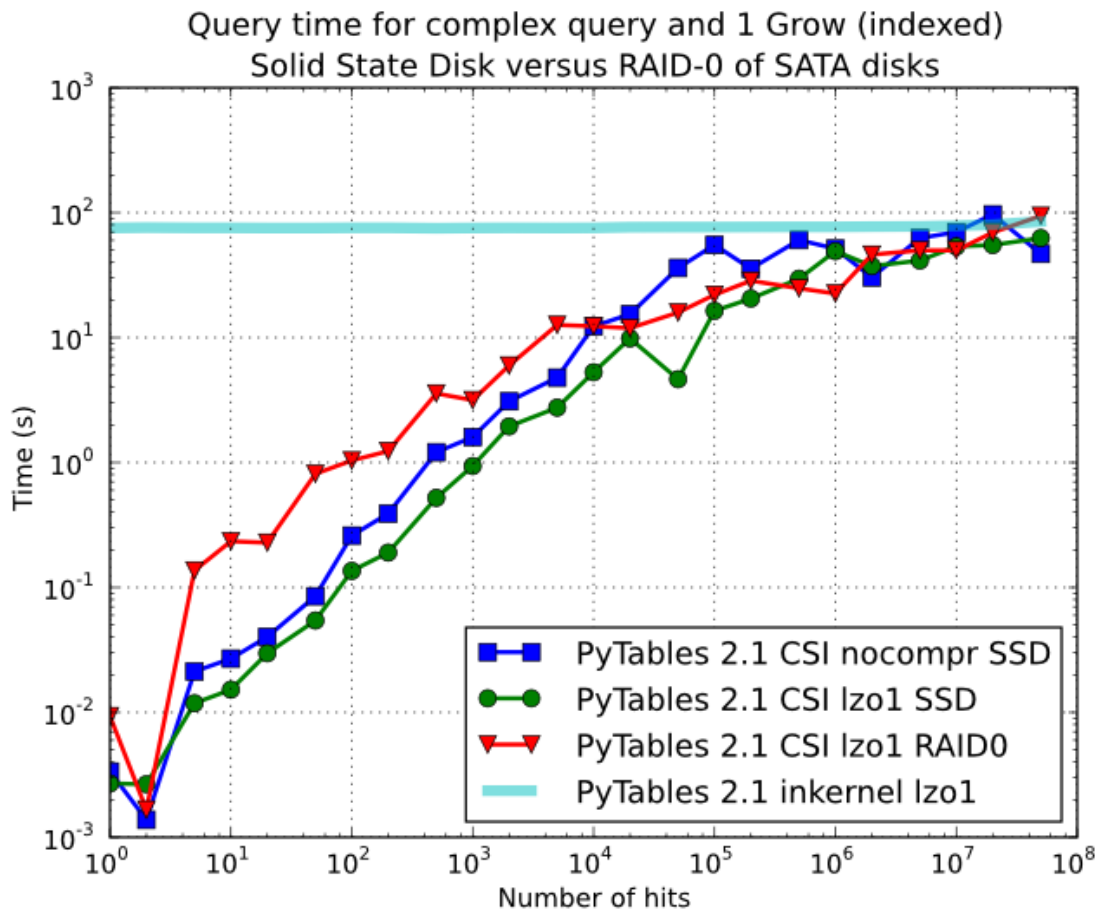


Figure 5.11: **Figure 11. Times for complex queries with a cold cache (mean of 5 first random queries) for different disk storage (SSD vs spinning disks).**

Finally, we should remark that SSD can't compete with traditional spinning disks in terms of capacity as they can only provide, for a similar cost, between 1/10th and 1/50th of the size of traditional disks. It is here where the compression capabilities of PyTables can be very helpful because both tables and indexes can be compressed and the final space can be reduced by typically 2x to 5x (4x to 10x when compared with traditional relational databases). Best of all, as already mentioned, performance is not degraded when compression is used, but actually *improved*. So, by using PyTables and SSD you can query larger datasets that otherwise would require spinning disks when using other databases

In fact, we were unable to run the PostgreSQL benchmark in this case because the space needed exceeded the capacity of our SSD., while allowing improvements in the speed of indexed queries between 2x (for medium to low selectivity

queries) and 10x (for high selectivity queries).

## 5.2.4 Achieving ultimate speed: sorted tables and beyond

**Warning:** Sorting a large table is a costly operation. The next procedure should only be performed when your dataset is mainly read-only and meant to be queried many times.

When querying large tables, most of the query time is spent in locating the interesting rows to be read from disk. In some occasions, you may have queries whose result depends *mainly* of one single column (a query with only one single condition is the trivial example), so we can guess that sorting the table by this column would lead to locate the interesting rows in a much more efficient way (because they would be mostly *contiguous*). We are going to confirm this guess.

For the case of the query that we have been using in the previous sections:

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

it is possible to determine, by analysing the data distribution and the query limits, that col4 is such a *main column*. So, by ordering the table by the col4 column (for example, by specifying setting the column to sort by in the `sortby` parameter in the `Table.copy()` method and re-indexing col2 and col4 afterwards, we should get much faster performance for our query. This is effectively demonstrated in [Figure 12](#), where one can see how queries with a low to medium (up to 10000) number of hits can be done in around 1 tenth of second for a RAID-0 setup and in around 1 hundredth of second for a SSD disk. This represents up to more that 100x improvement in speed with respect to the times with unsorted tables. On the other hand, when the number of hits is large (> 1 million), the query times grow almost linearly, showing a near-perfect scalability for both RAID-0 and SSD setups (the sequential access to disk becomes the bottleneck in this case).

Even though we have shown many ways to improve query times that should fulfill the needs of most of people, for those needing more, you can for sure discover new optimization opportunities. For example, querying against sorted tables is limited mainly by sequential access to data on disk and data compression capability, so you may want to read [Fine-tuning the chunksize](#), for ways on improving this aspect. Reading the other sections of this chapter will help in finding new roads for increasing the performance as well. You know, the limit for stopping the optimization process is basically your imagination (but, most plausibly, your available time ;-).

## 5.3 Compression issues

One of the beauties of PyTables is that it supports compression on tables and arrays <sup>2</sup>, although it is not used by default. Compression of big amounts of data might be a bit controversial feature, because it has a legend of being a very big consumer of CPU time resources. However, if you are willing to check if compression can help not only by reducing your dataset file size but *also* by improving I/O efficiency, specially when dealing with very large datasets, keep reading.

### 5.3.1 A study on supported compression libraries

The compression library used by default is the *Zlib* (see [\[ZLIB\]](#)). Since *HDF5 requires* it, you can safely use it and expect that your HDF5 files will be readable on any other platform that has HDF5 libraries installed. Zlib provides

---

<sup>2</sup> Except for Array objects.

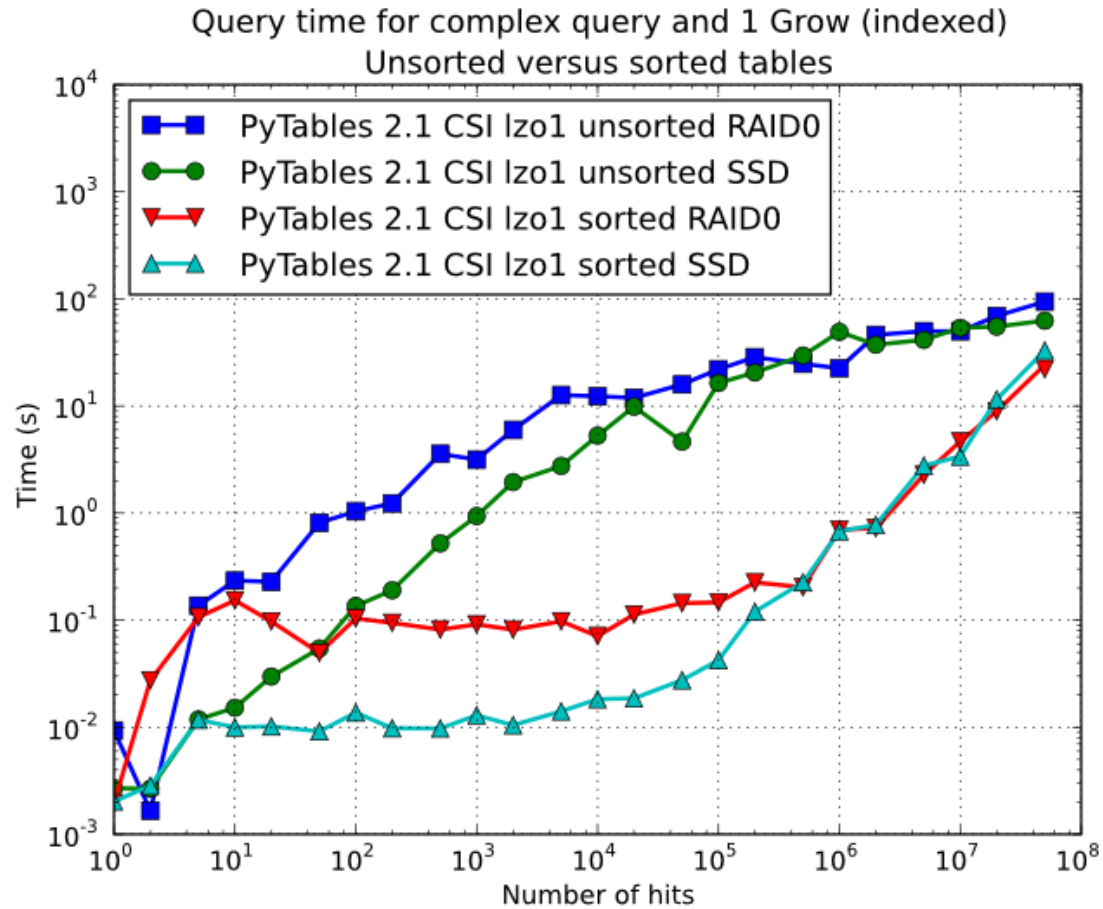


Figure 5.12: **Figure 12.** Times for complex queries with a cold cache (mean of 5 first random queries) for unsorted and sorted tables.

good compression ratio, although somewhat slow, and reasonably fast decompression. Because of that, it is a good candidate to be used for compressing you data.

However, in some situations it is critical to have a *very good decompression speed* (at the expense of lower compression ratios or more CPU wasted on compression, as we will see soon). In others, the emphasis is put in achieving the *maximum compression ratios*, no matter which reading speed will result. This is why support for two additional compressors has been added to PyTables: LZO (see [\[LZO\]](#)) and bzip2 (see [\[BZIP2\]](#)). Following the author of LZO (and checked by the author of this section, as you will see soon), LZO offers pretty fast compression and extremely fast decompression. In fact, LZO is so fast when compressing/decompressing that it may well happen (that depends on your data, of course) that writing or reading a compressed dataset is sometimes faster than if it is not compressed at all (specially when dealing with extremely large datasets). This fact is very important, specially if you have to deal with very large amounts of data. Regarding bzip2, it has a reputation of achieving excellent compression ratios, but at the price of spending much more CPU time, which results in very low compression/decompression speeds.

Be aware that the LZO and bzip2 support in PyTables is not standard on HDF5, so if you are going to use your PyTables files in other contexts different from PyTables you will not be able to read them. Still, see the [ptrepack](#) (where the ptrepack utility is described) to find a way to free your files from LZO or bzip2 dependencies, so that you can use these compressors locally with the warranty that you can replace them with Zlib (or even remove compression completely) if you want to use these files with other HDF5 tools or platforms afterwards.

In order to allow you to grasp what amount of compression can be achieved, and how this affects performance, a series of experiments has been carried out. All the results presented in this section (and in the next one) have been obtained with synthetic data and using PyTables 1.3. Also, the tests have been conducted on a IBM OpenPower 720 (e-series) with a PowerPC G5 at 1.65 GHz and a hard disk spinning at 15K RPM. As your data and platform may be totally different for your case, take this just as a guide because your mileage may vary. Finally, and to be able to play with tables with a number of rows as large as possible, the record size has been chosen to be small (16 bytes). Here is its definition:

```
class Bench(IsDescription):
    var1 = StringCol(length=4)
    var2 = IntCol()
    var3 = FloatCol()
```

With this setup, you can look at the compression ratios that can be achieved in [Figure 13](#). As you can see, LZO is the compressor that performs worse in this sense, but, curiously enough, there is not much difference between Zlib and bzip2.

Also, PyTables lets you select different compression levels for Zlib and bzip2, although you may get a bit disappointed by the small improvement that these compressors show when dealing with a combination of numbers and strings as in our example. As a reference, see plot [Figure 14](#) for a comparison of the compression achieved by selecting different levels of Zlib. Very oddly, the best compression ratio corresponds to level 1 (!). See later for an explanation and more figures on this subject.

Have also a look at [Figure 15](#). It shows how the speed of writing rows evolves as the size (number of rows) of the table grows. Even though in these graphs the size of one single row is 16 bytes, you can most probably extrapolate these figures to other row sizes.

In [Figure 16](#) you can see how compression affects the reading performance. In fact, what you see in the plot is an *in-kernel selection* speed, but provided that this operation is very fast (see [In-kernel searches](#)), we can accept it as an actual read test. Compared with the reference line without compression, the general trend here is that LZO does not affect too much the reading performance (and in some points it is actually better), Zlib makes speed drop to a half, while bzip2 is performing very slow (up to 8x slower).

Also, in the same [Figure 16](#) you can notice some strange peaks in the speed that we might be tempted to attribute to libraries on which PyTables relies (HDF5, compressors...), or to PyTables itself. However, [Figure 17](#) reveals that, if we put the file in the filesystem cache (by reading it several times before, for example), the evolution of the performance is much smoother. So, the most probable explanation would be that such peaks are a consequence of the underlying OS filesystem, rather than a flaw in PyTables (or any other library behind it). Another consequence that can be

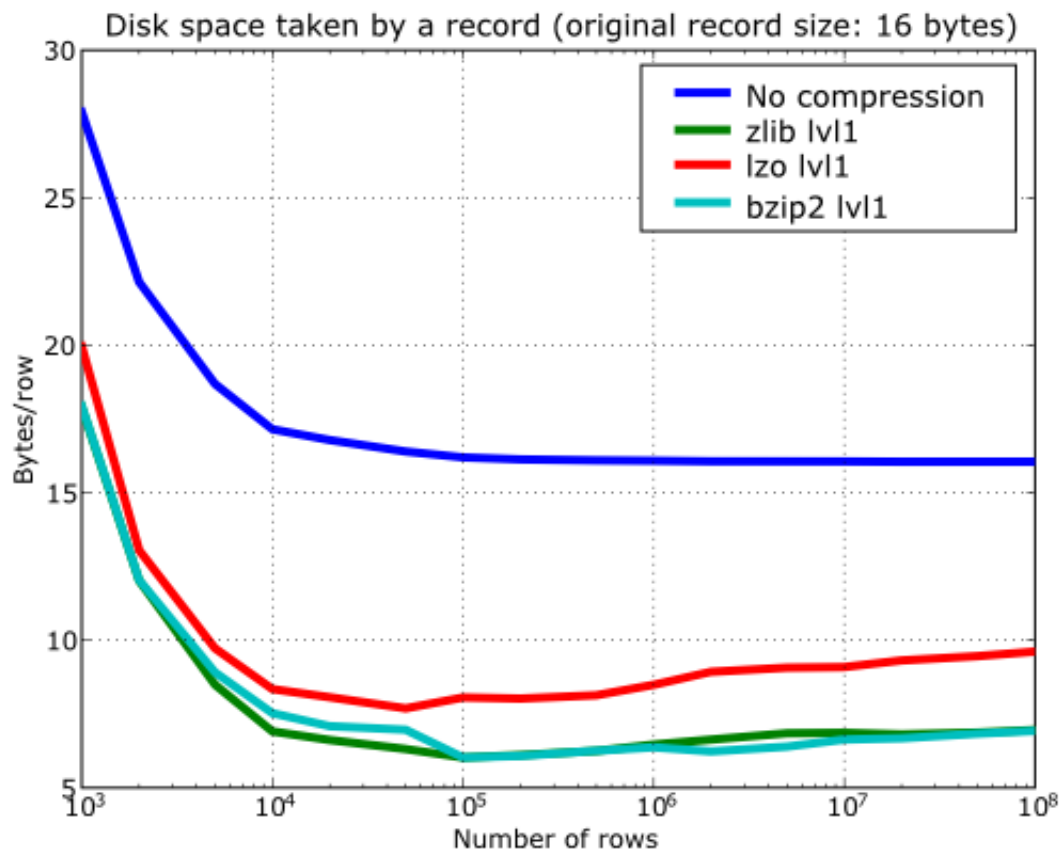


Figure 5.13: **Figure 13.** Comparison between different compression libraries.

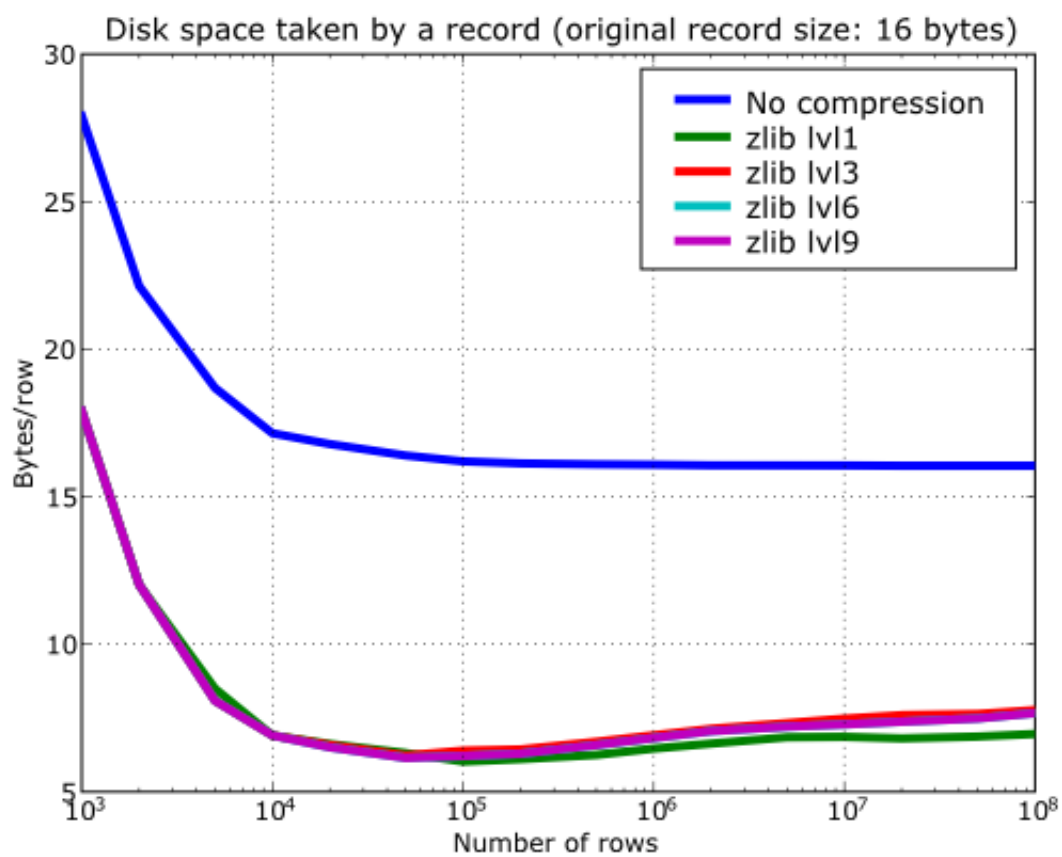


Figure 5.14: **Figure 14.** Comparison between different compression levels of Zlib.

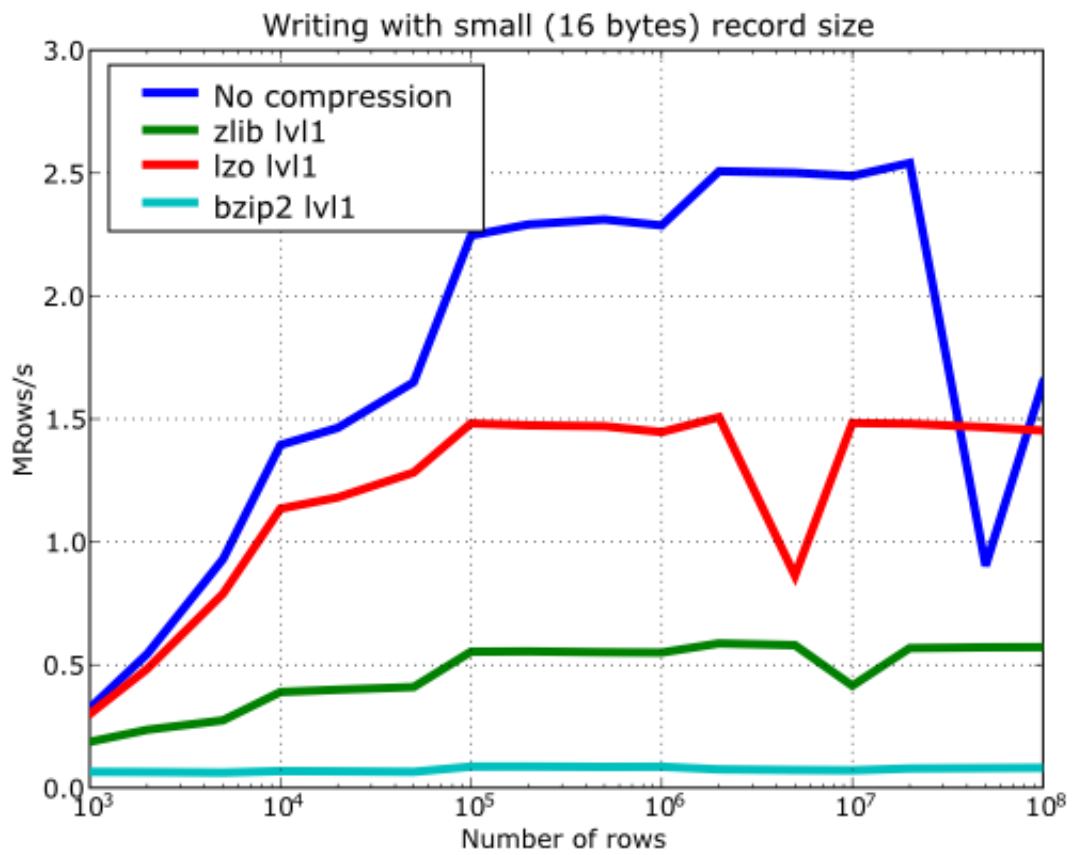


Figure 5.15: Figure 15. Writing tables with several compressors.

derived from the aforementioned plot is that LZO decomposition performance is much better than Zlib, allowing an improvement in overall speed of more than 2x, and perhaps more important, the read performance for really large datasets (i.e. when they do not fit in the OS filesystem cache) can be actually *better* than not using compression at all. Finally, one can see that reading performance is very badly affected when bzip2 is used (it is 10x slower than LZO and 4x than Zlib), but this was somewhat expected anyway.

