
# PyTables User's Guide

## Hierarchical datasets in Python - Release 2.1

**Francesc Alted**
**Ivan Vilata**
**Scott Prater**
**Vicent Mas**
**Tom Hedley**
**Antonio Valentino**
**Jeffrey Whitaker**

# PyTables User's Guide: Hierarchical datasets in Python - Release 2.1

by Francesc Alted, Ivan Vilata, Scott Prater, Vicent Mas, Tom Hedley, Antonio Valentino, and Jeffrey Whitaker

Published $LastChangedDate: 2008-12-18 19:52:18 +0100 (dj, 18 des 2008) $

# Table of Contents

# List of Figures

# List of Tables

# Part I. The PyTables Core Library

# Chapter 1. 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 [1]).

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
    class Properties(IsDescription):  # A sub-structure (nested data-type)
        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 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.


**Note**

Column indexing is only available in PyTables Pro.

- *Support for numerical arrays:* `NumPy` (see [8]), `Numeric` (see [9]) and `numarray` (see [10]) 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 efficency 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 `VLArray` objects). Besides, if a dataset is not supported, it will be mapped to a special `UnImplemented` class (see Section 4.11), 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* and *bzip2* 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`, `VLArray` 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* [1]. 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[2]. This is very *Pythonic* and intuitive in many cases. Check the tutorial Section 3.1.6 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 [19] for more facts about the advantages introduced by this new metadata cache system.

[1] PyTables does not support hard links – for the moment.
[2] I got this simple but powerful idea from the excellent `Objectify` module by David Mertz (see [4])

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:

```
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 = 1)  # 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 1.1 [3] . 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.

[3]We have used ViTables (see [21]) in order to create this snapshot.

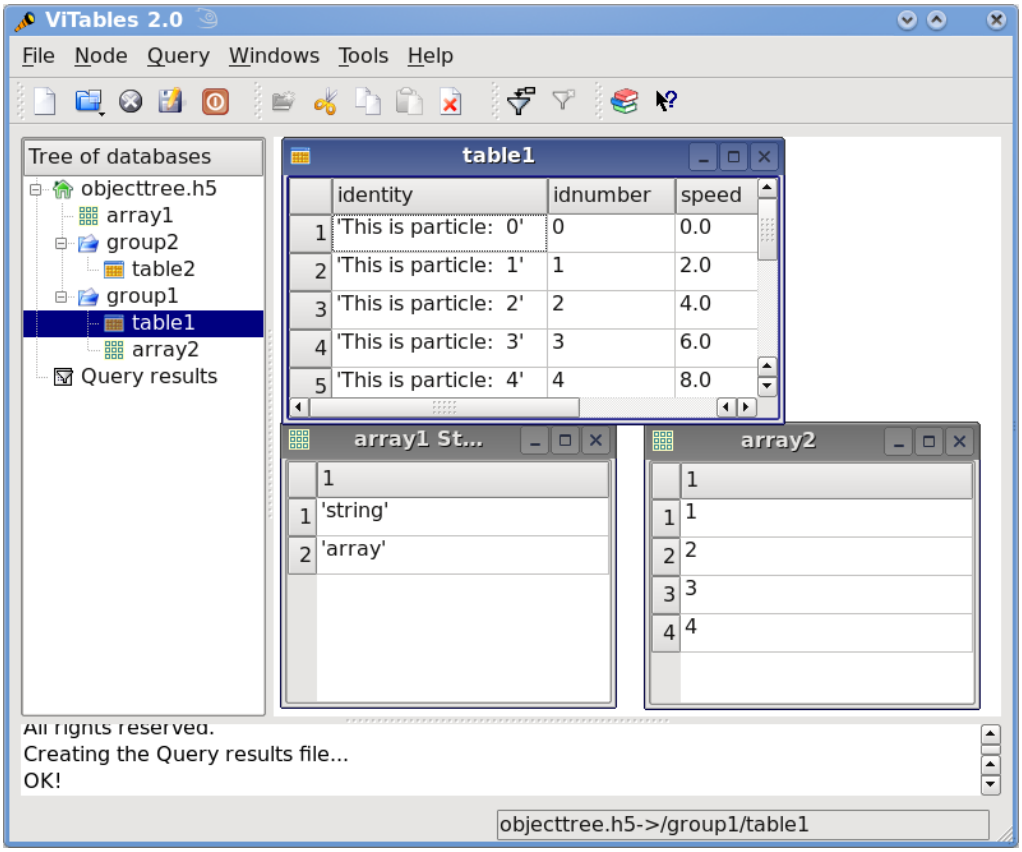


Figure 1.1. An HDF5 example with 2 subgroups, 2 tables and 1 array.

In Figure 1.2 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 understand it [4] . It will help you understand the relationships of in-memory PyTables objects.

[4]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.

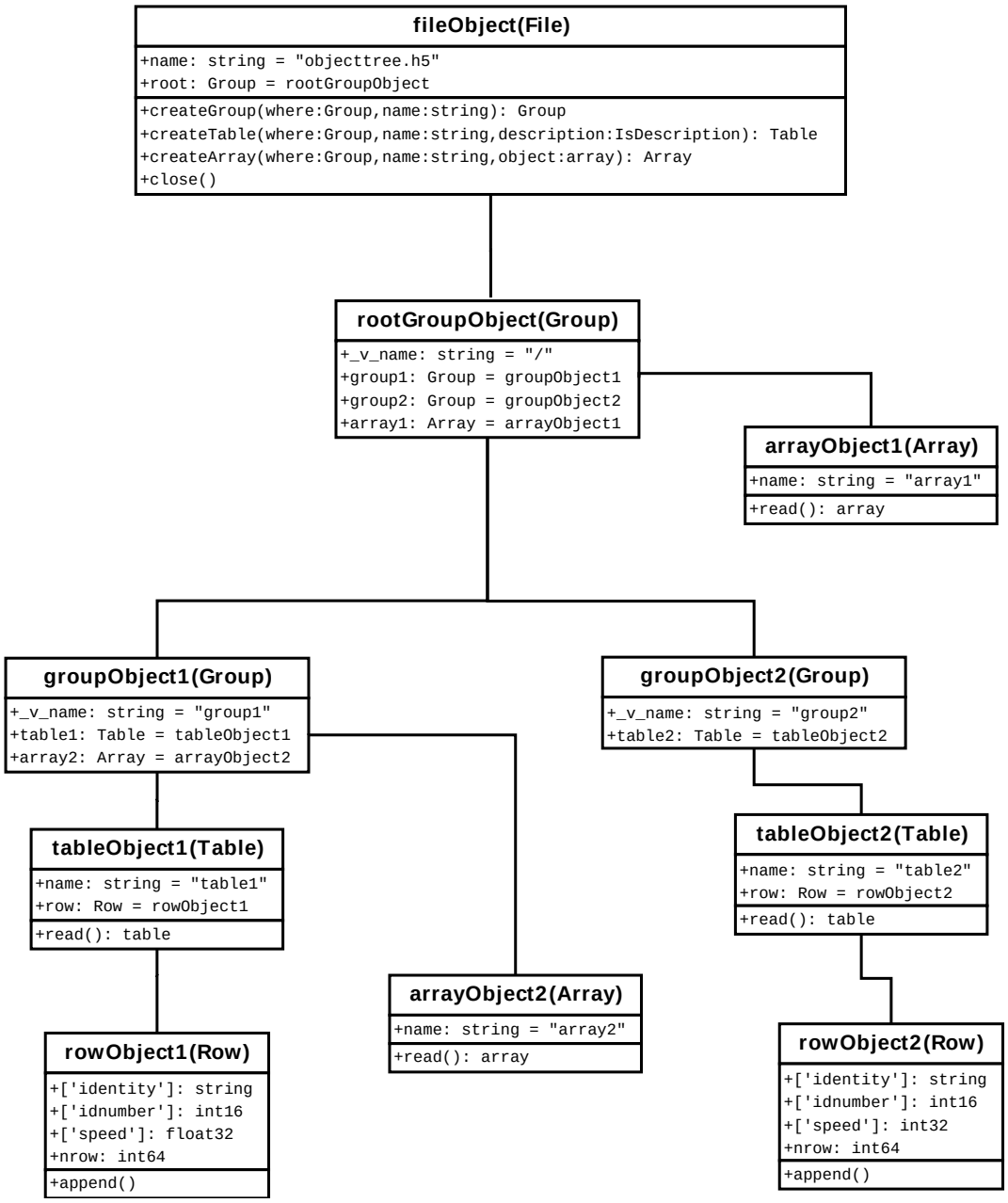


Figure 1.2. A PyTables object tree example.

# Chapter 2. 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 Section 2.2. 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 Pyrex (see [5]) and the C language. You can rebuild everything from scratch if you have Pyrex installed, but this is not necessary, as the Pyrex 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://hdfgroup.org/, and the `NumPy` (see [8]) package. Although you won't need `numarray` (see [10]) or `Numeric` (see [9]) in order to compile PyTables, they are supported; you only need a reasonably recent version of them (>= 1.5.2 for numarray and >= 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.

### 2.1.1. Prerequisites

First, make sure that you have at least Python 2.4, HDF5 1.6.5 and NumPy 1.2 or higher installed (for testing purposes, we are using HDF5 1.6.7/1.8.0 and NumPy 1.2 currently). If you don't, fetch and install them before proceeding.

Compile and install these packages (but see Section 2.2.1 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 [12]), which is also required by HDF5 as well. You may also optionally install the excellent `LZO` compression library (see [13] and Section 5.3). The high-performance bzip2 compression library can also be used with PyTables (see [14]).

**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` (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/hdf5-1.8.2
--lzo=/stuff/lzo-2.02
```

```
--bzip2=/stuff/bzip2-1.0.4
```

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/hdf5-1.8.2/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="hdf5-1.8.2 nsl"
```

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

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

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.

**Windows** You can get ready-to-use Windows binaries and other development files for most of the following libraries from the GnuWin32 project (see [15]).

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\5-165-win
set LZO_DIR=c:\stuff\lzo-1-08
set BZIP2_DIR=c:\stuff\bzip2-1-0-3
```

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

```
--hdf5=c:\stuff\5-165-win
--lzo=c:\stuff\lzo-1-08
--bzip2=c:\stuff\bzip2-1-0-3
```

## 2.1.2. PyTables package installation

Once you have installed the HDF5 library and the NumPy package, 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
   ```

   Depending on the compiler flags used when compiling your Python executable, there may appear many warnings. Don't worry, almost all of them are caused by variables declared but never used. That's normal in Pyrex extensions.

2. To run the test suite, execute this command:

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

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

or, if you prefer:

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

Both commands do the same thing.

**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()"
```

If you would like to see verbose output from the tests simply add the `-v` flag or the word `verbose` to the command line. 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:

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

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

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

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

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

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

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

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 `--show-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.

## 2.2.1. Windows prerequisites

First, make sure that you have Python 2.4 or higher and NumPy 1.2 or higher installed (PyTables binaries have been built using NumPy 1.2). The binaries already include DLLs for HDF5 (1.6.7), zlib1 (1.2.3), szlib (2.0, uncompression support only) and bzip2 (1.0.4). 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 Section 5.3 for hints about how it may be used to improve performance), fetch and install the `LZO` (choose v1.x, `LZO` v2.x is not supported in precompiled Windows builds) from [15]. Normally, you will only need to fetch and install the `<package>-<version>-bin.zip` file and copy the `lzo1.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.

# Chapter 3. Tutorials

> Seràs la clau que obre tots els panys,
> seràs la llum, la llum il.limitada,
> seràs confí 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 Chapter 4. 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` (or `numarray` or `Numeric`) arrays (and normally, you will) you will also need to import functions from them. So most PyTables programs begin with:

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

### 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 simple-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 Section 4.13.2 for a complete description of these subclasses. See also Appendix A 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 first-level `openFile` function (see description) to create a PyTables file:

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

`openFile()` (see description) 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 Section 4.2) 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 `createGroup()` method (see description) 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 Section 4.4) object instance that will be assigned to the variable `group`.

## 3.1.5. Creating a new table

Let's now create a `Table` (see Section 4.6) object as a branch off the newly-created group. We do that by calling the `createTable` (see description) 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, shuffle=False, fletcher32=False))
/ (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 Section 4.6.7) 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 Section 4.6.7). 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'] < 50 ]
>>> 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 (see description). 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.

Let's select the `name` column for the same set of cuts:

```
>>> names = [ x['name'] for x in table
          if x['TDCcount'] > 3 and 20 <= x['pressure'] < 50 ]
```

```
>>> names
['Particle:      5', 'Particle:      6', 'Particle:      7']
```

Note how we have omitted the `iterrows()` call in the list comprehension. The `Table` class has an implementation of the special method `__iter__()` that iterates over all the rows in the table. In fact, `iterrows()` internally calls this special `__iter__()` method. Accessing all the rows in a table using this method is very convenient, especially when working with the data interactively.

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()` (see description) and other related methods. See Appendix B and Section 5.2 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 `createArray` (see description) 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, `createArray()` accepts *names* (which is a regular Python list) as an *object* parameter. Actually, it accepts a variety of different regular objects (see description) 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
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'
```

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

Also, in Figure 3.1, you can admire how the `tutorial1.h5` looks like using the ViTables [http://www.vitables.org] graphical interface .

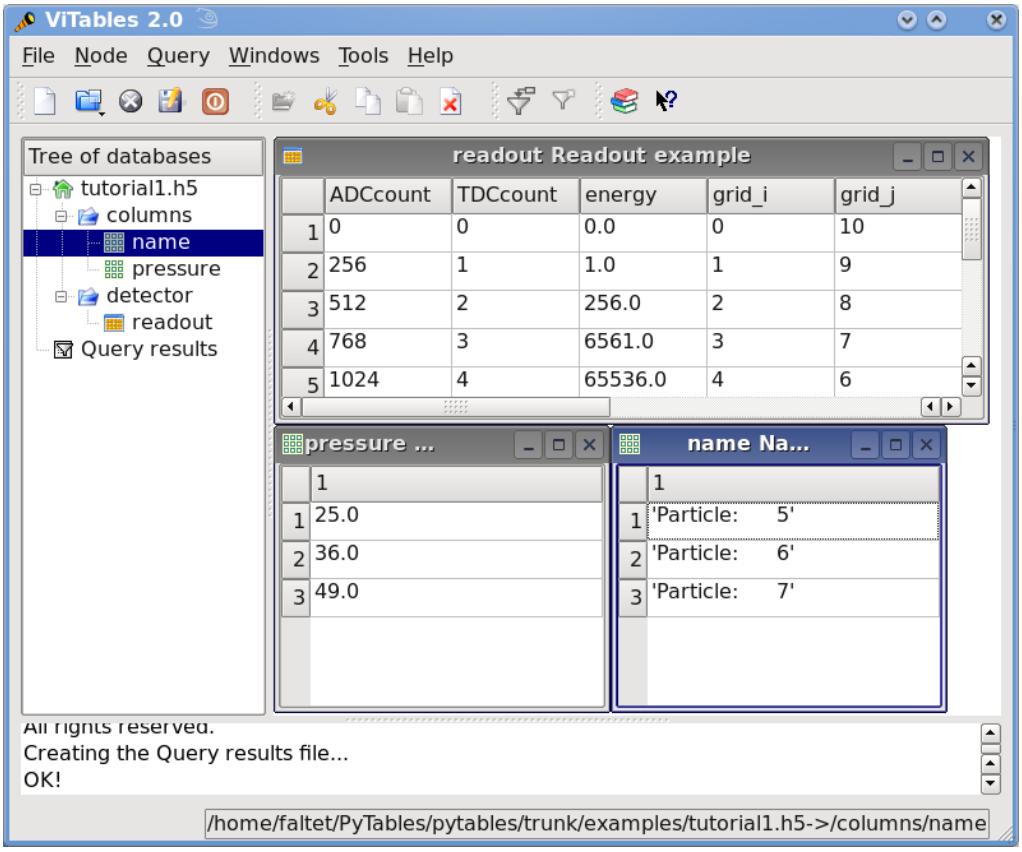


Figure 3.1. The initial version of the data file for tutorial 1, with a view of the data objects.

# 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 `walkGroups` method (see description) 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 `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'
```

`listNodes()` (see description) 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 `iterNodes()` (see description) that might be handy is 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 `walkNodes(where, classname)` special method of the `File` object (see description). 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 Section 4.5 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(classname)` method (see description), 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 Section 4.12). 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 Section 4.12 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 Section 4.6.1 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[, forceCSI][, start][, stop][, step])
 |  * read([start][, stop][, step][, field][, coords])
 |  * readCoordinates(coords[, field])
    * readSorted(sortby[, forceCSI][, 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 `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. `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 Section 4.7.1 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 `read()` method (see description) 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 description for a complete list of supported objects for the `Array` object class.

# 3.3. Commiting 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[1].

## 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[2].

[1] Appending data to arrays is also supported, but you need to create special objects called `EArray` (see Section 4.9 for more info).
[2] Note that you can append not only scalar values to tables, but also fully multidimensional array objects.

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 |" % \
         (r['name'], r['pressure'], r['energy'], r['grid_i'], r['grid_j'],
          r['TDCcount'])

Particle:      0 |         0.0 |           0 |      0 |     10 |        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 |
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 Section .

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 `modifyRows()`, there exists another method, called `modifyColumn()` to modify specific columns as well. Please check sections description and description for a more in-depth description of them.

Finally, it exists another way of modifying tables that is generally more handy than the described above. This new way uses the method `update()` (see description) 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 `__setitem__` special method (see description). 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[3] to refer to indexes that you want to modify. See description for more examples on how to use extended slicing in PyTables objects.

Similarly, with and 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[:]
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 the 5th to 9th rows (inclusive):

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

`removeRows(start, stop)` (see description) deletes therows 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()
```

[3]With the sole exception that you cannot use negative values for `step`.

```
>>> ^D
$
```

In Figure 3.2 you can see a graphical view of the PyTables file with the datasets we have just created. In Figure 3.3 are displayed the general properties of the table `/detector/readout`.

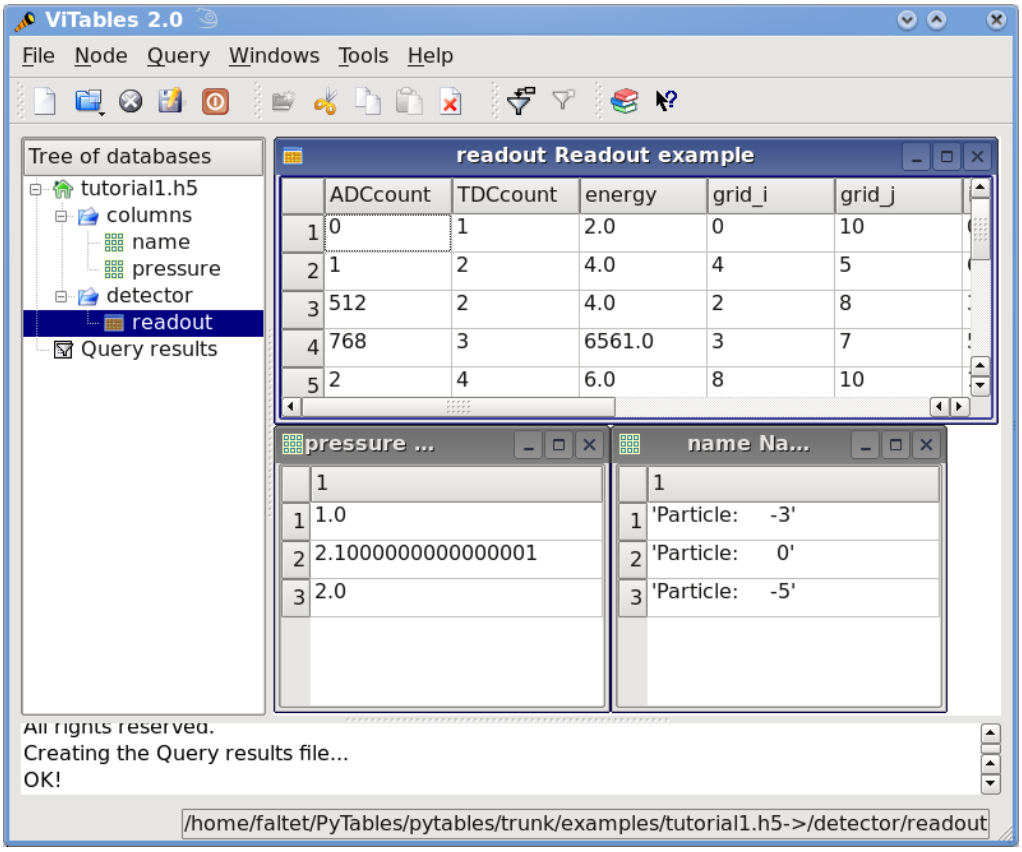


Figure 3.2. The final version of the data file for tutorial 1.

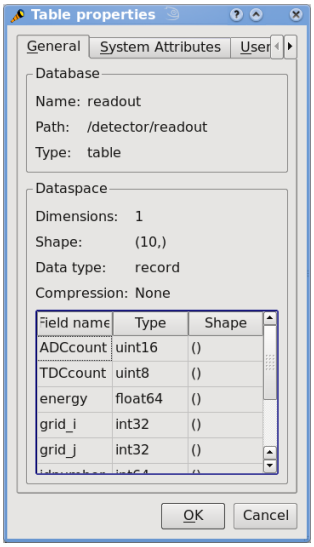


Figure 3.3. General properties of the `/detector/readout` table.

# 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 dictionary, as you can see in the `Event` description. See description about the different kinds of descriptor objects that can be passed to the `createTable()` 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)

# Another way to describe the columns of a table
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 3.4. In particular, note the multidimensional column cells in table `/Particles/TParticle2`.

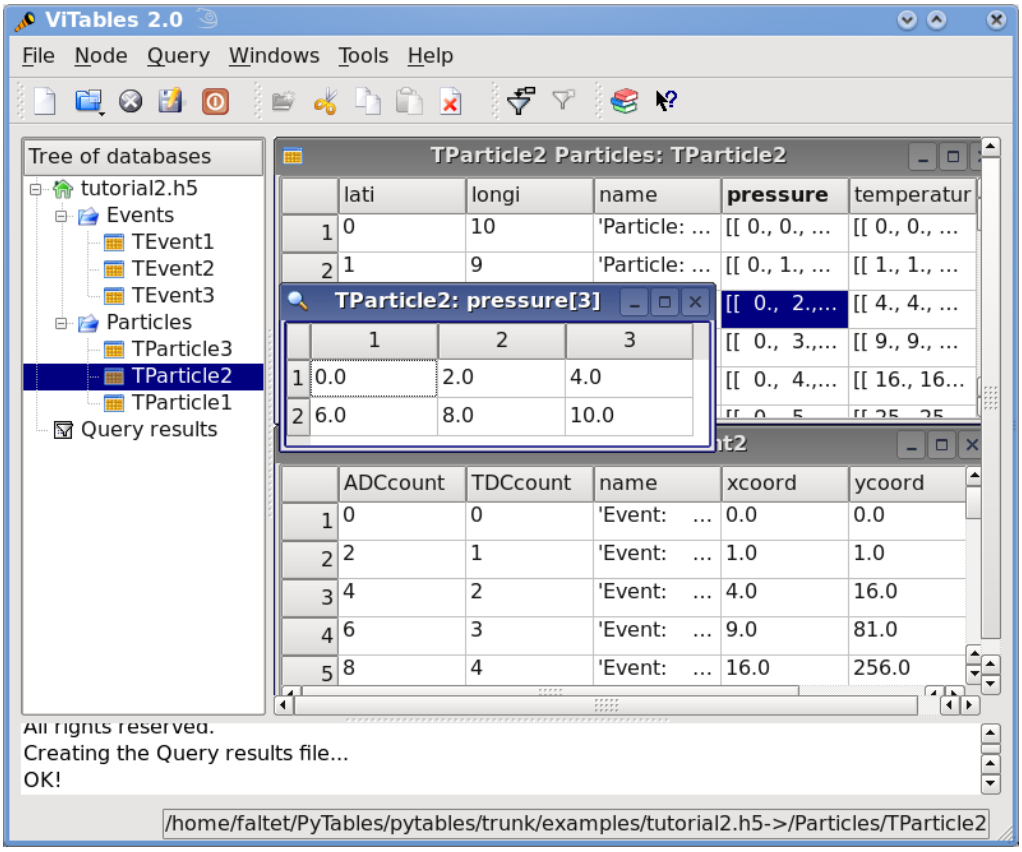


Figure 3.4. Table hierarchy for tutorial 2.

# 3.5. 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 [4].

[4] You can even *hide* nodes temporarily. Will you be able to find out how?

## 3.5.1. 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 `enableUndo` (see description) 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:

```
>>> 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 description 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 description):

```
>>> 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 `getCurrentMark()` method (see description) 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()` (see description) 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.

## 3.5.2. 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 `goto()` method (see description).

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`.

# 3.6. 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 Section 4.14.3) 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[5]:

```
>>> 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
```

[5] All these examples can be found in `examples/enum.py`.

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.

## 3.6.1. Enumerated columns

Columns of an enumerated type can be declared by using the `EnumCol` (see Section 4.13.2) 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[6].

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):
      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)
```

[6] In fact, only integer values are supported right now, but this may change in the future.

```
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 the section called "getEnum(colname)") to get the enumeration back.

## 3.6.2. Enumerated arrays

`EArray` and `VLArray` leaves can also be declared to store enumerated values by means of the `EnumAtom` (see Section 4.13.3) 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 Section 4.9.1) 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()
```

# 3.7. 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`.

## 3.7.1. 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.

## 3.7.2. 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, 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 Appendix D.

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.

## 3.7.3. Using Cols accessor

We can use the `cols` attribute object (see Section 4.6.8) 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.

## 3.7.4. Accessing meta-information of nested tables

Tables have an attribute called `description` which points to an instance of the `Description` class (see Section 4.6.6) 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(itemsize=10, shape=(), dflt='', pos=1),
    "value": Float64Col(shape=(), dflt=0.0, pos=2)},
  "info1": {
    "name": StringCol(itemsize=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(itemsize=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 record 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 Appendix D).

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([('info2', [('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(itemsize=10, shape=(), dflt='', pos=1)
column--> Float64Col(shape=(), dflt=0.0, pos=2)
column--> Description([('name', '()S10'), ('value', '()f8')])
column--> StringCol(itemsize=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 Section 4.6.6 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.