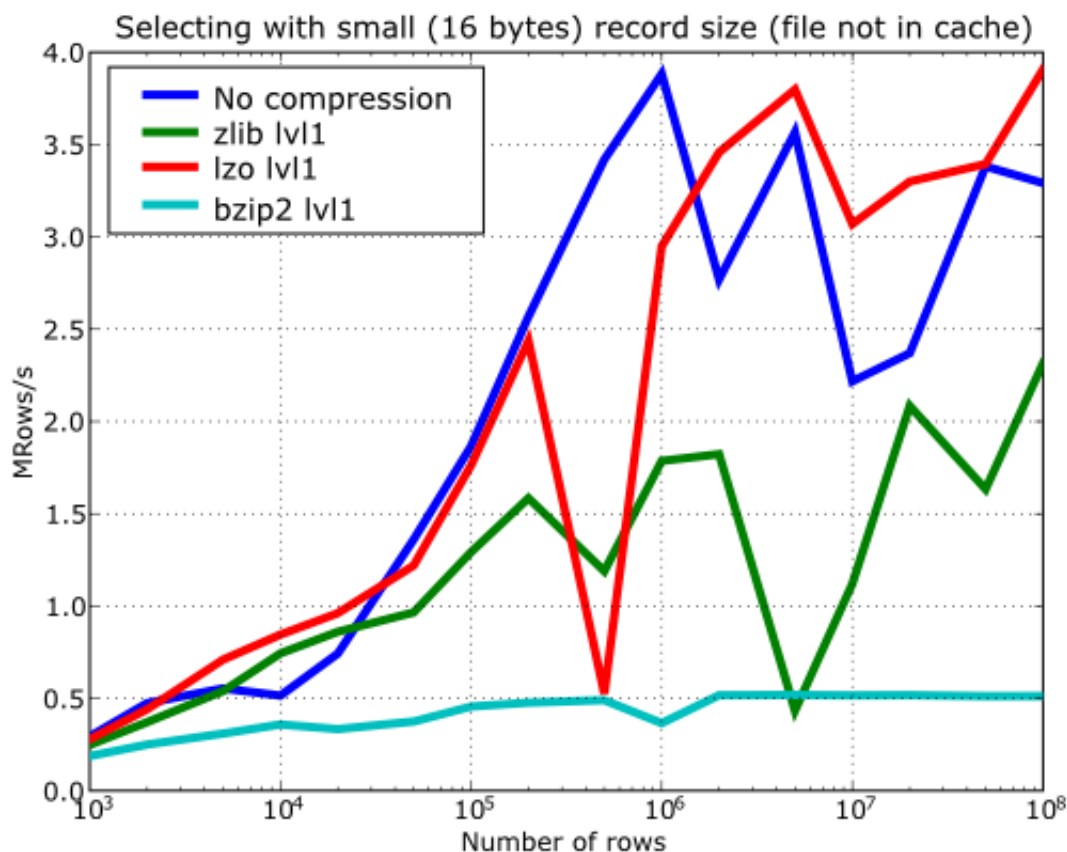


Figure 5.16: **Figure 16. Selecting values in tables with several compressors. The file is not in the OS cache.**

So, generally speaking and looking at the experiments above, you can expect that LZO will be the fastest in both compressing and decompressing, but the one that achieves the worse compression ratio (although that may be just OK for many situations, specially when used with shuffling - see [Shuffling \(or how to make the compression process more effective\)](#)). bzip2 is the slowest, by large, in both compressing and decompressing, and besides, it does not achieve any better compression ratio than Zlib. Zlib represents a balance between them: it's somewhat slow compressing (2x) and decompressing (3x) than LZO, but it normally achieves better compression ratios.

Finally, by looking at the plots [Figure 18](#), [Figure 19](#), and the aforementioned [Figure 14](#) you can see why the recommended compression level to use for all compression libraries is 1. This is the lowest level of compression, but as the size of the underlying HDF5 chunk size is normally rather small compared with the size of compression buffers, there is not much point in increasing the latter (i.e. increasing the compression level). Nonetheless, in some situations (like for example, in extremely large tables or arrays, where the computed chunk size can be rather large) you may want to check, on your own, how the different compression levels do actually affect your application.

You can select the compression library and level by setting the `complib` and `complevel` keywords in the `Filters` class (see [The Filters class](#)). A compression level of 0 will completely disable compression (the default), 1 is the less memory and CPU time demanding level, while 9 is the maximum level and the most memory demanding and CPU

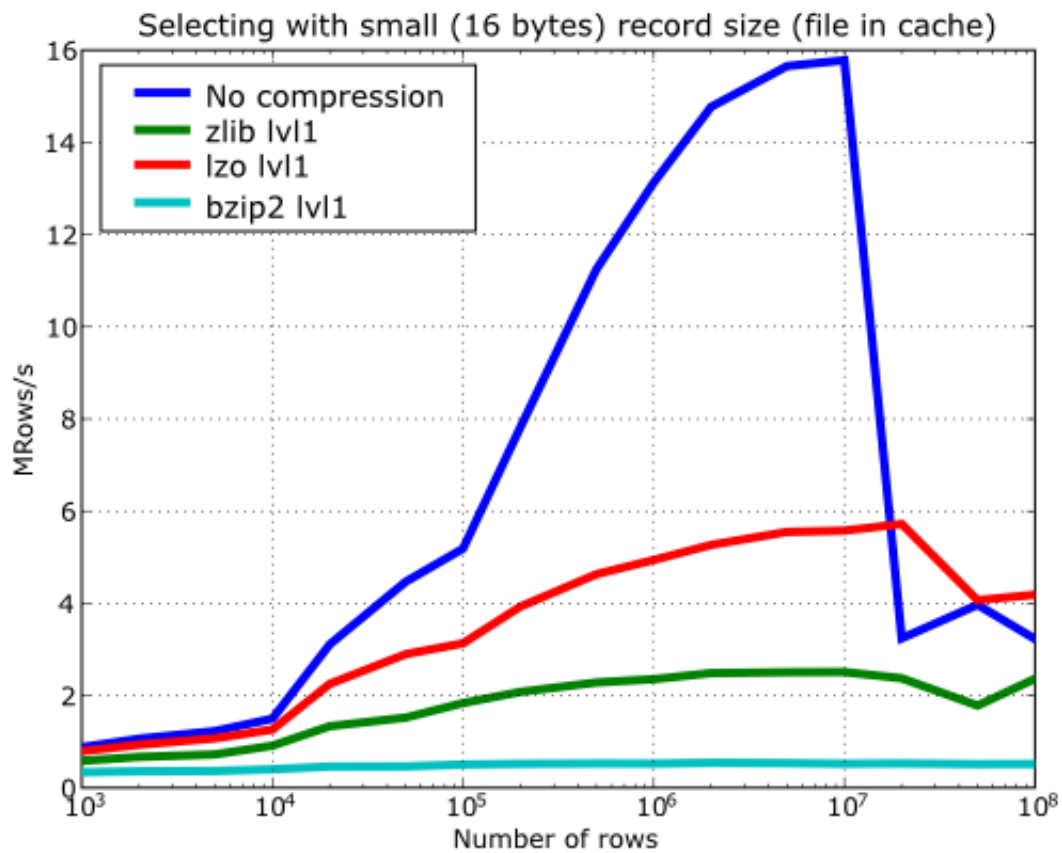


Figure 5.17: Figure 17. Selecting values in tables with several compressors. The file is in the OS cache.

intensive. Finally, have in mind that LZO is not accepting a compression level right now, so, when using LZO, 0 means that compression is not active, and any other value means that LZO is active.

So, in conclusion, if your ultimate goal is writing and reading as fast as possible, choose LZO. If you want to reduce as much as possible your data, while retaining acceptable read speed, choose Zlib. Finally, if portability is important for you, Zlib is your best bet. So, when you want to use bzip2? Well, looking at the results, it is difficult to recommend its use in general, but you may want to experiment with it in those cases where you know that it is well suited for your data pattern (for example, for dealing with repetitive string datasets).

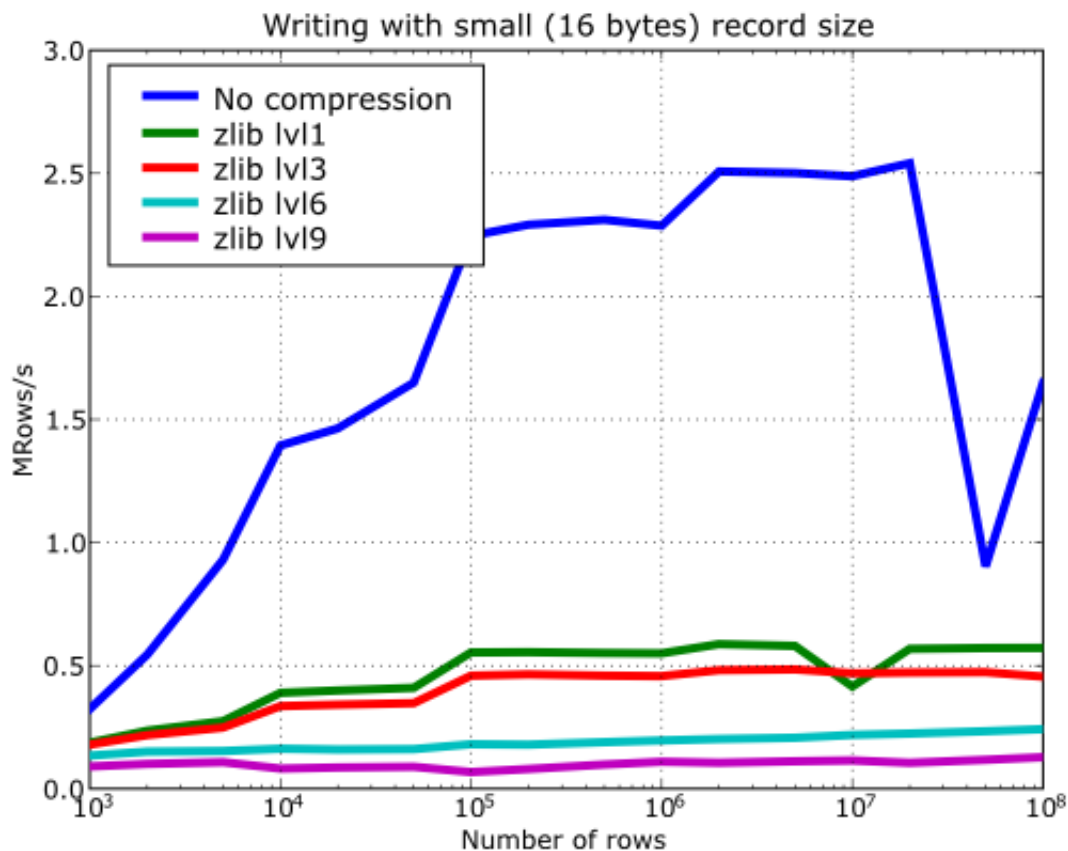


Figure 5.18: Figure 18. Writing in tables with different levels of compression.

### 5.3.2 Shuffling (or how to make the compression process more effective)

The HDF5 library provides an interesting filter that can leverage the results of your favorite compressor. Its name is *shuffle*, and because it can greatly benefit compression and it does not take many CPU resources (see below for a justification), it is active *by default* in PyTables whenever compression is activated (independently of the chosen compressor). It is deactivated when compression is off (which is the default, as you already should know). Of course, you can deactivate it if you want, but this is not recommended.

So, how does this mysterious filter exactly work? From the HDF5 reference manual:

```
"The shuffle filter de-interlaces a block of
data by reordering the bytes. All the bytes from one consistent byte
position of each data element are placed together in one block; all
```

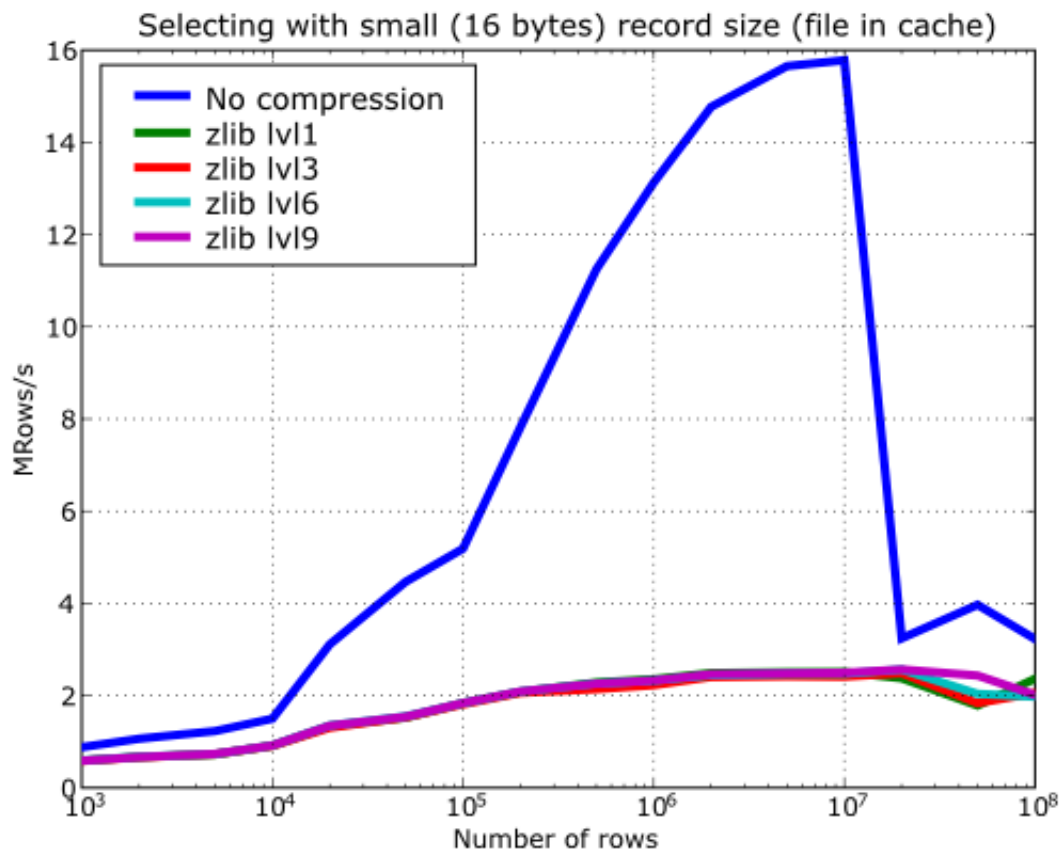


Figure 5.19: **Figure 19.** Selecting values in tables with different levels of compression. The file is in the OS cache.

bytes from a second consistent byte position of each data element are placed together a second block; etc. For example, given three data elements of a 4-byte datatype stored as 012301230123, shuffling will re-order data as 000111222333. This can be a valuable step in an effective compression algorithm because the bytes in each byte position are often closely related to each other and putting them together can increase the compression ratio."

In [Figure 20](#) you can see a benchmark that shows how the *shuffle* filter can help the different libraries in compressing data. In this experiment, shuffle has made LZO compress almost 3x more (!), while Zlib and bzip2 are seeing improvements of 2x. Once again, the data for this experiment is synthetic, and *shuffle* seems to do a great work with it, but in general, the results will vary in each case <sup>3</sup>.

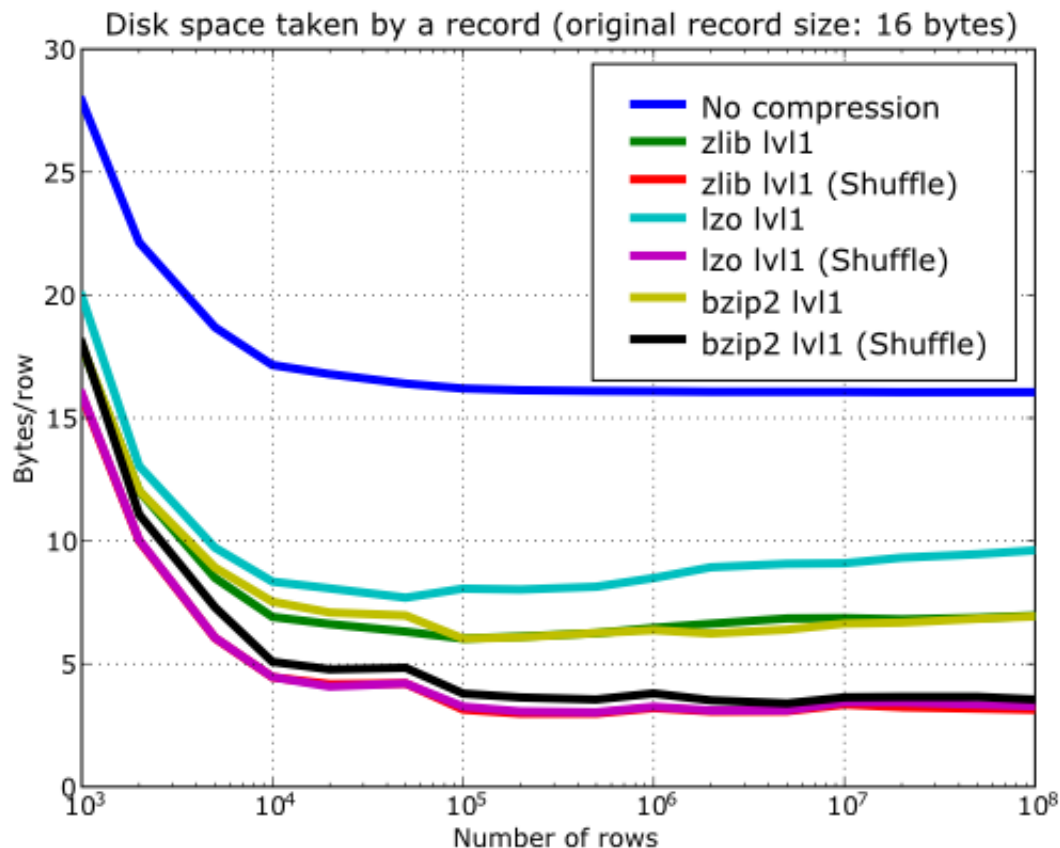


Figure 5.20: **Figure 20. Comparison between different compression libraries with and without the shuffle filter.**

At any rate, the most remarkable fact about the *shuffle* filter is the relatively high level of compression that compressor filters can achieve when used in combination with it. A curious thing to note is that the Bzip2 compression rate does not seem very much improved (less than a 40%), and what is more striking, Bzip2+shuffle does compress quite *less* than Zlib+shuffle or LZO+shuffle combinations, which is kind of unexpected. The thing that seems clear is that Bzip2 is not very good at compressing patterns that result of shuffle application. As always, you may want to experiment with your own data before widely applying the Bzip2+shuffle combination in order to avoid surprises.

Now, how does shuffling affect performance? Well, if you look at plots [Figure 21](#), [Figure 22](#) and [Figure 23](#), you will get

<sup>3</sup> Some users reported that the typical improvement with real data is between a factor 1.5x and 2.5x over the already compressed datasets.

a somewhat unexpected (but pleasant) surprise. Roughly, *shuffle* makes the writing process (shuffling+compressing) faster (approximately a 15% for LZO, 30% for Bzip2 and a 80% for Zlib), which is an interesting result by itself. But perhaps more exciting is the fact that the reading process (unshuffling+decompressing) is also accelerated by a similar extent (a 20% for LZO, 60% for Zlib and a 75% for Bzip2, roughly).

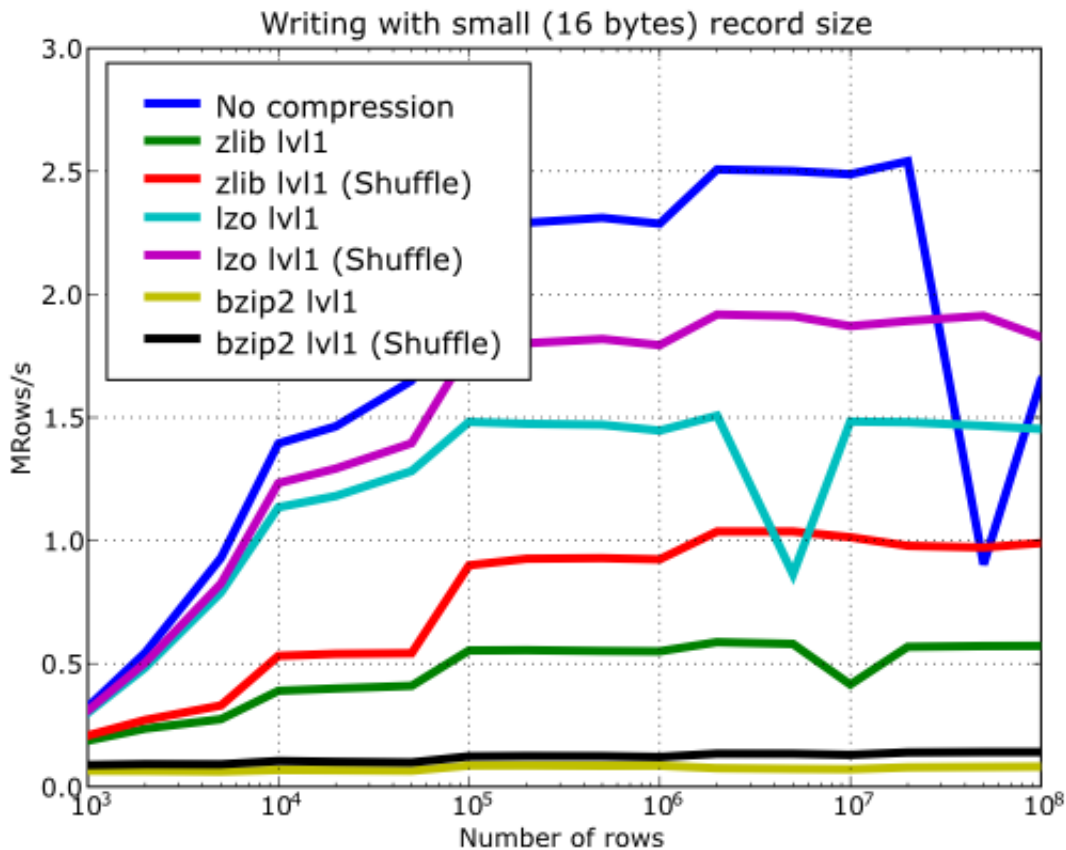


Figure 5.21: **Figure 21. Writing with different compression libraries with and without the shuffle filter.**

You may wonder why introducing another filter in the write/read pipelines does effectively accelerate the throughput. Well, maybe data elements are more similar or related column-wise than row-wise, i.e. contiguous elements in the same column are more alike, so shuffling makes the job of the compressor easier (faster) and more effective (greater ratios). As a side effect, compressed chunks do fit better in the CPU cache (at least, the chunks are smaller!) so that the process of unshuffle/decompress can make a better use of the cache (i.e. reducing the number of CPU cache faults).

So, given the potential gains (faster writing and reading, but specially much improved compression level), it is a good thing to have such a filter enabled by default in the battle for discovering redundancy when you want to compress your data, just as PyTables does.

## 5.4 Using Psyco

Psyco (see [\[PSYCO\]](#)) is a kind of specialized compiler for Python that typically accelerates Python applications with no change in source code. You can think of Psyco as a kind of just-in-time (JIT) compiler, a little bit like Java's, that emits machine code on the fly instead of interpreting your Python program step by step. The result is that your unmodified Python programs run faster.

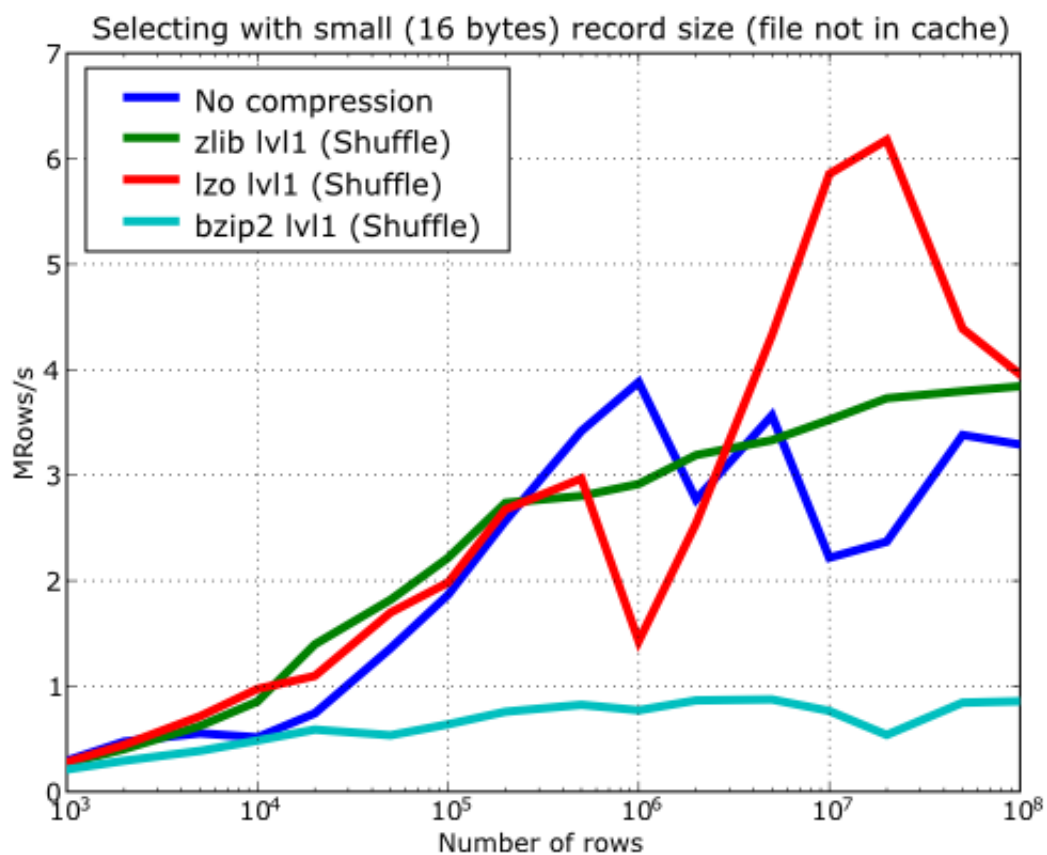


Figure 5.22: **Figure 22. Reading with different compression libraries with the shuffle filter. The file is not in OS cache.**

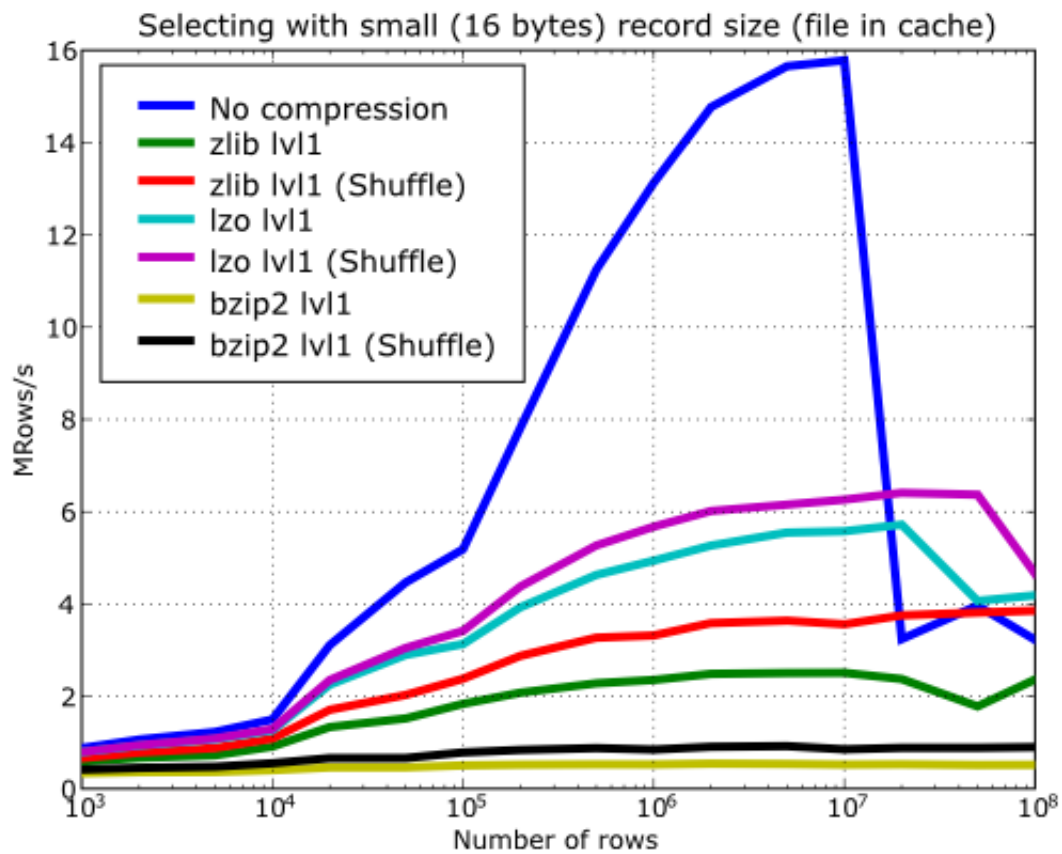


Figure 5.23: **Figure 23.** Reading with different compression libraries with and without the shuffle filter. The file is in OS cache.

Psyco is very easy to install and use, so in most scenarios it is worth to give it a try. However, it only runs on Intel 386 architectures, so if you are using other architectures, you are out of luck (and, moreover, it seems that there are no plans to support other platforms). Besides, with the addition of flexible (and very fast) in-kernel queries (by the way, they cannot be optimized at all by Psyco), the use of Psyco will only help in rather few scenarios. In fact, the only important situation that you might benefit right now from using Psyco (I mean, in PyTables contexts) is for speeding-up the write speed in tables when using the Row interface (see [The Row class](#)). But again, this latter case can also be accelerated by using the `Table.append()` method and building your own buffers<sup>4</sup>.

As an example, imagine that you have a small script that reads and selects data over a series of datasets, like this:

```
def readFile(filename):
    "Select data from all the tables in filename"
    fileh = openFile(filename, mode = "r")
    result = []
    for table in fileh("/", 'Table'):
        result = [p['var3'] for p in table if p['var2'] <= 20]
    fileh.close()
    return result

if __name__=="__main__":
    print readFile("myfile.h5")
```

In order to accelerate this piece of code, you can rewrite your main program to look like:

```
if __name__=="__main__":
    import psyco
    psyco.bind(readFile)
    print readFile("myfile.h5")
```

That's all! From now on, each time that you execute your Python script, Psyco will deploy its sophisticated algorithms so as to accelerate your calculations.

You can see in the graphs [Figure 24](#) and [Figure 25](#) how much I/O speed improvement you can get by using Psyco. By looking at this figures you can get an idea if these improvements are of your interest or not. In general, if you are not going to use compression you will take advantage of Psyco if your tables are medium sized (from a thousand to a million rows), and this advantage will disappear progressively when the number of rows grows well over one million. However if you use compression, you will probably see improvements even beyond this limit (see [Compression issues](#)). As always, there is no substitute for experimentation with your own dataset.

## 5.5 Getting the most from the node LRU cache

One limitation of the initial versions of PyTables was that they needed to load all nodes in a file completely before being ready to deal with them, making the opening times for files with a lot of nodes very high and unacceptable in many cases.

Starting from PyTables 1.2 on, a new lazy node loading schema was setup that avoids loading all the nodes of the *object tree* in memory. In addition, a new LRU cache was introduced in order to accelerate the access to already visited nodes. This cache (one per file) is responsible for keeping up the most recently visited nodes in memory and discard the least recent used ones. This represents a big advantage over the old schema, not only in terms of memory usage (as there is no need to load *every* node in memory), but it also adds very convenient optimizations for working interactively like, for example, speeding-up the opening times of files with lots of nodes, allowing to open almost any kind of file in typically less than one tenth of second (compare this with the more than 10 seconds for files with more than 10000 nodes in PyTables pre-1.2 era) as well as optimizing the access to frequently visited nodes. See for more info on the advantages (and also drawbacks) of this approach.

---

<sup>4</sup> So, there is not much point in using Psyco with recent versions of PyTables anymore.

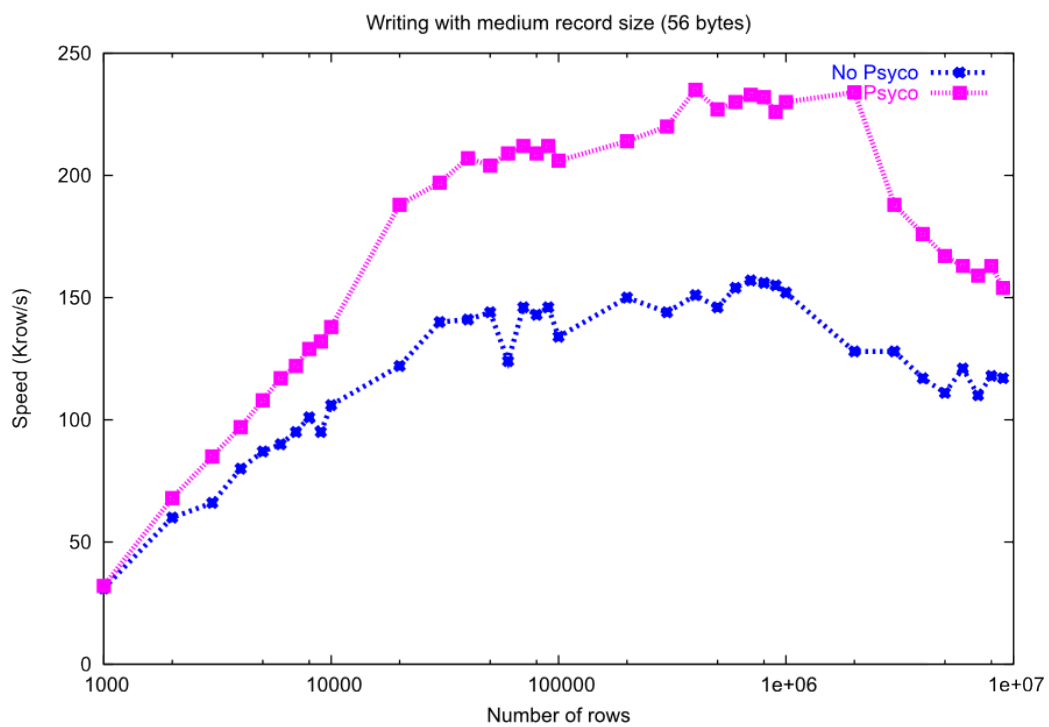


Figure 5.24: **Figure 24. Writing tables with/without Psyco.**

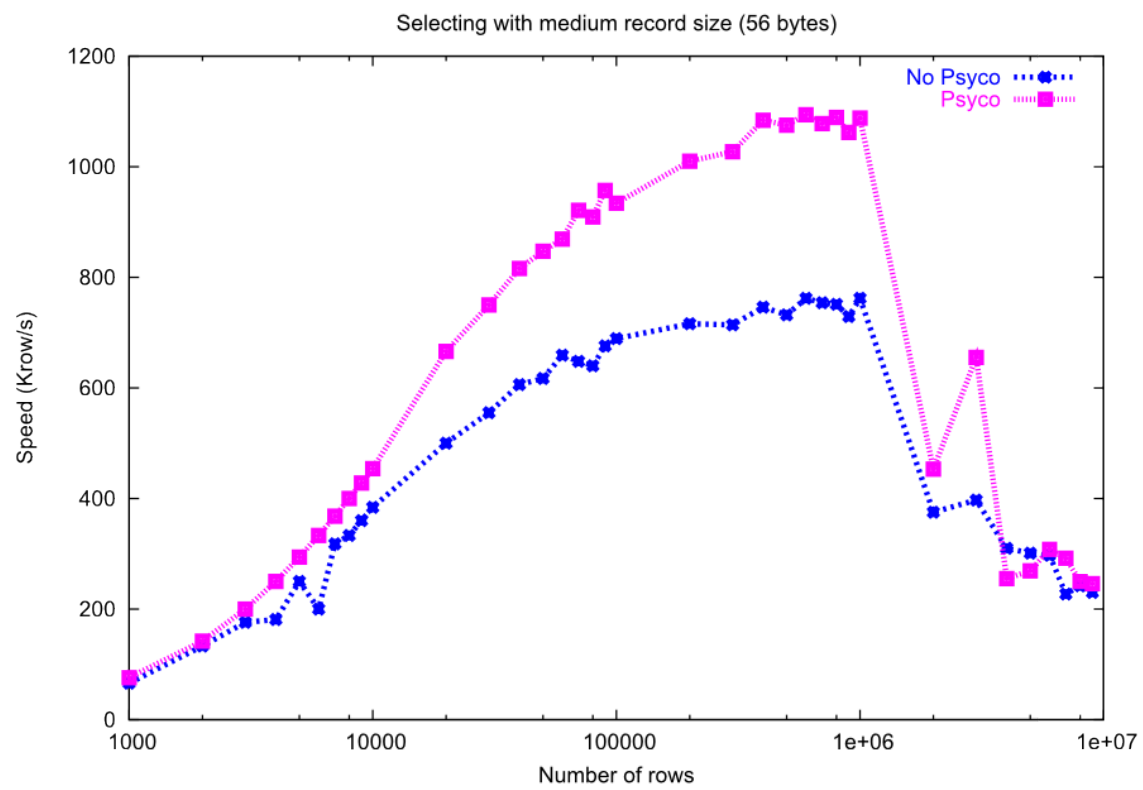


Figure 5.25: Figure 25. Reading tables with/without Psyc0.

One thing that deserves some discussion is the election of the parameter that sets the maximum amount of nodes to be kept in memory at any time. As PyTables is meant to be deployed in machines that can have potentially low memory, the default for it is quite conservative (you can look at its actual value in the `NODE_CACHE_SLOTS` parameter in module `tables/parameters.py`). However, if you usually need to deal with files that have many more nodes than the maximum default, and you have a lot of free memory in your system, then you may want to experiment in order to see which is the appropriate value of `NODE_CACHE_SLOTS` that fits better your needs.

As an example, look at the next code:

```
def browse_tables(filename):
    fileh = openFile(filename, 'a')
    group = fileh.root.newgroup
    for j in range(10):
        for tt in fileh.walkNodes(group, "Table"):
            title = tt.attrs.TITLE
            for row in tt:
                pass
    fileh.close()
```

We will be running the code above against a couple of files having a `/newgroup` containing 100 tables and 1000 tables respectively. In addition, this benchmark is run twice for two different values of the LRU cache size, specifically 256 and 1024. You can see the results in [table](#).

Table 5.1: Retrieval speed and memory consumption depending on the number of nodes in LRU cache.

| Number:                |            | 100 nodes   |      |           |      | 1000 nodes  |      |           |      |
|------------------------|------------|-------------|------|-----------|------|-------------|------|-----------|------|
| Mem and Speed          |            | Memory (MB) |      | Time (ms) |      | Memory (MB) |      | Time (ms) |      |
| Node is coming from... | Cache size | 256         | 1024 | 256       | 1024 | 256         | 1024 | 256       | 1024 |
| Disk                   |            | 14          | 14   | 1.24      | 1.24 | 51          | 66   | 1.33      | 1.31 |
| Cache                  |            | 14          | 14   | 0.53      | 0.52 | 65          | 73   | 1.35      | 0.68 |

From the data in [table](#), one can see that when the number of objects that you are dealing with does fit in cache, you will get better access times to them. Also, incrementing the node cache size effectively consumes more memory *only* if the total nodes exceeds the slots in cache; otherwise the memory consumption remains the same. It is also worth noting that incrementing the node cache size in the case you want to fit all your nodes in cache does not take much more memory than being too conservative. On the other hand, it might happen that the speed-up that you can achieve by allocating more slots in your cache is not worth the amount of memory used.

Also worth noting is that if you have a lot of memory available and performance is absolutely critical, you may want to try out a negative value for `NODE_CACHE_SLOTS`. This will cause that all the touched nodes will be kept in an internal dictionary and this is the faster way to load/retrieve nodes. However, and in order to avoid a large memory consumption, the user will be warned when the number of loaded nodes will reach the `-NODE_CACHE_SLOTS` value.

Finally, a value of zero in `NODE_CACHE_SLOTS` means that any cache mechanism is disabled.

At any rate, if you feel that this issue is important for you, there is no replacement for setting your own experiments up in order to proceed to fine-tune the `NODE_CACHE_SLOTS` parameter.

---

**Note:** PyTables >= 2.3 sports an optimized LRU cache node written in C, so you should expect significantly faster LRU cache operations when working with it.

---

## 5.6 Compacting your PyTables files

Let's suppose that you have a file where you have made a lot of row deletions on one or more tables, or deleted many leaves or even entire subtrees. These operations might leave *holes* (i.e. space that is not used anymore) in your files that may potentially affect not only the size of the files but, more importantly, the performance of I/O. This is because when you delete a lot of rows in a table, the space is not automatically recovered on the fly. In addition, if you add many more rows to a table than specified in the `expectedrows` keyword at creation time this may affect performance as well, as explained in [Informing PyTables about expected number of rows in tables or arrays](#).

In order to cope with these issues, you should be aware that PyTables includes a handy utility called `ptrepack` which can be very useful not only to compact *fragmented* files, but also to adjust some internal parameters in order to use better buffer and chunk sizes for optimum I/O speed. Please check the [ptrepack](#) for a brief tutorial on its use.

Another thing that you might want to use `ptrepack` for is changing the compression filters or compression levels on your existing data for different goals, like checking how this can affect both final size and I/O performance, or getting rid of the optional compressors like LZO or bzip2 in your existing files, in case you want to use them with generic HDF5 tools that do not have support for these filters.

---

## **Part II**

# **Complementary modules**



---

# FILENODE - SIMULATING A FILESYSTEM WITH PYTABLES

## 6.1 What is filenode?

filenode is a module which enables you to create a PyTables database of nodes which can be used like regular opened files in Python. In other words, you can store a file in a PyTables database, and read and write it as you would do with any other file in Python. Used in conjunction with PyTables hierarchical database organization, you can have your database turned into an open, extensible, efficient, high capacity, portable and metadata-rich filesystem for data exchange with other systems (including backup purposes).