# 3.8. 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.

# Chapter 4. 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`, `VLArray` 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

**__version__** The PyTables version number.

**hdf5Version** The underlying HDF5 library version number.

**is_pro** True for PyTables Professional edition, false otherwise.

### 4.1.2. Global functions

#### copyFile(srcfilename, dstfilename, overwrite=False, **kwargs)

An easy way of copying one PyTables file to another.

This function allows you to copy an existing PyTables file named `srcfilename` to another file called `dstfilename`. The source file must exist and be readable. The destination file can be overwritten in place if existing by asserting the `overwrite` argument.

This function is a shorthand for the `File.copyFile()` method, which acts on an already opened file. `kwargs` takes keyword arguments used to customize the copying process. See the documentation of `File.copyFile()` (see description) for a description of those arguments.

#### isHDF5File(filename)

Determine whether a file is in the HDF5 format.

When successful, it returns a true value if the file is an HDF5 file, false otherwise. If there were problems identifying the file, an `HDF5ExtError` is raised.

#### isPyTablesFile(filename)

Determine whether a file is in the PyTables format.

When successful, it returns a true value if the file is a PyTables file, false otherwise. The true value is the format version string of the file. If there were problems identifying the file, an `HDF5ExtError` is raised.

## lrange([start, ]stop[, step])

Iterate over long ranges.

This is similar to `xrange()`, but it allows 64-bit arguments on all platforms. The results of the iteration are sequentially yielded in the form of `numpy.int64` values, but getting random individual items is not supported.

Because of the Python 32-bit limitation on object lengths, the `length` attribute (which is also a `numpy.int64` value) should be used instead of the `len()` syntax.

Default `start` and `step` arguments are supported in the same way as in `xrange()`. When the standard `[x]range()` Python objects support 64-bit arguments, this iterator will be deprecated.

## openFile(filename, mode='r', title='', rootUEP="/", filters=None, **kwargs)

Open a PyTables (or generic HDF5) file and return a `File` object.

Arguments:

**filename**
The name of the file (supports environment variable expansion). It is suggested that file names have any of the `.h5`, `.hdf` or `.hdf5` extensions, although this is not mandatory.

**mode**
The mode in whichto open the file. It can be one of the following:

**'r'**
Read-only; no data can be modified.

**'w'**
Write; a new file is created (an existing file with the same name would be deleted).

**'a'**
Append; an existing file is opened for reading and writing, and if the file does not exist it is created.

**'r+'**
It is similar to `'a'`, but the file must already exist.

**title**
If the file is to be created, a `TITLE` string attribute will be set on the root group with the given value. Otherwise, the title will be read from disk, and this will not have any effect.

**rootUEP**
The root User Entry Point. This is a group in the HDF5 hierarchy which will be taken as the starting point to create the object tree. It can be whatever existing group in the file, named by its HDF5 path. If it does not exist, an `HDF5ExtError` is issued. Use this if you do not want to build the *entire* object tree, but rather only a *subtree* of it.

**filters**
An instance of the `Filters` (see Section 4.14.1) class that provides information about the desired I/O filters applicable to the leaves that hang directly from the *root group*, unless other filter properties are specified for these leaves. Besides, if you do not specify filter properties for child groups, they will inherit these ones, which will in turn propagate to child nodes.

In addition, it recognizes the names of parameters present in `tables/parameters.py` (and for PyTables Pro users, those in `tables/_parameters_pro.py` too) as additional keyword arguments. See Appendix C for a detailed info on the supported parameters.

**Note**

If you need to deal with a large number of nodes in an efficient way, please see Section 5.5 for more info and advices about the integrated node cache engine.

### print_versions()

Print all the versions of software that PyTables relies on.

### restrict_flavors(keep=['python'])

Disable all flavors except those in `keep`.

Providing an empty `keep` sequence implies disabling all flavors (but the internal one). If the sequence is not specified, only optional flavors are disabled.

**Important**

Once you disable a flavor, it can not be enabled again.

### split_type(type)

Split a PyTables `type` into a PyTables kind and an item size.

Returns a tuple of `(kind, itemsize)`. If no item size is present in the `type` (in the form of a precision), the returned item size is `None`.

```
>>> split_type('int32')
('int', 4)
>>> split_type('string')
('string', None)
>>> split_type('int20')
Traceback (most recent call last):
  ...
ValueError: precision must be a multiple of 8: 20
>>> split_type('foo bar')
Traceback (most recent call last):
  ...
ValueError: malformed type: 'foo bar'
```

### test(verbose=False, heavy=False)

Run all the tests in the test suite.

If `verbose` is set, the test suite will emit messages with full verbosity (not recommended unless you are looking into a certain problem).

If `heavy` is set, the test suite will be run in *heavy* mode (you should be careful with this because it can take a lot of time and resources from your computer).

### whichLibVersion(name)

Get version information about a C library.

If the library indicated by `name` is available, this function returns a 3-tuple containing the major library version as an integer, its full version as a string, and the version date as a string. If the library is not available, `None` is returned.

The currently supported library names are `hdf5`, `zlib`, `lzo` and `bzip2`. If another name is given, a `ValueError` is raised.

# 4.2. The `File` class

In-memory representation of a PyTables file.

An instance of this class is returned when a PyTables file is opened with the `openFile()` (see description) function. It offers methods to manipulate (create, rename, delete...) nodes and handle their attributes, as well as methods to traverse the object tree. The *user entry point* to the object tree attached to the HDF5 file is represented in the `rootUEP` attribute. Other attributes are available.

`File` objects support an *Undo/Redo mechanism* which can be enabled with the `enableUndo()` (see description) method. Once the Undo/Redo mechanism is enabled, explicit *marks* (with an optional unique name) can be set on the state of the database using the `mark()` (see description) method. There are two implicit marks which are always available: the initial mark (0) and the final mark (-1). Both the identifier of a mark and its name can be used in *undo* and *redo* operations.

Hierarchy manipulation operations (node creation, movement and removal) and attribute handling operations (setting and deleting) made after a mark can be undone by using the `undo()` (see description) method, which returns the database to the state of a past mark. If `undo()` is not followed by operations that modify the hierarchy or attributes, the `redo()` (see description) method can be used to return the database to the state of a future mark. Else, future states of the database are forgotten.

Note that data handling operations can not be undone nor redone by now. Also, hierarchy manipulation operations on nodes that do not support the Undo/Redo mechanism issue an `UndoRedoWarning` *before* changing the database.

The Undo/Redo mechanism is persistent between sessions and can only be disabled by calling the `disableUndo()` (see description) method.

File objects can also act as context managers when using the `with` statement introduced in Python 2.5. When exiting a context, the file is automatically closed.

## 4.2.1. `File` instance variables

**filename** The name of the opened file.

**format_version** The PyTables version number of this file.

**isopen** True if the underlying file is open, false otherwise.

**mode** The mode in which the file was opened.

**title** The title of the root group in the file.

**rootUEP** The UEP (user entry point) group name in the file (see the `openFile()` function in description).

**filters** Default filter properties for the root group (see Section 4.14.1).

**root** The *root* of the object tree hierarchy (a `Group` instance).

## 4.2.2. `File` methods — file handling

### close()

Flush all the alive leaves in object tree and close the file.

### copyFile(dstfilename, overwrite=False, **kwargs)

Copy the contents of this file to `dstfilename`.

`dstfilename` must be a path string indicating the name of the destination file. If it already exists, the copy will fail with an `IOError`, unless the `overwrite` argument is true, in which case the destination file will be overwritten in place. In this last case, the destination file should be closed or ugly errors will happen.

Additional keyword arguments may be passed to customize the copying process. For instance, title and filters may be changed, user attributes may be or may not be copied, data may be sub-sampled, stats may be collected, etc. Arguments unknown to nodes are simply ignored. Check the documentation for copying operations of nodes to see which options they support.

Copying a file usually has the beneficial side effect of creating a more compact and cleaner version of the original file.

### flush()

Flush all the alive leaves in the object tree.

### fileno()

Return the underlying OS integer file descriptor.

This is needed for lower-level file interfaces, such as the `fcntl` module.

### __enter__()

Enter a context and return the same file.

### __exit__([*exc_info])

Exit a context and close the file.

### __str__()

Return a short string representation of the object tree.

Example of use:

```
>>> f = tables.openFile('data/test.h5')
>>> print f
data/test.h5 (File) 'Table Benchmark'
Last modif.: 'Mon Sep 20 12:40:47 2004'
Object Tree:
/ (Group) 'Table Benchmark'
/tuple0 (Table(100,)) 'This is the table title'
/group0 (Group) ''
/group0/tuple1 (Table(100,)) 'This is the table title'
/group0/group1 (Group) ''
/group0/group1/tuple2 (Table(100,)) 'This is the table title'
```

```
/group0/group1/group2 (Group) ''
```

### __repr__()

Return a detailed string representation of the object tree.

## 4.2.3. `File` methods — hierarchy manipulation

### copyChildren(srcgroup, dstgroup, overwrite=False, recursive=False, createparents=False, **kwargs)

Copy the children of a group into another group.

This method copies the nodes hanging from the source group `srcgroup` into the destination group `dstgroup`. Existing destination nodes can be replaced by asserting the `overwrite` argument. If the `recursive` argument is true, all descendant nodes of `srcnode` are recursively copied. If `createparents` is true, the needed groups for the given destination parent group path to exist will be created.

`kwargs` takes keyword arguments used to customize the copying process. See the documentation of `Group._f_copyChildren()` (see description) for a description of those arguments.

### copyNode(where, newparent=None, newname=None, name=None, overwrite=False, recursive=False, createparents=False, **kwargs)

Copy the node specified by `where` and `name` to `newparent/newname`.

**where, name**
: These arguments work as in `File.getNode()` (see description), referencing the node to be acted upon.

**newparent**
: The destination group that the node will be copied into (a path name or a `Group` instance). If not specified or `None`, the current parent group is chosen as the new parent.

**newname**
: The name to be assigned to the new copy in its destination (a string). If it is not specified or `None`, the current name is chosen as the new name.

Additional keyword arguments may be passed to customize the copying process. The supported arguments depend on the kind of node being copied. See `Group._f_copy()` (description) and `Leaf.copy()` (description) for more information on their allowed keyword arguments.

This method returns the newly created copy of the source node (i.e. the destination node). See `Node._f_copy()` (description) for further details on the semantics of copying nodes.

### createArray(where, name, object, title='', byteorder=None, createparents=False)

Create a new array with the given `name` in `where` location. See the `Array` class (in Section 4.7) for more information on arrays.

**object**
: The array or scalar to be saved. Accepted types are NumPy arrays and scalars, `numarray` arrays and string arrays, Numeric arrays and scalars, as well as native Python sequences and scalars, provided that values are regular (i.e. they are not like `[[1,2],2]`) and homogeneous (i.e. all the elements are of the same type).

  Also, objects that have some of their dimensions equal to 0 are not supported (use an `EArray` node (see Section 4.9) if you want to store an array with one of its dimensions equal to 0).

**byteorder**
The byteorder of the data *on disk*, specified as `'little'` or `'big'`. If this is not specified, the byteorder is that of the given `object`.

See `File.createTable()` (description) for more information on the rest of parameters.

## createCArray(where, name, atom, shape, title='', filters=None, chunkshape=None, byteorder=None, createparents=False)

Create a new chunked array with the given `name` in `where` location. See the `CArray` class (in Section 4.8) for more information on chunked arrays.

**atom**
An `Atom` (see Section 4.13.3) instance representing the *type* and *shape* of the atomic objects to be saved.

**shape**
The shape of the new array.

**chunkshape**
The shape of the data chunk to be read or written in a single HDF5 I/O operation. Filters are applied to those chunks of data. The dimensionality of `chunkshape` must be the same as that of `shape`. If `None`, a sensible value is calculated (which is recommended).

See `File.createTable()` (description) for more information on the rest of parameters.

## createEArray(where, name, atom, shape, title='', filters=None, expectedrows=EXPECTED_ROWS_EARRAY, chunkshape=None, byteorder=None, createparents=False)

Create a new enlargeable array with the given `name` in `where` location. See the `EArray` (in Section 4.9) class for more information on enlargeable arrays.

**atom**
An `Atom` (see Section 4.13.3) instance representing the *type* and *shape* of the atomic objects to be saved.

**shape**
The shape of the new array. One (and only one) of the shape dimensions *must* be 0. The dimension being 0 means that the resulting `EArray` object can be extended along it. Multiple enlargeable dimensions are not supported right now.

**expectedrows**
A user estimate about the number of row elements that will be added to the growable dimension in the `EArray` node. If not provided, the default value is `EXPECTED_ROWS_EARRAY` (see `tables/parameters.py`). If you plan to create either a much smaller or a much bigger array try providing a guess; this will optimize the HDF5 B-Tree creation and management process time and the amount of memory used.

**chunkshape**
The shape of the data chunk to be read or written in a single HDF5 I/O operation. Filters are applied to those chunks of data. The dimensionality of `chunkshape` must be the same as that of `shape` (beware: no dimension should be 0 this time!). If `None`, a sensible value is calculated based on the `expectedrows` parameter (which is recommended).

**byteorder**
The byteorder of the data *on disk*, specified as `'little'` or `'big'`. If this is not specified, the byteorder is that of the platform.

See `File.createTable()` (description) for more information on the rest of parameters.

### createGroup(where, name, title='', filters=None, createparents=False)

Create a new group with the given `name` in `where` location. See the `Group` class (in Section 4.4) for more information on groups.

**filters**
An instance of the `Filters` class (see Section 4.14.1) that provides information about the desired I/O filters applicable to the leaves that hang directly from this new group (unless other filter properties are specified for these leaves). Besides, if you do not specify filter properties for its child groups, they will inherit these ones.

See `File.createTable()` (description) for more information on the rest of parameters.

### createTable(where, name, description, title='', filters=None, expectedrows=EXPECTED_ROWS_TABLE, chunkshape=None, byteorder=None, createparents=False)

Create a new table with the given `name` in `where` location. See the `Table` (in Section 4.6) class for more information on tables.

**where**
The parent group where the new table will hang from. It can be a path string (for example `'/level1/leaf5'`), or a `Group` instance (see Section 4.4).

**name**
The name of the new table.

**description**

This is an object that describes the table, i.e. how many columns it has, their names, types, shapes, etc. It can be any of the following:

**A user-defined class**
This should inherit from the `IsDescription` class (see Section 4.13.1) where table fields are specified.

**A dictionary**
For example, when you do not know beforehand which structure your table will have).

See Section 3.4 for an example of using a dictionary to describe a table.

**A `Description` instance**
You can use the `description` attribute of another table to create a new one with the same structure.

**A `NumPy (record) array` instance**
You can use a NumPy array, whether nested or not, and its field structure will be reflected in the new `Table` object. Moreover, if the array has actual data it will be injected into the newly created table. If you are using `numarray` instead of NumPy, you may use one of the objects below for the same purpose.

**A `RecArray` instance**
This object from the `numarray` package is also accepted, but it does not give you the possibility to create a nested table. Array data is injected into the new table.

**A `NestedRecArray` instance**
Finally, if you want to have nested columns in your table and you are using `numarray`, you can use this object. Array data is injected into the new table.

See Appendix D for a description of the `NestedRecArray` class.

**title**
A description for this node (it sets the `TITLE` HDF5 attribute on disk).

**filters**
An instance of the `Filters` class (see Section 4.14.1) that provides information about the desired I/O filters to be applied during the life of this object.

**expectedrows**
A user estimate of the number of records that will be in the table. If not provided, the default value is `EXPECTED_ROWS_TABLE` (see `tables/parameters.py`). If you plan to create a bigger table try providing a guess; this will optimize the HDF5 B-Tree creation and management process time and memory used.

See Section 5.1.1 for a discussion on the issue of providing a number of expected rows.

**chunkshape**
The shape of the data chunk to be read or written in a single HDF5 I/O operation. Filters are applied to those chunks of data. The rank of the `chunkshape` for tables must be 1. If `None`, a sensible value is calculated based on the `expectedrows` parameter (which is recommended).

**byteorder**
The byteorder of data *on disk*, specified as `'little'` or `'big'`. If this is not specified, the byteorder is that of the platform, unless you passed an array as the `description`, in which case its byteorder will be used.

**createparents**
Whether to create the needed groups for the parent path to exist (not done by default).

## createVLArray(where, name, atom, title='', filters=None, expectedsizeinMB=1.0, chunkshape=None, byteorder=None, createparents=False)

Create a new variable-length array with the given `name` in `where` location. See the `VLArray` (in Section 4.10) class for more information on variable-length arrays.

**atom**
An `Atom` (see Section 4.13.3) instance representing the *type* and *shape* of the atomic objects to be saved.

**expectedsizeinMB**
An user estimate about the size (in MB) in the final `VLArray` node. If not provided, the default value is 1 MB. If you plan to create either a much smaller or a much bigger array try providing a guess; this will optimize the HDF5 B-Tree creation and management process time and the amount of memory used. If you want to specify your own chunk size for I/O purposes, see also the `chunkshape` parameter below.

**chunkshape**
The shape of the data chunk to be read or written in a single HDF5 I/O operation. Filters are applied to those chunks of data. The dimensionality of `chunkshape` must be 1. If `None`, a sensible value is calculated (which is recommended).

See `File.createTable()` (description) for more information on the rest of parameters.

## moveNode(where, newparent=None, newname=None, name=None, overwrite=False, createparents=False)

Move the node specified by `where` and `name` to `newparent/newname`.

**where, name**
These arguments work as in `File.getNode()` (see description), referencing the node to be acted upon.

**newparent**
The destination group the node will be moved into (a path name or a `Group` instance). If it is not specified or `None`, the current parent group is chosen as the new parent.

**newname**
The new name to be assigned to the node in its destination (a string). If it is not specified or `None`, the current name is chosen as the new name.

The other arguments work as in `Node._f_move()` (see description).

### removeNode(where, name=None, recursive=False)

Remove the object node *name* under *where* location.

**where, name**
These arguments work as in `File.getNode()` (see description), referencing the node to be acted upon.

**recursive**
If not supplied or false, the node will be removed only if it has no children; if it does, a `NodeError` will be raised. If supplied with a true value, the node and all its descendants will be completely removed.

### renameNode(where, newname, name=None, overwrite=False)

Change the name of the node specified by `where` and `name` to `newname`.

**where, name**
These arguments work as in `File.getNode()` (see description), referencing the node to be acted upon.

**newname**
The new name to be assigned to the node (a string).

**overwrite**
Whether to recursively remove a node with the same `newname` if it already exists (not done by default).

## 4.2.4. `File` methods — tree traversal

### getNode(where, name=None, classname=None)

Get the node under `where` with the given `name`.

`where` can be a `Node` instance (see Section 4.3) or a path string leading to a node. If no `name` is specified, that node is returned.

If a `name` is specified, this must be a string with the name of a node under `where`. In this case the `where` argument can only lead to a `Group` (see Section 4.4) instance (else a `TypeError` is raised). The node called `name` under the group `where` is returned.

In both cases, if the node to be returned does not exist, a `NoSuchNodeError` is raised. Please note that hidden nodes are also considered.

If the `classname` argument is specified, it must be the name of a class derived from `Node`. If the node is found but it is not an instance of that class, a `NoSuchNodeError` is also raised.

### isVisibleNode(path)

Is the node under `path` visible?

If the node does not exist, a `NoSuchNodeError` is raised.

### iterNodes(where, classname=None)

Iterate over children nodes hanging from `where`.

**where**
This argument works as in `File.getNode()` (see description), referencing the node to be acted upon.

**classname**
If the name of a class derived from `Node` (see Section 4.3) is supplied, only instances of that class (or subclasses of it) will be returned.

The returned nodes are alphanumerically sorted by their name. This is an iterator version of `File.listNodes()` (see description).

### listNodes(where, classname=None)

Return a *list* with children nodes hanging from `where`.

This is a list-returning version of `File.iterNodes()` (see description).

### walkGroups(where='/')

Recursively iterate over groups (not leaves) hanging from `where`.

The `where` group itself is listed first (preorder), then each of its child groups (following an alphanumerical order) is also traversed, following the same procedure. If `where` is not supplied, the root group is used.

The `where` argument can be a path string or a `Group` instance (see Section 4.4).

### walkNodes(where="/", classname="")

Recursively iterate over nodes hanging from `where`.

**where**
If supplied, the iteration starts from (and includes) this group. It can be a path string or a `Group` instance (see Section 4.4).

**classname**
If the name of a class derived from `Node` (see Section 4.4) is supplied, only instances of that class (or subclasses of it) will be returned.

Example of use:

```
# Recursively print all the nodes hanging from '/detector'.
print "Nodes hanging from group '/detector':"
for node in h5file.walkNodes('/detector', classname='EArray'):
    print node
```

### __contains__(path)

Is there a node with that `path`?

Returns `True` if the file has a node with the given `path` (a string), `False` otherwise.

### __iter__()

Recursively iterate over the nodes in the tree.

This is equivalent to calling `File.walkNodes()` (see description) with no arguments.

Example of use:

```
# Recursively list all the nodes in the object tree.
h5file = tables.openFile('vlarray1.h5')
print "All nodes in the object tree:"
for node in h5file:
    print node
```

## 4.2.5. `File` methods — Undo/Redo support

### disableUndo()

Disable the Undo/Redo mechanism.

Disabling the Undo/Redo mechanism leaves the database in the current state and forgets past and future database states. This makes `File.mark()` (see description), `File.undo()` (see description), `File.redo()` (see description) and other methods fail with an `UndoRedoError`.

Calling this method when the Undo/Redo mechanism is already disabled raises an `UndoRedoError`.

### enableUndo(filters=Filters( complevel=1))

Enable the Undo/Redo mechanism.

This operation prepares the database for undoing and redoing modifications in the node hierarchy. This allows `File.mark()` (see description), `File.undo()` (see description), `File.redo()` (see description) and other methods to be called.

The `filters` argument, when specified, must be an instance of class `Filters` (see Section 4.14.1) and is meant for setting the compression values for the action log. The default is having compression enabled, as the gains in terms of space can be considerable. You may want to disable compression if you want maximum speed for Undo/Redo operations.

Calling this method when the Undo/Redo mechanism is already enabled raises an `UndoRedoError`.

### getCurrentMark()

Get the identifier of the current mark.

Returns the identifier of the current mark. This can be used to know the state of a database after an application crash, or to get the identifier of the initial implicit mark after a call to `File.enableUndo()` (see description).

This method can only be called when the Undo/Redo mechanism has been enabled. Otherwise, an `UndoRedoError` is raised.

### goto(mark)

Go to a specific mark of the database.

Returns the database to the state associated with the specified `mark`. Both the identifier of a mark and its name can be used.

This method can only be called when the Undo/Redo mechanism has been enabled. Otherwise, an `UndoRedoError` is raised.

### isUndoEnabled()

Is the Undo/Redo mechanism enabled?

Returns `True` if the Undo/Redo mechanism has been enabled for this file, `False` otherwise. Please note that this mechanism is persistent, so a newly opened PyTables file may already have Undo/Redo support enabled.

### mark(name=None)

Mark the state of the database.

Creates a mark for the current state of the database. A unique (and immutable) identifier for the mark is returned. An optional `name` (a string) can be assigned to the mark. Both the identifier of a mark and its name can be used in `File.undo()` (see description) and `File.redo()` (see description) operations. When the `name` has already been used for another mark, an `UndoRedoError` is raised.

This method can only be called when the Undo/Redo mechanism has been enabled. Otherwise, an `UndoRedoError` is raised.

### redo(mark=None)

Go to a future state of the database.

Returns the database to the state associated with the specified `mark`. Both the identifier of a mark and its name can be used. If the `mark` is omitted, the next created mark is used. If there are no future marks, or the specified `mark` is not newer than the current one, an `UndoRedoError` is raised.

This method can only be called when the Undo/Redo mechanism has been enabled. Otherwise, an `UndoRedoError` is raised.

### undo(mark=None)

Go to a past state of the database.

Returns the database to the state associated with the specified `mark`. Both the identifier of a mark and its name can be used. If the `mark` is omitted, the last created mark is used. If there are no past marks, or the specified `mark` is not older than the current one, an `UndoRedoError` is raised.

This method can only be called when the Undo/Redo mechanism has been enabled. Otherwise, an `UndoRedoError` is raised.

## 4.2.6. `File` methods — atttribute handling

### copyNodeAttrs(where, dstnode, name=None)

Copy PyTables attributes from one node to another.

**where, name**
These arguments work as in `File.getNode()` (see description), referencing the node to be acted upon.

**dstnode**
The destination node where the attributes will be copied to. It can be a path string or a `Node` instance (see Section 4.3).

### delNodeAttr(where, attrname, name=None)

Delete a PyTables attribute from the given node.

**where, name**
These arguments work as in `File.getNode()` (see description), referencing the node to be acted upon.

**attrname**
The name of the attribute to delete. If the named attribute does not exist, an `AttributeError` is raised.

### getNodeAttr(where, attrname, name=None)

Get a PyTables attribute from the given node.

**where, name**
These arguments work as in `File.getNode()` (see description), referencing the node to be acted upon.

**attrname**
The name of the attribute to retrieve. If the named attribute does not exist, an `AttributeError` is raised.

### setNodeAttr(where, attrname, attrvalue, name=None)

Set a PyTables attribute for the given node.

**where, name**
These arguments work as in `File.getNode()` (see description), referencing the node to be acted upon.

**attrname**
The name of the attribute to set.

**attrvalue**
The value of the attribute to set. Any kind of Python object (like strings, ints, floats, lists, tuples, dicts, small NumPy/Numeric/numarray objects...) can be stored as an attribute. However, if necessary, `cPickle` is automatically used so as to serialize objects that you might want to save. See the `AttributeSet` class (in Section 4.12) for details.

If the node already has a large number of attributes, a `PerformanceWarning` is issued.

# 4.3. The Node class

Abstract base class for all PyTables nodes.

This is the base class for *all* nodes in a PyTables hierarchy. It is an abstract class, i.e. it may not be directly instantiated; however, every node in the hierarchy is an instance of this class.

A PyTables node is always hosted in a PyTables *file*, under a *parent group*, at a certain *depth* in the node hierarchy. A node knows its own *name* in the parent group and its own *path name* in the file.