Between the main features of filenode, one can list:

- *Open:* Since it relies on PyTables, which in turn, sits over HDF5 (see [\[HDGG1\]](#)), a standard hierarchical data format from NCSA.
- *Extensible:* You can define new types of nodes, and their instances will be safely preserved (as are normal groups, leafs and attributes) by PyTables applications having no knowledge of their types. Moreover, the set of possible attributes for a node is not fixed, so you can define your own node attributes.
- *Efficient:* Thanks to PyTables' proven extreme efficiency on handling huge amounts of data, filenode can make use of PyTables' on-the-fly compression and decompression of data.
- *High capacity:* Since PyTables and HDF5 are designed for massive data storage (they use 64-bit addressing even where the platform does not support it natively).
- *Portable:* Since the HDF5 format has an architecture-neutral design, and the HDF5 libraries and PyTables are known to run under a variety of platforms. Besides that, a PyTables database fits into a single file, which poses no trouble for transportation.
- *Metadata-rich:* Since PyTables can store arbitrary key-value pairs (even Python objects!) for every database node. Metadata may include authorship, keywords, MIME types and encodings, ownership information, access control lists (ACL), decoding functions and anything you can imagine!

## 6.2 Finding a filenode node

filenode nodes can be recognized because they have a `NODE_TYPE` system attribute with a 'file' value. It is recommended that you use the `File.getNodeAttr()` method of `tables.File` class to get the `NODE_TYPE` attribute independently of the nature (group or leaf) of the node, so you do not need to care about.

## 6.3 filenode - simulating files inside PyTables

The filenode module is part of the nodes sub-package of PyTables. The recommended way to import the module is:

```
>>> from tables.nodes import filenode
```

However, filenode exports very few symbols, so you can import `*` for interactive usage. In fact, you will most probably only use the `NodeType` constant and the `newNode()` and `openNode()` calls.

The `NodeType` constant contains the value that the `NODE_TYPE` system attribute of a node file is expected to contain ('file', as we have seen). Although this is not expected to change, you should use `filenode.NodeType` instead of the literal 'file' when possible.

`newNode()` and `openNode()` are the equivalent to the Python `file()` call (alias `open()`) for ordinary files. Their arguments differ from that of `file()`, but this is the only point where you will note the difference between working with a node file and working with an ordinary file.

For this little tutorial, we will assume that we have a PyTables database opened for writing. Also, if you are somewhat lazy at typing sentences, the code that we are going to explain is included in the `examples/filenodes1.py` file.

You can create a brand new file with these sentences:

```
>>> import tables
>>> h5file = tables.openFile('fnode.h5', 'w')
```

### 6.3.1 Creating a new file node

Creation of a new file node is achieved with the `newNode()` call. You must tell it in which PyTables file you want to create it, where in the PyTables hierarchy you want to create the node and which will be its name. The PyTables file is the first argument to `newNode()`; it will be also called the 'host PyTables file'. The other two arguments must be given as keyword arguments `where` and `name`, respectively. As a result of the call, a brand new appendable and readable file node object is returned.

So let us create a new node file in the previously opened `h5file` PyTables file, named 'fnode\_test' and placed right under the root of the database hierarchy. This is that command:

```
>>> fnode = filenode.newNode(h5file, where='/', name='fnode_test')
```

That is basically all you need to create a file node. Simple, isn't it? From that point on, you can use `fnode` as any opened Python file (i.e. you can write data, read data, lines of text and so on).

`newNode()` accepts some more keyword arguments. You can give a title to your file with the `title` argument. You can use PyTables' compression features with the `filters` argument. If you know beforehand the size that your file will have, you can give its final file size in bytes to the `expectedsize` argument so that the PyTables library would be able to optimize the data access.

`newNode()` creates a PyTables node where it is told to. To prove it, we will try to get the `NODE_TYPE` attribute from the newly created node:

```
>>> print h5file.getNodeAttr('/fnode_test', 'NODE_TYPE')
file
```

### 6.3.2 Using a file node

As stated above, you can use the new node file as any other opened file. Let us try to write some text in and read it:

```
>>> print >> fnode, "This is a test text line."
>>> print >> fnode, "And this is another one."
>>> print >> fnode
>>> fnode.write("Of course, file methods can also be used.")
>>>
>>> fnode.seek(0) # Go back to the beginning of file.
>>>
>>> for line in fnode:
...     print repr(line)
'This is a test text line.\n'
'And this is another one.\n'
'\n'
'Of course, file methods can also be used.'
```

This was run on a Unix system, so newlines are expressed as ‘\n’. In fact, you can override the line separator for a file by setting its `lineSeparator` property to any string you want.

While using a file node, you should take care of closing it *before* you close the PyTables host file. Because of the way PyTables works, your data it will not be at a risk, but every operation you execute after closing the host file will fail with a `ValueError`. To close a file node, simply delete it or call its `close()` method:

```
>>> fnode.close()
>>> print fnode.closed
True
```

### 6.3.3 Opening an existing file node

If you have a file node that you created using `newNode()`, you can open it later by calling `openNode()`. Its arguments are similar to that of `file()` or `open()`: the first argument is the PyTables node that you want to open (i.e. a node with a `NODE_TYPE` attribute having a ‘file’ value), and the second argument is a mode string indicating how to open the file. Contrary to `file()`, `openNode()` can not be used to create a new file node.

File nodes can be opened in read-only mode (‘r’) or in read-and-append mode (‘a+’). Reading from a file node is allowed in both modes, but appending is only allowed in the second one. Just like Python files do, writing data to an appendable file places it after the file pointer if it is on or beyond the end of the file, or otherwise after the existing data. Let us see an example:

```
>>> node = h5file.root.fnode_test
>>> fnode = filenode.openNode(node, 'a+')
>>> print repr(fnode.readline())
'This is a test text line.\n'
>>> print fnode.tell()
26
>>> print >> fnode, "This is a new line."
>>> print repr(fnode.readline())
''
```

Of course, the data append process places the pointer at the end of the file, so the last `readline()` call hit EOF. Let us seek to the beginning of the file to see the whole contents of our file:

```
>>> fnode.seek(0)
>>> for line in fnode:
...     print repr(line)
'This is a test text line.\n'
'And this is another one.\n'
'\n'
'Of course, file methods can also be used.This is a new line.\n'
```

As you can check, the last string we wrote was correctly appended at the end of the file, instead of overwriting the second line, where the file pointer was positioned by the time of the appending.

### 6.3.4 Adding metadata to a file node

You can associate arbitrary metadata to any open node file, regardless of its mode, as long as the host PyTables file is writable. Of course, you could use the `setNodeAttr()` method of `tables.File` to do it directly on the proper node, but `filenode` offers a much more comfortable way to do it. `filenode` objects have an `attrs` property which gives you direct access to their corresponding `AttributeSet` object.

For instance, let us see how to associate MIME type metadata to our file node:

```
>>> fnode.attrs.content_type = 'text/plain; charset=us-ascii'
```

As simple as A-B-C. You can put nearly anything in an attribute, which opens the way to authorship, keywords, permissions and more. Moreover, there is not a fixed list of attributes. However, you should avoid names in all caps or starting with `'_'`, since PyTables and `filenode` may use them internally. Some valid examples:

```
>>> fnode.attrs.author = "Ivan Vilata i Balaguer"
>>> fnode.attrs.creation_date = '2004-10-20T13:25:25+0200'
>>> fnode.attrs.keywords_en = ["FileName", "test", "metadata"]
>>> fnode.attrs.keywords_ca = ["FileName", "prova", "metadades"]
>>> fnode.attrs.owner = 'ivan'
>>> fnode.attrs.acl = {'ivan': 'rw', '@users': 'r'}
```

You can check that these attributes get stored by running the `ptdump` command on the host PyTables file.

```
$ ptdump -a fnode.h5:/fnode_test
/fnode_test (EArray(113,)) ''
/fnode_test.attrs (AttributeSet), 14 attributes:
[CLASS := 'EARRAY',
EXTDIM := 0,
FLAVOR := 'numpy',
NODE_TYPE := 'file',
NODE_TYPE_VERSION := 2,
TITLE := '',
VERSION := '1.2',
acl := {'ivan': 'rw', '@users': 'r'},
author := 'Ivan Vilata i Balaguer',
content_type := 'text/plain; charset=us-ascii',
creation_date := '2004-10-20T13:25:25+0200',
keywords_ca := ['FileName', 'prova', 'metadades'],
keywords_en := ['FileName', 'test', 'metadata'],
owner := 'ivan']
```

Note that `filenode` makes no assumptions about the meaning of your metadata, so its handling is entirely left to your needs and imagination.

## 6.4 Complementary notes

You can use file nodes and PyTables groups to mimic a filesystem with files and directories. Since you can store nearly anything you want as file metadata, this enables you to use a PyTables file as a portable compressed backup, even between radically different platforms. Take this with a grain of salt, since node files are restricted in their naming (only valid Python identifiers are valid); however, remember that you can use node titles and metadata to overcome this limitation. Also, you may need to devise some strategy to represent special files such as devices, sockets and such (not necessarily using `filenode`).

We are eager to hear your opinion about filenode and its potential uses. Suggestions to improve filenode and create other node types are also welcome. Do not hesitate to contact us!

## 6.5 Current limitations

filenode is still a young piece of software, so it lacks some functionality. This is a list of known current limitations:

1. Node files can only be opened for read-only or read and append mode. This should be enhanced in the future.
2. Near future?
3. There is no universal newline support yet. This is likely to be implemented in a near future.
4. Sparse files (files with lots of zeros) are not treated specially; if you want them to take less space, you should be better off using compression.

These limitations still make filenode entirely adequate to work with most binary and text files. Of course, suggestions and patches are welcome.

## 6.6 filenode module reference

### 6.6.1 Global constants

`tables.nodes.filenode.NodeType`

Value for `NODE_TYPE` node system attribute.

`tables.nodes.filenode.NodeTypeVersions`

Supported values for `NODE_TYPE_VERSION` node system attribute.

### 6.6.2 Global functions

`tables.nodes.filenode.newNode(h5file, where, name, title="", filters=None, expectedsize=1000)`

Creates a new file node object in the specified PyTables file object. Additional named arguments `where` and `name` must be passed to specify where the file node is to be created. Other named arguments such as `title` and `filters` may also be passed. The special named argument `expectedsize`, indicating an estimate of the file size in bytes, may also be passed. It returns the file node object.

`tables.nodes.filenode.openNode(node, mode = 'r')`

Opens an existing file node. Returns a file node object from the existing specified PyTables node. If `mode` is not specified or it is `'r'`, the file can only be read, and the pointer is positioned at the beginning of the file. If `mode` is `'a+'`, the file can be read and appended, and the pointer is positioned at the end of the file.

### 6.6.3 The FileNode abstract class

**class** `tables.nodes.filenode.FileNode`

This is the ancestor of `ROFileNode` and `RAFileNode` (see below). Instances of these classes are returned when `newNode()` or `openNode()` are called. It represents a new file node associated with a PyTables node, providing a standard Python file interface to it.

This abstract class provides only an implementation of the reading methods needed to implement a file-like object over a PyTables node. The attribute set of the node becomes available via the `attrs` property. You can add attributes there, but try to avoid attribute names in all caps or starting with `'_'`, since they may clash with internal attributes.

The node used as storage is also made available via the read-only attribute node. Please do not tamper with this object unless unavoidably, since you may break the operation of the file node object.

The `lineSeparator` property contains the string used as a line separator, and defaults to `os.linesep`. It can be set to any reasonably-sized string you want.

The constructor sets the `closed`, `softspace` and `_lineSeparator` attributes to their initial values, as well as the `node` attribute to `None`. Sub-classes should set the `node`, `mode` and `offset` attributes.

Version 1 implements the file storage as a `UInt8` uni-dimensional `EArray`.

## FileNode methods

`FileNode.getLineSeparator()`

Returns the line separator string.

`FileNode.setLineSeparator()`

Sets the line separator string.

`FileNode.getAttrs()`

Returns the attribute set of the file node.

`FileNode.close()`

Flushes the file and closes it. The `node` attribute becomes `None` and the `attrs` property becomes no longer available.

`FileNode.next()`

Returns the next line of text. Raises `StopIteration` when lines are exhausted. See `file.next.__doc__` for more information.

`FileNode.read(size=None)`

Reads at most `size` bytes. See `file.read.__doc__` for more information

`FileNode.readline(size=-1)`

Reads the next text line. See `file.readline.__doc__` for more information.

`FileNode.readlines(sizehint=-1)`

Reads the text lines. See `file.readlines.__doc__` for more information.

`FileNode.seek(offset, whence=0)`

Moves to a new file position. See `file.seek.__doc__` for more information.

`FileNode.tell()`

Gets the current file position. See `file.tell.__doc__` for more information.

`FileNode.xreadlines()`

For backward compatibility. See `file.xreadlines.__doc__` for more information.

## 6.6.4 The ROFileNode class

**class** `tables.nodes.filenode.ROFileNode`

Instances of this class are returned when `openNode()` is called in read-only mode ('r'). This is a descendant of `FileNode` class, so it inherits all its methods. Moreover, it does not define any other useful method, just some protections against users intents to write on file.

### 6.6.5 The RAFileNode class

**class** `tables.nodes.filenode.RAFileNode`

Instances of this class are returned when either `newNode()` is called or when `openNode()` is called in append mode ('a+'). This is a descendant of `FileNode` class, so it inherits all its methods. It provides additional methods that allow to write on file nodes.

`RAFileNode.flush()`

Flushes the file node. See `file.flush.__doc__` for more information.

`RAFileNode.truncate(size=None)`

Truncates the file node to at most `size` bytes. Currently, this method only makes sense to grow the file node, since data can not be rewritten nor deleted. See `file.truncate.__doc__` for more information.

`RAFileNode.write(string)`

Writes the string to the file. Writing an empty string does nothing, but requires the file to be open. See `file.write.__doc__` for more information.

`RAFileNode.writelines(sequence)`

Writes the sequence of strings to the file. See `file.writelines.__doc__` for more information.



# NETCDF3 - A PYTABLES NETCDF3 EMULATION API (DEPRECATED)

|                                                                                                                                                                |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <p><b>Warning:</b> The <code>tables.netcdf3</code> module is not actively maintained anymore. It is deprecated and will be removed in the future versions.</p> |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------|

## 7.1 What is netcdf3?

The netCDF format is a popular format for binary files. It is portable between machines and self-describing, i.e. it contains the information necessary to interpret its contents. A free library provides convenient access to these files (see ). A very nice python interface to that library is available in the Scientific Python NetCDF module (see ). Although it is somewhat less efficient and flexible than HDF5, netCDF is geared for storing gridded data and is quite easy to use. It has become a de facto standard for gridded data, especially in meteorology and oceanography. The next version of netCDF (netCDF 4) will actually be a software layer on top of HDF5 (see ). The `tables.netcdf3` package does not create HDF5 files that are compatible with netCDF 4 (although this is a long-term goal).

## 7.2 Using the `tables.netcdf3` package

The package `tables.netcdf3` emulates the `Scientific.IO.NetCDF` API using PyTables. It presents the data in the form of objects that behave very much like arrays. A `tables.netcdf3` file contains any number of dimensions and variables, both of which have unique names. Each variable has a shape defined by a set of dimensions, and optionally attributes whose values can be numbers, number sequences, or strings. One dimension of a file can be defined as *unlimited*, meaning that the file can grow along that direction. In the sections that follow, a step-by-step tutorial shows how to create and modify a `tables.netcdf3` file. All of the code snippets presented here are included in `examples/netCDF_example.py`. The `tables.netcdf3` package is designed to be used as a drop-in replacement for `Scientific.IO.NetCDF`, with only minor modifications to existing code. The differences between `tables.netcdf3` and `Scientific.IO.NetCDF` are summarized in the last section of this chapter.

### 7.2.1 Creating/Opening/Closing a `tables.netcdf3` file

To create a `tables.netcdf3` file from python, you simply call the `NetCDFFile` constructor. This is also the method used to open an existing `tables.netcdf3` file. The object returned is an instance of the `NetCDFFile` class and all future access must be done through this object. If the file is open for write access ('w' or 'a'), you may write any type of new data including new dimensions, variables and attributes. The optional `history` keyword argument can be used to set

the history NetCDFFile global file attribute. Closing the tables.netcdf3 file is accomplished via the close method of NetCDFFile object.

Here's an example:

```
>>> import tables.netcdf3 as NetCDF
>>> import time
>>> history = 'Created ' + time.ctime(time.time())
>>> file = NetCDF.NetCDFFile('test.h5', 'w', history=history)
>>> file.close()
```

## 7.2.2 Dimensions in a tables.netcdf3 file

NetCDF defines the sizes of all variables in terms of dimensions, so before any variables can be created the dimensions they use must be created first. A dimension is created using the createDimension method of the NetCDFFile object. A Python string is used to set the name of the dimension, and an integer value is used to set the size. To create an *unlimited* dimension (a dimension that can be appended to), the size value is set to None:

```
>>> import tables.netcdf3 as NetCDF
>>> file = NetCDF.NetCDFFile('test.h5', 'a')
>>> file.createDimension('level', 12)
>>> file.createDimension('time', None)
>>> file.createDimension('lat', 90)
```

All of the dimension names and their associated sizes are stored in a Python dictionary:

```
>>> print file.dimensions
{'lat': 90, 'time': None, 'level': 12}
```

## 7.2.3 Variables in a tables.netcdf3 file

Most of the data in a tables.netcdf3 file is stored in a netCDF variable (except for global attributes). To create a netCDF variable, use the createVariable method of the NetCDFFile object. The createVariable method has three mandatory arguments, the variable name (a Python string), the variable datatype described by a single character Numeric type-code string which can be one of f (Float32), d (Float64), i (Int32), l (Int32), s (Int16), c (CharType - length 1), F (Complex32), D (Complex64) or I (Int8), and a tuple containing the variable's dimension names (defined previously with createDimension). The dimensions themselves are usually defined as variables, called coordinate variables. The createVariable method returns an instance of the NetCDFVariable class whose methods can be used later to access and set variable data and attributes:

```
>>> times = file.createVariable('time', 'd', ('time',))
>>> levels = file.createVariable('level', 'i', ('level',))
>>> latitudes = file.createVariable('latitude', 'f', ('lat',))
>>> temp = file.createVariable('temp', 'f', ('time', 'level', 'lat',))
>>> pressure = file.createVariable('pressure', 'i', ('level', 'lat',))
```

All of the variables in the file are stored in a Python dictionary, in the same way as the dimensions:

```
>>> print file.variables
{'latitude': <tables.netcdf3.NetCDFVariable instance at 0x244f350>,
 'pressure': <tables.netcdf3.NetCDFVariable instance at 0x244f508>,
 'level': <tables.netcdf3.NetCDFVariable instance at 0x244f0d0>,
 'temp': <tables.netcdf3.NetCDFVariable instance at 0x244f3a0>,
 'time': <tables.netcdf3.NetCDFVariable instance at 0x2564c88>}
```

## 7.2.4 Attributes in a tables.netcdf3 file

There are two types of attributes in a tables.netcdf3 file, global (or file) and variable. Global attributes provide information about the dataset, or file, as a whole. Variable attributes provide information about one of the variables in the file. Global attributes are set by assigning values to NetCDFFile instance variables. Variable attributes are set by assigning values to NetCDFVariable instance variables.

Attributes can be strings, numbers or sequences. Returning to our example:

```
>>> file.description = 'bogus example to illustrate the use of tables.netcdf3'
>>> file.source = 'PyTables Users Guide'
>>> latitudes.units = 'degrees north'
>>> pressure.units = 'hPa'
>>> temp.units = 'K'
>>> times.units = 'days since January 1, 2005'
>>> times.scale_factor = 1
```

The ncattrs method of the NetCDFFile object can be used to retrieve the names of all the global attributes. This method is provided as a convenience, since using the built-in dir Python function will return a bunch of private methods and attributes that cannot (or should not) be modified by the user. Similarly, the ncattrs method of a NetCDFVariable object returns all of the netCDF variable attribute names. These functions can be used to easily print all of the attributes currently defined, like this:

```
>>> for name in file.ncattrs():
...     print 'Global attr', name, '=', getattr(file,name)
Global attr description = bogus example to illustrate the use of tables.netcdf3
Global attr history = Created Mon Nov 7 10:30:56 2005
Global attr source = PyTables Users Guide
```

Note that the ncattrs function is not part of the Scientific.IO.NetCDF interface.

## 7.2.5 Writing data to and retrieving data from a tables.netcdf3 variable

Now that you have a netCDF variable object, how do you put data into it? If the variable has no *unlimited* dimension, you just treat it like a Numeric array object and assign data to a slice:

```
>>> import numpy
>>> levels[:] = numpy.arange(12)+1
>>> latitudes[:] = numpy.arange(-89,90,2)
>>> for lev in levels[:]:
>>>     pressure[:, :] = 1000.-100.*lev
>>> print 'levels = ', levels[:]
levels = [ 1  2  3  4  5  6  7  8  9 10 11 12]
>>> print 'latitudes =\n', latitudes[:]
latitudes =
[-89. -87. -85. -83. -81. -79. -77. -75. -73. -71. -69. -67. -65. -63.
-61. -59. -57. -55. -53. -51. -49. -47. -45. -43. -41. -39. -37. -35.
-33. -31. -29. -27. -25. -23. -21. -19. -17. -15. -13. -11. -9. -7.
-5. -3. -1. 1. 3. 5. 7. 9. 11. 13. 15. 17. 19. 21.
23. 25. 27. 29. 31. 33. 35. 37. 39. 41. 43. 45. 47. 49.
51. 53. 55. 57. 59. 61. 63. 65. 67. 69. 71. 73. 75. 77.
79. 81. 83. 85. 87. 89.]
```

Note that retrieving data from the netCDF variable object works just like a Numeric array too. If the netCDF variable has an *unlimited* dimension, and there is not yet an entry for the data along that dimension, the append method must be used:

```
>>> for n in range(10):
>>>     times.append(n)
>>> print 'times = ', times[:]
times = [ 0.  1.  2.  3.  4.  5.  6.  7.  8.  9.]
```

The data you append must have either the same number of dimensions as the NetCDFVariable, or one less. The shape of the data you append must be the same as the NetCDFVariable for all of the dimensions except the *unlimited* dimension. The length of the data along the *unlimited* dimension controls how many entries along the *unlimited* dimension are appended. If the data you append has one fewer number of dimensions than the NetCDFVariable, it is assumed that you are appending one entry along the *unlimited* dimension. For example, if the NetCDFVariable has shape (10,50,100) (where the dimension length of length 10 is the *unlimited* dimension), and you append an array of shape (50,100), the NetCDFVariable will subsequently have a shape of (11,50,100). If you append an array with shape (5,50,100), the NetCDFVariable will have a new shape of (15,50,100). Appending an array whose last two dimensions do not have a shape (50,100) will raise an exception. This append method does not exist in the Scientific.IO.NetCDF interface, instead entries are appended along the *unlimited* dimension one at a time by assigning to a slice. This is the biggest difference between the tables.netcdf3 and Scientific.IO.NetCDF interfaces.

Once data has been appended to any variable with an *unlimited* dimension, the sync method can be used to synchronize the sizes of all the other variables with an *unlimited* dimension. This is done by filling in missing values (given by the default netCDF\_FillValue, which is intended to indicate that the data was never defined). The sync method is automatically invoked with a NetCDFFile object is closed. Once the sync method has been invoked, the filled-in values can be assigned real data with slices:

```
>>> print 'temp.shape before sync = ', temp.shape
temp.shape before sync = (0, 12, 90)
>>> file.sync()
>>> print 'temp.shape after sync = ', temp.shape
temp.shape after sync = (10, 12, 90)
>>> from numpy import random_array
>>> for n in range(10):
...     temp[n] = 10.*random_array.random(pressure.shape)
...     print 'time, min/max temp, temp[n,0,0] = ', \
times[n], min(temp[n].flat), max(temp[n].flat), temp[n,0,0]
time, min/max temp, temp[n,0,0] = 0.0 0.0122650898993 9.99259281158 6.13053750992
time, min/max temp, temp[n,0,0] = 1.0 0.00115821603686 9.9915933609 6.68516159058
time, min/max temp, temp[n,0,0] = 2.0 0.0152112031356 9.98737239838 3.60537290573
time, min/max temp, temp[n,0,0] = 3.0 0.0112022599205 9.99535560608 6.24249696732
time, min/max temp, temp[n,0,0] = 4.0 0.00519315246493 9.99831295013 0.225010097027
time, min/max temp, temp[n,0,0] = 5.0 0.00978941563517 9.9843454361 4.56814193726
time, min/max temp, temp[n,0,0] = 6.0 0.0159023851156 9.99160385132 6.36837291718
time, min/max temp, temp[n,0,0] = 7.0 0.0019518379122 9.99939727783 1.42762875557
time, min/max temp, temp[n,0,0] = 8.0 0.00390585977584 9.9909954071 2.79601073265
time, min/max temp, temp[n,0,0] = 9.0 0.0106026884168 9.99195957184 8.18835449219
```

Note that appending data along an *unlimited* dimension always increases the length of the variable along that dimension. Assigning data to a variable with an *unlimited* dimension with a slice operation does not change its shape. Finally, before closing the file we can get a summary of its contents simply by printing the NetCDFFile object. This produces output very similar to running 'ncdump -h' on a netCDF file:

```
>>> print file
test.h5 {
dimensions:
lat = 90 ;
time = UNLIMITED ; // (10 currently)
level = 12 ;
variables:
float latitude('lat',) ;
latitude:units = 'degrees north' ;
```

```

int pressure('level', 'lat') ;
pressure:units = 'hPa' ;
int level('level',) ;
float temp('time', 'level', 'lat') ;
temp:units = 'K' ;
double time('time',) ;
time:scale_factor = 1 ;
time:units = 'days since January 1, 2005' ;
// global attributes:
:description = 'bogus example to illustrate the use of tables.netcdf3' ;
:history = 'Created Wed Nov 9 12:29:13 2005' ;
:source = 'PyTables Users Guide' ;
}

```

## 7.2.6 Efficient compression of tables.netcdf3 variables

Data stored in NetCDFVariable objects is compressed on disk by default. The parameters for the default compression are determined from a Filters class instance (see section *The Filters class*) with complevel=6, complib='zlib' and shuffle=True. To change the default compression, simply pass a Filters instance to createVariable with the filters keyword. If your data only has a certain number of digits of precision (say for example, it is temperature data that was measured with a precision of 0.1 degrees), you can dramatically improve compression by quantizing (or truncating) the data using the least\_significant\_digit keyword argument to createVariable. The *least significant digit* is the power of ten of the smallest decimal place in the data that is a reliable value. For example if the data has a precision of 0.1, then setting least\_significant\_digit=1 will cause data the data to be quantized using `numpy.around(scale*data)/scale`, where `scale = 2**bits`, and bits is determined so that a precision of 0.1 is retained (in this case bits=4).

In our example, try replacing the line:

```
>>> temp = file.createVariable('temp','f',('time','level','lat',))
```

with:

```
>>> temp = file.createVariable('temp','f',('time','level','lat',), least_significant_digit=1)
```

and see how much smaller the resulting file is.

The least\_significant\_digit keyword argument is not allowed in Scientific.IO.NetCDF, since netCDF version 3 does not support compression. The flexible, fast and efficient compression available in HDF5 is the main reason I wrote the tables.netcdf3 package - my netCDF files were just getting too big.

The createVariable method has one other keyword argument not found in Scientific.IO.NetCDF - expectedsize. The expectedsize keyword can be used to set the expected number of entries along the *unlimited* dimension (default 10000). If you expect that your data will have an order of magnitude more or less than 10000 entries along the *unlimited* dimension, you may consider setting this keyword to improve efficiency (see *Informing PyTables about expected number of rows in tables or arrays* for details).

## 7.3 tables.netcdf3 package reference

Deprecated since version 2.3.

### 7.3.1 Global constants

`tables.netcdf3._fillvalue_dict`

Dictionary whose keys are NetCDFVariable single character typecodes and whose values are the netCDF \_Fill-

Value for that typecode.

`tables.netcdf3.ScientificIONetCDF_imported`  
True if Scientific.IO.NetCDF is installed and can be imported.

### 7.3.2 The NetCDFFile class

**class** `tables.netcdf3.NetCDFFile` (*filename, mode='r', history=None*)

Opens an existing `tables.netcdf3` file (`mode = 'r'` or `'a'`) or creates a new one (`mode = 'w'`). The `history` keyword can be used to set the `NetCDFFile.history` global attribute (if `mode = 'a'` or `'w'`).

A `NetCDFFile` object has two standard attributes: `dimensions` and `variables`. The values of both are dictionaries, mapping dimension names to their associated lengths and variable names to variables. All other attributes correspond to global attributes defined in a netCDF file. Global file attributes are created by assigning to an attribute of the `NetCDFFile` object.

#### NetCDFFile methods

`NetCDFFile.close()`

Closes the file (after invoking the `sync` method).

`NetCDFFile.sync()`

Synchronizes the size of variables along the *unlimited* dimension, by filling in data with default `netCDF_FillValue`. Returns the length of the *unlimited* dimension. Invoked automatically when the `NetCDFFile` object is closed.

`NetCDFFile.ncattrs()`

Returns a list with the names of all currently defined netCDF global file attributes.

`NetCDFFile.createDimension` (*name, length*)

Creates a netCDF dimension with a name given by the Python string `name` and a size given by the integer `size`. If `size = None`, the dimension is *unlimited* (i.e. it can grow dynamically). There can be only one *unlimited* dimension in a file.

`NetCDFFile.createVariable` (*name, type, dimensions, least\_significant\_digit= None, expected-size=10000, filters=None*)

Creates a new variable with the given `name`, `type`, and `dimensions`. The `type` is a one-letter Numeric typecode string which can be one of `f` (`Float32`), `d` (`Float64`), `i` (`Int32`), `l` (`Int32`), `s` (`Int16`), `c` (`CharType - length 1`), `F` (`Complex32`), `D` (`Complex64`) or `l` (`Int8`); the predefined type constants from `Numeric` can also be used. The `F` and `D` types are not supported in netCDF or `Scientific.IO.NetCDF`, if they are used in a `tables.netcdf3` file, that file cannot be converted to a true netCDF file nor can it be shared over the Internet with `OPeNDAP`. Dimensions must be a tuple containing dimension names (strings) that have been defined previously by `createDimensions`. The `least_significant_digit` is the power of ten of the smallest decimal place in the variable's data that is a reliable value. If this keyword is specified, the variable's data truncated to this precision to improve compression. The `expectedsize` keyword can be used to set the expected number of entries along the *unlimited* dimension (default 10000). If you expect that your data with have an order of magnitude more or less than 10000 entries along the *unlimited* dimension, you may consider setting this keyword to improve efficiency (see [Informing PyTables about expected number of rows in tables or arrays](#) for details). The `filters` keyword is a `PyTables Filters` instance that describes how to store the data on disk. The default corresponds to `complevel=6`, `complib='zlib'`, `shuffle=True` and `fletcher32=False`.

`NetCDFFile.nctoh5` (*filename, unpackshort=True, filters=None*)

Imports the data in a netCDF version 3 file (`filename`) into a `NetCDFFile` object using `Scientific.IO.NetCDF` (`ScientificIONetCDF_imported` must be `True`). If `unpackshort=True`, data packed as short integers (type `s`) in the netCDF file will be unpacked to type `f` using the `scale_factor` and `add_offset` netCDF variable attributes. The `filters` keyword can be set to a `PyTables Filters` instance to change the default parameters used to compress

the data in the `tables.netcdf3` file. The default corresponds to `complevel=6`, `complib='zlib'`, `shuffle=True` and `fletcher32=False`.

`NetCDFFile.h5tonc(filename, packshort=False, scale_factor=None, add_offset=None)`

Exports the data in a `tables.netcdf3` file defined by the `NetCDFFile` instance into a netCDF version 3 file using `Scientific.IO.NetCDF` (`ScientificIONetCDF_imported` must be `True`). If `packshort=True` the dictionaries `scale_factor` and `add_offset` are used to pack data of type `f` as short integers (of types) in the netCDF file. Since netCDF version 3 does not provide automatic compression, packing as short integers is a commonly used way of saving disk space (see this [page](#) for more details). The keys of these dictionaries are the variable names to pack, the values are the `scale_factors` and offsets to use in the packing. The data are packed so that the original Float32 values can be reconstructed by multiplying the `scale_factor` and adding `add_offset`. The resulting netCDF file will have the `scale_factor` and `add_offset` variable attributes set appropriately.

### 7.3.3 The `NetCDFVariable` class

**class** `tables.netcdf3.NetCDFVariable`

The `NetCDFVariable` constructor is not called explicitly, rather an `NetCDFVariable` instance is returned by an invocation of `NetCDFFile.createVariable`. `NetCDFVariable` objects behave like arrays, and have the standard attributes of arrays (such as `shape`). Data can be assigned or extracted from `NetCDFVariable` objects via slices.

#### NetCDFVariable methods

`NetCDFVariable.typecode()`

Returns a single character typecode describing the type of the variable, one of `f` (Float32), `d` (Float64), `i` (Int32), `l` (Int32), `s` (Int16), `c` (CharType - length 1), `F` (Complex32), `D` (Complex64) or `l` (Int8).

`NetCDFVariable.append(data)`

Append data to a variable along its *unlimited* dimension. The data you append must have either the same number of dimensions as the `NetCDFVariable`, or one less. The shape of the data you append must be the same as the `NetCDFVariable` for all of the dimensions except the *unlimited* dimension. The length of the data long the *unlimited* dimension controls how many entries along the *unlimited* dimension are appended. If the data you append has one fewer number of dimensions than the `NetCDFVariable`, it is assumed that you are appending one entry along the *unlimited* dimension. For variables without an *unlimited* dimension, data can simply be assigned to a slice without using the append method.

`NetCDFVariable.ncattrs()`

Returns a list with all the names of the currently defined netCDF variable attributes.

`NetCDFVariable.assignValue(data)`

Provided for compatibility with `Scientific.IO.NetCDF`. Assigns data to the variable. If the variable has an *unlimited* dimension, it is equivalent to `append(data)`. If the variable has no *unlimited* dimension, it is equivalent to assigning data to the variable with the slice `[:]`.

`NetCDFVariable.getValue()`

Provided for compatibility with `Scientific.IO.NetCDF`. Returns all the data in the variable. Equivalent to extracting the slice `[:]` from the variable.

## 7.4 Converting between true netCDF files and `tables.netcdf3` files

If `Scientific.IO.NetCDF` is installed, `tables.netcdf3` provides facilities for converting between true netCDF version 3 files and `tables.netcdf3` hdf5 files via the `NetCDFFile.h5tonc()` and `NetCDFFile.nctoh5()` class methods. Also, the `nctoh5` command-line utility (see [nctoh5](#)) uses the `NetCDFFile.nctoh5()` class method.

As an example, look how to convert a `tables.netcdf3` hdf5 file to a true netCDF version 3 file (named `test.nc`):

```
>>> scale_factor = {'temp': 1.75e-4}
>>> add_offset = {'temp': 5.}
>>> file.h5tonc('test.nc', packshort=True, \
scale_factor=scale_factor, add_offset=add_offset)
packing temp as short integers ...
>>> file.close()
```

The dictionaries `scale_factor` and `add_offset` are used to optionally pack the data as short integers in the netCDF file. Since netCDF version 3 does not provide automatic compression, packing as short integers is a commonly used way of saving disk space (see this [page](#) for more details). The keys of these dictionaries are the variable names to pack, the values are the scale\_factors and offsets to use in the packing. The resulting netCDF file will have the `scale_factor` and `add_offset` variable attributes set appropriately.

To convert the netCDF file back to a `tables.netcdf3` hdf5 file:

```
>>> history = 'Convert from netCDF ' + time.ctime(time.time())
>>> file = NetCDF.NetCDFFile('test2.h5', 'w', history=history)
>>> nobjects, nbytes = file.nctoh5('test.nc', unpackshort=True)
>>> print nobjects, ' objects converted from netCDF, totaling', nbytes, 'bytes'
5 objects converted from netCDF, totaling 48008 bytes
>>> temp = file.variables['temp']
>>> times = file.variables['time']
>>> print 'temp.shape after h5 --> netCDF --> h5 conversion = ', temp.shape
temp.shape after h5 --> netCDF --> h5 conversion = (10, 12, 90)
>>> for n in range(10):
...     print 'time, min/max temp, temp[n,0,0] = ', \
times[n], min(temp[n].flat), max(temp[n].flat), temp[n,0,0]
time, min/max temp, temp[n,0,0] = 0.0 0.0123250000179 9.99257469177 6.13049983978
time, min/max temp, temp[n,0,0] = 1.0 0.001300000000354 9.99152469635 6.68507480621
time, min/max temp, temp[n,0,0] = 2.0 0.01530000000864 9.98732471466 3.60542488098
time, min/max temp, temp[n,0,0] = 3.0 0.0112749999389 9.99520015717 6.2423248291
time, min/max temp, temp[n,0,0] = 4.0 0.00532499980181 9.99817466736 0.225124999881
time, min/max temp, temp[n,0,0] = 5.0 0.00987500045449 9.98417472839 4.56827497482
time, min/max temp, temp[n,0,0] = 6.0 0.016000000076 9.99152469635 6.36832523346
time, min/max temp, temp[n,0,0] = 7.0 0.00200000009499 9.99922466278 1.42772495747
time, min/max temp, temp[n,0,0] = 8.0 0.00392499985173 9.9908246994 2.79605007172
time, min/max temp, temp[n,0,0] = 9.0 0.0107500003651 9.99187469482 8.18832492828
>>> file.close()
```

Setting `unpackshort=True` tells `nctoh5` to unpack all of the variables which have the `scale_factor` and `add_offset` attributes back to floating point arrays. Note that `tables.netcdf3` files have some features not supported in netCDF (such as Complex data types and the ability to make any dimension *unlimited*). `tables.netcdf3` files which utilize these features cannot be converted to netCDF using `NetCDFFile.h5tonc`.

## 7.5 tables.netcdf3 file structure

A `tables.netcdf3` file consists of array objects (either EArrays or CArrays) located in the root group of a pytables hdf5 file. Each of the array objects must have a `dimensions` attribute, consisting of a tuple of dimension names (the length of this tuple should be the same as the rank of the array object). Any array objects with one of the supported datatypes in a pytables file that conforms to this simple structure can be read with the `tables.netcdf3` package.

## 7.6 Sharing data in tables.netcdf3 files over the Internet with OPeNDAP

tables.netcdf3 datasets can be shared over the Internet with the OPeNDAP protocol (<http://opendap.org>), via the python OPeNDAP module. A plugin for the python opendap server is included with the pytables distribution (contrib/h5\_dap\_plugin.py). Simply copy that file into the plugins directory of the opendap python module source distribution, run python setup.py install, point the opendap server to the directory containing your tables.netcdf3 files, and away you go. Any OPeNDAP aware client (such as Matlab or IDL) will now be able to access your data over http as if it were a local disk file. The only restriction is that your tables.netcdf3 files must have the extension .h5 or .hdf5. Unfortunately, tables.netcdf3 itself cannot act as an OPeNDAP client, although there is a client included in the opendap python module, and Scientific.IO.NetCDF can act as an OPeNDAP client if it is linked with the OPeNDAP netCDF client library. Either of these python modules can be used to remotely access tables.netcdf3 datasets with OPeNDAP.

## 7.7 Differences between the Scientific.IO.NetCDF API and the tables.netcdf3 API

1. tables.netcdf3 data is stored in an HDF5 file instead of a netCDF file.
2. Although each variable can have only one *unlimited* dimension in a tables.netcdf3 file, it need not be the first as in a true NetCDF file. Complex data types F (Complex32) and D (Complex64) are supported in tables.netcdf3, but are not supported in netCDF (or Scientific.IO.NetCDF). Files with variables that have these datatypes, or an *unlimited* dimension other than the first, cannot be converted to netCDF using h5tonc.
3. Variables in a tables.netcdf3 file are compressed on disk by default using HDF5 zlib compression with the *shuffle* filter. If the *least\_significant\_digit* keyword is used when a variable is created with the createVariable method, data will be truncated (quantized) before being written to the file. This can significantly improve compression. For example, if least\_significant\_digit=1, data will be quantized using `numpy.around(scale*data)/scale`, where `scale = 2**bits`, and bits is determined so that a precision of 0.1 is retained (in this case bits=4). From [http://www.esrl.noaa.gov/psd/data/gridded/conventions/cdc\\_netcdf\\_standard.shtml](http://www.esrl.noaa.gov/psd/data/gridded/conventions/cdc_netcdf_standard.shtml):

```
"least_significant_digit -- power of ten of the smallest
decimal place in unpacked data that is a reliable value."
```

Automatic data compression is not available in netCDF version 3, and hence is not available in the Scientific.IO.NetCDF module.

4. In tables.netcdf3, data must be appended to a variable with an *unlimited* dimension using the append method of the netCDF variable object. In Scientific.IO.NetCDF, data can be added along an *unlimited* dimension by assigning it to a slice (there is no append method). The sync method of a tables.netcdf3 NetCDFVariable object synchronizes the size of all variables with an *unlimited* dimension by filling in data using the default netCDF \_FillValue. The sync method is automatically invoked with a NetCDFFile object is closed. In Scientific.IO.NetCDF, the sync() method flushes the data to disk.
5. The tables.netcdf3 createVariable() method has three extra optional keyword arguments not found in the Scientific.IO.NetCDF interface, *least\_significant\_digit* (see item (2) above), *expectedsize* and *filters*. The *expectedsize* keyword applies only to variables with an *unlimited* dimension, and is an estimate of the number of entries that will be added along that dimension (default 1000). This estimate is used to optimize HDF5 file access and memory usage. The *filters* keyword is a PyTables filters instance that describes how to store the data on disk. The default corresponds to `complevel=6`, `complib='zlib'`, `shuffle=True` and `fletcher32=False`.
6. tables.netcdf3 data can be saved to a true netCDF file using the NetCDFFile class method h5tonc (if Scientific.IO.NetCDF is installed). The *unlimited* dimension must be the first (for all variables in the file) in order to

use the `h5tonc` method. Data can also be imported from a true netCDF file and saved in an HDF5 `tables.netcdf3` file using the `nctoh5` class method.

7. In `tables.netcdf3` a list of attributes corresponding to global netCDF attributes defined in the file can be obtained with the `NetCDFFile ncattrs` method. Similarly, netCDF variable attributes can be obtained with the `NetCDFVariable ncattrs` method. These functions are not available in the `Scientific.IO.NetCDF` API.
8. You should not define `tables.netcdf3` global or variable attributes that start with `_NetCDF_`. Those names are reserved for internal use.
9. Output similar to `'ncdump -h'` can be obtained by simply printing a `tables.netcdf3` `NetCDFFile` instance.

## **Part III**

# **Appendixes**



# SUPPORTED DATA TYPES IN PYTABLES

All PyTables datasets can handle the complete set of data types supported by the NumPy (see [\[NUMPY\]](#)), numarray (see [\[NUMARRAY\]](#)) and Numeric (see [\[NUMERIC\]](#)) packages in Python. The data types for table fields can be set via instances of the Col class and its descendants (see [The Col class and its descendants](#)), while the data type of array elements can be set through the use of the Atom class and its descendants (see [The Atom class and its descendants](#)).

**Warning:** The use of numarray and Numeric in PyTables is now deprecated. Support for these packages will be removed in future versions.

PyTables uses ordinary strings to represent its *types*, with most of them matching the names of NumPy scalar types. Usually, a PyTables type consists of two parts: a *kind* and a *precision* in bits. The precision may be omitted in types with just one supported precision (like bool) or with a non-fixed size (like string).