All the previous information is location-dependent, i.e. it may change when moving or renaming a node in the hierarchy. A node also has location-independent information, such as its *HDF5 object identifier* and its *attribute set*.

This class gathers the operations and attributes (both location-dependent and independent) which are common to all PyTables nodes, whatever their type is. Nonetheless, due to natural naming restrictions, the names of all of these members start with a reserved prefix (see the `Group` class in Section 4.4).

Sub-classes with no children (i.e. *leaf nodes*) may define new methods, attributes and properties to avoid natural naming restrictions. For instance, `_v_attrs` may be shortened to `attrs` and `_f_rename` to `rename`. However, the original methods and attributes should still be available.

## 4.3.1. Node instance variables — location dependent

**_v_depth** The depth of this node in the tree (an non-negative integer value).

**_v_file** The hosting `File` instance (see Section 4.2).

**_v_name** The name of this node in its parent group (a string).

**_v_parent** The parent `Group` instance (see Section 4.4).

**_v_pathname** The path of this node in the tree (a string).

## 4.3.2. Node instance variables — location independent

**_v_attrs** The associated `AttributeSet` instance (see Section 4.12).

**_v_isopen** Whether this node is open or not.

**_v_objectID** A node identifier (may change from run to run).

## 4.3.3. Node instance variables — attribute shorthands

**_v_title** A description of this node. A shorthand for `TITLE` attribute.

## 4.3.4. Node methods — hierarchy manipulation

### _f_close()

Close this node in the tree.

This releases all resources held by the node, so it should not be used again. On nodes with data, it may be flushed to disk.

You should not need to close nodes manually because they are automatically opened/closed when they are loaded/evicted from the integrated LRU cache.

### _f_copy(newparent=None, newname=None, overwrite=False, recursive=False, createparents=False, **kwargs)

Copy this node and return the new node.

Creates and returns a copy of the node, maybe in a different place in the hierarchy. `newparent` can be a `Group` object (see Section 4.4) or a pathname in string form. If it is not specified or `None`, the current parent group is chosen as the new parent. `newname` must be a string with a new name. If it is not specified or `None`, the current name is chosen as the new name. If `recursive` copy is stated, all descendants are copied as well. If `createparents` is true, the needed groups for the given new parent group path to exist will be created.

Copying a node across databases is supported but can not be undone. Copying a node over itself is not allowed, nor it is recursively copying a node into itself. These result in a `NodeError`. Copying over another existing node is similarly not allowed, unless the optional `overwrite` argument is true, in which case that node is recursively removed before copying.

Additional keyword arguments may be passed to customize the copying process. For instance, title and filters may be changed, user attributes may be or may not be copied, data may be sub-sampled, stats may be collected, etc. See the documentation for the particular node type.

Using only the first argument is equivalent to copying the node to a new location without changing its name. Using only the second argument is equivalent to making a copy of the node in the same group.

### _f_isVisible()

Is this node visible?

### _f_move(newparent=None, newname=None, overwrite=False, createparents=False)

Move or rename this node.

Moves a node into a new parent group, or changes the name of the node. `newparent` can be a `Group` object (see Section 4.4) or a pathname in string form. If it is not specified or `None`, the current parent group is chosen as the new parent. `newname` must be a string with a new name. If it is not specified or `None`, the current name is chosen as the new name. If `createparents` is true, the needed groups for the given new parent group path to exist will be created.

Moving a node across databases is not allowed, nor it is moving a node *into* itself. These result in a `NodeError`. However, moving a node *over* itself is allowed and simply does nothing. Moving over another existing node is similarly not allowed, unless the optional `overwrite` argument is true, in which case that node is recursively removed before moving.

Usually, only the first argument will be used, effectively moving the node to a new location without changing its name. Using only the second argument is equivalent to renaming the node in place.

### _f_remove(recursive=False)

Remove this node from the hierarchy.

If the node has children, recursive removal must be stated by giving `recursive` a true value; otherwise, a `NodeError` will be raised.

### _f_rename(newname, overwrite=False)

Rename this node in place.

Changes the name of a node to *newname* (a string). If a node with the same `newname` already exists and `overwrite` is true, recursively remove it before renaming.

## 4.3.5. Node methods — attribute handling

### _f_delAttr(name)

Delete a PyTables attribute from this node.

If the named attribute does not exist, an `AttributeError` is raised.

### _f_getAttr(name)

Get a PyTables attribute from this node.

If the named attribute does not exist, an `AttributeError` is raised.

### _f_setAttr(name, value)

Set a PyTables attribute for this node.

If the node already has a large number of attributes, a `PerformanceWarning` is issued.

# 4.4. The Group class

Basic PyTables grouping structure.

Instances of this class are grouping structures containing *child* instances of zero or more groups or leaves, together with supporting metadata. Each group has exactly one *parent* group.

Working with groups and leaves is similar in many ways to working with directories and files, respectively, in a Unix filesystem. As with Unix directories and files, objects in the object tree are often described by giving their full (or absolute) path names. This full path can be specified either as a string (like in `'/group1/group2'`) or as a complete object path written in *natural naming* schema (like in `file.root.group1.group2`). See Section 1.2 for more information on natural naming.

A collateral effect of the *natural naming* schema is that the names of members in the `Group` class and its instances must be carefully chosen to avoid colliding with existing children node names. For this reason and to avoid polluting the children namespace all members in a `Group` start with some reserved prefix, like `_f_` (for public methods), `_g_` (for private ones), `_v_` (for instance variables) or `_c_` (for class variables). Any attempt to create a new child node whose name starts with one of these prefixes will raise a `ValueError` exception.

Another effect of natural naming is that children named after Python keywords or having names not valid as Python identifiers (e.g. `class`, `$a` or `44`) can not be accessed using the `node.child` syntax. You will be forced to use `node._f_getChild(child)` to access them (which is recommended for programmatic accesses).

You will also need to use `_f_getChild()` to access an existing child node if you set a Python attribute in the `Group` with the same name as that node (you will get a `NaturalNameWarning` when doing this).

## 4.4.1. `Group` instance variables

The following instance variables are provided in addition to those in `Node` (see Section 4.3):

**_v_nchildren** — The number of children hanging from this group.

**_v_filters** — Default filter properties for child nodes.

You can (and are encouraged to) use this property to get, set and delete the `FILTERS` HDF5 attribute of the group, which stores a `Filters` instance (see Section 4.14.1). When the group has no such attribute, a default `Filters` instance is used.

**_v_groups** — Dictionary with all groups hanging from this group.

**_v_hidden** — Dictionary with all hidden nodes hanging from this group.

**_v_leaves** — Dictionary with all leaves hanging from this group.

**_v_children** — Dictionary with all nodes hanging from this group.

## 4.4.2. `Group` methods

*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 Section 4.2, 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 Section 4.3):

### _f_close()

Close this group and all its descendents.

This method has the behavior described in `Node._f_close()` (see description). It should be noted that this operation closes all the nodes descending from this group.

You should not need to close nodes manually because they are automatically opened/closed when they are loaded/evicted from the integrated LRU cache.

## _f_copy(newparent, newname, overwrite=False, recursive=False, createparents=False, **kwargs)

Copy this node and return the new one.

This method has the behavior described in `Node._f_copy()` (see description). In addition, it recognizes the following keyword arguments:

**title**
The new title for the destination. If omitted or `None`, the original title is used. This only applies to the topmost node in recursive copies.

**filters**
Specifying this parameter overrides the original filter properties in the source node. If specified, it must be an instance of the `Filters` class (see Section 4.14.1). The default is to copy the filter properties from the source node.

**copyuserattrs**
You can prevent the user attributes from being copied by setting this parameter to `False`. The default is to copy them.

**stats**
This argument may be used to collect statistics on the copy process. When used, it should be a dictionary with keys `'groups'`, `'leaves'` and `'bytes'` having a numeric value. Their values will be incremented to reflect the number of groups, leaves and bytes, respectively, that have been copied during the operation.

## _f_copyChildren(dstgroup, overwrite=False, recursive=False, createparents=False, **kwargs)

Copy the children of this group into another group.

Children hanging directly from this group are copied into `dstgroup`, which can be a `Group` (see Section 4.4) object or its pathname in string form. If `createparents` is true, the needed groups for the given destination group path to exist will be created.

The operation will fail with a `NodeError` if there is a child node in the destination group with the same name as one of the copied children from this one, unless `overwrite` is true; in this case, the former child node is recursively removed before copying the later.

By default, nodes descending from children groups of this node are not copied. If the `recursive` argument is true, all descendant nodes of this node are recursively copied.

Additional keyword arguments may be passed to customize the copying process. For instance, title and filters may be changed, user attributes may be or may not be copied, data may be sub-sampled, stats may be collected, etc. Arguments unknown to nodes are simply ignored. Check the documentation for copying operations of nodes to see which options they support.

## _f_getChild(childname)

Get the child called `childname` of this group.

If the child exists (be it visible or not), it is returned. Else, a `NoSuchNodeError` is raised.

Using this method is recommended over `getattr()` when doing programmatic accesses to children if `childname` is unknown beforehand or when its name is not a valid Python identifier.

### _f_iterNodes(classname=None)

Iterate over children nodes.

Child nodes are yielded alphanumerically sorted by node name. If the name of a class derived from `Node` (see Section 4.3) is supplied in the `classname` parameter, only instances of that class (or subclasses of it) will be returned.

This is an iterator version of `Group._f_listNodes()` (see description).

### _f_listNodes(classname=None)

Return a *list* with children nodes.

This is a list-returning version of `Group._f_iterNodes()` (see description).

### _f_walkGroups()

Recursively iterate over descendent groups (not leaves).

This method starts by yielding *self*, and then it goes on to recursively iterate over all child groups in alphanumerical order, top to bottom (preorder), following the same procedure.

### _f_walkNodes(classname=None)

Iterate over descendent nodes.

This method recursively walks *self* top to bottom (preorder), iterating over child groups in alphanumerical order, and yielding nodes. If `classname` is supplied, only instances of the named class are yielded.

If *classname* is `Group`, it behaves like *Group._f_walkGroups()* (see the section called "_f_walkGroups()"), yielding only groups. If you don't want a recursive behavior, use *Group._f_iterNodes()* (see description) instead.

Example of use:

```
# Recursively print all the arrays hanging from '/'
print "Arrays in the object tree '/':"
for array in h5file.root._f_walkNodes('Array', recursive=True):
    print array
```

## 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:

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

# 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
```

### __contains__(name)

Is there a child with that `name`?

Returns a true value if the group has a child node (visible or hidden) with the given *name* (a string), false otherwise.

### __delattr__(name)

Delete a Python attribute called `name`.

This method deletes an *ordinary Python attribute* from the object. It does *not* remove children nodes from this group; for that, use `File.removeNode()` (see description) or `Node._f_remove()` (see description). It does *neither* delete a PyTables node attribute; for that, use `File.delNodeAttr()` (see description), `Node._f_delAttr()` (see description) or `Node._v_attrs` (see Section 4.3.2).

If there is an attribute and a child node with the same `name`, the child node will be made accessible again via natural naming.

### __getattr__(name)

Get a Python attribute or child node called `name`.

If the object has a Python attribute called `name`, its value is returned. Else, if the node has a child node called `name`, it is returned. Else, an `AttributeError` is raised.

### __iter__()

Iterate over the child nodes hanging directly from the group.

This iterator is *not* recursive. Example of use:

```
# Non-recursively list all the nodes hanging from '/detector'
print "Nodes in '/detector' group:"
for node in h5file.root.detector:
    print node
```

### __repr__()

Return a detailed string representation of the group.

Example of use:

```
>>> f = tables.openFile('data/test.h5')
>>> f.root.group0
/group0 (Group) 'First Group'
  children := ['tuple1' (Table), 'group1' (Group)]
```

### __setattr__(name, value)

Set a Python attribute called `name` with the given `value`.

This method stores an *ordinary Python attribute* in the object. It does *not* store new children nodes under this group; for that, use the `File.create*()` methods (see the `File` class in Section 4.2). It does *neither* store a PyTables node attribute; for that, use `File.setNodeAttr()` (see description), `Node._f_setAttr()` (see description) or `Node._v_attrs` (see Section 4.3.2).

If there is already a child node with the same `name`, a `NaturalNameWarning` will be issued and the child node will not be accessible via natural naming nor `getattr()`. It will still be available via `File.getNode()` (see description), `Group._f_getChild()` (see description) and children dictionaries in the group (if visible).

### __str__()

Return a short string representation of the group.

Example of use:

```
>>> f=tables.openFile('data/test.h5')
>>> print f.root.group0
/group0 (Group) 'First Group'
```

# 4.5. The `Leaf` class

Abstract base class for all PyTables leaves.

A leaf is a node (see the `Node` class in Section 4.3) which hangs from a group (see the `Group` class in Section 4.4) but, unlike a group, it can not have any further children below it (i.e. it is an end node).

This definition includes all nodes which contain actual data (datasets handled by the `Table` —see Section 4.6—, `Array` —see Section 4.7—, `CArray` —see Section 4.8—, `EArray` —see Section 4.9— and `VLArray` —see Section 4.10— classes) and unsupported nodes (the `UnImplemented` class —Section 4.11) —these classes do in fact inherit from `Leaf`.

## 4.5.1. `Leaf` instance variables

These instance variables are provided in addition to those in `Node` (see Section 4.3):

**byteorder** The byte ordering of the leaf data *on disk*.

**chunkshape** The HDF5 chunk size for chunked leaves (a tuple).

This is read-only because you cannot change the chunk size of a leaf once it has been created.

**extdim** The index of the enlargeable dimension (-1 if none).

**filters** Filter properties for this leaf —see `Filters` in Section 4.14.1.

**flavor** The type of data object read from this leaf.

It can be any of `'numpy'`, `'numarray'`, `'numeric'` or `'python'` (the set of supported flavors depends on which packages you have installed on your system).

You can (and are encouraged to) use this property to get, set and delete the `FLAVOR` HDF5 attribute of the leaf. When the leaf has no such attribute, the default flavor is used.

**maindim** The dimension along which iterators work.

Its value is 0 (i.e. the first dimension) when the dataset is not extendable, and `self.extdim` (where available) for extendable ones.

**nrows** The length of the main dimension of the leaf data.

**nrowsinbuf** The number of rows that fit in internal input buffers.

You can change this to fine-tune the speed or memory requirements of your application.

**shape** The shape of data in the leaf.

## 4.5.2. `Leaf` instance variables — aliases

The following are just easier-to-write aliases to their `Node` (see Section 4.3) counterparts (indicated between parentheses):

**attrs** The associated `AttributeSet` instance —see Section 4.12— (`Node._v_attrs`).

**name** The name of this node in its parent group (`Node._v_name`).

**objectID** A node identifier (may change from run to run). (`Node._v_objectID`).

**title** A description for this node (`Node._v_title`).

## 4.5.3. `Leaf` methods

### close(flush=True)

Close this node in the tree.

This method is completely equivalent to `Leaf._f_close()` (see description).

### copy(newparent, newname, overwrite=False, createparents=False, **kwargs)

Copy this node and return the new one.

This method has the behavior described in `Node._f_copy()` (see description). Please note that there is no `recursive` flag since leaves do not have child nodes. In addition, this method recognizes the following keyword arguments:

**title**
The new title for the destination. If omitted or `None`, the original title is used.

**filters**
Specifying this parameter overrides the original filter properties in the source node. If specified, it must be an instance of the `Filters` class (see Section 4.14.1). The default is to copy the filter properties from the source node.

**copyuserattrs**
You can prevent the user attributes from being copied by setting this parameter to `False`. The default is to copy them.

**start, stop, step**
Specify the range of rows to be copied; the default is to copy all the rows.

**stats**
This argument may be used to collect statistics on the copy process. When used, it should be a dictionary with keys `'groups'`, `'leaves'` and `'bytes'` having a numeric value. Their values will be incremented to reflect the number of groups, leaves and bytes, respectively, that have been copied during the operation.

**chunkshape**
The chunkshape of the new leaf. It supports a couple of special values. A value of `keep` means that the chunkshape will be the same than original leaf (this is the default). A value of `auto` means that a new shape will be computed automatically in order to ensure best performance when accessing the dataset through the main dimension. Any other value should be an integer or a tuple matching the dimensions of the leaf.

**Warning**

Note that unknown parameters passed to this method will be ignored, so may want to double check the spell of these (i.e. if you write them incorrectly, they will most probably be ignored).

## delAttr(name)

Delete a PyTables attribute from this node.

This method has the behavior described in `Node._f_delAttr()` (see description).

## flush()

Flush pending data to disk.

Saves whatever remaining buffered data to disk. It also releases I/O buffers, so if you are filling many datasets in the same PyTables session, please call `flush()` extensively so as to help PyTables to keep memory requirements low.

## getAttr(name)

Get a PyTables attribute from this node.

This method has the behavior described in `Node._f_getAttr()` (see description).

## isVisible()

Is this node visible?

This method has the behavior described in `Node._f_isVisible()` (see description).

## move(newparent=None, newname=None, overwrite=False, createparents=False)

Move or rename this node.

This method has the behavior described in `Node._f_move()` (see description).

## rename(newname)

Rename this node in place.

This method has the behavior described in `Node._f_rename()` (see description).

## remove()

Remove this node from the hierarchy.

This method has the behavior described in `Node._f_remove()` (see description). Please note that there is no `recursive` flag since leaves do not have child nodes.

### setAttr(name, value)

Set a PyTables attribute for this node.

This method has the behavior described in `Node._f_setAttr()` (see description).

### truncate(size)

Truncate the main dimension to be `size` rows.

If the main dimension previously was larger than this `size`, the extra data is lost. If the main dimension previously was shorter, it is extended, and the extended part is filled with the default values.

The truncation operation can only be applied to *enlargeable* datasets, else a `TypeError` will be raised.

**Warning**

If you are using the HDF5 1.6.x series, and due to limitations of them, `size` must be greater than zero (i.e. the dataset can not be completely emptied). A `ValueError` will be issued if you are using HDF5 1.6.x and try to pass a zero size to this method. Also, HDF5 1.6.x has the problem that it cannot work against `CArray` objects (again, a `ValueError` will be issued). HDF5 1.8.x doesn't undergo these problems.

### __len__()

Return the length of the main dimension of the leaf data.

Please note that this may raise an `OverflowError` on 32-bit platforms for datasets having more than 2**31-1 rows. This is a limitation of Python that you can work around by using the `nrows` or `shape` attributes.

### _f_close(flush=True)

Close this node in the tree.

This method has the behavior described in `Node._f_close()` (see description). Besides that, the optional argument `flush` tells whether to flush pending data to disk or not before closing.

## 4.6. The `Table` class

This class represents heterogeneous datasets in an HDF5 file.

Tables are leaves (see the `Leaf` class in Section 4.5) 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*.

Fields can have any type supported by the `Col` class (see Section 4.13.2) and its descendants, which support multidimensional data. Moreover, a field can be *nested* (to an arbitrary depth), meaning that it includes further fields inside. A field named `x` inside a nested field `a` in a table can be accessed as the field `a/x` (its *path name*) from the table.

The structure of a table is declared by its description, which is made available in the `Table.description` attribute (see Section 4.6.1).

This class provides new methods to read, write and search table data efficiently. It also provides special Python methods to allow accessing the table as a normal sequence or array (with extended slicing supported).

PyTables supports *in-kernel* searches working simultaneously on several columns using complex conditions. These are faster than selections using Python expressions. See the `Tables.where()` method —description— for more information on in-kernel searches. See also Section 5.2.1 for a detailed review of the advantages and shortcomings of in-kernel searches.

Non-nested columns can be *indexed*. Searching an indexed column can be several times faster than searching a non-nested one. Search methods automatically take advantage of indexing where available.

**Note**

Column indexing is only available in PyTables Pro.

When iterating a table, an object from the `Row` (see Section 4.6.7) class is used. This object allows to read and write data one row at a time, as well as to perform queries which are not supported by in-kernel syntax (at a much lower speed, of course). See the tutorial sections in Chapter 3 on how to use the `Row` interface.

Objects of this class support access to individual columns via *natural naming* through the `Table.cols` accessor (see Section 4.6.1). Nested columns are mapped to `Cols` instances, and non-nested ones to `Column` instances. See the `Column` class in Section 4.6.9 for examples of this feature.

## 4.6.1. `Table` instance variables

The following instance variables are provided in addition to those in `Leaf` (see Section 4.5). 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.

This value gets into effect whenever a column is altered. If you don't have automatic indexing activated and you want to do an immediate update use `Table.flushRowsToIndex()` (see the section called "flushRowsToIndex()"); for immediate reindexing of invalidated indexes, use `Table.reIndexDirty()` (see the section called "reIndexDirty()").

This value is persistent.

**Note**

Column indexing is only available in PyTables Pro.

**coldescrs**

Maps the name of a column to its `Col` description (see Section 4.13.2).

**coldflts**

Maps the name of a column to its default value.

**coldtypes**

Maps the name of a column to its NumPy data type.

**colindexed**

Is the column which name is used as a key indexed?

**Note**

Column indexing is only available in PyTables Pro.

**colindexes** A dictionary with the indexes of the indexed columns.

**colinstances** Maps the name of a column to its `Column` (see Section 4.6.9) or `Cols` (see Section 4.6.8) instance.

**colnames** A list containing the names of *top-level* columns in the table.

**colpathnames** A list containing the pathnames of *bottom-level* columns in the table.

These are the leaf columns obtained when walking the table description left-to-right, bottom-first. Columns inside a nested column have slashes (`/`) separating name components in their pathname.

**cols** A `Cols` instance that provides *natural naming* access to non-nested (`Column`, see Section 4.6.9) and nested (`Cols`, see Section 4.6.8) columns.

**coltypes** Maps the name of a column to its PyTables data type.

**description** A `Description` instance (see Section 4.6.6) reflecting the structure of the table.

**extdim** The index of the enlargeable dimension (always 0 for tables).

**indexed** Does this table have any indexed columns?

**indexedcolpathnames** List of the pathnames of indexed columns in the table.

**nrows** The current number of rows in the table.

**row** The associated `Row` instance (see Section 4.6.7).

**rowsize** The size in bytes of each row in the table.

## 4.6.2. `Table` methods — reading

### col(name)

Get a column from the table.

If a column called `name` exists in the table, it is read and returned as a NumPy object, or as a `numarray` object (depending on the flavor of the table). If it does not exist, a `KeyError` is raised.

Example of use:

```
narray = table.col('var2')
```

That statement is equivalent to:

```
narray = table.read(field='var2')
```

Here you can see how this method can be used as a shorthand for the `Table.read()` method (see description).

### iterrows(start=None, stop=None, step=None)

Iterate over the table using a `Row` instance (see Section 4.6.7).

If a range is not supplied, *all the rows* in the table are iterated upon —you can also use the `Table.__iter__()` special method (see description) for that purpose. If you want to iterate over a given *range of rows* in the table, you may use the `start`, `stop` and `step` parameters, which have the same meaning as in `Table.read()` (see description).

Example of use:

```
result = [ row['var2'] for row in table.iterrows(step=5)
           if row['var1'] <= 20 ]
```

**Note**

This iterator can be nested (see `Table.where()` —description— for an example).

**Warning**

When in the middle of a table row iterator, you should not use methods that can change the number of rows in the table (like `Table.append()` or `Table.removeRows()`) or unexpected errors will happen.

### itersequence(sequence)

Iterate over a `sequence` of row coordinates.

**Note**

This iterator can be nested (see `Table.where()` —description— for an example).

### itersorted(sortby, forceCSI=False, start=None, stop=None, step=None)

Iterate over the table data sorted by the given `sortby` column.

`sortby` column must have associated a completely sorted index (CSI) so as to ensure a fully sorted order. You can use the `forceCSI` argument in order to force the creation of a CSI index in case that one does not exist yet.

The meaning of the `start`, `stop` and `step` arguments is the same as in `Table.read()` (see description). However, in this case a negative value of `step` is supported, meaning that the results will be returned in reverse sorted order.


**Note**

Column indexing is only available in PyTables Pro.

### read(start=None, stop=None, step=None, field=None)

Get data in the table as a (record) array.

The `start`, `stop` and `step` parameters can be used to select only a *range of rows* in the table. Their meanings are the same as in the built-in `range()` Python function, except that negative values of `step` are not allowed yet. Moreover, if only `start` is specified, then `stop` will be set to `start+1`. If you do not specify neither `start` nor `stop`, then *all the rows* in the table are selected.

If `field` is supplied only the named column will be selected. If the column is not nested, an *array* of the current flavor will be returned; if it is, a *record array* will be used instead. I no `field` is specified, all the columns will be returned in a record array of the current flavor. More specifically, when the flavor is `'numarray'` and a record array is needed, a `NestedRecArray` (see Appendix D) will be returned.

Columns under a nested column can be specified in the `field` parameter by using a slash character (`/`) as a separator (e.g. `'position/x'`).

### readCoordinates(coords, field=None)

Get a set of rows given their indexes as a (record) array.

This method works much like the `read()` method (see description), but it uses a sequence (`coords`) of row indexes to select the wanted columns, instead of a column range.

The selected rows are returned in an array or record array of the current flavor.

### readSorted(sortby, forceCSI=False, field=None, start=None, stop=None, step=None)

Read table data sorted by the given `sortby` column.

`sortby` column must have associated a completely sorted index (CSI) so as to ensure a fully sorted order. You can use the `forceCSI` argument in order to force the creation of a CSI index in case that one does not exist yet.

If `field` is supplied only the named column will be selected. If the column is not nested, an *array* of the current flavor will be returned; if it is, a *record array* will be used instead. If no `field` is specified, all the columns will be returned in a record array of the current flavor.

The meaning of the `start`, `stop` and `step` arguments is the same as in `Table.read()` (see description). However, in this case a negative value of `step` is supported, meaning that the results will be returned in reverse sorted order.

**Note**

Column indexing is only available in PyTables Pro.

### __getitem__(key)

Get a row or a range of rows from the table.

If `key` argument is an integer, the corresponding table row is returned as a record of the current flavor. If `key` is a slice, the range of rows determined by it is returned as a record array of the current flavor.

Example of use:

```
record = table[4]
recarray = table[4:1000:2]
```

Those statements are equivalent to:

```
record = table.read(start=4)[0]
recarray = table.read(start=4, stop=1000, step=2)
```

Here you can see how indexing and slicing can be used as shorthands for the `read()` (see description) method.

### __iter__()

Iterate over the table using a `Row` instance (see Section 4.6.7).

This is equivalent to calling `Table.iterrows()` (see description) with default arguments, i.e. it iterates over *all the rows* in the table.

Example of use:

```
result = [ row['var2'] for row in table
           if row['var1'] <= 20 ]
```

Which is equivalent to:

```
result = [ row['var2'] for row in table.iterrows()
           if row['var1'] <= 20 ]
```

**Note**

This iterator can be nested (see `Table.where()` —description— for an example).

## 4.6.3. `Table` methods — writing

### append(rows)

Append a sequence of `rows` to the end of the table.

The `rows` argument may be any object which can be converted to a record array compliant with the table structure (otherwise, a `ValueError` is raised). This includes NumPy record arrays, `RecArray` or `NestedRecArray` objects if `numarray` is available, lists of tuples or array records, and a string or Python buffer.

Example of use:

```
from tables import *
class Particle(IsDescription):
    name        = StringCol(16, pos=1) # 16-character String
    lati        = IntCol(pos=2)        # integer
    longi       = IntCol(pos=3)        # integer
    pressure    = Float32Col(pos=4)    # float  (single-precision)
    temperature = FloatCol(pos=5)      # double (double-precision)

fileh = openFile('test4.h5', mode='w')
table = fileh.createTable(fileh.root, 'table', Particle, "A table")
# Append several rows in only one call
table.append([("Particle:     10", 10, 0, 10*10, 10**2),
              ("Particle:     11", 11, -1, 11*11, 11**2),
              ("Particle:     12", 12, -2, 12*12, 12**2)])
fileh.close()
```

See Appendix D if you are using `numarray` and you want to append data to nested columns.

### modifyColumn(start=None, stop=None, step=1, column=None, colname=None)

Modify one single column in the row slice `[start:stop:step]`.

The `colname` argument specifies the name of the column in the table to be modified with the data given in `column`. This method returns the number of rows modified. Should the modification exceed the length of the table, an `IndexError` is raised before changing data.

The `column` argument may be any object which can be converted to a (record) array compliant with the structure of the column to be modified (otherwise, a `ValueError` is raised). This includes NumPy (record) arrays, `NumArray`, `RecArray` or `NestedRecArray` objects if `numarray` is available, Numeric arrays if available, lists of scalars, tuples or array records, and a string or Python buffer.

See Appendix D if you are using `numarray` and you want to modify data in a nested column.

## modifyColumns(start=None, stop=None, step=1, columns=None, names=None)

Modify a series of columns in the row slice `[start:stop:step]`.

The `names` argument specifies the names of the columns in the table to be modified with the data given in `columns`. This method returns the number of rows modified. Should the modification exceed the length of the table, an `IndexError` is raised before changing data.

The `columns` argument may be any object which can be converted to a record array compliant with the structure of the columns to be modified (otherwise, a `ValueError` is raised). This includes NumPy record arrays, `RecArray` or `NestedRecArray` objects if `numarray` is available, lists of tuples or array records, and a string or Python buffer.

See Appendix D if you are using `numarray` and you want to modify data in nested columns.

## modifyRows(start=None, stop=None, step=1, rows=None)

Modify a series of rows in the slice `[start:stop:step]`.

The values in the selected rows will be modified with the data given in `rows`. This method returns the number of rows modified. Should the modification exceed the length of the table, an `IndexError` is raised before changing data.

The possible values for the `rows` argument are the same as in `Table.append()` (see description).

See Appendix D if you are using `numarray` and you want to modify data in nested columns.

## removeRows(start, stop=None)

Remove a range of rows in the table.

If only `start` is supplied, only this row is to be deleted. If a range is supplied, i.e. both the `start` and `stop` parameters are passed, all the rows in the range are removed. A `step` parameter is not supported, and it is not foreseen to be implemented anytime soon.

**start**
: Sets the starting row to be removed. It accepts negative values meaning that the count starts from the end. A value of 0 means the first row.

**stop**
: Sets the last row to be removed to `stop-1`, i.e. the end point is omitted (in the Python `range()` tradition). Negative values are also accepted. A special value of `None` (the default) means removing just the row supplied in `start`.

## __setitem__(key, value)

Set a row or a range of rows in the table.

It takes different actions depending on the type of the `key` parameter: if it is an integer, the corresponding table row is set to `value` (a record or sequence capable of being converted to the table structure). If `key` is a slice, the row slice determined by it is set to `value` (a record array or sequence capable of being converted to the table structure).

Example of use:

```
# Modify just one existing row
table[2] = [456,'db2',1.2]
# Modify two existing rows
```

```
rows = numpy.rec.array([[457,'db1',1.2],[6,'de2',1.3]],
                       formats='i4,a3,f8')
table[1:3:2] = rows
```

Which is equivalent to:

```
table.modifyRows(start=2, rows=[456,'db2',1.2])
rows = numpy.rec.array([[457,'db1',1.2],[6,'de2',1.3]],
                       formats='i4,a3,f8')
table.modifyRows(start=1, stop=3, step=2, rows=rows)
```

See Appendix D if you are using `numarray` and you want to modify data in nested columns.

## 4.6.4. `Table` methods — querying

### getWhereList(condition, condvars=None, sort=False, start=None, stop=None, step=None)

Get the row coordinates fulfilling the given `condition`.

The coordinates are returned as a list of the current flavor. `sort` means that you want to retrieve the coordinates ordered. The default is to not sort them.

The meaning of the other arguments is the same as in the `Table.where()` method (see description).

### readWhere(condition, condvars=None, field=None, start=None, stop=None, step=None)

Read table data fulfilling the given *condition*.

This method is similar to `Table.read()` (see description), having their common arguments and return values the same meanings. However, only the rows fulfilling the *condition* are included in the result.

The meaning of the other arguments is the same as in the `Table.where()` method (see description).

### where(condition, condvars=None, start=None, stop=None, step=None)

Iterate over values fulfilling a `condition`.

This method returns a `Row` iterator (see Section 4.6.7) which only selects rows in the table that satisfy the given `condition` (an expression-like string). For more information on condition syntax, see Appendix B.

The `condvars` mapping may be used to define the variable names appearing in the `condition`. `condvars` should consist of identifier-like strings pointing to `Column` (see Section 4.6.9) instances *of this table*, or to other values (which will be converted to arrays). A default set of condition variables is provided where each top-level, non-nested column with an identifier-like name appears. Variables in `condvars` override the default ones.

When `condvars` is not provided or `None`, the current local and global namespace is sought instead of `condvars`. The previous mechanism is mostly intended for interactive usage. To disable it, just specify a (maybe empty) mapping as `condvars`.

If a range is supplied (by setting some of the `start`, `stop` or `step` parameters), only the rows in that range `and` fulfilling the `condition` are used. The meaning of the `start`, `stop` and `step` parameters is the same as in the `range()` Python function, except that negative values of `step` are `not` allowed. Moreover, if only `start` is specified, then `stop` will be set to `start+1`.

When possible, indexed columns participating in the condition will be used to speed up the search. It is recommended that you place the indexed columns as left and out in the condition as possible. Anyway, this method has always better performance than regular Python selections on the table. Please check the Section 5.2 for more information about the performance of the different searching modes.

### Note

Column indexing is only available in PyTables Pro.

You can mix this method with regular Python selections in order to support even more complex queries. It is strongly recommended that you pass the most restrictive condition as the parameter to this method if you want to achieve maximum performance.

Example of use:

```
>>> passvalues = [ row['col3'] for row in
...                table.where('(col1 > 0) & (col2 <= 20)', step=5)
...                if your_function(row['col2']) ]
>>> print "Values that pass the cuts:", passvalues
```

Note that, from PyTables 1.1 on, you can nest several iterators over the same table. For example:

```
for p in rout.where('pressure < 16'):
    for q in rout.where('pressure < 9'):
        for n in rout.where('energy < 10'):
            print "pressure, energy:", p['pressure'], n['energy']
```

In this example, iterators returned by `Table.where()` have been used, but you may as well use any of the other reading iterators that `Table` objects offer. See the file `examples/nested-iter.py` for the full code.

### Warning

When in the middle of a table row iterator, you should not use methods that can change the number of rows in the table (like `Table.append()` or `Table.removeRows()`) or unexpected errors will happen.

## whereAppend(dstTable, condition, condvars=None, start=None, stop=None, step=None)

Append rows fulfilling the `condition` to the `dstTable` table.

`dstTable` must be capable of taking the rows resulting from the query, i.e. it must have columns with the expected names and compatible types. The meaning of the other arguments is the same as in the `Table.where()` method (see description).

The number of rows appended to `dstTable` is returned as a result.

## willQueryUseIndexing(condition, condvars=None)

Will a query for the `condition` use indexing?

The meaning of the `condition` and *condvars* arguments is the same as in the `Table.where()` method (see description). If `condition` can use indexing, this method returns a frozenset with the path names of the columns whose index is usable. Otherwise, it returns an empty list.

This method is mainly intended for testing. Keep in mind that changing the set of indexed columns or their dirtyness may make this method return different values for the same arguments at different times.

**Note**

Column indexing is only available in PyTables Pro.

## 4.6.5. `Table` methods — other

### copy(newparent=None, newname=None, overwrite=False, createparents=False, **kwargs)

Copy this table and return the new one.

This method has the behavior and keywords described in `Leaf.copy()` (see description. Moreover, it recognises the next additional keyword arguments:

**sortby**
If specified, and `sortby` corresponds to a column with a completely sorted index (CSI), then the copy will be sorted by the values on this column. A reverse sorted copy can be achieved by specifying a negative value for the `step` keyword. If omitted or `None`, the original table order is used.

**forceCSI**
If true, and a CSI index does not exist for the `sortby` column, one will be built prior to method execution. If false, the CSI creation will not be forced (this may cause the raise of an error). In case a CSI index already exists for the `sortby` column, this parameter does nothing.

**propindexes**
If true, the existing indexes in the source table are propagated (created) to the new one. If false (the default), the indexes are not propagated.

### flushRowsToIndex()

Add remaining rows in buffers to non-dirty indexes.

This can be useful when you have chosen non-automatic indexing for the table (see the `Table.autoIndex` property in Section 4.6.1) and you want to update the indexes on it.

**Note**

Column indexing is only available in PyTables Pro.

### getEnum(colname)

Get the enumerated type associated with the named column.

If the column named `colname` (a string) exists and is of an enumerated type, the corresponding `Enum` instance (see Section 4.14.3) is returned. If it is not of an enumerated type, a `TypeError` is raised. If the column does not exist, a `KeyError` is raised.

### reIndex()

Recompute all the existing indexes in the table.

This can be useful when you suspect that, for any reason, the index information for columns is no longer valid and want to rebuild the indexes on it.

**Note**

Column indexing is only available in PyTables Pro.

### reIndexDirty()

Recompute the existing indexes in table, *if* they are dirty.

This can be useful when you have set `Table.autoIndex` (see Section 4.6.1) to false for the table and you want to update the indexes after a invalidating index operation (`Table.removeRows()`, for example).

**Note**

Column indexing is only available in PyTables Pro.

## 4.6.6. The `Description` class

This class represents descriptions of the structure of tables.

An instance of this class is automatically bound to `Table` (see Section 4.6) objects when they are created. It provides a browseable representation of the structure of the table, made of non-nested (`Col` —see Section 4.13.2) and nested (`Description`) columns. It also contains information that will allow you to build `NestedRecArray` (see Appendix D) objects suited for the different columns in a table (be they nested or not).

Column definitions under a description can be accessed as attributes of it (*natural naming*). For instance, if `table.description` is a `Description` instance with a column named `col1` under it, the later can be accessed as `table.description.col1`. If `col1` is nested and contains a `col2` column, this can be accessed as `table.description.col1.col2`. Because of natural naming, the names of members start with special prefixes, like in the `Group` class (see Section 4.4).

### `Description` instance variables

**_v_colObjects** — A dictionary mapping the names of the columns hanging directly from the associated table or nested column to their respective descriptions (`Col` —see Section 4.13.2— or `Description` —see Section 4.6.6— instances).

**_v_dflts** — A dictionary mapping the names of non-nested columns hanging directly from the associated table or nested column to their respective default values.

**_v_dtype** — The NumPy type which reflects the structure of this table or nested column. You can use this as the `dtype` argument of NumPy array factories.

**_v_dtypes** — A dictionary mapping the names of non-nested columns hanging directly from the associated table or nested column to their respective NumPy types.

**_v_is_nested** — Whether the associated table or nested column contains further nested columns or not.

**_v_itemsize** — The size in bytes of an item in this table or nested column.

**_v_name** — The name of this description group. The name of the root group is `'/'`.

**_v_names** — A list of the names of the columns hanging directly from the associated table or nested column. The order of the names matches the order of their respective columns in the containing table.

**_v_nestedDescr** A nested list of pairs of `(name, format)` tuples for all the columns under this table or nested column. You can use this as the `dtype` and `descr` arguments of NumPy array and `NestedRecArray` (see Appendix D) factories, respectively.

**_v_nestedFormats** A nested list of the NumPy string formats (and shapes) of all the columns under this table or nested column. You can use this as the `formats` argument of NumPy array and `NestedRecArray` (see Appendix D) factories.

**_v_nestedlvl** The level of the associated table or nested column in the nested datatype.

**_v_nestedNames** A nested list of the names of all the columns under this table or nested column. You can use this as the `names` argument of NumPy array and `NestedRecArray` (see Appendix D) factories.

**_v_pathnames** A list of the pathnames of all the columns under this table or nested column (in preorder). If it does not contain nested columns, this is exactly the same as the `Description._v_names` attribute.

**_v_types** A dictionary mapping the names of non-nested columns hanging directly from the associated table or nested column to their respective PyTables types.

### `Description` methods

#### _f_walk(type='All')

Iterate over nested columns.

If `type` is `'All'` (the default), all column description objects (`Col` and `Description` instances) are yielded in top-to-bottom order (preorder).

If `type` is `'Col'` or `'Description'`, only column descriptions of that type are yielded.

## 4.6.7. The `Row` class

Table row iterator and field accessor.

Instances of this class are used to fetch and set the values of individual table fields. It works very much like a dictionary, where keys are the pathnames or positions (extended slicing is supported) of the fields in the associated table in a specific row.

This class provides an *iterator interface* so that you can use the same `Row` instance to access successive table rows one after the other. There are also some important methods that are useful for acessing, adding and modifying values in tables.

### `Row` instance variables

**nrow**
The current row number.

This poperty is useful for knowing which row is being dealt with in the middle of a loop or iterator.

### `Row` methods

#### append()

Add a new row of data to the end of the dataset.

Once you have filled the proper fields for the current row, calling this method actually appends the new data to the *output buffer* (which will eventually be dumped to disk). If you have not set the value of a field, the default value of the column will be used.

Example of use:

```
row = table.row
for i in xrange(nrows):
    row['col1'] = i-1
    row['col2'] = 'a'
    row['col3'] = -1.0
    row.append()
table.flush()
```

**Warning**

After completion of the loop in which `Row.append()` has been called, it is always convenient to make a call to `Table.flush()` in order to avoid losing the last rows that may still remain in internal buffers.

### fetch_all_fields()

Retrieve all the fields in the current row.

Contrarily to `row[:]` (see the section called "Row special methods"), this returns row data as a NumPy void scalar. For instance:

```
[row.fetch_all_fields() for row in table.where('col1 < 3')]
```

will select all the rows that fullfill the given condition as a list of NumPy records.

### update()

Change the data of the current row in the dataset.

This method allows you to modify values in a table when you are in the middle of a table iterator like `Table.iterrows()` (see description) or `Table.where()` (see description).

Once you have filled the proper fields for the current row, calling this method actually changes data in the *output buffer* (which will eventually be dumped to disk). If you have not set the value of a field, its original value will be used.

Examples of use:

```
for row in table.iterrows(step=10):
    row['col1'] = row.nrow
    row['col2'] = 'b'
    row['col3'] = 0.0
    row.update()
table.flush()
```

which modifies every tenth row in table. Or:

```
for row in table.where('col1 > 3'):
    row['col1'] = row.nrow
    row['col2'] = 'b'
    row['col3'] = 0.0
```

```
    row.update()
table.flush()
```

which just updates the rows with values bigger than 3 in the first column.

**Warning**

After completion of the loop in which `Row.update()` has been called, it is always convenient to make a call to `Table.flush()` in order to avoid losing changed rows that may still remain in internal buffers.

## Row special methods

### __getitem__(key)

Get the row field specified by the `key`.

The `key` can be a string (the name of the field), an integer (the position of the field) or a slice (the range of field positions). When `key` is a slice, the returned value is a *tuple* containing the values of the specified fields.

Examples of use:

```
res = [row['var3'] for row in table.where('var2 < 20')]
```

which selects the `var3` field for all the rows that fullfill the condition. Or:

```
res = [row[4] for row in table if row[1] < 20]
```

which selects the field in the *4th* position for all the rows that fullfill the condition. Or:

```
res = [row[:] for row in table if row['var2'] < 20]
```

which selects the all the fields (in the form of a *tuple*) for all the rows that fullfill the condition. Or:

```
res = [row[1::2] for row in table.iterrows(2, 3000, 3)]
```

which selects all the fields in even positions (in the form of a *tuple*) for all the rows in the slice `[2:3000:3]`.

### __setitem__(key, value)

Set the `key` row field to the specified `value`.

Differently from its `__getitem__()` counterpart, in this case `key` can only be a string (the name of the field). The changes done via `__setitem__()` will not take effect on the data on disk until any of the `Row.append()` (see description) or `Row.update()` (see description) methods are called.

Example of use:

```
for row in table.iterrows(step=10):
    row['col1'] = row.nrow
    row['col2'] = 'b'
    row['col3'] = 0.0
    row.update()
table.flush()
```

which modifies every tenth row in the table.

## 4.6.8. The `Cols` class

Container for columns in a table or nested column.

This class is used as an *accessor* to the columns in a table or nested column. It supports the *natural naming* convention, so that you can access the different columns as attributes which lead to `Column` instances (for non-nested columns) or other `Cols` instances (for nested columns).

For instance, if `table.cols` is a `Cols` instance with a column named `col1` under it, the later can be accessed as `table.cols.col1`. If `col1` is nested and contains a `col2` column, this can be accessed as `table.cols.col1.col2` and so on. Because of natural naming, the names of members start with special prefixes, like in the `Group` class (see Section 4.4).

Like the `Column` class (see Section 4.6.9), `Cols` supports item access to read and write ranges of values in the table or nested column.

### `Cols` instance variables

**_v_colnames** A list of the names of the columns hanging directly from the associated table or nested column. The order of the names matches the order of their respective columns in the containing table.

**_v_colpathnames** A list of the pathnames of all the columns under the associated table or nested column (in preorder). If it does not contain nested columns, this is exactly the same as the `Cols._v_colnames` attribute.

**_v_desc** The associated `Description` instance (see Section 4.6.6).

**_v_table** The parent `Table` instance (see Section 4.6).

### `Cols` methods

#### _f_col(colname)

Get an accessor to the column `colname`.

This method returns a `Column` instance (see Section 4.6.9) if the requested column is not nested, and a `Cols` instance (see Section 4.6.8) if it is. You may use full column pathnames in `colname`.

Calling `cols._f_col('col1/col2')` is equivalent to using `cols.col1.col2`. However, the first syntax is more intended for programmatic use. It is also better if you want to access columns with names that are not valid Python identifiers.

#### __getitem__(key)

Get a row or a range of rows from a table or nested column.

If `key` argument is an integer, the corresponding nested type row is returned as a record of the current flavor. If `key` is a slice, the range of rows determined by it is returned as a record array of the current flavor.

Example of use:

```
record = table.cols[4]  # equivalent to table[4]
recarray = table.cols.Info[4:1000:2]
```

Those statements are equivalent to:

```
nrecord = table.read(start=4)[0]
nrecarray = table.read(start=4, stop=1000, step=2).field('Info')
```

Here you can see how a mix of natural naming, indexing and slicing can be used as shorthands for the `Table.read()` (see description) method.

### __len__()

Get the number of elements in the column.

This matches the length in rows of the parent table.

### __setitem__(key)

Set a row or a range of rows in a table or nested column.

If `key` argument is an integer, the corresponding row is set to `value`. If `key` is a slice, the range of rows determined by it is set to `value`.

Example of use:

```
table.cols[4] = record
table.cols.Info[4:1000:2] = recarray
```

Those statements are equivalent to:

```
table.modifyRows(4, rows=record)
table.modifyColumn(4, 1000, 2, colname='Info', column=recarray)
```

Here you can see how a mix of natural naming, indexing and slicing can be used as shorthands for the `Table.modifyRows()` (see description) and `Table.modifyColumn()` (see description) methods.

## 4.6.9. The `Column` class

Accessor for a non-nested column in a table.

Each instance of this class is associated with one *non-nested* column of a table. These instances are mainly used to read and write data from the table columns using item access (like the `Cols` class —see Section 4.6.8), but there are a few other associated methods to deal with indexes.

**Note**

Column indexing is only available in PyTables Pro.

### `Column` instance variables

**descr** The `Description` (see Section 4.6.6) instance of the parent table or nested column.

**dtype** The NumPy `dtype` that most closely matches this column.

**index** The `Index` instance (see Section 4.14.2) associated with this column (`None` if the column is not indexed).

**is_indexed** True if the column is indexed, false otherwise.

**name** The name of the associated column.

**pathname** The complete pathname of the associated column (the same as `Column.name` if the column is not inside a nested column).

**table** The parent `Table` instance (see Section 4.6).

**type** The PyTables type of the column (a string).

## `Column` methods

### `createIndex(optlevel=6, kind="medium", filters=None, tmp_dir=None)`

Create an index for this column.

Keyword arguments:

**optlevel**
The optimization level for building the index. The levels ranges from 0 (no optimization) up to 9 (maximum optimization). Higher levels of optimization mean better chances for reducing the entropy of the index at the price of using more CPU, memory and I/O resources for creating the index.

**kind**
The kind of the index to be built. It can take the `'ultralight'`, `'light'`, `'medium'` or `'full'` values. Lighter kinds (`'ultralight'` and `'light'`) mean that the index takes less space on disk, but will perform queries slower. Heavier kinds (`'medium'` and `'full'`) mean better chances for reducing the entropy of the index (increasing the query speed) at the price of using more disk space as well as more CPU, memory and I/O resources for creating the index.

Note that selecting a `full` kind with an `optlevel` of 9 (the maximum) guarantees the creation of an index with zero entropy, that is, a completely sorted index (CSI) — provided that the number of rows in the table does not exceed the 2**48 figure (that is more than 100 trillions of rows). See `Column.createCSIndex()` (description) method for a more direct way to create a CSI index.

**filters**
Specify the `Filters` instance used to compress the index. If `None`, default index filters will be used (currently, zlib level 1 with shuffling).

**tmp_dir**
When `kind` is other than `'ultralight'`, a temporary file is created during the index build process. You can use the `tmp_dir` argument to specify the directory for this temporary file. The default is to create it in the same directory as the file containing the original table.


**Note**

Column indexing is only available in PyTables Pro.

### `createCSIndex(filters=None, tmp_dir=None)`

Create a completely sorted index (CSI) for this column.

This method guarantees the creation of an index with zero entropy, that is, a completely sorted index (CSI) -- provided that the number of rows in the table does not exceed the 2**48 figure (that is more than 100 trillions of rows). A CSI index is needed for some table methods (like `Table.itersorted()` or `Table.readSorted()`) in order to ensure completely sorted results.

For the meaning of `filters` and `tmp_dir` arguments see `Column.createIndex()` (description).

**Note**

This method is equivalent to ``Column.createIndex(kind='full', optlevel=9, ...).

### `reIndex()`

Recompute the index associated with this column.

This can be useful when you suspect that, for any reason, the index information is no longer valid and you want to rebuild it.

This method does nothing if the column is not indexed.

**Note**

Column indexing is only available in PyTables Pro.

### `reIndexDirty()`

Recompute the associated index only if it is dirty.

This can be useful when you have set `Table.autoIndex` (see Section 4.6.1) to false for the table and you want to update the column's index after an invalidating index operation (like. `Table.removeRows()` —see description).

This method does nothing if the column is not indexed.

**Note**

Column indexing is only available in PyTables Pro.

### `removeIndex()`

Remove the index associated with this column.

This method does nothing if the column is not indexed. The removed index can be created again by calling the `Column.createIndex()` method (see description).

**Note**

Column indexing is only available in PyTables Pro.

## `Column` special methods

### `__getitem__(key)`

Get a row or a range of rows from a column.

If `key` argument is an integer, the corresponding element in the column is returned as an object of the current flavor. If `key` is a slice, the range of elements determined by it is returned as an array of the current flavor.

Example of use:

```
print "Column handlers:"
for name in table.colnames:
    print table.cols._f_col(name)
```

```
print "Select table.cols.name[1]-->", table.cols.name[1]
print "Select table.cols.name[1:2]-->", table.cols.name[1:2]
print "Select table.cols.name[:]-->", table.cols.name[:]
print "Select table.cols._f_col('name')[:]-->", table.cols._f_col('name')[:]
```

The output of this for a certain arbitrary table is:

```
Column handlers:
/table.cols.name (Column(), string, idx=None)
/table.cols.lati (Column(), int32, idx=None)
/table.cols.longi (Column(), int32, idx=None)
/table.cols.vector (Column(2,), int32, idx=None)
/table.cols.matrix2D (Column(2, 2), float64, idx=None)
Select table.cols.name[1]--> Particle:     11
Select table.cols.name[1:2]--> ['Particle:     11']
Select table.cols.name[:]--> ['Particle:     10'
 'Particle:     11' 'Particle:     12'
 'Particle:     13' 'Particle:     14']
Select table.cols._f_col('name')[:]--> ['Particle:     10'
 'Particle:     11' 'Particle:     12'
 'Particle:     13' 'Particle:     14']
```

See the `examples/table2.py` file for a more complete example.

### __len__()

Get the number of elements in the column.

This matches the length in rows of the parent table.

### __setitem__(key, value)

Set a row or a range of rows in a column.

If `key` argument is an integer, the corresponding element is set to `value`. If `key` is a slice, the range of elements determined by it is set to `value`.

Example of use:

```
# Modify row 1
table.cols.col1[1] = -1
# Modify rows 1 and 3
table.cols.col1[1::2] = [2,3]
```

Which is equivalent to:

```
# Modify row 1
table.modifyColumns(start=1, columns=[[-1]], names=['col1'])
# Modify rows 1 and 3
columns = numpy.rec.fromarrays([[2,3]], formats='i4')
table.modifyColumns(start=1, step=2, columns=columns, names=['col1'])
```

# 4.7. The `Array` class

This class represents homogeneous datasets in an HDF5 file.

This class provides methods to write or read data to or from array objects in the file. This class does not allow you neither to enlarge nor compress the datasets on disk; use the `EArray` class (see Section 4.9) if you want enlargeable dataset support or compression features, or `CArray` (see Section 4.8) if you just want compression.

An interesting property of the `Array` class is that it remembers the *flavor* of the object that has been saved so that if you saved, for example, a `list`, you will get a `list` during readings afterwards; if you saved a NumPy array, you will get a NumPy object, and so forth.

Note that this class inherits all the public attributes and methods that `Leaf` (see Section 4.5) already provides. However, as `Array` instances have no internal I/O buffers, it is not necessary to use the `flush()` method they inherit from `Leaf` in order to save their internal state to disk. When a writing method call returns, all the data is already on disk.

## 4.7.1. `Array` instance variables

**atom** — An `Atom` (see Section 4.13.3) instance representing the *type* and *shape* of the atomic objects to be saved.

**rowsize** — The size of the rows in dimensions orthogonal to *maindim*.

**nrow** — On iterators, this is the index of the current row.

## 4.7.2. `Array` methods

### getEnum()

Get the enumerated type associated with this array.

If this array is of an enumerated type, the corresponding `Enum` instance (see Section 4.14.3) is returned. If it is not of an enumerated type, a `TypeError` is raised.

### iterrows(start=None, stop=None, step=None)

Iterate over the rows of the array.

This method returns an iterator yielding an object of the current flavor for each selected row in the array. The returned rows are taken from the *main dimension*.

If a range is not supplied, *all the rows* in the array are iterated upon —you can also use the `Array.__iter__()` special method (see description) for that purpose. If you only want to iterate over a given *range of rows* in the array, you may use the `start`, `stop` and `step` parameters, which have the same meaning as in `Array.read()` (see description).

Example of use:

```
result = [row for row in arrayInstance.iterrows(step=4)]
```

### next()

Get the next element of the array during an iteration.

The element is returned as an object of the current flavor.

### read(start=None, stop=None, step=None)

Get data in the array as an object of the current flavor.

The `start`, `stop` and `step` parameters can be used to select only a *range of rows* in the array. Their meanings are the same as in the built-in `range()` Python function, except that negative values of `step` are not allowed yet. Moreover, if only `start` is specified, then `stop` will be set to `start+1`. If you do not specify neither `start` nor `stop`, then *all the rows* in the array are selected.

## 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)))`).

### __getitem__(key)

Get a row, a range of rows or a slice from the array.

The set of tokens allowed for the `key` is the same as that for extended slicing in Python (including the `Ellipsis` or `...` token). The result is an object of the current flavor; its shape depends on the kind of slice used as `key` and the shape of the array itself.

Example of use:

```
array1 = array[4]  # array1.shape == array.shape[1:]
array2 = array[4:1000:2]  # len(array2.shape) == len(array.shape)
array3 = array[::2, 1:4, :]
array4 = array[1, ..., ::2, 1:4, 4:]  # general slice selection
```

### __iter__()

Iterate over the rows of the array.

This is equivalent to calling `Array.iterrows()` (see description) with default arguments, i.e. it iterates over *all the rows* in the array.

Example of use:

```
result = [row[2] for row in array]
```

Which is equivalent to:

```
result = [row[2] for row in array.iterrows()]
```

### __setitem__(key, value)

Set a row, a range of rows or a slice in the array.

It takes different actions depending on the type of the `key` parameter: if it is an integer, the corresponding array row is set to `value` (the value is broadcast when needed). If `key` is a slice, the row slice determined by it is set to `value` (as usual, if the slice to be updated exceeds the actual shape of the array, only the values in the existing range are updated).

If `value` is a multidimensional object, then its shape must be compatible with the shape determined by `key`, otherwise, a `ValueError` will be raised.

Example of use:

```
a1[0] = 333        # assign an integer to a Integer Array row
a2[0] = 'b'        # assign a string to a string Array row
```

```
a3[1:4] = 5        # broadcast 5 to slice 1:4
a4[1:4:2] = 'xXx'  # broadcast 'xXx' to slice 1:4:2
# General slice update (a5.shape = (4,3,2,8,5,10).
a5[1, ..., ::2, 1:4, 4:] = arange(1728, shape=(4,3,2,4,3,6))
```

# 4.8. The CArray class

This class represents homogeneous datasets in an HDF5 file.

The difference between a `CArray` and a normal `Array` (see Section 4.7), from which it inherits, is that a `CArray` has a chunked layout and, as a consequence, it supports compression. You can use datasets of this class to easily save or load arrays to or from disk, with compression support included.

## 4.8.1. Example of use

See below a small example of the use of the `CArray` class. The code is available in `examples/carray1.py`:

```
import numpy
import tables

fileName = 'carray1.h5'
shape = (200, 300)
atom = tables.UInt8Atom()
filters = tables.Filters(complevel=5, complib='zlib')

h5f = tables.openFile(fileName, 'w')
ca = h5f.createCArray(h5f.root, 'carray', atom, shape, filters=filters)
# Fill a hyperslab in ``ca``.
ca[10:60, 20:70] = numpy.ones((50, 50))
h5f.close()

# Re-open and read another hyperslab
h5f = tables.openFile(fileName)
print h5f
print h5f.root.carray[8:12, 18:22]
h5f.close()
```

The output for the previous script is something like:

```
carray1.h5 (File) ''
Last modif.: 'Thu Apr 12 10:15:38 2007'
Object Tree:
/ (RootGroup) ''
/carray (CArray(200, 300), shuffle, zlib(5)) ''

[[0 0 0 0]
 [0 0 0 0]
 [0 0 1 1]
 [0 0 1 1]]
```

# 4.9. The EArray class

This class represents extendible, homogeneous datasets in an HDF5 file.

The main difference between an `EArray` and a `CArray` (see Section 4.8), from which it inherits, is that the former can be enlarged along one of its dimensions, the *enlargeable dimension*. That means that the `Leaf.extdim` attribute (see Section 4.5.1) of any `EArray` instance will always be non-negative. Multiple enlargeable dimensions might be supported in the future.

New rows can be added to the end of an enlargeable array by using the `EArray.append()` method (see the section called "append(sequence)").

## 4.9.1. `EArray` methods

### append(sequence)

Add a `sequence` of data to the end of the dataset.

The sequence must have the same type as the array; otherwise a `TypeError` is raised. In the same way, the dimensions of the `sequence` must conform to the shape of the array, that is, all dimensions must match, with the exception of the enlargeable dimension, which can be of any length (even 0!). If the shape of the `sequence` is invalid, a `ValueError` is raised.

## 4.9.2. Example of use

See below a small example of the use of the `EArray` class. The code is available in `examples/earray1.py`:

```
import tables
import numpy

fileh = tables.openFile('earray1.h5', mode='w')
a = tables.StringAtom(itemsize=8)
# Use ``a`` as the object type for the enlargeable array.
array_c = fileh.createEArray(fileh.root, 'array_c', a, (0,), "Chars")
array_c.append(numpy.array(['a'*2, 'b'*4], dtype='S8'))
array_c.append(numpy.array(['a'*6, 'b'*8, 'c'*10], dtype='S8'))

# Read the string ``EArray`` we have created on disk.
for s in array_c:
    print 'array_c[%s] => %r' % (array_c.nrow, s)
# Close the file.
fileh.close()
```

The output for the previous script is something like:

```
array_c[0] => 'aa'
array_c[1] => 'bbbb'
array_c[2] => 'aaaaaa'
array_c[3] => 'bbbbbbbb'
array_c[4] => 'cccccccc'
```

# 4.10. The `VLArray` class

This class represents variable length (ragged) arrays in an HDF5 file.

Instances of this class represent array objects in the object tree with the property that their rows can have a *variable* number of homogeneous elements, called *atoms*. Like `Table` datasets (see Section 4.6), variable length arrays can have only one dimension, and the elements (atoms) of their rows can be fully multidimensional. `VLArray` objects do also support compression.

When reading a range of rows from a `VLArray`, you will *always* get a Python list of objects of the current flavor (each of them for a row), which may have different lengths.

This class provides methods to write or read data to or from variable length array objects in the file. Note that it also inherits all the public attributes and methods that `Leaf` (see Section 4.5) already provides.

## 4.10.1. `VLArray` instance variables

**atom** An `Atom` (see Section 4.13.3) instance representing the *type* and *shape* of the atomic objects to be saved. You may use a *pseudo-atom* for storing a serialized object or variable length string per row.

**flavor** The type of data object read from this leaf.

Please note that when reading several rows of `VLArray` data, the flavor only applies to the *components* of the returned Python list, not to the list itself.

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

## 4.10.2. `VLArray` methods

### append(sequence)

Add a `sequence` of data to the end of the dataset.

This method appends the objects in the `sequence` to a *single row* in this array. The type and shape of individual objects must be compliant with the atoms in the array. In the case of serialized objects and variable length strings, the object or string to append is itself the `sequence`.

### getEnum()

Get the enumerated type associated with this array.

If this array is of an enumerated type, the corresponding `Enum` instance (see Section 4.14.3) is returned. If it is not of an enumerated type, a `TypeError` is raised.

### iterrows(start=None, stop=None, step=None)

Iterate over the rows of the array.

This method returns an iterator yielding an object of the current flavor for each selected row in the array.

If a range is not supplied, *all the rows* in the array are iterated upon —you can also use the `VLArray.__iter__()` (see description) special method for that purpose. If you only want to iterate over a given *range of rows* in the array, you may use the `start`, `stop` and `step` parameters, which have the same meaning as in `VLArray.read()` (see description).

Example of use:

```
for row in vlarray.iterrows(step=4):
    print '%s[%d]--> %s' % (vlarray.name, vlarray.nrow, row)
```

### next()

Get the next element of the array during an iteration.

The element is returned as a list of objects of the current flavor.

### read(start=None, stop=None, step=1)

Get data in the array as a list of objects of the current flavor.

Please note that, as the lengths of the different rows are variable, the returned value is a *Python list* (not an array of the current flavor), with as many entries as specified rows in the range parameters.

The `start`, `stop` and `step` parameters can be used to select only a *range of rows* in the array. Their meanings are the same as in the built-in `range()` Python function, except that negative values of `step` are not allowed yet. Moreover, if only `start` is specified, then `stop` will be set to `start+1`. If you do not specify neither `start` nor `stop`, then *all the rows* in the array are selected.

## 4.10.3. `VLArray` special methods

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

### __getitem__(key)

Get a row or a range of rows from the array.

If `key` argument is an integer, the corresponding array row is returned as an object of the current flavor. If `key` is a slice, the range of rows determined by it is returned as a list of objects of the current flavor.

Example of use:

```
a_row = vlarray[4]
a_list = vlarray[4:1000:2]
```

### __iter__()

Iterate over the rows of the array.

This is equivalent to calling `VLArray.iterrows()` (see description) with default arguments, i.e. it iterates over *all the rows* in the array.

Example of use:

```
result = [row for row in vlarray]
```

Which is equivalent to:

```
result = [row for row in vlarray.iterrows()]
```

### __setitem__(keys, value)

Set a row in the array.

It takes different actions depending on the type of the `key` parameter: if it is an integer, the corresponding array row is set to `value`. If the `key` is a tuple, the first element refers to the row to be modified, and the second element to the range within the row to be updated with the `value` (so it can be an integer or a slice).

The type and shape of the `value` must be compatible with the type and shape determined by the `key`, otherwise, a `TypeError` or a `ValueError` will be raised.

**Note**

When updating the rows of a `VLArray` object which uses a pseudo-atom, there is a problem: you can only update values with *exactly* the same size in bytes than the original row. This is very difficult to meet with `object` pseudo-atoms, because `cPickle` applied on a Python object does not guarantee to return the same number of bytes than over another object, even if they are of the same class. This effectively limits the kinds of objects than can be updated in variable-length arrays.

Example of use:

```
vlarray[0] = vlarray[0] * 2 + 3
vlarray[99, 3:] = arange(96) * 2 + 3
# Negative values for start and stop (but not step) are supported.
vlarray[99, -99:-89:2] = vlarray[5] * 2 + 3
```

## 4.10.4. Example of use

See below a small example of the use of the `VLArray` class. The code is available in `examples/vlarray1.py`:

```
import tables
from numpy import *

# Create a VLArray:
fileh = tables.openFile('vlarray1.h5', mode='w')
vlarray = fileh.createVLArray(fileh.root, 'vlarray1',
                              tables.Int32Atom(shape=()),
                              "ragged array of ints",
                              filters=tables.Filters(1))
# Append some (variable length) rows:
vlarray.append(array([5, 6]))
vlarray.append(array([5, 6, 7]))
vlarray.append([5, 6, 9, 8])

# Now, read it through an iterator:
print '-->', vlarray.title
for x in vlarray:
    print '%s[%d]--> %s' % (vlarray.name, vlarray.nrow, x)

# Now, do the same with native Python strings.
vlarray2 = fileh.createVLArray(fileh.root, 'vlarray2',
                              tables.StringAtom(itemsize=2),
                              "ragged array of strings",
                              filters=tables.Filters(1))
vlarray2.flavor = 'python'
# Append some (variable length) rows:
print '-->', vlarray2.title
vlarray2.append(['5', '66'])
vlarray2.append(['5', '6', '77'])
vlarray2.append(['5', '6', '9', '88'])

# Now, read it through an iterator:
for x in vlarray2:
    print '%s[%d]--> %s' % (vlarray2.name, vlarray2.nrow, x)
```

```
# Close the file.
fileh.close()
```

The output for the previous script is something like:

```
--> ragged array of ints
vlarray1[0]--> [5 6]
vlarray1[1]--> [5 6 7]
vlarray1[2]--> [5 6 9 8]
--> ragged array of strings
vlarray2[0]--> ['5', '66']
vlarray2[1]--> ['5', '6', '77']
vlarray2[2]--> ['5', '6', '9', '88']
```

# 4.11. The `UnImplemented` class

This class represents datasets not supported by PyTables in an HDF5 file.

When reading a generic HDF5 file (i.e. one that has not been created with PyTables, but with some other HDF5 library based tool), chances are that the specific combination of datatypes or dataspaces in some dataset might not be supported by PyTables yet. In such a case, this dataset will be mapped into an `UnImplemented` instance and the user will still be able to access the complete object tree of the generic HDF5 file. The user will also be able to *read and write the attributes* of the dataset, *access some of its metadata*, and perform *certain hierarchy manipulation operations* like deleting or moving (but not copying) the node. Of course, the user will not be able to read the actual data on it.

This is an elegant way to allow users to work with generic HDF5 files despite the fact that some of its datasets are not supported by PyTables. However, if you are really interested in having full access to an unimplemented dataset, please get in contact with the developer team.

This class does not have any public instance variables or methods, except those inherited from the `Leaf` class (see Section 4.5).

# 4.12. The `AttributeSet` class

Container for the HDF5 attributes of a `Node` (see Section 4.3).

This class provides methods to create new HDF5 node attributes, and to get, rename or delete existing ones.

Like in `Group` instances (see Section 4.4), `AttributeSet` instances make use of the *natural naming* convention, i.e. you can access the attributes on disk as if they were normal Python attributes of the `AttributeSet` instance.

This offers the user a very convenient way to access HDF5 node attributes. However, for this reason and in order not to pollute the object namespace, one can not assign *normal* attributes to `AttributeSet` instances, and their members use names which start by special prefixes as happens with `Group` objects.

## 4.12.1. Notes on native and pickled attributes

The values of most basic types are saved as HDF5 native data in the HDF5 file. This includes Python `bool`, `int`, `float`, `complex` and `str` (but not `long` nor `unicode`) values, as well as their NumPy scalar versions and *homogeneous* NumPy arrays of them. When read, these values are always loaded as NumPy scalar or array objects, as needed.

For that reason, attributes in native HDF5 files will be always mapped into NumPy objects. Specifically, a multidimensional attribute will be mapped into a multidimensional `ndarray` and a scalar will be mapped into a

NumPy scalar object (for example, a scalar `H5T_NATIVE_LLONG` will be read and returned as a `numpy.int64` scalar).

However, other kinds of values are serialized using `cPickle`, so you only will be able to correctly retrieve them using a Python-aware HDF5 library. Thus, if you want to save Python scalar values and make sure you are able to read them with generic HDF5 tools, you should make use of *scalar or homogeneous array NumPy objects* (for example, `numpy.int64(1)` or `numpy.array([1, 2, 3], dtype='int16')`).

One more piece of advice: because of the various potential difficulties in restoring a Python object stored in an attribute, you may end up getting a `cPickle` string where a Python object is expected. If this is the case, you may wish to run `cPickle.loads()` on that string to get an idea of where things went wrong, as shown in this example:

```
>>> import os, tempfile
>>> import tables
>>>
>>> class MyClass(object):
...   foo = 'bar'
...
>>> myObject = MyClass()  # save object of custom class in HDF5 attr
>>> h5fname = tempfile.mktemp(suffix='.h5')
>>> h5f = tables.openFile(h5fname, 'w')
>>> h5f.root._v_attrs.obj = myObject  # store the object
>>> print h5f.root._v_attrs.obj.foo  # retrieve it
bar
>>> h5f.close()
>>>
>>> del MyClass, myObject  # delete class of object and reopen file
>>> h5f = tables.openFile(h5fname, 'r')
>>> print repr(h5f.root._v_attrs.obj)
'ccopy_reg\n_reconstructor...
>>> import cPickle  # let's unpickle that to see what went wrong
>>> cPickle.loads(h5f.root._v_attrs.obj)
Traceback (most recent call last):
  ...
AttributeError: 'module' object has no attribute 'MyClass'
>>> # So the problem was not in the stored object,
... # but in the *environment* where it was restored.
... h5f.close()
>>> os.remove(h5fname)
```

## 4.12.2. `AttributeSet` instance variables

**_v_attrnames** A list with all attribute names.

**_v_attrnamessys** A list with system attribute names.

**_v_attrnamesuser** A list with user attribute names.

**_v_node** The `Node` instance (see Section 4.3) this attribute set is associated with.

## 4.12.3. `AttributeSet` methods

Note that this class overrides the `__getattr__()`, `__setattr__()` and `__delattr__()` special methods. This allows you to read, assign or delete attributes on disk by just using the next constructs:

```
leaf.attrs.myattr = 'str attr'    # set a string (native support)
leaf.attrs.myattr2 = 3            # set an integer (native support)
leaf.attrs.myattr3 = [3, (1, 2)]  # a generic object (Pickled)
attrib = leaf.attrs.myattr        # get the attribute ``myattr``
del leaf.attrs.myattr             # delete the attribute ``myattr``
```

In addition, the dictionary-like `__getitem__()`, `__setitem__()` and `__delitem__()` methods are available, so you may write things like this:

```
for name in node._v_attrs._f_list():
    print "name: %s, value: %s" % (name, node._v_attrs[name])
```

Use whatever idiom you prefer to access the attributes.

If an attribute is set on a target node that already has a large number of attributes, a `PerformanceWarning` will be issued.

### _f_copy(where)

Copy attributes to the `where` node.

Copies all user and certain system attributes to the given `where` node (a `Node` instance —see Section 4.3), replacing the existing ones.

### _f_list(attrset='user')

Get a list of attribute names.

The `attrset` string selects the attribute set to be used. A `'user'` value returns only user attributes (this is the default). A `'sys'` value returns only system attributes. Finally, `'all'` returns both system and user attributes.

### _f_rename(oldattrname, newattrname)

Rename an attribute from `oldattrname` to `newattrname`.

### __contains__(name)

Is there an attribute with that `name`?

A true value is returned if the attribute set has an attribute with the given name, false otherwise.

# 4.13. Declarative classes

In this section a series of classes that are meant to *declare* datatypes that are required for primary PyTables datasets (like `Table` or `VLArray`) are described.

## 4.13.1. The `IsDescription` class

Description of the structure of a table or nested column.

This class is designed to be used as an easy, yet meaningful way to describe the structure of new `Table` (see Section 4.6) datasets or nested columns through the definition of *derived classes*. In order to define such a class, you must declare it as descendant of `IsDescription`, with as many attributes as columns you want in your table. The name of each attribute will become the name of a column, and its value will hold a description of it.

Ordinary columns can be described using instances of the `Col` class (see Section 4.13.2). Nested columns can be described by using classes derived from `IsDescription`, instances of it, or name-description dictionaries. Derived classes can be declared in place (in which case the column takes the name of the class) or referenced by name.

Nested columns can have a `_v_pos` special attribute which sets the *relative* position of the column among sibling columns *also having explicit positions*. The `pos` constructor argument of `Col` intances is used for the same purpose. Columns with no explicit position will be placed afterwards in alphanumeric order.

Once you have created a description object, you can pass it to the `Table` constructor, where all the information it contains will be used to define the table structure. See the Section 3.4 for an example on how that works.

### `IsDescription` special attributes

These are the special attributes that the user can specify *when declaring* an `IsDescription` subclass to complement its *metadata*.

**_v_pos** Sets the position of a possible nested column description among its sibling columns.

### `IsDescription` class variables

The following attributes are *automatically created* when an `IsDescription` subclass is declared. Please note that declared columns can no longer be accessed as normal class variables after its creation.

**columns** Maps the name of each column in the description to its own descriptive object.

## 4.13.2. The `Col` class and its descendants

Defines a non-nested column.

`Col` instances are used as a means to declare the different properties of a non-nested column in a table or nested column. `Col` classes are descendants of their equivalent `Atom` classes (see Section 4.13.3), but their instances have an additional `_v_pos` attribute that is used to decide the position of the column inside its parent table or nested column (see the `IsDescription` class in Section 4.13.1 for more information on column positions).

In the same fashion as `Atom`, you should use a particular `Col` descendant class whenever you know the exact type you will need when writing your code. Otherwise, you may use one of the `Col.from_*()` factory methods.

### `Col` instance variables

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

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

### `Col` factory methods

#### from_atom(atom, pos=None)

Create a `Col` definition from a PyTables `atom`.

An optional position may be specified as the `pos` argument.

#### from_dtype(dtype, dflt=None, pos=None)

Create a `Col` definition from a NumPy `dtype`.

Optional default value and position may be specified as the `dflt` and `pos` arguments, respectively. Information in the `dtype` not represented in a `Col` is ignored.

### from_kind(kind, itemsize=None, shape=(), dflt=None, pos=None)

Create a `Col` definition from a PyTables `kind`.

Optional item size, shape, default value and position may be specified as the `itemsize`, `shape`, `dflt` and `pos` arguments, respectively. Bear in mind that not all columns support a default item size.

### from_sctype(sctype, shape=(), dflt=None, pos=None)

Create a `Col` definition from a NumPy scalar type `sctype`.

Optional shape, default value and position may be specified as the `shape`, `dflt` and `pos` arguments, respectively. Information in the `sctype` not represented in a `Col` is ignored.

### from_type(type, shape=(), dflt=None, pos=None)

Create a `Col` definition from a PyTables `type`.

Optional shape, default value and position may be specified as the `shape`, `dflt` and `pos` arguments, respectively.

### `Col` constructors

For each `TYPEAtom` class there is a matching `TYPECol` class with the same constructor signature, plus an additional `pos` parameter, which defaults to `None` and may take an integer. This argument is used to set the `_v_pos` attribute.

## 4.13.3. The `Atom` class and its descendants.

Defines the type of atomic cells stored in a dataset.

The meaning of *atomic* is that individual elements of a cell can not be extracted directly by indexing (i.e. `__getitem__()`) the dataset; e.g. if a dataset has shape (2, 2) and its atoms have shape (3,), to get the third element of the cell at (1, 0) one should use `dataset[1,0][2]` instead of `dataset[1,0,2]`.

The `Atom` class is meant to declare the different properties of the *base element* (also known as *atom*) of `CArray`, `EArray` and `VLArray` datasets, although they are also used to describe the base elements of `Array` datasets. Atoms have the property that their length is always the same. However, you can grow datasets along the extensible dimension in the case of `EArray` or put a variable number of them on a `VLArray` row. Moreover, they are not restricted to scalar values, and they can be *fully multidimensional objects*.

A series of descendant classes are offered in order to make the use of these element descriptions easier. You should use a particular `Atom` descendant class whenever you know the exact type you will need when writing your code. Otherwise, you may use one of the `Atom.from_*()` factory methods.

### `Atom` instance variables

**dflt** The default value of the atom.

If the user does not supply a value for an element while filling a dataset, this default value will be written to disk. If the user supplies a scalar value for a multidimensional atom, this value is automatically *broadcast* to all the items in the atom cell. If `dflt` is not supplied, an appropriate zero value (or *null* string) will be chosen by default. Please note that default values are kept internally as NumPy objects.

| | |
|---|---|
| **dtype** | The NumPy `dtype` that most closely matches this atom. |
| **itemsize** | Size in bytes of a sigle item in the atom.<br><br>Specially useful for atoms of the `string` kind. |
| **kind** | The PyTables kind of the atom (a string). For a relation of the data kinds supported by PyTables and more information about them, see Appendix A. |
| **recarrtype** | String type to be used in `numpy.rec.array()`. |
| **shape** | The shape of the atom (a tuple, `()` for scalar atoms). |
| **size** | Total size in bytes of the atom. |
| **type** | The PyTables type of the atom (a string). For a relation of the data types supported by PyTables and more information about them, see Appendix A. |

Atoms can be compared with atoms and other objects for strict (in)equality without having to compare individual attributes:

```
>>> atom1 = StringAtom(itemsize=10)  # same as ``atom2``
>>> atom2 = Atom.from_kind('string', 10)  # same as ``atom1``
>>> atom3 = IntAtom()
>>> atom1 == 'foo'
False
>>> atom1 == atom2
True
>>> atom2 != atom1
False
>>> atom1 == atom3
False
>>> atom3 != atom2
True
```

## `Atom` methods

### copy(**override)

Get a copy of the atom, possibly overriding some arguments.

Constructor arguments to be overridden must be passed as keyword arguments.