There are eight kinds of types supported by PyTables:

- bool: Boolean (true/false) types. Supported precisions: 8 (default) bits.
- int: Signed integer types. Supported precisions: 8, 16, 32 (default) and 64 bits.
- uint: Unsigned integer types. Supported precisions: 8, 16, 32 (default) and 64 bits.
- float: Floating point types. Supported precisions: 16, 32 and 64 (default) bits (see [note on floating point types](#)).
- complex: Complex number types. Supported precisions: 64 (32+32) and 128 (64+64, default) bits.
- string: Raw string types. Supported precisions: 8-bit positive multiples.
- time: Data/time types. Supported precisions: 32 and 64 (default) bits.
- enum: Enumerated types. Precision depends on base type.

---

**Note:** Floating point types.

The half precision floating point data type (float16) is only available if `numpy` supports it on the host platform. Also it is required `numpy >= 1.6.0`.

---

The time and enum kinds are a little bit special, since they represent HDF5 types which have no direct Python counterpart, though atoms of these kinds have a more-or-less equivalent NumPy data type.

There are two types of time: 4-byte signed integer (time32) and 8-byte double precision floating point (time64). Both of them reflect the number of seconds since the Unix epoch, i.e. Jan 1 00:00:00 UTC 1970. They are stored in memory as NumPy's int32 and float64, respectively, and in the HDF5 file using the H5T\_TIME class. Integer times

are stored on disk as such, while floating point times are split into two signed integer values representing seconds and microseconds (beware: smaller decimals will be lost!).

PyTables also supports HDF5 H5T\_ENUM *enumerations* (restricted sets of unique name and unique value pairs). The NumPy representation of an enumerated value (an Enum, see [The Enum class](#)) depends on the concrete *base type* used to store the enumeration in the HDF5 file. Currently, only scalar integer values (both signed and unsigned) are supported in enumerations. This restriction may be lifted when HDF5 supports other kinds of enumerated values.

Here you have a quick reference to the complete set of supported data types:

Table 8.1: Data types supported for array elements and tables columns in PyTables.

| Type Code            | Description              | C Type                  | Size (in bytes) | Python Counterpart |
|----------------------|--------------------------|-------------------------|-----------------|--------------------|
| bool                 | boolean                  | unsigned char           | 1               | bool               |
| int8                 | 8-bit integer            | signed char             | 1               | int                |
| uint8                | 8-bit unsigned integer   | unsigned char           | 1               | int                |
| int16                | 16-bit integer           | short                   | 2               | int                |
| uint16               | 16-bit unsigned integer  | unsigned short          | 2               | int                |
| int32                | integer                  | int                     | 4               | int                |
| uint32               | unsigned integer         | unsigned int            | 4               | long               |
| int64                | 64-bit integer           | long long               | 8               | long               |
| uint64               | unsigned 64-bit integer  | unsigned long long      | 8               | long               |
| float16 <sup>1</sup> | half-precision float     | •                       | 2               | •                  |
| float32              | single-precision float   | float                   | 4               | float              |
| float64              | double-precision float   | double                  | 8               | float              |
| complex64            | single-precision complex | struct { float r, i; }  | 8               | complex            |
| complex128           | double-precision complex | struct { double r, i; } | 16              | complex            |
| string               | arbitrary length string  | char[]                  | •               | str                |
| time32               | integer time             | POSIX's time_t          | 4               | int                |
| time64               | floating point time      | POSIX's struct timeval  | 8               | float              |
| enum                 | enumerated value         | enum                    | •               | •                  |

---

# CONDITION SYNTAX

Conditions in PyTables are used in methods related with in-kernel and indexed searches such as `Table.where()` or `Table.readWhere()`. They are interpreted using Numexpr, a powerful package for achieving C-speed computation of array operations (see [\[NUMEXPR\]](#)).

A condition on a table is just a *string* containing a Python expression involving *at least one column*, and maybe some constants and external variables, all combined with algebraic operators and functions. The result of a valid condition is always a *boolean array* of the same length as the table, where the *i*-th element is true if the value of the expression on the *i*-th row of the table evaluates to true.

That is the reason why multidimensional fields in a table are not supported in conditions, since the truth value of each resulting multidimensional boolean value is not obvious. . Usually, a method using a condition will only consider the rows where the boolean result is true.

For instance, the condition `'sqrt(x*x + y*y) < 1'` applied on a table with `x` and `y` columns consisting of floating point numbers results in a boolean array where the *i*-th element is true if (unsurprisingly) the value of the square root of the sum of squares of `x` and `y` is less than 1. The `sqrt()` function works element-wise, the `1` constant is adequately broadcast to an array of ones of the length of the table for evaluation, and the *less than* operator makes the result a valid boolean array. A condition like `'mycolumn'` alone will not usually be valid, unless `mycolumn` is itself a column of scalar, boolean values.

In the previous conditions, `mycolumn`, `x` and `y` are examples of *variables* which are associated with columns. Methods supporting conditions do usually provide their own ways of binding variable names to columns and other values. You can read the documentation of `Table.where()` for more information on that. Also, please note that the names `None`, `True` and `False`, besides the names of functions (see below) *can not be overridden*, but you can always define other new names for the objects you intend to use.

Values in a condition may have the following types:

- 8-bit boolean (`bool`).
- 32-bit signed integer (`int`).
- 64-bit signed integer (`long`).
- 32-bit, single-precision floating point number (`float` or `float32`).
- 64-bit, double-precision floating point number (`double` or `float64`).
- 2x64-bit, double-precision complex number (`complex`).
- Raw string of bytes (`str`).

Nevertheless, if the type passed is not among the above ones, it will be silently upcasted, so you don't need to worry too much about passing supported types, except for the Unsigned 64 bits integer, that cannot be upcasted to any of the supported types.

However, the types in PyTables conditions are somewhat stricter than those of Python. For instance, the *only* valid constants for booleans are True and False, and they are *never* automatically cast to integers. The type strengthening also affects the availability of operators and functions. Beyond that, the usual type inference rules apply.

Conditions support the set of operators listed below:

- Logical operators: &, |, ~.
- Comparison operators: <, <=, ==, !=, >=, >.
- Unary arithmetic operators: -.
- Binary arithmetic operators: +, -, \*, /, \*\*, %.

Types do not support all operators. Boolean values only support logical and strict (in)equality comparison operators, while strings only support comparisons, numbers do not work with logical operators, and complex comparisons can only check for strict (in)equality. Unsupported operations (including invalid castings) raise `NotImplementedError` exceptions.

You may have noticed the special meaning of the usually bitwise operators &, | and ~. Because of the way Python handles the short-circuiting of logical operators and the truth values of their operands, conditions must use the bitwise operator equivalents instead. This is not difficult to remember, but you must be careful because bitwise operators have a *higher precedence* than logical operators. For instance, 'a and b == c' (*a is true AND b is equal to c*) is *not* equivalent to 'a & b == c' (*a AND b is equal to c*). The safest way to avoid confusions is to use *parentheses* around logical operators, like this: 'a & (b == c)'. Another effect of short-circuiting is that expressions like '0 < x < 1' will *not* work as expected; you should use '(0 < x) & (x < 1)'

All of this may be solved if Python supported overloadable boolean operators (see PEP 335) or some kind of non-shortcircuiting boolean operators (like C's &&, || and !).

You can also use the following functions in conditions:

- `where(bool, number1, number2)`: number - number1 if the bool condition is true, number2 otherwise.
- `{sin,cos,tan}(float|complex)`: float|complex - trigonometric sine, cosine or tangent.
- `{arcsin,arccos,arctan}(float|complex)`: float|complex - trigonometric inverse sine, cosine or tangent.
- `arctan2(float1, float2)`: float - trigonometric inverse tangent of float1/float2.
- `{sinh,cosh,tanh}(float|complex)`: float|complex - hyperbolic sine, cosine or tangent.
- `{arcsinh,arccosh,arctanh}(float|complex)`: float|complex - hyperbolic inverse sine, cosine or tangent.
- `{log,log10,log1p}(float|complex)`: float|complex - natural, base-10 and log(1+x) logarithms.
- `{exp,expm1}(float|complex)`: float|complex - exponential and exponential minus one.
- `sqrt(float|complex)`: float|complex - square root.
- `{abs}(float|complex)`: float|complex - absolute value.
- `{real,imag}(complex)`: float - real or imaginary part of complex.
- `complex(float, float)`: complex - complex from real and imaginary parts.

# PYTABLES PARAMETER FILES

PyTables issues warnings when certain limits are exceeded. Those limits are not intrinsic limitations of the underlying software, but rather are proactive measures to avoid large resource consumptions. The default limits should be enough for most of cases, and users should try to respect them. However, in some situations, it can be convenient to increase (or decrease) these limits.

Also, and in order to get maximum performance, PyTables implements a series of sophisticated features, like I/O buffers or different kind of caches (for nodes, chunks and other internal metadata). These features comes with a default set of parameters that ensures a decent performance in most of situations. But, as there is always a need for every case, it is handy to have the possibility to fine-tune some of these parameters.

Because of these reasons, PyTables implements a couple of ways to change the values of these parameters. All the *tunable* parameters live in the `tables/parameters.py`. The user can choose to change them in the parameter files themselves for a global and persistent change. Moreover, if he wants a finer control, he can pass any of these parameters directly to the `tables.openFile()` function, and the new parameters will only take effect in the corresponding file (the defaults will continue to be in the parameter files).

A description of all of the tunable parameters follows. As the defaults stated here may change from release to release, please check with your actual parameter files so as to know your actual default values.

**Warning:** Changing the next parameters may have a very bad effect in the resource consumption and performance of your PyTables scripts. Please be careful when touching these!

## 10.1 Tunable parameters in `parameters.py`.

### 10.1.1 Recommended maximum values

`tables.parameters.MAX_COLUMNS`

Maximum number of columns in Table objects before a PerformanceWarning is issued. This limit is somewhat arbitrary and can be increased.

`tables.parameters.MAX_NODE_ATTRS`

Maximum allowed number of attributes in a node

`tables.parameters.MAX_GROUP_WIDTH`

Maximum depth in object tree allowed.

`tables.parameters.MAX_UNDO_PATH_LENGTH`

Maximum length of paths allowed in undo/redo operations.

### 10.1.2 Cache limits

`tables.parameters.CHUNK_CACHE_NELMTS`

Number of elements for HDF5 chunk cache.

`tables.parameters.CHUNK_CACHE_PREEMPT`

Chunk preemption policy. This value should be between 0 and 1 inclusive and indicates how much chunks that have been fully read are favored for preemption. A value of zero means fully read chunks are treated no differently than other chunks (the preemption is strictly LRU) while a value of one means fully read chunks are always preempted before other chunks.

`tables.parameters.CHUNK_CACHE_SIZE`

Size (in bytes) for HDF5 chunk cache.

`tables.parameters.COND_CACHE_SLOTS`

Maximum number of conditions for table queries to be kept in memory.

`tables.parameters.METADATA_CACHE_SIZE`

Size (in bytes) of the HDF5 metadata cache.

`tables.parameters.NODE_CACHE_SLOTS`

Maximum number of unreferenced nodes to be kept in memory.

If positive, this is the number of *unreferenced* nodes to be kept in the metadata cache. Least recently used nodes are unloaded from memory when this number of loaded nodes is reached. To load a node again, simply access it as usual. Nodes referenced by user variables are not taken into account nor unloaded.

Negative value means that all the touched nodes will be kept in an internal dictionary. This is the faster way to load/retrieve nodes. However, and in order to avoid a large memory consumption, the user will be warned when the number of loaded nodes will reach the `-NODE_CACHE_SLOTS` value.

Finally, a value of zero means that any cache mechanism is disabled.

### 10.1.3 Parameters for the different internal caches

`tables.parameters.BOUNDS_MAX_SIZE`

The maximum size for bounds values cached during index lookups.

`tables.parameters.BOUNDS_MAX_SLOTS`

The maximum number of slots for the BOUNDS cache.

`tables.parameters.ITERSEQ_MAX_ELEMENTS`

The maximum number of iterator elements cached in data lookups.

`tables.parameters.ITERSEQ_MAX_SIZE`

The maximum space that will take ITERSEQ cache (in bytes).

`tables.parameters.ITERSEQ_MAX_SLOTS`

The maximum number of slots in ITERSEQ cache.

`tables.parameters.LIMBOUNDS_MAX_SIZE`

The maximum size for the query limits (for example, (lim1, lim2) in conditions like `lim1 <= col < lim2`) cached during index lookups (in bytes).

`tables.parameters.LIMBOUNDS_MAX_SLOTS`

The maximum number of slots for LIMBOUNDS cache.

`tables.parameters.TABLE_MAX_SIZE`

The maximum size for table chunks cached during index queries.

`tables.parameters.SORTED_MAX_SIZE`

The maximum size for sorted values cached during index lookups.

`tables.parameters.SORTEDLR_MAX_SIZE`

The maximum size for chunks in last row cached in index lookups (in bytes).

`tables.parameters.SORTEDLR_MAX_SLOTS`

The maximum number of chunks for SORTEDLR cache.

### 10.1.4 Parameters for general cache behaviour

**Warning:** The next parameters will not take any effect if passed to the `openFile()` function, so they can only be changed in a *global* way. You can change them in the file, but this is strongly discouraged unless you know well what you are doing.

`tables.parameters.DISABLE_EVERY_CYCLES`

The number of cycles in which a cache will be forced to be disabled if the hit ratio is lower than the `LOWEST_HIT_RATIO` (see below). This value should provide time enough to check whether the cache is being efficient or not.

`tables.parameters.ENABLE_EVERY_CYCLES`

The number of cycles in which a cache will be forced to be (re-)enabled, irregardless of the hit ratio. This will provide a chance for checking if we are in a better scenario for doing caching again.

`tables.parameters.LOWEST_HIT_RATIO`

The minimum acceptable hit ratio for a cache to avoid disabling (and freeing) it.

### 10.1.5 Parameters for the I/O buffer in Leaf objects

`tables.parameters.IO_BUFFER_SIZE`

The PyTables internal buffer size for I/O purposes. Should not exceed the amount of highest level cache size in your CPU.

`tables.parameters.BUFFER_TIMES`

The maximum buffersize/rowsize ratio before issuing a PerformanceWarning.

### 10.1.6 Miscellaneous

`tables.parameters.EXPECTED_ROWS_EARRAY`

Default expected number of rows for EArray objects.

`tables.parameters.EXPECTED_ROWS_TABLE`

Default expected number of rows for Table objects.

`tables.parameters.PYTABLES_SYS_ATTRS`

Set this to False if you don't want to create PyTables system attributes in datasets. Also, if set to False the possible existing system attributes are not considered for guessing the class of the node during its loading from disk (this work is delegated to the PyTables' class discoverer function for general HDF5 files).

`tables.parameters.MAX_THREADS`

The maximum number of threads that PyTables should use internally (mainly in Blosc and Numexpr currently). If None, it is automatically set to the number of cores in your machine. In general, it is a good idea to set this to the number of cores in your machine or, when your machine has many of them (e.g. > 4), perhaps one less than this.

---

**Note:** currently `MAX_THREADS` is only used as a fall-back if `tables.parameters.MAX_NUMEXPR_THREADS` or `tables.parameters.MAX_BLOSC_THREADS` are not set.

---

Deprecated since version 2.4.

`tables.parameters`.**MAX\_NUMEXPR\_THREADS**

The maximum number of threads that PyTables should use internally in Numexpr. If None, it is automatically set to the number of cores in your machine. In general, it is a good idea to set this to the number of cores in your machine or, when your machine has many of them (e.g. > 4), perhaps one less than this.

`tables.parameters`.**MAX\_BLOSC\_THREADS**

The maximum number of threads that PyTables should use internally in Blosc. If None, it is automatically set to the number of cores in your machine. In general, it is a good idea to set this to the number of cores in your machine or, when your machine has many of them (e.g. > 4), perhaps one less than this.

# USING NESTED RECORD ARRAYS (DEPRECATED)

## 11.1 Introduction

Nested record arrays are a generalization of the record array concept as it appears in the `numarray` package. Basically, a nested record array is a record array that supports nested datatypes. It means that columns can contain not only regular datatypes but also nested datatypes.

**Warning:** PyTables nested record arrays were implemented to overcome a limitation of the record arrays in the `numarray` package. However, as this functionality is already present in NumPy, current users should not need the package `tables.nra` anymore and it will be deprecated soon.

Each nested record array is a `NestedRecArray` object in the `tables.nra` package. Nested record arrays are intended to be as compatible as possible with ordinary record arrays (in fact the `NestedRecArray` class inherits from `RecArray`). As a consequence, the user can deal with nested record arrays nearly in the same way that he does with ordinary record arrays.

The easiest way to create a nested record array is to use the `array()` function in the `tables.nra` package. The only difference between this function and its non-nested capable analogous is that now, we *must* provide an structure for the buffer being stored. For instance:

```
>>> from tables.nra import array
>>> nra1 = array(
...     [(1, (0.5, 1.0), ('a1', 1j)), (2, (0, 0), ('a2', 1+.1j))],
...     formats=['Int64', ' (2,)Float32', ['a2', 'Complex64']])
```

will create a two rows nested record array with two regular fields (columns), and one nested field with two sub-fields.

The field structure of the nested record array is specified by the keyword argument `formats`. This argument only supports sequences of strings and other sequences. Each string defines the shape and type of a non-nested field. Each sequence contains the formats of the sub-fields of a nested field. Optionally, we can also pass an additional `names` keyword argument containing the names of fields and sub-fields:

```
>>> nra2 = array(
...     [(1, (0.5, 1.0), ('a1', 1j)), (2, (0, 0), ('a2', 1+.1j))],
...     names=['id', 'pos', ('info', ['name', 'value'])],
...     formats=['Int64', ' (2,)Float32', ['a2', 'Complex64']])
```

The `names` argument only supports lists of strings and 2-tuples. Each string defines the name of a non-nested field. Each 2-tuple contains the name of a nested field and a list describing the names of its sub-fields. If the `names` argument

is not passed then all fields are automatically named (c1, c2 etc. on each nested field) so, in our first example, the fields will be named as ['c1', 'c2', ('c3', ['c1', 'c2'])].

Another way to specify the nested record array structure is to use the descr keyword argument:

```
>>> nra3 = array(
...     [(1, (0.5, 1.0), ('a1', 1j)), (2, (0, 0), ('a2', 1+.1j))],
...     descr=[('id', 'Int64'), ('pos', '(2,)Float32'),
...            ('info', [('name', 'a2'), ('value', 'Complex64')])])
>>>
>>> nra3
array(
[(1L, array([ 0.5,  1. ], type=Float32), ('a1', 1j)),
 (2L, array([ 0.,  0.], type=Float32), ('a2', (1+0.10000000000000001j)))]
descr=[('id', 'Int64'), ('pos', '(2,)Float32'), ('info', [('name', 'a2'),
 ('value', 'Complex64')])],
shape=2)
>>>
```

The descr argument is a list of 2-tuples, each of them describing a field. The first value in a tuple is the name of the field, while the second one is a description of its structure. If the second value is a string, it defines the format (shape and type) of a non-nested field. Else, it is a list of 2-tuples describing the sub-fields of a nested field.

As you can see, the descr list is a mix of the names and formats arguments. In fact, this argument is intended to replace formats and names, so they cannot be used at the same time.

Of course the structure of all three keyword arguments must match that of the elements (rows) in the buffer being stored.

Sometimes it is convenient to create nested arrays by processing a set of columns. In these cases the function fromarrays comes handy. This function works in a very similar way to the array function, but the passed buffer is a list of columns. For instance:

```
>>> from tables.nra import fromarrays
>>> nra = fromarrays([[1, 2], [4, 5]], descr=[('x', 'f8'), ('y', 'f4')])
>>>
>>> nra
array(
[(1.0,  4.0),
 (2.0,  5.0)],
descr=[('x', 'f8'), ('y', 'f4')],
shape=2)
```

Columns can be passed as nested arrays, what makes really straightforward to combine different nested arrays to get a new one, as you can see in the following examples:

```
>>> nra1 = fromarrays([nra, [7, 8]], descr=[('2D', [('x', 'f8'), ('y', 'f4')]),
... ('z', 'f4')])
>>>
>>> nra1
array(
[((1.0,  4.0), 7.0),
 ((2.0,  5.0), 8.0)],
descr=[('2D', [('x', 'f8'), ('y', 'f4')]), ('z', 'f4')],
shape=2)
>>>
>>> nra2 = fromarrays([nra1.field('2D/x'), nra1.field('z')], descr=[('x', 'f8'),
 ('z', 'f4')])
>>>
>>> nra2
```

```
array(
  [(1.0, 7.0),
   (2.0, 8.0)],
  descr=[('x', 'f8'), ('z', 'f4')],
  shape=2)
```

Finally it's worth to mention a small group of utility functions in the `tables.nra.nestedrecords` module, `makeFormats`, `makeNames` and `makeDescr`, that can be useful to obtain the structure specification to be used with the `array` and `fromarrays` functions. Given a description list, `makeFormats` gets the corresponding formats list. In the same way `makeNames` gets the names list. On the other hand the `descr` list can be obtained from formats and names lists using the `makeDescr` function. For example:

```
>>> from tables.nra.nestedrecords import makeDescr, makeFormats, makeNames
>>> descr = [('2D', [(('x', 'f8'), ('y', 'f4'))], ('z', 'f4'))]
>>>
>>> formats = makeFormats(descr)
>>> formats
[('f8', 'f4'), 'f4']
>>> names = makeNames(descr)
>>> names
[('2D', [('x', 'y')], 'z')]
>>> d1 = makeDescr(formats, names)
>>> d1
[('2D', [(('x', 'f8'), ('y', 'f4'))], ('z', 'f4'))]
>>> # If no names are passed then they are automatically generated
>>> d2 = makeDescr(formats)
>>> d2
[('c1', [(('c1', 'f8'), ('c2', 'f4'))], ('c2', 'f4'))]
```

## 11.2 NestedRecArray methods

To access the fields in the nested record array use the `field()` method:

```
>>> print nra2.field('id')
[1, 2]
>>>
```

The `field()` method accepts also names of sub-fields. It will consist of several field name components separated by the string `'/'`

This way of specifying the names of sub-fields is *very* specific to the implementation of `numarray` nested arrays of `PyTables`. Particularly, if you are using `NumPy` arrays, keep in mind that sub-fields in such arrays must be accessed one at a time, like this: `numpy_array['info']['name']`, and not like this: `numpy_array['info/name']`., for instance:

```
>>> print nra2.field('info/name')
['a1', 'a2']
>>>
```

Finally, the top level fields of the nested recarray can be accessed passing an integer argument to the `field()` method:

```
>>> print nra2.field(1)
[[ 0.5 1. ] [ 0.  0. ]]
>>>
```

An alternative to the `field()` method is the use of the `fields` attribute. It is intended mainly for interactive usage in the Python console. For example:

```
>>> nra2.fields.id
[1, 2]
>>> nra2.fields.info.fields.name
['a1', 'a2']
>>>
```

Rows of nested recarrays can be read using the typical index syntax. The rows are retrieved as `NestedRecord` objects:

```
>>> print nra2[0]
(1L, array([ 0.5,  1. ], type=Float32), ('a1', 1j))
>>>
>>> nra2[0].__class__
<class tables.nra.nestedrecords.NestedRecord at 0x413cbb9c>
```

Slicing is also supported in the usual way:

```
>>> print nra2[0:2]
NestedRecArray[
(1L, array([ 0.5,  1. ], type=Float32), ('a1', 1j)),
(2L, array([ 0.,  0.], type=Float32), ('a2', (1+0.10000000000000001j)))
]
>>>
```

Another useful method is `asRecArray()`. It converts a nested array to a non-nested equivalent array.

This method creates a new vanilla `RecArray` instance equivalent to this one by flattening its fields. Only bottom-level fields included in the array. Sub-fields are named by pre-pending the names of their parent fields up to the top-level fields, using `'/'` as a separator. The data area of the array is copied into the new one. For example, calling `nra3.asRecArray()` would return the same array as calling:

```
>>> ra = numarray.records.array(
...     [(1, (0.5, 1.0), 'a1', 1j), (2, (0, 0), 'a2', 1+.1j)],
...     names=['id', 'pos', 'info/name', 'info/value'],
...     formats=['<Int64', '(2,<)Float32', '<a2', '<Complex64'])
```

Note that the shape of multidimensional fields is kept.

## 11.3 NestedRecord objects

Each element of the nested record array is a `NestedRecord`, i.e. a `Record` with support for nested datatypes. As said before, we can do indexing as usual:

```
>>> print nral[0]
(1, (0.5, 1.0), ('a1', 1j))
>>>
```

Using `NestedRecord` objects is quite similar to using `Record` objects. To get the data of a field we use the `field()` method. As an argument to this method we pass a field name. Sub-field names can be passed in the way described for `NestedRecArray.field()`. The `fields` attribute is also present and works as it does in `NestedRecArray`.

Field data can be set with the `setField()` method. It takes two arguments, the field name and its value. Sub-field names can be passed as usual. Finally, the `asRecord()` method converts a nested record into a non-nested equivalent record.

# UTILITIES

PyTables comes with a couple of utilities that make the life easier to the user. One is called `ptdump` and lets you see the contents of a PyTables file (or generic HDF5 file, if supported). The other one is named `ptrepack` that allows to (recursively) copy sub-hierarchies of objects present in a file into another one, changing, if desired, some of the filters applied to the leaves during the copy process.

Normally, these utilities will be installed somewhere in your `PATH` during the process of installation of the PyTables package, so that you can invoke them from any place in your file system after the installation has successfully finished.

## 12.1 ptdump

As has been said before, `ptdump` utility allows you look into the contents of your PyTables files. It lets you see not only the data but also the metadata (that is, the *structure* and additional information in the form of *attributes*).

### 12.1.1 Usage

For instructions on how to use it, just pass the `-h` flag to the command:

```
$ ptdump -h
```

to see the message usage:

```
usage: ptdump [-d] [-v] [-a] [-c] [-i] [-R start,stop,step] [-h] file[:nodepath]
  -d -- Dump data information on leaves
  -v -- Dump more metainformation on nodes
  -a -- Show attributes in nodes (only useful when -v or -d are active)
  -c -- Show info of columns in tables (only useful when -v or -d are active)
  -i -- Show info of indexed columns (only useful when -v or -d are active)
  -R RANGE -- Select a RANGE of rows in the form "start,stop,step"
  -h -- Print help on usage
```

Read on for a brief introduction to this utility.

### 12.1.2 A small tutorial on ptdump

Let's suppose that we want to know only the *structure* of a file. In order to do that, just don't pass any flag, just the file as parameter.

```
$ ptdump vllarray1.h5
/ (RootGroup) ''
/vllarray1 (VLLArray(3,), shuffle, zlib(1)) 'ragged array of ints'
/vllarray2 (VLLArray(3,), shuffle, zlib(1)) 'ragged array of strings'
```

we can see that the file contains just a leaf object called `vllarray1`, that is an instance of `VLLArray`, has 4 rows, and two filters has been used in order to create it: `shuffle` and `zlib` (with a compression level of 1).

Let's say we want more meta-information. Just add the `-v` (verbose) flag:

```
$ ptdump -v vllarray1.h5
/ (RootGroup) ''
/vllarray1 (VLLArray(3,), shuffle, zlib(1)) 'ragged array of ints'
atom = Int32Atom(shape=(), dflt=0)
    byteorder = 'little'
    nrows = 3
    flavor = 'numeric'
/vllarray2 (VLLArray(3,), shuffle, zlib(1)) 'ragged array of strings'
atom = StringAtom(itemsizes=2, shape=(), dflt='')
    byteorder = 'irrelevant'
    nrows = 3
    flavor = 'python'
```

so we can see more info about the atoms that are the components of the `vllarray1` dataset, i.e. they are scalars of type `Int32` and with NumPy *flavor*.

If we want information about the attributes on the nodes, we must add the `-a` flag:

```
$ ptdump -va vllarray1.h5
/ (RootGroup) ''
/._v_attrs (AttributeSet), 5 attributes:
    [CLASS := 'GROUP',
     PYTABLES_FORMAT_VERSION := '2.0',
     TITLE := '',
     VERSION := '1.0']
/vllarray1 (VLLArray(3,), shuffle, zlib(1)) 'ragged array of ints'
atom = Int32Atom(shape=(), dflt=0)
    byteorder = 'little'
    nrows = 3
    flavor = 'numeric'
/vllarray1._v_attrs (AttributeSet), 4 attributes:
    [CLASS := 'VLLARRAY',
     FLAVOR := 'numeric',
     TITLE := 'ragged array of ints',
     VERSION := '1.2']
/vllarray2 (VLLArray(3,), shuffle, zlib(1)) 'ragged array of strings'
atom = StringAtom(itemsizes=2, shape=(), dflt='')
    byteorder = 'irrelevant'
    nrows = 3
    flavor = 'python'
/vllarray2._v_attrs (AttributeSet), 4 attributes:
    [CLASS := 'VLLARRAY',
     FLAVOR := 'python',
     TITLE := 'ragged array of strings',
     VERSION := '1.2']
```

Let's have a look at the real data:

```
$ ptdump -d vllarray1.h5
/ (RootGroup) ''
/vllarray1 (VLLArray(3,), shuffle, zlib(1)) 'ragged array of ints'
  Data dump:
[0] [5 6]
[1] [5 6 7]
[2] [5 6 9 8]
/vllarray2 (VLLArray(3,), shuffle, zlib(1)) 'ragged array of strings'
  Data dump:
[0] ['5', '66']
[1] ['5', '6', '77']
[2] ['5', '6', '9', '88']
```

We see here a data dump of the 4 rows in `vllarray1` object, in the form of a list. Because the object is a VLA, we see a different number of integers on each row.

Say that we are interested only on a specific *row range* of the `/vllarray1` object:

```
ptdump -R2,3 -d vllarray1.h5:/vllarray1
/vllarray1 (VLLArray(3,), shuffle, zlib(1)) 'ragged array of ints'
  Data dump:
[2] [5 6 9 8]
```

Here, we have specified the range of rows between 2 and 4 (the upper limit excluded, as usual in Python). See how we have selected only the `/vllarray1` object for doing the dump (`vllarray1.h5:/vllarray1`).

Finally, you can mix several information at once:

```
$ ptdump -R2,3 -vad vllarray1.h5:/vllarray1
/vllarray1 (VLLArray(3,), shuffle, zlib(1)) 'ragged array of ints'
  atom = Int32Atom(shape=(), dtype=0)
  byteorder = 'little'
  nrows = 3
  flavor = 'numeric'
/vllarray1._v_attrs (AttributeSet), 4 attributes:
[CLASS := 'VLLARRAY',
 FLAVOR := 'numeric',
 TITLE := 'ragged array of ints',
 VERSION := '1.2']
  Data dump:
[2] [5 6 9 8]
```

## 12.2 ptrepack

This utility is a very powerful one and lets you copy any leaf, group or complete subtree into another file. During the copy process you are allowed to change the filter properties if you want so. Also, in the case of duplicated pathnames, you can decide if you want to overwrite already existing nodes on the destination file. Generally speaking, `ptrepack` can be useful in many situations, like replicating a subtree in another file, change the filters in objects and see how affect this to the compression degree or I/O performance, consolidating specific data in repositories or even *importing* generic HDF5 files and create true PyTables counterparts.

### 12.2.1 Usage

For instructions on how to use it, just pass the `-h` flag to the command:

```
$ ptrepack -h
```

to see the message usage:

```
usage: ptrepack [-h] [-v] [-o] [-R start,stop,step] [--non-recursive] [--dest-title=title] [--dont-create-sysattrs]
               [-d dest] [-s source] [-c chunksize] [-f fletcher32] [-k keep] [-u upgrade-flavors]
               [-r regenerate-old-indexes] [-sby sortby] [-c check-csi] [-p propagate-indexes]
               [-h] -- Print usage message.
               -v -- Show more information.
               -o -- Overwrite destination file.
               -R RANGE -- Select a RANGE of rows (in the form "start,stop,step")
                        during the copy of \*all* the leaves. Default values are
                        "None,None,1", which means a copy of all the rows.
               --non-recursive -- Do not do a recursive copy. Default is to do it.
               --dest-title=title -- Title for the new file (if not specified,
                        the source is copied).
               --dont-create-sysattrs -- Do not create sys attrs (default is to do it).
               --dont-copy-userattrs -- Do not copy the user attrs (default is to do it).
               --overwrite-nodes -- Overwrite destination nodes if they exist. Default is
                        to not overwrite them.
               --complevel=(0-9) -- Set a compression level (0 for no compression, which
                        is the default).
               --complib=lib -- Set the compression library to be used during the copy.
                        lib can be set to "zlib", "lzo", "bzip2" or "blosc". Defaults to
                        "zlib".
               --shuffle=(0|1) -- Activate or not the shuffling filter (default is active
                        if complevel>0).
               --fletcher32=(0|1) -- Whether to activate or not the fletcher32 filter
                        (not active by default).
               --keep-source-filters -- Use the original filters in source files. The
                        default is not doing that if any of --complevel, --complib, --shuffle
                        or --fletcher32 option is specified.
               --chunkshape=("keep"|"auto"|int|tuple) -- Set a chunkshape. A value
                        of "auto" computes a sensible value for the chunkshape of the
                        leaves copied. The default is to "keep" the original value.
               --upgrade-flavors -- When repacking PyTables 1.x files, the flavor of
                        leaves will be unset. With this, such a leaves will be serialized
                        as objects with the internal flavor ('numpy' for 2.x series).
               --dont-regenerate-old-indexes -- Disable regenerating old indexes. The
                        default is to regenerate old indexes as they are found.
               --sortby=column -- Do a table copy sorted by the index in "column".
                        For reversing the order, use a negative value in the "step" part of
                        "RANGE" (see "-R" flag). Only applies to table objects.
               --check-csi -- Force the check for a CSI index for the --sortby column.
               --propindexes -- Propagate the indexes existing in original tables. The
                        default is to not propagate them. Only applies to table objects.
```

Read on for a brief introduction to this utility.

## 12.2.2 A small tutorial on ptrepack

Imagine that we have ended the tutorial 1 (see the output of `examples/tutorial1-1.py`), and we want to copy our reduced data (i.e. those datasets that hangs from the `/column` group) to another file. First, let's remember the content of the `examples/tutorial1.h5`:

```
$ ptdump tutorial1.h5
/ (RootGroup) 'Test file'
/columns (Group) 'Pressure and Name'
/columns/name (Array(3,)) 'Name column selection'
```

```
/columns/pressure (Array(3,)) 'Pressure column selection'
/detector (Group) 'Detector information'
/detector/readout (Table(10,)) 'Readout example'
```

Now, copy the /columns to other non-existing file. That's easy:

```
$ ptrepack tutorial1.h5:/columns reduced.h5
```

That's all. Let's see the contents of the newly created reduced.h5 file:

```
$ ptdump reduced.h5
/ (RootGroup) ''
/name (Array(3,)) 'Name column selection'
/pressure (Array(3,)) 'Pressure column selection'
```

so, you have copied the children of /columns group into the *root* of the reduced.h5 file.

Now, you suddenly realized that what you intended to do was to copy all the hierarchy, the group /columns itself included. You can do that by just specifying the destination group:

```
$ ptrepack tutorial1.h5:/columns reduced.h5:/columns
$ ptdump reduced.h5
/ (RootGroup) ''
/name (Array(3,)) 'Name column selection'
/pressure (Array(3,)) 'Pressure column selection'
/columns (Group) ''
/columns/name (Array(3,)) 'Name column selection'
/columns/pressure (Array(3,)) 'Pressure column selection'
```

OK. Much better. But you want to get rid of the existing nodes on the new file. You can achieve this by adding the -o flag:

```
$ ptrepack -o tutorial1.h5:/columns reduced.h5:/columns
$ ptdump reduced.h5
/ (RootGroup) ''
/columns (Group) ''
/columns/name (Array(3,)) 'Name column selection'
/columns/pressure (Array(3,)) 'Pressure column selection'
```

where you can see how the old contents of the reduced.h5 file has been overwritten.

You can copy just one single node in the repacking operation and change its name in destination:

```
$ ptrepack tutorial1.h5:/detector/readout reduced.h5:/rawdata
$ ptdump reduced.h5
/ (RootGroup) ''
/rawdata (Table(10,)) 'Readout example'
/columns (Group) ''
/columns/name (Array(3,)) 'Name column selection'
/columns/pressure (Array(3,)) 'Pressure column selection'
```

where the /detector/readout has been copied to /rawdata in destination.

We can change the filter properties as well:

```
$ ptrepack --complevel=1 tutorial1.h5:/detector/readout reduced.h5:/rawdata
Problems doing the copy from 'tutorial1.h5:/detector/readout' to 'reduced.h5:/rawdata'
The error was --> tables.exceptions.NodeError: destination group '\'\'' already has a node named \'
The destination file looks like:
/ (RootGroup) ''
/rawdata (Table(10,)) 'Readout example'
```

```
/columns (Group) ''
/columns/name (Array(3,)) 'Name column selection'
/columns/pressure (Array(3,)) 'Pressure column selection'
Traceback (most recent call last):
  File "utils/ptrepack", line 3, in ?
    main()
  File ".../tables/scripts/ptrepack.py", line 349, in main
    stats = stats, start = start, stop = stop, step = step)
  File ".../tables/scripts/ptrepack.py", line 107, in copyLeaf
    raise RuntimeError, "Please check that the node names are not
    duplicated in destination, and if so, add the --overwrite-nodes flag
    if desired."
RuntimeError: Please check that the node names are not duplicated in
destination, and if so, add the --overwrite-nodes flag if desired.
```

Oops! We ran into problems: we forgot that the `/rawdata` pathname already existed in destination file. Let's add the `--overwrite-nodes`, as the verbose error suggested:

```
$ ptrepack --overwrite-nodes --complevel=1 tutorial1.h5:/detector/readout
reduced.h5:/rawdata
$ ptdump reduced.h5
/ (RootGroup) ''
/rawdata (Table(10,), shuffle, zlib(1)) 'Readout example'
/columns (Group) ''
/columns/name (Array(3,)) 'Name column selection'
/columns/pressure (Array(3,)) 'Pressure column selection'
```

you can check how the filter properties has been changed for the `/rawdata` table. Check as the other nodes still exists.

Finally, let's copy a *slice* of the readout table in origin to destination, under a new group called `/slices` and with the name, for example, `aslice`:

```
$ ptrepack -R1,8,3 tutorial1.h5:/detector/readout reduced.h5:/slices/aslice
$ ptdump reduced.h5
/ (RootGroup) ''
/rawdata (Table(10,), shuffle, zlib(1)) 'Readout example'
/columns (Group) ''
/columns/name (Array(3,)) 'Name column selection'
/columns/pressure (Array(3,)) 'Pressure column selection'
/slices (Group) ''
/slices/aslice (Table(3,)) 'Readout example'
```

note how only 3 rows of the original readout table has been copied to the new `aslice` destination. Note as well how the previously nonexistent `slices` group has been created in the same operation.

## 12.3 nctoh5

This tool is able to convert a file in [NetCDF](#) format to a PyTables file (and hence, to a HDF5 file). However, for this to work, you will need the NetCDF interface for Python that comes with the excellent Scientific Python (see [\[SCIPY1\]](#)) package. This script was initially contributed by Jeff Whitaker. It has been updated to support selectable filters from the command line and some other small improvements.

If you want other file formats to be converted to PyTables, have a look at the SciPy (see [\[SCIPY1\]](#)) project (subpackage `io`), and look for different methods to import them into NumPy/Numeric/numarray objects. Following the SciPy documentation, you can read, among other formats, ASCII files (`read_array`), binary files in C or Fortran (`fopen`) and MATLAB (version 4, 5 or 6) files (`loadmat`). Once you have the content of your files as NumPy/Numeric/numarray

objects, you can save them as regular (E)Arrays in PyTables files. Remember, if you end with a nice converter, do not forget to contribute it back to the community. Thanks!

### 12.3.1 Usage

For instructions on how to use it, just pass the `-h` flag to the command:

```
$ nctoh5 -h
```

to see the message usage:

```
usage: nctoh5 [-h] [-v] [-o] [--complevel=(0-9)] [--complib=lib] [--shuffle=(0|1)] [--fletcher32=(0|1)]
  -h -- Print usage message.
  -v -- Show more information.
  -o -- Overwrite destination file.
 --complevel=(0-9) -- Set a compression level (0 for no compression, which
                    is the default).
 --complib=lib -- Set the compression library to be used during the copy.
                    lib can be set to "zlib" or "lzo". Defaults to "zlib".
 --shuffle=(0|1) -- Activate or not the shuffling filter (default is active
                    if complevel>0).
 --fletcher32=(0|1) -- Whether to activate or not the fletcher32 filter (not
                    active by default).
```



# PYTABLES FILE FORMAT

PyTables has a powerful capability to deal with native HDF5 files created with another tools. However, there are situations where you may want to create truly native PyTables files with those tools while retaining full compatibility with the PyTables format. That is perfectly possible, and in this appendix is presented the format that you should endow to your own-generated files in order to get a fully PyTables compatible file.

We are going to describe the *2.0 version of PyTables file format* (introduced in PyTables version 2.0). As time goes by, some changes might be introduced (and documented here) in order to cope with new necessities. However, the changes will be carefully pondered so as to ensure backward compatibility whenever is possible.

A PyTables file is composed with arbitrarily large amounts of HDF5 groups (Groups in PyTables naming scheme) and datasets (Leaves in PyTables naming scheme). For groups, the only requirements are that they must have some *system attributes* available. By convention, system attributes in PyTables are written in upper case, and user attributes in lower case but this is not enforced by the software. In the case of datasets, besides the mandatory system attributes, some conditions are further needed in their storage layout, as well as in the datatypes used in there, as we will see shortly.

As a final remark, you can use any filter as you want to create a PyTables file, provided that the filter is a standard one in HDF5, like *zlib*, *shuffle* or *gzip* (although the last one can not be used from within PyTables to create a new file, datasets compressed with *gzip* can be read, because it is the HDF5 library which does the decompression transparently).

## 13.1 Mandatory attributes for a File

The File object is, in fact, an special HDF5 *group* structure that is *root* for the rest of the objects on the object tree. The next attributes are mandatory for the HDF5 *root group* structure in PyTables files:

- **CLASS**: This attribute should always be set to 'GROUP' for group structures.
- **PYTABLES\_FORMAT\_VERSION**: It represents the internal format version, and currently should be set to the '2.0' string.
- **TITLE**: A string where the user can put some description on what is this group used for.
- **VERSION**: Should contains the string '1.0'.

## 13.2 Mandatory attributes for a Group

The next attributes are mandatory for *group* structures:

- **CLASS**: This attribute should always be set to 'GROUP' for group structures.
- **TITLE**: A string where the user can put some description on what is this group used for.
- **VERSION**: Should contains the string '1.0'.

## 13.3 Optional attributes for a Group

The next attributes are optional for *group* structures:

- **FILTERS**: When present, this attribute contains the filter properties (a *Filters* instance, see section *The Filters class*) that may be inherited by leaves or groups created immediately under this group. This is a packed 64-bit integer structure, where
  - *byte 0* (the least-significant byte) is the compression level (*complevel*).
  - *byte 1* is the compression library used (*complib*): 0 when irrelevant, 1 for Zlib, 2 for LZO and 3 for Bzip2.
  - *byte 2* indicates which parameterless filters are enabled (*shuffle* and *fletcher32*): bit 0 is for *Shuffle* while bit 1 is for *\*Fletcher32\**.
  - other bytes are reserved for future use.

## 13.4 Mandatory attributes, storage layout and supported data types for Leaves

This depends on the kind of Leaf. The format for each type follows.