```
>>> atom1 = StringAtom(itemsize=12)
>>> atom2 = atom1.copy()
>>> print atom1
StringAtom(itemsize=12, shape=(), dflt='')
>>> print atom2
StringAtom(itemsize=12, shape=(), dflt='')
>>> atom1 is atom2
False
>>> atom3 = atom1.copy(itemsize=100, shape=(2, 2))
>>> print atom3
StringAtom(itemsize=100, shape=(2, 2), dflt='')
>>> atom1.copy(foobar=42)
Traceback (most recent call last):
  ...
```

```
TypeError: __init__() got an unexpected keyword argument 'foobar'
```

## `Atom` factory methods

### from_dtype(dtype, dflt=None)

Create an `Atom` from a NumPy `dtype`.

An optional default value may be specified as the `dflt` argument. Information in the `dtype` not represented in an `Atom` is ignored.

```
>>> import numpy
>>> Atom.from_dtype(numpy.dtype((numpy.int16, (2, 2))))
Int16Atom(shape=(2, 2), dflt=0)
>>> Atom.from_dtype(numpy.dtype('S5'), dflt='hello')
StringAtom(itemsize=5, shape=(), dflt='hello')
>>> Atom.from_dtype(numpy.dtype('Float64'))
Float64Atom(shape=(), dflt=0.0)
```

### from_kind(kind, itemsize=None, shape=(), dflt=None)

Create an `Atom` from a PyTables `kind`.

Optional item size, shape and default value may be specified as the `itemsize`, `shape` and `dflt` arguments, respectively. Bear in mind that not all atoms support a default item size.

```
>>> Atom.from_kind('int', itemsize=2, shape=(2, 2))
Int16Atom(shape=(2, 2), dflt=0)
>>> Atom.from_kind('int', shape=(2, 2))
Int32Atom(shape=(2, 2), dflt=0)
>>> Atom.from_kind('int', shape=1)
Int32Atom(shape=(1,), dflt=0)
>>> Atom.from_kind('string', itemsize=5, dflt='hello')
StringAtom(itemsize=5, shape=(), dflt='hello')
>>> Atom.from_kind('string', dflt='hello')
Traceback (most recent call last):
  ...
ValueError: no default item size for kind ``string``
>>> Atom.from_kind('Float')
Traceback (most recent call last):
  ...
ValueError: unknown kind: 'Float'
```

Moreover, some kinds with atypical constructor signatures are not supported; you need to use the proper constructor:

```
>>> Atom.from_kind('enum')
Traceback (most recent call last):
  ...
ValueError: the ``enum`` kind is not supported...
```

### from_sctype(sctype, shape=(), dflt=None)

Create an `Atom` from a NumPy scalar type `sctype`.

Optional shape and default value may be specified as the `shape` and `dflt` arguments, respectively. Information in the `sctype` not represented in an `Atom` is ignored.

```
>>> import numpy
>>> Atom.from_sctype(numpy.int16, shape=(2, 2))
Int16Atom(shape=(2, 2), dflt=0)
>>> Atom.from_sctype('S5', dflt='hello')
Traceback (most recent call last):
  ...
ValueError: unknown NumPy scalar type: 'S5'
>>> Atom.from_sctype('Float64')
Float64Atom(shape=(), dflt=0.0)
```

### from_type(type, shape=(), dflt=None)

Create an `Atom` from a PyTables `type`.

Optional shape and default value may be specified as the `shape` and `dflt` arguments, respectively.

```
>>> Atom.from_type('bool')
BoolAtom(shape=(), dflt=False)
>>> Atom.from_type('int16', shape=(2, 2))
Int16Atom(shape=(2, 2), dflt=0)
>>> Atom.from_type('string40', dflt='hello')
Traceback (most recent call last):
  ...
ValueError: unknown type: 'string40'
>>> Atom.from_type('Float64')
Traceback (most recent call last):
  ...
ValueError: unknown type: 'Float64'
```

## `Atom` constructors

There are some common arguments for most `Atom`-derived constructors:

**itemsize**
For types with a non-fixed size, this sets the size in bytes of individual items in the atom.

**shape**
Sets the shape of the atom. An integer shape of `N` is equivalent to the tuple `(N,)`.

**dflt**
Sets the default value for the atom.

A relation of the different constructors with their parameters follows.

### StringAtom(itemsize, shape=(), dflt='')

Defines an atom of type `string`.

The item size is the *maximum* length in characters of strings.

### BoolAtom(shape=(), dflt=False)

Defines an atom of type `bool`.

### IntAtom(itemsize=4, shape=(), dflt=0)

Defines an atom of a signed integral type (`int` kind).

### Int8Atom(shape=(), dflt=0)

Defines an atom of type `int8`.

### Int16Atom(shape=(), dflt=0)

Defines an atom of type `int16`.

### Int32Atom(shape=(), dflt=0)

Defines an atom of type `int32`.

### Int64Atom(shape=(), dflt=0)

Defines an atom of type `int64`.

### UIntAtom(itemsize=4, shape=(), dflt=0)

Defines an atom of an unsigned integral type (`uint` kind).

### UInt8Atom(shape=(), dflt=0)

Defines an atom of type `uint8`.

### UInt16Atom(shape=(), dflt=0)

Defines an atom of type `uint16`.

### UInt32Atom(shape=(), dflt=0)

Defines an atom of type `uint32`.

### UInt64Atom(shape=(), dflt=0)

Defines an atom of type `uint64`.

### Float32Atom(shape=(), dflt=0.0)

Defines an atom of type `float32`.

### Float64Atom(shape=(), dflt=0.0)

Defines an atom of type `float64`.

### ComplexAtom(itemsize, shape=(), dflt=0j)

Defines an atom of kind `complex`.

Allowed item sizes are 8 (single precision) and 16 (double precision). This class must be used instead of more concrete ones to avoid confusions with `numarray`-like precision specifications used in PyTables 1.X.

### TimeAtom(itemsize=4, shape=(), dflt=0)

Defines an atom of time type (`time` kind).

There are two distinct supported types of time: a 32 bit integer value and a 64 bit floating point value. Both of them reflect the number of seconds since the Unix epoch. This atom has the property of being stored using the HDF5 time datatypes.

### Time32Atom(shape=(), dflt=0)

Defines an atom of type `time32`.

### Time64Atom(shape=(), dflt=0.0)

Defines an atom of type `time64`.

### EnumAtom(enum, dflt, base, shape=())

Description of an atom of an enumerated type.

Instances of this class describe the atom type used to store enumerated values. Those values belong to an enumerated type, defined by the first argument (`enum`) in the constructor of the atom, which accepts the same kinds of arguments as the `Enum` class (see Section 4.14.3). The enumerated type is stored in the `enum` attribute of the atom.

A default value must be specified as the second argument (`dflt`) in the constructor; it must be the *name* (a string) of one of the enumerated values in the enumerated type. When the atom is created, the corresponding concrete value is broadcast and stored in the `dflt` attribute (setting different default values for items in a multidimensional atom is not supported yet). If the name does not match any value in the enumerated type, a `KeyError` is raised.

Another atom must be specified as the `base` argument in order to determine the base type used for storing the values of enumerated values in memory and disk. This *storage atom* is kept in the `base` attribute of the created atom. As a shorthand, you may specify a PyTables type instead of the storage atom, implying that this has a scalar shape.

The storage atom should be able to represent each and every concrete value in the enumeration. If it is not, a `TypeError` is raised. The default value of the storage atom is ignored.

The `type` attribute of enumerated atoms is always `enum`.

Enumerated atoms also support comparisons with other objects:

```
>>> enum = ['T0', 'T1', 'T2']
>>> atom1 = EnumAtom(enum, 'T0', 'int8')  # same as ``atom2``
>>> atom2 = EnumAtom(enum, 'T0', Int8Atom())  # same as ``atom1``
>>> atom3 = EnumAtom(enum, 'T0', 'int16')
>>> atom4 = Int8Atom()
>>> atom1 == enum
False
>>> atom1 == atom2
True
>>> atom2 != atom1
False
>>> atom1 == atom3
False
>>> atom1 == atom4
False
>>> atom4 != atom1
True
```

#### Examples

The next C `enum` construction:

```
enum myEnum {
  T0,
  T1,
```

```
  T2
};
```

would correspond to the following PyTables declaration:

```
>>> myEnumAtom = EnumAtom(['T0', 'T1', 'T2'], 'T0', 'int32')
```

Please note the `dflt` argument with a value of `'T0'`. Since the concrete value matching `T0` is unknown right now (we have not used explicit concrete values), using the name is the only option left for defining a default value for the atom.

The chosen representation of values for this enumerated atom uses unsigned 32-bit integers, which surely wastes quite a lot of memory. Another size could be selected by using the `base` argument (this time with a full-blown storage atom):

```
>>> myEnumAtom = EnumAtom(['T0', 'T1', 'T2'], 'T0', UInt8Atom())
```

You can also define multidimensional arrays for data elements:

```
>>> myEnumAtom = EnumAtom(
...     ['T0', 'T1', 'T2'], 'T0', base='uint32', shape=(3,2))
```

for 3x2 arrays of `uint32`.

## 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 Section 4.10), 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 `dflt` 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()

Defines an atom of type `object`.

This class is meant to fit *any* kind of Python object in a row of a `VLArray` dataset by using `cPickle` behind the scenes. Due to the fact that you can not foresee how long will be the output of the `cPickle` serialization (i.e. the atom already has a *variable* length), you can only fit *one object per row*. However, you can still group several objects in a single tuple or list and pass it to the `VLArray.append()` method (see description).

Object atoms do not accept parameters and they cause the reads of rows to always return Python objects. You can regard `object` atoms as an easy way to save an arbitrary number of generic Python objects in a `VLArray` dataset.

### VLStringAtom()

Defines an atom of type `vlstring`.

This class describes a *row* of the `VLArray` class, rather than an atom. It differs from the `StringAtom` class in that you can only add *one instance of it to one specific row*, i.e. the `VLArray.append()` method (see description) only accepts one object when the base atom is of this type.

Like `StringAtom`, this class does not make assumptions on the encoding of the string, and raw bytes are stored as is. Unicode strings are supported as long as no character is out of the ASCII set; otherwise, you will need to *explicitly*

convert them to strings before you can save them. For full Unicode support, using `VLUnicodeAtom` (see description) is recommended.

Variable-length string atoms do not accept parameters and they cause the reads of rows to always return Python strings. You can regard `vlstring` atoms as an easy way to save generic variable length strings.

### VLUnicodeAtom()

Defines an atom of type `vlunicode`.

This class describes a *row* of the `VLArray` class, rather than an atom. It is very similar to `VLStringAtom` (see description), but it stores Unicode strings (using 32-bit characters a la UCS-4, so all strings of the same length also take up the same space).

This class does not make assumptions on the encoding of plain input strings. Plain strings are supported as long as no character is out of the ASCII set; otherwise, you will need to *explicitly* convert them to Unicode before you can save them.

Variable-length Unicode atoms do not accept parameters and they cause the reads of rows to always return Python Unicode strings. You can regard `vlunicode` atoms as an easy way to save variable length Unicode strings.

# 4.14. Helper classes

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

## 4.14.1. The `Filters` class

Container for filter properties.

This class is meant to serve as a container that keeps information about the filter properties associated with the chunked leaves, that is `Table`, `CArray`, `EArray` and `VLArray`.

Instances of this class can be directly compared for equality.

### `Filters` instance variables

**fletcher32** Whether the *Fletcher32* filter is active or not.

**complevel** The compression level (0 disables compression).

**complib** The compression filter used (irrelevant when compression is not enabled).

**shuffle** Whether the *Shuffle* filter is active or not.

### Example of use

This is a small example on using the `Filters` class:

```
import numpy
from tables import *

fileh = openFile('test5.h5', mode='w')
atom = Float32Atom()
filters = Filters(complevel=1, complib='lzo', fletcher32=True)
arr = fileh.createEArray(fileh.root, 'earray', atom, (0,2),
```

```
                          "A growable array", filters=filters)
# Append several rows in only one call
arr.append(numpy.array([[1., 2.],
                        [2., 3.],
                        [3., 4.]], dtype=numpy.float32))

# Print information on that enlargeable array
print "Result Array:"
print repr(arr)

fileh.close()
```

This enforces the use of the LZO library, a compression level of 1 and a Fletcher32 checksum filter as well. See the output of this example:

```
Result Array:
/earray (EArray(3, 2), fletcher32, shuffle, lzo(1)) 'A growable array'
  type = float32
  shape = (3, 2)
  itemsize = 4
  nrows = 3
  extdim = 0
  flavor = 'numpy'
  byteorder = 'little'
```

## `Filters(complevel=0, complib='zlib', shuffle=True, fletcher32=False)`

Create a new `Filters` instance.

**complevel**
Specifies a compression level for data. The allowed range is 0-9. A value of 0 (the default) disables compression.

**complib**
Specifies the compression library to be used. Right now, 'zlib' (the default), 'lzo' and 'bzip2' are supported. Specifying a compression library which is not available in the system issues a `FiltersWarning` and sets the library to the default one.

See Section 5.3 for some advice on which library is better suited to your needs.

**shuffle**
Whether or not to use the *Shuffle* filter in the HDF5 library. This is normally used to improve the compression ratio. A false value disables shuffling and a true one enables it. The default value depends on whether compression is enabled or not; if compression is enabled, shuffling defaults to be enabled, else shuffling is disabled. Shuffling can only be used when compression is enabled.

**fletcher32**
Whether or not to use the *Fletcher32* filter in the HDF5 library. This is used to add a checksum on each data chunk. A false value (the default) disables the checksum.

## `copy(override)`

Get a copy of the filters, possibly overriding some arguments.

Constructor arguments to be overridden must be passed as keyword arguments.

Using this method is recommended over replacing the attributes of an instance, since instances of this class may become immutable in the future.

```
>>> filters1 = Filters()
>>> filters2 = filters1.copy()
>>> filters1 == filters2
True
>>> filters1 is filters2
False
>>> filters3 = filters1.copy(complevel=1)
Traceback (most recent call last):
  ...
ValueError: compression library ``None`` is not supported...
>>> filters3 = filters1.copy(complevel=1, complib='zlib')
>>> print filters1
Filters(complevel=0, shuffle=False, fletcher32=False)
>>> print filters3
Filters(complevel=1, complib='zlib', shuffle=False, fletcher32=False)
>>> filters1.copy(foobar=42)
Traceback (most recent call last):
  ...
TypeError: __init__() got an unexpected keyword argument 'foobar'
```

## 4.14.2. The `Index` class

Represents the index of a column in a table.

This class is used to keep the indexing information for columns in a `Table` dataset (see Section 4.6). It is actually a descendant of the `Group` class (see Section 4.4), with some added functionality. An `Index` is always associated with one and only one column in the table.

**Note**

This class is mainly intended for internal use, but some of its documented attributes and methods may be interesting for the programmer.

**Note**

Column indexing is only available in PyTables Pro.

### `Index` instance variables

**column** The `Column` (see Section 4.6.9) instance for the indexed column.

**dirty** Whether the index is dirty or not.

Dirty indexes are out of sync with column data, so they exist but they are not usable.

**filters** Filter properties for this index —see `Filters` in Section 4.14.1.

**nelements** The number of currently indexed row for this column.

**is_CSI** Whether the index is completely sorted or not.

### `Index` methods

#### readSorted(start=None, stop=None, step=None)

Return the sorted values of index in the specified range.

The meaning of the `start`, `stop` and `step` arguments is the same as in `Table.readSorted()` (see description).

#### readIndices(start=None, stop=None, step=None)

Return the indices values of index in the specified range.

The meaning of the `start`, `stop` and `step` arguments is the same as in `Table.readSorted()` (see description).

### `Index` special methods

#### __getitem__(key)

Return the indices values of index in the specified range.

If `key` argument is an integer, the corresponding index is returned. If `key` is a slice, the range of indices determined by it is returned. A negative value of `step` in slice is supported, meaning that the results will be returned in reverse order.

This method is equivalent to `Index.readIndices()` (see description).

## 4.14.3. The `Enum` class

Enumerated type.

Each instance of this class represents an enumerated type. The values of the type must be declared *exhaustively* and named with *strings*, and they might be given explicit concrete values, though this is not compulsory. Once the type is defined, it can not be modified.

There are three ways of defining an enumerated type. Each one of them corresponds to the type of the only argument in the constructor of `Enum`:

- *Sequence of names*: each enumerated value is named using a string, and its order is determined by its position in the sequence; the concrete value is assigned automatically:

```
>>> boolEnum = Enum(['True', 'False'])
```

- *Mapping of names*: each enumerated value is named by a string and given an explicit concrete value. All of the concrete values must be different, or a `ValueError` will be raised.

```
>>> priority = Enum({'red': 20, 'orange': 10, 'green': 0})
>>> colors = Enum({'red': 1, 'blue': 1})
Traceback (most recent call last):
  ...
ValueError: enumerated values contain duplicate concrete values: 1
```

- *Enumerated type*: in that case, a copy of the original enumerated type is created. Both enumerated types are considered equal.

```
>>> prio2 = Enum(priority)
>>> priority == prio2
True
```

Please note that names starting with _ are not allowed, since they are reserved for internal usage:

```
>>> prio2 = Enum(['_xx'])
Traceback (most recent call last):
  ...
ValueError: name of enumerated value can not start with ``_``: '_xx'
```

The concrete value of an enumerated value is obtained by getting its name as an attribute of the `Enum` instance (see `__getattr__()`) or as an item (see `__getitem__()`). This allows comparisons between enumerated values and assigning them to ordinary Python variables:

```
>>> redv = priority.red
>>> redv == priority['red']
True
>>> redv > priority.green
True
>>> priority.red == priority.orange
False
```

The name of the enumerated value corresponding to a concrete value can also be obtained by using the `__call__()` method of the enumerated type. In this way you get the symbolic name to use it later with `__getitem__()`:

```
>>> priority(redv)
'red'
>>> priority.red == priority[priority(priority.red)]
True
```

(If you ask, the `__getitem__()` method is not used for this purpose to avoid ambiguity in the case of using strings as concrete values.)

## `Enum` special methods

### `__call__(value, *default)`

Get the name of the enumerated value with that concrete `value`.

If there is no value with that concrete value in the enumeration and a second argument is given as a `default`, this is returned. Else, a `ValueError` is raised.

This method can be used for checking that a concrete value belongs to the set of concrete values in an enumerated type.

### `__contains__(name)`

Is there an enumerated value with that `name` in the type?

If the enumerated type has an enumerated value with that `name`, `True` is returned. Otherwise, `False` is returned. The `name` must be a string.

This method does *not* check for concrete values matching a value in an enumerated type. For that, please use the `Enum.__call__()` method (see description).

### `__eq__(other)`

Is the `other` enumerated type equivalent to this one?

Two enumerated types are equivalent if they have exactly the same enumerated values (i.e. with the same names and concrete values).

### __getattr__(name)

Get the concrete value of the enumerated value with that `name`.

The `name` of the enumerated value must be a string. If there is no value with that `name` in the enumeration, an `AttributeError` is raised.

### __getitem__(name)

Get the concrete value of the enumerated value with that `name`.

The `name` of the enumerated value must be a string. If there is no value with that `name` in the enumeration, a `KeyError` is raised.

### __iter__()

Iterate over the enumerated values.

Enumerated values are returned as `(name, value)` pairs *in no particular order*.

### __len__()

Return the number of enumerated values in the enumerated type.

### __repr__()

Return the canonical string representation of the enumeration. The output of this method can be evaluated to give a new enumeration object that will compare equal to this one.

# Chapter 5. 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 tune it 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[1]. You should provide this information at leaf creation time by passing this value to the `expectedrows` argument of the `createTable()` method (see description) or `createEArray()` method (see Section ). For `VLArray` leaves, you must pass the expected size in MBytes by using the argument `expectedsizeinMB` of `createVLArray()` (see Section ) instead.

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!

[1] `CArray` nodes, though not extensible, are chunked and have their optimum chunk size automatically computed at creation time, since their final shape is known.

## 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 four 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`. Before each measurement, the OS cache has been emptied in order to remove its effects.

In figures 5.1, 5.2, 5.3 and 5.4, you can see how the chunksize affects different aspects, like creation time, file sizes, sequential read time and random read time. As you can see, if you properly inform PyTables about the extent of your datasets, you will get an automatic chunksize value (128 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 5.1). On the contrary, if your goal is to optimize the random access time for an uncompressed dataset, you may want to increase the chunksize up to 1 MB (see Figure 5.4).

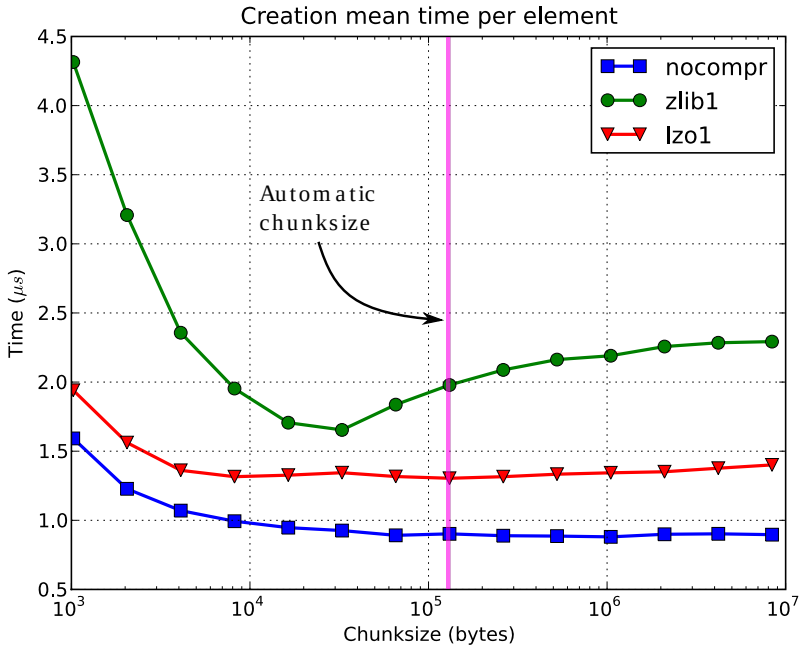


Figure 5.1. Creation time per element for a 15 GB EArray and different chunksizes.

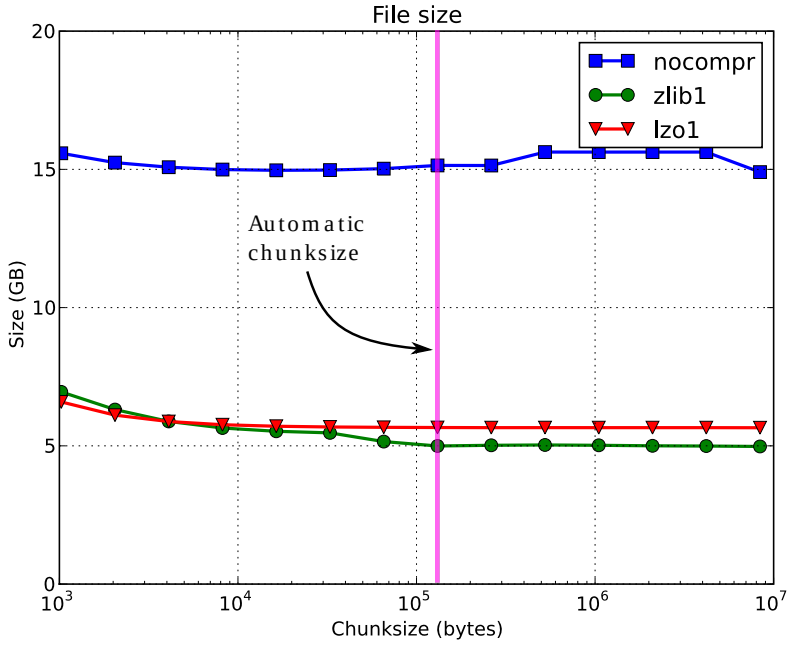


Figure 5.2. File sizes for a 15 GB EArray and different chunksizes.

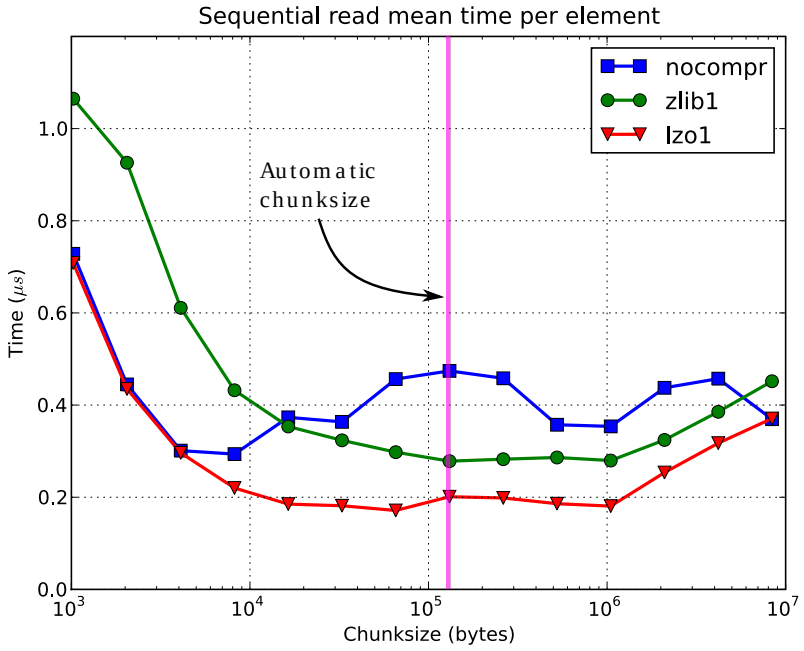


Figure 5.3. Sequential access time per element for a 15 GB EArray and different chunksizes.

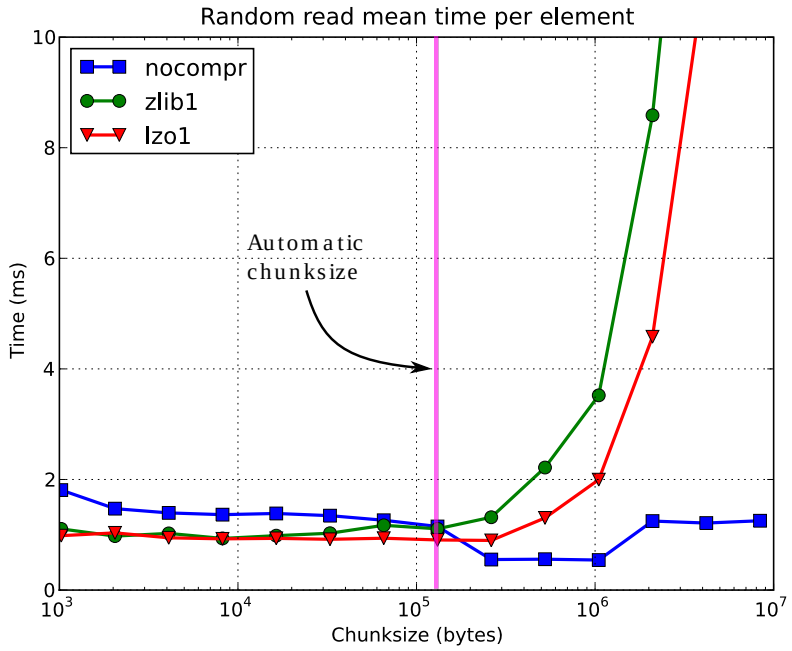


Figure 5.4. Random access time per element for a 15 GB EArray and different chunksizes.

As you see, by manually specifying the chunksize of a leave you will not normally bring a drastic increase in performance, but at least, you will have the opportunity to fine-tune such an important parameter for your application.

Finally, it is worth to not that adjusting the chunksize can be specially important if you decide that you want to access your dataset by blocks of certain dimensions. In this case, you may want to set your `chunkshape` to be the same than these dimensions. In this case, 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 phase is your best ally.

# 5.2. Accelerating your searches

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.where()` iterator and related query methods (see Section 4.6.4). 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 5.5 and Figure 5.6.

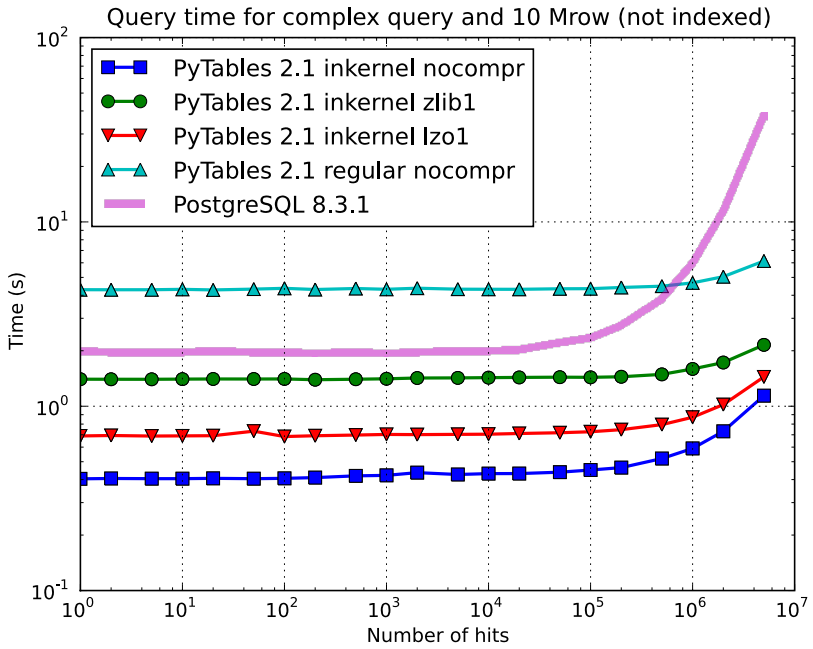


Figure 5.5. Times for non-indexed complex queries in a small table with 10 millions of rows: the data fits in memory.

By looking at Figure 5.5 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 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 [11]), 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 5.6), 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 Section 5.3, “Compression issues”).

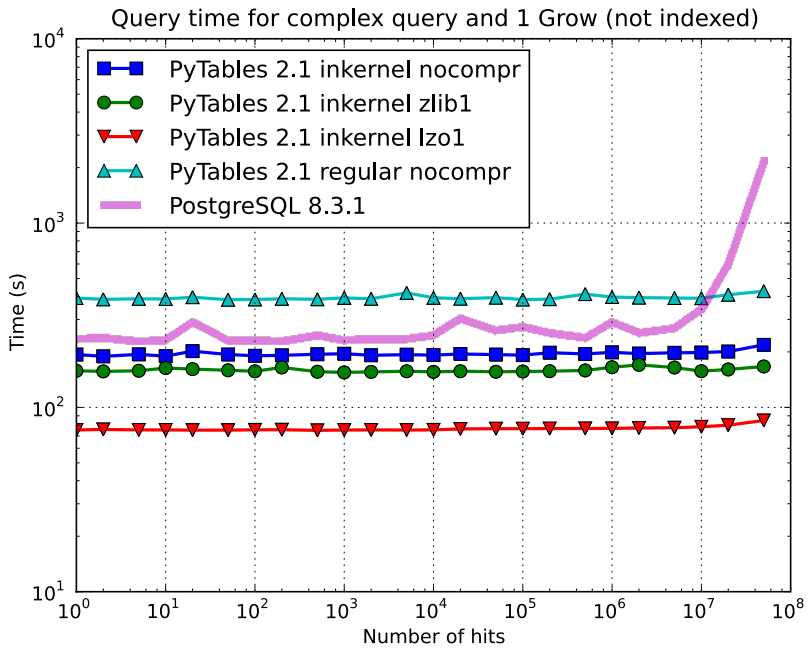


Figure 5.6. Times for non-indexed complex queries in a large table with 1 billion of rows: the data does not fit in memory.

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 Appendix B for more information on in-kernel condition syntax.

## 5.2.2. Indexed searches

**Note**

Indexing is only available in PyTables Pro.

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 [20]). 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 (see description) 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 Pro 2.1 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 (see description).

One important aspect of indexing in PyTables Pro 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 5.7, 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 5.8.

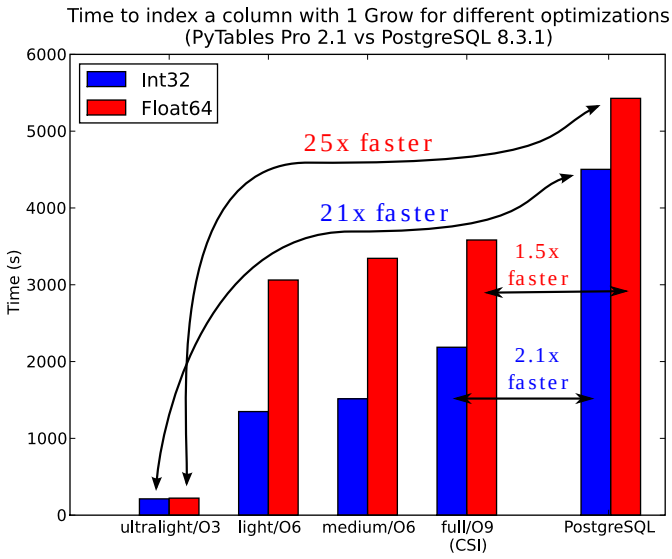


Figure 5.7. Times for indexing an `Int32` and `Float64` column.

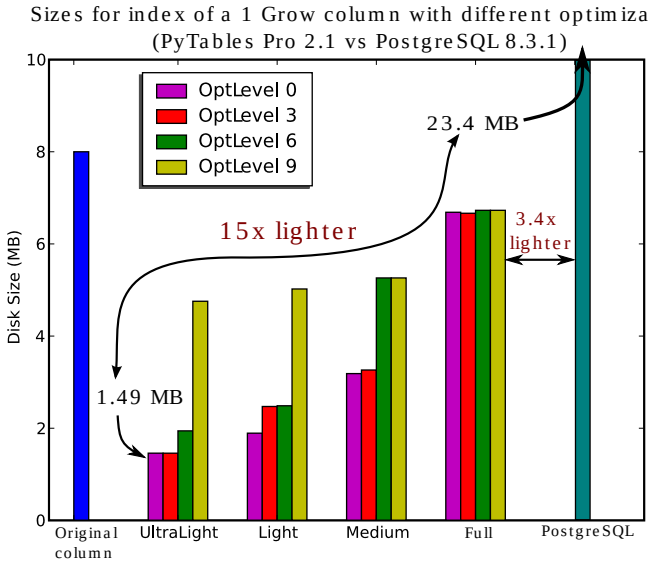


Figure 5.8. Sizes for an index of a `Float64` column with 1 billion of rows.

The user can select the index quality by passing the desired `optlevel` and `kind` arguments to the `createIndex()` method (see description). We can see in figures 5.7 and 5.8 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 5.9.

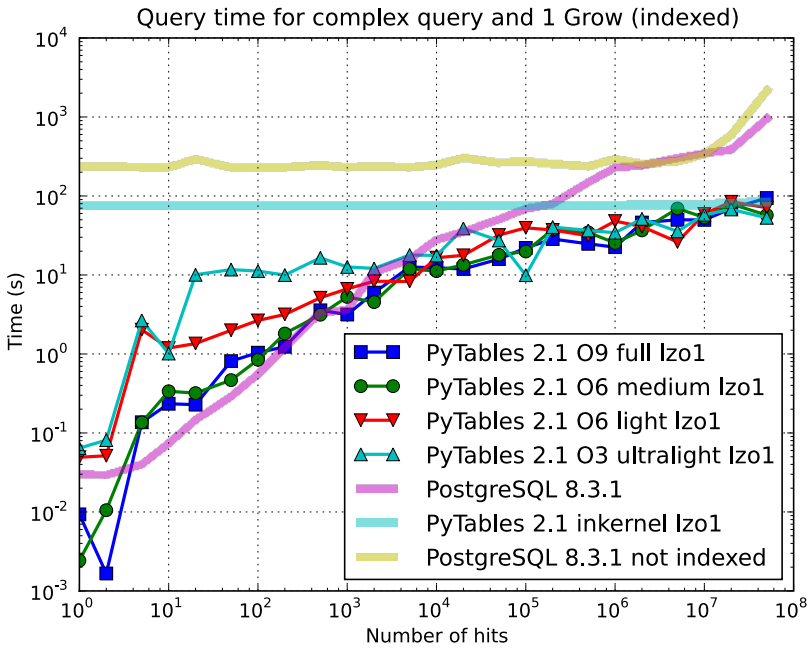


Figure 5.9. Times for complex queries with a cold cache (mean of 5 first random queries) for different optimization levels. Benchmark made on a machine with Intel Core2 (64-bit) @ 3 GHz processor with RAID-0 disk storage.

Of course, compression also has an effect when doing indexed queries, although not very noticeable, as can be seen in Figure 5.10. 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.

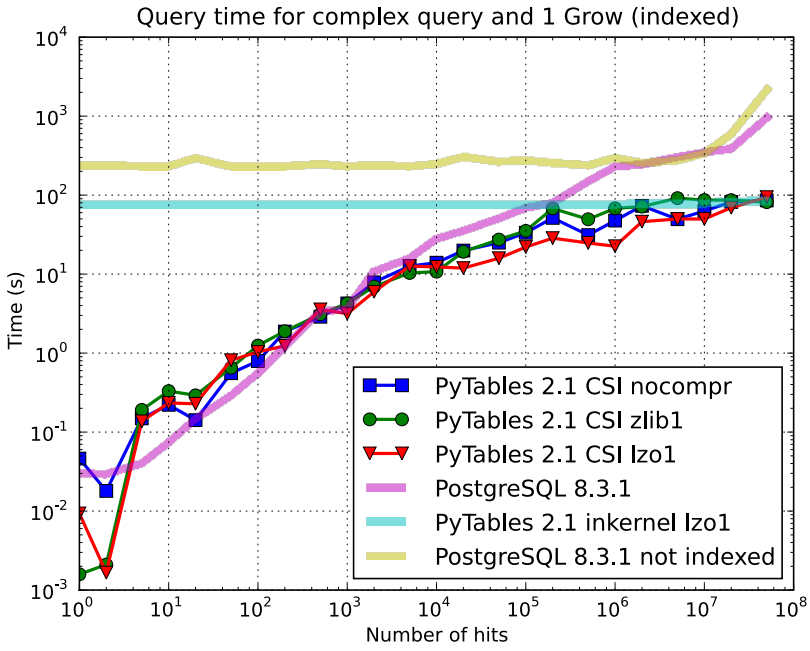


Figure 5.10. Times for complex queries with a cold cache
(mean of 5 first random queries) for different compressors.

You can find a more complete description and benchmarks about OPSI, the indexing system of PyTables Pro in [20].

## 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 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 5.11. 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.

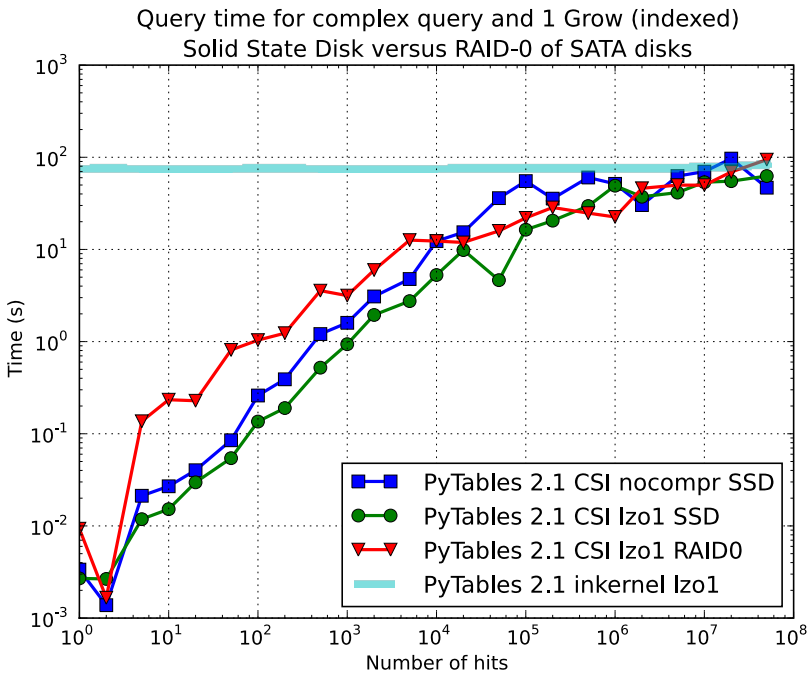


Figure 5.11. Times for complex queries with a cold cache (mean of 5 first random queries) for different disk storage (SSD vs spinning disks).

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 Pro and SSD you can query larger datasets that otherwise would require spinning disks when using other databases [2], 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)) |
```

[2] In fact, we were unable to run the PostgreSQL benchmark in this case because the space needed exceeded the capacity of our SSD.

```
      ((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 (sorting of tables is supported from PyTables Pro 2.1 on), and re-indexing `col2` and `col4` afterwards, we should get much faster performance for our query. This is effectively demonstrated in Figure 5.12, 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).

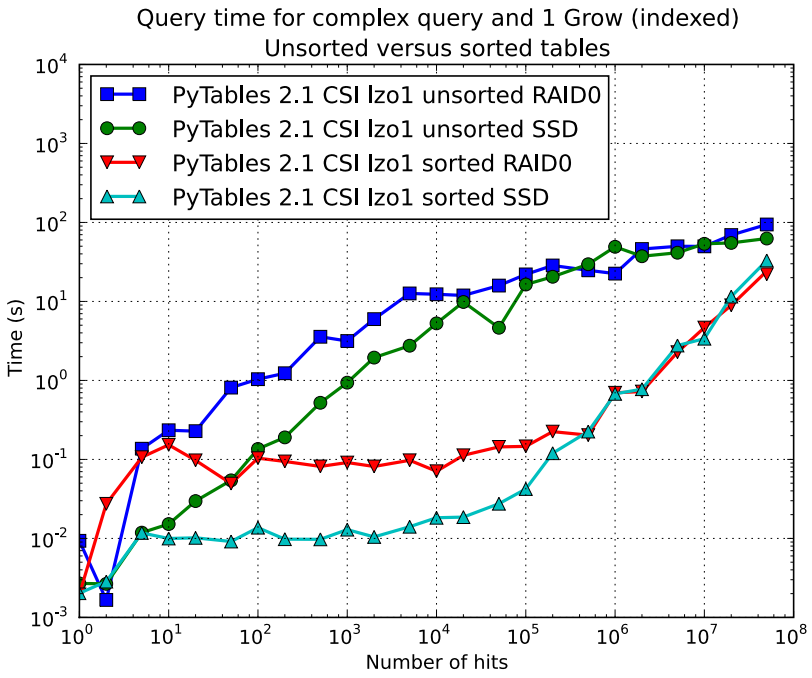


Figure 5.12. Times for complex queries with a cold cache
(mean of 5 first random queries) for unsorted and sorted tables.

Another thing worth to be noted in Figure 5.12, is that, for very large number of hits, and more exactly in the 500,000 hits point, the time to query in the case of a sorted table, can be more than 2x times faster, which may sound a bit suspicious provided that we already stated that the speed for the case of the unsorted table was only limited by the I/O speed of the disk, so how sorting a table can actually improve I/O times?. Well, the answer is that, as the table is sorted by a column, and due to the data distribution chosen, the entropy of the resulting table is reduced quite a lot, so allowing the compressor (LZO in this case) to do a much better job and producing datasets more than 2x smaller (and hence, requiring less time to be read). So, in the end, the there is no contradiction in having improved query times in the low selectivity scenario.

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 Section 5.1.2, for ways on improving this (or others) 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 your imagination (and most plausibly your time ;-).

# 5.3. Compression issues

One of the beauties of PyTables is that it supports compression on tables and arrays[3], 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 [12]). 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 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 [13]) and bzip2 (see [14]). 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 Section E.2 (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 5.13. As you can see, LZO is the compressor that performs worse in this sense, but, curiosly enough, there is not much difference between Zlib and bzip2.

[3] Except for `Array` objects.

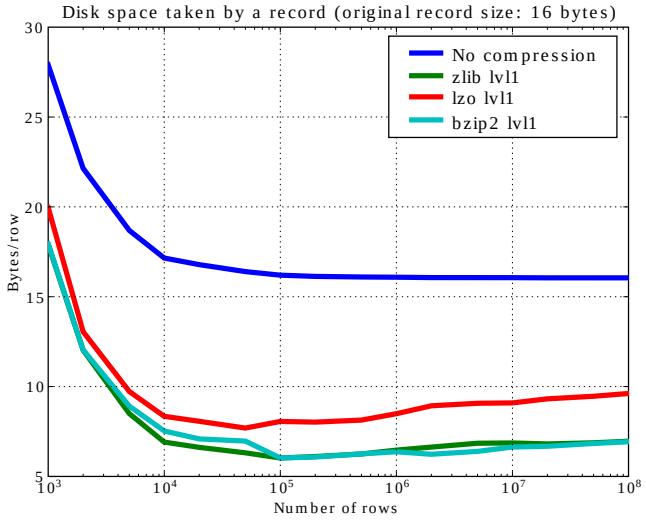


Figure 5.13. Comparison between different compression libraries.

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 5.14 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.

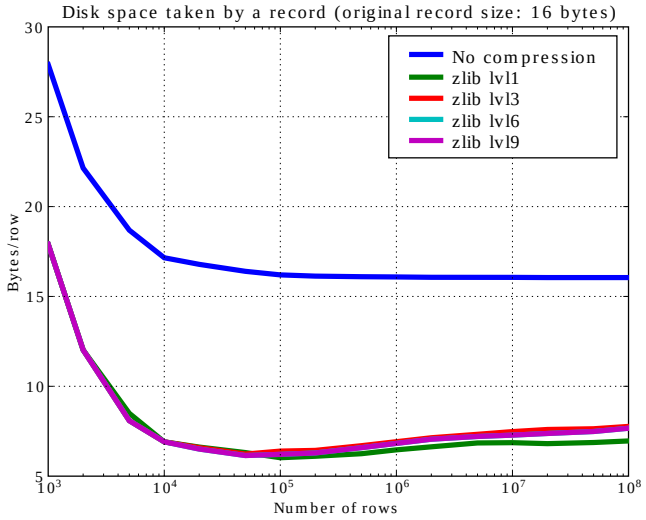


Figure 5.14. Comparison between different compression levels of Zlib.

Have also a look at Figure 5.15. 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.

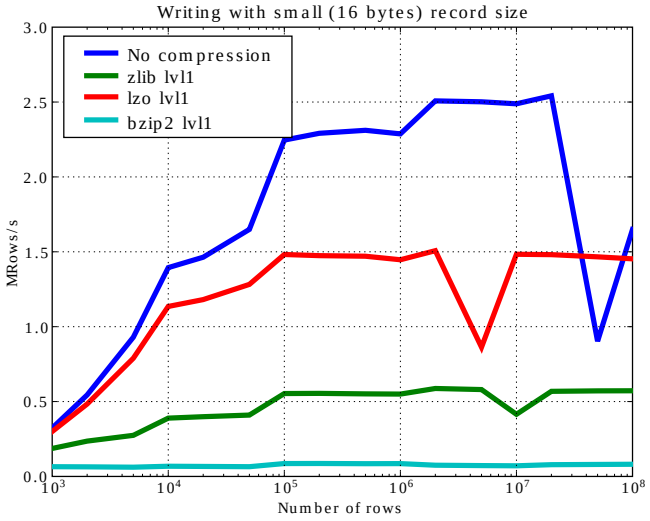


Figure 5.15. Writing tables with several compressors.

In Figure 5.16 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 Section 5.2.1), 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 5.16 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 5.17 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 derived from the aforementioned plot is that LZO decompression 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.

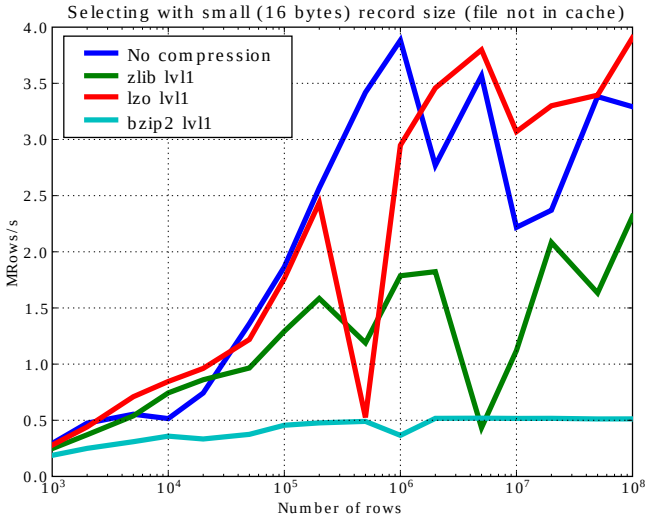


Figure 5.16. Selecting values in tables with several compressors. The file is not in the OS cache.

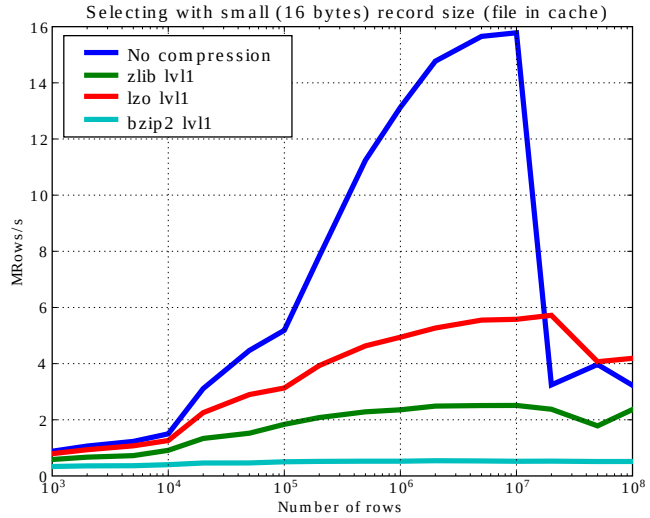


Figure 5.17. Selecting values in tables with several compressors. The file is in the OS cache.

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 Section 5.3.2). 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 5.18, 5.19, and the aforementioned 5.14 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 Section 4.14.1). 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 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).

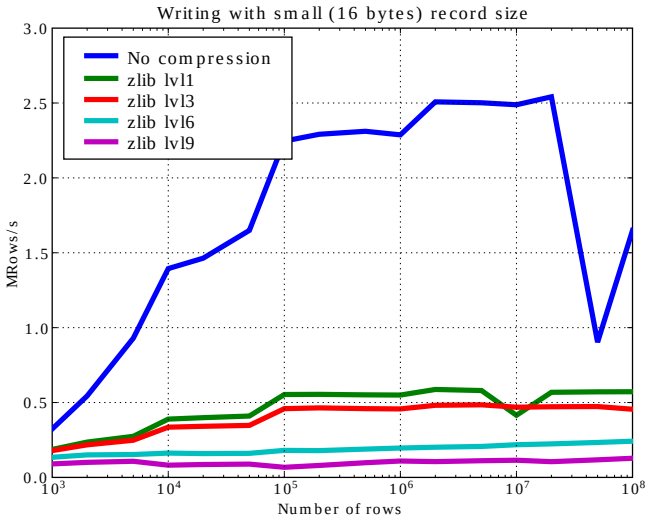


Figure 5.18. Writing in tables with different levels of compression.

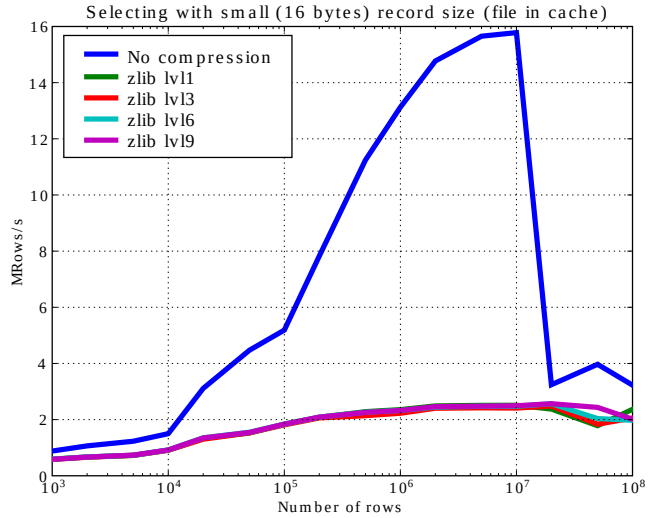


Figure 5.19. Selecting values in tables with different levels of compression. The file is in the OS cache.

## 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 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 5.20 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[4].

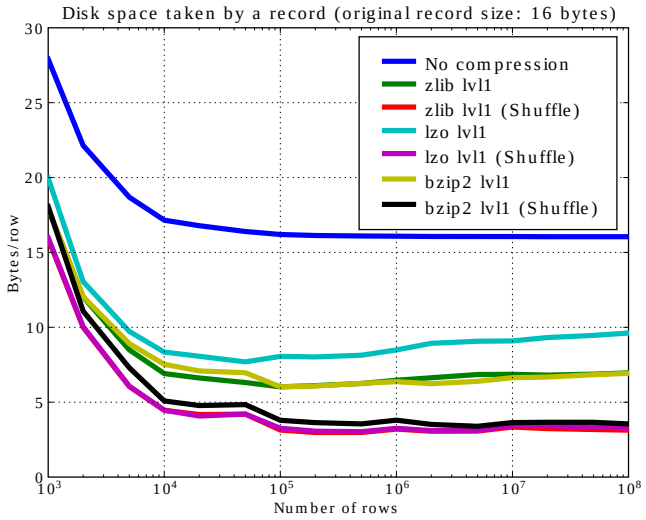


Figure 5.20. Comparison between different compression libraries with and without the *shuffle* filter.

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 5.21, 5.22 and 5.23, you will get a somewhat unexpected (but pleasant) surprise. Roughly, *shuffle* makes the writing process (shuffling+compressing) faster (aproximately a 15% for LZO, 30% for Bzip2 and a 80% for Zlib), which is an interesting result by itself. But

[4]Some users reported that the typical improvement with real data is between a factor 1.5x and 2.5x over the already compressed datasets.

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).

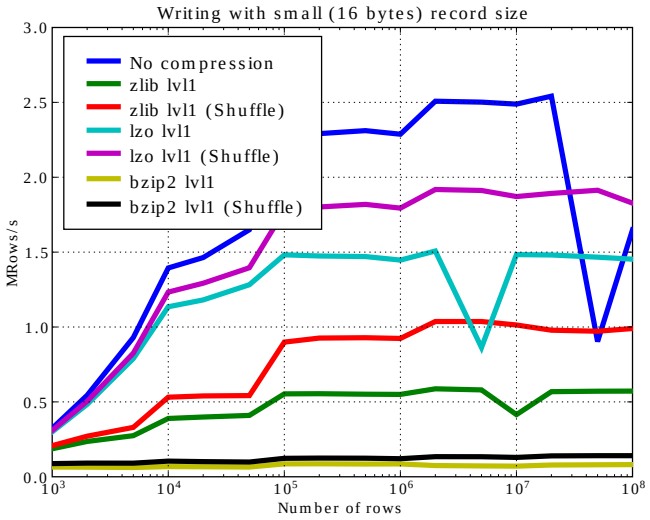


Figure 5.21. Writing with different compression libraries with and without the *shuffle* filter.

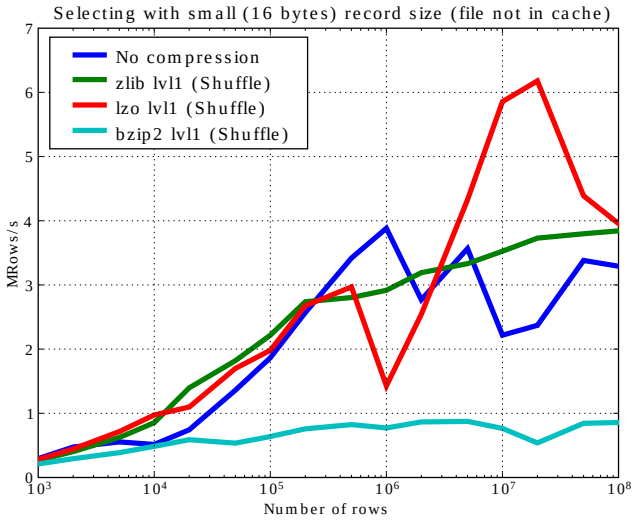


Figure 5.22. Reading with different compression libraries with the *shuffle* filter. The file is not in OS cache.

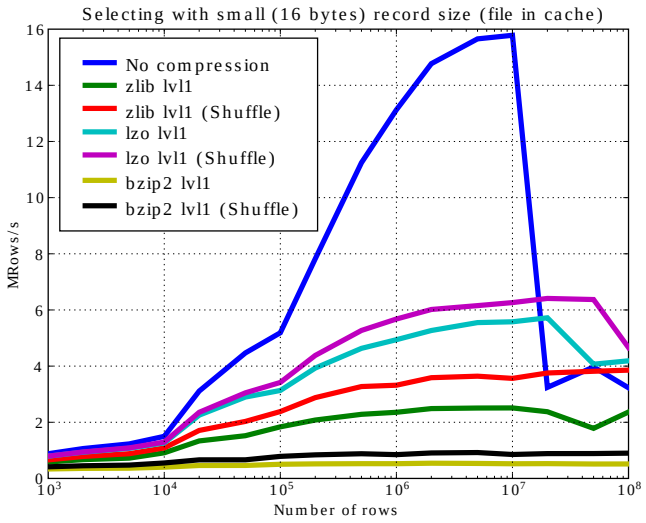


Figure 5.23. Reading with different compression libraries with and without the *shuffle* filter. The file is in OS cache.

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 [16]) 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.

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 Section 4.6.7, “The `Row` class”). But again, this latter case can also be accelerated by using the Table.append() (see description) method and building your own buffers.

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 5.24 and 5.25 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 Section 5.3). As always, there is no substitute for experimentation with your own dataset.

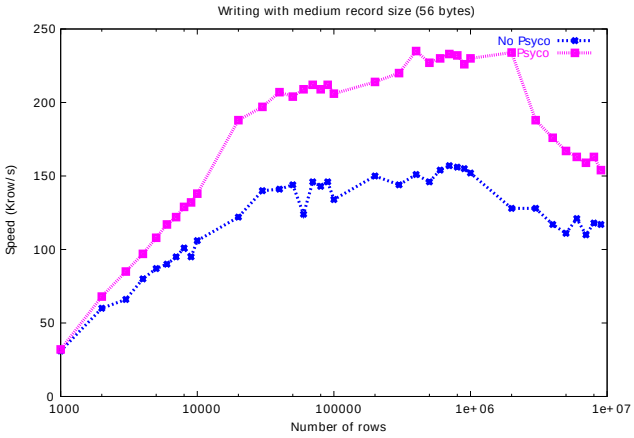


Figure 5.24. Writing tables with/without Psyco.

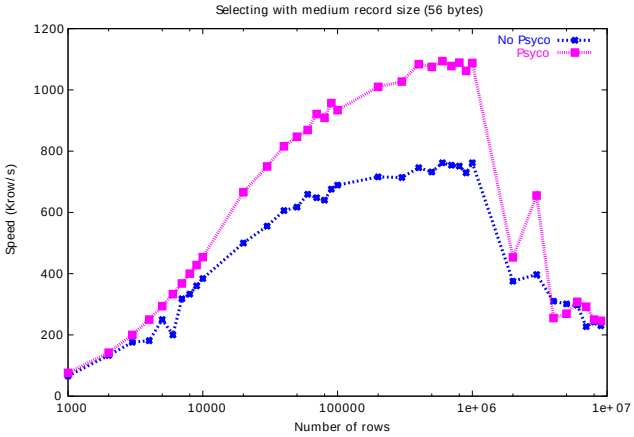


Figure 5.25. Reading tables with/without Psyco.

# 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, a new LRU cache was introduced that avoids loading all the nodes of the *object tree* in memory. This cache (one per file) is responsible of loading just up to a certain amount of nodes and discard the least recent used ones when there is a need to load new ones. This represents a big advantage over the old schema, specially 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). See [19] for more info on the advantages (and also drawbacks) of this approach.

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 5.1.

| | | 100 nodes | | | | 1000 nodes | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | 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 |

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

From the data in Table 5.1, 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 comsumption, 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 Pro 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 Section 5.1.1.

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 Section E.2 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

# Chapter 6. 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 [1]), 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 `getNodeAttr()` method (see description) 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 = ["FileNode", "test", "metadata"]
>>> fnode.attrs.keywords_ca = ["FileNode", "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 := ['FileNode', 'prova', 'metadades'],
keywords_en := ['FileNode', '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. There is no universal newline support yet. This is likely to be implemented in a near future.

3. 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

**NodeType** Value for `NODE_TYPE` node system attribute.

**NodeTypeVersions** Supported values for `NODE_TYPE_VERSION` node system attribute.

## 6.6.2. Global functions

### 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.

### 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

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

#### getLineSeparator()

Returns the line separator string.

#### setLineSeparator()

Sets the line separator string.

#### getAttrs()

Returns the attribute set of the file node.

#### close()

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

#### next()

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

#### read(size=None)

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

#### readline(size=-1)

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

#### readlines(sizehint=-1)

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

#### seek(offset, whence=0)

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

#### tell()

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

#### xreadlines()

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

## 6.6.4. The `ROFileNode` class

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

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.

### flush()

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

### 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.

### 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.

### writelines(sequence)

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

# Chapter 7. netcdf3 - a PyTables NetCDF3 emulation API

## 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 [6]). A very nice python interface to that library is available in the `Scientific Python NetCDF` module (see [17]). 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 [7]). 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.NetCDFFile.createDimension('level', 12)
>>> file.NetCDFFile.createDimension('time', None)
>>> file.NetCDFFile.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 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 `1` (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 long the *unlimited* dimension controls how may 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 numarray 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 Guilde' ;
}
```