### 13.4.1 Table format

#### Mandatory attributes

The next attributes are mandatory for *table* structures:

- **CLASS**: Must be set to 'TABLE'.
- **TITLE**: A string where the user can put some description on what is this dataset used for.
- **VERSION**: Should contain the string '2.6'.
- **FIELD\_X\_NAME**: It contains the names of the different fields. The X means the number of the field, zero-based (beware, order do matter). You should add as many attributes of this kind as fields you have in your records.
- **FIELD\_X\_FILL**: It contains the default values of the different fields. All the datatypes are supported natively, except for complex types that are currently serialized using Pickle. The X means the number of the field, zero-based (beware, order do matter). You should add as many attributes of this kind as fields you have in your records. These fields are meant for saving the default values persistently and their existence is optional.
- **NROWS**: This should contain the number of *compound* data type entries in the dataset. It must be an *int* data type.

#### Storage Layout

A Table has a *dataspace* with a *1-dimensional chunked* layout.

#### Datatypes supported

The datatype of the elements (rows) of Table must be the `H5T_COMPOUND` *compound* data type, and each of these compound components must be built with only the next HDF5 data types *classes*:

- **H5T\_BITFIELD:** This class is used to represent the Bool type. Such a type must be build using a H5T\_NATIVE\_B8 datatype, followed by a HDF5 H5Tset\_precision call to set its precision to be just 1 bit.
- **H5T\_INTEGER: This includes the next data types:**
  - **H5T\_NATIVE\_SCHAR:** This represents a *signed char* C type, but it is effectively used to represent an Int8 type.
  - **H5T\_NATIVE\_UCHAR:** This represents an *unsigned char* C type, but it is effectively used to represent an UInt8 type.
  - **H5T\_NATIVE\_SHORT:** This represents a *short* C type, and it is effectively used to represent an Int16 type.
  - **H5T\_NATIVE\_USHORT:** This represents an *unsigned short* C type, and it is effectively used to represent an UInt16 type.
  - **H5T\_NATIVE\_INT:** This represents an *int* C type, and it is effectively used to represent an Int32 type.
  - **H5T\_NATIVE\_UINT:** This represents an *unsigned int* C type, and it is effectively used to represent an UInt32 type.
  - **H5T\_NATIVE\_LONG:** This represents a *long* C type, and it is effectively used to represent an Int32 or an Int64, depending on whether you are running a 32-bit or 64-bit architecture.
  - **H5T\_NATIVE\_ULONG:** This represents an *unsigned long* C type, and it is effectively used to represent an UInt32 or an UInt64, depending on whether you are running a 32-bit or 64-bit architecture.
  - **H5T\_NATIVE\_LLONG:** This represents a *long long* C type (`__int64`, if you are using a Windows system) and it is effectively used to represent an Int64 type.
  - **H5T\_NATIVE\_ULLONG:** This represents an *unsigned long long* C type (beware: this type does not have a correspondence on Windows systems) and it is effectively used to represent an UInt64 type.
- **H5T\_FLOAT: This includes the next datatypes:**
  - **H5T\_NATIVE\_FLOAT:** This represents a *float* C type and it is effectively used to represent an Float32 type.
  - **H5T\_NATIVE\_DOUBLE:** This represents a *double* C type and it is effectively used to represent an Float64 type.
- **H5T\_TIME: This includes the next datatypes:**
  - **H5T\_UNIX\_D32:** This represents a POSIX *time\_t* C type and it is effectively used to represent a ‘Time32’ aliasing type, which corresponds to an Int32 type.
  - **H5T\_UNIX\_D64:** This represents a POSIX *struct timeval* C type and it is effectively used to represent a ‘Time64’ aliasing type, which corresponds to a Float64 type.
- **H5T\_STRING:** The datatype used to describe strings in PyTables is H5T\_C\_S1 (i.e. a *string* C type) followed with a call to the HDF5 H5Tset\_size() function to set their length.
- **H5T\_ARRAY:** This allows the construction of homogeneous, multidimensional arrays, so that you can include such objects in compound records. The types supported as elements of H5T\_ARRAY data types are the ones described above. Currently, PyTables does not support nested H5T\_ARRAY types.
- **H5T\_COMPOUND:** This allows the support for datatypes that are compounds of compounds (this is also known as *nested types* along this manual).

This support can also be used for defining complex numbers. Its format is described below:

The H5T\_COMPOUND type class contains two members. Both members must have the H5T\_FLOAT atomic datatype class. The name of the first member should be “r” and represents the real part. The name of the second member should be “i” and represents the imaginary part. The *precision* property of both of the

H5T\_FLOAT members must be either 32 significant bits (e.g. H5T\_NATIVE\_FLOAT) or 64 significant bits (e.g. H5T\_NATIVE\_DOUBLE). They represent Complex32 and Complex64 types respectively.

### 13.4.2 Array format

#### Mandatory attributes

The next attributes are mandatory for *array* structures:

- *CLASS*: Must be set to 'ARRAY'.
- *TITLE*: A string where the user can put some description on what is this dataset used for.
- *VERSION*: Should contain the string '2.3'.

#### Storage Layout

An Array has a *dataspace* with a *N-dimensional contiguous* layout (if you prefer a *chunked* layout see EArray below).

#### Datatypes supported

The elements of Array must have either HDF5 *atomic* data types or a *compound* data type representing a complex number. The atomic data types can currently be one of the next HDF5 data type *classes*: H5T\_BITFIELD, H5T\_INTEGER, H5T\_FLOAT and H5T\_STRING. The H5T\_TIME class is also supported for reading existing Array objects, but not for creating them. See the Table format description in [Table format](#) for more info about these types.

In addition to the HDF5 atomic data types, the Array format supports complex numbers with the H5T\_COMPOUND data type class. See the Table format description in [Table format](#) for more info about this special type.

You should note that H5T\_ARRAY class datatypes are not allowed in Array objects.

### 13.4.3 CArray format

#### Mandatory attributes

The next attributes are mandatory for *CArray* structures:

- *CLASS*: Must be set to 'CARRAY'.
- *TITLE*: A string where the user can put some description on what is this dataset used for.
- *VERSION*: Should contain the string '1.0'.

#### Storage Layout

An CArray has a *dataspace* with a *N-dimensional chunked* layout.

## Datatypes supported

The elements of CArray must have either HDF5 *atomic* data types or a *compound* data type representing a complex number. The atomic data types can currently be one of the next HDF5 data type *classes*: H5T\_BITFIELD, H5T\_INTEGER, H5T\_FLOAT and H5T\_STRING. The H5T\_TIME class is also supported for reading existing CArray objects, but not for creating them. See the Table format description in [Table format](#) for more info about these types.

In addition to the HDF5 atomic data types, the CArray format supports complex numbers with the H5T\_COMPOUND data type class. See the Table format description in [Table format](#) for more info about this special type.

You should note that H5T\_ARRAY class datatypes are not allowed yet in Array objects.

### 13.4.4 EArray format

#### Mandatory attributes

The next attributes are mandatory for *earray* structures:

- *CLASS*: Must be set to 'EARRAY'.
- *EXTDIM*: (*Integer*) Must be set to the extendable dimension. Only one extendable dimension is supported right now.
- *TITLE*: A string where the user can put some description on what is this dataset used for.
- *VERSION*: Should contain the string '1.3'.

#### Storage Layout

An EArray has a *dataspace* with a *N-dimensional chunked* layout.

## Datatypes supported

The elements of EArray are allowed to have the same data types as for the elements in the Array format. They can be one of the HDF5 *atomic* data type *classes*: H5T\_BITFIELD, H5T\_INTEGER, H5T\_FLOAT, H5T\_TIME or H5T\_STRING, see the Table format description in [Table format](#) for more info about these types. They can also be a H5T\_COMPOUND datatype representing a complex number, see the Table format description in [Table format](#).

You should note that H5T\_ARRAY class data types are not allowed in EArray objects.

### 13.4.5 VLArray format

#### Mandatory attributes

The next attributes are mandatory for *vlarray* structures:

- *CLASS*: Must be set to 'VLARRAY'.
- *PSEUDOATOM*: This is used so as to specify the kind of pseudo-atom (see [VLArray format](#)) for the VLArray. It can take the values 'vlstring', 'vlunicode' or 'object'. If your atom is not a pseudo-atom then you should not specify it.
- *TITLE*: A string where the user can put some description on what is this dataset used for.
- *VERSION*: Should contain the string '1.3'.

## Storage Layout

An VArray has a *dataspace* with a *1-dimensional chunked* layout.

## Data types supported

The data type of the elements (rows) of VArray objects must be the H5T\_VLEN *variable-length* (or VL for short) datatype, and the base datatype specified for the VL datatype can be of any *atomic* HDF5 datatype that is listed in the Table format description *Table format*. That includes the classes:

- H5T\_BITFIELD
- H5T\_INTEGER
- H5T\_FLOAT
- H5T\_TIME
- H5T\_STRING
- H5T\_ARRAY

They can also be a H5T\_COMPOUND data type representing a complex number, see the Table format description in *Table format* for a detailed description.

You should note that this does not include another VL datatype, or a compound datatype that does not fit the description of a complex number. Note as well that, for object and vlstring pseudo-atoms, the base for the VL datatype is always a H5T\_NATIVE\_UCHAR (H5T\_NATIVE\_UINT for vlunicode). That means that the complete row entry in the dataset has to be used in order to fully serialize the object or the variable length string.

## 13.5 Optional attributes for Leaves

The next attributes are optional for *leaves*:

- *FLAVOR*: This is meant to provide the information about the kind of object kept in the Leaf, i.e. when the dataset is read, it will be converted to the indicated flavor. It can take one the next string values:
  - “*numpy*”: Read data (structures arrays, arrays, records, scalars) will be returned as NumPy objects.
  - “*numarray*”: Read data will be returned as numarray objects (deprecated).
  - “*numeric*”: Read data will be returned as Numeric objects (deprecated).
  - “*python*”: Read data will be returned as Python lists, tuples, or scalars.

## BIBLIOGRAPHY

- [HDFG1]** The HDF Group. What is HDF5?. Concise description about HDF5 capabilities and its differences from earlier versions (HDF4). <http://www.hdfgroup.org/whatishdf5.html>.
- [HDFG2]** The HDF Group. Introduction to HDF5. Introduction to the HDF5 data model and programming model. <http://www.hdfgroup.org/HDF5/doc/H5.intro.html>.
- [HDFG3]** The HDF Group. The HDF5 table programming model. Examples on using HDF5 tables with the C API. <http://www.hdfgroup.org/HDF5/Tutor/h5table.html>.
- [MERTZ]** David Mertz. Objectify. On the ‘Pythonic’ treatment of XML documents as objects(II). Article describing XML Objectify, a Python module that allows working with XML documents as Python objects. Some of the ideas presented here are used in PyTables. <http://www.ibm.com/developerworks/xml/library/xml-matters2/index.html>.
- [CYTHON]** Stefan Behnel, Robert Bradshaw, Dag Sverre Seljebotn, and Greg Ewing. Cython. A language that makes writing C extensions for the Python language as easy as Python itself. <http://cython.org>.
- [NETCDF1]** Glenn Davis, Russ Rew, Steve Emmerson, John Caron, and Harvey Davies. NetCDF. Network Common Data Form. An interface for array-oriented data access and a library that provides an implementation of the interface. <http://www.unidata.ucar.edu/software/netcdf/>.
- [NETCDF2]** Russ Rew, Mike Folk, and et al. NetCDF-4. Network Common Data Form version 4. Merging the NetCDF and HDF5 Libraries. <http://www.unidata.ucar.edu/software/netcdf/index.html>.
- [NUMPY]** Travis Oliphant and et al. NumPy. Scientific Computing with Numerical Python. The latest and most powerful re-implementation of Numeric to date. It implements all the features that can be found in Numeric and numarray, plus a bunch of new others. In general, it is more efficient as well. <http://numpy.scipy.org/>.
- [NUMERIC]** David Ascher, Paul F Dubois, Konrad Hinsien, Jim Hugunin, and Travis Oliphant. Numerical Python. Package to speed-up arithmetic operations on arrays of numbers. <http://sourceforge.net/projects/numpy/>.
- [NUMARRAY]** Perry Greenfield, Todd Miller, Richard L White, J. C. Hsu, Paul Barrett, Jochen Küpper, and Peter J Verveer. Numarray. Reimplementation of Numeric which adds the ability to efficiently manipulate large numeric arrays in ways similar to Matlab and IDL. Among others, Numarray provides the record array extension. <http://stsdas.stsci.edu/numarray/>.
- [NUMEXPR]** David Cooke, Francesc Altred, and et al. Numexpr. Fast evaluation of array expressions by using a vector-based virtual machine. It is an enhanced computing kernel that is generally faster (between 1x and 10x, depending on the kind of operations) than NumPy at evaluating complex array expressions. <http://code.google.com/p/numexpr>.
- [ZLIB]** JeanLoup Gailly and Mark Adler. zlib. A Massively Spiffy Yet Delicately Unobtrusive Compression Library. A standard library for compression purposes. <http://www.gzip.org/zlib/>.

- [LZO]** Markus F Oberhumer. LZO. A data compression library which is suitable for data de-/compression in real-time. It offers pretty fast compression and decompression with reasonable compression ratio. <http://www.oberhumer.com/opensource/>.
- [BZIP2]** Julian Seward. bzip2. A high performance lossless compressor. It offers very high compression ratios within reasonable times. <http://www.bzip.org/>.
- [BLOSC]** Francesc Alted. Blosc. A blocking, shuffling and loss-less compression library. A compressor designed to transmit data from memory to CPU (and back) faster than a plain memcpy(). <http://blosc.pytables.org/>.
- [GNUWIN32]** Alexis Wilke, Jerry S., Kees Zeelenberg, and Mathias Michaelis. GnuWin32. GNU (and other) tools ported to Win32. GnuWin32 provides native Win32-versions of GNU tools, or tools with a similar open source licence. <http://gnuwin32.sourceforge.net/>.
- [PYSCO]** Armin Rigo. Psyco. A Python specializing compiler. Run existing Python software faster, with no change in your source. <http://psyco.sourceforge.net>.
- [SCIPY1]** Konrad Hinsen. Scientific Python. Collection of Python modules useful for scientific computing. <http://dirac.cnrs-orleans.fr/ScientificPython>.
- [SCIPY2]** Eric Jones, Travis Oliphant, Pearu Peterson, and et al. SciPy. Scientific tools for Python. SciPy supplements the popular Numeric module, gathering a variety of high level science and engineering modules together as a single package. <http://www.scipy.org>.
- [OPTIM]** Francesc Alted and Ivan Vilata. Optimization of file openings in PyTables. This document explores the savings of the opening process in terms of both CPU time and memory, due to the adoption of a LRU cache for the nodes in the object tree. <http://www.pytables.org/docs/NewObjectTreeCache.pdf>.
- [OPSI]** Francesc Alted and Ivan Vilata. OPSI: The indexing system of PyTables 2 Professional Edition. Exhaustive description and benchmarks about the indexing engine that comes with PyTables Pro. <http://www.pytables.org/docs/OPSI-indexes.pdf>.
- [VITABLES]** Vicent Mas. ViTables. A GUI for PyTables/HDF5 files. It is a graphical tool for browsing and editing files in both PyTables and HDF5 formats. <http://vitables.org>.
- [GIT]** Git is a free and open source, distributed version control system designed to handle everything from small to very large projects with speed and efficiency <http://git-scm.com>.
- [SPHINX]** Sphinx is a tool that makes it easy to create intelligent and beautiful documentation, written by Georg Brandl and licensed under the BSD license <http://sphinx.pocoo.org/>.

# INDEX

## Symbols

`_fillvalue_dict` (in module `tables.netcdf3`), 119  
`_v_pos` (`tables.Col` attribute), 69

## A

`append()` (`tables.netcdf3.NetCDFVariable` method), 121  
`assignValue()` (`tables.netcdf3.NetCDFVariable` method), 121  
`atom` (`tables.Array` attribute), 66

## B

`BOUNDS_MAX_SIZE` (in module `tables.parameters`), 132  
`BOUNDS_MAX_SLOTS` (in module `tables.parameters`), 132  
`BUFFER_TIMES` (in module `tables.parameters`), 133  
`BZIP2_DIR`, 18, 19

## C

`CHUNK_CACHE_NELMTS` (in module `tables.parameters`), 132  
`CHUNK_CACHE_PREEMPT` (in module `tables.parameters`), 132  
`CHUNK_CACHE_SIZE` (in module `tables.parameters`), 132  
`close()` (`tables.netcdf3.NetCDFFile` method), 120  
`close()` (`tables.nodes.filenode.FileNode` method), 112  
`COND_CACHE_SLOTS` (in module `tables.parameters`), 132  
`createDimension()` (`tables.netcdf3.NetCDFFile` method), 120  
`createVariable()` (`tables.netcdf3.NetCDFFile` method), 120

## D

`DISABLE_EVERY_CYCLES` (in module `tables.parameters`), 133

## E

`ENABLE_EVERY_CYCLES` (in module `tables.parameters`), 133

environment variable

`BZIP2_DIR`, 18, 19  
`HDF5_DIR`, 18, 19  
`LD_LIBRARY_PATH`, 18  
`LIBS`, 18  
`LZO_DIR`, 18, 19  
`PATH`, 21  
`PYTHONPATH`, 19–21  
`EXPECTED_ROWS_EARRAY` (in module `tables.parameters`), 133  
`EXPECTED_ROWS_TABLE` (in module `tables.parameters`), 133

## F

`FileNode` (class in `tables.nodes.filenode`), 111  
`flush()` (`tables.nodes.filenode.RAFileNode` method), 113

## G

`getAttrs()` (`tables.nodes.filenode.FileNode` method), 112  
`getLineSeparator()` (`tables.nodes.filenode.FileNode` method), 112  
`getValue()` (`tables.netcdf3.NetCDFVariable` method), 121

## H

`h5tonc()` (`tables.netcdf3.NetCDFFile` method), 121  
`HDF5_DIR`, 18, 19

## I

`IO_BUFFER_SIZE` (in module `tables.parameters`), 133  
`ITERSEQ_MAX_ELEMENTS` (in module `tables.parameters`), 132  
`ITERSEQ_MAX_SIZE` (in module `tables.parameters`), 132  
`ITERSEQ_MAX_SLOTS` (in module `tables.parameters`), 132

## L

`LD_LIBRARY_PATH`, 18  
`LIBS`, 18  
`LIMBOUNDS_MAX_SIZE` (in module `tables.parameters`), 132

LIMBOUNDS\_MAX\_SLOTS (in module tables.parameters), [132](#)  
LOWEST\_HIT\_RATIO (in module tables.parameters), [133](#)  
LZO\_DIR, [18](#), [19](#)

## M

MAX\_BLOSC\_THREADS (in module tables.parameters), [134](#)  
MAX\_COLUMNS (in module tables.parameters), [131](#)  
MAX\_GROUP\_WIDTH (in module tables.parameters), [131](#)  
MAX\_NODE\_ATTRS (in module tables.parameters), [131](#)  
MAX\_NUMEXPR\_THREADS (in module tables.parameters), [134](#)  
MAX\_THREADS (in module tables.parameters), [133](#)  
MAX\_UNDO\_PATH\_LENGTH (in module tables.parameters), [131](#)  
METADATA\_CACHE\_SIZE (in module tables.parameters), [132](#)

## N

ncattrs() (tables.netcdf3.NetCDFFile method), [120](#)  
ncattrs() (tables.netcdf3.NetCDFVariable method), [121](#)  
nctoh5() (tables.netcdf3.NetCDFFile method), [120](#)  
nelements (tables.tables.index.Index attribute), [69](#)  
NetCDFFile (class in tables.netcdf3), [120](#)  
NetCDFVariable (class in tables.netcdf3), [121](#)  
newNode() (in module tables.nodes.filenode), [111](#)  
next() (tables.nodes.filenode.FileNode method), [112](#)  
NODE\_CACHE\_SLOTS (in module tables.parameters), [132](#)  
NodeType (in module tables.nodes.filenode), [111](#)  
NodeTypeVersions (in module tables.nodes.filenode), [111](#)  
nrow (tables.Array attribute), [66](#)

## O

openNode() (in module tables.nodes.filenode), [111](#)

## P

PATH, [21](#)  
PYTABLES\_SYS\_ATTRS (in module tables.parameters), [133](#)  
PYTHONPATH, [19–21](#)

## R

RAFileNode (class in tables.nodes.filenode), [113](#)  
read() (tables.nodes.filenode.FileNode method), [112](#)  
readline() (tables.nodes.filenode.FileNode method), [112](#)  
readlines() (tables.nodes.filenode.FileNode method), [112](#)  
ROFileNode (class in tables.nodes.filenode), [112](#)

## S

ScientificIONetCDF\_imported (in module tables.netcdf3), [120](#)  
seek() (tables.nodes.filenode.FileNode method), [112](#)  
setLineSeparator() (tables.nodes.filenode.FileNode method), [112](#)  
size\_in\_memory (tables.Leaf attribute), [65](#)  
SORTED\_MAX\_SIZE (in module tables.parameters), [132](#)  
SORTEDLR\_MAX\_SIZE (in module tables.parameters), [133](#)  
SORTEDLR\_MAX\_SLOTS (in module tables.parameters), [133](#)  
sync() (tables.netcdf3.NetCDFFile method), [120](#)

## T

TABLE\_MAX\_SIZE (in module tables.parameters), [132](#)  
tell() (tables.nodes.filenode.FileNode method), [112](#)  
truncate() (tables.nodes.filenode.RAFileNode method), [113](#)  
typecode() (tables.netcdf3.NetCDFVariable method), [121](#)

## W

write() (tables.nodes.filenode.RAFileNode method), [113](#)  
writelines() (tables.nodes.filenode.RAFileNode method), [113](#)

## X

xreadlines() (tables.nodes.filenode.FileNode method), [112](#)