### 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 Section 4.14.1) 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 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 Section 5.1.1 for details).

## 7.3. `tables.netcdf3` package reference

### 7.3.1. Global constants

**_fillvalue_dict** Dictionary whose keys are `NetCDFVariable` single character typecodes and whose values are the netCDF _FillValue for that typecode.

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

## 7.3.2. The `NetCDFFile` class

*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

#### close()

Closes the file (after invoking the `sync` method).

#### 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.

#### ncattrs()

Returns a list with the names of all currently defined netCDF global file attributes.

#### 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.

#### createVariable(name, type, dimensions, least_significant_digit= None, expectedsize=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 `1` (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 Section 5.1.1 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`.

#### 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`.

#### 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 type `s`) 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 [http://www.cdc.noaa.gov/cdc/conventions/cdc_netcdf_standard.shtml] 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

The `NetCDFVariable` constructor is not called explicitly, rather an `NetCDFVarible` 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

#### 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 `1` (Int8).

#### 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 may 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.

#### ncattrs()

Returns a list with all the names of the currently defined netCDF variable attributes.

#### assignValue(data)

Provided for compatiblity 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 `[:]`.

#### getValue()

Provided for compatiblity 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 Section E.3) 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 [http://www.cdc.noaa.gov/cdc/conventions/cdc_netcdf_standard.shtml] 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 0.00130000000354 9.99152469635
 6.68507480621
time, min/max temp, temp[n,0,0] = 2.0 0.0153000000864 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 0.01600000076 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 (http://opendap.oceanografia.org). 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 acess `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.cdc.noaa.gov/cdc/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

# Appendix A. Supported data types in PyTables

All PyTables datasets can handle the complete set of data types supported by the NumPy (see [8]), `numarray` (see [10]) and Numeric (see [9]) packages in Python. The data types for table fields can be set via instances of the `Col` class and its descendants (see Section 4.13.2), while the data type of array elements can be set through the use of the `Atom` class and its descendants (see Section 4.13.3).

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: 32 and 64 (default) bits.
- `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.

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 Section 4.14.3) 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 on enumerated values.

Here you have a quick reference to the complete set of supported data types:

| 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 |
| 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 | - | - |

Table A.1. Data types supported for array elements and tables columns in PyTables.

# Appendix B. Condition syntax

Conditions in PyTables are used in methods related with in-kernel and indexed searches such as `Table.where()` (see description) or `Table.readWhere()` (see description). They are interpreted using a customized version of Numexpr, a powerful package for C-speed computation of array operations (see [11]).

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 [1]. 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()` (see description) 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`).
- 64-bit, double-precision floating point number (`float`).
- 2x64-bit, double-precision complex number (`complex`).
- Raw string of bytes (`str`).

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: +, -, *, /, **, %.

[1] 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.

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)'` [2]

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 sinus, cosinus or tangent.
- `{arcsin,arccos,arctan}(float|complex): float|complex` — trigonometric inverse sinus, cosinus or tangent.
- `arctan2(float1, float2): float` — trigonometric inverse tangent of `float1/float2`.
- `{sinh,cosh,tanh}(float|complex): float|complex` — hyperbolic sinus, cosinus or tangent.
- `{arcsinh,arccosh,arctanh}(float|complex): float|complex` — hyperbolic inverse sinus, cosinus 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.
- `{real,imag}(complex): float` — real or imaginary part of complex.
- `complex(float, float): complex` — complex from real and imaginary parts.

[2] 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 `!`).

# Appendix C. 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 this, PyTables implements a couple of ways to change these values. All of the *tunable* parameters live in the `tables/parameters.py` (and `tables/_parameters_pro.py`, for PyTables Pro users). The user can choose to change them in the parameter files themselves for a global and persistent change. Moreover, if she wants a finer control, she can pass any of these parameters directly to the `openFile()` function (see description), 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. Please see your parameter files so as to know the 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!

## C.1. Tunable parameters in `tables/parameters.py`.

### C.1.1. Recommended maximum values

`MAX_COLUMNS`
: Maximum number of columns in `Table` objects before a `PerformanceWarning` is issued. This limit is somewhat arbitrary and can be increased.

`MAX_COLUMNS`

`MAX_NODE_ATTRS`
: Maximum allowed number of attributes in a node

`MAX_GROUP_WIDTH`
: Maximum depth in object tree allowed.

`MAX_UNDO_PATH_LENGTH`
: Maximum length of paths allowed in undo/redo operations.

### C.1.2. Cache limits

`METADATA_CACHE_SIZE`
: Size (in bytes) of the HDF5 metadata cache. This only takes effect if using HDF5 1.8.x series.

`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 comsumption, 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.

### C.1.3. Parameters for the I/O buffer in `Table` objects.

`CHUNKTIMES`
The buffersize/chunksize ratio.

`BUFFERTIMES`
The maximum buffersize/rowsize ratio before issuing a `PerformanceWarning`.

### C.1.4. Miscellaneous

`EXPECTED_ROWS_EARRAY`
Default expected number of rows for `EArray` objects.

`EXPECTED_ROWS_TABLE`
Default expected number of rows for `Table` objects.

`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).

## C.2. Tunable parameters in `tables/_parameters_pro.py`.


**Note**

These parameters are only available in PyTables Pro.

### C.2.1. Parameters for the different internal caches

`BOUNDS_MAX_SIZE`
The maximum size for bounds values cached during index lookups.

`BOUNDS_MAX_SLOTS`
The maximum number of slots for the `BOUNDS` cache.

`ITERSEQ_MAX_ELEMENTS`
The maximum number of iterator elements cached in data lookups.

`ITERSEQ_MAX_SIZE`
The maximum space that will take `ITERSEQ` cache (in bytes).

`ITERSEQ_MAX_SLOTS`
The maximum number of slots in `ITERSEQ` cache.

`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).

`LIMBOUNDS_MAX_SLOTS`
The maximum number of slots for `LIMBOUNDS` cache.

`TABLE_MAX_SIZE`
The maximum size for table chunks cached during index queries.

`SORTED_MAX_SIZE`
The maximum size for sorted values cached during index lookups.

`SORTEDLR_MAX_SIZE`
The maximum size for chunks in last row cached in index lookups (in bytes).

`SORTEDLR_MAX_SLOTS`
The maximum number of chunks for `SORTEDLR` cache.

## C.2.2. Parameters for general cache behaviour

### Warning

The next parameters will not be effective 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.

`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.

`ENABLE_EVERY_CYCLES`
The number of cycles in which a cache will be forced to be (re-)enabled, irregardingly of the hit ratio. This will provide a chance for checking if we are in a better scenario for doing caching again.

`LOWEST_HIT_RATIO`
The minimum acceptable hit ratio for a cache to avoid disabling (and freeing) it.

# Appendix D. Using nested record arrays

## D.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')]
```

# D.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 `'/'` [1], 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] 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']`.

```
[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.

# D.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 nra1[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.

# Appendix E. 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.

## E.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*).

### E.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.

### E.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 vlarray1.h5
vlarray1.h5 (File) ''
Last modif.: 'Mon Jan  8 16:21:25 2007'
Object Tree:
/ (RootGroup) ''
/vlarray1 (VLArray(3,), shuffle, zlib(1)) 'ragged array of ints'
/vlarray2 (VLArray(3,), shuffle, zlib(1)) 'ragged array of strings'
```

we can see that the file contains just a leaf object called `vlarray1`, that is an instance of `VLArray`, 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 vlarray1.h5
/ (RootGroup) ''
/vlarray1 (VLArray(3,), shuffle, zlib(1)) 'ragged array of ints'
  atom = Int32Atom(shape=(), dflt=0)
  byteorder = 'little'
  nrows = 3
  flavor = 'numeric'
/vlarray2 (VLArray(3,), shuffle, zlib(1)) 'ragged array of strings'
  atom = StringAtom(itemsize=2, shape=(), dflt='')
  byteorder = 'irrelevant'
  nrows = 3
  flavor = 'python'
```

so we can see more info about the atoms that are the components of the `vlarray1` dataset, i.e. they are scalars of type `Int32` and with `Numeric` *flavor*.

If we want information about the attributes on the nodes, we must add the `-a` flag:

```
$ ptdump -va vlarray1.h5
/ (RootGroup) ''
  /._v_attrs (AttributeSet), 5 attributes:
   [CLASS := 'GROUP',
    PYTABLES_FORMAT_VERSION := '2.0',
    TITLE := '',
    VERSION := '1.0']
/vlarray1 (VLArray(3,), shuffle, zlib(1)) 'ragged array of ints'
  atom = Int32Atom(shape=(), dflt=0)
  byteorder = 'little'
  nrows = 3
  flavor = 'numeric'
  /vlarray1._v_attrs (AttributeSet), 4 attributes:
   [CLASS := 'VLARRAY',
    FLAVOR := 'numeric',
    TITLE := 'ragged array of ints',
    VERSION := '1.2']
/vlarray2 (VLArray(3,), shuffle, zlib(1)) 'ragged array of strings'
  atom = StringAtom(itemsize=2, shape=(), dflt='')
  byteorder = 'irrelevant'
  nrows = 3
  flavor = 'python'
  /vlarray2._v_attrs (AttributeSet), 4 attributes:
   [CLASS := 'VLARRAY',
    FLAVOR := 'python',
    TITLE := 'ragged array of strings',
    VERSION := '1.2']
```

Let's have a look at the real data:

```
$ ptdump -d vlarray1.h5
/ (RootGroup) ''
/vlarray1 (VLArray(3,), shuffle, zlib(1)) 'ragged array of ints'
  Data dump:
[0] [5 6]
[1] [5 6 7]
[2] [5 6 9 8]
```

```
/vlarray2 (VLArray(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 `vlarray1` 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 `/vlarray1` object:

```
ptdump -R2,3 -d vlarray1.h5:/vlarray1
/vlarray1 (VLArray(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 `/vlarray1` object for doing the dump (`vlarray1.h5:/vlarray1`).

Finally, you can mix several information at once:

```
$ ptdump -R2,3 -vad vlarray1.h5:/vlarray1
/vlarray1 (VLArray(3,), shuffle, zlib(1)) 'ragged array of ints'
  atom = Int32Atom(shape=(), dflt=0)
  byteorder = 'little'
  nrows = 3
  flavor = 'numeric'
  /vlarray1._v_attrs (AttributeSet), 4 attributes:
   [CLASS := 'VLARRAY',
    FLAVOR := 'numeric',
    TITLE := 'ragged array of ints',
    VERSION := '1.2']
  Data dump:
[2] [5 6 9 8]
```

# E.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 may 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.

## E.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-copyuser-attrs] [--overwrite-nodes] [--complevel=(0-9)]
 [--complib=lib] [--shuffle=(0|1)] [--fletcher32=(0|1)] [--keep-source-
```

```
filters] [--chunkshape=value] [--upgrade-flavors] [--dont-regenerate-old-
indexes] [--sortby=column] [--forceCSI] [--propindexes] sourcefile:sourcegroup
 destfile:destgroup
   -h -- Print usage message.
   -v -- Show more information.
   -o -- Overwite 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-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" or "bzip2". 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 values of "column".
       This requires an existing 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.
   --forceCSI -- Force the creation of a CSI index in case one is not
       available for the --sortby column (this implies the modification of
       the *source* file).  The default is to not create it.
   --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.

## E.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
```

```
tutorial1.h5 (File) 'Test file'
Last modif.: 'Mon Jan  8 16:30:30 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'
```

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
reduced.h5 (File) ''
Last modif.: 'Mon Jan  8 16:31:31 2007'
Object Tree:
/ (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
reduced.h5 (File) ''
Last modif.: 'Mon Jan  8 16:32:25 2007'
Object Tree:
/ (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
reduced.h5 (File) ''
Last modif.: 'Mon Jan  8 16:33:08 2007'
Object Tree:
/ (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
reduced.h5 (File) ''
Last modif.: 'Mon Jan  8 16:33:59 2007'
Object Tree:
/ (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 ``rawdata``; you may want to use the ``overwrite`` argument
The destination file looks like:
reduced.h5 (File) ''
Last modif.: 'Mon Jan  8 16:33:59 2007'
Object Tree:
/ (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.
```

Ooops! 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
reduced.h5 (File) ''
Last modif.: 'Mon Jan  8 16:36:54 2007'
Object Tree:
/ (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
reduced.h5 (File) ''
Last modif.: 'Mon Jan  8 16:38:13 2007'
Object Tree:
/ (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 inexistent `slices` group has been created in the same operation.

# E.3. nctoh5

This tool is able to convert a file in `NetCDF` [http://www.unidata.ucar.edu/packages/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 [17]) 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 [18]) 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 conversor, do not forget to contribute it back to the community. Thanks!

## E.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)] netcdffilename hdf5filename
   -h -- Print usage message.
   -v -- Show more information.
   -o -- Overwite 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", "lzo" or "ucl". 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).
```

# Appendix F. PyTables File Format

PyTables has a powerful capability to deal with native HDF5 files created with another tools. However, there are situations were you may want to create truly native PyTables files with those tools while retaining fully compatibility with 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 *szip* (although the last one can not be used from within PyTables to create a new file, datasets compressed with szip can be read, because it is the HDF5 library which do the decompression transparently).

## F.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'`.

## F.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'`.

## F.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 Section 4.14.1) 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.

# F.4. Mandatory attributes, storage layout and supported data types for Leaves

This depends on the kind of `Leaf`. The format for each type follows.

## F.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 suported 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.

## F.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 Section F.4.1 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 Section F.4.1 for more info about this special type.

You should note that H5T_ARRAY class datatypes are not allowed in `Array` objects.

## F.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 Section F.4.1 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 Section F.4.1 for more info about this special type.

You should note that H5T_ARRAY class datatypes are not allowed yet in `Array` objects.

## F.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 Section F.4.1 for more info about these types. They can also be a H5T_COMPOUND datatype representing a complex number, see the `Table` format description in Section F.4.1.

You should note that H5T_ARRAY class data types are not allowed in `EArray` objects.

## F.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 Section F.4.5) 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 `VLArray` has a *dataspace* with a *1-dimensional chunked* layout.

### Data types supported

The data type of the elements (rows) of `VLArray` 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 Section F.4.1. 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 Section F.4.1 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.

# F.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 (record arrays, arrays, records, scalars) will be returned as `NumPy` objects.

**"numarray"**
Read data will be returned as `numarray` objects.

**"numeric"**
Read data will be returned as `Numeric` objects.

**"python"**
Read data will be returned as Python lists, tuples or scalars.

# Bibliography


[1] The HDF Group. *What is HDF5?*. Concise description about HDF5 capabilities and its differences from earlier versions (HDF4). http://hdfgroup.org/whatishdf5.html .

[2] The HDF Group. *Introduction to HDF5*. Introduction to the HDF5 data model and programming model. http://hdfgroup.org/HDF5/doc/H5.intro.html .

[3] The HDF Group. *The HDF5 table programming model*. Examples on using HDF5 tables with the C API. http://hdfgroup.org/HDF5/Tutor/h5table.html .

[4] 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-106.ibm.com/developerworks/xml/library/xml-matters2/index.html .

[5] Greg Ewing. *Pyrex. A Language for Writing Python Extension Modules*. http://www.cosc.canterbury.ac.nz/~greg/python/Pyrex .

[6] 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/packages/netcdf/ .

[7] 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/netcdf-4/ .

[8] 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://numeric.scipy.org/ .

[9] David Ascher, Paul F Dubois, Konrad Hinsen, Jim Hugunin, and Travis Oliphant. *Numerical Python*. Package to speed-up arithmetic operations on arrays of numbers. http://sourceforge.net/projects/numpy/ .

[10] 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/ .

[11] David Cooke and Timothy Hochberg. *Numexpr. Fast evaluation of array expressions by using a vector-based virtual machine*. It is an enhaced 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 .

[12] 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/ .

[13] Markus F Oberhumer. *LZO. A data compression library which is suitable for data de-/compression in real-time*. It offers pretty fast compression and extremly fast decompression with reasonable compression ratio. http://www.oberhumer.com/opensource/ .

[14] Julian Seward. *bzip2. A high performance lossless compressor*. It offers very high compression ratios within reasonable times. http://www.bzip.org/ .

[15] 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/ .

[16] Armin Rigo. *Psyco. A Python specializing compiler*. Run existing Python software faster, with no change in your source. http://psyco.sourceforge.net .

[17] Konrad Hinsen. *Scientific Python*. Collection of Python modules useful for scientific computing. http://starship.python.net/~hinsen/ScientificPython/ .

[18] 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 .

[19] 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 .

[20] 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 .

[21] 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://www.vitables.org .